The Algorithmic Development of a Fully Asynchronous Conjugate Gradient Method

Zachary Atkins – atkins12@llnl.gov
Alyson Fox – fox33@llnl.gov
Agnieszka Międlar – amiedlar@ku.edu
Colin Ponce – ponce11@llnl.gov
Christopher Vogl – vogl2@llnl.gov

April 4, 2022

LLNL-PRES-833329

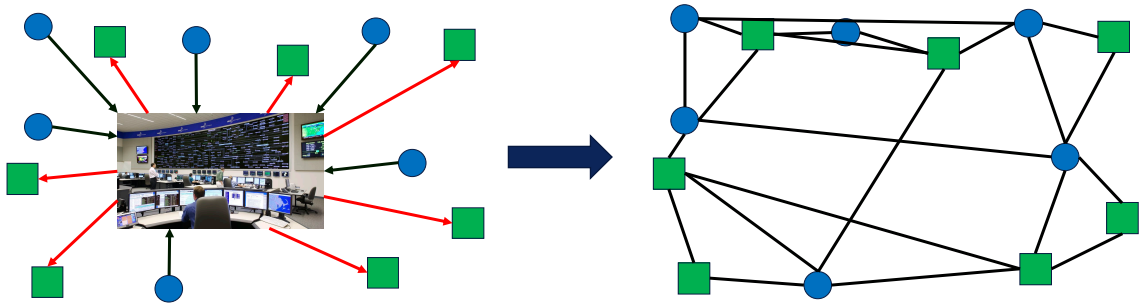
Lawrence Livermore National Laboratory

About Me

- BS in Computer Science and Mathematics, University of Kansas
- Yearlong appointment at Lawrence Livermore National Laboratory
- Starting CS Ph.D. program at CU Boulder in Fall 2022

Motivation

- Modern decentralized computing environments
  - Spatially distributed
  - Low-power and/or heterogenous computational units (nodes)
- Lack of iterative solvers with desirable properties
  - Data locality
  - Asynchronous
  - Avoid global (all-to-all) communication
    - i.e. no dot products or MPI allreduce



Motivation

- Asynchronous Jacobi (ASJ) is too slow
  - Scales poorly for ill-conditioned matrices
  - Requires large number of iterations for even low accuracy
- Conjugate gradient (CG) is extremely fast
  - Strong convergence guarantees
  - Widely used in HPC
- But, CG requires synchronous all-to-all communication
  - We will try to remove this and make an asynchronous algorithm

Classical CG Method

Formulation (Hestenes & Stiefel, 1952)

- Solve the linear system $A\mathbf{x} = \mathbf{b}$, A symmetric positive-definite (SPD)
- Initialize $\mathbf{p}^{(0)} = \mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$, $\mathbf{x}^{(0)}$ arbitrary
- For $t = 0,1,2,\ldots$

1. Compute $\alpha^{(t)} = \dfrac{\|\mathbf{r}^{(t)}\|^2}{\langle \mathbf{p}^{(t)}, A\mathbf{p}^{(t)} \rangle}$
2. Update $\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} + \alpha^{(t)}\mathbf{p}^{(t)}$
3. Update $\mathbf{r}^{(t+1)} = \mathbf{r}^{(k)} - \alpha^{(t)}A\mathbf{p}^{(t)}$
4. Compute $\beta^{(t)} = \dfrac{\|\mathbf{r}^{(t+1)}\|^2}{\|\mathbf{r}^{(t)}\|^2}$
5. Update $\mathbf{p}^{(t+1)} = \mathbf{r}^{(t+1)} + \beta^{(t)}\mathbf{p}^{(t)}$

