Proceedings of the 18th Conference on Computer Science and Intelligence Systems pp. 441–450
DOI: 10.15439/2023F8932
ISSN 2300-5963 ACSIS, Vol. 35

Resilient s-ACD for Asynchronous Collaborative Solutions of Systems of Linear Equations

Lucas Erlandson*, Zachary Atkins†, Alyson Fox*, Christopher J. Vogl*, Agnieszka Międlar‡, and Colin Ponce*
*Center for Applied Scientific Computing
Lawrence Livermore National Lab
ORCIDS: 0000-0003-4544-6148, None, 0000-0002-3855-694X, 0000-0002-6720-8805
Emails: {erlandson3,fox33,vogl2,ponce11}@llnl.gov
†University of Colorado Boulder
ORCID: 0000-0002-2491-0725
Email: zach.atkins@colorado.edu
‡Department of Mathematics, Virginia Tech
Blacksburg, VA
ORCID: 0000-0002-2995-7426
Email: amiedlar@vt.edu

***Abstract*—Solving systems of linear equations is a critical component of nearly all scientific computing methods. Traditional algorithms that rely on synchronization become prohibitively expensive in computing paradigms where communication is costly, such as heterogeneous hardware, edge computing, and unreliable environments. In this paper, we introduce an s-step Approximate Conjugate Directions (s-ACD) method and develop resiliency measures that can address a variety of different data error scenarios. This method leverages a Conjugate Gradient (CG) approach locally while using Conjugate Directions (CD) globally to achieve asynchronicity. We demonstrate with numerical experiments that s-ACD admits scaling with respect to the condition number that is comparable with CG on the tested 2D Poisson problem. Furthermore, through the addition of resiliency measures, our method is able to cope with data errors, allowing it to be used effectively in unreliable environments.**

I. Introduction

SOLVING a system of linear equations $A\mathbf{x} = \mathbf{b}$ is a critical kernel in many applications, studied in great detail across applications, as well as for both iterative [1], [2] and direct solves [3], [4], [5], [6]. However, even iterative methods such as Krylov subspace methods, which have reduced serialization [7], require global synchronization. One of the most popular of such methods, the Conjugate Gradient (CG) method, computes global inner product at each iteration [8], [9]. The burden of this synchronization cost is increasing in modern computing environments due to two reasons: 1) as the number of parallel processes rapidly increases, the cost of global synchronization does too; 2) new environments are being considered for computationally expensive tasks, e.g. distributed (drones, power grid) and heterogeneous (accelerators) computing. Due to these factors, there is a critical need for asynchronous algorithms that can operate without the need for synchronization at every iteration and are able to handle the increase of data errors introduced by the increased number of failure points and unreliabilities.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and was supported by the LLNL-LDRD Program under Project No. 21-FS-007 and 22-ERD-045. The work of Agnieszka Międlar was supported by the National Science Foundation (NSF) under Grant DMS-2144181 and DMS-2324958. LLNL-CONF-849356.

The contributions of this paper are four-fold:

1) We introduce the s-step Approximate Conjugate Directions (s-ACD) method, which is a novel asynchronous Krylov-like linear system solver that achieves scaling with respect to the condition number that is comparable with CG on the tested 2D Poisson problem.
2) We introduce and investigate a demonstrative scenario that results in data errors by introducing unreliabilities into the calculations.
3) We develop resiliency measures that are used in s-ACD to detect corruption and could be used in other iterative methods.
4) We provide a comparative study using numerical experiments of the newly introduced developments.

In Section II, we begin by providing a background of iterative and asynchronous methods. Additionally, we briefly discuss *Skywing* [10], the collaborative autonomy framework that provides the agent-based approach that our implementation utilizes. Following this, Section III describes the s-ACD method for the solution of linear systems. Section IV discusses how the corruption of calculations in an unreliable environment are modeled and what situations the resulting corruption type and failure model might apply to. Then, Section V introduces methods that can be used for adding resiliency to iterative solvers, which results in the Resilient s-ACD method. Section VI provides a variety of numerical experiments developed to test the properties of the various methods to demonstrate their efficacy. Final concluding remarks are presented in Section VII.

II. Background

First, let us consider the Krylov subspace methods, a class of iterative methods where an initial guess of the solution to $A\mathbf{x} = \mathbf{b}$ is updated by iteratively building up a Krylov subspace span$\{\mathbf{b}, A\mathbf{b}, A^2\mathbf{b}, \ldots\}$. For symmetric positive definite matrices (SPD), the CG method, a particular Krylov solver, constructs a series of direction vectors $\mathbf{p}^\kappa$ that are A-conjugate to each other, as well as residual vectors $\mathbf{r}^\kappa$ that span the same Krylov subspace for iteration κ [8], [9]. Asymptotically, this method achieves convergence to a given tolerance in $\mathcal{O}(\sqrt{\text{cond}\,(A)})$ iterations for a problem with condition number $\text{cond}\,(A)$, making it the solver of choice for SPD matrices. However, global communication is needed to ensure the orthogonality necessary for the method to be robust.

To reduce the cost of global communication, methods such as the communication-avoiding s-step algorithms [11], [12] and communication hiding pipeline methods [13], [14] have been proposed in the synchronous case [15]. To address the issue of unreliable computing environments, some fault-tolerant or resilient CG methods are also available [16], [17], [18]. However, both of these classes of methods still require a high level of global synchronization for orthogonality to be preserved. In sufficiently distributed environments, these costs may become too restrictive, leading to the need of methods that do not require global synchronization and exact orthogonality.

For the solution of linear systems, chaotic or asynchronous methods [19], [20], [21] such as asynchronous Jacobi [22], [23], [24] have been developed to provide asynchronicity to already existing solvers. Although resiliency has been added to asynchronous Jacobi [25] to make it more fault-tolerant, the number of iterations scales proportional to the condition number, driving the need for more powerful asynchronous linear solvers.

A. Skywing

Edge computing, in which many small devices exist and work together in an unstructured setting, is a rapidly growing field in computing. Edge computing applications pose a unique set of challenges:

1) Both the physical and cyber environments are highly unreliable, as devices are placed in uncontrolled locations, e.g. homes or along power lines. As such, they can readily and unexpectedly break, get unplugged, or become compromised by cyberattacks.
2) The collection of participating devices is often quite heterogeneous, with a range of vendors and device capabilities.
3) The computational workflows are frequently streaming workflows that continually monitor and respond to some needs, rather than being a single computational task that terminates upon completion.

While traditional parallel computing paradigms, such as HPC or database computing, each have some of these challenges in common, the combination is unique to edge computing.

This paper details a new method in the *collaborative autonomy* paradigm, a class of methods in which multiple computational units work independently of each other but towards a common goal. Through adapting to unreliability present in the environment, these methods can provide reliable computing in unreliable environments.

Existing software platforms like Apache Hadoop [26] and Apache Spark [27] are designed for large-scale, "big data" computing work, but they largely implement leader-follower patterns and perform computing in batches. These approaches, while effective in controlled cluster environments, lack the resilience necessary to withstand common faults in edge computing applications such as hardware faults and, increasingly, cyber intrusions. Other parallel computing frameworks, such as OpenMP and MPI, do not necessarily rely on leader-follower paradigms, but are more naturally designed for well-controlled environments and terminating computational tasks.

Skywing is a software platform developed at Lawrence Livermore National Lab, which follows a publication/subscription paradigm. This allows any agent involved in the computation to subscribe or publish to a stream of data, and any data on a stream an agent is subscribed to is considered agnostic. Because of the unstructured nature, this enables increased flexibility, particularly for consensus based methods. Skywing aims to provide *method composition* to enable a modular approach, allowing users to utilize appropriate levels of resiliency for each module. The source code of Skywing is available on GitHub [10].

