Iterative Methods for Solving Large Sparse Linear Systems

Student Sarah Lawrence
Instructor James Quinlan

Abstract

This paper talks about the implementation and analysis of iterative methods for solving large, sparse linear systems. It focus on Conjugate Gradient (CG), GMRES, and BiCGSTAB. The study begins with a brief history of iterative methods, followed by detailed descriptions and implementations of each algorithm. Then the performance is evaluated using selected metrics, and the results are illustrated with plots to compare convergence rates under different conditions.

Introduction

The goal is to implement iterative methods for solving large, sparse linear systems (conjugate gradient, GMRES, BiCGSTAB) and study convergence rates for different conditions. Iterative methods start with an initial guess and repeatedly filter it to get closer to the actual solution. Iterative methods don't necessarily produce the exact solution; instead, they get to a specified tolerance. Direct methods, on the other hand, use a sequence of steps to obtain an exact solution. Because it obtains the exact solution, direct methods are too slow and memory-intensive to handle large sparse matrices.

Iterative methods are for solving large, sparse linear systems. A sparse linear system/matrix is an equation that takes the form of Ax=b. This is where A is a large $n * n$ matrix, and where most of the entries are zero making it sparse. The methods that will be used are Conjugate Gradient, GMRES, and BiCGSTAB since they are specific iterative algorithms to solve large, sparse linear systems. These methods will be taken and studied to see their convergence rates for different conditions. Before discussing the implementation, let's review the history of iterative methods.

History of Iterative Methods

Iterative methods were originally used for nonlinear problems. However, these problems were already solved by eliminating one variable after the other (Gaussian elimination) or other suitable methods. The importance of iterative methods was not discovered till computers started to tackle larger and larger problems. [4] As Gauss once said, "You will hardly ever again eliminate directly, at least not when you have more than 2 unknowns" [1]. Gauss realized that direct elimination becomes complicated and very slow as the number of unknowns increases. Gauss was the first to formalize and popularize solving large systems of linear equations. Unfortunately, he never published his works. After the death of Gauss in 1855, hundreds of notes were discovered in his desk. It required half a century to edit and publish the enormous amount of work Gauss produced. Only in 1903 did Werke publish volume 9 with most of Gauss's works. [4] That's how Gaussian elimination became well known, but this is a direct method.

Another type of method to tackle solving large systems of linear equations was Carl Gustav Jacobi. Unaware of the Gauss procedure, Carl Gustav Jacobi in 1804-1851 presented a new method for solving the linear system. [4] The method he created is a substitution-based method. It calculates the next iteration's values by using the previous iteration's values. This became another popular iterative method. Soon, more people jumped into solving large linear systems.

Philipp Ludwig von Seidel was a student of Jacobi in K¨ Königsberg. He sometimes performed numerical calculations for Jacobi. Once he became a professor himself, he made various applications of linear systems to scientific research. Seidel ended up refining Gauss's approach to produce what is now called the Gauss-Seidel iterative method. Some other algorithms that soon followed are Conjugate Gradient, GMRES, and BiCGSTAB.

Tested Methodology

First Iterative Method

Conjugate Gradient(CG)

Background

Conjugate Gradient was invented by Magnus Hestenes, Eduard Stiefel, and Cornelius Lanczos. They were both mathematicians associated with the National Bureau of Standards. ([3]) This method is for solving large, sparse systems of linear equations of the form Ax=b, but A has to be a positive-definite or indefinite matrix

$$Ax = b$$

- A symmetric, positive-definite matrix
- x is the unknown vector
- b is the known right-hand side vector

A has to be positive-definite because it is convex. If it's convex, there is one unique minimum, so CG can converge. However, if A is not positive definite and instead flat or saddle-shaped, a minimum might not exist. An excellent visual by Jonathan Richard Shewchuk [7] shows what these look like. Figure 1 shows my reproduction of his visual.

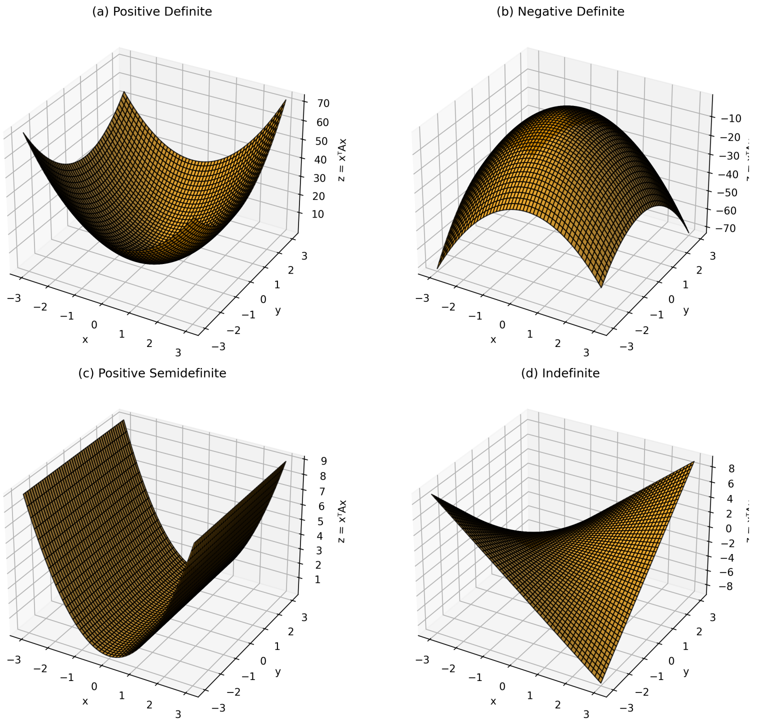


Figure 1: Visualization of Matrix Types

- (a) Positive-definite matrix → bowl → Converges
- (b) Negative-definite matrix → inverted bowl → Fails or unstable
- (c) Positive-indefinite → valley → Diverges
- (d) Indefinite matrix → saddle → Breaks down

(a) is what will be primarily focused on. Having examined the three-dimensional shape. It's now time to present its two-dimensional representation as a heatmap. The flat heatmap of a positive definite matrix can be seen in Figure 2.

In this image, all diagonal entries are positive. This is due to the fact that they are often much larger than off-diagonal terms. Notice how there is mirror symmetry across the main diagonal. But why are symmetric properties important? They are because CG relies on inner-product geometry to produce conjugate directions and guarantee convergence. Without symmetry, that isn't possible. CG constructs its iterations inside the Krylov subspace. The symmetry and positive-definiteness of A ensure that the method produces mutually A-orthogonal search directions within this subspace. In short, CG is taking advantage of the unique properties of positive-definite matrices and the symmetry to generate conjugate directions instead of arbitrary directions. This allows it to converge in at most an n×n system. This method is favored because it is memory efferent and converges faster than methods like Jacobi or Gauss Seidel. It is faster due to it builds optimal search directions that avoid redoing work.

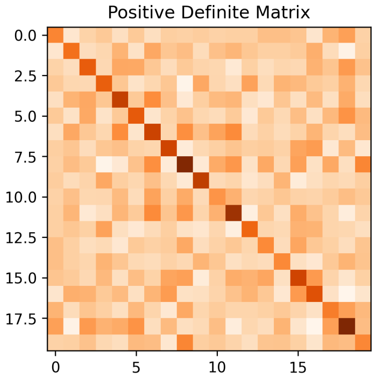


Figure 2: Positive Definite Matrix Heatmap

Implementation

To implement the Conjugate Gradient (CG) method, it is important to first identify its required inputs. CG needs the known vectors A and b, the default values for the initial guess, tolerance, and max iterations. This is the input for the function. Next to not mess up the original values, the initial estimate x is copied. Then the first residual was acquired. A copy of this residual is used as the initial search direction. Then the squared norm of the residual is also calculated. The first residual is then stored for plotting purposes. The code for the setup can be seen below.

Initialization:

```
# conjugate gradient
function conjugate_gradient(A, b; x0=zero(b), tol=1e-8, maxiter=1000)
    x = copy(x0) # current estimate
    r = b - A * x # residual: how far x is from the true solution
    p = copy(r) # search direction
    rs_old = dot(r, r) # squared norm of the residual

    residuals = [sqrt(rs_old)] # store the first residual
```

Next, after setup, whether it converged or not needs to be checked. This is done by seeing if the residual norm is small enough; if it has, then CG has converged. The code for this can be seen in ”Check Convergence:”. If it hasn't converged, the search direction needs to be updated. Now using $beta = rs_new/rs_old$ we determine how much of the previous direction is kept (called beta). Then, adding the new residual with beta times the previous direction makes the new direction. Finally, the once new squared norm of the residual is now becomes the old residual. This can be seen in ”New Search Direction:”.

Check Convergence:

```
# Check Convergence
rs_new = dot(r, r)
push!(residuals, sqrt(rs_new))
if sqrt(rs_new) < tol
    rintln("Converged in $k iterations")
    return x
```

New Search Direction:

```
# Update Search Direction
beta = rs_new / rs_old
p = r + beta * p
rs_old = rs_new
```

Second Iterative Method

Generalized Minimal Residual method (GMRES)

Background

GMRES was invented by Yousef Saad and Martin H. Schultz. They were both associated with the Society for Industrial and Applied Mathematics. [6] This method is the same as CG, which is used for solving large, sparse systems of linear equations of the form Ax=b. However, there are some differences. One being CG requires A to be symmetric positive definite. GMRES works for any nonsingular matrix, including nonsymmetric and indefinite matrices. Unlike CG, GMRES does not require A to be symmetric or positive definite.

$$Ax = b$$

- A is a general square matrix (can be nonsymmetric)
- x is the unknown vector
- b is the known right-hand side vector

Another important difference is the Krylov subspace. CG enforces A-conjugacy and minimizes the error in the A-norm, whereas GMRES uses the Arnoldi process to build an orthonormal basis and minimizes the residual norm at each iteration. [9]

Finally, unlike CG, which has a fixed memory, GMRES has a growing storage and computational cost problem. This is due to it storing all previous Krylov basis vectors.

Implementation

To implement the Generalized Minimal Residual method, it is important to first identify its required inputs. GMRES needs the known vectors A and b, the default values for the initial guess, tolerance, and max iterations. Next, the number of entries in the vector b is collected. Then the initial guess is copied, and the initial residuals are acquired, like in the CG function. The magnitude of that initial error is then calculated. Finally, the norms are stored at each step. After the initialization, the code moves into the Arnoldi process. The Arnoldi process builds a basis of vectors V for the Krylov subspace by repeatedly multiplying by A and orthogonalizing the result using modified Gram–Schmidt. The modified Gram–Schmidt is used to keep the basis vectors stable when orthogonalizing the new vector w against all previous basis vectors.

Initialization:

```
function gmres_residuals(A, b; x0=zero(b),
tol=1e-8, maxiter=100)
    n = length(b) # number of entries in the vector b
    x = copy(x0) # current estimate
    r0 = b - A*x # initial residual
    beta = norm(r0) # magnitude of that initial error

    residuals = [beta] # residuals array
                       # stores the norms at each step
```

Arnoldi process:

```
# Arnoldi process
    V = zeros(n, maxiter+1) # basis vectors
    H = zeros(maxiter+1, maxiter)
    # upper Hessenberg matrix
    V[:,1] = r0 / beta # orthonormal basis
                       # vector

    for j in 1:maxiter
        w = A * V[:,j]
        # modified Gram{Schmidt
        for i in 1:j
            H[i,j] = dot(V[:,i], w)
            w -= H[i,j] * V[:,i]
        end
        # new basis vector
        H[j+1,j] = norm(w)
        if H[j+1,j] != 0
            V[:,j+1] = w / H[j+1,j]
        end
```

Now that the Arnoldi process is completed, let's move into the least squares problem. We use least squares to minimize the residual within the Krylov subspace. This can be seen in "Least Squares Min". Then we check whether the weather convergence has occurred. This can be seen in "Check Convergence".

Least Squares Min:

```
# Solve least squares min ||beta e1 - H y||
        e1 = zeros(j+1); e1[1] = beta
        y = H[1:j+1,1:j] \ e1
        x_approx = x0 + V[:,1:j]*y
```

Check Convergence:

```
# residual norm for plotting
        res = norm(b - A*x_approx)
        push!(residuals, res)

# check convergence
    if res < tol
        println("GMRES converged in $j iterat
        return x_approx, residuals
    end
```

Third Iterative Method

Biconjugate gradient stabilized (BiCGSTAB)

Background

BiCGSTAB was "introduced by H.A. Van der Vorst, S. Cools, and Wim Vanroose proposed an optimization known as pipelined BiCGStab". [5] Like CG and GMRES, BiCGSTAB is an iterative Krylov subspace method used for solving large, sparse linear systems of the form Ax=b.

$$Ax = b$$

- A is a general square matrix (can be nonsymmetric)
- x is the unknown vector
- b is the known right-hand side vector

BiCGSTAB is designed for general nonsymmetric matrices and was introduced as a smoother converging version of both BiCG and CGS methods. It combines the low memory cost of CG-type methods with smoother and more stable convergence than BiCG.