Matrix vector product and dot product

Dot product

- Need to compute a matrix-vector product and two dot products at each iteration

Classical CG Method

Distributed Memory Implementation

- Given an SPD, likely sparse, matrix $A \in \mathbb{R}^{m \times m}$ and vector $\mathbf{b} \in \mathbb{R}^m$
- Distribute rows of A over N agents by the partition $\mathcal{P} = \bigcup_{k=1}^{N} \mathcal{P}_k$
- Notation :
  - Denote by $A_{[k]}$ the $n_k := |\mathcal{P}_k|$ rows local to agent k
  - Denote by $\mathbf{v}_{[k]} \in \mathbb{R}^{n_k}$ the part of the vector $\mathbf{v} \in \mathbb{R}^m$ corresponding to the partition $\mathcal{P}_k$
  - Denote by $\mathbf{v}_k \in \mathbb{R}^m$ the copy of the global vector $\mathbf{v} \in \mathbb{R}^m$ on agent k
- Store local parts of all vectors and matrix: $\mathbf{x}_{[k]}^{(t)}$, $\mathbf{r}_{[k]}^{(t)}$, $\mathbf{p}_{[k]}^{(t)}$, and $A_{[k]}$
- Compute dot products using all-reduce and matrix-vector product using neighbor communication

Reducing the Impact of Communication in CG

Existing Approaches

- Pipelined CG
  - Communication hiding
    - Ghysels & Vanroose, 2014
    - Eller & Gropp, 2016
- s-step CG
  - Communication avoiding
    - Chronopoulos & Gear, 1989
    - Carson, 2020
- Pipelined s-step CG
  - Both communication avoiding and hiding
    - Tiwari & Vadhiyar, 2021
- Not truly asynchronous

Toward Asynchronous CG

CG as a Conjugate Directions Method

- What is a conjugate directions method (CD-method)?
  - Direction vectors $\mathbf{p}^{(t)}$ are chosen to be mutually conjugate (A-orthogonal)
  - That is, $\left\langle \mathbf{p}^{(t)}, \mathbf{p}^{(\ell)} \right\rangle_A = \left\langle \mathbf{p}^{(t)}, A\mathbf{p}^{(\ell)} \right\rangle = 0$ for $\ell \neq t$
  - Step size looks familiar:

$$\alpha^{(t)} = \frac{\left\langle \mathbf{r}^{(t)},\ \mathbf{p}^{(t)} \right\rangle}{\left\langle \mathbf{p}^{(t)},\ A\mathbf{p}^{(t)} \right\rangle}$$

- CG is a special case of the CD-method
  - Generate $\mathbf{p}^{(t)}$ by A-orthogonalization of the residual vectors $\mathbf{r}^{(t)}$
  - Leads to orthogonal residuals
  - Formulation used extensively by (Hestenes & Stiefel, 1952)

Towards Asynchronous CG

Algorithmic Requirements

- Aim to at least satisfy requirements of a CD-method
  - Direction vectors must be conjugate
- Direction vector should only use local and asynchronously received data
- Requires storage of some past direction vectors
- Need to compute A-product of the direction vector without more communication

Asynchronous s-Approximate CD Method (s-ACD)

Initialization

- Given:
  - SPD $A \in \mathbb{R}^{m \times m}$ partitioned over N agents as

$$A = \begin{bmatrix} A_{[1]} \\ \vdots \\ A_{[k]} \\ \vdots \\ A_{[N]} \end{bmatrix}$$

  - Vector $\mathbf{b} \in \mathbb{R}^m$
- Initialize on each agent k:
  - Solution vector $\mathbf{x}_k^{(0)} = \mathbf{0}$,
  - Residual vector and **local direction vector** $\mathbf{r}_k^{(0)} = \mathbf{p}_k^{(0)} = \mathbf{b}$

Asynchronous s-Approximate CD Method (s-ACD)

