Parallel Computing 86 (2019) 66–81

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Performance of asynchronous optimized Schwarz with one-sided communication

Ichitaro Yamazaki[a,*], Edmond Chow[b], Aurelien Bouteiller[a], Jack Dongarra[a]

[a] University of Tennessee, Knoxville, TN, USA
[b] Georgia Institute of Technology, Atlanta, Georgia, USA

ARTICLE INFO

Article history:
Received 26 June 2018
Revised 19 February 2019
Accepted 14 May 2019
Available online 15 May 2019

ABSTRACT

In asynchronous iterative methods on distributed-memory computers, processes update their local solutions using data from other processes without an implicit or explicit global synchronization that corresponds to advancing the global iteration counter. In this work, we test the asynchronous optimized Schwarz domain-decomposition iterative method using various one-sided (remote direct memory access) communication schemes with passive target completion. The results show that when one-sided communication is well-supported, the asynchronous version of optimized Schwarz can outperform the synchronous version even for perfectly balanced partitionings of the problem on a supercomputer with uniform nodes.

1. Introduction

Global synchronization in a bulk-synchronous distributed-memory parallel program can be explicit (e.g., use of barriers or collective communications) or implicit (e.g., all processes exchange data with their "neighbors"). All the participating processes must wait for the slowest process at the synchronization point. Hence, synchronization exposes any load imbalance or non-uniform hardware performance (e.g., non-uniform cost of communication among the processes, system noise [1]). Such imbalances tend to grow as more processes are used on larger distributed-memory computers. Consequently, the time needed for the explicit or implicit global synchronizations can become a large portion of the execution time of a distributed-memory code. The US Department of Energy has identified the effect of synchronization as one of the primary performance-limiting factors on anticipated exascale supercomputers [2,3].

A common solution to address synchronization issues has been to move away from bulk-synchronous programming to task-based parallel programming. However, task-based programming does not fit many algorithms of interest, such as iterative methods, which are mathematically defined to operate on data in lock-step fashion. To break away from lock-step operation, asynchronous iterative methods have been proposed. In these mathematically-different methods, processes utilize whatever data is available from other processes without waiting to synchronize at every iteration. Hence, in an asynchronous iterative method, before new data arrives from slower processes, faster processes perform extra iterations with old data to improve their local solutions. The slower processes then iterate with more accurate data from the faster processes and may converge with fewer local iterations. Overall, the asynchronous method may reach the global solution in less time than the synchronous method.

Previous research on asynchronous solvers (such as [4,5]) focused on using two-sided communication supported by the Message Passing Interface (MPI). To send the data to a remote process using two-sided communication, the process must coordinate the data transfer with the remote process. We feel it is more natural for asynchronous solvers to use one-sided remote direct memory access (RDMA) operations. In these operations, the values in the remote memory can be read or written without the interaction of the remote process. Thus, unlike using two-sided MPI, to complete the communication requested by the origin process, the remote process does not need to be involved (e.g., enter the MPI library). Communication latency may be reduced. Moreover, for messages longer than the eager limit, the two-sided communication may not overlap with computation. However, although modern implementations of MPI provide access to the RDMA capabilities of the underlying network hardware, these capabilities are not always well-supported by different versions of MPI for different hardware. In this paper, we study various implementations of asynchronous communication.

* Corresponding author. The author has recently moved to Sandia National Laboratories, New Mexico, U.S.A.
E-mail address: iyamaza@sandia.gov (I. Yamazaki).

https://doi.org/10.1016/j.parco.2019.05.004
0167-8191/

For comparison, we also test asynchronous communication using Symmetric Hierarchical MEMory (SHMEM), which provides similar functionality as MPI one-sided communication. In contrast to MPI, however, SHMEM follows the Partitioned Global Address Space (PGAS) programming model, which assumes a global memory address space that is logically partitioned across processes. In SHMEM, therefore, a process can access remote data through the "symmetric" memory that has the same size and address space on all the processes. SHMEM was previously used to implement Jacobi's method [4].

Our study is based on the asynchronous version of a fast-converging domain decomposition solver called optimized Schwarz [6–8]. Compared with classical Schwarz (whose convergence rate is similar to those of other fixed-point iterative methods like block Jacobi or block Gauss-Seidel), optimized Schwarz can converge significantly faster when its parameters are properly chosen. The results in this paper, however, are applicable to other iterative methods.

Previous work on asynchronous Schwarz methods, in addition to using two-sided MPI as mentioned above, only tested the methods using irregular meshes partitioned across distributed-memory computers with non-uniform nodes, i.e., the nodes can have different CPU types [5,9]. In this scenario, it is easy to show that asynchronous methods will outperform synchronous versions of the method. In our study, we study the performance of the asynchronous method for problems on 2-D regular meshes that are evenly distributed among the processes running on theoretically identical nodes (that are relevant to current and future supercomputers, including exascale supercomputers). We find that the asynchronous method converges, and when asynchronous communication is well-supported, it can outperform the synchronous method even for balanced partitioning due to performance non-uniformities.

An additional issue with asynchronous iterative methods is how to detect convergence of the iterations. In synchronous methods, a global residual norm computation is performed, usually at every iteration. This corresponds to an explicit global synchronization. In asynchronous methods, no global iteration count is available, so a global residual norm while iterations are progressing is difficult to define. Not being the focus of this paper, we use a simple convergence detection technique for our experimental tests.

The rest of the paper is organized as follows. After listing related work in Section 2, we outline, in Section 3, the algorithms studied in this paper. We then describe our implementations of the algorithms in Section 4. Finally, we list our experimental setups in Section 5, and benchmark the communication subroutines and present the solver performance in Sections 6 and 7, respectively. Final remarks are made in Section 8.

2. Related work

Previous work studied asynchronous variants of the classical and optimized Schwarz methods on distributed-memory computers; see [5,9–12], respectively. Previous work also proved convergence of the asynchronous optimized Schwarz method in certain conditions (there is no convergence proof for the case of 2-D grid partitioning used in this paper), and demonstrated the potential of the method over standard (synchronous) optimized Schwarz when using unbalanced partitions on nodes with different CPU types [5]. In contrast, we are interested in asynchronous performance in the case of balanced partitions on a current supercomputer with identical nodes and a high-bandwidth interconnect. Further, we use true asynchronous remote memory access rather than attempt to simulate asynchronous remote memory access with non-blocking two-sided communication. In recent years, one-sided MPI was used for other types of iterative solvers such as Jacobi or Southwell [13–16], including in the asynchronous case. While previous work focused on showing the potential of the asynchronous iteration to improve performance, our focus, in addition to focusing on balanced partitions and a more powerful domain decomposition iterative solver, is to study the effects of various asynchronous communication schemes on solver performance. In [4], SHMEM was also tested for the asynchronous Jacobi method, while only two-sided communication was utilized for MPI. In [17], a few different ways of using two-sided and one-sided MPI to implement the asynchronous communication for a general iterative algorithm were proposed, but without performance results. We extend this study by testing several options (e.g., different ways of flushing the local data in Section 4.2) and provide benchmark results (in Section 6).

A number of asynchronous termination detection algorithms have been proposed for asynchronous solvers to detect global convergence without synchronization [18,19]. In our implementation, global convergence is reached when all the processes detect that their local solutions satisfy a local convergence criteria (see Section 4.3). While waiting for notification of global convergence, each process continues to iterate, potentially updating its local solution using new data from neighboring processes. The challenge of detecting global convergence is that, although local convergence may have been achieved using old data from a neighboring process, the process may need to perform additional iterations to recover local convergence once new data arrives from the neighboring processes. The previously-proposed algorithms [18,19] integrate several mechanisms to ensure that all the processes achieve local convergence at the same time. In this paper, to detect global convergence, we rely on a simple algorithm that is based on the binary-tree arrangement of the processes (see Section 4.3).

High-level communication libraries have been developed for asynchronous solvers [20,21]. These, however, use non-blocking two-sided communication primitives and do not use truly asynchronous communication. (These libraries are also not currently publicly available.) In this work, we use the one-sided communication routines of MPI, which are readily available on distributed-memory computers and whose vendor-optimized implementations are often available on many supercomputers. In the context of MPICH, a plugin called Casper has been developed to help ensure asynchronous progress of one-sided "accumulate" communication using a progress process [22] (in contrast to using a progress thread). There is also an MPI-3 based PGAS runtime system, called DART [23], which uses the progress process to ensure the asynchronous progress of the communication. In [24,25], the programmability of different programming environments for asynchronous iterative algorithms was discussed in the context of multithreaded and grid computing.

Although we experimentally show the effect of the parameters used in optimized Schwarz on solver performance, there have been analytical studies for selecting the optimal values for these parameters on several model problems [5,7].

3. Optimized Schwarz method

The optimized Schwarz method solves a boundary value problem of the form

$$\begin{aligned} \mathcal{L}(\mathbf{x}) &= \mathbf{b} \text{ in } \Omega \\ \mathcal{C}(\mathbf{x}) &= \mathbf{c} \text{ on } \partial\Omega, \end{aligned} \tag{1}$$

where $\mathcal{L}$ and $\mathcal{C}$ are partial differential operators defined on the domain Ω and its boundary $\partial\Omega$, respectively. In the Schwarz method, the domain Ω is partitioned into multiple subdomains Ω_p either with or without overlap (for $p = 1, 2, \ldots, n_p$ where n_p is the number of subdomains and equivalently the number of processes).

Then, at each iteration, the approximation to the solution $\mathbf{x}$ is updated by solving the restriction of the problem onto each subdomain, using a certain interface condition between subdomains. In the classical Schwarz method, a Dirichlet condition is used on the interface between the subdomains. The independent subdomain problems can be solved in parallel.

To improve the convergence of the classical Schwarz, optimized Schwarz uses a Robin boundary condition. In this paper, we use the OO0 condition

$$\frac{\partial \mathbf{x}^{(a)}}{\partial n} + \alpha \mathbf{x}^{(a)} = \frac{\partial \mathbf{x}^{(b)}}{\partial n} + \alpha \mathbf{x}^{(b)} \text{ on } \partial\Omega_p \setminus \partial\Omega, \tag{2}$$

on the interface involving subdomains a and b, where n is a direction normal to the interface, and where α is a tuning parameter chosen to optimize the convergence of the method. The optimized Schwarz method has been successfully applied to many models and applications, including Maxwell equations [26,27], Helmholtz equations [28–30], fluid dynamics [31], convection-diffusion [32], and reaction-diffusion problems [33].

4. Implementation of optimized Schwarz

For our experiments, we focus on solving the 2-D Poisson equation with Dirichlet boundary conditions on a square domain. We use a 5-point finite difference discretization, leading to the linear algebraic equations $L\mathbf{x} = \mathbf{b}$. On a distributed-memory computer, we assume that the processes are organized on a q-by-q processor grid such that the pth process is the (i, j)th process on the grid with $i := \lfloor p/q \rfloor$ and $j := p \bmod q$. The global finite difference mesh is partitioned into non-overlapping subdomains, one subdomain per process. Let m-by-m denote the size of the local mesh for each of these subdomains. Fig. 1 shows an example partitioning with $q = 3$ and $m = 2$.

To form a set of overlapping subdomains, each non-overlapping subdomain is extended in each direction (except at the boundaries of the global domain) by γ grid points. Fig. 1 highlights one of these overlapping subdomains. In the figure, the interface of this subdomain corresponds to the blue points, which are called *local interface* points. The non-local grid points that are directly connected to the local interface points are called *external interface* points, shown in red. The points corresponding to the non-overlapping subdomain, shown in green, are called *interior points*.

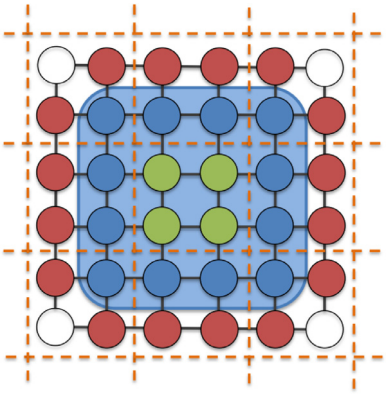


Fig. 1. Example finite difference mesh. Orange dashed lines show the partitioning of the 2-D mesh with $q = 3$ and $m = 2$. One non-overlapping subdomain is shown in green. Its corresponding overlapping subdomain (with $\gamma = 1$) is the union of the green and the blue points. For this subdomain, its interface corresponds to the blue points, which are called local interface points. The red points connected to the blue points are called external interface points. The green points are called interior points. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

The solution and right-hand-side vectors are distributed according to this domain decomposition scheme.

The local matrix corresponding to the overlapping subdomain in this example has size 16-by-16. The matrix is ordered using a natural, row-major numbering of the grid points Each process has a local right-hand-side vector (both the non-overlapped and overlapped portions), and is responsible for computing the non-overlapped part of the global solution vector.

For the Robin interface condition (2), we approximate the derivative with a one-sided finite difference formula. Thus, our pth local submatrix A_p is given by

$$A_p = L_p - D_p,$$

where L_p is the sparse Laplacian matrix and D_p is the diagonal matrix that arises from the interface condition. For our example, assuming a unit mesh spacing, D_p is the 16-by-16 diagonal matrix

$$D_p = \text{diag}(2\beta, \beta, \beta, 2\beta, \quad \beta, 0, 0, \beta, \quad \beta, 0, 0, \beta, \quad 2\beta, \beta, \beta, 2\beta)$$

(spacing is used to help identify how the elements of this matrix are mapped to mesh points) where $\beta = \frac{1}{1+\alpha}$. Thus, when $\alpha = \infty$ (or equivalently $\beta = 0$), the method is equivalent to classical Schwarz (with Dirichlet interface conditions), i.e., $A_p = L_p$.

For our implementation used in this paper, we store the local submatrix A_p in compressed sparse row (CSR) format. We use a sparse direct method for solving the local problems, in particular, the multifrontal symmetric indefinite linear solver `MA57` from the HSL mathematical software library [34].

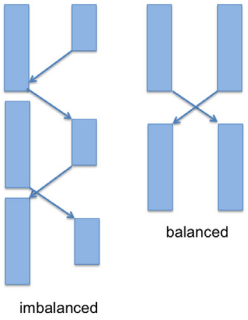
To update the local interface elements of the right-hand-side vector, each process sends elements of the current solution vector to its neighboring process (which correspond to the local and external interface elements of the neighboring process). Then, the neighboring process computes the derivatives normal to the interface. It is also possible for each process to compute and send the derivatives to neighboring processes. This halves the communication volume. However, as we describe in Section 4.3, with the minimum overlap ($\gamma = 1$), these local interface elements are also used by the neighboring process for the local convergence test (to compute the local residual norm). Thus, in our current implementation, we let the neighboring process compute the derivatives.

In the following two sections, we describe our design of the synchronous and asynchronous point-to-point communication of the interface elements (Sections 4.1 and 4.2). We also describe our implementation of the asynchronous termination detection algorithm for checking for global convergence without synchronization (Section 4.3).

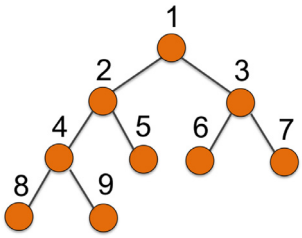
4.1. Synchronous communication

For synchronous iterations, we implement the point-to-point neighborhood communication using MPI two-sided communication subroutines. More specifically, each process first packs the interior elements to be sent to each neighboring process into a communication buffer and calls `MPI_Isend`. Similarly, to receive the interface's elements, each process calls `MPI_Irecv` for each of the neighboring processes, and then unpacks the elements into the local vector after calling `MPI_Wait` for `MPI_Irecv`. Finally, each process calls `MPI_Wait` to wait for the completion of `MPI_Isend`. This leads to synchronizations among neighboring processes.

When load imbalance exists between a pair of neighboring processes, it may be possible to overlap the point-to-point communication behind the local computation. Namely, we can delay synchronizing `MPI_Isend` until we start packing the interface elements into the communication buffer at the next iteration. Similarly, after unpacking the interface elements into the local vector, we can call `MPI_Irecv` to prepare to receive the message from



(a) Overlaping two-sided communication.



(b) Binary tree arrangement of processes.

Fig. 2. Implementation illustrations. In (a), boxes represent local computation, while arrows show point-to-point communication of the interface elements. In (b), processes are arranged in a binary tree for asynchronous termination detection.

the next iteration (we used `MPI_Wait`, instead of `MPI_Waitall`, to overlap the communication with other local tasks such as unpacking or packing the message). Unfortunately, when the load is balanced (e.g., between a pair of the slowest processes), this does not overlap the communication because the processes need to wait for the `MPI_Recv` message before starting the next iteration. Fig. 2(a) illustrates the communication with imbalanced and balanced load.

4.2. Asynchronous communication

MPI-2 introduced one-sided communication between "origin" and "target" processes (e.g., `MPI_Put` and `MPI_Get`). In contrast to two-sided communication, one-sided communication allows an origin process to access the target's remote memory without synchronizing with the target process, making it a more natural communication scheme for asynchronous iterative methods. Each process can designate part of its local memory as a remotely-accessible memory called a "window object" created through a collective MPI call among the participating processes (e.g., `MPI_Win_create`). We can then open a time frame, called an "epoch," within which the participating processes can access the window. For instance, in the "passive" data synchronization mode, an origin process can open an epoch with a specific target process using the target's MPI rank. We used this passive data access mode to implement our asynchronous communication, which is all point-to-point. There are two other access modes called "active," and "generalized active," that open the epoch with all the participating processes or with the subset of the processes using an MPI sub-communicator, respectively.

For our asynchronous implementation, like our synchronous implementation, interior elements are first packed into a communication buffer before calling `MPI_Put` to push all these elements into the remotely-accessible memory of the neighboring processes at once. A separate window object is created for each neighboring process to potentially perform the communication with different processes in parallel. After each call to `MPI_Put`, we call either `MPI_Win_flush` or `MPI_Win_flush_local` such that the origin process waits for the completion of the data transfer at the target or origin process, respectively (the target process is not involved). To compute the derivatives, each process unpacks the locally-available interface elements in its remotely-accessible memory into the local solution vector at each iteration. Compared to two-sided communication, the execution of `MPI_Put` does not involve the target process. This not only avoids synchronization among the neighboring processes, but also allows communication to be overlapped with the local computation or pipelined with other communication. To overlap the communication, we place `flush` or `flush_local` to flush the data in the message buffer for the previous `put` just before using the buffer for the next communication.

As a comparison, we also used SHMEM to implement asynchronous communication in a similar manner. In particular, since we focus on the balanced problem, we allocate a fixed-size communication buffer in the symmetric memory on each process for receiving the interface data. At each step, each process then packs the interface elements into the local buffer and uses `shmem_double_put` to send the local buffer's data into the target process's remote buffer; `shmem_double_put` returns when the data is copied out of the local buffer, and hence, when it returns, the data may not be delivered at the target yet. Finally, we call `shmem_quiet` to locally wait for the completion of the `put` operations issued by the origin process and to ensure ordering of the remote write operations.

4.3. Termination detection

Our solver terminates the asynchronous iterations when the local convergence criterion, $\|\bar{\mathbf{r}}_p\|_2^2 < \tau^2 \|\bar{\mathbf{b}}_p\|_2^2$, on all the processes is satisfied. Here, $\bar{\mathbf{r}}_p$ and $\bar{\mathbf{b}}_p$ are the residual and right-hand-side vectors on the interior points of the pth subdomain, and τ is a user-specified threshold. When the local convergence criterion is satisfied on all the processes, then $\|\mathbf{r}\|_2 < \tau \|\mathbf{b}\|_2$, i.e., the global convergence criterion is also satisfied.

To compute the local residual norm on the interior points at each step, each process uses the current interior elements of the solution vector (which are computed by this process), and the non-local elements directly connected to the interior points (which are received from the neighboring process). The interior submatrix with the edges to the neighboring subdomains is stored in CSR format for the process to locally apply the sparse matrix-vector multiply and compute the residual vector. Our detection is based on the minimum residual norm in order to avoid the process coming out of the local convergence state.

Our asynchronous termination detection procedure uses MPI one-sided communication. We implemented an asynchronous all-reduce of the local convergence notifications and an asynchronous broadcast of the global termination notification based on a binary-tree arrangement of the processes (see Fig. 2(b)). At each iteration, each process, except the root process, checks whether the local convergence criterion has been satisfied and if the process has received convergence notifications from both of its child processes. If these two conditions are met, the process uses `MPI_Put` to send an integer flag indicating convergence to its parent process. Once the root process receives the notification from its child processes and satisfies its own local convergence criterion, it then sends its children the global termination notification, which is sent down along the binary tree. We note that processes continue iterating only until they receive the global termination notification. Fig. 3 shows our implementation of the asynchronous termination detection procedure using MPI one-sided communication subroutines. Our SHMEM implementation of the termination detection procedure is identical to the MPI implementation, except that it uses `shmem_int_put` instead of `MPI_Put`.

After the local convergence criterion is satisfied, as a process receives new interface elements, its local residual norm could increase such that the local convergence criterion is no longer satisfied. Nevertheless, we found that this algorithm is sufficient in our experiments, i.e., by the time the process receives the global termination notification, its residual norm generally satisfies the local convergence criterion. After the asynchronous iterations are terminated on all the processes, the global residual norm was explicitly

```
if (((converged[0] == 1 && converged[1] == 1) // both children converged
     || (converged[0] == 1 && ip == np/2-1)   // only one child
     || (ip >= np/2 && converged[0] != 2))    // leaf
     && *converged_all_local > 0)             // locally converged
{
    if (ip == 0) {
        // on the top, start going down
        converged[2] = 1;
    } else {
        // push to parent
        int p  = (ip-1)/2;
        int id = (ip%2 == 0 ? 1 : 0);
        MPI_Put(&ione, 1, MPI_INT, p, id, 1, MPI_INT, window_converged);
        if (option == 1) {
            MPI_Win_flush(p, window_converged);
        } else if (option == 2) {
            MPI_Win_flush_local(p, window_converged);
        }
    }
    converged[0] = 2; // to push up only once
}
```

(a) Push local convergence notification up the tree.

```
if (converged[2] == 1) {
    int p = 2*ip+1;
    if (p < np) { // to left child
        MPI_Put(&ione, 1, MPI_INT, p, 2, 1, MPI_INT, window_converged);
        if (option == 1) {
            MPI_Win_flush(p, window_converged);
        } else if (option == 2) {
            MPI_Win_flush_local(p, window_converged);
        }
    }
    p ++;
    if (p < np) { // to right child
        MPI_Put(&ione, 1, MPI_INT, p, 2, 1, MPI_INT, window_converged);
        if (option == 1) {
            MPI_Win_flush(p, window_converged);
        } else if (option == 2) {
            MPI_Win_flush_local(p, window_converged);
        }
    }
    converged[1] ++;
}
```

(b) Push global convergence notification down the tree.

Fig. 3. The binary-tree based asynchronous termination detection ('`ip`' and '`np`' are the process id and the number of processes, respectively). When the first or second entry of the array `converged` is one, it indicates that the left or right child has locally converged, while the third entry of the array indicates that global convergence is achieved.

computed and was of the same order of magnitude as that specified by the global convergence criterion.

We could implement a synchronous termination detection algorithm in the same fashion. However, we used `MPI_Allreduce` on an integer flag of which the value of one or zero indicates local convergence or nonconvergence, respectively. Therefore, when the resulting value of the all-reduce is n_p, this indicates that global convergence was achieved. Many supercomputers have the vendor-optimized version of `MPI_Allreduce`, which should perform better than a user-implemented binary-tree based algorithm.

5. Experimental setup

We conducted all of our experiments on either the Haswell or Knights Landing (KNL) nodes of the Cori supercomputer at NERSC. Each of the Haswell nodes has two 16-core Intel Xeon E5-2698 v3 Haswell CPUs and 128 GB of main memory, while each of the KNL nodes has 68-core Intel Xeon Phi 7250 KNL CPUs on a single socket, and 16 GB of MCDRAM and 96 GB of DDR4 memories. These nodes are connected through the Cray Aries interconnect with Dragonfly topology.

We loaded the latest version (`7.7.0`) of the Cray MPI (a derivative of MPICH), which is system installed on Cori. The C solver calls `MPI_Init_thread` with the `MPI_THREAD_MULTIPLE` mode, and with the following environment values: `MPICH_MAX_THREAD_SAFETY=multiple`, `MPICH_NEMESIS_ASYNC_PROGRESS=1`, and `MV2_ENABLE_AFFINITY=0`. We then compiled the code using Cori's compiler wrapper `cc` for the Intel compiler version `2018.0.1.163`. Although Cray MPI is the default on Cori, Intel MPI can also be used. Thus, to test Intel MPI on the Haswell nodes, we compiled our code using the Intel C compiler `mpiicc` from Intel library version `2018.0.1.163` with the `-O3` optimization flag. For these experiments with Intel MPI, we used the environment values `I_MPI_ASYNC_PROGRESS=1` and `I_MPI_PROGRESS_PIN=yes`. To perform the dense vector operations, we linked our solver with Intel's Math Kernel Library (MKL) on the Haswell nodes; on the KNL nodes, we linked it with the Cray Scientific Libraries package, LibSci.

For our experiments with SHMEM on the Haswell CPUs, we used the default Cray SHMEM version 7.6.2. We then compiled the code using Cori's compiler wrapper `cc` for the default Intel compiler and linked it with MKL.

Since we focus on square processor grids, we launched eight processes per socket, and thus 16 processes per node, on Haswell nodes, while we used 36 processes per node on KNL nodes. We consider the computed local solution had converged when the local residual ℓ_2-norm is reduced by at least seven orders of

magnitude. To show the variation in performance, we report the results of 3–5 independent runs.

6. Communication benchmark results

Before investigating the performance of the solver, we profile the performance of MPI in various modes on the Cori supercomputer.

6.1. Benchmark results on Haswell

Fig. 4(a) shows benchmark results for different modes of communication using Cray MPI across two Haswell nodes. Data was exchanged 1000 times for different data lengths, N. Two-sided communication (i.e., `MPI_Send` and `MPI_Recv`) is compared to three different ways of performing one-sided communication:

- Use `MPI_Win_lock_all` and `MPI_Win_unlock_all`. Send all data (length N) with a single `MPI_Put`.
- Use `MPI_Win_lock` and `MPI_Win_unlock` with `MPI_LOCK_SHARED`. Send all data (length N) with a single `MPI_Put`.
- Use `MPI_Win_lock` and `MPI_Win_unlock` with `MPI_LOCK_SHARED`. Send each double precision element with a separate `MPI_Put`.

In all cases, the epoch is opened once before the first `put` and closed after the last `put`.

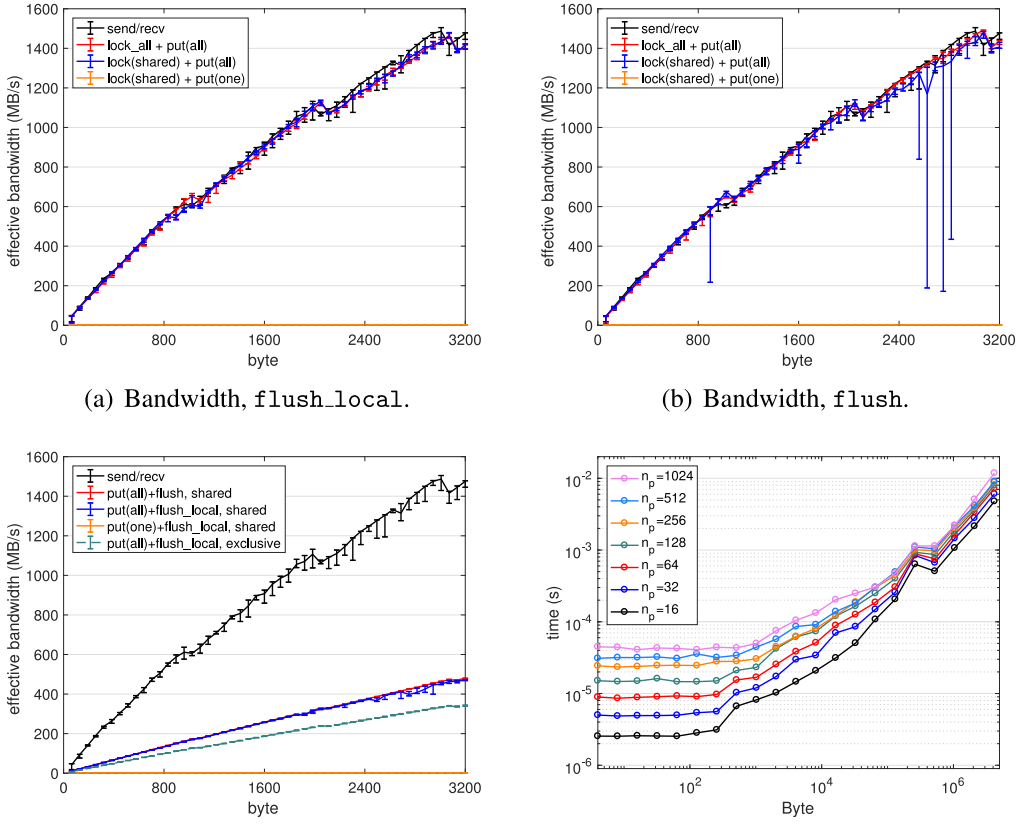
We used `MPI_Win_flush_local` after putting all the data (length N), which ensures that the data is flushed from the origin process; note however that this does not enforce an immediate update at the target. We see that sending one element at a time leads to significantly lower bandwidth utilization compared to sending one accumulated message at once. We also see that using `lock_all` or `lock` did not affect the performance. In these cases, one-sided communication obtained about the same performance as two-sided communication.

Fig. 4(b) shows results for the same cases as Fig. 4(a), except that `MPI_Win_flush` is used to make sure the data transfer is completed at the target process, in contrast with `MPI_Win_flush_local` which enforces only origin completion. We did not see significant differences in the average performance using either `flush` or `flush_local`. However, some runs gave very low bandwidth, as shown by the error bars in the figure, indicating how the tighter synchronization with the target in `flush` propagate delays to the origin process and increase performance variability.

Fig. 4(c) shows the performance where we now open and close the epoch for each one-sided communication. We also tested the alternative of opening an epoch using `MPI_Win_lock` for exclusive data access. We clearly see that although opening the epoch for shared data access was slightly cheaper than opening the epoch for exclusive data access, opening and closing the epoch each time added significant overhead to one-sided communication.

All these benchmark results show that, here, one-sided internode communication is not faster than two-sided communication—i.e., any performance benefit of the asynchronous solver is not due to faster data transfer. The results also suggest aggregating messages if possible, and avoiding opening and closing the epoch if not necessary.

In Fig. 4(d), we used the Intel MPI Benchmark (IMB) for benchmarking the performance of all-reduce (since our synchronous



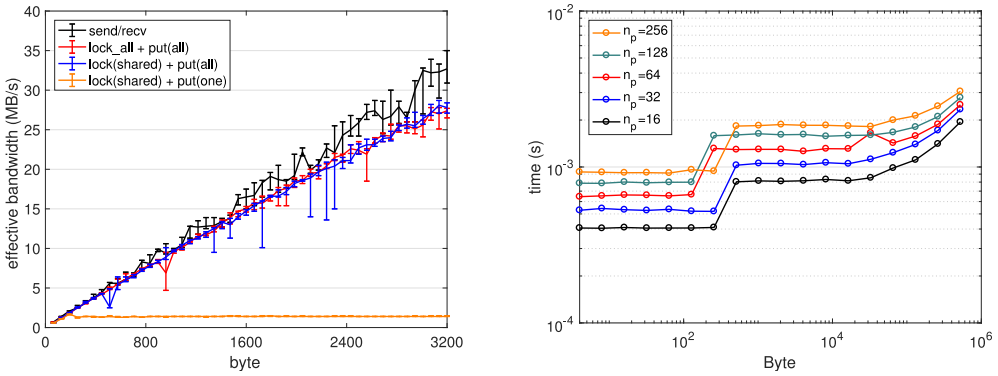
(a) Bandwidth, `flush_local`.