This method was developed to fix the issue of non-symmetric linear systems while also avoiding the irregular convergence patterns of the conjugate gradient squared (CGS) method. [5] CGS is not the same as CG, since CGS is for solving nonsymmetric systems while CG is for symmetric systems.

BiCGSTAB is favored due to it's structure allowing for efficient parallelization. [8] BiCGSTAB also handles some matrices better than GMRES. This is because GMRES requires storing all Krylov basis vectors. This is costly for large systems. However, BiCGSTAB only stores a small number of vectors and scalars each iteration. This makes it more memory-efficient. One down side is that BiCGSTAB may not always provide the smoothest convergence and can still exhibit breakdowns [2], but it is still widely used for its memory usage and convergence speed.

Implementation

To implement the Biconjugate gradient stabilized method, it is important to first identify its required inputs. BiCGSTAB needs the known vectors A and b, the default values for the initial guess, tolerance, and max iterations. After setting up the function, inputs its time for initialization. In the code, the first thing that is initialized is the size of the problem. Then, the current guess of the solution. Next is the residual and the fixed copy of the initial residual. The initializing scalars to 1. Finally, set the two auxiliary vectors. The code for this is in "Initialization".

Initialization:

```
function bicgstab_residuals(A, b; x0=zero(b), tol=1e-8, maxiter=1000)
    n = length(b) # determines the size of the problem
    x = copy(x0) # current approximation of the solution
    r = b - A*x # residual
    r_hat = copy(r) # fixed copy of the initial residual (used for or computing scalars)
    rho_old = alpha = omega = 1.0 # scalars initialized to 1
    v = zeros(n) #  auxiliary vector used in the iterations
    p = zeros(n) # auxiliary vector used in the iterations

    residuals = [norm(r)] # list storing the norm of the residual at each step
```

After initialization, it finds the search direction. The code for this can be seen in "Search Direction". The search direction p in BiCGSTAB determines the direction along which the solution is updated in each iteration. It is updated using a combination of the current residual and the previous search direction. This update ensures that each new direction builds on prior progress while achieving convergence. Updating variables after this is important this is because vector v = A*p and the scalar alpha determine how far to move along the search direction. This is so that the residual is reduced as much as possible in that direction. Then s, t, and omega remove remaining error components. This allows the method to update x and a smaller residual r. The norm of r is saved for analysis.

Search Direction:

```
for k in 1:maxiter
        rho = dot(r_hat, r)
        # the method breaks down because a division by zero
        if rho == 0
            println("Breakdown: rho=0")
            break
        end
        # update the search direction p
        if k == 1
            p = r
        else
            beta = (rho/rho_old) * (alpha/omega)
            p = r + beta*(p - omega*v)
        end
```

Updating Variables:

```
v = A*p # matrix-vector product
alpha = rho / dot(r_hat, v) # reduce
s = r - alpha*v # residual after
               # moving
t = A*s # stabilize the
        # correction step
omega = dot(t,s)/dot(t,t)
x += alpha*p + omega*s
r = s - omega*t

push!(residuals, norm(r)) # residual
```

Finally, there is a check to see whether the residual has become small enough to declare convergence. BiCGSTAB can sometimes not have convergence and instead have a numerical breakdown that prevents the algorithm from continuing. If neither happens, the algorithm updates the scalar rho_old and goes to the next iteration.

Final Checks:

```
        # if residual is below the tolerance then stop
        if norm(r) < tol
            println("BiCGSTAB converged in $k iterations")
            return x, residuals
        end

        # check for breakdown
        # if omega = 0, the algorithm cannot continue.
        if omega == 0
            println("Breakdown: omega=0")
            break
        end

        rho_old = rho # is updated for the next iteration
        end
    println("BiCGSTAB max iterations reached")
    return x, residuals
end
```

Analysis

Prior Evaluation Methods

Residual norm is the most common method for evaluating iterative methods. The residual is: $r = b - Ax$. This determines how far we are from the correct value b. [**?**] A small residual means that the computed solution almost satisfies Ax=b. However, a large residual means that the solution is still far from the truth.

The number of Iterations is also a common method for evaluating iterative methods. This can show how fast a method is. This is because fewer iterations mean faster convergence. What also determines how fast a method is is the CPU time (Runtime). Number of iterations tells you the algorithm's efficiency in terms of convergence per iteration. While runtime tells you the performance for solving a problem on a system.

There are more methods, such as memory usage, matrix-vector multiplications count, and more, but we won't focus on those.

Used Evaluation Methods

Now, to study convergence rates for different conditions. First, what is a condition number? well-conditioned, easy to solve numerically, while ill-conditioned is not.

For the results, CG, GMRES, and BiCGSTAB were used. They were tested agents three different conditions. Since CG requires the matrix to be symmetric and positive definite, all of our tests will use matrices with these properties to ensure fair comparisons. The code for making the matrices can be seen in ”Creating SPD Matrices”.

We then define the conditions. The first being a 10-well-conditioned matrix. Next is a 1000 ill-conditioned matrix. Finally is a 10000000 ill-conditioned matrix. To study their convergence rates, let's look at plotting the residual for every iteration to see how fast convergence occurs. The plotting code can be seen in the code block ”Plot Images”. Next, lets look at the residual norm, and comparing time and iteration to see how well each algorithm did.

Creating SPD Matrices:

```
using Random, LinearAlgebra

# Fix the seed
Random.seed!(123)  # <- same seed every time

# Define your SPD matrices and vector b
n = 5
A1 = spd_matrix(n, 10)
A2 = spd_matrix(n, 1000)
A3 = spd_matrix(n, 1000000000)
b = randn(n)  # this will always be
              # the same with the same seed

# Run conjugate gradient
x1, res1 = conjugate_gradient(A1, b)
x2, res2 = conjugate_gradient(A2, b)
x3, res3 = conjugate_gradient(A3, b)
```

Plot Images:

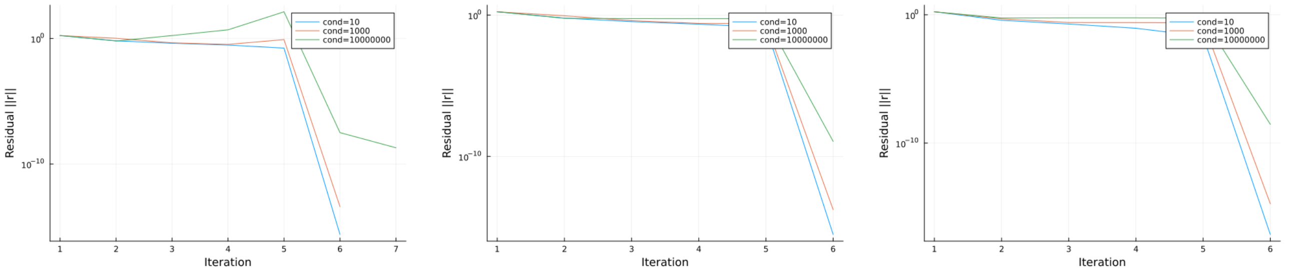
```
using Plots

plt = plot(1:length(res1), res1,
yscale=:log10, label="cond=10",
           xlabel="Iteration",
           ylabel="Residual ||r||")
plot!(plt, 1:length(res2), res2,
label="cond=1000")
plot!(plt, 1:length(res3), res3,
label="cond=10000000")

# Save the plot as a PNG (or PDF, SVG, etc.)
savefig(plt, "cg_residuals.png")
```

Final Results

Let's first start by looking at Figures 3,4, and 5. In the figures, there are residuals for every iteration. Figure 3 shows that condition 10 and 1,000 convergence at iteration 5 and fully drop off at 6. However, 10,000,000 converges at iteration 6 and fully drops off at 7. Visually the Figures 4 and 5 are slightly different, but the most important part is that both have all their conditions converging at Iteration 6. What does this tell us? So far just when it just converges. What about which method performs the best for each condition? Or how long did each method take? The plots are nice but hard to compare, so let's look at the residual values for each method.



((a)) CG residuals

((b)) GMRES residuals

((c)) BiCGSTAB residuals

Figure 3: Residual plots for CG, GMRES, and BiCGSTAB

In Table 1, there are more visible differences between the values. As mentioned earlier, a small residual indicates that the computed solution almost satisfies Ax=b. The orange values show the best residual for that condition. BiCGSTAB over both condition 10 and 1,000 of this metric performed the best. This means BiCGSTAB converged to the most accurate solution for these conditions. However, GMRES performed better on condition 10,000,000. It can be inferred that BiCGSTAB could still be outperformed under higher conditions.

Table 1

Method	Residual (10)	Residual (1000)	Residual (10,000,000)
CG	2.975240411612116e-16	4.010693545804971e-13	1.9565518504412037e-9
GMRES	4.734502448698718e-15	2.1173778872382343e-13	1.1603178684643001e-9
BiCGSTAB	4.5615749882693775e-18	1.1902555678395238e-14	2.9083651242161397e-9

Table 2 shows the time required by each method. If bigger matrices are being run, a method being slightly faster per solves can save hours or days. This is important if you're solving complex, large, sparse linear systems. The orange is again the best result for each condition. BiCGSTAB did the best for condition 10. However, CG was the quickest for conditions 1,000 and 10,000,000. Even with another iteration in condition 10,000,000, CG still ran faster than the others.

Table 2

Method	Time (10)	Iteration (10)	Time (1000)	Iteration (1000)
CG	0.000301573	5	0.000132228	5
GMRES	0.000241092	5	0.000205129	5
BiCGSTAB	0.000122345	5	0.000267802	5

Method	Time (10,000,000)	Iteration (10,000,000)
CG	0.000123069	6
GMRES	0.000258137	5
BiCGSTAB	0.000136422	5

Based on these results, it can be assumed that Tables 1 and 2 don't seem to have a correlation. This can be deduced since CG had the worst residual results, but doesn't always have the best time results for every condition. Same with BiCGSTAB, this method has the best residual results for two conditions, but has the best time for condition 10, and then the worst time result for condition 1,000.

Conclusion

All in all, with proven results and circumstances that BiCGSTAB had the most accurate convergence and CG was the fastest to converge. This can be seen in Tables 1 and 2. CG being faster might be likely due to how it builds optimal search directions that avoid redoing work. BiCGSTAB being more accurate might be because its update formula directly forces the residual to be small and smooth in each iteration. GMRES overall didn't preform the worst or the best, and it did get better in Table 1. If I were to continue this study, I would perform more tests with GMRES and BiCGSTAB to see if GMRES would do better for higher conditions.

References

[1] Gauss, C.F.: Letter to gerling. Werke **9**, 281 (1823)

[2] Graves-Morris, P.R.: The breakdowns of bicgstab. Numerical Algorithms **29**(1–3), 97–105 (2002). https://doi.org/10.1023/A:1014864007293, https://doi.org/10.1023/A:1014864007293

[3] Hestenes, M.R., Stiefel, E.: Methods of conjugate gradients for solving linear systems. Journal of Research of the National Bureau of Standards **49**(6), 28 (1952), https://nvlpubs.nist.gov/nistpubs/jres/049/jresv49n6p409$_a$1*b.pdf*

[4] Martin J Gander, P.H., Wanner, G.: Landmarks in the history or iterative methods p. 89 (2025), https://www.unige.ch/ gander/Preprints/LandmarksPaper.pdf

[5] Mykhailo Havdiak1, J.I.A., Iakymchuk, R.: Robustness and accuracy in pipelined bi-conjugate gradient stabilized method: A comparative study. arXiv preprint p. 8 (2024), https://arxiv.org/pdf/2404.13216

[6] Saad, Y., Schultz, M.H.: Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems* **7**, 14 (1986), https://web.stanford.edu/class/cme324/saad-schultz.pdf

[7] Shewchuk, J.R.: An introduction to the conjugate gradient method without the agonizing pain. Journal of Research of the National Bureau of Standards p. 64 (1994), https://www.cs.cmu.edu/ quake-papers/painless-conjugate-gradient.pdf

[8] Yang, L.T., Brent, R.P.: The improved bicgstab method for large and sparse unsymmetric linear systems on parallel distributed memory architectures

[9] Zhang, J., Luo, Y.: Preprocessed gmres for fast solution of linear equations. arXiv preprint p. 19 (2024), https://arxiv.org/pdf/2404.06018