Algorithm

At iteration t_k of agent k,

1. Asynchronously share ***partial*** **local direction vector** $\mathbf{p}_{[k]}^{(t_k)}$ and ***partial*** $\boldsymbol{A}$**-product** $\mathbf{w}_k^{(t_k)} := A_{[k]}^T \mathbf{p}_{[k]}^{(t_k)}$
2. Receive available updates $\left\{\mathbf{w}_\ell^{(t_\ell)}, \mathbf{p}_{[\ell]}^{(t_\ell)}\right\}_{\ell \in \mathcal{U}_k}$ from other nodes
3. Define the **asynchronous direction vector** $\widetilde{\mathbf{p}}_k^{(t_k)} \in \mathbb{R}^m$ block-wise as

$$\widetilde{\mathbf{p}}_{k,[\ell]}^{(t_k)} = \begin{cases} \mathbf{p}_{[k]}^{(t_k)}, & \ell = k \\ \mathbf{p}_{[\ell]}^{(t_\ell)}, & \ell \in \mathcal{U}_k \\ \mathbf{0}_{n_\ell}, & else \end{cases}$$

- Allows us to compute the **exact matrix-vector product**

$$\widetilde{\mathbf{w}}_k^{(t_k)} := A\widetilde{\mathbf{p}}_k^{(t_k)} = \mathbf{w}_k^{(t_k)} + \sum_{\ell \in \mathcal{U}_k} \mathbf{w}_\ell^{(t_\ell)}$$

- An issue: The **asynchronous direction vector** $\widetilde{\mathbf{p}}_k^{(t_k)}$ is probably not conjugate to earlier directions!

Asynchronous s-Approximate CD Method (s-ACD)

Algorithm (cont.)

At iteration t_k of agent k (cont.),

4. Define the **s-conjugate direction vector** $\mathbf{d}_k^{(t_k)}$ recursively by $\mathbf{d}_k^{(0)} := \tilde{\mathbf{p}}_k^{(0)}$ and

$$\mathbf{d}_k^{(t_k)} := \tilde{\mathbf{p}}_k^{(t_k)} - \sum_{i=1}^{\min(s,\, t_k)} \operatorname{proj}_A\left(\tilde{\mathbf{p}}_k^{(t_k)}, \mathbf{d}_k^{(t_k-i)}\right) \mathbf{d}_k^{(t_k-i)}, \qquad t_k > 0$$

where the projection operator under the A-inner product is defined by

$$\operatorname{proj}_A(\mathbf{u}, \mathbf{v}) := \frac{\langle \mathbf{u}, A\mathbf{v} \rangle}{\langle \mathbf{v}, A\mathbf{v} \rangle}$$

Note, the **exact A-product** can be defined recursively by $A\mathbf{d}_k^{(0)} := \tilde{\mathbf{w}}_k^{(0)}$ and

$$A\mathbf{d}_k^{(t_k)} := \tilde{\mathbf{w}}_k^{(t_k)} - \sum_{i=1}^{\min(s,\, t_k)} \operatorname{proj}_A\left(\tilde{\mathbf{p}}_k^{(t_k)}, \mathbf{d}_k^{(t_k-i)}\right) A\mathbf{d}_k^{(t_k-i)}, \qquad t_k > 0$$

- Requires storing the s previous conjugate direction vectors and their A-products
- Ensures conjugacy against last s directions with ***no additional communication!***

Asynchronous s-Approximate CD Method (s-ACD)

Algorithm (cont.)

At iteration t_k of agent k (cont.),

5. Compute $\alpha_k^{(t_k)} := \dfrac{\left\langle \mathbf{r}_k^{(t_k)}, \mathbf{d}_k^{(t_k)} \right\rangle}{\left\langle \mathbf{d}_k^{(t_k)}, A\mathbf{d}_k^{(t_k)} \right\rangle}$
6. Update $\mathbf{x}_k^{(t_k+1)} := \mathbf{x}_k^{(t_k)} + \alpha_k^{(t_k)} \mathbf{d}_k^{(t_k)}$
7. Update $\mathbf{r}_k^{(t_k+1)} := \mathbf{r}_k^{(t_k)} - \alpha_k^{(t_k)} A\mathbf{d}_k^{(t_k)}$