(b) Bandwidth, `flush`.

(c) Bandwidth, `lock`.

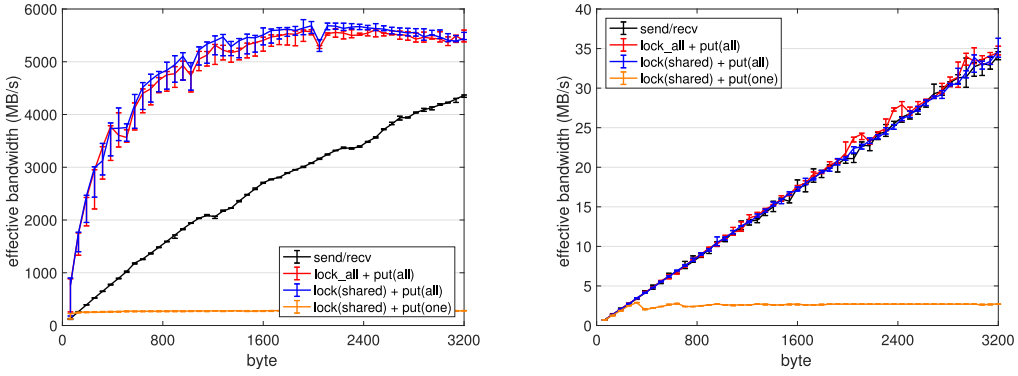
(d) Latency (n_p is the number of processes).

Fig. 4. Performance of Cray MPI. In (a)–(c), the markers show the median bandwidths of the five separate runs, while the error bars show the maximum and minimum bandwidths.



(a) Bandwidth, `flush_local`. (b) Latency (n_p is the number of processes).

Fig. 5. Performance of Intel MPI using the same configurations as in Fig. 4 (a) and (d).



(a) Cray MPI, `flush_local`. (b) Intel MPI, `flush_local`.

Fig. 6. Effective bandwidth between two processes on the same node.

termination detection algorithm sums a single integer value among all the processes). The figure shows the increasing cost of all-reduce as the number of MPI processes increases.

Fig. 5(a) and 5(b) show results for Intel MPI corresponding to Fig. 4(a) and 4(d), respectively. Intel MPI is not officially supported on Cori, and we clearly see the lower performance of Intel MPI for short messages compared to the vendor-optimized Cray MPI. We will exploit this performance difference between these two implementations, giving different relative communication vs. computation cost, when testing solver performance.

Fig. 6 shows performance of Cray MPI for processes on the same node. Notably, significantly higher bandwidth can be utilized in this case compared to those on two different nodes, as shown earlier in Fig. 4(a). Even with the balanced partitioning of the problem, these different costs of inter/intra-node communication could lead to a load imbalance among the processes—where the synchronous iteration with two-sided communication will block at the global or neighborhood synchronization points until the communication with all the participating processes are completed. On the other hand, asynchronous communication avoids these synchronization points. Furthermore, using Cray MPI between the two processes on the same node, one-sided communication was faster than two-sided communication.

Finally, to study how well `MPI_Put` overlaps with local computation, we modified the non-blocking collective benchmark in IMB. The results in Table 1 show that although Intel MPI asynchronous communication was slower than that of Cray MPI, it provided better overlap of between 50 ~ 57%, compared with the overlap of between 16 ~ 35% obtained by Cray MPI.

6.2. Benchmark results on knights landing

Fig. 7 shows the benchmark results of using one-sided or two-sided MPI on the KNL nodes of Cori. Communication is slower between a pair of processes on KNL nodes than on the Haswell nodes; multiple KNL processes may need to utilize the network interface to saturate the bandwidth available. We also see, in the KNL case, that data transfer is significantly faster between processes on the same node. These non-uniform costs of inter/intra-node communication again lead to imbalance among the processes even when work is partitioned evenly.

7. Solver performance results

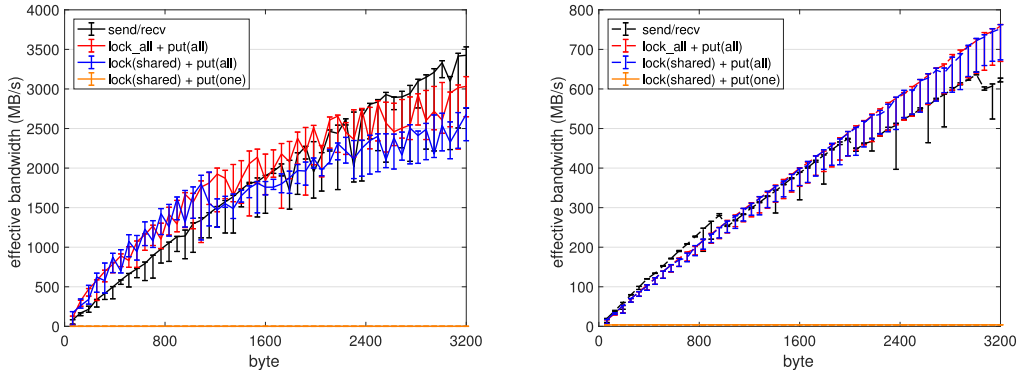
We now compare the performance of synchronous and asynchronous versions of the optimized Schwarz solver.

7.1. Convergence behavior with MATLAB

Since it is challenging to analyze solver behavior once asynchronicity is enabled, we first study solver performance by simulating the asynchronous iterations using MATLAB. We use a 2×2 processor grid, and these four processes, or subdomains, are identified by (1,1), (1,2), (2,1), and (2,2). Fig. 8 shows the convergence

Table 1
Communication and computation overlap; (1) pure communication time t_{pure}: the time between a call to `MPI_Put` immediately followed by a call to `MPI_Win_flush_local`, (2) computation time t_{CPU}: 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}, (3) total time t_{ovrl}: the time to finish the simultaneous communication and computation, and (4) overlap: the percentage computed as $(t_{\text{pure}} + t_{\text{CPU}} - t_{\text{ovrl}})/\min(t_{\text{pure}}, t_{\text{CPU}})$. The time is in microseconds.

	Cray MPI				Intel MPI			
bytes	t_{ovrl}	t_{pure}	t_{CPU}	overlap %	t_{ovrl}	t_{pure}	t_{CPU}	overlap %
4	4.28	2.30	2.37	16.59	155.94	98.78	123.24	53.62
8	4.22	2.28	2.32	16.23	160.53	99.14	128.57	52.26
16	4.10	2.22	2.24	15.80	155.23	95.66	125.15	52.40
32	4.16	2.27	2.30	18.05	160.60	98.20	128.68	51.51
64	4.05	2.17	2.22	15.28	159.99	96.07	128.11	50.10
128	4.18	2.29	2.34	19.12	160.20	98.69	127.80	51.87
256	4.40	2.61	2.66	32.75	158.63	98.71	125.78	52.36
512	4.33	2.62	2.62	34.87	152.51	99.99	120.96	56.58
1024	4.37	2.66	2.66	35.61	153.46	100.81	122.56	57.04



(a) Bandwidth within 1 node. (b) Bandwidth across 2 nodes.

Fig. 7. Performance of Cray MPI on KNL nodes using `flush_local`.

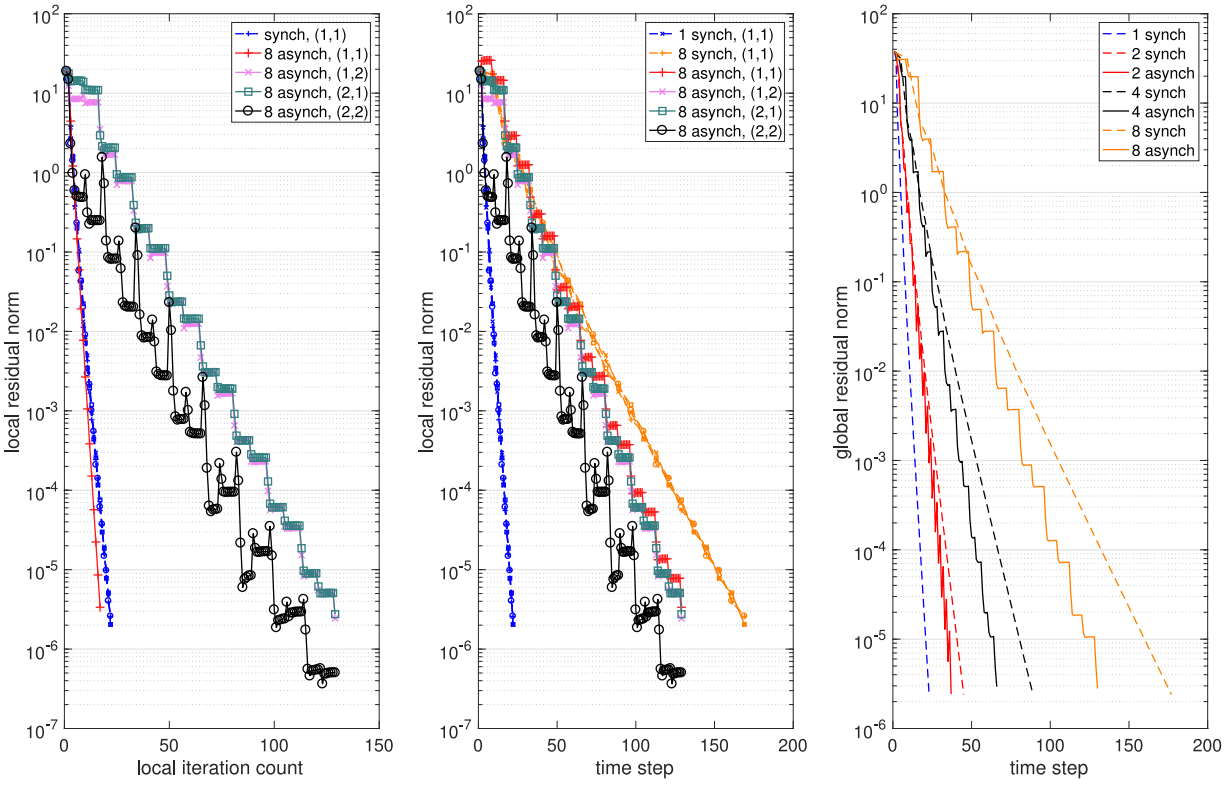
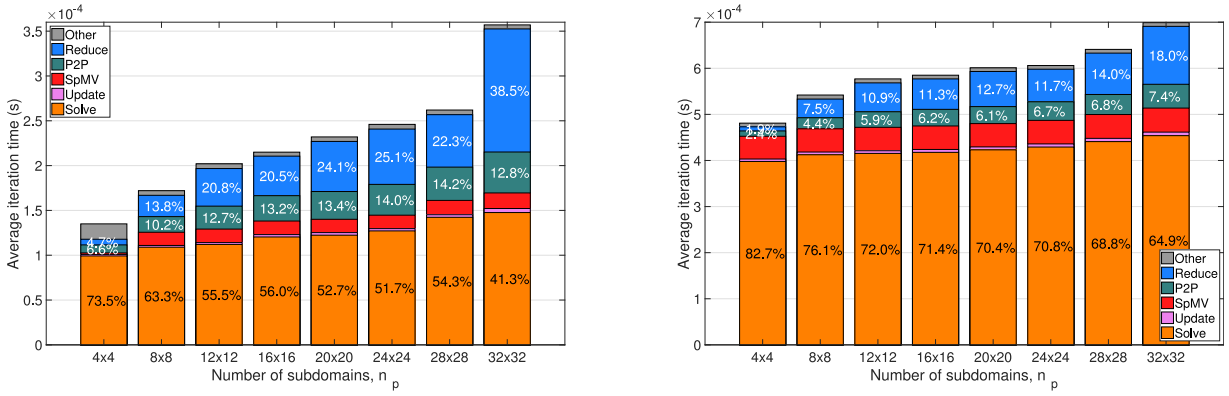


