# Conjugate Gradient Method

Jin-Tong Feng | Chih-Yuan Chang | Yu Voon Ng

**Theory**

Jin-Tong Feng

**Implementation**

Chih-Yuan Chang

**Parallelization**

Yu Voon Ng

**Theory**

Jin-Tong Feng

**Implementation**

Chih-Yuan Chang

**Parallelization**

Yu Voon Ng

## Introduction

Linear relation

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

$$\longleftrightarrow$$

$$\nabla f(\overrightarrow{x}) = \frac{1}{2}A^T\overrightarrow{x} + \frac{1}{2}A\overrightarrow{x} - \overrightarrow{b}$$
$$= A\overrightarrow{x} - \overrightarrow{b} = 0$$

Solving the minimum

$$f(\overrightarrow{x}) = \frac{1}{2}\overrightarrow{x}A\overrightarrow{x} - b\overrightarrow{x} + c$$

| | | |
|---|---|---|
| $A$: symmetric positive definite matrix | $\overrightarrow{x}$: unknown | $\overrightarrow{b}$: given |

# Introduction

## Some definitions…

### Error

$$\overrightarrow{e_i} = \overrightarrow{x_i} - \overrightarrow{x}$$

### Residual

$$\overrightarrow{r_i} = \overrightarrow{b_i} - A\overrightarrow{x_i} = -\nabla f(\overrightarrow{x_i})$$

Solving the minimum

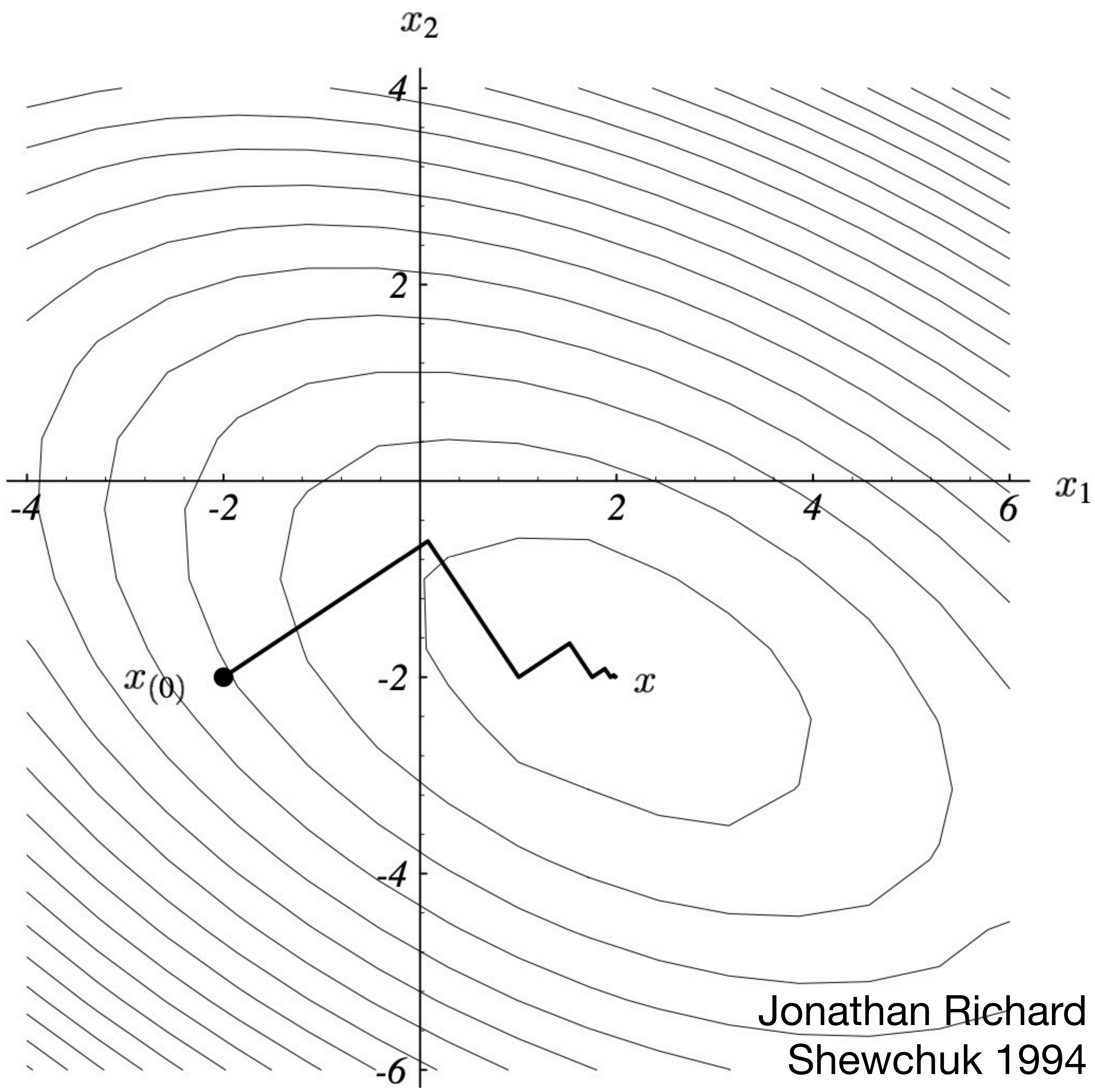
$$f(\overrightarrow{x}) = \frac{1}{2}\overrightarrow{x}A\overrightarrow{x} - b\overrightarrow{x} + c$$