- Note, if $\mathbf{r}_k^{(t_k)} = \mathbf{b} - A\mathbf{x}_k^{(t_k)}$, then

$$\mathbf{r}_k^{(t_k+1)} = \mathbf{b} - A\mathbf{x}_k^{(t_k)} - \alpha_k^{(t_k)} A\mathbf{d}_k^{(t_k)} = \mathbf{b} - A\left(\mathbf{x}_k^{(t_k)} - \alpha_k^{(t_k)} \mathbf{d}_k^{(t_k)}\right) = \mathbf{b} - A\mathbf{x}_k^{(t_k+1)}$$

- So, the residual is still correct!

Asynchronous s-Approximate CD Method (s-ACD)

Algorithm (cont.)

At iteration t_k of agent k (cont.),

8. Save the **s-conjugate direction vector** $\mathbf{d}_k^{(t_k)}$ and its A-product $\mathbf{v}_k^{(t_k)}$

9. Finally, compute the next **local direction vector** $\mathbf{p}_k^{(t_k+1)}$ using prior directions:

$$\mathbf{p}_k^{(t_k+1)} := \mathbf{r}_k^{(t_k+1)} - \sum_{i=0}^{\min(s,\, t_k)-1} \operatorname{proj}_A\left(\mathbf{r}_k^{(t_k+1)}, \mathbf{d}_k^{(t_k-i)}\right) \mathbf{d}_k^{(t_k-i)}$$

{ Since the new direction is conjugate to the prior s directions, we refer to it as the asynchronous s-approximate CD method }

Asynchronous s-Approximate CD Method (s-ACD)

High-Level Summary

1. Asynchronously communicate **partial local direction vectors** and **partial A-products**
   - Send **partial local direction vector** and **partial A-product**
   - Receive updates from some other agents
   - No update from an agent ⇔ update of all zeros
2. Construct the **asynchronous direction vector** and its exact **A-product**
   - Concatenate the sent and received **partial local direction vectors**
   - Sum the sent and received **partial A-products**
3. Compute the **conjugate direction vector** and its exact **A-product**
   - A-orthogonalize **asynchronous direction vector** against s previous **conjugate direction vectors**
4. Perform a local CG iteration using the **conjugate direction vector**
5. Compute next **local direction vector**
   - A-orthogonalize the **local residual vector** against previous and current **conjugate direction vectors**

Asynchronous s-Approximate CD Method (s-ACD)

Trade-offs

- Pros:
  - Only communicate once per iteration
  - No synchronization!
- Cons:
  - Send vectors rather than scalars
  - Store $2s$ additional vectors
  - Computing conjugate direction vectors

Asynchronous s-Approximate CD Method (s-ACD)

Final Notes

- Some issues remain:
  - Since direction vectors may differ between nodes, solution vector desyncs
  - Similarly, residual differs between nodes, so no clear stopping criteria
- A solution to both:
  - Send local solution vector $\mathbf{x}_k^{(t_k)}$ and local part of residual $\mathbf{r}_{[k]}^{(t_k)}$ at each iteration
  - Periodically restart the CD method
    - Average the solution vectors as $\tilde{\mathbf{x}}_k^{(t_k)} = \frac{1}{N}\sum_{\ell=1}^{N} \mathbf{x}_k^{(t_k)}$
    - Reconstruct residual vector $\tilde{\mathbf{r}}_k^{(t_k)}$ using most recent $\left\{\mathbf{r}_{[\ell]}^{(t_\ell)}\right\}$ like we did with asynchronous direction vectors
    - Restart the method with $\mathbf{x}_k^{(0)} = \tilde{\mathbf{x}}_k^{(t_k)}$ and $\mathbf{r}_k^{(0)} = \mathbf{p}_k^{(0)} = \tilde{\mathbf{r}}_k^{(t_k)}$
  - Experimentally found that restarting every ~15 iterations provided optimal speedup