III. s-ACD

A. Problem Statement

Consider solving the linear system

$$A\mathbf{x} = \mathbf{b} \tag{1}$$

for $\mathbf{x} \in \mathbb{R}^m$, where $A \in \mathbb{R}^{m\times m}$ and $\mathbf{b} \in \mathbb{R}^m$. Assume the linear system is distributed across N agents according to a non-overlapping partition. Denote $A_i \in \mathbb{R}^{m_i \times m}$, $m_i < m$, as the block of rows of matrix A that are stored on agent i, $i = 1, \ldots, N$. Then

$$A = [A_1^T \cdots A_N^T]^T. \tag{2}$$

Given a SPD matrix A, two vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^m$ are A-conjugate if and only if

$$\langle \mathbf{u},\ \mathbf{v} \rangle_A \coloneqq \langle \mathbf{u},\ A\mathbf{v} \rangle = \mathbf{u}^T A\mathbf{v} = 0. \tag{3}$$

This paper establishes an asynchronous iterative method for solving $A\mathbf{x} = \mathbf{b}$, where agent i computes successive approximations to the solution vector $\mathbf{x}$, denoted $\mathbf{x}^0$, $\mathbf{x}^1$, etc…

B. Conjugate Directions Algorithm

Due to the high communication costs of edge computing environments, the classical Conjugate Gradient (CG) method cannot be directly used due to the need for significant synchronization when computing orthogonal direction vectors. However, we can utilize a variant called the Conjugate Directions

(CD) method by relaxing the global orthogonality constraints. The CD method, introduced by Hestenes and Stiefel in [9], is a generalization of the classical CG method. It solves the problem iteratively by computing a sequence of conjugate direction vectors. CG defines the new search direction based on a residual vector and the previously computed search directions, while the CD uses only the previous search directions. In this section, a short introduction to the CD method is given. For more details, see Hestenes and Stiefel [9].

Denote a vector $\mathbf{z} \in \mathbb{R}^m$ at iteration κ as $\mathbf{z}^\kappa$. Let $\mathbf{x}^0$ be an initial guess to the solution, then set the initial residual $\mathbf{r}^0 = \mathbf{b} - A\mathbf{x}^0 \in \mathbb{R}^m$ and select an arbitrary initial direction $\mathbf{p}^0 \in \mathbb{R}^m$. At each iteration $\kappa = 0, 1, \ldots$, the new solution approximation and the residual are computed as

$$\alpha^\kappa = \frac{\langle \mathbf{p}^\kappa,\ \mathbf{r}^\kappa \rangle}{\langle \mathbf{p}^\kappa,\ A\mathbf{p}^\kappa \rangle}, \tag{4}$$

$$\mathbf{x}^{\kappa+1} = \mathbf{x}^\kappa + \alpha^\kappa \mathbf{p}^\kappa, \tag{5}$$

$$\mathbf{r}^{\kappa+1} = \mathbf{r}^\kappa - \alpha^\kappa A\mathbf{p}^\kappa. \tag{6}$$

A new direction vector $\mathbf{p}^{\kappa+1}$ is chosen such that

$$\left\langle \mathbf{p}^{\kappa+1},\ \mathbf{p}^\iota \right\rangle_A = 0, \quad \iota = 0, \ldots, \kappa. \tag{7}$$

In the special case of the Conjugate Gradient (CG) method, we initialize the first direction vector as $\mathbf{p}^0 = \mathbf{r}^0$ and compute the subsequent direction vectors using a three-term recurrence relation, i.e.,

$$\beta^\kappa = -\frac{\|\mathbf{r}^{\kappa+1}\|_2}{\|\mathbf{r}^\kappa\|_2} = \frac{\langle \mathbf{r}^{\kappa+1},\ \mathbf{p}^\kappa \rangle_A}{\langle \mathbf{p}^\kappa,\ \mathbf{p}^\kappa \rangle_A}, \tag{8}$$

$$\mathbf{p}^{\kappa+1} = \mathbf{r}^{\kappa+1} - \beta^\kappa \mathbf{p}^\kappa. \tag{9}$$

The second formulation for β^κ in equation (8) represents the coefficient used to orthogonalize the new residual vector $\mathbf{r}^{\kappa+1}$ against the prior direction vector $\mathbf{p}^\kappa$ using Gram-Schmidt orthogonalization with the A-norm. In other words, $\mathbf{p}^{\kappa+1}$ is computed by A-orthogonalizing the new residual vector $\mathbf{r}^{\kappa+1}$ against the prior direction vector $\mathbf{p}^\kappa$. Our method combines the Conjugate Direction (CD) method globally while allowing each device to perform Conjugate Gradient (CG) steps locally. This approach achieves improved scaling compared to asynchronous Jacobi (for which some convergence is presented in [23]) without requiring the synchronization at each iteration as CG does.

C. Asynchronous s-Approximate Conjugate Directions (s-ACD)

Within the framework of the Conjugate Directions (CD) method, our objective is to design a fully asynchronous method. First, we introduce the following notation. Let $\mathbf{z}^{\psi(i,j,\kappa)} \in \mathbb{R}^m$ denote the local copy of the vector $\mathbf{z}$ from agent j received by node i at iteration κ. Let $\mathbf{z}_i \in \mathbb{R}^{m_i}$ denote the subvector of $\mathbf{z}$ corresponding to the block of elements that agent i is approximating. Each agent has access to its local portion A_i of the matrix A, the full right-hand side vector $\mathbf{b}$, and maintains a set of local variables: a local residual vector $\mathbf{r}^\kappa \in \mathbb{R}^m$, a local solution vector $\mathbf{x}^\kappa \in \mathbb{R}^m$, and a local direction vector $\mathbf{p}^\kappa \in \mathbb{R}^m$ for each iteration κ. In this context, $\kappa = 1, 2, \ldots$ represents the local iteration count of agent i. It is important to emphasize that the iteration count may vary between agents due to the asynchronous nature. Therefore, the agents may be at different stages of the iterative process at any given time.

Each agent will initialize its local vectors as follows for $\kappa = 0$:

$$\mathbf{x}^0 = \mathbf{0}, \tag{10}$$

$$\mathbf{r}^0 = \mathbf{b}, \tag{11}$$

$$\mathbf{p}^0 = \mathbf{b}. \tag{12}$$

Then, each agent will asynchronously advance from local iteration κ to $\kappa + 1$ using the following steps:

1) Compute the local matrix-vector product $\mathbf{w}^\kappa \coloneqq A_i^T \mathbf{p}_i^\kappa$, where $\mathbf{w}^\kappa \in \mathbb{R}^m$ and $\mathbf{p}_i^\kappa \in \mathbb{R}^m$ is the subvector of $\mathbf{p}^\kappa$ corresponding to the block of elements that agent i is responsible for.
2) Asynchronously send the vector $\mathbf{w}^\kappa$ from step 1 and the vector $\mathbf{p}_i^\kappa$ to the other agents.
3) Receive available updates from other agents
$$\left\{ \mathbf{w}^{\psi(i,j,\kappa)},\ \mathbf{p}_j^{\psi(i,j,\kappa)} \right\}_{j \in \mathcal{U}_i^\kappa},$$
where $\mathcal{U}_i^\kappa$ is the set of updates, i.e. $j \in \mathcal{U}_i^\kappa$ if and only if agent i during its local iteration κ received an update from agent j.

Note that for the restart mechanism introduced later, $\mathbf{r}^\kappa$ and $\mathbf{x}^\kappa$ are also sent and received during these communications. Note that the non-blocking communication allows agents to send and receive information in an asynchronous fashion, which enables parallelism while avoiding the need for synchronization at every iteration.

