SIAM J. SCI. COMPUT.
Vol. 42, No. 6, pp. C384–C409

SCALABLE ASYNCHRONOUS DOMAIN DECOMPOSITION SOLVERS*

CHRISTIAN GLUSA†, ERIK G. BOMAN†, EDMOND CHOW‡, SIVASANKARAN RAJAMANICKAM†, AND DANIEL B. SZYLD§

Abstract. Parallel implementations of linear iterative solvers generally alternate between phases of data exchange and phases of local computation. Increasingly large problem sizes and more heterogeneous compute architectures make load balancing and the design of low latency network interconnects that are able to satisfy the communication requirements of linear solvers very challenging tasks. In particular, global communication patterns such as inner products become increasingly limiting at scale. We explore the use of asynchronous communication based on one-sided Message Passing Interface primitives in the context of domain decomposition solvers. In particular, a scalable asynchronous two-level Schwarz method is presented. We discuss practical issues encountered in the development of a scalable solver and show experimental results obtained on a state-of-the-art supercomputer system that illustrate the benefits of asynchronous solvers in load balanced as well as load imbalanced scenarios. Using the novel method, we can observe speedups of up to four times over its classical synchronous equivalent.

Key words. asynchronous iteration, domain decomposition, Schwarz methods, chaotic relaxation

AMS subject classifications. 68W10, 65Y05, 68W15, 65N55

DOI. 10.1137/19M1291303

1. Introduction.

Multilevel methods such as multigrid and domain decomposition are among the most efficient and scalable solvers for partial differential equations developed to date. Adapting them to the next generation of supercomputers and improving their performance and scalability is crucial in the push toward exascale. Domain decomposition methods subdivide the global problem into subdomains, and then alternate between local solves and boundary data exchange. This puts a significant stress on the network interconnect, since all processes try to communicate at once. On the other hand, during the solve phase, the network is underutilized. The use of nonblocking communication can only alleviate this issue, but not fully resolve it. In asynchronous methods, on the other hand, computation and communication

*Submitted to the journal's Software and High-Performance Computing section October 11, 2019; accepted for publication (in revised form) August 3, 2020; published electronically December 14, 2020. Part of this work has been accepted for publication in the form of a proceedings paper by the 25th International Domain Decomposition Conference.

https://doi.org/10.1137/19M1291303

Funding: 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-SC-0016564. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

†Center for Computing Research, Sandia National Laboratories, Albuquerque, NM 87123 USA (caglusa@sandia.gov, egboman@sandia.gov, srajama@sandia.gov).

‡School of Computational Science and Engineering, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-4017 USA (echow@cc.gatech.edu).

§Temple University, Philadelphia, PA 19122-6094 USA (szyld@temple.edu).

occur at the same time, with some processes performing computation while others communicate, so that the network is consistently in use.

The term "asynchronous" can have several different meanings in the literature. In computer science, it is sometimes used to describe communication patterns that are nonblocking, so that computation and communication can be overlapped. Iterative algorithms that use such "asynchronous" communication yield the same iterates (results) up to round-off error, as they do not change the mathematical algorithm. In applied mathematics, on the other hand, "asynchronous" denotes parallel algorithms where each process (processor) proceeds at its own speed without synchronization. Thus, asynchronous algorithms go beyond the widely used bulk-synchronous parallel (BSP) model. More importantly, they are mathematically different than synchronous methods and generate different iterates. The earliest work in this area was called "chaotic relaxation" [11]. Both types of asynchronous approaches are expected to play an important role in future supercomputers. In this paper, we focus on asynchronous methods in the mathematical sense, and we will use the terms "asynchronous" and "synchronous" to distinguish between methods that are asynchronous and synchronous in the mathematical sense.

Domain decomposition solvers [16, 34, 33] are often used as preconditioners in Krylov subspace iterations. Unfortunately, the computation of inner products and norms widely used in Krylov methods requires global communication. Global communication primitives, such as `MPI_Reduce`, asymptotically scale as the logarithm of the number of processes involved. This can become a limiting factor when very large process counts are used. The underlying domain decomposition method, however, can do away with globally synchronous communication, assuming the coarse problem in multilevel methods can be solved in a parallel way. Therefore, we will focus on using domain decomposition methods purely as iterative methods in the present work. We will note, however, that the discussed algorithms could be coupled with existing pipelined methods [22] which alleviate the global synchronization requirement of Krylov solvers.

Another issue that is crucial to good scaling behavior is load imbalance. Load imbalance might occur due to heterogeneous hardware in the system, network noise, dynamic power capping [1], or local, problem specific causes, such as iteration counts for local solves that vary from subdomain to subdomain. The latter are especially difficult to predict, so that load balancing cannot occur before the actual solve. Therefore, processes in a synchronous parallel program must be idle until its slowest process has finished. In an asynchronous method, local computation can continue, and potentially improve the quality of the global solution.

An added benefit of asynchronous methods is that, since the interdependence between subdomains has been weakened, fault tolerance [9, 10] can be more easily achieved. When one process must stop, be it for a hard or a soft fault, it can be replaced without having to halt every other process.

The main drawback of asynchronous iterations is the fact that deterministic behavior is sacrificed. Consecutive runs do not produce the same result. (But one would hope that they are at most a distance proportional to the convergence tolerance apart from each other.) This also makes the mathematical analysis of asynchronous methods significantly more difficult than for its synchronous counterparts. Analytical frameworks for asynchronous linear (and nonlinear) iterations have long been available [11, 4, 5, 18], but generally cannot produce sharp convergence bounds except in the simplest cases.

The main contributions of our work are

- a novel asynchronous two-level domain decomposition method, scalable to thousands of processors,
- an empirical study of one-sided Message Passing Interface (MPI) performance in a scientific computing setting.
- empirical comparisons of synchronous and asynchronous variants of domain decomposition solvers on a state-of-the-art parallel computer.

Our work demonstrates that asynchronous methods have the potential of outperforming conventional synchronous solvers and offer a viable alternative in the push toward exascale.

The present work is structured as follows: In section 2, we present overlapping domain decomposition methods and explain their use in synchronous and asynchronous fashion. For a general introduction to domain decomposition methods we refer the reader to [16, 34, 33]. The section concludes with a convergence analysis of the presented one- and two-level methods. Section 3 is dedicated to a description of the presently available mechanisms in MPI and hardware to achieve truly asynchronous communication. Numerical experiments exploring asynchronous communication and using the presented domain decomposition methods are given in section 4, where we compare the strong and weak scaling behavior of synchronous and asynchronous solvers with and without load imbalance.

1.1. Related work. An asynchronous one-level domain decomposition solver with optimized artificial boundary conditions was proposed in [30]; see also [21, 20, 17] for its analysis in two different settings. An implementation of asynchronous optimized Schwarz is described in [36]. An optimization package that leverages asynchronous coordinate updates is presented in [31]. An asynchronous multigrid method for shared memory systems was proposed in [35]. Synchronization reducing Krylov methods have a long history [14]. However, preconditioning such methods is unresolved apart from some simple preconditioners [13]. Recent work extends their applicability to one-level domain decomposition preconditioning [37]. Pipelined Krylov methods [22] reduce synchronization costs by overlapping inner products with matrix-vector products and preconditioner applications and can be used with any preconditioner.

2. Domain decomposition methods.

2.1. One-level restricted additive Schwarz (RAS). We want to solve the global system

$$\boldsymbol{A}u = f,$$

where $\boldsymbol{A} \in \mathbb{R}^{N\times N}$ arises from the finite element or finite difference discretization of a partial differential equation. Informally, one-level domain decomposition solvers break up the global system of equations into overlapping subproblems that cover the whole global system. This requires that the matrix $\boldsymbol{A}$ is sparse and couples unknowns only in a local manner.

The iteration then alternates between computation of the global residual, which involves communication, and local solves for solution corrections. Special attention needs to be paid to the unknowns in the overlap in order to avoid overcorrection. Below, we describe the different methods considered in this work in detail in order to understand what data is required to be exchanged and how the methods can be executed in asynchronous fashion.

Based on the graph of $\boldsymbol{A}$ or geometric information for the underlying problem the unknowns are grouped into P overlapping sets $\mathcal{N}_p$ of size N_p, $p = 1, \ldots, P$. An

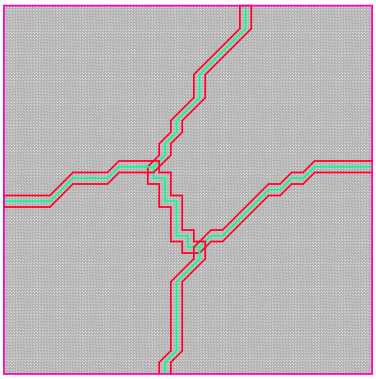

FIG. 2.1. *Partitioning of a uniform triangular mesh of the unit square into four overlapping subdomains. The nonoverlapping partitioning produced using METIS* [25] *is shown in green; the extended overlapping subdomains are shown in red.*

example of such a partitioning is given in Figure 2.1. We further split the sets $\mathcal{N}_p$ into

$$\mathcal{S}_p := \left\{ j \in \mathcal{N}_p \mid \exists k \in \mathcal{N}_p^c : \boldsymbol{A}_{jk} \neq 0 \right\},$$

i.e., unknowns that are on the boundary of the set $\mathcal{N}_p$, and interior unknowns $\mathcal{I}_p := \mathcal{N}_p \setminus \mathcal{S}_p$.

The notation throughout this section is based on Dolean, Jolivet, and Nataf [16]. We call the restriction to the pth set $\boldsymbol{R}_p \in \mathbb{R}^{N_p \times N}$. The entries of the matrices $\boldsymbol{R}_p$ are all either one or zero, with exactly one entry per row and at most one entry per column being nonzero. The local parts of $\boldsymbol{A}$ are given by

$$\boldsymbol{A}_p = \boldsymbol{R}_p \boldsymbol{A} \boldsymbol{R}_p^T \in \mathbb{R}^{N_p \times N_p}.$$

Furthermore, we require a partition of unity, represented by diagonal weighting matrices $\boldsymbol{D}_p$, such that the discrete partition of unity property holds

$$\boldsymbol{I} = \sum_{p=1}^{P} \boldsymbol{R}_p^T \boldsymbol{D}_p \boldsymbol{R}_p. \tag{2.1}$$

In what follows, we will assume that $\boldsymbol{D}_p$ are Boolean, i.e., their entries are either zero or one. This means that every (potentially shared) unknown has a special attachment with exactly one subdomain. We will furthermore require that $(\boldsymbol{D}_p)_{jj} = 0$ for all surface unknowns $j \in \mathcal{S}_p$. See Figure 2.2 for an example. One way of satisfying these restrictions is to extend overlaps starting with a *nonoverlapping* partition and then define the special attachment via the partition.

Consequently,

$$\boldsymbol{D}_p \boldsymbol{R}_p \boldsymbol{R}_q^T \boldsymbol{D}_q = \boldsymbol{0} \quad \text{for } p \neq q \tag{2.2}$$

and

$$\boldsymbol{D}_p \boldsymbol{R}_p \boldsymbol{R}_p^T \boldsymbol{D}_p = \boldsymbol{D}_p. \tag{2.3}$$

Moreover, the identity

$$\boldsymbol{R}_p \boldsymbol{A} \boldsymbol{R}_q^T \boldsymbol{D}_q = \boldsymbol{R}_p \boldsymbol{R}_q^T \boldsymbol{R}_q \boldsymbol{A} \boldsymbol{R}_q^T \boldsymbol{D}_q \tag{2.4}$$

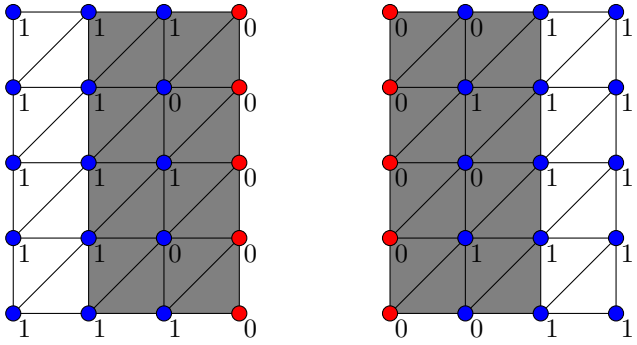

FIG. 2.2. *Two overlapping subdomains. The overlap between the subdomains is shaded in gray; the respective surface sets $\mathcal{S}_\bullet$ are shown by red circles, and the interior unknowns $\mathcal{I}_\bullet$ as blue circle. The diagonal values of the respective $\boldsymbol{D}_\bullet$ are shown next to the nodes.*

holds, since for any $u_q \in \mathbb{R}^{N_q}$, $\boldsymbol{D}_q u_q$ is supported on the interior unknowns $\mathcal{I}_p$, and hence $\boldsymbol{A}\boldsymbol{R}_q^T \boldsymbol{D}_q u_q$ is supported in $\mathcal{N}_q$. But on $\mathcal{N}_q$, $\boldsymbol{R}_q^T \boldsymbol{R}_q$ acts as the identity.

A stationary iterative method based on the splitting $\boldsymbol{A} = \boldsymbol{M} - \boldsymbol{N}$ is given globally as

$$u^{n+1} = u^n + \boldsymbol{M}^{-1}\left(f - \boldsymbol{A}u^n\right),$$

where $\boldsymbol{M}^{-1}$ is a preconditioner for $\boldsymbol{A}$.

This means that we need to calculate the residual $r^n = f - \boldsymbol{A}u^n$. Its local part on node p is given by

$$\begin{aligned}
\boldsymbol{R}_p r^n &= \boldsymbol{R}_p f - \boldsymbol{R}_p \boldsymbol{A} u^n \\
&= \boldsymbol{R}_p \left(\sum_{q=1}^{P} \boldsymbol{R}_q^T \boldsymbol{D}_q \boldsymbol{R}_q \right) f - \boldsymbol{R}_p \boldsymbol{A} \left(\sum_{q=1}^{P} \boldsymbol{R}_q^T \boldsymbol{D}_q \boldsymbol{R}_q \right) u^n \\
&= \sum_{q=1}^{P} \boldsymbol{R}_p \boldsymbol{R}_q^T \boldsymbol{D}_q \boldsymbol{R}_q f - \sum_{q=1}^{P} \boldsymbol{R}_p \boldsymbol{R}_q^T \boldsymbol{A}_q \boldsymbol{D}_q \boldsymbol{R}_q u^n \\
&= \sum_{q=1}^{P} \boldsymbol{R}_p \boldsymbol{R}_q^T \left(\boldsymbol{D}_q \boldsymbol{R}_q f - \boldsymbol{A}_q \boldsymbol{D}_q \boldsymbol{R}_q u^n \right),
\end{aligned}$$

where we used (2.1) and (2.4). This means that in order to obtain the local part of the global residual, we first compute locally $\boldsymbol{D}_p \boldsymbol{R}_p f - \boldsymbol{A}_p \boldsymbol{D}_p \boldsymbol{R}_p u^n$ on every node p, and then communicate and accumulate the overlapping parts of these local residual vectors. The latter operation is represented by the operator $\sum_{q=1}^{P} \boldsymbol{R}_p \boldsymbol{R}_q^T$.

The *restricted additive Schwarz (RAS) preconditioner* [8, 7] is given by

$$\boldsymbol{M}_{RAS}^{-1} = \sum_{p=1}^{P} \boldsymbol{R}_p^T \boldsymbol{D}_p \boldsymbol{A}_p^{-1} \boldsymbol{R}_p.$$

RAS is widely used and is the default option for overlapping domain decomposition preconditioners in PETSc [3]. It can be thought of as a variant of the additive Schwarz preconditioner

$$\boldsymbol{M}_{AS}^{-1} = \sum_{p=1}^{P} \boldsymbol{R}_p^T \boldsymbol{A}_p^{-1} \boldsymbol{R}_p$$

that is convergent as an iterative method, since the damping by $\boldsymbol{D}_p$ in the overlapping parts avoids overcorrection; see [18]. Note that for a natural choice of $\boldsymbol{D}_p$, the number of communication steps is cut in half as there is no communication associated with $\boldsymbol{R}_p^T \boldsymbol{D}_p$.

Now, the local part of the RAS iteration is given by

$$\begin{aligned}\boldsymbol{R}_p u^{n+1} &= \boldsymbol{R}_p u^n + \boldsymbol{R}_p \boldsymbol{M}_{RAS}^{-1} r^n \\ &= \boldsymbol{R}_p u^n + \sum_{q=1}^{P} \boldsymbol{R}_p \boldsymbol{R}_q^T \boldsymbol{D}_q \boldsymbol{A}_q^{-1} \boldsymbol{R}_q r^n.\end{aligned}$$

If we set $u_p^n = \boldsymbol{R}_p u^n$ and $r_p^n = \boldsymbol{R}_p r^n$ as the local parts of the solution and residual, respectively, the RAS iteration is

$$\begin{aligned} r_p^n &= \sum_{q=1}^{P} \boldsymbol{R}_p \boldsymbol{R}_q^T \left(\boldsymbol{D}_q \boldsymbol{R}_q f - \boldsymbol{A}_q \boldsymbol{D}_q u_q^n \right), \\ u_p^{n+1} &= u_p^n + \sum_{q=1}^{P} \boldsymbol{R}_p \boldsymbol{R}_q^T \boldsymbol{D}_q \boldsymbol{A}_q^{-1} r_q^n. \end{aligned}$$

This seems to suggest that the update step requires neighborhood communication as well. But in fact, in the next iteration, computation of the residual only requires $\boldsymbol{D}_p u_p^{n+1}$. From (2.2), (2.3), we see that the iterative scheme without the communication step in the update

$$r_p^n = \sum_{q=1}^{P} \boldsymbol{R}_p \boldsymbol{R}_q^T \left(\boldsymbol{D}_q \boldsymbol{R}_q f - \boldsymbol{A}_q \boldsymbol{D}_q w_q^n \right), \tag{2.5}$$

$$w_p^{n+1} = w_p^n + \boldsymbol{A}_p^{-1} r_p^n \tag{2.6}$$

is equivalent because $\boldsymbol{D}_p u_p^n = \boldsymbol{D}_p w_p^n$ for all n. The solution u_p^n can be recovered from w_p^n in the postprocessing step

$$u_p^n = \boldsymbol{R}_p u^n = \sum_{q=1}^{P} \boldsymbol{R}_p \boldsymbol{R}_q^T \boldsymbol{D}_q \boldsymbol{R}_q u^n = \sum_{q=1}^{P} \boldsymbol{R}_p \boldsymbol{R}_q^T \boldsymbol{D}_q w_q^n.$$

Finally, we use the norm of the residual in the stopping criterion. The norm can be computed from local quantities as

$$\begin{aligned}\|r^n\|^2 = r^n \cdot r^n &= r^n \cdot \left(\sum_{p=1}^{P} \boldsymbol{R}_p^T \boldsymbol{D}_p \boldsymbol{R}_p r^n \right) \\ &= \sum_{p=1}^{P} \left(\boldsymbol{R}_p r^n \right) \cdot \left(\boldsymbol{D}_p \boldsymbol{R}_p r^n \right) = \sum_{p=1}^{P} r_p^n \cdot \left(\boldsymbol{D}_p r_p^n \right).\end{aligned}$$

In conclusion, we can give the local form of RAS as in Algorithm 2.1, where we have dropped the superscript n for the iteration number. In fact, Algorithm 2.1 describes both the synchronous *and* the asynchronous version of RAS. In the synchronous version, line 4 is executed in lock step fashion by all subdomains using nonblocking two-sided communication primitives. This communication step could be overlapped by computation. However, in established frameworks such as Trilinos, such

Algorithm 2.1 Restricted additive Schwarz in local form; "↔" signifies communication.

1: $w_p \leftarrow 0$
2: **while** not converged **do**
3: Local residual: $s_p \leftarrow \boldsymbol{D}_p \boldsymbol{R}_p f - \boldsymbol{A}_p \boldsymbol{D}_p w_p$
4: Accumulate: $r_p \leftarrow \sum_{q=1}^{P} \boldsymbol{R}_p \boldsymbol{R}_q^T s_q$ ↔
5: Solve: $\boldsymbol{A}_p v_p = r_p$
6: Update: $w_p \leftarrow w_p + v_p$
7: **end while**
8: Postprocess: $u_p \leftarrow \sum_{q=1}^{P} \boldsymbol{R}_p \boldsymbol{R}_q^T \boldsymbol{D}_q w_q$ ↔

overlapping requires major changes to the framework.[1] PETSc allows some overlap of computation and communication with two-phase assembly [3]. It is possible to modify such established libraries for the asynchronous iterations of this paper. However, in order to keep the focus on algorithmic development, we developed a library that supports the one-sided communication primitives, and we build the new solvers using the communication primitives.

In the asynchronous variant, each subdomain exposes a memory region for remote access. On execution of line 4, the relevant components of the current local residual vector $s_p = \boldsymbol{D}_p \boldsymbol{R}_p f - \boldsymbol{A}_p \boldsymbol{D}_p w_p$ are written to the neighboring subdomains, and the latest locally available data s_q from every neighbor q is used. We refer to section 4.1 for a discussion of the options for actually achieving this neighborhood exchange in practice. The implementation of a convergence check (as used on line 2) that does not require synchronization is detailed in section 4.4.

2.2. Two-level synchronous RAS. In order to improve the scalability of the solver, a mechanism of global information exchange is required. Let $\boldsymbol{R}_0 \in \mathbb{R}^{n_0 \times n}$ be the restriction from the fine grid problem to a coarser mesh, and let the coarse-grid matrix $\boldsymbol{A}_0$ be given by the Galerkin relation $\boldsymbol{A}_0 = \boldsymbol{R}_0 \boldsymbol{A} \boldsymbol{R}_0^T$. The coarse-grid solve can be incorporated in the RAS iteration either in additive fashion,

$$u^{n+1} = u^n + \left(\frac{1}{2} \boldsymbol{M}_{RAS}^{-1} + \frac{1}{2} \boldsymbol{R}_0^T \boldsymbol{A}_0^{-1} \boldsymbol{R}_0 \right) (f - \boldsymbol{A} u^n), \tag{2.7}$$

or in multiplicative fashion,

$$\begin{aligned} u^{n+1/2} &= u^n + \boldsymbol{R}_0^T \boldsymbol{A}_0^{-1} \boldsymbol{R}_0 (f - \boldsymbol{A} u^n), \\ u^{n+1} &= u^{n+1/2} + \boldsymbol{M}_{RAS}^{-1} \left(f - \boldsymbol{A} u^{n+1/2} \right). \end{aligned}$$

In what follows, we focus on the additive version, since it naturally lends itself to asynchronous iterations: subdomain solves and coarse-grid solves are independent of each other.

We now determine the local form of the global algorithm. It is understood that the solve with $\boldsymbol{A}_0$ itself might be distributed over several processes. This internal computation is not meant to be performed in an asynchronous manner, which is why we do not need to further explore the local form of the coarse-grid solve. For simplicity of exposition we therefore do not describe the solution of the coarse-grid problem itself

[1] https://github.com/trilinos/Trilinos/issues/767.

Algorithm 2.2 Synchronous RAS with additive coarse grid in local form; "↔" signifies communication.

$w_p \leftarrow 0$
while not converged **do**
  On subdomains
    Local residual: $s_p \leftarrow \boldsymbol{D}_p \boldsymbol{R}_p f - \boldsymbol{A}_p \boldsymbol{D}_p w_p$
    Send $\boldsymbol{R}_0 \boldsymbol{R}_p^T s_p$ to coarse grid ↔
    Accumulate: $r_p \leftarrow \sum_{q=1}^{P} \boldsymbol{R}_p \boldsymbol{R}_q^T s_q$ ↔
    Solve: $\boldsymbol{A}_p v_p = r_p$
    Update: $w_p \leftarrow w_p + \frac{1}{2} v_p$
    Receive $c_p = \boldsymbol{R}_p \boldsymbol{R}_0^T v_0$ from coarse grid ↔
    Update: $w_p \leftarrow w_p + \frac{1}{2} c_p$
  On coarse grid
    Receive $\boldsymbol{R}_0 \boldsymbol{R}_p^T s_p$ from subdomains ↔
    Accumulate $r_0 = \sum_{p=1}^{P} \boldsymbol{R}_0 \boldsymbol{R}_p^T s_p$
    Solve $\boldsymbol{A}_0 v_0 = r_0$
    Send $c_p = \boldsymbol{R}_p \boldsymbol{R}_0^T v_0$, $p = 1, \ldots, P$ to subdomains ↔
end while
On subdomains
  Postprocess $u_p \leftarrow \sum_{q=1}^{P} \boldsymbol{R}_p \boldsymbol{R}_q^T \boldsymbol{D}_q w_q$ ↔

in local form, i.e., we will simply write $\boldsymbol{A}_0^{-1}$. The local part of the coarse-grid update is

$$\begin{aligned} &\frac{1}{2} \boldsymbol{R}_p \boldsymbol{R}_0^T \boldsymbol{A}_0^{-1} \boldsymbol{R}_0 \left(f - \boldsymbol{A} u^n\right) \\ &= \frac{1}{2} \left(\boldsymbol{R}_p \boldsymbol{R}_0^T\right) \boldsymbol{A}_0^{-1} \sum_{p=1}^{P} \left(\boldsymbol{R}_0 \boldsymbol{R}_p^T\right) \left(\boldsymbol{D}_p \boldsymbol{R}_p f - \boldsymbol{A}_p \boldsymbol{R}_p u^n\right). \end{aligned}$$

Here, the operators $(\boldsymbol{R}_0 \boldsymbol{R}_p^T)$ and $(\boldsymbol{R}_p \boldsymbol{R}_0^T)$ encode the communication from subdomain p to the coarse grid and vice versa. We notice that while the communication among subdomains consists in one neighborhood data exchange per iteration, the coarse-grid solve involves sending data from the subdomains to the coarse grid and sending a solution from the coarse grid to the subdomains. In conclusion, the local form of RAS with an additive coarse grid is given in Algorithm 2.2. Again, we have dropped the superscript for the iteration number. The communication between coarse and fine grid can be implemented in multiple ways. Since we want to allow the coarse grid solve to be distributed itself and the same coarse unknown can be owned by several coarse grid ranks (just as is the case for the fine grid), we do not consider options involving `MPI_Reduce`/`MPI_Bcast` or `MPI_Gather`/`MPI_Scatter` or their nonblocking equivalents. Instead, we opted for use of `MPI_Isend` and `MPI_Irecv`. A future improvement could involve the use of intercommunicators and `MPI_Iallgatherv` or other collectives. The advantage of the current approach is that the changes between synchronous and asynchronous implementation of the communication layer (described in the next section) are minimal.

2.3. Two-level asynchronous RAS. From the mathematical description (2.7) of two-level additive RAS, one might be tempted to see the coarse-grid problem simply as an additional subdomain. From Algorithm 2.2 the fundamental differences between

the subdomains and the coarse-grid problem become apparent. Subdomains determine the right-hand side for their local solve and correct it by transmitting boundary data to their neighbors. The coarse grid, on the other hand, receives its entire right-hand side from the subdomains, and hence it has to communicate with every single one of them.

In order to perform asynchronous coarse-grid solves, we therefore need to make sure that all the right-hand side data necessary for the solve has been received by the processes responsible for the coarse grid. Moreover, corrections sent by the coarse grid should be used exactly once by the subdomains. This is achieved by not only allocating memory regions to hold the coarse-grid right-hand side on the coarse-grid processes and the coarse-grid correction on the subdomains, but also Boolean variables that are polled to determine whether writing or reading right-hand side or solution data is permitted. More precisely, writing of the local subdomain residuals to the coarse-grid memory region of r_0 is contingent upon the state of the Boolean variable $\texttt{canWriteRHS}_p$ (see Algorithm 2.3). When $\texttt{canWriteRHS}_p$ is `True`, right-hand side data is written to the coarse grid; otherwise this operation is omitted. Here, the subscripts are used to signify the MPI rank owning the accessed memory region. As before, index 0 corresponds to the (potentially distributed) coarse grid and indices $1, \ldots, P$ correspond to the subdomains. To improve readability, we show access to a memory region on the calling process in blue, while remote access is printed in red.

In a similar fashion, the coarse grid checks whether every subdomain has written a right-hand side to r_0 by polling the state of the local Boolean array $\texttt{RHSisReady}_0$. The communication of the obtained coarse-grid solution back to the subdomains follows the same pattern, using the variables $\texttt{solutionIsReady}_p$. The subdomains update their current iterate using the local subdomain solution and the coarse-grid solution. If the latter is not available, the subdomain solution is used unweighted. If both solutions are available, then the same weighting $(1/2, 1/2)$ as in the synchronous case (2.7) is used. We note that the algorithm is asynchronous despite the data dependencies. Coarse grid and subdomain solves do not wait for each other.

We determined by experiments that overall performance is adversely affected if the coarse grid constantly polls the status variable $\texttt{RHSisReady}_0$, waiting for all subdomains to provide right-hand side information. Therefore, we added a sleep statement into its work loop. If the sleep interval is too short, the sleep statement is ineffective. If the sleep interval is too large, the coarse grid will be underused. Keeping the ratio of attempted coarse-grid solves (i.e., reads from $\texttt{RHSisReady}_0$) to actual performed coarse-grid solves at around $1/20$ has been proven effective to us. This can easily be achieved by an adaptive procedure that counts both successful solves and solve attempts and then either increases or decreases the sleep interval accordingly.

2.4. Convergence analysis of asynchronous iterations. We present below the mathematical framework used to describe and study asynchronous algorithms. We modify the model introduced by Bertsekas [5, 6] to take into account the fact that data available at a process p from another process q might have been produced during different local iterations. This issue can arise when data is accessed on process p while it is being overwritten by a new transmission from process q.

For a mathematical model of these asynchronous iterations on P processors, let us denote by $\{\sigma_n\}_{n\in\mathbb{N}}$ the sequence of nonempty subsets of $\{1, \ldots, P\}$, defining which processes update their components at the "iteration" n, where here "iteration" can be thought of as a time stamp. We call these *sets of update indices*. Define further for

Algorithm 2.3 Asynchronous RAS with additive coarse grid in local form. Variables printed in blue are exposed memory regions that are local to the calling process. Red variables are remote memory regions. Subscripts denote the owning process of the variable. Array access is denoted by "$[\cdot]$".

while not converged **do**
  On subdomains
    Local residual: $s_p \leftarrow \boldsymbol{D}_p \boldsymbol{R}_p f - \boldsymbol{A}_p \boldsymbol{D}_p w_p$
    if $\texttt{canWriteRHS}_p$ **then**
      $r_0 \leftarrow r_0 + \boldsymbol{R}_0 \boldsymbol{R}_p^T s_p$
      $\texttt{canWriteRHS}_p \leftarrow \texttt{False}$
      $\texttt{RHSisReady}_0[p] \leftarrow \texttt{True}$
    end if
    Accumulate asynchronously: $r_p \leftarrow \sum_{q=1}^{P} \boldsymbol{R}_p \boldsymbol{R}_q^T s_q$
    Solve: $\boldsymbol{A}_p v_p = r_p$
    if $\texttt{solutionIsReady}_p$ **then**
      Update: $w_p \leftarrow w_p + \frac{1}{2} v_p + \frac{1}{2} c_p$
      $\texttt{solutionIsReady}_p \leftarrow \texttt{False}$
    else
      Update: $w_p \leftarrow w_p + v_p$
    end if
  On coarse grid
    if $\texttt{RHSisReady}_0[p]$ $\forall p = 1, \ldots, P$ **then**
      Solve $\boldsymbol{A}_0 v_0 = r_0$
      for $p = 1, \ldots, P$ **do**
        $\texttt{RHSisReady}_0[p] \leftarrow \texttt{False}$
        $\texttt{canWriteRHS}_p \leftarrow \texttt{True}$
        $c_p \leftarrow \boldsymbol{R}_p \boldsymbol{R}_0^T v_0$
        $\texttt{solutionIsReady}_p \leftarrow \texttt{True}$
      end for
    else
      Sleep (time adjusted adaptively)
    end if
end while
On subdomains
  Postprocess synchronously $u_p \leftarrow \sum_{q=1}^{P} \boldsymbol{R}_p \boldsymbol{R}_q^T \boldsymbol{D}_q w_q$

$p, q \in \{1, \ldots, P\}$, $\{\tau_{q,n}^{(p)}\}_{n \in \mathbb{N}}$ a sequence of integer vectors, where $(\tau_{q,n}^{(p)})_i$, $1 \le i \le N_q$ represents the iteration number (or time stamp) of the ith component of data coming from process q and available on process p at the beginning of the computation of the process which produces $u_{p,n}$ at time n. Thus, these are the time stamps of previous computations that are used by process p, and thus, the quantities $n - \tau_{q,n,i}^{(p)}$ are sometimes called *delays*. We use the notation

$$X_p = \mathbb{R}^{N_p} \qquad \text{and} \qquad \widetilde{X} = X_1 \times \cdots \times X_P$$

to denote local and global solution spaces, and $\mathcal{T}_{p,n} : \widetilde{X} \to X_p$ the rule that is used to update the local iterate $u_{p,n}$ at iteration n. We can now define, for each process p,

the asynchronous iterations as follows:

$$u_{p,n} = \begin{cases} \mathcal{T}_{p,n}\left(u_{1,n}^{(p)}, \ldots, u_{P,n}^{(p)}\right) & \text{if } p \in \sigma_n, \\ u_{p,n-1} & \text{if } p \notin \sigma_n. \end{cases} \tag{2.8}$$

The iteration is initialized using some initial guess for $u_{p,0}$, and we used the notation $u_{q,n}^{(p)} := u_{q,\tau_q^{(p)}(n)}$ to denote the data from process q that is available to process p at time n.

In other words, at time n, either $u_{p,\bullet}$ is not updated (if $p \notin \sigma_n$) or it is updated with the result of applying the (local) operator $\mathcal{T}_{p,n}$ to the variables computed at times $\tau_\bullet^{(p)}$. For comparison, the corresponding synchronous iteration is given by

$$u_{p,n} = \mathcal{T}_{p,n}\left(u_{1,n-1}, \ldots, u_{P,n-1}\right), \tag{2.9}$$

or, in compact form, as

$$\widetilde{u}_n = \widetilde{\mathcal{T}}_n\left(\widetilde{u}_{n-1}\right), \tag{2.10}$$

where

$$\widetilde{u}_n = (u_{1,n}, \ldots, u_{P,n}) \qquad \text{and} \qquad \widetilde{\mathcal{T}}_n = (\mathcal{T}_{1,n}, \mathcal{T}_{2,n}, \ldots, \mathcal{T}_{P,n}).$$

We further assume that the three following conditions are satisfied:

$$\forall p, q \in \{1, \ldots, P\}, 1 \le i \le N_q, \forall n \in \mathbb{N}^*, \left(\tau_{q,n}^{(p)}\right)_i \le n, \tag{2.11}$$

$$\forall p \in \{1, \ldots, P\}, \operatorname{card}\{n \in \mathbb{N}^* \mid p \in \sigma_n\} = \infty, \tag{2.12}$$

$$\forall p, q \in \{1, \ldots, P\}, 1 \le i \le N_q, \lim_{n \to +\infty} \left(\tau_{q,n}^{(p)}\right)_i = \infty. \tag{2.13}$$

Condition (2.11) indicates that data used at the time n must have been produced before time n, i.e., time does not flow backward. Condition (2.12) means that no process will ever stop updating its components. Condition (2.13) corresponds to the fact that new data will always be provided to the process. In other words, no process will have a piece of data that is never updated.

We note that these assumptions pose no significant restrictions on the iterations that we consider, but are necessary for the analysis.

Assume that each X_p is a normed linear space, equipped with a norm $\|\cdot\|_p$. Given a positive vector $w \in \mathbb{R}_{>0}^P$, the weighted norm $\|\cdot\|_w$ on the product space X is defined to be

$$\|\widetilde{u}\|_w = \max_{p=1,\ldots,P} \frac{\|u_p\|_p}{w_p}.$$

We are ready to present a convergence theorem for asynchronous iterative algorithms, whose proof can be found in [18, Theorem 3.3].

THEOREM 2.1. *Assume that there exists $\widetilde{u}^* \in X$ such that $\widetilde{\mathcal{T}}_n(\widetilde{u}^*) = \widetilde{u}^*$ for all n. Moreover, assume that there exists $\gamma \in [0, 1)$ and $w \in \mathbb{R}_{>0}^P$ such that for all n we have*

$$\left\|\widetilde{\mathcal{T}}_n(\widetilde{u}) - \widetilde{u}^*\right\|_w \le \gamma \|\widetilde{u} - \widetilde{u}^*\|_w.$$

Then the asynchronous iterates $\widetilde{u}_n$ converge to $\widetilde{u}^*$, the unique common fixed point of all $\widetilde{\mathcal{T}}_n$.

In view of (2.5) and (2.6), we have

$$\mathcal{T}_{p,n}^{1L}(w_1,\dots,w_P) = w_p + \boldsymbol{A}_p^{-1}\sum_{q=1}^{P}\boldsymbol{R}_p\boldsymbol{R}_q^T\left(\boldsymbol{D}_q\boldsymbol{R}_q f - \boldsymbol{A}_q\boldsymbol{D}_q w_q\right)$$

for the one-level method. We immediately observe that the mappings $\mathcal{T}_{p,\bullet}^{1L}$ do not depend on n and that the iteration is stationary.

In order to tackle the two-level method, based on Algorithm 2.3 we set

$$\mathcal{T}_{0,n}^{2L}(v_0,w_1,\dots,w_P) = \boldsymbol{A}_0^{-1}\sum_{q=1}^{P}\boldsymbol{R}_0\boldsymbol{R}_q^T\left(\boldsymbol{D}_q\boldsymbol{R}_q f - \boldsymbol{A}_q\boldsymbol{D}_q w_q\right).$$

Moreover, for $p = 1,\dots,P$, we set

$$\mathcal{T}_{p,n}^{2L}(v_0,w_1,\dots,w_P) = w_p + \frac{1}{2}\boldsymbol{R}_p\boldsymbol{R}_0^T v_0 + \frac{1}{2}\boldsymbol{A}_p^{-1}\sum_{q=1}^{P}\boldsymbol{R}_p\boldsymbol{R}_q^T\left(\boldsymbol{D}_q\boldsymbol{R}_q f - \boldsymbol{A}_q\boldsymbol{D}_q w_q\right)$$

for iteration numbers n that include coarse-grid updates, and

$$\mathcal{T}_{p,n}^{2L}(v_0,w_1,\dots,w_P) = \mathcal{T}_{p,n}^{1L}(w_1,\dots,w_P)$$

for iterations n without coarse-grid update. Status variables such as $\texttt{canWriteRHS}_p$ act implicitly as constraints on the sets of update indices σ_n and do not appear in the definition of the mappings $\mathcal{T}_{p,n}^{2L}$.

It has been shown in [19] that both the one- and the two-level iterations are contracting in a weighted max-norm, provided that $\boldsymbol{A}$ is a nonsingular M-matrix, i.e., if $\boldsymbol{A}$ has nonpositive off-diagonal elements and all entries of $\boldsymbol{A}^{-1}$ are nonnegative.

Thus, we have the following result.

THEOREM 2.2. *The one-level method given in* (2.5) *and* (2.6) *and the two-level method given in Algorithm* 2.3 *converge, provided that* $\boldsymbol{A}$ *is a nonsingular M-matrix and that the conditions* 2.11–2.13 *hold.*

For further extensions of the theory, such as inexact subsolves with $\boldsymbol{A}_p^{-1}$ replaced by some (potentially nonstationary) $\boldsymbol{S}_{p,n} \approx \boldsymbol{A}_p^{-1}$, we refer the reader to [19].

3. One-sided Message Passing Interface. In order to drive the asynchronous method in a distributed memory setting, we use a one-sided approach wherein the remote process incurs minimal overhead for servicing received messages from the sender process. The one-sided approach is achieved in MPI using the Remote Memory Access (RMA) semantics, wherein every process exposes a part of its local memory window to remote processes for read as well as write operations. However, in reality, a synchronization between the source and the target process is required for progress of the underlying application. This active synchronization step, while still preserving the asynchronous nature of the algorithm, is expensive and might erode the natural gains obtained from the asynchronous method. Therefore in order to extract the maximum gains from an asynchronous method, a passive approach is required. A passive approach entails transmission of messages, which causes little to no interference to the target process. As a result, the target process does not need to yield its operating system time for servicing incoming message interrupts and therefore does not participate in the communication process. The RMA framework on MPI implements passive target synchronization with the help of two sets of primitives `MPI_Win_`

lock/MPI_Win_unlock and MPI_Win_lock_all/MPI_Win_unlock_all. While the former involves opening and closing the exposure epoch on remote nodes for each access operation, the latter only requires opening and closing of access epoch once during the application lifetime, incurring less target synchronization overhead.

RMA's passive one-sided communication can leverage a hardware mechanism known as Remote Direct Memory Access (RDMA) [28] when available. It allows RMA to directly map memory windows to the RDMA engine, allowing messages written by remote processes to be directly read by each process. This leads to minimum disturbance to the remote process and achieving a truly passive, one-sided communication scheme.

RDMA is usually a hardware characteristic that may not be supported by all machines. Though we expect one-sided communication of RMA to be able to handle progress of communication in an entirely asynchronous manner, it generally fails to do so since MPI does not guarantee asynchronous progress. In such a case, asynchronous progress may be enforced by allocating certain auxiliary cores to ghost processes that solely perform the task of asynchronous progress control. As a consequence we obtain an RDMA agnostic system while simultaneously obtaining the benefits of RDMA. Even in the presence of RDMA, an asynchronous progress control mechanism can be complementary since the low-level RDMA engine may not be capable of handling high volumes of communication. Casper [32] and Intel Asynchronous Progress Control (APC) are two such implementations that provide ghost processes for asynchronous progress control.

4. Implementation and numerical experiments.

4.1. Comparison of one-sided MPI communication options. There are a multitude of options for achieving asynchronous neighborhood exchange. Data that is supposed to be moved from rank p to rank q could be held in MPI windows on either p or q. In the first case, rank p will write the data to its local buffer using MPI_Put, and rank q will retrieve it from the remote buffer using MPI_Get. In the second case, rank p writes the data to the remote memory region using a MPI_Put, and q retrieves using a local MPI_Get.

The second distinction comes from the type of locking mechanism used. Exclusive or shared locks can either be applied for each individual memory access (MPI_Win_lock) or windows can be locked in shared fashion for all subsequent access (MPI_WIN_lock_all). In the latter case, windows can be flushed using any of the available flush operations.

We benchmark the different available options in a simple test case in order to determine which one should be used in the implementation of our domain decomposition solvers. The performance of one-sided MPI communication depends on the support provided by the MPI implementation as well as the network hardware. These experiments are performed on the Haswell partition of Cori at the National Energy Research Scientific Computing Center (NERSC), using the default Cray MPICH, version 7.7.3. Since one-sided MPI has not been widely adopted, performance variations compared to the classical two-sided routines can be expected to be much more significant. It should be noted that different network hardware and better support in future MPI versions could further improve timings for one-sided MPI routines.

64 MPI ranks are arranged in a three dimensional regular periodic grid (3D torus), and each rank repeatedly exchanges a vector of doubles with its 26 neighbors. This test mimics the communication pattern in the neighborhood exchange of the one-level method. For each of the possible communication options as given in Table 4.1, we

TABLE 4.1

Results of communication test described in section 4.1 *on* 64 *MPI ranks. The listed operations are either performed once per neighborhood communication phase or for each individual neighborhood exchange. If* `MPI_Win_lock_all`/`MPI_Win_unlock_all` *is used, the column "global lock" has a* ✓*. We measured the time for* 50,000 *repetitions and the fraction of neighborhood exchanges leading to incompletely written data.*

Global lock	Per comm phase	Per neighbor	Time in seconds	Inconsistency fraction
✗	`MPI_Win_lock`(EXCLUSIVE) `MPI_Win_unlock`	local `MPI_Put`, remote `MPI_Get`	34.6	0.0
✗	`MPI_Win_lock`(EXCLUSIVE) `MPI_Win_unlock`	remote `MPI_Put`, local `MPI_Get`	37.8	0.0
✗	`MPI_Win_lock`(SHARED) `MPI_Win_unlock`	local `MPI_Put`, remote `MPI_Get`	31.8	0.00151
✗	`MPI_Win_lock`(SHARED) `MPI_Win_unlock`	remote `MPI_Put`, local `MPI_Get`	33.0	0.00254
n/a	`MPI_Wait_all`	`MPI_Isend`, `MPI_Irecv`	9.59	0.0
✓	-	local `MPI_Put`, remote `MPI_Get`	25.8	0.123
✓	-	remote `MPI_Put`, local `MPI_Get`	8.42	0.00716
✓	`MPI_flush_all`	local `MPI_Put`, remote `MPI_Get`	22.1	0.117
✓	`MPI_flush_all`	remote `MPI_Put`, local `MPI_Get`	9.06	0.00491
✓	`MPI_flush_local_all`	local `MPI_Put`, remote `MPI_Get`	22.1	0.099
✓	`MPI_flush_local_all`	remote `MPI_Put`, local `MPI_Get`	9.02	0.00501
✓	`MPI_flush_local`	local `MPI_Put`, remote `MPI_Get`	24.1	0.172
✓	`MPI_flush_local`	remote `MPI_Put`, local `MPI_Get`	10.7	0.00198
✓	`MPI_flush`	local `MPI_Put`, remote `MPI_Get`	21.8	0.105
✓	`MPI_flush`	remote `MPI_Put`, local `MPI_Get`	11.2	0.00207

measure the time it takes to perform 50,000 exchanges of vectors of 500 doubles. By exchanging vectors that have a constant value corresponding to the exchange iteration, we can also measure how often inconsistent data is accessed (i.e., data that is accessed before it has been completely been transmitted). This phenomenon does not occur when using two-sided communication, since completion is guaranteed by the implementation. While the absolute number of accesses to incomplete writes is probably quite dependent on the ratio of computation to communication, we are interested in the susceptibility of the different communication options.

We make several observations. Unsurprisingly, the use of exclusive locks does not perform well in terms of time. However, the use of shared locks in every communica-

tion phase performs equally poorly, which is why we decide to use global locking and unlocking (`MPI_Win_lock_all` / `MPI_Win_unlock_all`) in what follows. Using global locking, we see that using remote puts instead of remote gets is significantly faster.

We also observe that unless exclusive locks are used, we always experience access to inconsistent data. This might not be of too much importance within our application, since it amounts to using residual information that is only slightly more outdated. Finally, we observe that using global locking and puts results in faster communication than classical two-sided nonblocking communication.

Based on the above results, we choose to use global locking using `MPI_Win_lock_all` / `MPI_Win_unlock_all`, paired with remote `MPI_Put`s and local `MPI_Get`s and `MPI_flush_all`, since it appears to provide a good balance of speed and consistency. We note, however, that these results might depend significantly on characteristics of the system and the MPI implementation.

4.2. Performance metrics. The average contraction factor per iteration is defined as $\widetilde{\rho} = (r_{\text{final}}/r_0)^{\frac{1}{K}}$, where r_0 is the norm of the initial residual vector, r_{final} is the norm of the final residual vector, and K is the number of iterations that were taken to decrease the residual from r_0 to r_{final}. For an asynchronous method, the number of iterations varies from subdomain to subdomain, and hence $\widetilde{\rho}$ is not well-defined. The following generalization permits us to compare synchronous methods with their asynchronous counterpart:

$$\widehat{\rho} = \left(\frac{r_{\text{final}}}{r_0}\right)^{\frac{\tau_{\text{sync}}}{T}}.$$

Here, T is the total iteration time, and τ_{sync} is the average time for a single iteration of the synchronous method. In the synchronous case, since $T = \tau_{\text{sync}} K$, $\widehat{\rho}$ recovers $\widetilde{\rho}$. The approximate contraction factor $\widehat{\rho}$ can be interpreted as the average contraction of the residual norm in the time of a single synchronous iteration. As will be visible in the results to follow, we note here that $\widehat{\rho}$ for the asynchronous method obviously depends on the total iteration time for the synchronous method. Assume that the total iteration time for the synchronous method doubles, but the time taken by the asynchronous one stays constant. Consequently, the approximate contraction factor for the synchronous method stays constant, but the contraction factor for the asynchronous method gets squared and therefore decreases.

4.3. Test problem. As a test problem, we solve

$$-\Delta u = f \qquad \text{in } \Omega = [0,1]^d, \qquad u = 0 \qquad \text{on } \partial\Omega,$$

where the right-hand side is $f = d\pi^2 \prod_{k=1}^{d} \sin(\pi x_k)$. The corresponding solution is $u = \prod_{k=1}^{d} \sin(\pi x)$. We discretize Ω using a uniform simplicial mesh and approximate the solution using piecewise linear finite elements. We note that the arising system matrix $\boldsymbol{A}$ is a nonsingular M-matrix, and therefore Theorem 2.2 applies. Furthermore, we mention that the generalization of the test problem to convection-diffusion problems with nonconstant diffusion coefficient is possible, but does not alter the numerical results obtained below in a significant way, which is why we only present the case of the standard Poisson problem.

4.4. Convergence detection. In classical synchronous iterative methods, a stopping criterion of the form $\|r\| < \varepsilon$ is evaluated at every iteration. Here, r is the residual vector, ε is a prescribed tolerance (that might be chosen as a function of the

discretization error), and $\|\cdot\|$ is an appropriate norm. The global quantity $\|r\|$ needs to be computed as the sum of local contributions from all the subdomains. This implies that convergence detection in asynchronous methods is not straightforward, since collective communication primitives require synchronization. In the numerical examples below, we use a simplistic convergence criterion, consisting of writing the local contributions to a master rank, say, rank 0. This master rank sums the contributions and determines if this approximation of the global residual norm is smaller than the prescribed tolerance. If so, the master rank declares global convergence and notifies the other ranks by sending a nonblocking message. This simplistic convergence detection mechanism has several drawbacks. For one, the global residual is updated by the master rank, which might not happen frequently enough. Hence it is possible that the iteration continues despite the true global residual norm already being smaller than the tolerance. Moreover, the mechanism puts an increased load on the network connection to the master rank, since every subdomain writes to its memory region. Finally, since the local contributions to the residual norm are not necessarily monotonically decreasing, the criterion might actually detect convergence when the true global residual is not yet smaller than the tolerance. The delicate topic of asynchronous convergence detection has been treated in much detail in the literature, and we refer to [2, 29] for an overview of more elaborate approaches. While these detection schemes mostly address the shortcomings of the above approach, their correct implementation turns out to be quite involved. Since we are not observing any major issues with our simplistic convergence detection scheme for the test problems that we consider, we have not implemented any of the schemes available in the literature.

4.5. Platform and implementation details. All runs are performed on the Haswell partition of the Cori supercomputer at NERSC. While all of the code was written from scratch, the differences between the synchronous and the asynchronous code path are limited, since only the communication layer and the stopping criterion need to be changed (e.g., compare Algorithms 2.2 and 2.3). We stress that the asynchronous solver uses one-sided communication only in the solve phase. Therefore, we record solve times only, since the time to set up the solver is unaffected by the type of communication in the solution phase. Furthermore, all subdomains are synchronized via an `MPI_Barrier` before entering the solve phase. One MPI rank is used per core, i.e., 32 ranks per Haswell node. Moreover, one subdomain is assigned to each MPI rank. The underlying mesh is partitioned either into uniformly sized rectangular subdomains or using the METIS library [25]. In the latter case, the option to minimize the overall communication volume is used. Our solvers handle general unstructured matrices, and the structure of the mesh is not exploited. We either use

- direct solvers for subdomain and coarse-grid problems, provided by SuperLU [27, 15], or
- conjugate gradient method preconditioned with an incomplete Cholesky factorization for the subdomain problems and a geometric multigrid solver for the coarse problem.

The latter option would allow for a distributed coarse-grid solve and is therefore in principle more scalable. In all numerical examples, we will use only a single core for the coarse-grid solve.

4.6. Comparison against HPDDM. We verify the performance of the synchronous version of our code against the HPDDM library [24, 23, 16] using the 2D and 3D test problems from section 4.3. In all cases, we set up a GMRES solver and use a two-level additive RAS as right preconditioner. The reason for using a Krylov method

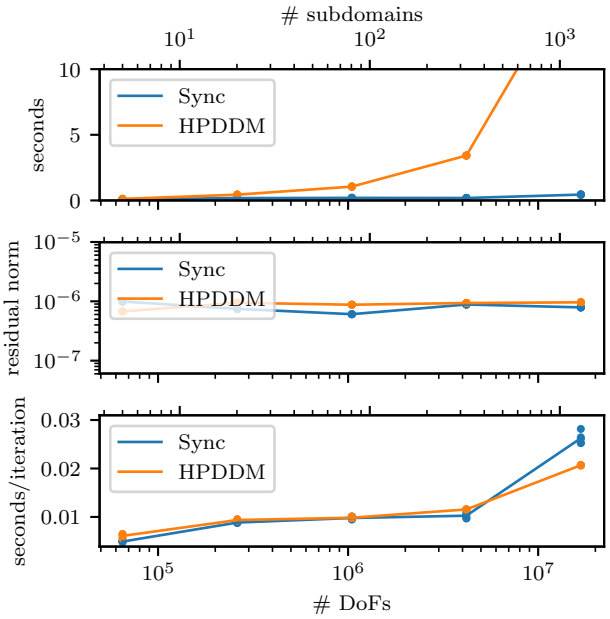

FIG. 4.1. *Weak scaling of GMRES preconditioned by two-level additive RAS using the synchronous version of our code (Sync) and HPDDM for the 2D test problem in a load balanced case. From top to bottom: total solution time, final residual norm, and time per iteration. Mean values are given by solid lines, and individual runs as dots.*

here is that domain decomposition methods in general are commonly used as preconditioners, and HPDDM is most likely developed with that use case in mind. HPDDM was linked against the Intel Math Kernel Library, SuiteSparse [12], and ARPACK [26] and the option for coarse grid data exchange using `MPI_Igather/MPI_Iscatter` was enabled. We use the following parameters in HPDDM: `-hpddm_krylov_method=gmres -hpddm_schwarz_method=ras -hpddm_schwarz_coarse_correction=additive -hpddm_geneo_nu=NU`, where `NU` is chosen so that the size of the coarse grid matches our solver. In two dimensions the subdomains consist of roughly 20,000 unknowns, and the coarse grid contains about 16 unknowns per subdomain. In three dimensions the subdomains consist of roughly 40,000 unknowns, and the coarse grid contains about 1 unknown per subdomain. In Figures 4.1 and 4.2 we plot the results of weak scaling experiments: overall solve time, the reached residual norm, and the time per iteration. We repeated each run 5 times. Mean values are given by solid lines, and individual runs as dots. We observe that while the time to convergence behaves quite differently for both implementations, the time per iteration follows the same trend. HPDDM behaves slightly better at large subdomain count, which could be explained by the use of MPI collectives, but might also be an artifact of the difference in convergence behavior or the difference in coarse solvers (HPDDM uses Cholmod). We can therefore use our synchronous method as a base of comparison for the newly developed asynchronous solver.

4.7. One-level RAS, 2D test problem, strong scaling. We compare synchronous and asynchronous one-level RAS in a strong scaling experiment, where we fix the global problem size of a 2D test problem to about 261,000 unknowns and vary the number of subdomains between 4 and 256. We cannot expect good scaling behav-

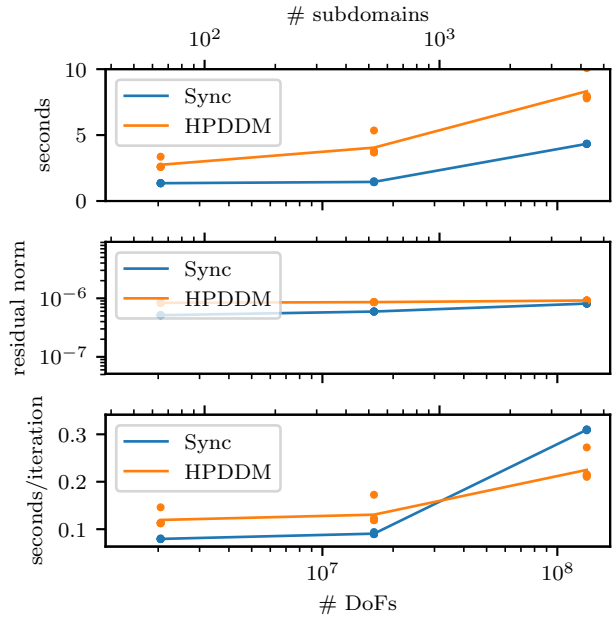

FIG. 4.2. *Weak scaling of GMRES preconditioned by two-level additive RAS using the synchronous version of our code (Sync) and HPDDM for the 3D test problem in a load balanced case. From top to bottom: total solution time, final residual norm, and time per iteration. Mean values are given by solid lines, and individual runs as dots.*

ior for this one-level method, since increasing the number of subdomains adversely affects the rate of convergence. The iteration is terminated based on the simplistic convergence criterion described in section 4.4. In Figure 4.3 we show solve time, final residual norm, and approximate rate of convergence. It can be observed that the synchronous method is faster for smaller numbers of subdomains, yet comparatively slower for larger number of subdomains. The crossover point between the two regimes appears to be at 64 subdomains.

An important question is whether the asynchronous method happens to converge because every subdomain performs the same number of local iterations, and hence the asynchronous method just mirrors the synchronous one, merely with a different communication method. The histogram in Figure 4.4 shows that this is not the case. The number of local iterations varies significantly. The slowest subdomain performs barely more than 11,000 iterations, whereas the fastest one almost reaches 16,000. The problem was load balanced by the number of degrees of freedom in each subdomain, and thus the local solves are also approximately balanced but the communication is likely slightly imbalanced. This means that in this scenario, system and network noise are the main contributions to the observed variations in local iteration counts. For comparison, when only 4 subdomains were used, the local iteration counts were 1497, 1500, 1504, and 1527.

The advantage of asynchronous RAS becomes even clearer when the experiment is repeated under load imbalance. We create an artificial load imbalance by choosing one of the subdomains to be 50% larger than the rest. In Figure 4.5 it is observed that the asynchronous method outperforms the synchronous one in all but the 4 subdomain case.

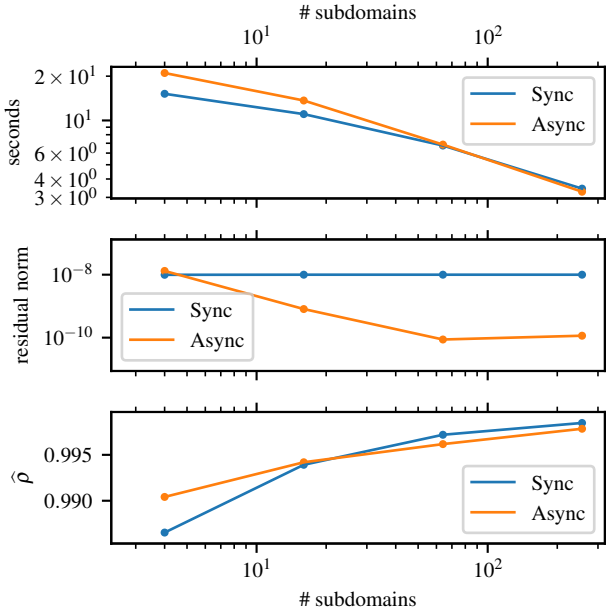

FIG. 4.3. *Strong scaling of synchronous and asynchronous one-level RAS for the 2D test problem with system size of approximately* 261,000 *unknowns. The subdomains are load balanced. From top to bottom: solution time, final residual norm, and the resulting approximate contraction factor* $\widehat{\rho}$*. It can be observed that the synchronous method is significantly faster than for smaller numbers of subdomains (cores), yet comparatively slower for larger number of subdomains, as shown by the contraction factor.*

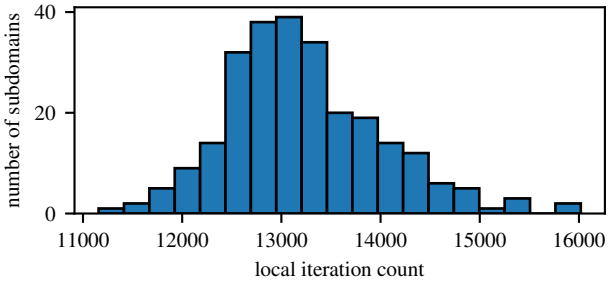

FIG. 4.4. *Histogram of local iteration counts for asynchronous one-level RAS for the 2D test problem with* 256 *subdomains (load balanced case).*

4.8. Two-level RAS, 2D test problem. In order to gauge the performance and scalability of the synchronous and asynchronous two-level RAS solvers, we perform weak and strong scaling experiments.

4.8.1. Weak scaling. In the weak scaling experiment the number of subdomains P and the global number of degrees of freedom (DoFs) are increased proportionally. We use 16, 64, 256, and 1024 subdomains to solve the 2D test problem. The local number of unknowns on each subdomain is kept constant at almost 20,000. The coarse-grid problem increases in size proportionally to the number of subdomains,

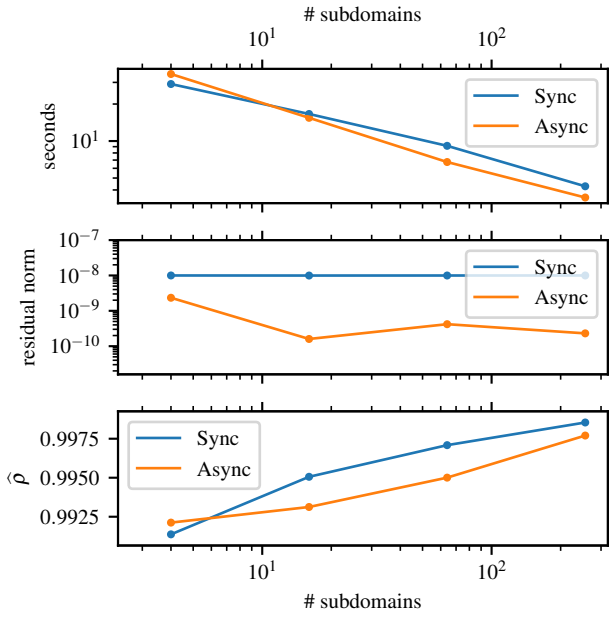

FIG. 4.5. *Strong scaling of synchronous and asynchronous one-level RAS for the* 2*D test problem with a system size of approximately* 261,000 *unknowns under load imbalance: one subdomain is* 50% *larger than the rest. From top to bottom: solution time, final residual norm, and approximate contraction factor* $\widehat{\rho}$*. It can be observed that the asynchronous method outperforms the synchronous one in all but the* 4 *subdomain case, as shown by the contraction factor. The advantage of the asynchronous method over the synchronous one is increased, as compared to Figure* 4.3.

with approximately 16 unknowns per subdomain. Again, the iteration is terminated based on the simplistic convergence criterion described in section 4.4.

In Figure 4.6 we plot the solution time, the achieved residual norm and the average contraction factor $\widehat{\rho}$ depending on the global problem size. Both the synchronous and the asynchronous method reach the prescribed tolerance of 10^{-8}. Due to the lack of an efficient mechanism of convergence detection, the asynchronous method ends up iterating longer than necessary, so that the final residual norm often is smaller than 10^{-9}. The number of iterations in the synchronous case is about 110, whereas the number of local iterations in the asynchronous case varies between 110 and 150 (see Figure 4.7). The iteration counts are significantly lower than for the one-level methods. One can observe that for 16, 64, and 256 subdomains, the asynchronous and the synchronous methods take almost the same time for the solve. For 1024 subdomains, however, the synchronous method is seen to take significantly more time. This can be explained by the fact that for 1024 subdomains, the size of the coarse grid is comparable to the size of the subdomains, and hence the coarse-grid solve which exchanges information with all the subdomains slows down the overall progress. For the asynchronous case this is not observed, since the subdomains do not have to wait for information from the coarse grid. This explains why we see better weak scalability for the asynchronous method than for the synchronous variant, and why we can observe a speedup of two times the asynchronous method over its synchronous counterpart. The third subplot of Figure 4.6 shows that the asynchronous method outperforms its synchronous equivalent in all but the smallest problem.

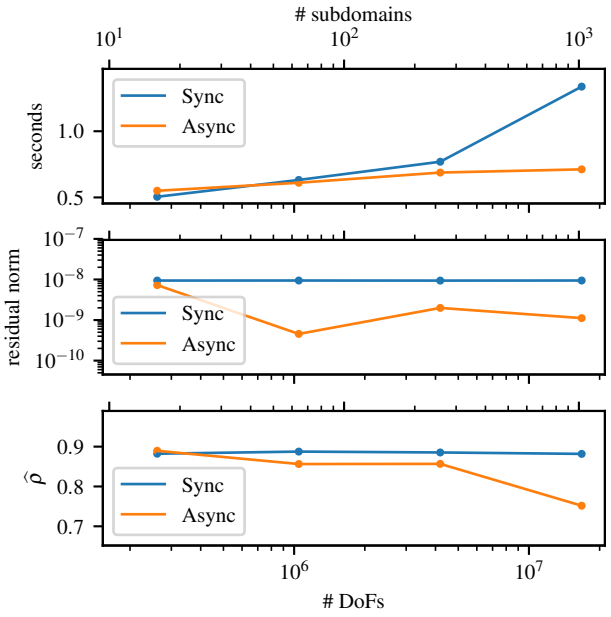

FIG. 4.6. *Weak scaling of synchronous and asynchronous two-level additive RAS for the* 2*D test problem, load balanced case. From top to bottom: total solution time, final residual norm, and approximate contraction factor* $\widehat{\rho}$*. One can observe that for* 16*,* 64*, and* 256 *subdomains, the asynchronous and the synchronous method take almost the same time for the solve, with a slight advantage for the asynchronous method. For* 1024 *subdomains, however, the synchronous method is seen to take significantly more time, since the coarse grid, due to its size, starts to be the limiting factor. The asynchronous method is not affected by this.*

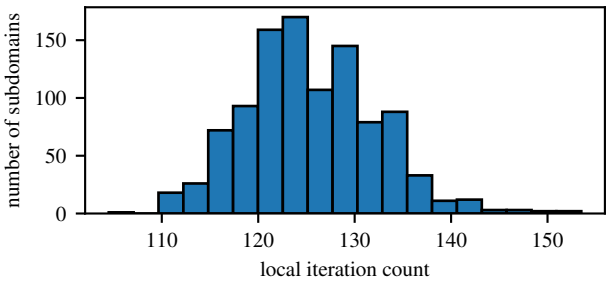

FIG. 4.7. *Histogram of local iteration counts asynchronous two-level additive RAS for the* 2*D test problem with* 1024 *subdomains.*

To further illustrate the effect of load imbalance, we repeat the previous experiment with one subdomain being 50% larger than the rest. The results are shown in Figure 4.8. While the results are mostly consistent with the previous case, it can be seen that, as expected, the performance advantage of the asynchronous method over the synchronous one has increased. Even before the size of the coarse-grid system is comparable to the size of the typical subdomain problem, the asynchronous method outperforms its synchronous counterpart.

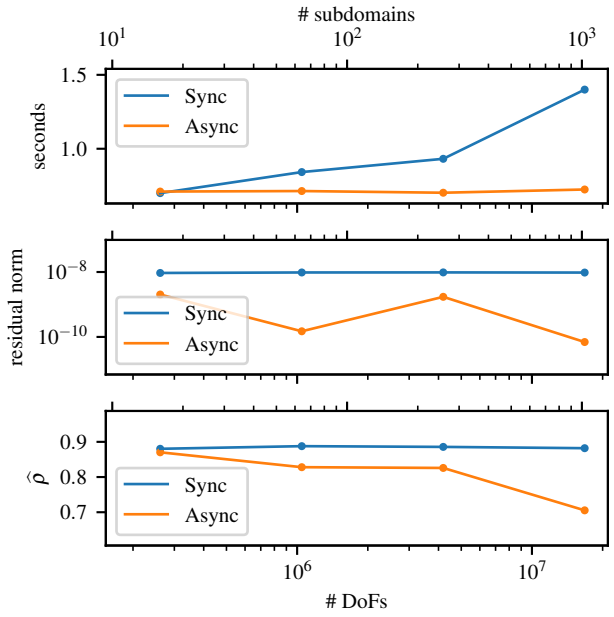

FIG. 4.8. *Weak scaling of synchronous and asynchronous two-level additive RAS for the 2D test problem under load imbalance: one subdomain is 50% larger than all the other ones. From top to bottom: total solution time, final residual norm, and approximate contraction factor* $\widehat{\rho}$. *The advantage of the asynchronous method over the synchronous one is increased, as compared to Figure* 4.6.

4.8.2. Strong scaling. For the strong scaling experiment the global number of degrees of freedom used to discretize the 2D test problem is fixed at about 4 million. The coarse-grid problem consists of approximately 4,000 unknowns. The number of subdomains used on the fine level takes values in $\{4, 16, 64, 256\}$. This means that the coarse-grid problem is always smaller than the typical subdomain problem, and no slowdown due to an imbalance of the computational cost of coarse and fine solve should arise.

The timing results are shown in the top of Figure 4.9. Both the synchronous and asynchronous methods display good strong scaling behavior. It is observed that the synchronous method is faster than the asynchronous method for a smaller subdomain count. But already for 64 subdomains this behavior is reversed, and the asynchronous method outperforms the synchronous one. This suggests that synchronization is an important factor already at a modest core count.

At the bottom of Figure 4.9, we show the timing results in the case of load imbalance. It can be seen that the asynchronous method is faster than the synchronous one independent of the number of subdomains and that its performance advantage increases as more processes are used.

4.9. Two-level RAS with iterative subsolves, 3D test problem. The density of the subdomain matrices $\boldsymbol{A}_p$ in three dimensions (about 15 entries per row) is higher than for the 2D test problem (about 7 entries per row). This means that direct factorization leads to more fill-in and thereby is more expensive. Therefore, we solve the subdomain and coarse problem of the three dimensional test case using iterative

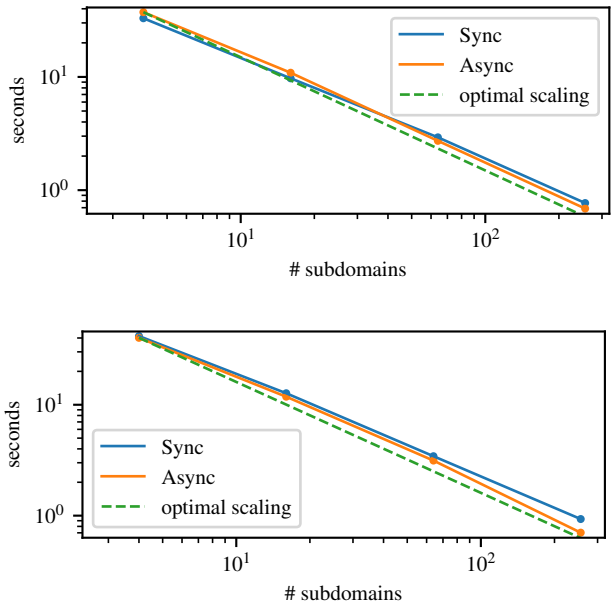

FIG. 4.9. *Strong scaling of synchronous and asynchronous two-level additive RAS for the* 2*D test problem. On top: load balanced subdomains. At the bottom: load imbalance; one subdomain is* 50% *larger than the others.*

solvers. For the subdomains, we use a conjugate gradient solver preconditioned by an incomplete Cholesky factorization. We employ a relative tolerance of 1/10, which has been determined experimentally to be sufficient. The coarse-grid problem is solved using a single V-cycle of a geometric multigrid solver with one step of Gauss–Seidel for pre- and post-smoothing. The use of multigrid allows us to solve the coarse-grid problem in a distributed fashion when it becomes too large for a single MPI rank. The global problem is partitioned into uniformly sized regular subdomains. The local number of unknowns on each subdomain is kept constant at about 40,000. The coarse-grid problem increases in size proportionally to the number of subdomains, with approximately one unknown per subdomain.

The results of a weak scaling experiment are shown in Figure 4.10. We observe behavior that is similar to the 2D case. We notice, however, that the size of the coarse-grid problem (and hence the solution of the coarse-grid problem) are not the issue here. At 4096 ranks, the coarse-grid problem is an order of magnitude smaller than the typical subdomain problem. The apparent slowdown of the synchronous method is caused by the cost of exchanging information between the coarse grid and the subdomains. While a slowdown is also visible in the asynchronous method, it is much less pronounced, resulting in a speedup of four times over the synchronous method.

We also observe that compared to the 2D case where direct solvers were used, both synchronous and asynchronous iterations terminate almost exactly once the prescribed tolerance has been achieved. The reason for this is that convergence checks occur much more frequently as the tolerance is reached, since the iterative subsolves converge to their local tolerance typically within one iteration.

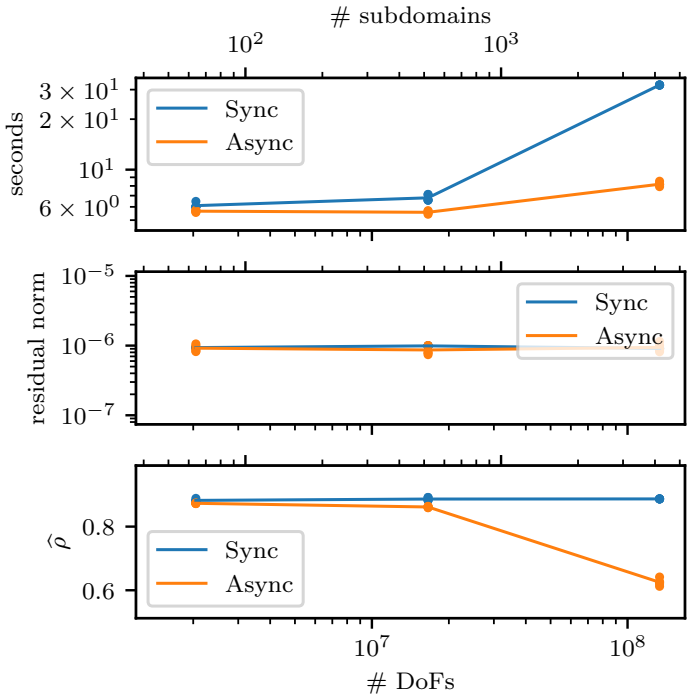

FIG. 4.10. *Weak scaling of synchronous and asynchronous two-level additive RAS, load balanced 3D case. From top to bottom: total solution time, final residual norm, and approximate contraction factor $\widehat{\rho}$. One can observe that for 64 and 512 subdomains, the asynchronous and the synchronous method take almost the same time for the solve, with a slight advantage for the asynchronous method. For 4096 subdomains, however, the synchronous method is seen to take significantly more time. The reason for this is not the solution of the coarse-grid problem, as in two dimensions, but the cost of the data exchange. The effect on the asynchronous method is much less pronounced.*

5. Conclusion. In the present work, we have explored the use of asynchronous alternatives to conventional (synchronous) one-level and two-level domain decomposition solvers. To the best of our knowledge, we proposed the first truly asynchronous two-level method, where each processor can do a different number of updates (iterations). Several options to achieve asynchronous communication were tested, and we found that our use case benefited most from using `MPI_Win_lock_all` / `MPI_Win_unlock_all`, remote `MPI_Puts` and local `MPI_Gets`. The numerical results presented demonstrate that asynchronous iterations can be considered a viable alternative to synchronous methods, despite partial availability of information from neighbors. Asynchronous methods seem to be beneficial already at a modest core count, even for load balanced scenarios. In the presence of load imbalance, their performance advantage becomes even clearer, and we observed speedups up to four times. While we focused our attention on a particular Schwarz method, it is of inherent interest to explore asynchronous variants of other, potentially more effective domain decomposition methods involving deflation or nonoverlapping decompositions (such as FETI and BDDC) or more than two levels. The presented inclusion of a novel asynchronous coarse-grid correction paves the way for asynchronous methods to be used in extremely scalable parallel solvers.

Acknowledgment. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.
SAND Number: SAND2020-8220 J.

REFERENCES

[1] V. Ahlgren, S. Andersson, J. M. Brandt, N. P. Cardo, S. Chunduri, J. Enos, P. Fields, A. Gentile, R. Gerber, J. Greenseid, A. Greiner, B. Hadri, Y. He, D. Hoppe, U. Kaila, K. Kelly, M. Klein, A. Kristiansen, S. Leak, M. Mason, K. Pedretti, J.-G. Piccinali, J. Repik, J. Rogers, S. Salminen, M. Showerman, C. Whitney, and J. Williams, *Cray System Monitoring: Successes Requirements and Priorities*, Technical report, Sandia National Lab., Albuquerque, NM, 2018.

[2] J. M. Bahi, S. Contassot-Vivier, R. Couturier, and F. Vernier, *A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms*, IEEE Trans. Parallel Distrib. Syst., 16 (2005), pp. 4–13.

[3] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, *PETSc Users Manual*, Technical report ANL-95/11—Revision 3.13, Argonne National Laboratory, 2020, https://www.mcs.anl.gov/petsc.

[4] G. M. Baudet, *Asynchronous iterative methods for multiprocessors*, J. ACM, 25 (1978), pp. 226–244.

[5] D. P. Bertsekas, *Distributed asynchronous computation of fixed points*, Math. Program., 27 (1983), pp. 107–120.

[6] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and distributed computation: Numerical methods*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[7] X.-C. Cai, M. Dryja, and M. Sarkis, *Restricted additive Schwarz preconditioners with harmonic overlap for symmetric positive definite linear systems*, SIAM J. Numer. Anal., 41 (2003), pp. 1209–1231.

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

[9] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, *Toward exascale resilience*, Int. J. High Perform. Comput. Appl., 23 (2009), pp. 374–388, https://doi.org/10.1177/1094342009347767.

[10] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, *Toward exascale resilience: 2014 update*, Supercomput. Front. Innov., 1 (2014), pp. 5–28, https://doi.org/10.14529/jsfi140101.

[11] D. Chazan and W. Miranker, *Chaotic relaxation*, Linear Algebra Appl., 2 (1969), pp. 199–222.

[12] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, *Algorithm* 887: *Cholmod, supernodal sparse Cholesky factorization and update/downdate*, ACM Trans. Math. Softw., 35 (2008), https://doi.org/10.1145/1391989.1391995.

[13] A. T. Chronopoulos and C. W. Gear, *On the efficient implementation of preconditioned s-step conjugate gradient methods on multiprocessors with memory hierarchy*, Parallel Comput., 11 (1989), pp. 37–53.

[14] A. T. Chronopoulos and C. W. Gear, *s-step iterative methods for symmetric linear systems*, J. Comput. Appl. Math., 25 (1989), pp. 153–168, https://.doi.org/10.1016/0377-0427(89)90045-9.

[15] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu, *A supernodal approach to sparse partial pivoting*, SIAM J. Matrix Anal. Appl., 20 (1999), pp. 720–755.

[16] V. Dolean, P. Jolivet, and F. Nataf, *An Introduction to Domain Decomposition Methods: Algorithms, Theory, and Parallel Implementation*, SIAM, Philadelphia, 2015, https://doi.org/10.1137/1.9781611974065.ch1.

[17] M. El Haddad, J. C. Garay, F. Magoulès, and D. B. Szyld, *Synchronous and asynchronous optimized Schwarz methods for one-way subdivision of bounded domains*, Numerical Linear Algebra Appl., 27 (2020), e2279.

[18] A. Frommer and D. B. Szyld, *On asynchronous iterations*, J. Comput. Appl. Math., 123 (2000), pp. 201–216.

[19] A. Frommer and D. B. Szyld, *An algebraic convergence theory for restricted additive Schwarz methods using weighted max norms*, SIAM J. Numer. Anal., 39 (2001), pp. 463–479.

[20] J. C. Garay, F. Magoulès, and D. B. Szyld, *Synchronous and asynchronous optimized Schwarz method for Poisson's equation in rectangular domains*, Technical report 17-10-18, Department of Mathematics, Temple University, Philadelphia, PA, 2018.

[21] J. C. Garay, F. Magoulès, and D. B. Szyld, *Convergence of asynchronous optimized Schwarz methods in the plane*, in Proceedings of Domain Decomposition Methods in Science and Engineering XXIV, P. E. Bjørstad, S. C. Brenner, L. Halpern, H. H. Kim, R. Kornhuber, T. Rahman, and O. B. Widlund, eds., Lecture Notes Comput. Sci. Eng., 2018, Springer, Berlin, pp. 333–341.

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

[23] P. Jolivet, F. Hecht, F. Nataf, and C. Prudhomme, *Scalable domain decomposition preconditioners for heterogeneous elliptic problems*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 13, Association for Computing Machinery, New York, 2013, https://doi.org/10.1145/2503210.2503212.

[24] P. Jolivet and F. Nataf, *HPDDM—High-Performance Unified Framework for Domain Decomposition Methods*, 2020, https://github.com/hpddm/hpddm.

[25] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1998), pp. 359–392, https://doi.org/10.1137/S1064827595287997.

[26] R. B. Lehoucq, D. C. Sorensen, and C. Yang, *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*, SIAM, Philadelphia, 1998.

[27] X. Li, J. Demmel, J. Gilbert, L. Grigori, M. Shao, and I. Yamazaki, *SuperLU Users' Guide*, Technical report LBNL-44289, Lawrence Berkeley National Laboratory, 1999, https://portal.nersc.gov/project/sparse/superlu/.

[28] J. Liu, J. Wu, and D. K. Panda, *High performance RDMA-based MPI implementation over InfiniBand*, Internat. J. Parallel Program., 32 (2004), pp. 167–198.

[29] F. Magoulès and G. Gbikpi-Benissan, *Distributed convergence detection based on global residual error under asynchronous iterations*, IEEE Trans. Parallel Distrib. Syst., 29 (2018), pp. 830–842.

[30] F. Magoulès, D. B. Szyld, and C. Venet, *Asynchronous optimized Schwarz methods with and without overlap*, Numer. Math., 137 (2017), pp. 199–227, https://doi.org/10.1007/s00211-017-0872-z.

[31] Z. Peng, Y. Xu, M. Yan, and W. Yin, *ARock: An Algorithmic Framework for Asynchronous Parallel Coordinate Updates*, SIAM J. Sci. Comput., 38 (2016), pp. A2851–A2879, https://doi.org/10.1137/15M1024950.

[32] M. Si, A. J. Pena, J. Hammond, P. Balaji, M. Takagi, and 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) IEEE, 2015, pp. 665–676.

[33] B. Smith, P. Bjorstad, and W. Gropp, *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*, Cambridge University Press, Cambridge, 2004.

[34] A. Toselli and O. Widlund, *Domain Decomposition Methods: Algorithms and Theory*, Springer Ser. Computer Math. 34, Springer, Berlin, 2006.

[35] J. Wolfson-Pou and E. Chow, *Asynchronous multigrid methods*, in Proceedings of the 33rd IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2019, pp. 101–110.

[36] I. Yamazaki, E. Chow, A. Bouteiller, and J. Dongarra, *Performance of Asynchronous Optimized Schwarz with One-sided Communication*, Parallel Comput., 86 (2019), pp. 66–81.

[37] I. Yamazaki, S. Rajamanickam, E. G. Boman, M. Hoemmen, M. A. Heroux, and S. Tomov, *Domain decomposition preconditioners for communication-avoiding Krylov methods on a hybrid CPU/GPU cluster*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Press, Piscataway, NJ, 2014, pp. 933–944.