Numerical Results

Test Problem

- 2D Poisson problem

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -f(x,y), \qquad (x,y) \in (0,1)\times(0,1)$$

- Homogeneous Dirichlet boundary conditions

$$u(x,0) = u(x,1) = 0, \qquad x \in [0,1]$$
$$u(0,y) = u(1,y) = 0, \qquad y \in [0,1]$$

- Discretized over a 20×20 grid of internal nodes, yields the linear system

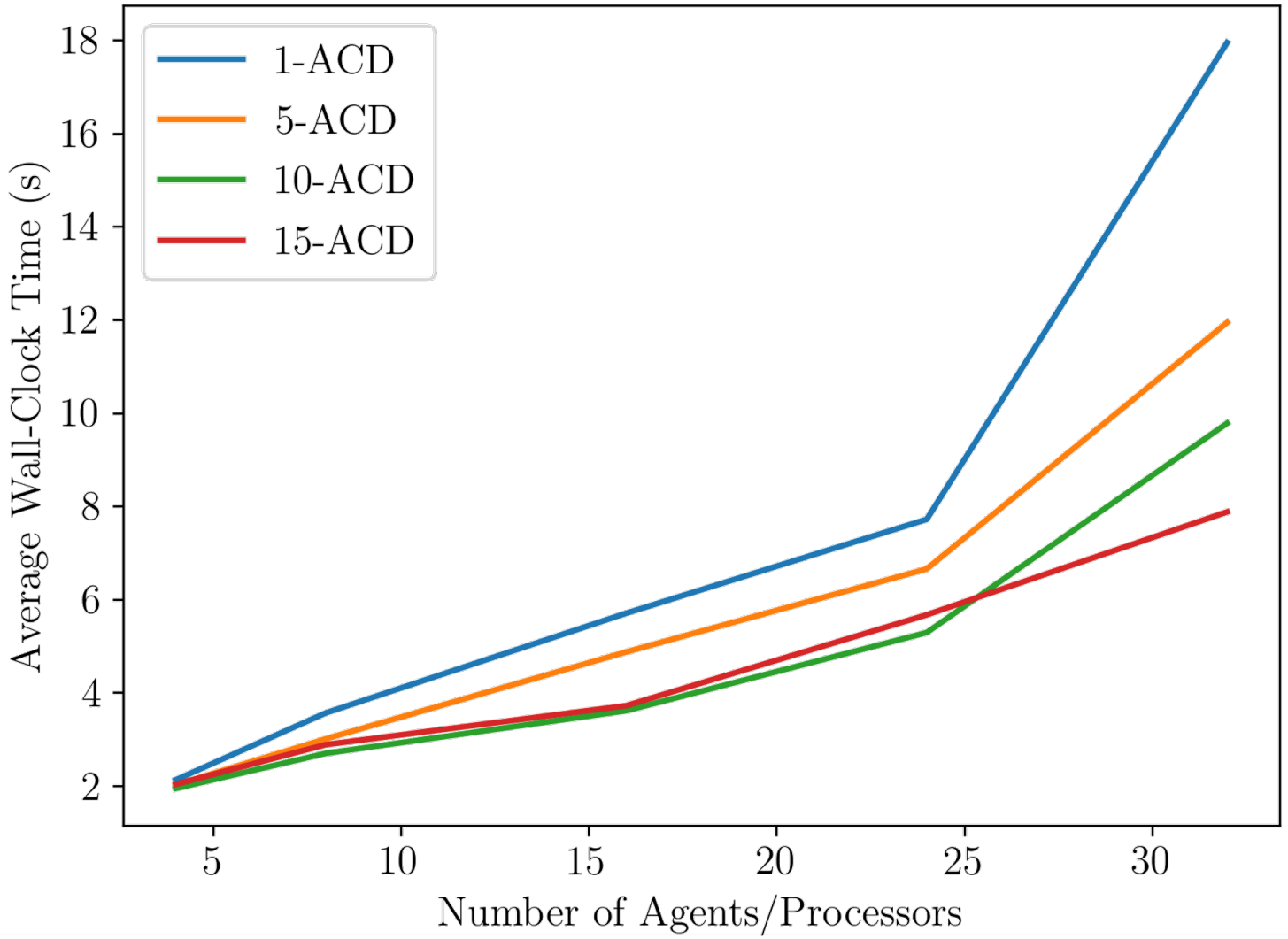
$$A\mathbf{x} = \mathbf{b}$$

Numerical Results
Test Problem

- Matrix $A \in \mathbb{R}^{400\times400}$ is given by the 2D discrete Laplacian on a 20×20 grid
- Right-hand side vector b is discretization of
$$f(x,y) = \pi^2 \sin \pi x \sin \pi y$$
- Note that with this f, the 2D Poisson problem yields the exact solution
$$u(x,y) = \sin \pi x \sin \pi y$$
- We perform strong scaling tests of s-ACD with different values of s
- Additionally, we compare results with a low tolerance against a high tolerance

