A Two-Level Restricted Additive Schwarz Method for Asynchronous Computations

Faycal Chaouqui and Daniel B. Szyld

1 Introduction

In this paper, we investigate the parallel performance of both synchronous and asynchronous domain decomposition methods (DDMs) for the solution of algebraic systems coming from the discretization of partial differential equations (PDEs). In particular, we extend the ideas introduced in [8] for different types of coarse space corrections. We consider a PDE of the form $L(u) = f$ on $\Omega \subset \mathbb{R}^2$ such that $u|_\Omega = 0$. The operator L after discretization yields a large sparse system of algebraic equations of the form

$$A\mathbf{u} = \mathbf{f}, \tag{1}$$

where $A \in \mathbb{R}^{n\times n}$ and $\mathbf{f} \in \mathbb{R}^n$. Here, we focus our attention on the Restricted Additive Schwarz (RAS) domain decomposition solver [2, 4]. For the sake of simplicity, we assume that $L = -\Delta$. We assume that the domain Ω is decomposed into p overlapping subdomains $\Omega_1, \ldots, \Omega_p$. Let $R_i^\top$, $i = 1, \ldots, p$, denotes the boolean matrix that maps the local degrees of freedom defined in Ω_i to Ω. We define the local stiffness matrix $A_i = R_i A R_i^\top$. Let us also define the diagonal matrices D_i, $i = 1, \ldots p$, such that we satisfy the partition of unity, i.e., $\sum_{i=1}^p R_i^\top D_i R_i = I$, where I denotes the identity matrix in $\mathbb{R}^{n\times n}$. The RAS iteration is then defined as

$$\mathbf{u}^{k+1} = \mathbf{u}^k + \sum_{i=1}^{p} R_i^\top D_i A_i^{-1} R_i (\mathbf{f} - A\mathbf{u}^k). \tag{2}$$

We note that in our case, the matrices D_i correspond to diagonal boolean matrices that are 1 in the non-overlapping partition, and 0 otherwise. We note also that there

Faycal Chaouqui
COMSOL, INC, Burlington, Mass., USA, e-mail: chaouqui@temple.edu

Daniel Szyld
Temple University, Philadelphia, Pa., USA, e-mail: szyld@temple.edu

are other ways for choosing those matrices, and we refer the reader to [4, 5]. In the next section, we will describe briefly the asynchronous RAS method.

2 Asynchronous restricted additive Schwarz

We briefly describe asynchronous iterations (see, e.g., [3]) for fixed point problems defined on a product space $U = U_1 \times \cdots \times U_p$, of the form $\mathbf{u} = \mathcal{T}\mathbf{u}$ with a unique solution. In other words, we have $\mathbf{u} = (\mathbf{u}_1, \ldots, \mathbf{u}_p)$ and $\mathcal{T} = (\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_p)$, with $\mathcal{T}_s \colon U \to U_s$. We have in mind that the operation in process s, of the form $\mathbf{u}_s = \mathcal{T}_s(\mathbf{u}_1, \ldots, \mathbf{u}_p)$ is performed without synchronization, i.e., without waiting for other processors to send new information.

For a mathematical model of these asynchronous iterations on p processors, we follow the model introduced by Bertsekas [1]. To that end, we define a time stamp k, $k \in \mathbb{N}$, and denote by $\{\sigma(k)\}_{k\in\mathbb{N}}$ the sequence of non-empty subsets of $\{1, \ldots, p\}$, defining which processes update their components at the time stamp k. Define also for $s, q \in \{1, \ldots, p\}$, $\left\{\tau_q^s(k)\right\}_{k\in\mathbb{N}}$ a sequence of integers, representing the update number (or time stamp) of the data coming from process q and available on process s at the time k. Thus, a *delay* would be $k - \tau_q^s(k)$. We begin with an initial approximation $\mathbf{u}^0 = \left(\mathbf{u}_1^0, \ldots, \mathbf{u}_p^0\right)$, and define, for each process s, the asynchronous iterations as follows.

$$\mathbf{u}_s^{k+1} = \begin{cases} \mathcal{T}_s\left(\mathbf{u}_1^{\tau_1^s(k)}, \ldots, \mathbf{u}_p^{\tau_p^s(k))}\right) & \text{if } s \in \sigma(k+1), \\ \mathbf{u}_s^k & \text{if } s \notin \sigma(k+1)\ . \end{cases} \tag{3}$$

In this model, one also assumes that the three following natural conditions are satisfied

$$\forall s, q \in \{1, \ldots, p\}, \forall k \in \mathbb{N}, \tau_q^s(k) \le k, \tag{4}$$

$$\forall s \in \{1, \ldots, p\}, \text{card}\,\{k \in \mathbb{N} | s \in \sigma(k)\} = +\infty, \tag{5}$$

$$\forall s, q \in \{1, \ldots, p\}, \lim_{k\to+\infty} \tau_q^s(k) = +\infty. \tag{6}$$

Condition (4) represents the fact that data used at the time k must have been produced before time k, i.e., time does not flow backward. Condition (5) indicates that no process will ever stop updating its components. Condition (6) means 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.

One important theoretical result states that for a fixed point problem, say $T(\mathbf{u}) = \mathbf{u}$, on a product space, under conditions (4)–(6), if there is a norm such that the map T is contracting, i.e., if the (synchronous) fixed point iteration converges, then, the corresponding asynchronous iteration converges as well; see, e.g., [3] and references therein. For the RAS iteration, the map $\mathcal{T}_s$ defined in (3) is equivalent to

Algorithm 1 (Asynchronous RAS)

```
1: Input: $\mathbf{u}^0$.
2: Output: $\mathbf{u} \approx \mathbf{u}^*$.
3: Set $\mathbf{r}^0 = \mathbf{f} - A\mathbf{u}^0$, converged = false.
4: In parallel, each processor core $s$:
5: while converged = false do
6:     Set $\mathbf{u}_s = \mathcal{T}_s(\mathbf{u}_1, \dots, \mathbf{u}_p)$          ▷ Update subdomain $s$
7:     Compute $\|D_s\mathbf{r}_s\|_2$          ▷ Compute local residual norm
8:     if s == 1 then
9:         Compute $\|\mathbf{r}\|_2 = \sqrt{\sum_{i=1}^{p} \|D_i\mathbf{r}_i\|_2^2}$
10:        if $\|\mathbf{r}\|_2/\|\mathbf{r}^0\|_2 \leq \epsilon$ then          ▷ Check global convergence
11:            converged = true
12:        end if
13:    end if
14: end while (for processor s)
15: Set $\mathbf{u} = \sum_{s=1}^{p} R_s^\top D_s\mathbf{u}_s$          ▷ Assemble global solution
```

$$\mathcal{T}_s(\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_p) = \mathbf{u}_s + R_s \sum_{i=1}^{p} R_i^\top D_i A_i^{-1} \mathbf{r}_i, \tag{7}$$

where $\mathbf{r}_i = R_i(\mathbf{f} - A\mathbf{u})$ it the local residual for the subdomain Ω_i, $i = 1, \dots p$. The implementation of iteration (3) is presented in Algorithm 1.

In Algorithm 1 each processor core computes and updates the components of the local vector as well as the corresponding local residual norms. A processor core is then in charge of accumulating all the local residuals and computing the global residual. The algorithm then stops when the global residual is smaller than the tolerance. We provide results of numerical examples illustrating the performance of both synchronous and asynchronous RAS. We consider $\Omega = [0, 1] \times [0, 1]$ decomposed into regular squares with a total of p subdomains and a minimal overlap. The source term **f** is chosen such that $\sin(\pi x)\sin(\pi y)$ corresponds to the exact monodomain solution. We partition the domain into $p = 4 \times 4$ subdomains with a total of 10k discretization points. We note that each processor core was assigned to one subdomain. All the tests were carried out on a shared memory machine which consists of 88 CPU cores / 176 threads and 1536GB of RAM. The implementation of Algorithm 1 was in C++ and the parallelization uses the OpenMP multithreading directives. We run two different types of experiments. In the first run, we assume all processors run at the same speed and compare both the timings required by both synchronous and asynchronous to reach a specified tolerance. This is illustrated in Figure 1 (left). We can see that in this case the synchronous is faster than the asynchronous. To show the advantage of the asynchronous approach, we repeat the experiment but with one processor core twice as slow. This can be realized by measuring the time needed for a single update and then forcing the processor to sleep (idle) for that amount of time. In this manner we mimic heterogeneous architectures, as well as cases where one subdomain is larger than the others. We can observe from Figure 1 (right) that the asynchronous is faster than the synchronous in this case.

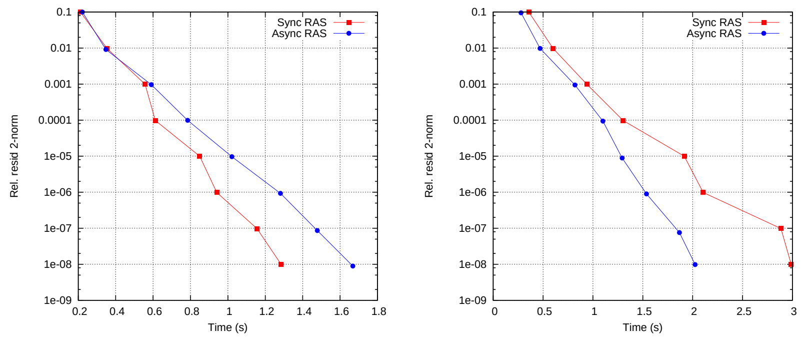

Fig. 1 Left: CPU time versus relative residual 2-norm for synchronous and asynchronous RAS with $p = 16$. Right: Same but one thread is twice as slow.

3 Two-level asynchronous restricted additive Schwarz

A second level is an essential component to obtain a robust domain decomposition method. It relies generally on solving a smaller problem on a coarser mesh so that there is global communication between the subdomains. The coarse space allows us then to construct the coarse restriction matrix R_0. The two-level RAS is then defined as

$$\begin{aligned} \mathbf{u}^{k+1/2} &= \mathbf{u}^k + \sum_{i=1}^{p} R_i^\top D_i A_i^{-1} R_i(\mathbf{f} - A\mathbf{u}^k) \\ \mathbf{u}^{k+1} &= \mathbf{u}^{k+1/2} + R_0^\top A_0^{-1} R_0(\mathbf{f} - A\mathbf{u}^{k+1/2}). \end{aligned} \tag{8}$$

In order to use iteration (8) asynchronously, we need to use the coarse grid in an additive way. This can be done by using a weighted additive version or multiplicative/additive variant of (8). The corresponding two-level mapping $\tilde{\mathcal{T}}$ can be expressed in the case of the additive variant as

$$\tilde{\mathcal{T}}_s(\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_p) = \mathbf{u}_s + R_s \left(\frac{1}{2} \sum_{i=1}^{p} R_i^\top D_i A_i^{-1} \mathbf{r}_i + \frac{1}{2} R_0^\top A_0^{-1} R_0 \sum_{i=1}^{p} R_i^\top D_i \mathbf{r}_i \right). \tag{9}$$

For work using a multiplicative additive variant, we mention [7, 9]. To avoid over-correction from the coarse grid, we have to make sure that no subdomain is corrected again until all the remaining subdomains have updated at least once [8]. We present in Algorithm 2, the implementation of the asynchronous two-level RAS.

We describe now the coarse correction we use. We would like the coarse grid to ensure the scalability of the method as a solver. In the same spirit of [6], we use harmonically extended coarse basis functions. Let n_i denotes the number of cross points for each subdomain $\Omega_i, i = 1, \dots p$. Let $\varphi_i^j, j = 1, \dots n_i$ define a piecewise

Algorithm 2 (Asynchronous two-level RAS)

```
1: Input: $\mathbf{u}^0$.
2: Output: $\mathbf{u} \approx \mathbf{u}^*$.
3: Set $\mathbf{r}^0 = \mathbf{f} - A\mathbf{u}^0$, converged = false.
4: Set update[s]=false, and correction[s]=false, $s = 1, \dots, p$.
5: In parallel, each processor $s$:
6: while converged = false do
7:     if $s > 0$ then
8:         if correction[s] then                        ▷ Check if coarse correction is needed
9:             Set $\mathbf{u}_s = \tilde{\mathcal{T}}_s(\mathbf{u}_1, \dots, \mathbf{u}_p)$          ▷ Update subdomain $s$
10:            Set correction[s]=false
11:        else
12:            Set $\mathbf{u}_s = \mathcal{T}_s(\mathbf{u}_1, \dots, \mathbf{u}_p)$          ▷ Update subdomain $s$
13:            Set update[s]=true
14:        end if
15:        Compute $\|D_s \mathbf{r}_s\|_2$                  ▷ Compute local residual norm
16:        if s == 1 then
17:            Compute $\|\mathbf{r}\|_2 = \sqrt{\sum_{i=1}^p \|D_i \mathbf{r}_i\|_2^2}$
18:            if $\|\mathbf{r}\|_2 / \|\mathbf{r}^0\|_2 \le \epsilon$ then            ▷ Check global convergence
19:                converged = true
20:            end if
21:        end if
22:    else
23:        if update[q], $\forall q = 1, \dots, p$ then        ▷ Check if all subdomains updated
24:            Compute the coarse correction.
25:            Set correction[i]=true, $i = 1, \dots, p$
26:            Set update[i]=false, $i = 1, \dots, p$
27:        end if
28:    end if
29: end while (for processor $s$)
30: Set $\mathbf{u} = \sum_{s=1}^p R_s^\top D_s \mathbf{u}_s$               ▷ Assemble global solution
```

linear function on $\partial\Omega_i$ that is 1 at one cross point and 0 on the others. We define the coarse basis functions ϕ_i^j, $j = 1, \dots, n_i, i = 1, \dots p$, as the solution of

$$\begin{cases} L|_{\Omega_i}(\phi_i^j) = 0, \text{on } \Omega_i \\ \qquad \phi_i^j = \varphi_i^j, \text{ on } \partial\Omega_i \, . \end{cases} \tag{10}$$

We define our coarse space $\mathcal{Z} \subset \mathbb{R}^d$ as the span of extended coarse functions ϕ_i^j, i.e.,

$$\mathcal{Z} = \text{span}\left\{ R_i^\top \phi_i^j, j = 1, \dots, n_i \, i = 1, \dots, p \right\}. \tag{11}$$

The columns of the matrix $R_0^\top$ forms a basis of $\mathcal{Z}$. We show in Table 1 the number of iterations needed to reach a tolerance $\epsilon = 10^{-8}$ with this specific coarse space for (synchronous) RAS as a solver. We can see that the two-level method outperforms the one-level method and is also scalable, i.e., the number of iterations does not increase when we grow the number of subdomains. We also report the iterations required

Table 1 Weak scalability of additive two-level RAS

p	n	dim $\mathcal{Z}$	dim $\mathcal{Z}_{MG}$	#iter (RAS)	#iter (RAS+$\mathcal{Z}$)	#iter (RAS+$\mathcal{Z}_{MG}$)
16	14400	36	49	630	93	144
25	22500	64	81	953	98	107
36	32400	100	121	1344	99	108
49	44100	144	169	1802	100	99
64	57600	196	225	2325	100	99

for two-level RAS constructed using a multigrid (MG) approach with four levels of coarsening. We can observe that the coarse grid considered in our simulation is asymptotically similar to MG for our model problem L. However, it has a smaller coarse grid size.

Next, we test the performance of the two-level asynchronous Algorithm 2 by comparing the time needed to reach a specified tolerance. In Figure 2 we plot the timing versus the residual norm for both synchronous and asynchronous two-level methods. We can observe that in this case the synchronous is faster than the asynchronous. We also see that the timing required to converge is faster than for the one-level method. The introduction of heterogeneity among processors yields a faster asynchronous two-level method. We note that as is the case for the local subdomains, the coarse problem was solved exactly since it is small for the coarse space defined (11). In Table 2, we report the timings required for both synchronous and asynchronous one and two-level RAS for processors with random time delays. We realize this by adding a random time delay to each processor core that follows a uniform density function of the form $\mathcal{U}(0, \varepsilon T_s)$, where T_s is the timings required for the processor s to finish its workload, and $\varepsilon = 0.01, 0.1, 1$. We can observe from Table 2 that the introduction of heterogeneities in the computation, even with a small magnitude reveals the advantages of asynchronous computations.

In Figure 3 we test the weak scalability of both the synchronous and asynchronous methods. To do so, we fix the tolerance to $\epsilon = 10^{-6}$ and the subdomain's size to 1600,

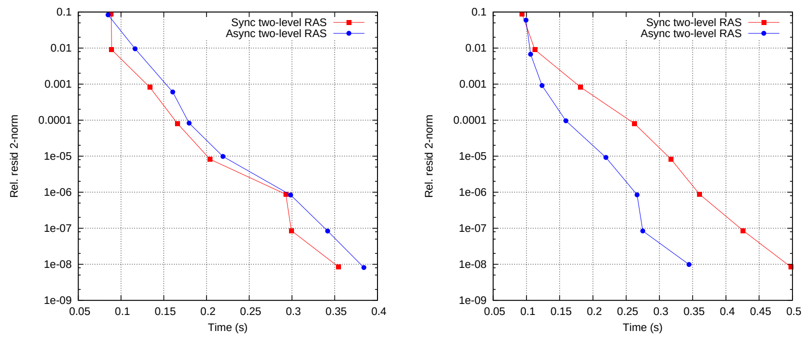

Fig. 2 Left: CPU time versus relative residual 2-norm for two-level synchronous and asynchronous RAS with $p = 16$. Right: Same but one thread is twice as slow.

Table 2 Timing required (in sec) of synchronous and asynchronous one- and two-level RAS to reach a tolerance of 10^{-8} for different levels of heterogeneities.

ε	Sync RAS	Async RAS	Sync two-level RAS	Async two-level RAS
0.01	2.3173	2.1973	0.4547	0.3539
0.1	2.4788	2.4051	0.4075	0.3601
1	5.9415	5.7202	1.0837	1.0295

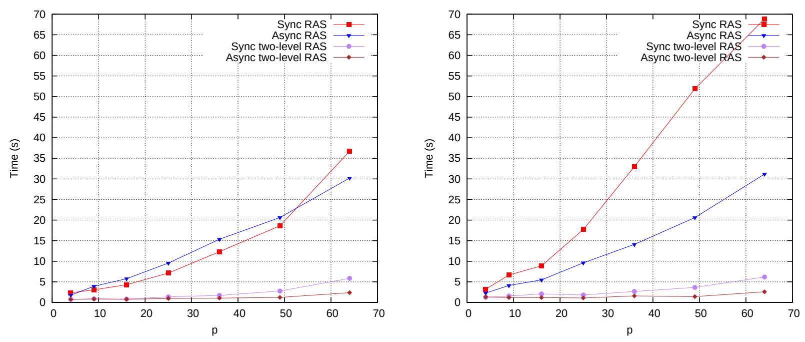

Fig. 3 Left: The number of subdomains versus the CPU time needed for convergence for one and two-level synchronous and asynchronous RAS. Right: Same but with a processor core twice as slow.

then run the two-level algorithms and measure the CPU time required to converge. We also plot the time required for the synchronous one as well. In Figure 3 (left), all the processors run at the same speed and there is no load imbalance. We can observe that in this case, the two-level asynchronous method is the fastest among all the four methods. The one-level synchronous method is still slightly faster than the one-level asynchronous (except for $p = 64$). In Figure 3 (right) we repeat the same experiment, but with one processor core twice as slow. We can see now that the asynchronous method outperforms the synchronous method. This is true for both the one- and two-level methods. Observe also that while the two-level synchronous method is slightly slower in the simulated heterogeneous architecture (for $p = 64$, 5.85 sec vs. 2.38 sec), the asynchronous method is faster (2.59 sec vs. 6.17 sec). The introduction of heterogeneity clearly shows how asynchronous can be effective in practice.

4 Conclusion

In this paper, we analyzed the performance of one and two-level synchronous and asynchronous RAS. In particular, we used a specific coarse grid correction for our asynchronous computations. Our numerical results suggest that the asynchronous methods exhibit good performance. In particular, we observed that for heterogeneous hardware, the asynchronous outperforms the synchronous method. This was valid for both the one and two-level methods.

References

1. Bertsekas, D. P. Distributed asynchronous computation of fixed points. *Mathematical Programming* **27**(1), 107–120 (1983).
2. Cai, X.-C. and Sarkis, M. A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM Journal on Scientific Computing* **21**, 792–797 (1999).
3. Frommer, A. and Szyld, D. B. On Asynchronous Iterations. *Journal of Computational and Applied Mathematics* **123**, 201–216 (2000).
4. Frommer, A. and Szyld, D. B. An algebraic convergence theory for restricted additive Schwarz methods using weighted max norms. *SIAM Journal on Numerical Analysis* **39**, 463–479 (2001).
5. Gander, M. J. Does the Partition of Unity Influence the Convergence of Schwarz Methods? In: Haynes, R., MacLachlan, S., Cai, X.-C., Halpern, L., Kim, H., Klawonn, A., and Widlund, O. (eds.), *Domain decomposition methods in science and engineering XXV*, *Lecture Notes in Computational Science and Engineering*, vol. 138, 3–15. Springer, Cham (2018).
6. Gander, M. J. and Loneland, A. SHEM: An optimal coarse space for RAS and its multiscale approximation. In: Lee, C.-O., Cai, X.-C., Keyes, D., Kim, H., Klawonn, A., Park, E.-J., and Widlund, O. (eds.), *Domain decomposition methods in science and engineering XXIII*, *Lecture Notes in Computational Science and Engineering*, vol. 116, 313–321. Springer (2017).
7. Gbikpi-Benissan, G. and Magoulès, F. Asynchronous Multiplicative Coarse-Space Correction. *SIAM Journal on Scientific Computing* **44**, C237–C259 (2022).
8. Glusa, C., Boman, E. G., Chow, E., Rajamanickam, S., and Szyld, D. B. Scalable Asynchronous Domain Decomposition Solvers. *SIAM Journal on Scientific Computing* **42**, C384–C409 (2020).
9. Wolfson-Pou, J. and Chow, E. Asynchronous multigrid methods. In: *2019 IEEE international parallel and distributed processing symposium (IPDPS)*, 101–110. IEEE (2019).