Once the updates have been received, each agent i will assemble the *asynchronous direction vector* $\widetilde{\mathbf{p}}^\kappa \in \mathbb{R}^m$ blockwise according to the partition:

$$\widetilde{\mathbf{p}}^\kappa = \begin{cases} \mathbf{p}_i^\kappa, & j = i; \\ \mathbf{p}_j^{\psi(i,j,\kappa)}, & j \in \mathcal{U}_i^\kappa; \\ \mathbf{0}_j, & \text{otherwise}, \end{cases}$$

where $\mathbf{p}_j^{\psi(i,j,\kappa)}$ represents the partial local direction vector agent i received from agent j at iteration κ. Since the matrix A is SPD,

$$A\mathbf{z} = A^T\mathbf{z} = \sum_{j=1}^{N} {A_j}^T \mathbf{z}_j \text{ for } \mathbf{z} \in \mathbb{R}^m.$$

Thus, the exact A matrix-vector product of the asynchronous direction vector, denoted $\widetilde{\mathbf{w}}^\kappa \coloneqq A\widetilde{\mathbf{p}}^\kappa$, can be computed using only the received and local partial matrix-vector products:

$$\begin{aligned} \widetilde{\mathbf{w}}^\kappa \coloneqq A\widetilde{\mathbf{p}}^\kappa &= A_i^T \mathbf{p}_i^\kappa + \sum_{j \in \mathcal{U}_i^\kappa} {A_j}^T \mathbf{p}_j^{\psi(i,j,\kappa)}, \\ &= \mathbf{w}^\kappa + \sum_{j \in \mathcal{U}_i^\kappa} \mathbf{w}^{\psi(i,j,\kappa)}. \end{aligned}$$

We use the asynchronous direction vector $\widetilde{\mathbf{w}}$ to construct the *s-conjugate direction vector*, denoted $\mathbf{d}^\kappa$. It is essential that $\mathbf{d}^\kappa$ is A-conjugate to the s prior s-conjugate direction vectors $\mathbf{d}^{\kappa-s-1},\dots,\mathbf{d}^{\kappa-1}$. To achieve this, we can employ a method such as Gram-Schmidt orthogonalization. In order to ensure conjugacy with prior direction vectors, additional storage of the prior s conjugate direction vectors, $\left\{\mathbf{d}^{\kappa-\ell}\right\}_{\ell=1}^{s}$, and their A-products, $\left\{\mathbf{v}^{\kappa-\ell}\right\}_{\ell=1}^{s}$, is necessary. These vectors are defined recursively, with $\mathbf{d}^0=\widetilde{\mathbf{p}}^0$, $\mathbf{v}^0=\widetilde{\mathbf{w}}^0$, and for $\kappa>0$,

$$\begin{aligned}\mathbf{d}^\kappa &= \widetilde{\mathbf{p}}^\kappa - \sum_{\ell=1}^{\min(s,\kappa)} \mathrm{GS}_A\big(\widetilde{\mathbf{p}}^\kappa,\mathbf{d}^{\kappa-\ell}\big)\mathbf{d}^{\kappa-\ell}, \qquad (13)\\ \mathbf{v}^\kappa &= A\mathbf{d}^\kappa = \widetilde{\mathbf{w}}^\kappa - \sum_{\ell=1}^{\min(s,\kappa)} \mathrm{GS}_A\big(\widetilde{\mathbf{p}}^\kappa,\mathbf{d}^{\kappa-\ell}\big)\mathbf{v}^{\kappa-\ell},\end{aligned}$$

where $\mathrm{GS}_A(\widetilde{\mathbf{p}}^\kappa,\mathbf{d}^{\kappa-\ell})$ is the magnitude of the projection of the vector $\widetilde{\mathbf{p}}^\kappa$ onto the vector $\mathbf{d}^{\kappa-\ell}$ under the A-inner product, i.e.,

$$\begin{aligned}\mathrm{GS}_A\big(\widetilde{\mathbf{p}}^\kappa,\mathbf{d}^{\kappa-\ell}\big) &\coloneqq \frac{\left\langle\widetilde{\mathbf{p}}^\kappa,\ \mathbf{d}^{\kappa-\ell}\right\rangle_A}{\left\langle\mathbf{d}^{\kappa-\ell},\ \mathbf{d}^{\kappa-\ell}\right\rangle_A},\\ &= \frac{\left\langle\widetilde{\mathbf{p}}^\kappa,\ \mathbf{v}^{\kappa-\ell}\right\rangle}{\left\langle\mathbf{d}^{\kappa-\ell},\ \mathbf{v}^{\kappa-\ell}\right\rangle}.\end{aligned}$$

Note that the exact matrix-vector product $\mathbf{v}^\kappa = A\mathbf{d}^\kappa$ is ensured due to the definition of $\widetilde{\mathbf{w}}^\kappa \coloneqq A\widetilde{\mathbf{p}}^\kappa$.

In the following theorem, we prove that $\mathbf{d}^\kappa$ is A-conjugate to the prior s conjugate direction vectors $\left\{\mathbf{d}^{\kappa-\ell-1}\right\}_{\ell=1}^{s}$.

Theorem 1. *Let $\mathbf{d}^\kappa,\mathbf{v}^\kappa$ be defined as in* (13)*. Then for $\ell=1,\dots,\min(s,\kappa)$,*

$$\left\langle\mathbf{d}^\kappa,\ \mathbf{d}^{\kappa-\ell}\right\rangle_A = 0.$$

Proof. We proceed by induction on the iteration number κ. For $\kappa=0$, the statement holds trivially. Suppose that the statement holds true for $\kappa=0,\dots,\iota-1$. We now show the statement holds true for $\kappa=\iota$.

By the definition of $\mathbf{d}^\kappa$, for $1\le\ell\le\min(s,\kappa)$

$$\begin{aligned}&\left\langle\mathbf{d}^\kappa,\ \mathbf{d}^{\kappa-\ell}\right\rangle_A\\ &= \left\langle\widetilde{\mathbf{p}}^\kappa - \sum_{\nu=1}^{\min(s,\kappa)} \mathrm{GS}_A\big(\widetilde{\mathbf{p}}^\kappa,\mathbf{d}^{\kappa-\nu}\big)\mathbf{d}^{\kappa-\nu},\ \mathbf{d}^{\kappa-\ell}\right\rangle\\ &= \left\langle\widetilde{\mathbf{p}}^\kappa,\ \mathbf{d}^{\kappa-\ell}\right\rangle_A -\\ &\sum_{\nu=1}^{\min(s,\kappa)} \mathrm{GS}_A\big(\widetilde{\mathbf{p}}^\kappa,\mathbf{d}^{\kappa-j}\big)\left\langle\mathbf{d}^{\kappa-\nu},\ \mathbf{d}^{\kappa-\ell}\right\rangle_A.\end{aligned}$$

By the induction hypothesis, $\left\langle\mathbf{d}^{\kappa-\nu},\ \mathbf{d}^{\kappa-\ell}\right\rangle_A=0$ for all $\nu=1,\dots,\min(s,\kappa)$ such that $\nu\neq\ell$. Hence,

$$\begin{aligned}&\left\langle\mathbf{d}^\kappa,\ \mathbf{d}^{\kappa-\ell}\right\rangle\\ &= \left\langle\widetilde{\mathbf{p}}^\kappa,\ \mathbf{d}^{\kappa-\ell}\right\rangle_A - \mathrm{GS}_A\big(\widetilde{\mathbf{p}}^\kappa,\mathbf{d}^{\kappa-\ell}\big)\left\langle\mathbf{d}^{\kappa-\ell},\ \mathbf{d}^{\kappa-\ell}\right\rangle_A,\\ &= \left\langle\widetilde{\mathbf{p}}^\kappa,\ \mathbf{d}^{\kappa-\ell}\right\rangle_A - \frac{\left\langle\widetilde{\mathbf{p}}^\kappa,\ \mathbf{d}^{\kappa-\ell}\right\rangle_A}{\left\langle\mathbf{d}^{\kappa-\ell},\ \mathbf{d}^{\kappa-\ell}\right\rangle_A}\left\langle\mathbf{d}^{\kappa-\ell},\ \mathbf{d}^{\kappa-\ell}\right\rangle_A,\\ &= 0.\end{aligned}$$

Thus at iteration κ, we have that $\left\langle\mathbf{d}^\kappa,\ \mathbf{d}^{\kappa-\ell}\right\rangle_A=0$ for $\ell=1,\dots,\min(s,\kappa)$. □

Using this definition of $\mathbf{d}^\kappa$, we can proceed in a manner similar to the Conjugate Directions (CD) method. Define the step size

$$\alpha^\kappa \coloneqq \frac{\left\langle\mathbf{r}^\kappa,\ \mathbf{d}^\kappa\right\rangle}{\left\langle\mathbf{d}^\kappa,\ \mathbf{v}^\kappa\right\rangle}. \qquad (14)$$

Using the step size α^κ, the approximate solution and residual vectors are updated

$$\mathbf{x}^{\kappa+1} \coloneqq \mathbf{x}^\kappa + \alpha^\kappa\mathbf{d}^\kappa, \qquad (15)$$

$$\mathbf{r}^{\kappa+1} \coloneqq \mathbf{r}^\kappa - \alpha^\kappa\mathbf{v}^\kappa. \qquad (16)$$

Finally, the next local direction vector is computed by enforcing the new residual $\mathbf{r}^{\kappa+1}$ to be s-conjugate with the s-conjugate direction vectors $\mathbf{d}^{\kappa-\ell},\ 0\le\ell\le\min(s,\kappa)-1$,

$$\mathbf{p}^{\kappa+1} \coloneqq \mathbf{r}^{\kappa+1} - \sum_{\ell=0}^{\min(s,\kappa)-1} \mathrm{GS}_A\big(\mathbf{r}^{\kappa+1},\mathbf{d}^{\kappa-\ell}\big)\mathbf{d}^{\kappa-\ell}. \qquad (17)$$

Restarting: Due to the asynchronous nature of the s-ACD algorithm, it is possible that the direction vectors, and consequently the approximate solution vectors, differ between agents at each local iteration. To address the potential stagnation that can result from such a scenario, we incorporate an asynchronous restarting procedure. By introducing these asynchronous restarts, we provide an opportunity for the agents to realign their progress, mitigate the effects of asynchronicity, and make collective advancements towards the true solution. The frequency of the restarts can be adjusted based on the specific requirements and characteristics of the problem being solved. As mentioned earlier, during the s-ACD communication stage, each agent will send its local solution vector $\mathbf{x}^\kappa$ and local residual vector $\mathbf{r}^\kappa$. This exchange allows each agent to have an updated understanding of the current state of the (global) computation, enabling more efficient choices of direction vectors for the subsequent iterations.

The algorithm is restarted periodically after detecting stagnation in the residual norm. The detection of stagnation and following restart is purely a local decision and calculation. The restarting will be performed if a specified number of iterations have passed since the last restart and the residual norm has decreased less than a prescribed tolerance. This involves resetting the necessary variables, such as solution vectors, residual vectors, and direction vectors, to a common starting point. By doing so, the agents can restart from a more unified state and resume the algorithm to overcome the

convergence stagnation. When a restart is deemed necessary, the local approximation to the asynchronous residual vector $\widetilde{\mathbf{r}}^{\kappa}$ is constructed by averaging the most recent updates $\{\mathbf{r}^{\psi(i,j,\kappa)}\}$ received from each neighbor. If no updates have been received from agent j, then set $\mathbf{r}^{\psi(i,j,0)} \coloneqq \mathbf{b}$, i.e.,

$$\widetilde{\mathbf{r}}^{\kappa} = \frac{1}{N}\sum_{j=1}^{N} \mathbf{r}^{\psi(i,j,\kappa)},$$

where $\mathbf{r}^{\psi(i,i,\kappa)} = \mathbf{r}^{\kappa}$. Then, local approximation to the asynchronous solution vector $\widetilde{\mathbf{x}}^{\kappa}$ is computed by averaging the most recently received solution vectors $\{\mathbf{x}^{\psi(i,j,\kappa)}\}$. If no updates have been received from an agent j, then set $\mathbf{x}^{\psi(i,j,0)} = \mathbf{0}$, i.e.,

$$\widetilde{\mathbf{x}}^{\kappa} = \frac{1}{N}\sum_{j=1}^{N} \mathbf{x}^{\psi(i,j,\kappa)}, \tag{18}$$

where $\mathbf{x}^{\psi(i,i,\kappa)} = \mathbf{x}^{\kappa}$. Note that

$$\begin{aligned}\mathbf{b} - A\widetilde{\mathbf{x}}^{\kappa} &= \mathbf{b} - \frac{1}{N}\sum_{j=1}^{N} A\mathbf{x}^{\psi(i,j,\kappa)},\\ &= \frac{1}{N}\sum_{j=1}^{N}\Big(\mathbf{b} - A\mathbf{x}^{\psi(i,j,\kappa)}\Big),\\ &= \frac{1}{N}\sum_{j=1}^{N}\mathbf{r}^{\psi(i,j,\kappa)},\\ &= \widetilde{\mathbf{r}}^{\kappa}.\end{aligned}$$

Thus, the restarting procedure maintains the accuracy of the residual. Additionally, we found empirically that explicitly recomputing the local partial residual as

$$\widetilde{\mathbf{r}}_i^{\kappa} = \mathbf{b}_i - A_i\widetilde{\mathbf{x}}^{\kappa}$$

generally improves convergence and enables convergence for some non-symmetric matrices A. Modifications to (13) and (17) are required to account for the possibility that the s prior s-conjugate direction vectors may no longer be consistent after a restart. To address this issue, we introduce the set $\mathcal{S}$, which represents the subset of "active" s-conjugate direction vectors. After a restart, this set is reset to $\mathcal{S} = \emptyset$. After each iteration, we update $\mathcal{S}$ by taking the union of the set with the newly computed direction vector $\mathbf{d}^{\kappa}$, i.e., $\mathcal{S} = \mathcal{S} \cup \{\mathbf{d}^{\kappa}\}$. Only the vectors within the set $\mathcal{S}$ can be used in the s-step orthogonalization process. Thus, (13) and (17) need to be modified accordingly, i.e.,

$$\begin{aligned}\mathbf{d}^{\kappa} &= \widetilde{\mathbf{p}}^{\kappa} - \sum_{\ell=1}^{\min(s,|\mathcal{S}|)} \mathrm{GS}_A\big(\widetilde{\mathbf{p}}^{\kappa}, \mathbf{d}^{\kappa-\ell}\big)\mathbf{d}^{\kappa-\ell}, \qquad (19)\\ \mathbf{v}^{\kappa} &= A\mathbf{d}^{\kappa} = \widetilde{\mathbf{w}}^{\kappa} - \sum_{\ell=1}^{\min(s,|\mathcal{S}|)} \mathrm{GS}_A\big(\widetilde{\mathbf{p}}^{\kappa}, \mathbf{d}^{\kappa-\ell}\big)\mathbf{v}^{\kappa-\ell},\end{aligned}$$

and

$$\mathbf{p}^{\kappa+1} \coloneqq \mathbf{r}^{\kappa+1} - \sum_{\ell=0}^{\min(s,|\mathcal{S}|-1)} \mathrm{GS}_A\big(\mathbf{r}^{\kappa+1}, \mathbf{d}^{\kappa-\ell}\big)\mathbf{d}^{\kappa-\ell}. \tag{20}$$

To complete the restarting procedure, all the agents set their local vectors accordingly: $\mathbf{x}^{\kappa} = \widetilde{\mathbf{x}}^{\kappa}$, $\mathbf{p}^{\kappa} = \widetilde{\mathbf{r}}^{\kappa}$, and $\mathbf{r}^{\kappa} = \widetilde{\mathbf{r}}^{\kappa}$.

IV. Data Errors and Corruption Modeling

As increased parallelism and new environments are considered, the likelihood of errors increases and so too does the costs associated with data errors. Practical Krylov methods introduce restarts to handle errors introduced through finite precision calculations. However, the restarts alone are not enough when larger or more discrete data errors happen, the methods can lose the underlying subspace and orthogonality properties that they rely on. Thus, additional resiliency measures must be considered in environments where such large disruptions are expected. Even the s-ACD method above, which has the self-correcting restart mechanism, is still susceptible to errors introduced through numerous pathways including malicious injection, disconnection of agents, corruption introduced into the signal, delays in communication, agents entering a failed state, bit-errors introduced locally, etc. We focus on data corruption where the original data being communicated at a single iteration is replaced by other values.

One important note is that we only corrupt one vector at a time. Because the $\mathbf{x}$ and $\mathbf{r}$ vectors are only used during the restart, to accurately model the errors, we must force a restart after a corruption occurs, otherwise corruptions in $\mathbf{x}$ and $\mathbf{r}$ may be masked. To understand this, imagine we corrupt a transmission of vector $\mathbf{x}$ at iteration ι, and we do not force a restart. The receiving agent receives this corrupted $\mathbf{x}$ at iteration κ, stores it, but it does not use it in the calculation because a restart does not happen. However, at iteration $\kappa' > \kappa$ when a restart occurs, the transmitted vector $\mathbf{x}$ from iteration κ' is stored (overwriting the previous corrupted vector with an uncorrupted vector) and is used during the restart. Thus, the corrupted $\mathbf{x}$ that was stored at iteration κ was overwritten and never used, leading to the corruption being masked.

The failure model we consider is that of *one-off* corruption, meaning that at some point in time, one or more agents all have corruption applied to one of the vectors transmitted during that iteration. This failure model has been chosen for multiple reasons:

- such corruption clearly partitions time into a "before" and "after" corruption portions, where the "before" portion should be identical to the uncorrupted case,
- one can easily visually identify where the corruption occurs,
- it is simple to implement,
- it forms the basis for other forms of corruption and can relatively easily be generalized to the other forms.

V. Resilient s-ACD

While the s-ACD method has the self-correcting restart mechanism that allows it to be resilient to the presence of non-orthogonal directions introduced by the asynchronous approach (and somewhat resilient to other data errors), additional resiliency measures would be able to decrease the impact of other data errors. In this section, we introduce resiliency in

Algorithm 1 s-Approximate Asynchronous Conjugate Directions.

for node $i \leftarrow 1$ **to** N **do**
  INPUT: global vector $\mathbf{b}$, local portion A_i of the A matrix
  OUTPUT: Each node i has a local approximation $\mathbf{x}^\kappa$ to the solution vector $\mathbf{x}$
  for $j \leftarrow 1$ **to** $N, j \neq i$ **do**
    initialize $\mathbf{x}_j = \mathbf{0}$, $\mathbf{r}_j = \mathbf{b}_j, \mathbf{p}_j = \mathbf{b}_j$
  end for
  for $\kappa \leftarrow 0$ **to** $t_{\texttt{max}}$ **do**
    $\mathbf{w}^\kappa \leftarrow A_i^T \mathbf{p}^\kappa$
    Send $\mathbf{w}^\kappa$, $\mathbf{p}_i^\kappa$, $\mathbf{x}^\kappa$, $\mathbf{r}^\kappa$
    $\left\{\mathbf{w}^{\psi(i,j,\kappa)}, \mathbf{p}_j^{\psi(i,j,\kappa)}, \mathbf{x}^{\psi(i,j,\kappa)}, \mathbf{r}^{\psi(i,j,\kappa)}\right\}_{j \in \mathcal{U}_i^\kappa} \leftarrow$ `ReceiveAsync`
    Set $\mathcal{U}_i^\kappa :=$ set of node indices from which updates were received
    $\widetilde{\mathbf{p}}_i^\kappa = \mathbf{p}_i^\kappa$
    $\widetilde{\mathbf{w}}^\kappa = \mathbf{w}^\kappa$
    for $j \in \mathcal{U}_i^\kappa$ **do**
      $\widetilde{\mathbf{p}}_j^\kappa = \mathbf{p}_j^{\psi(i,j,\kappa)}$
      $\widetilde{\mathbf{w}}^\kappa = \widetilde{\mathbf{w}}^\kappa + \mathbf{w}^{\psi(i,j,\kappa)}$
      $\mathbf{x}^{\psi(i,j,\ell)} = \mathbf{x}^{\psi(i,j,\kappa)}$
      $\mathbf{x}^{\psi(i,j,\ell)} = \mathbf{r}^{\psi(i,j,\kappa)}$
    end for
    $\mathbf{d}^\kappa = \widetilde{\mathbf{p}}^\kappa - \sum_{\nu=1}^{\min(s,|\mathcal{S}|)} \mathrm{GS}_A(\widetilde{\mathbf{p}}^\kappa, \mathbf{d}^{\kappa-\nu})\mathbf{d}^{\kappa-\nu}$
    $\mathbf{v}^\kappa = \widetilde{\mathbf{w}}^\kappa - \sum_{\nu=1}^{\min(s,|\mathcal{S}|)} \mathrm{GS}_A(\widetilde{\mathbf{p}}^\kappa, \mathbf{d}^{\kappa-\nu})\mathbf{v}^{\kappa-\nu}$
    $\alpha^\kappa = \langle \mathbf{r}^\kappa, \mathbf{d}^\kappa \rangle / \langle \mathbf{d}^\kappa, \mathbf{v}^\kappa \rangle$
    $\mathbf{x}^{\kappa+1} = \mathbf{x}^\kappa + \alpha^\kappa \mathbf{d}^\kappa$
    $\mathbf{r}^{\kappa+1} = \mathbf{r}^\kappa - \alpha^\kappa \mathbf{v}^\kappa$

$$\mathbf{p}^{\kappa+1} = \mathbf{r}_i^{\kappa+1} - \sum_{\nu=0}^{\min(s,|\mathcal{S}|-1)} \mathrm{GS}_A(\mathbf{r}^{\kappa+1}, \mathbf{d}^{\kappa-\nu})\mathbf{d}^{\kappa-\nu} \tag{21}$$

    if $\|\mathbf{r}^{\kappa+1}\|/\|\mathbf{b}\| < \epsilon$ **then**
      return $\mathbf{x}^{\kappa+1}$
    end if
    if ShouldRestart($\mathbf{r}^{\kappa+1}$, κ) **then**
      $\mathbf{x}^\kappa = \frac{1}{N}\left(\mathbf{x}^\kappa + \sum_{j=0, j\neq i}^{N} \mathbf{x}^{\psi(i,j,\ell)}\right)$
      $\mathbf{r}^\kappa = \frac{1}{N}\left(\mathbf{r}^\kappa + \sum_{j=0, j\neq i}^{N} \mathbf{x}^{\psi(i,j,\ell)}\right)$
      $\mathbf{r}_i^\kappa = \mathbf{b}_i - A_i \mathbf{x}^\kappa$
      $\mathbf{p}^\kappa = \mathbf{r}^\kappa$
      $\mathcal{S} = \emptyset$
    end if
  end for
end for

two stages: the detection stage and the correction stage. The benefit of this approach is to separate the tasks of identifying deviations from "normal" behavior and the ability to correct said behavior.

A. Detection Stage

To detect data errors, we need to be able to identify when the method is in a state that is not "normal." In CG, this can be done through the orthogonality conditions or monotonically decreasing quantities — which if violated indicate that something is not as expected. However, in order to achieve asynchronicity, s-ACD loses the orthogonality conditions. Instead, we can develop different detection schemes leveraging knowledge from CG.

1) Checksum: The checksum method is based off on idea of checksums, i.e., calculating a quantity locally, transmitting it with the information, and checking that the received quantity, when recalculated locally, is consistent. One simple way of doing this is using the inner-product of two vectors. The downside of this method is that it relies on a trustworthy sender (the sender can adjust both the checksum and sent vectors accordingly) and requires additional communication. However, it comes with a number of pros, e.g., it requires only a small amount of local computation (perhaps some that is being performed anyway), provides per-agent detection, puts constraints on the possible malicious vectors that can be used, and is cheap and easy to implement. In particular, we use $\mathbf{p}^T\mathbf{w}$, where only the local portions of both vectors are used (as only the local portion of $\mathbf{p}$ is sent and available to check with). This is done by calculating the checksum before an agent sends its vectors (Alg. 2), and comparing the received value against the recalculated value at the arrival (Alg. 3).