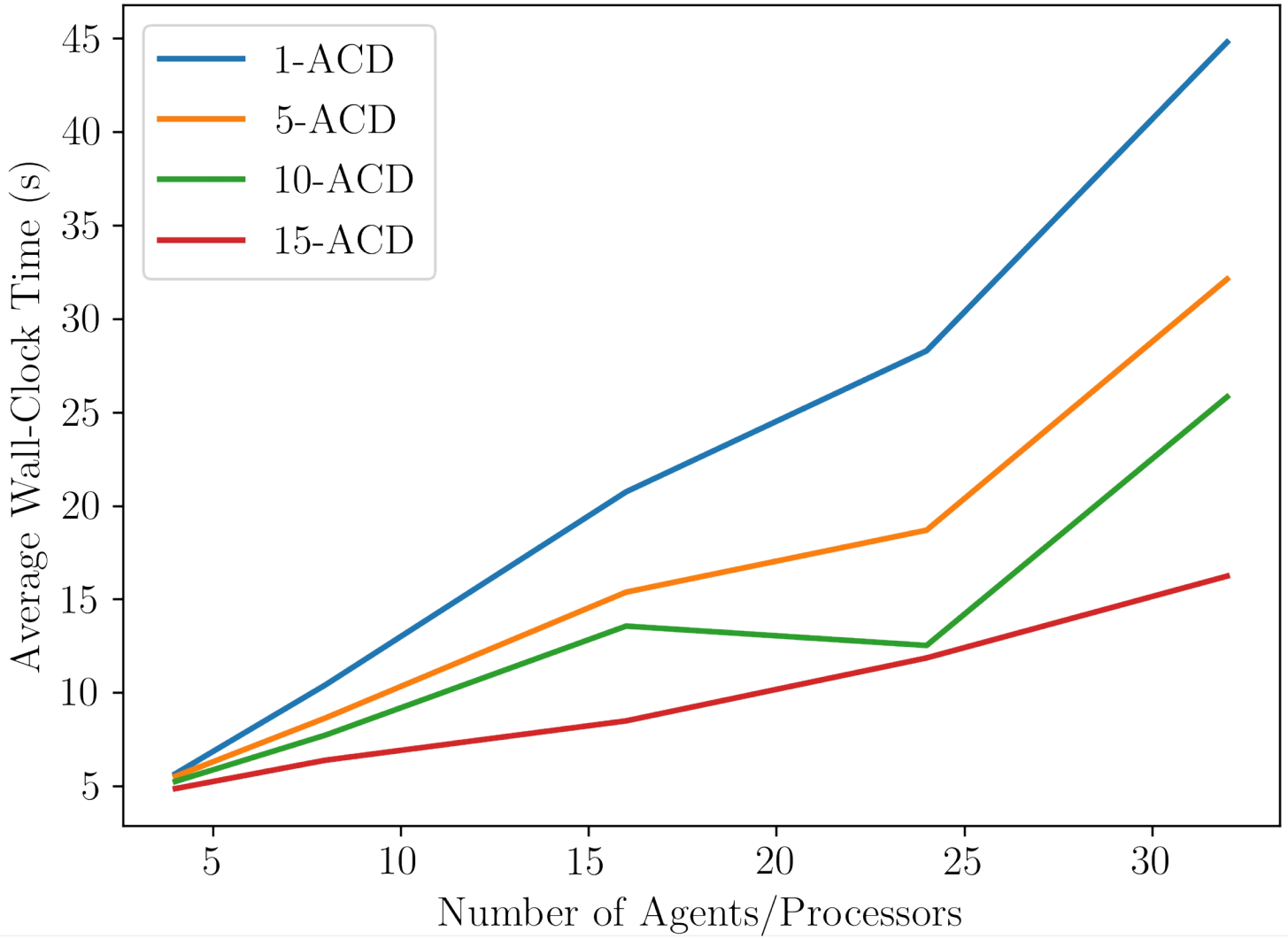
Numerical Results

Convergence Time for Different s Values (tolerance 1e-3)



Numerical Results

Convergence Time for Different s Values (tolerance 1e-8)