Fig. 8. Convergence of synchronous and asynchronous iterations, where "8 asynch, (1,1)" and "8 asynch" show the (1,1)th process' local residual norm and the global residual norm, respectively, when one of the processes runs 8 × slower than the others ($m = 64,\ p = 2,\ \alpha = 0.055$) (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.).

Table 2
One-sided communication schemes for asynchronous solver using MPI or SHMEM.

Notation	Description
`flush`	call `MPI_Win_flush` right after `MPI_Put` to complete data copy at target
`flush-local`	call `MPI_Win_flush_local` right after `MPI_Put` to complete data copy at origin
`overlap`	call `MPI_Win_flush` before next `MPI_Put` to overlap data copy with computation
`overlap-local`	call `MPI_Win_flush_local` before next `MPI_Put` to overlap data copy with computation
`put`	call `shmem_fence` right after `shmem_double_put`
`put-overlap`	call `shmem_fence` before next `shmem_double_put`



(a) $m = 32$. (b) $m = 64$.

Fig. 9. Breakdown of average synchronous iteration time (Cray MPI), where "Reduce" is the global collective needed for the convergence check, "P2P" is the neighborhood communication of the interface elements, "SpMV" is the sparse-matrix vector multiply needed for computing the local residual norm, "Update" is for updating the interface elements of the right-hand-side, and "Solve" is the local subdomain solve.

of the residual norms where we assumed that the communication is instantaneous and hence it cannot be overlapped with the computation, but process (1,1) is running 8 × slower than the others.

The plot on the left shows the convergence of the local residual norms with respect to the local iteration step. We see that the residual norm for the slower process (denoted by the red cross mark '+') converges faster than those obtained when all the processes are progressing at the same speed (shown as the dotted blue lines). This is because the slower process updates its local solution using more accurate boundary information from the faster neighboring processes.

The plot in the middle then shows the local residual norm with respect to the time step, where the unit time is considered to be the one iteration step of the faster processes. We see that the convergence of the neighboring process is slowed down by the slow process because they do not receive the update on the boundary from the (1,1)th process for a fixed number of iterations. Since the (2,2)th process has a smaller overlap with the (1,1)th process its local convergence (denoted by the black circle markers 'o') is affected less by the slow process. However, the slowdown of the convergence eventually propagates from the (1,1)th process to all the processes. In addition, especially in the beginning, the local residual norm of the (2,2)th process spikes up when the new information arrives from the slow process, and a couple iterations are needed to bring the residual norm back down.

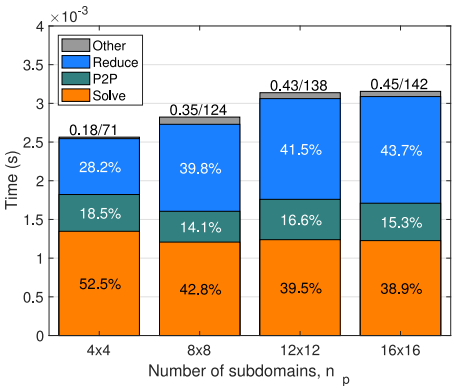
Finally, the figure on the right shows the global residual norm and demonstrates that, compared with the synchronous iteration, the asynchronous iteration may reduce the time to solution, even without hiding the communication, because the slower process converges with fewer local iterations.

7.2. Solver performance on Haswell

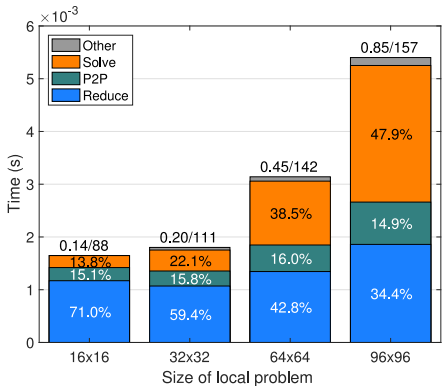
We now compare the performance of synchronous and asynchronous optimized Schwarz on the Cori supercomputer. In Table 2, we list the one-sided communication schemes used by our asynchronous solver in our experiments.

Fig. 9 shows the breakdown of the average synchronous iteration time using Cray MPI on the Haswell CPUs. We fixed the local problem size but varied the number of subdomains. Since the local problem size is fixed, the time needed for the local subdomain solve and for the point-to-point communication are expected to stay constant, while the time for the all-reduce is expected to increase with the increasing number of subdomains (e.g., $O(\log(n_p))$). We observed that as we increase the process count, a small number of processes spend more time solving the local problem at some iterations (due to hardware noises). This increased the average solve time slightly with the increasing number of processes. Nevertheless, we see that on a large enough number of processes, the communication becomes significant in the iteration time. In particular, the all-reduce needed to check for the global convergence is often more expensive than the point-to-point communication for exchanging the overlap elements with the neighboring processes. This may be because the communication latency cost for the global collective is greater than that for the neighborhood communication. In addition, however, the all-reduce time includes the faster processes' idling time while waiting for the slowest processes due to the load imbalance introduced by the neighborhood communication. Without termination detection, this idling time could appear at the neighborhood synchronization points (i.e., the implicit global synchronization).

Fig. 10(a) shows the breakdown of the iteration time using Intel MPI. We observed that compared with using the Cray compiler wrapper, the local solve time was longer using the Intel compiler without the Cray compiler wrapper. However, using Intel MPI, the communication becomes significant even on a smaller number of subdomains. Fig. 10(b) shows the breakdown of the average iteration time where we fixed the number of subdomains but varied the local problem size, again using Intel MPI. Since the number of subdomains is fixed, the time needed for the all-reduce

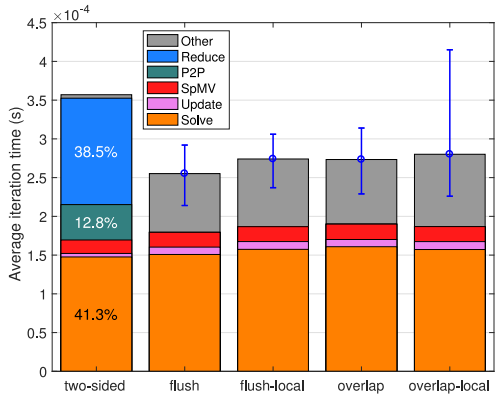


(a) Different numbers of subdomains (with fixed local problem size, $m = 64$ and $\alpha = 0.0065$).

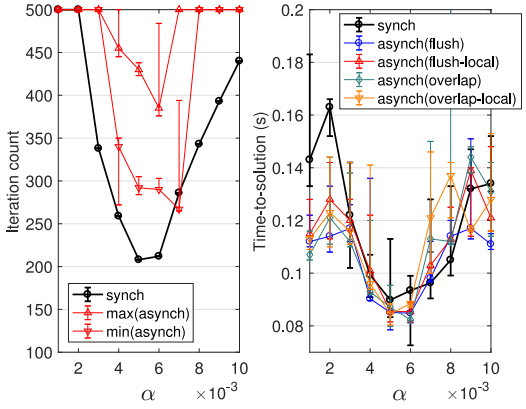


(b) Different local problem sizes (with fixed 16-by-16 processor grid, and $\alpha = 0.022$, 0.012, 0.0065, 0.0048 on $m = 16$, 32, 64, 96, respectively).

Fig. 10. Breakdown of average synchronous iteration time (Intel MPI). On top of each bar, we show the total solution time/the number of iterations.



(a) Breakdown of average iteration time using different asynchronous communication schemes, compared with the synchronous iteration (leftmost bar). The error bars show the maximum, mean, and minimum average iteration times among the processes.



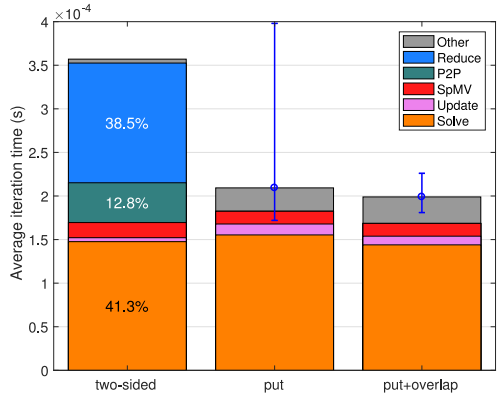
(b) Effect of α on the solver performance. In the legend, "flush"/"flush-local" calls `flush`/`flush_local` after `Put`, while "overlap"/"overlap-local" aims to overlap the communication with the computation using `flush`/`flush_local`.

Fig. 11. Performance of asynchronous solver using Cray MPI with $m = 32$ on 32-by-32 grid. The asynchronous communication did not seem to overlap with the local computation, and the asynchronous solver did not obtain significant performance gain.

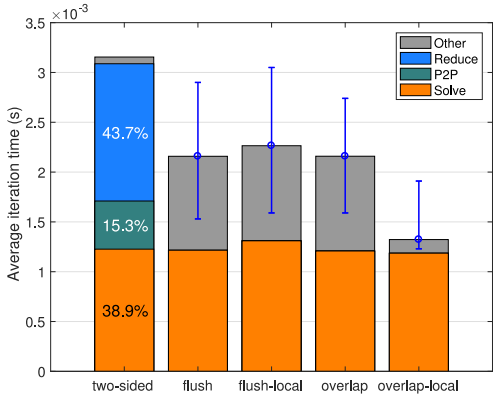
communication is expected to stay the same. On the other hand, we expected the time for the point-to-point communication to increase linearly with the problem size m, while the time for the local subdomain solve is expected to increase cubically with m. Again, we see that the communication can be significant in the iteration time.

Fig. 11(a) then compares the average asynchronous and synchronous iteration time using Cray MPI. Even with the balanced partition on the homogeneous nodes, the error bars in the figure show significant variations in the iteration time among the processes. The asynchronous termination detection algorithm avoids the global all-reduce needed by the synchronous algorithm, making the asynchronous iteration faster than the synchronous iteration. In all the cases using Cray MPI, the asynchronous communication seems not to overlap with the local computation. Fig. 11(b) shows the corresponding time-to-solution using the different asynchronous communication schemes. Using Cray MPI, the asynchronous iterations obtained a solution time similar to that of the synchronous iteration. The primary reason is that the asynchronous communication did not progress behind the local computation. This led not only to the sub-optimal asynchronous iteration time, but also to the processes iterating with old information on the boundary, and an afferent increase in the iteration count for all the processes. Overall, compared to the synchronous iteration, the asynchronous iteration could not reduce the time-to-solution because its reduction in the iteration time could not offset the increase in the iteration count.

Fig. 12 shows similar breakdowns for the asynchronous iteration using Cray SHMEM and Intel MPI. We now see that the average asynchronous iteration time is much shorter using Cray SHMEM's `put-overlap` or Intel MPI's `overlap-local` that is designed to overlap the communication. Fig. 13 shows the corresponding time-to-solution using the different one-sided communication configurations. We achieved the best solver performance

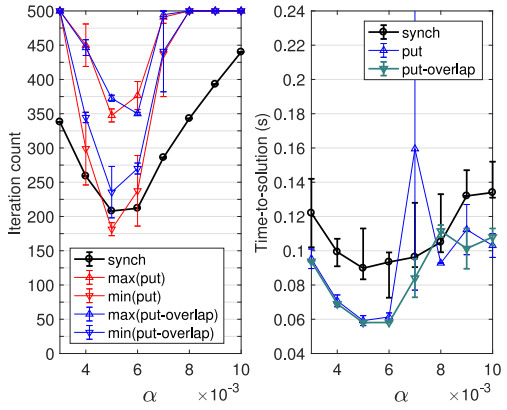


(a) Cray SHMEM with $m = 32$ on 32-by-32 grid.

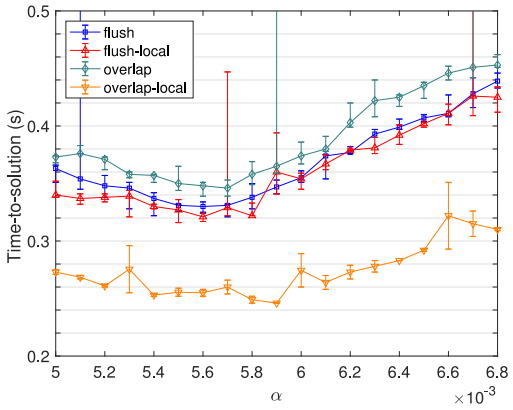


(b) Intel MPI with $m = 64$ on 16-by-16 grid.

Fig. 12. Breakdown of average asynchronous iteration time. The error bars show the maximum, mean, and minimum average iteration times among the processes.



(a) Cray SHMEM with $m = 32$ on 32-by-32 processor grid. With the optimal α, the median and minimum time to solution was faster using the asynchronous iteration with the respective speedups of about $1.5\times$ and $1.2\times$, compared with using the synchronous iteration.



(b) Effect of asynchronous communication schemes using Intel MPI with $m = 64$ on 16-by-16 processor grid.

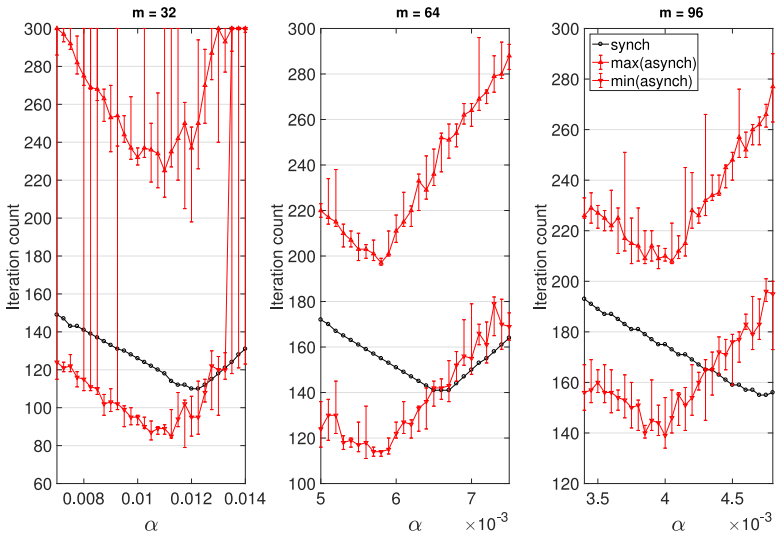
Fig. 13. Effect of parameter α on asynchronous solver performance.

using the communication scheme that allows good overlapping of the communication and local computation.

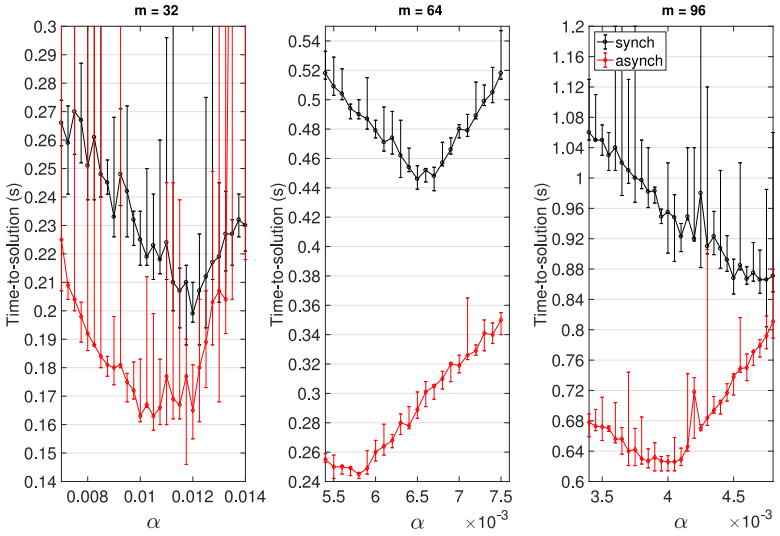
To show the benefits of the asynchronous iteration, we provide more detailed performance results using Intel MPI. Fig. 14 shows the effects of the parameter α on the performance of the synchronous and asynchronous solvers for three different local problem sizes. We use `flush_local` for the asynchronous communication to overlap with the computation (i.e., `overlap-local`). As a larger value of α is used, the Robin transmission condition (used by the optimized Schwarz on the interface) becomes closer to the Dirichlet condition used by the classical Schwarz method, leading to a larger iteration count. Optimal performance is obtained using a smaller α for a larger local problem, and the figures show optimized Schwarz's superior performance over the classical Schwarz. We also see that the asynchronous method tends to prefer a smaller α compared to the synchronous method. When the local problem is smaller, communication becomes more expensive than the local computation. Therefore, each process is likely to perform more iterations with the older values of the solution vector on the interfaces. The arrival of the new interface values could result in a significant increase in the local residual norm. As a result, there is a wider variation in the iteration counts needed by the asynchronous methods on a smaller m.

We now compare the convergence of the residual norm using synchronous and asynchronous iterations. At each iteration, each MPI process locally checks for the solution convergence by computing its local residual norm on its interior points. The process saves this local residual norm, along with the time stamp, in its local data structure. To plot the convergence history of the global residual norm, after the iteration terminates, we estimate the global residual norm by accumulating the local residual norms along with the time stamp. With the asynchronous communication, the local residual norm may be computed using the solution vector's interface elements that may not be most recent. We explicitly computed the global residual norm after the iteration was terminated by the asynchronous algorithm, and it was on the same order as the specified tolerance.

Fig. 15(a) and 15(b) show the convergence of the global residual norm with respect to time when termination detection is turned off and on, respectively. We clearly see that synchronous termination detection adds significant overhead to the iteration time, while asynchronous termination detection only has a small overhead. In both cases, the asynchronous iteration reached the global convergence faster than the synchronous iteration. In the figures, we also show the convergence of the synchronous iteration when we delay synchronizing `MPI_Isend` to hide the

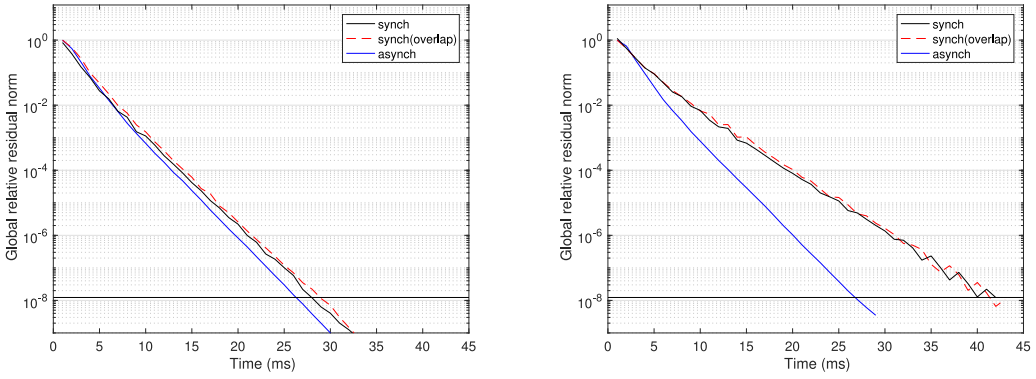


(a) Iteration count.



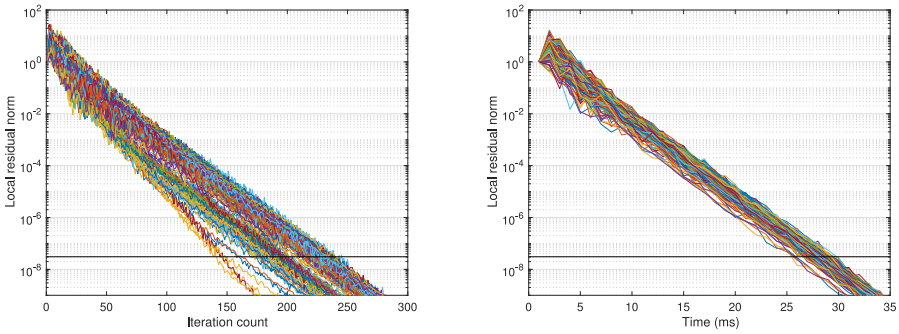
(b) Time to solution.

Fig. 14. Effect of the parameter α on the performance of synchronous and asynchronous solvers using 16-by-16 processor grid. Using the optimal value of α, both the median and minimum time to solution is faster using the asynchronous iteration with the speedup of about $1.2\times$, $1.8\times$, or $1.3\times$ with $m = 32$, 64, or 96, respectively, compared with using the synchronous iteration. However, for a small local problem (e.g., $m = 32$) where the iteration time is dominated by the communication, the performance of the asynchronous solver can become unstable due to the irregular convergence.

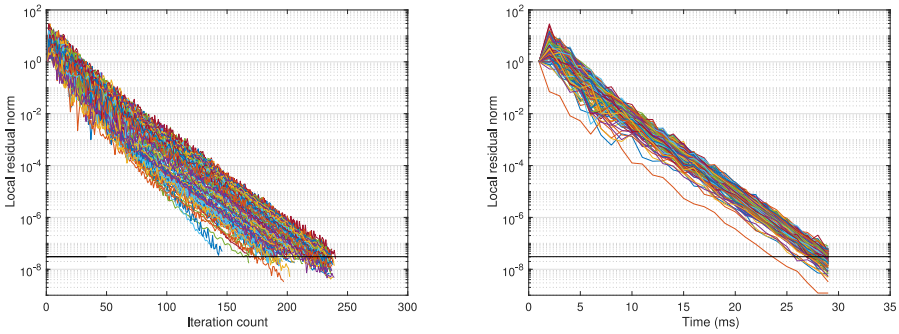


(a) Without termination detection. (b) With termination detection.

Fig. 15. Global residual norm convergence history ($m = 64$ and $\alpha = 0.0065$ on 16-by-16 processor grid).

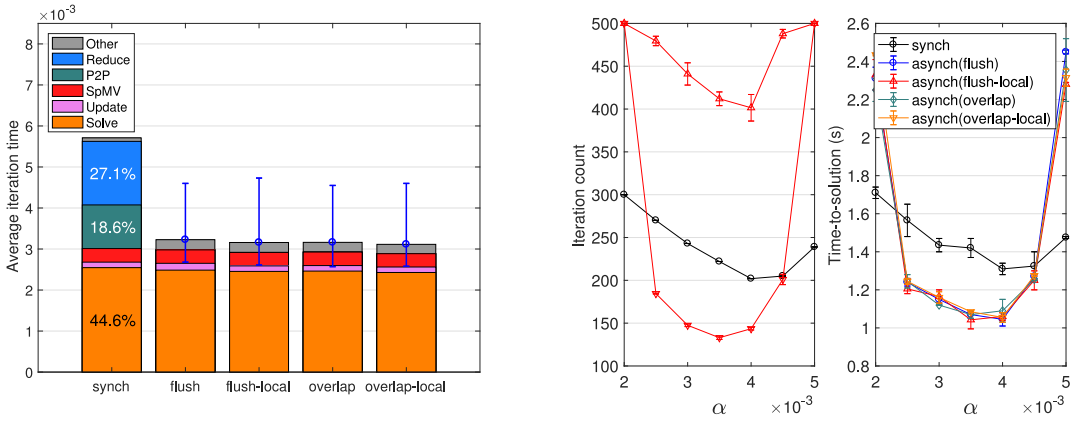


(a) Without termination detection (vs. iteration count). (b) Without termination detection (vs. time).



(c) With termination detection (vs. iteration count). (d) With termination detection (vs. time).

Fig. 16. Local residual norm convergence history ($m = 64$ and $\alpha = 0.0065$ on 16-by-16 processor grid). The synchronous method converged with 141 iterations.



(a) Breakdown of iteration time. (b) Effect of α on time-to-solution.

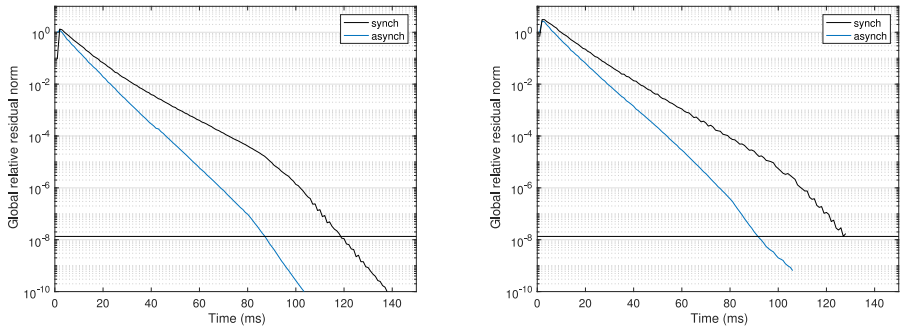
Fig. 17. Breakdown of average synchronous iteration time on KNL, with $m = 64$ and 24-by-24 processor grid. Using the optimal value of α, both the median and minimum time to solution was faster using the asynchronous iteration with the speedup of about 1.2×, compared with using the synchronous iteration (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.).

communication (as described in Section 4.1). Unfortunately, the communication may not overlap between the processes without load imbalance (e.g., a pair of the slowest neighboring processes on different nodes), and we see that the iteration time was not reduced by trying to overlap the communication.

Fig. 16 then shows the variation in the convergence of the local residual norms across the different processes that used the asynchronous iterations. Although there are some spikes in the local residual norm convergence, they are small enough to validate our global convergence detection based on the minimum local residual norm.

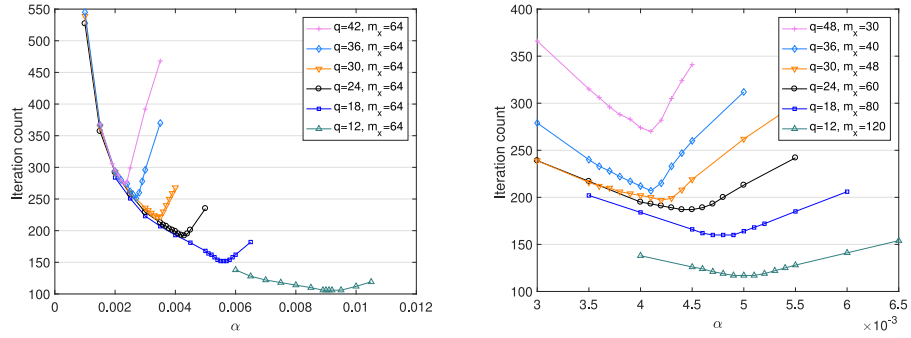
7.3. Solver performance on knights landing

To study the performance of asynchronous optimized Schwarz on a different hardware architecture, we show its performance on Cori's KNL nodes in Fig. 17. The main difference from the results for the Haswell nodes is that, using Cray MPI on the KNL nodes, the average asynchronous iteration time was shorter relative to the synchronous iteration time (e.g., more processes per node). Because of this, the asynchronous method was able to outperform the synchronous method. To summarize the results, Fig. 18 shows the convergence of the global residual norm with and without



(a) Without termination detection. (b) With termination detection.

Fig. 18. Global residual norm convergence history ($m = 64$ and $\alpha = 0.004$ on 24-by-24 processor grid).



(a) Weak scaling. (b) Strong scaling.

Fig. 19. Effect of parameter α on synchronous optimized Schwarz iteration counts, using a q-by-q processor grid and m_x-by-m_x local mesh size.

Table 3
Parallel scaling of synchronous and asynchronous iterations with termination detection. In the synchronous case, two variants are tested, one using `MPI_Wait` (standard variant) and one using `MPI_Waitall`. In the asynchronous case, the range given by "iters" is the fewest and largest number of local iterations that were executed across all processes. The label "rel. rnorm" denotes the relative residual norm reduction that was measured after completion of the iterations.

(a) Weak scaling ($m = 64$).

q	12	18	24	30	36	42
α	0.0090	0.0055	0.0043	0.0035	0.0027	0.0023
time synch wait (s)	0.616	0.738	0.800	0.982	1.13	1.28
time synch waitall (s)	0.646	0.670	0.990	1.120	1.10	1.51
α	0.0086	0.0051	0.0038	0.0031	0.0026	0.0023
time asynch (s)	0.586	0.611	0.741	0.883	1.06	1.28
iters synch	105	161	196	249	274	284
iters asynch	154 ∼ 229	138 ∼ 228	206 ∼ 278	273 ∼ 331	294 ∼ 371	345 ∼ 449
rel. rnorm synch	4.5×10^{-8}	4.9×10^{-8}	5.1×10^{-8}	6.4×10^{-8}	5.0×10^{-8}	5.4×10^{-8}
rel. rnorm asynch	6.6×10^{-10}	4.0×10^{-9}	3.0×10^{-9}	3.7×10^{-9}	7.6×10^{-9}	3.0×10^{-10}

(b) Strong scaling ($m \cdot q = 1440$).

q	12	18	24	30	36	48
α	0.0050	0.0048	0.0045	0.0042	0.0041	0.0041
time synch wait (s)	1.75	0.994	1.150	0.631	0.512	0.463
time synch waitall (s)	1.82	0.879	0.851	0.992	0.477	0.647
α	0.0044	0.0041	0.0041	0.0041	0.0041	0.0041
time asynch (s)	1.89	0.992	0.669	0.543	0.505	0.413
iters synch	126	166	193	203	213	273
iters asynch	127 ∼ 169	176 ∼ 229	155 ∼ 275	280 ∼ 366	338 ∼ 462	414 ∼ 635
rel. rnorm synch	5.1×10^{-8}	1.0×10^{-7}	7.2×10^{-8}	5.7×10^{-8}	6.3×10^{-8}	6.5×10^{-8}
rel. rnorm asynch	3.4×10^{-8}	8.3×10^{-9}	3.8×10^{-9}	6.1×10^{-10}	9.2×10^{-10}	7.3×10^{-10}

Table 4
Parallel scaling of synchronous and asynchronous iterations with timer.

(a) Weak scaling ($m = 64$ for 0.5 s).						
q	12	18	24	30	36	42
α	0.0090	0.0055	0.0043	0.0035	0.0027	0.0023
rel. rnorm synch	3.9×10^{-5}	1.3×10^{-7}	4.1×10^{-4}	2.5×10^{-3}	2.6×10^{-2}	3.6×10^{-2}
α	0.0086	0.0051	0.0038	0.0031	0.0026	0.0023
rel. rnorm asynch	8.1×10^{-9}	5.8×10^{-6}	7.4×10^{-6}	2.0×10^{-4}	5.7×10^{-4}	2.3×10^{-3}
(b) Strong scaling ($m \cdot q = 1440$ for 0.3 s).						
q	12	18	24	30	36	48
α	0.0050	0.0048	0.0045	0.0042	0.0041	0.0041
rel. rnorm synch	7.4×10^{-1}	3.6×10^{-2}	5.4×10^{-2}	1.1×10^{-2}	9.2×10^{-5}	1.7×10^{-5}
α	0.0044	0.0041	0.0041	0.0041	0.0041	0.0041
rel. rnorm asynch	3.3×10^{-1}	2.6×10^{-2}	7.9×10^{-4}	5.7×10^{-7}	9.1×10^{-7}	5.9×10^{-9}

termination detection. In this case, the performance of the synchronous iteration was not significantly improved by removing the global synchronization needed for the convergence check. In Fig. 17(a), the time needed for the global synchronization (colored in blue) is the combination of the time needed for the collective communication and the idling time needed to wait for the slowest process. Avoiding the global synchronization reduces the collective communication cost, while the load imbalance is eventually exposed at the neighborhood synchronization points to exchange the interface data. Cray MPI had a greater difference in the inter and intra communication costs, compared to Intel MPI used in Fig. 15, where removing the global synchronization had more significant effects.

Finally, we compare the synchronous and asynchronous optimized Schwarz methods in terms of their weak and strong parallel scaling behavior. In these tests, we selected values of α that obtain the fastest convergence of the synchronous method for each combination of local grid size m-by-m and processor grid size q-by-q (see Fig. 19). We then adjusted these values of α for the asynchronous method. The asynchronous method typically prefers a smaller value of α than the synchronous method, as shown previously, for example, in Fig. 14.

Table 3 shows the parallel scaling behavior of the methods with termination detection activated. The time-to-solution is similar or smaller for the asynchronous method than for the synchronous method. We also observe that, especially with a larger number of processes, the asynchronous method usually performs more iterations before terminating. This indicates that there is a delay in detecting global convergence in the asynchronous method. Consequently, the measured residual norm after completion of the iterations is smaller in the asynchronous method than in the synchronous method.

Table 4 shows results without termination detection. Each process simply terminates after a specified time. Generally, as the problem size increases (weak scaling), the convergence rate slows down. As the number of subdomains increases while the global problem size is fixed (strong scaling), the convergence rate improves. We observe that the measured residual norm after completion of the iterations is generally smaller for the asynchronous method than for the synchronous method. For strong scaling, the communication to computation ratio increases as the number of processes increases. When this ratio is larger, we observe that the asynchronous method has a greater advantage over the synchronous method.

8. Conclusion

In this paper, we used MPI one-sided communication to implement asynchronous optimized Schwarz on distributed-memory computers. Our experimental results suggest that the performance of the asynchronous solver depends heavily on the software and hardware support for remote memory access communication. With support for overlapping communication with computation, the asynchronous solver may outperform the synchronous solver, even for a balanced distribution of the problem on the current leadership supercomputer with uniform nodes and high-bandwidth interconnect.

In future work, we will study the effect of race conditions on asynchronous iterative methods, where two processes writing to the same buffer simultaneously may lead to an erroneous result. We plan to compare the cost of error detection and correction against the potential slow down of convergence without such mechanisms (with respect to the frequency of errors).

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Numbers #DE-SC0016513 and #DE-SC-0016564. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

References

[1] K.B. Ferreira, P. Bridges, R. Brightwell, Characterizing application sensitivity to OS interference using kernel-level noise injection, in: Proceedings of the SC - International Conference for High Performance Computing, Networking, Storage and Analysis, 2008, pp. 1–12.
[2] Office of Science and Office of Advanced Scientific Computing Research, Scientific grand challenges: architectures and technology for extreme scale computing, Technical Report, U.S. Department of Energy, 2009.
[3] Office of Science and Office of Advanced Scientific Computing Research, Exascale Programming Challenges, Technical Report, U.S. Department of Energy, 2011.
[4] I. Bethune, J. Bull, N. Dingle, N. Higham, Performance analysis of asynchronous Jacobi's method implemented in MPI, SHMEM and openmp, Int. J. High Perform. Comput. Appl. 28 (2014) 97–111.
[5] F. Magoulès, D. Szyld, C. Venet, Asynchronous optimized Schwarz methods with and without overlap, Numer. Math. 137 (2017) 199–227.
[6] V. Dolean, P. Jolivet, F. Nataf, An Introduction to Domain Decomposition Methods: Algorithms, Theory, and Parallel Implementation, Society for Industrial and Applied Mathematics, Philadelphia, 2015.
[7] M. Gander, Optimized Schwarz methods, SIAM J. Numer. Anal. 44 (2006) 699–731.
[8] F. Nataf, Recent Developments on Optimized Schwarz Methods, in: O.B. Widlund, D.E. Keyes (Eds.), Domain Decomposition Methods in Science and Engineering XVI, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 115–125.
[9] F. Magoulès, Asynchronous Schwarz methods for peta and exascale computing, in: B. Topping, P. Iványi (Eds.), Developments in Parallel, Distributed, Grid and Cloud Computing for Engineering, Saxe-Coburg, Stirlingshire, UK, 2013, pp. 229–248.

[10] M. Chau, T. Garcia, P. Spiteri, Asynchronous Schwarz methods applied to constrained mechanical structures in grid environment, Adv. Eng. Softw. 74 (2014) 1–15.
[11] A. Frommer, H. Schwandt, D. Szyld, Asynchronous weighted additive Schwarz methods, Electron. Trans. Numer. Anal. 5 (1997) 48–61.
[12] E. Laitinen, A. Lapin, J. Pieskä, Asynchronous domain decomposition methods for continuous casting problem, J. Comput. Appl. Math. 194 (2003) 393–413.
[13] J. Wolfson-Pou, E. Chow, Reducing communication in distributed asynchronous iterative methods, in: Proceedings of the ICCS Workshop on Mathematical Methods and Algorithms for Extreme Scale, 2016, pp. 1906–1916.
[14] J. Wolfson-Pou, E. Chow, Distributed Southwell: an iterative method with low communication costs, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC), 2017.
[15] J. Wolfson-Pou, E. Chow, Convergence models and surprising results for the asynchronous Jacobi method, in: Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2018.
[16] J. Wolfson-Pou, E. Chow, Modeling the asynchronous Jacobi method without communication delays, J. Parallel Distrib. Comput. (2019) toappear.
[17] D. El Baz, Communication study and implementation analysis of parallel asynchronous iterative algorithms on message passing architectures, in: Proceedings of the EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing, 2007, pp. 77–83.
[18] J. Bahi, S. Contassot-Vivier, R. Couturier, F. Vernier, A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms, IEEE Trans. Parallel Distrib. Syst. 16 (2005) 4–13.
[19] J. Bahi, S. Contassot-Vivier, R. Couturier, An efficient and robust decentralized algorithm for detecting the global convergence in asynchronous iterative algorithms, in: Proceedings of the International Conference on High Performance Computing for Computational Science (VECPAR), 2008, pp. 240–254.
[20] F. Magoulès, G. Gbikpi-Benissan, JACK: An asynchronous communication kernel library for iterative algorithms, J. Supercomput. 73 (2017) 3468.
[21] F. Magoulès, G. Gbikpi-Benissan, JACK2: An MPI-based communication library with non-blocking synchronization for asynchronous iterations, Adv. Eng. Softw. 119 (2018) 116–133.
[22] M. Si, A.J. Peña, J. Hammond, P. Balaji, M. Takagi, Y. Ishikawa, Casper: An asynchronous progress model for MPI RMA on many-core architectures, in: Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2015, pp. 665–676.
[23] H. Zhou, J. Gracia, Asynchronous progress design for a MPI-based PGAS one-sided communication system, in: Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS), 2016, pp. 999–1006.
[24] J.M. Bahi, S. Contassot-Vivier, R. Couturier, Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms, IEEE Trans. Parallel Distrib. Syst. 16 (2005) 289–299.
[25] J.M. Bahi, S. Contassot-Vivier, R. Couturier, Performance comparison of parallel programming environments for implementing AIAC algorithms, J. Supercomput. 35 (2006) 227–244.
[26] V. Dolean, M. Gander, L. Gerardo-Giorda, Optimized Schwarz methods for Maxwell's equations, SIAM J. Scient. Comput. 31 (2009) 2193–2213.
[27] Y.-X. He, L. Li, S. Lanteri, T.-Z. Huang, Optimized Schwarz algorithms for solving time-harmonic Maxwell's equations discretized by a hybridizable discontinuous Galerkin method, Comput. Phys. Commun. 200 (2016) 176–181.
[28] P. Chevalier, F. Nataf, Symmetrized method with optimized second-order conditions for the Helmholtz equation, in: Domain Decomposition Methods, 10, 1997, pp. 400–407.
[29] M. Gander, F. Magoulès, F. Nataf, Optimized Schwarz methods without overlap for the Helmholtz equation, SIAM J. Scient. Comput. 24 (2002) 38–60.
[30] M. Gander, L. Halpern, F. Magoulès, An optimized Schwarz method with two-sided robin transmission conditions for the Helmholtz equation, Int. J. Numer. Methods Fluids 55 (2007) 163–175.
[31] O. Dubois, M. Gander, S. Loisel, A. St-Cyr, D. Szyld, The optimized Schwarz method with a coarse grid correction, SIAM J. Scient. Comput. 34 (2012) A421–A458.
[32] V. Martin, An optimized Schwarz waveform relaxation method for the unsteady convection diffusion equation in two dimensions, Appl. Numer. Math. 52 (2005) 401–428.
[33] M. Gander, L. Halpern, Optimized Schwarz waveform relaxation methods for advection reaction diffusion problems, SIAM J. Numer. Anal. 45 (2007) 666–697.
[34] The HSL Mathematical Software Library, MA57: Sparse symmetric system: multifrontal method, http://www.hsl.rl.ac.uk/catalogue/ma57.html.