Algorithm 2 Checksum calculation before sending.

$\gamma^\kappa \leftarrow \mathbf{w}_i^{\kappa T} \mathbf{p}_i^\kappa$
Send $\mathbf{w}^\kappa$, $\mathbf{p}_i^\kappa$, $\mathbf{x}^\kappa$, $\mathbf{r}^\kappa$, γ^κ

Algorithm 3 Checksum calculation after receiving.

for $j \in \mathcal{U}_k^\kappa$ **do**
  if $\gamma^\kappa == \mathbf{w}^{\psi(i,j,\kappa)^T} \mathbf{p}_j^{\psi(i,j,\kappa)}$ **then**
    $\widetilde{\mathbf{p}}_j^\kappa = \mathbf{p}_j^{\psi(i,j,\kappa)}$
    $\widetilde{\mathbf{w}}^\kappa = \widetilde{\mathbf{w}}^\kappa + \mathbf{w}^{\psi(i,j,\kappa)}$
    $\mathbf{x}^{\psi(i,j,\ell)} = \mathbf{x}^{\psi(i,j,\kappa)}$
    $\mathbf{x}^{\psi(i,j,\ell)} = \mathbf{r}^{\psi(i,j,\kappa)}$
  else
    Mark update κ from agent j as corrupted
  end if
end for

2) General: As mentioned previously, one way of detecting corruption is to detect when the result from a calculation is different from what it should be. One can use "metrics" (also called *indicator variables*), which are simple scalars that change over time, and see when they change in unexpected

ways. This allows us to adapt to different convergence speeds and parts of the convergence without relying too heavily on tunable parameters. While we lose precise orthogonality conditions and monotonically decreasing quantities in s-ACD, we do still have some relations that are roughly predictable. For example, as iterations progress, $\langle \mathbf{d}^\kappa,\ \mathbf{v}^\kappa \rangle$, $\langle \mathbf{p}^\kappa,\ \mathbf{p}^\kappa \rangle_A$, and $\langle \mathbf{r}^\kappa,\ \mathbf{r}^\kappa \rangle$ all tend to decrease. Thus, we can monitor these values and determine when they increase between successive iterations more than expected.

Because there is significant variation of the metrics over the course of the solve, we should not look directly at the successive difference of a timeseries metric ξ between iterations. Instead, we apply a smoothing step, take the difference between the smoothed values, and compare that against a smoothed version of the difference of smoothed values. When the ratio of these quantities gets above a specified value (which tends to be quite robust), we mark this iteration as corrupted.

To perform the smoothing we use a running average with a window size σ of 15 iterations, which allows us to perform these calculations online. If we let ξ be a $\mathbb{R}^\kappa$ timeseries with ξ^i being the value at point i in time, then we define the smoothed timeseries as

$$\text{smooth}(\xi, \sigma, i) = \frac{\sum_{j=\max(0,i-\sigma)}^{i} \xi^j}{\min(i, \sigma)}.$$

We define the relative successive difference to be

$$\text{diff}(\xi, i) = \frac{\xi^{i+1} - \xi^i}{|\xi^i|}.$$

We consider a timeseries ξ to be corrupted at time i if $\text{smooth}(\text{diff}(\text{smooth}(\xi, size, i), i), size, i) > \epsilon$ for some tolerance $\epsilon > 0$.

In particular, we track the timeseries defined by $\langle \mathbf{v}^\kappa,\ \mathbf{d}^\kappa \rangle$ and $\langle \mathbf{r}^\kappa,\ \mathbf{r}^\kappa \rangle$. The biggest drawbacks of this method are that it does not provide per-agent detection and introduces additional computational steps. However, it is easily generalizable to other methods, requires only local computation, doesn't rely on a trustworthy sender, and the computational complexity can be mitigated by updating the differences and smoothed timeseries at each time step rather than recalculating the entire timeseries.

3) Algorithm-Based: The final class of detection methods that we will discuss are the algorithm-based metrics. These rely on knowing specific analytical relations within the calculations, such as the orthogonality conditions in CG. Although we do not have the orthogonality conditions, we do know that, $A\mathbf{x} = \mathbf{b}$ and $\mathbf{r} = \mathbf{b} - A\mathbf{x}$. There are many solution approximations and associated residuals that can be calculated, all of which should be similar to each other. Thus, if one of these vectors differs significantly from the others (or what is expected via explicit calculation of the residual), this indicates that a corruption is likely to have occurred. We compare the incoming solution and residual vectors from other agents against the updated value of the currently considered solution and residual, as well as the most recent solution and residual that have been checked by the detection mechanisms and identified as uncorrupted.

Due to the many comparisons and matrix-vector products involved, this is a computationally expensive check. However, it doesn't require any additional communication and provides per agent detection. Furthermore, the adaptive properties of the general metrics can be explored, although they are not currently used in our implementation (Alg 4).

Algorithm 4 Algorithm-based check for agent i, which compares the proposed update when applied to the incoming solution and residual vectors from agent j, with baseline updated solution and residual vectors.

for j=1, …, N **do**
  $\mathcal{X}_j := \mathbf{x}^{\psi(i,j,\kappa)} + \alpha^\kappa \mathbf{d}^\kappa$
  $\mathcal{R}^{exp} := \mathbf{b}_i - A_i \mathcal{X}_j$
  $\mathcal{R}^{iter} := \mathbf{r}^{\psi(i,j,\kappa)} + \alpha^\kappa \mathbf{d}^\kappa$
  $\mathbf{x}^S :=$ the last $\mathbf{x}^{\psi(i,j,\iota)}$ vector considered to be uncorrupted
  $\mathbf{r}^S :=$ the last $\mathbf{x}^{\psi(i,j,\iota)}$ vector considered to be uncorrupted
  for $(x, r) \in ((\mathbf{x}^\kappa, \mathbf{r}^\kappa), (\mathbf{x}^S, \mathbf{r}^S))$ **do**
    $\mathfrak{X}_j := x + \alpha^\kappa \mathbf{d}^\kappa$
    $\mathfrak{R}^{exp} := \mathbf{b}_i - A_i \mathfrak{X}_j$
    $\mathfrak{R}^{iter} := r - \alpha^\kappa \mathbf{v}^\kappa$
    if $(\frac{||\mathfrak{X}_j|| - ||\mathcal{X}_j||}{min(||\mathfrak{X}_j||, ||\mathcal{X}_j||)} > \epsilon_1)$ or $(\frac{||\mathfrak{R}^{exp}|| - ||\mathcal{R}^{exp}||}{min(||\mathfrak{R}^{exp}||, ||\mathcal{R}^{exp}||)} > \epsilon_2)$ or $(\frac{||\mathfrak{R}^{iter}|| - ||\mathcal{R}^{iter}||}{min(||\mathfrak{R}^{iter}||, ||\mathcal{R}^{iter}||)} > \epsilon_3)$ **then**
      Agent i considers agent j's communication at iteration ι as corrupted.
    end if
  end for
end for

B. Correction Phase

Once a transmission has been identified to contain a data error, a correction must be performed. Under traditional methods, this might require restarting the entire computation [28], however, we are able to utilize a more nuanced approach which reduces the amount of redundant calculations. We introduce a simple rejection approach, which performs well for the class of investigated data errors. The simplest form of correction is to ignore the updates associated with an iteration that has been flagged as corrupted. If a specific agent has been identified as the source of the corruption, it is possible to ignore the updates from that agent only.