Numerical Results

Comparison with CG of Condition Number vs Iterations Necessary for Convergence

- CG iteration bound is given as

$$\tau_{CG}(A) = \log_{c(A)} {}^{\epsilon}/_{2}$$

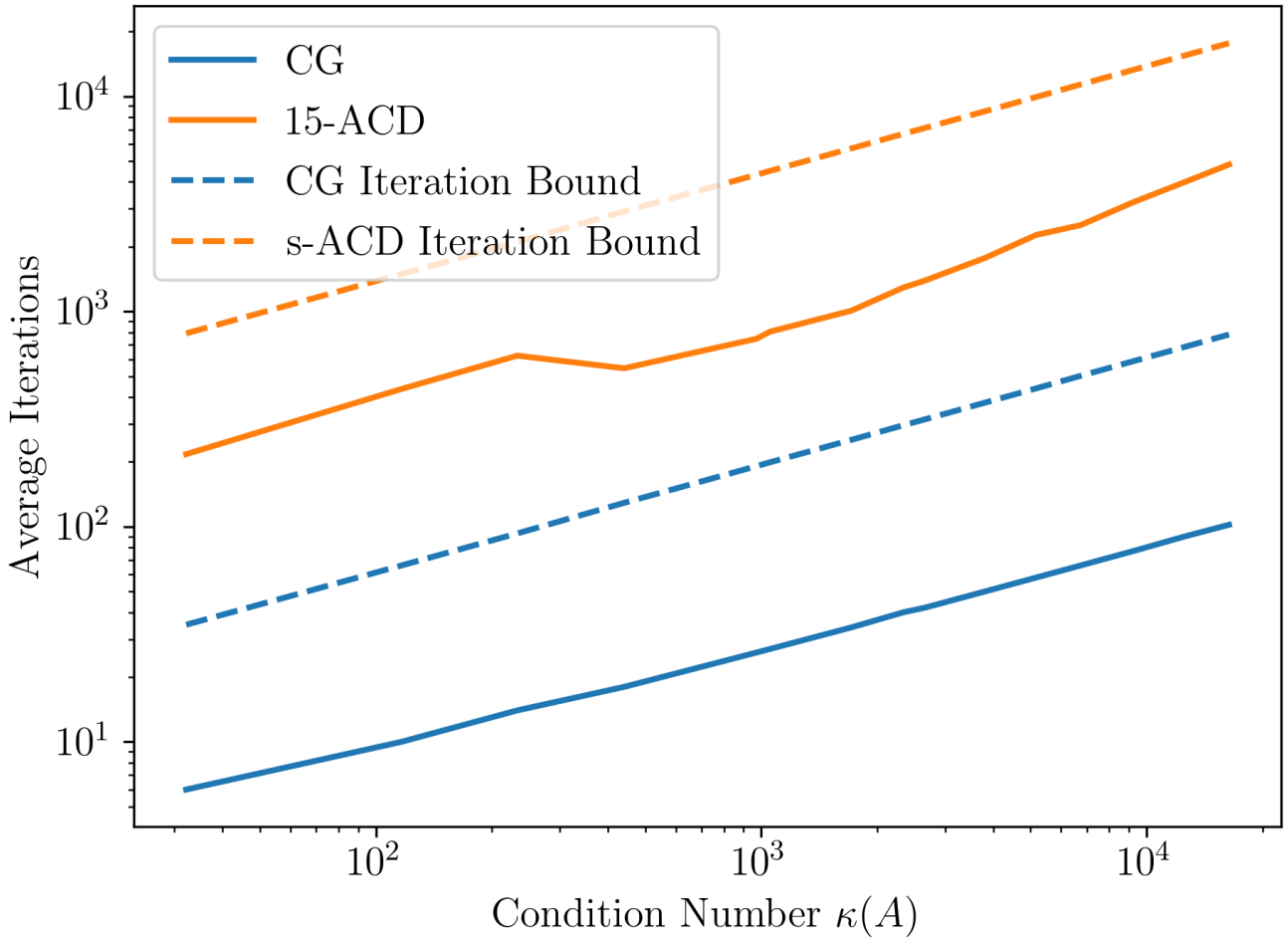
where $\epsilon = 10^{-3}$ and

$$c(A) = \frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1}$$

- ACD iteration bound is given as

$$\tau_{ACD}(A) = 22.7\tau_{CG}(A)$$

  - Constant is derived from the slope of the strong scaling experiments



Future Work

- Theoretical convergence results
- Improve orthogonality of residuals
- Other methods for restarting
  - Robust averaging on solution vector and residual
  - Could be run in a concurrent Skynet job
- Add resiliency measures
  - Communication delay
  - Data corruption
  - Agent failure/restart

Funded by LLNL LDRD project 22-ERD-045

References

Carson, E. C. (2020). An adaptive s-step conjugate gradient algorithm with dynamic basis updating. *Applications of Mathematics*, *65*(2), 123–151.

Chronopoulos, A. T., & Gear, C. W. (1989). S-step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics*, *25*(2), 153–168.

Eller, P., & Gropp, W. (2016). Scalable Non-blocking Preconditioned Conjugate Gradient Methods. *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*.

Ghysels, P., & Vanroose, W. (2014). Hiding global synchronization latency in the preconditioned Conjugate Gradient algorithm. *Parallel Computing*, *40*(7), 224–238.

Hestenes, M. R., & Stiefel, E. (1952). Methods of Conjugate Gradients for Solving. *Journal of research of the National Bureau of Standards*, *49*(6), 409.

Saad, Y. (2003). *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics.

Tiwari, M., & Vadhiyar, S. (2021). Pipelined Preconditioned s-step Conjugate Gradient Methods for Distributed Memory Systems. *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, 215–225.

Questions?

Implementation Details

Collaborative Autonomy with Skynet

- Skynet is an LLNL-developed framework for decentralized computing
  - Existing frameworks (e.g. MPI) only provide communication primitives
- In contrast, Skynet provides
  - Subscription-based communication of most C++ built-in types and std::vector
  - Flexible high-level iterative templates, which allow for more sophisticated communication patterns
  - Built-in implementations of algorithms such as the Jacobi method and robust push-sum averaging
- Uses TCP for flexible communication
  - Same program can run on an HPC cluster, across a collection of microcontrollers, etc.
  - Only configuration necessary is neighbor IP addresses and listening ports
- See Colin Ponce's talk at Session 11A on Friday morning

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC