Asynchronous One-Level and Two-Level Domain Decomposition Solvers

Christian Glusa, Erik G. Boman, Edmond Chow, Sivasankaran Rajamanickam, and Paritosh Ramanan

1 Introduction

Multilevel methods such as multigrid and domain decomposition are among the most efficient and scalable solvers developed to date. Adapting them to the next generation of supercomputers and improving their performance and scalability is crucial for exascale computing and beyond. Domain decomposition methods subdivide the global problem into subdomains, and then alternate between local solves and boundary data exchange. This puts 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 under-utilized. The use of non-blocking communication can only alleviate this issue, but not solve it. In asynchronous methods, on the other hand, computation and communication occur concurrently, with some processes performing computation while others communicate, so that the network is consistently in use.

Unfortunately, the term "asynchronous" can have several different meanings in the literature. In computer science, it is sometimes used to describe communication patterns that are non-blocking, such that computation and communication can be overlapped. Iterative algorithms that use such "asynchronous" communication typically still yield the same iterates (results), just more efficiently. 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

Christian Glusa, Erik G. Boman, Sivasankaran Rajamanickam
Center for Computing Research, Sandia National Laboratories, Albuquerque, New Mexico 87185, USA, e-mail: \{caglusa,egboman,srajama\}@sandia.gov

Edmond Chow, Paritosh Ramanan
School of Computational Science and Engineering, College of Computing, Georgia Institute of Technology, Atlanta, Georgia, USA e-mail: paritoshpr@gatech.edu,echow@cc.gatech.edu

relaxation" [6]. Both approaches are expected to play an important role on future supercomputers. In this paper, we focus on the mathematically asynchronous methods.

Domain decomposition solvers [8, 16, 15] 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. We note, however, that the discussed algorithms could be coupled with existing pipelined methods [10] 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, or due to local, problem specific causes, such as iteration counts for local sub-solves that vary from region to region. Especially the latter are difficult to predict, so that load balancing cannot occur before the actual solve. Therefore, a synchronous parallel application has to be idle until its slowest process has finished. In an asynchronous method, local computation can continue, and improve the quality of the global solution. An added benefit of asynchronous methods is that, since the interdependence of one subdomain on the others has been weakened, fault tolerance [4, 5] can be more easily achieved.

The main drawback of asynchronous iterations is the fact that deterministic behavior is sacrificed. Consecutive runs do not produce the same result. (The results do match up to a factor proportional to the convergence tolerance.) This also makes the mathematical analysis of asynchronous methods significantly more difficult than the analysis of their synchronous counterparts. Analytical frameworks for asynchronous linear iterations have long been available [6, 2, 3, 9], but generally cannot produce sharp convergence bounds except for in the simplest of cases.

2 Domain decomposition methods

We want to solve the system $\boldsymbol{A}\mathbf{u} = \mathbf{f}$, where $\boldsymbol{A} \in \mathbb{R}^{N\times N}$. Informally speaking, one-level domain decomposition solvers break up the global system into overlapping subproblems that cover the global system. The iteration alternates between computation of the global residual, involving communication, and local solves for corrections. Special attention is paid to unknowns in the overlap to avoid over-correction.

We use the notation of [8] and denote subdomain matrices by $\boldsymbol{A}_p$, restrictions by $\boldsymbol{R}_p$, and the discrete partition of unity by $\boldsymbol{D}_p$. The local form of the restricted additive Schwarz iteration (RAS) is given in Figure 1. A detailed derivation of the algorithm can be found in [11]. In fact, Figure 1 describes both the synchronous *and* the asynchronous version of RAS. In the synchronous version Line 4 is executed in lock step by all subdomains using non-blocking two-sided communication primitives. In

1: $\mathbf{w}_p \leftarrow \mathbf{0}$
2: **while** not converged **do**
3: Local residual: $\mathbf{t}_p \leftarrow \boldsymbol{D}_p \boldsymbol{R}_p \mathbf{f} - \boldsymbol{A}_p \boldsymbol{D}_p \mathbf{w}_p$
4: Accumulate: $\mathbf{r}_p \leftarrow \sum_{q=1}^{P} \boldsymbol{R}_p \boldsymbol{R}_q^T \mathbf{t}_q$
5: Solve: $\boldsymbol{A}_p \mathbf{v}_p = \mathbf{r}_p$
6: Update: $\mathbf{w}_p \leftarrow \mathbf{w}_p + \mathbf{v}_p$
7: **end while**
8: Post-process: $\mathbf{u}_p \leftarrow \sum_{q=1}^{P} \boldsymbol{R}_p \boldsymbol{R}_q^T \boldsymbol{D}_q \mathbf{w}_q$

Fig. 1: Restricted additive Schwarz (RAS) in local form. $\boldsymbol{A}_p$ are subdomain matrices, $\boldsymbol{R}_p$ are subdomain restrictions, $\boldsymbol{D}_p$ are the discrete partition of unity.

the asynchronous variant, each subdomain exposes a memory region to remote access via MPI one-sided primitives. On execution of Line 4, the relevant components of current local residual $\mathbf{t}_p$ are written to the neighboring subdomains, and the latest locally available data $\mathbf{t}_q$ from neighbors q is used.

In order to improve the scalability of the solver, a mechanism of global information exchange is required [15, 16]. Let $\boldsymbol{R}_0 \in \mathbb{R}^{n \times n_0}$ 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 in additive fashion $\mathbf{u}^{n+1} = \mathbf{u}^n + \left(\frac{1}{2} \boldsymbol{M}_{RAS}^{-1} + \frac{1}{2} \boldsymbol{R}_0^T \boldsymbol{A}_0^{-1} \boldsymbol{R}_0 \right) (\mathbf{f} - \boldsymbol{A}\mathbf{u}^n)$, where $\boldsymbol{M}_{RAS}^{-1}$ denotes the preconditioner associated with the RAS iteration described above. We focus on the additive version, since it lends itself to asynchronous iterations: subdomain solves and coarse-grid solves are independent of each other. From the mathematical description of two-level additive RAS, one might be tempted to see the coarse-grid problem simply as an additional subdomain. However, 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 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 on 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 rank 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 is permitted. More precisely, writing of the local subdomain residuals to the coarse grid memory region of $\mathbf{r}_0$ is contingent upon the state of the Boolean variable $\texttt{canWriteRHS}_p$. (See Figure 2.) 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 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 light gray, while remote access is printed in dark gray. In a similar fashion, the coarse grid checks whether every subdomain has written a right-hand side to $\mathbf{r}_0$ by polling

```
1:  while not converged do
2:      On subdomains
3:          Local residual: t_p ← D_p R_p f − A_p D_p w_p
4:          if canWriteRHS_p then
5:              r_0 ← r_0 + R_0 R_p^T t_p
6:              canWriteRHS_p ← False
7:              RHSisReady_0[p] ← True
8:          end if
9:          Accumulate asynchronously:
10:         r_p ← Σ_{q=1}^{P} R_p R_q^T t_q
11:         Solve: A_p v_p = r_p
12:         Update: w_p ← w_p + (1/2) v_p
13:         if solutionIsReady_p then
14:             Update: w_p ← w_p + (1/2) c_p
15:             solutionIsReady_p ← False
16:         end if
17:     On coarse grid
18:         if RHSisReady_0[p] ∀p = 1, ..., P then
19:             Solve A_0 v_0 = r_0
20:             for p = 1, ..., P do
21:                 RHSisReady_0[p] ← False
22:                 canWriteRHS_p ← True
23:                 c_p ← R_p R_0^T v_0
24:                 solutionIsReady_p ← True
25:             end for
26:         else
27:             Sleep
28:         end if
29: end while
30: On subdomains
31:     Post-process synchronously
32:     u_p ← Σ_{q=1}^{P} R_p R_q^T D_q w_q
```

Fig. 2: Asynchronous RAS with additive coarse grid in local form. Variables printed in light gray are exposed memory regions that are local to the calling process. Dark gray variables are remote memory regions.

the state of the local Boolean array $\texttt{RHSisReady}_0$. We notice that the algorithm is asynchronous despite the data dependencies. Coarse grid and subdomain solves do not wait for each other.

Since we determined by experiments that performance is adversely affected if the coarse grid constantly polls the status variable $\texttt{RHSisReady}_0$, we added a sleep statement into its work loop. The sleep interval should not be too large, since this results in under-usage of the coarse grid. Keeping the ratio of attempted coarse grid solves 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 solves and solve attempts and either increases or decreases the sleep interval accordingly.

We conclude this section with a note on convergence theory for the asynchronous case. Contrary to the synchronous case, where the condition $\rho(\boldsymbol{E}) < 1$ on the iteration matrix $\boldsymbol{E}$ of the method is necessary and sufficient for convergence, asynchronous convergence is guaranteed if $\boldsymbol{E}$ is a block H-matrix which is a P-contraction [9]. Obtaining a prediction for a rate at which the asynchronous method converges appears to be more elusive which is why we limit ourselves to experimental comparisons.

3 Numerical Experiments

The performance of linear iterative methods is typically measured by the average contraction factor per iteration $\widetilde{\rho} = (r_{\text{final}}/r_0)^{\frac{1}{K}}$, where r_0 is the norm of the initial residual vector, r_{final} the norm of the final residual vector, and K 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: $\widehat{\rho} = (r_{\text{final}}/r_0)^{\frac{\tau_{\text{sync}}}{T}}$. Here, T is the total iteration time, and τ_{sync} is the average time for a single iteration in the synchronous case. 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.

We expect the performance of the asynchronous method relative to its synchronous counterpart to be essentially dependent on the communication stencil. Here, we limit ourselves to a simple 2D problem. Further experiments for more complicated PDEs and as well as in 3D are part of future work.

As a test problem, we solve $-\Delta u = f$ in $\Omega = [0,1]^2$ subject to the boundary condition $u = 0$ on $\partial\Omega$, where the right-hand side is $f = 2\pi^2 \sin(\pi x) \sin(\pi y)$ and the corresponding solution is $u = \sin(\pi x) \sin(\pi y)$. We discretize Ω using a uniform triangular mesh and approximate the solution using piece-wise linear finite elements.

In classical synchronous iterative methods, a stopping criterion of the form $r < \varepsilon$ is evaluated at every iteration. Here, r is the norm of the residual vector and ε is a prescribed tolerance. 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 terminate the iteration using a simplistic convergence criterion where each process writes its local contribution to the residual norm to a master rank, say rank 0. The master rank sums the contributions, and exposes the result through another MPI window. Each subdomain can retrieve this estimate of the global residual norm, and terminates if it is smaller than the prescribed tolerance. 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 [1, 14] for an overview of more elaborate approaches.

All runs are performed on the Haswell partition of Cori at NERSC. While the code was written from scratch, the differences between the synchronous and the asynchronous code paths are limited, since only the communication layer and stopping criterion need to be changed. One MPI rank is used per core, i.e. 32 ranks per Haswell node. For the two-level method, the coarse grid solve is performed on a single rank. The underlying mesh is partitioned using METIS [12]. Both subdomain and coarse grid problems are factored and solved using the SuperLU [13, 7] direct solver. This choice is guided by the desire to eliminate the impact that inexact solves such as preconditioned iterative sub-solves might have on the overall convergence.

One-level RAS We compare synchronous and asynchronous one-level RAS in a strong scaling experiment, where we fix the global problem size to about 261,000 unknowns, and vary the number of subdomains between 4 and 256. We obviously cannot expect good scaling for this one-level method, since increasing the number of subdomains adversely affects the rate of convergence. In Figure 3a we display solve time, final residual norm and approximate rate of convergence. It can be observed that the synchronous method is faster for smaller subdomain count, yet comparatively slower for larger number of subdomains. The crossover point is at 64 subdomains.

An important question is whether the asynchronous method converges because every subdomain performs the same number of local iterations, and hence the asynchronous method just mirrors the synchronous one, merely with the communication method replaced. The histogram in Figure 3c shows that this is not the case. The number of local iterations varies significantly between 11,000 and 16,000 iterations. The problem was load balanced by the number of unknowns, thus the local solves are also approximately balanced but the communication is likely slightly imbalanced.

The advantage of asynchronous RAS becomes even clearer when the experiment is repeated with one of the subdomains being 50% larger, thereby artificially creating load imbalance. In Figure 3b we observe that the asynchronous method outperforms the synchronous one in all but the smallest run.

Two-level RAS In order to gauge the performance and scalability of the synchronous and asynchronous two-level RAS solvers, we perform a weak scaling experiments. We use 16, 64, 256 and 1024 subdomains. 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, with approximately 16 unknowns per subdomain. In Figure 4a we plot the solution time, the achieved residual norm and the average contraction factor $\widehat{\rho}$. 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 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 4c.) One can observe that for 16, 64 and 256 subdomains, asynchronous and synchronous method take almost the same time. For 1024 subdomains, however, the synchronous method is seen to take drastically more time. For this case 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. The third subplot of Figure 4a shows that the asynchronous method outperforms its synchronous equivalent in all but the smallest problem.

To further illustrate the effect of load imbalance, we repeat the previous experiment with one subdomain being 50% larger. The results are shown in Figure 4b. The results are consistent with the previous case, and the performance advantage of the asynchronous method over the synchronous one has increased. Even when the size

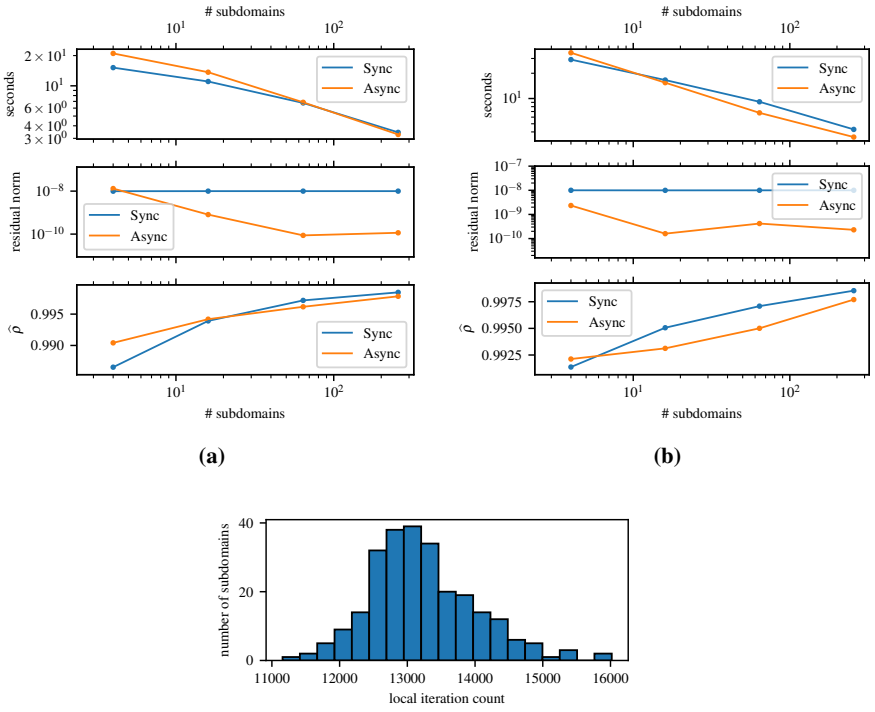

Fig. 3: *(a)* Performance of synchronous and asynchronous one-level RAS for a system size of approximately 261,000 unknowns. The subdomains are load balanced. From top to bottom: Solution time, final residual norm, and the resulting approximate contraction factor. *(b)* Performance of synchronous and asynchronous one-level RAS for a system size of approximately 261,000 unknowns under load imbalance: one subdomain is 50% larger than the rest. *(c)* Histogram of local iteration counts asynchronous one-level RAS with 256 subdomains in the balanced case.

of the coarse grid system is smaller than the size of the typical subdomain problem, the asynchronous method outperforms its synchronous counterpart.

Acknowledgements We thank Daniel Szyld for helpful discussions on asynchronous methods. 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. 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

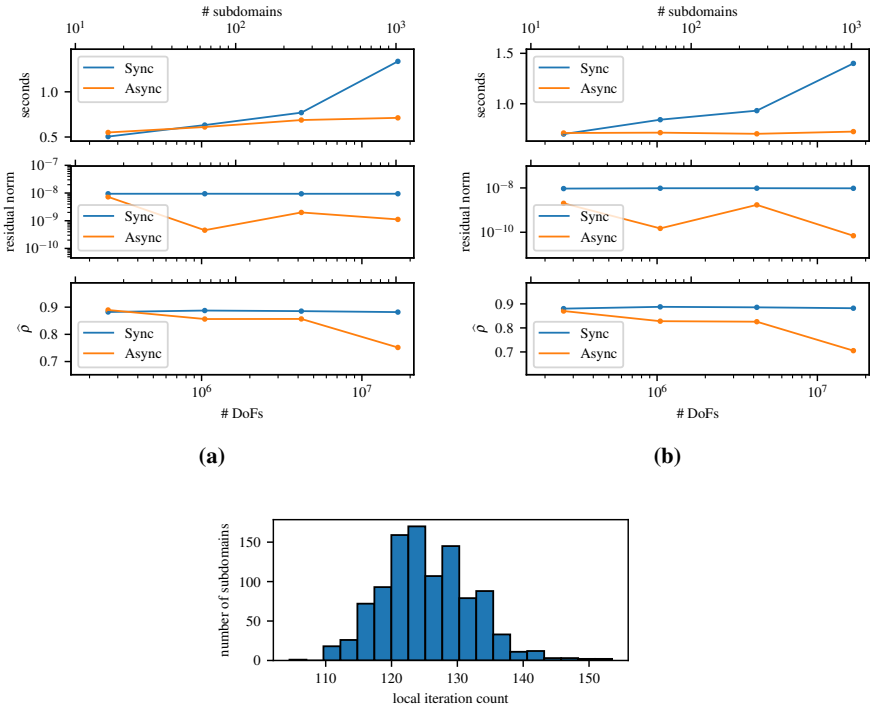

Fig. 4: *(a)* Weak scaling of synchronous and asynchronous two-level additive RAS, load balanced case. From top to bottom: Total solution time, final residual norm, and approximate contraction factor. *(b)* Weak scaling of synchronous and asynchronous two-level additive RAS under load imbalance: one subdomain is 50% larger than all the other ones. *(c)* Histogram of local iteration counts asynchronous two-level additive RAS with 1024 subdomains in the balanced case.

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.

References

1. Bahi, J.M., Contassot-Vivier, S., Couturier, R., Vernier, F.: A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms. IEEE Transactions on Parallel and Distributed Systems **16**(1), 4–13 (2005)
2. Baudet, G.M.: Asynchronous iterative methods for multiprocessors. Journal of the ACM (JACM) **25**(2), 226–244 (1978)
3. Bertsekas, D.P.: Distributed asynchronous computation of fixed points. Mathematical Programming **27**(1), 107–120 (1983)
4. Cappello, F., Geist, A., Gropp, B., Kale, L., Kramer, B., Snir, M.: Toward exascale resilience. International Journal of High Performance Computing Applications **23**(4), 374–388 (2009)

5. Cappello, F., Geist, A., Gropp, W., Kale, S., Kramer, B., Snir, M.: Toward exascale resilience: 2014 update. Supercomputing frontiers and innovations **1**(1), 5–28 (2014)
6. Chazan, D., Miranker, W.: Chaotic relaxation. Linear algebra and its applications **2**(2), 199–222 (1969)
7. Demmel, J.W., Eisenstat, S.C., Gilbert, J.R., Li, X.S., Liu, J.W.H.: A supernodal approach to sparse partial pivoting. SIAM J. Matrix Analysis and Applications **20**(3), 720–755 (1999)
8. Dolean, V., Jolivet, P., Nataf, F.: An introduction to domain decomposition methods: algorithms, theory, and parallel implementation, vol. 144. SIAM (2015)
9. Frommer, A., Szyld, D.B.: On asynchronous iterations. Journal of Computational and Applied Mathematics **123**(1), 201–216 (2000)
10. Ghysels, P., Ashby, T.J., Meerbergen, K., Vanroose, W.: Hiding global communication latency in the GMRES algorithm on massively parallel machines. SIAM Journal on Scientific Computing **35**(1), C48–C71 (2013)
11. Glusa, C., Ramanan, P., Boman, E.G., Chow, E., Rajamanickam, S.: Asynchronous One-Level and Two-Level Domain Decomposition Solvers. ArXiv e-prints (2018)
12. Karypis, G., Kumar, V.: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. SIAM Journal on Scientific Computing **20**(1), 359–392 (1998)
13. Li, X., Demmel, J., Gilbert, J., iL. Grigori, Shao, M., Yamazaki, I.: SuperLU Users' Guide. Tech. Rep. LBNL-44289, Lawrence Berkeley National Laboratory (1999)
14. Magoulès, F., Gbikpi-Benissan, G.: Distributed convergence detection based on global residual error under asynchronous iterations. IEEE Transactions on Parallel and Distributed Systems (2017)
15. Smith, B., Bjorstad, P., Gropp, W.: Domain decomposition: parallel multilevel methods for elliptic partial differential equations. Cambridge university press (2004)
16. Toselli, A., Widlund, O.: Domain decomposition methods: algorithms and theory, vol. 3. Springer (2005)