VI. Numerical Experiments

To demonstrate the performance of the newly proposed s-ACD and resiliency methods, a number of numerical experiments are conducted. These experiments are performed on a MacBook Pro (2019) with a 2.4GHz 8-Core Intel Core i9 CPU, and 64GB of 2667 MHz DDR4 Memory. Unless otherwise stated, experiments are run with four agents. When a 2D Poisson problem is used, it has homogeneous Dirichlet boundary conditions and is discretized with first order central finite

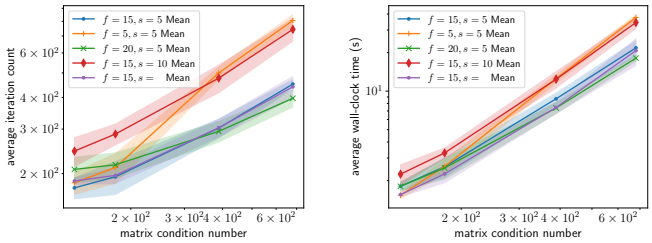
differences with a right-hand side $g(x, y) = \sin(\pi x)\sin(\pi y)$ for a point at location (x, y). When a random SPD matrix is used, it has a condition number of 50, with a right-hand side defined by a vector of all ones.

Fig. 1 displays a variety of restarting frequencies f and s-step sizes s when using s-ACD over five runs, where the mean of the runs is plotted in a solid line and the 95% confidence interval displayed in the corresponding shaded region. We can see that having a large s-step size s relative to restart size f, such as the cases of $f = 15, s = 10$ and $f = 5, s = 5$ results in both a higher iteration count and a slower convergence. This is likely due to when the s-step size and restart sizes are similar, the number of full s sized s-step updates performed is small. There was not a significant difference in the other tested combinations. For the following tests, $f = 15$ and $s = 5$.

Fig. 2 demonstrates the scaling of the s-ACD method against the scaling of the asynchronous Jacobi and serial CG methods when changing the number of rows (and hence the condition number) of a 2D Poisson matrix with five runs, with the means plotted with solid lines and the 95% confidence interval plotted in the shaded region. We can see that our s-ACD method achieves better asymptotic scaling than the asynchronous Jacobi method, although not as good as serial CG, especially for larger condition numbers. The scaling achieved on systems with condition numbers in the range of 10^2 to 10^3 is comparable with CG, with a significant improvement in the absolute number of iterations compared to asynchronous Jacobi. The development of convergence theory could be used to better understand the results seen, and asynchronous preconditioning can improve the convergence further.

A 100×100 random SPD problem is used for Fig. 3, where different vectors are corrupted during communication, and the resulting l^2-norms are plotted for each agent, demonstrating the impact of these data errors on the metrics. The metrics are very noisy, due to the loss of orthogonality and independent nature of each agent. We see spikes around each restart, as the local residual and solution vectors are changed, potentially significantly, compared to the previous iteration. We observe that in all cases, the convergence slows down after the corruption at one second, while in the $\mathbf{x}, \mathbf{p}, \mathbf{r}$ cases, there is a significant spike in the observed metrics. This is because corrupting the $\mathbf{x}, \mathbf{r}, \mathbf{p}$ vectors (directly or indirectly) permanently destabilizes the subspace and has immediate consequences, while the $\mathbf{w}$ vector is recalculated at each iteration from the $\mathbf{p}$ vector.

The varying steps of the "generic" correction scheme are demonstrated in Fig. 4, which is a random SPD problem with a data error occurring after one second, separating out the global post-processed metrics from the metrics visible to each agent (Fig. 4a). We see that the global post-processed metric is very smooth, ignoring the spike at each restart, indicating that the overall behavior of the agents approximates is similar to the behavior of CG. We observe that the local metrics do correlate strongly with the trend of the global post-processed metrics, indicating that they can be used as a proxy for the overall convergence. While in synchronous CG, we would expect these metrics to be monotonically decreasing, the asynchronous algorithm removes these guarantees. Thus, the procedure of smoothing and successive differences must be used and through this adaptive method is able to detect when the corruption happens via local computations. It is clear that after these procedures (Fig. 4e) the aberration can be easily detected.



(a) Comparing by iteration count. (b) Comparing by time.

Fig. 1: Comparing the scaling of different restarting frequencies f and s-step sizes s for s-ACD on a 2D Poisson problem discretized with finite differences with four agents and five runs proceeding until a tolerance of 1e-5 is reached. The shaded region represent a 95% confidence interval.

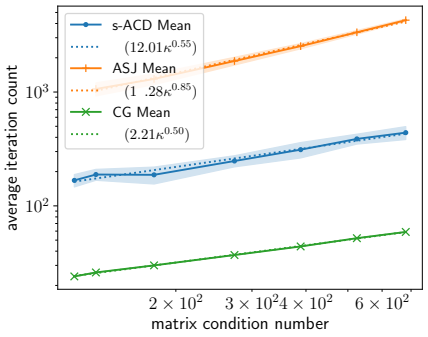
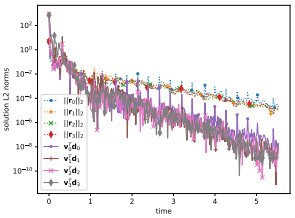
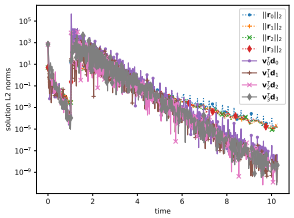


Fig. 2: Scaling of s-ACD vs asynchronous Jacobi (ASJ) and serial CG method on a 2D Poisson problem discretized with finite differences with four agents and five runs proceeding until a tolerance of 1e-5 is reached. The shaded region represents the 95% confidence interval.

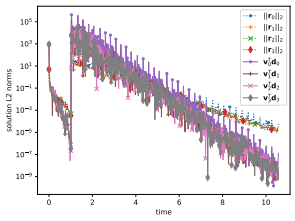
Finally, Fig. 5 shows the residual of a 100×100 random SPD problem for four agents for the s-ACD method with and without the resiliency measures, for 30 runs with the mean of runs plotted as the solid line, while the dotted lines correspond to individual runs and the shaded region to the 95% confidence interval. By enabling all three sets of resiliency measure discussed above, the data errors are successfully detected and corrected. We can see that adding the resiliency measures annihilates the impact of the corruption, leading to four times faster convergence than without resiliency measures.



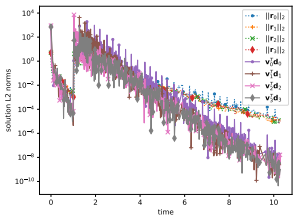
(a) Corruption in $\mathbf{w}$



(b) Corruption in $\mathbf{r}$



(c) Corruption in $\mathbf{x}$

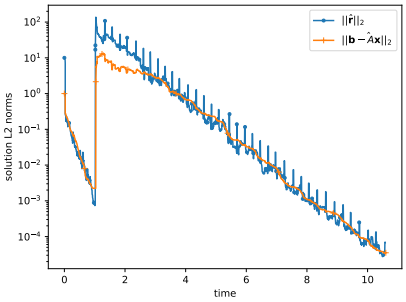


(d) Corruption in $\mathbf{p}$

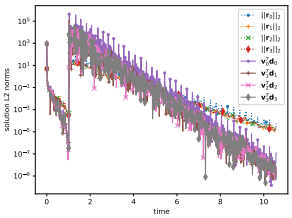
Fig. 3: Demonstrating the impact of errors introduced into different vectors of s-ACD after one second into all agents, where the metric of each agent is displayed. The problem considered is a 100×100 random SPD problem with condition number 50.

VII. Conclusion

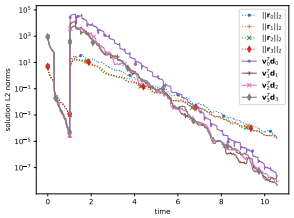
We have seen that due to the increased parallelism and new computational paradigms, asynchronous and resilient methods should be developed. In this paper, we have developed the s-ACD method that combines the CD method globally with the CG method locally. This provides scaling with respect to the condition number comparable with CG on the tested 2D Poisson problem, while ensuring complete asynchronicity, as global orthogonalization is no longer required, as well as some resiliency. Furthermore, we developed three detection techniques: a "generic" detection scheme, a "checksum" detection scheme, and an algorithm-based detection scheme. These methods were applied to s-ACD, creating the resilient s-ACD method. Numerical experiments were performed to demonstrate that this resilient s-ACD method is able to handle the introduction of data errors into the communication pattern, resulting in a significant decrease of iterations compared to the uncorrupted case. Future improvements include developing theory for the s-ACD method and resilient variation, adding more elaborate correction methods such as rollback, as well as developing asynchronous preconditioners to allow the considered methods to scale to larger problems.



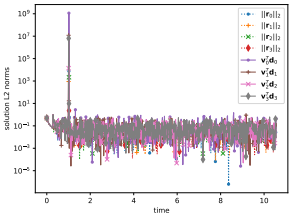
(a) Global metrics via post-processing.



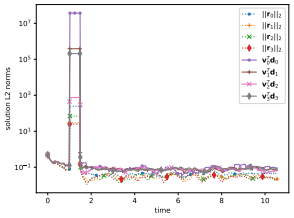
(b) Metrics



(c) Smoothed Metrics



(d) Difference of Smoothed Metrics



(e) Smoothed Differences of Smoothed Metrics

Fig. 4: Different stages of the post-processing pipeline applied to the metrics when an error is introduced after one second into all agents for the s-ACD method (without resiliency measures). The problem considered is a 100×100 random SPD with condition number 50.

References

[1] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[2] C. T. Kelley, *Iterative methods for linear and nonlinear equations*. SIAM, 1995.

[3] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct methods for sparse matrices*. Oxford University Press, 2017.

[4] T. A. Davis, *Direct methods for sparse linear systems*. SIAM, 2006.

[5] J. J. Dongarra, I. S. Duff, D. C. Sorensen, H. A. Van der Vorst, and others, *Solving linear systems on vector and shared memory computers*. SIAM Philadelphia, 1991.

[6] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. Van der Vorst, *Numerical linear algebra for high-performance computers*. SIAM, 1998.

[7] Y. Saad, "Krylov subspace methods on supercomputers," *SIAM Journal on Scientific and Statistical Computing*, vol. 10, no. 6, pp. 1200–1232, 1989. doi: https://doi.org/10.1137/0910073

[8] C. Lanczos, "Solution of systems of linear equations by minimized iterations," *J. Res. Nat. Bur. Standards*, vol. 49, no. 1, pp. 33–53, 1952.

[9] M. R. Hestenes, E. Stiefel, and others, "Methods of conjugate gradients for solving linear systems," *Journal of Research of the National Bureau of Standards*, vol. 49, no. 6, pp. 409–436, 1952.

[10] C. Ponce, K. Harter, A. Fox, and C. Vogl, "Skywing," [Computer Software] https://doi.org/10.11578/dc.20221110.2, nov 2022. [Online]. Available: https://doi.org/10.11578/dc.20221110.2

[11] E. C. Carson, "An adaptive s-step conjugate gradient algorithm with dynamic basis updating," *Applications of Mathematics*, vol. 65, no. 2, pp. 123–151, 2020. doi: https://doi.org/10.21136/AM.2020.0136-19

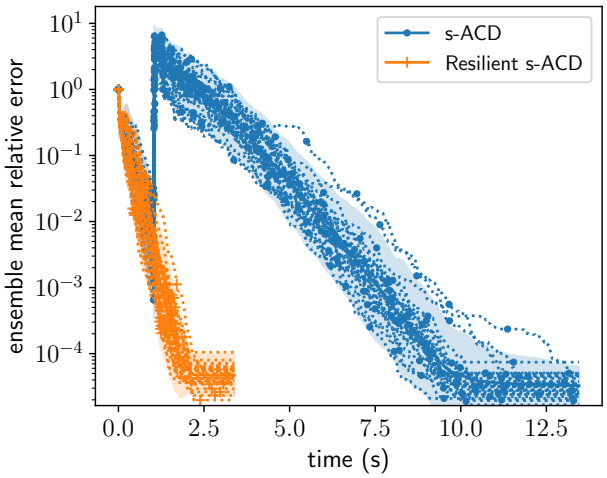


Fig. 5: Comparing the convergence of s-ACD ($f = 15, s = 5$) with and without correction where the $\mathbf{x}$ vector is corrupted on all four agents after one second for a random 100×100 SPD matrix with condition number 50 until a tolerance of 1e-5 is reached, averaged over 30 runs with the 95% confidence interval shown in the shaded region. The tail seen at the end is due to the agents ensuring that global convergence has been reached.

[12] A. Chronopoulos and C. Gear, "s-step iterative methods for symmetric linear systems," *Journal of Computational and Applied Mathematics*, vol. 25, no. 2, pp. 153–168, 1989. doi: https://doi.org/10.1016/0377-0427(89)90045-9. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0377042789900459

[13] S. Cools, J. Cornelis, P. Ghysels, and W. Vanroose, "Improving strong scaling of the conjugate gradient method for solving large linear systems using global reduction pipelining," *arXiv preprint arXiv:1905.06850*, 2019. doi: https://doi.org/10.48550/arXiv.1905.06850

[14] P. R. Eller and W. Gropp, "Scalable non-blocking preconditioned conjugate gradient methods," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016. doi: https://doi.org/10.1109/SC.2016.17 pp. 204–215.

[15] M. Tiwari and S. Vadhiyar, "Pipelined Preconditioned s-step Conjugate Gradient Methods for Distributed Memory Systems," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021. doi: https://doi.org/10.1109/Cluster48925.2021.00061 pp. 215–225.

[16] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, "Fault tolerant preconditioned conjugate gradient for sparse linear system solution," in *Proceedings of the 26th ACM international conference on Supercomputing*, 2012. doi: https://doi.org/10.1145/2304576.2304588 pp. 69–78.

[17] M. E. Ozturk, M. Renardy, Y. Li, G. Agrawal, and C.-S. Chou, "A Novel Approach for Handling Soft Error in Conjugate Gradients," in *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, 2018. doi: 10.1109/HiPC.2018.00030 pp. 193–202.

[18] A. Schöll, C. Braun, M. A. Kochte, and H.-J. Wunderlich, "Efficient online fault-tolerance for the preconditioned conjugate gradient method," in *2015 IEEE 21st International On-Line Testing Symposium (IOLTS)*, 2015. doi: 10.1109/IOLTS.2015.7229839 pp. 95–100.

[19] D. Chazan and W. Miranker, "Chaotic relaxation," *Linear algebra and its applications*, vol. 2, no. 2, pp. 199–222, 1969. doi: https://doi.org/10.1016/0024-3795(69)90028-7

[20] A. Frommer and D. B. Szyld, "On asynchronous iterations," *Journal of computational and applied mathematics*, vol. 123, no. 1-2, pp. 201–216, 2000. doi: https://doi.org/10.1016/S0377-0427(00)00409-X

[21] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, *Parallel iterative algorithms: from sequential to grid computing*. CRC Press, 2007.

[22] J. Hook and N. Dingle, "Performance analysis of asynchronous parallel Jacobi," *Numerical Algorithms*, vol. 77, pp. 831–866, 2018. doi: https://doi.org/10.1007/s11075-017-0342-9

[23] J. Wolfson-Pou and E. Chow, "Convergence models and surprising results for the asynchronous Jacobi method," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018. doi: 10.1109/IPDPS.2018.00103 pp. 940–949.

[24] ——, "Modeling the asynchronous Jacobi method without communication delays," *Journal of Parallel and Distributed Computing*, vol. 128, pp. 84–98, 2019. doi: https://doi.org/10.1016/j.jpdc.2019.02.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731518304751

[25] H. Anzt, J. Dongarra, and E. S. Quintana-Ortí, "Tuning stationary iterative solvers for fault resilience," in *Proceedings of the 6th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, 2015. doi: https://doi.org/10.1145/2832080.2832081 pp. 1–8.

[26] Apache Software Foundation, "Hadoop," 2021, version Number: 3.3.03. [Online]. Available: https://hadoop.apache.org

[27] ——, "Spark," 2021, version Number: 3.3.0. [Online]. Available: https://spark.apache.org

[28] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010. doi: https://doi.org/10.1109/SC.2010.18 pp. 1–11.