At the minimum, both error and residual are zero

# Gradient Descent

Time Complexity for a
2D poisson equation

| $\mathcal{O}(n^2)$ | $\mathcal{O}(n^{1.5})$ |
|---|---|
| gradient descent | conjugate gradient |



Jonathan Richard
Shewchuk 1994

## Conjugacy

- We hope that we do not move along a direction chosen before.
- Suppose that we have n independent vectors with

$$\overrightarrow{d_i} A \overrightarrow{d_j} = 0, \forall i \neq j$$

known as Conjugacy or A-orthogonality.

- $\overrightarrow{d_i}$ can be derived using Gram-Schmidt conjugation from $\overrightarrow{u}$ **arbitrary linear independent vectors** → a $\mathcal{O}(n^3)$ algorithm.

## Conjugate Gradient Method

- We take the **residuals as search directions**

$$\overrightarrow{r_i} = \vec{r}_{i-1} - \alpha_{i-1} A \overrightarrow{d}_{i-1}$$

- A subspace constructed by directions of steps is

$$D_i = \text{span}\left\{\overrightarrow{d_0}, \overrightarrow{d_1}, \cdots, \overrightarrow{d}_{i-1}\right\}$$

known as Krylov subspace, which can be further expressed as

$$D_i = \text{span}\left\{\overrightarrow{r_0}, A\overrightarrow{r_0}, \cdots, A^{i-1}\overrightarrow{r_0}\right\}$$

## Conjugate Gradient Method

- Due to conjugacy, we have

$$\overrightarrow{r_i} \cdot \overrightarrow{r_j} = 0, \ \forall i \neq j$$

which automatically guarantees A-orthogonality among a new step and old steps!

- The lengths of steps are

$$\beta_{i+1} = \frac{\vec{r}_{i+1} \cdot \vec{r}_{i+1}}{\overrightarrow{r_i} \cdot \overrightarrow{r_i}}$$

$$\vec{d}_0 = \vec{r}_0 = \vec{b} - A\vec{x}_0$$

$$\alpha_i = \frac{\vec{r}_i \cdot \vec{r}_i}{\vec{d}_i A \vec{d}_i}$$

$$\vec{x}_{i+1} = \vec{x}_i + \alpha_i \vec{d}_i$$

$$r_{i+1} = \vec{r}_i - \alpha_i A \vec{d}_i$$

$$\beta_{i+1} = \frac{\vec{r}_{i+1} \cdot \vec{r}_{i+1}}{\vec{r}_i \cdot \vec{r}_i}$$

$$\vec{d}_{i+1} = \vec{r}_i + \beta_{i+1} \vec{d}_i$$

Theory

Jin-Tong Feng

## Implementation

Chih-Yuan Chang

Parallelization

Yu Voon Ng

## Code Logic

```
while (np.dot(r.T, r)/N**4 > 10e-16) == True:  # still iterating
    alpha = np.dot(r.T, r) / np.dot(d.T, np.dot(A, d))
    x = x + alpha * d  # step to next guess
    rnew = r - alpha * np.dot(A, d)  # update residual r
    beta = np.dot(rnew.T, rnew) / np.dot(r.T, r) #correction
    r = rnew
    d = r + beta * d  # compute new search direction
```

## The Poisson eq.

$$\frac{\partial^2 t(x,y)}{\partial x^2} + \frac{\partial^2 t(x,y)}{\partial y^2} = 0$$

$$\frac{\partial^2 t(x,y)}{\partial x^2} = \lim_{h \to 0} \frac{t(x-h,y) - 2t(x,y) + t(x+h,y)}{h^2}$$

$$\frac{\partial^2 t(x,y)}{\partial y^2} = \lim_{h \to 0} \frac{t(x,y-h) - 2t(x,y) + t(x,y+h)}{h^2}$$

## The Poisson eq.

$$\frac{\partial^2 t(x,y)}{\partial x^2} + \frac{\partial^2 t(x,y)}{\partial y^2} = 0$$

Instead of asking the code for the t value at all points (an infinite problem), we split up the area into $n^2$ cells, if n = 6, then:

| | | | | | |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 |
| 7 | 8 | 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 | 17 | 18 |
| 19 | 20 | 21 | 22 | 23 | 24 |
| 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | 32 | 33 | 34 | 35 | 36 |

# Simplification

$$\frac{t(i-1)-2t(i)+t(i+1)}{1/(k+1)^2}+\frac{t(i-k)-2t(i)+t(i+k)}{1/(k+1)^2}=0$$

Larger k
- more accurate
- more cost

$$t(i-k)+t(i-1)-4t(i)+t(i+1)+t(i+k)=0$$

Down Left Center Right Up

Each cell has its own equation, that we can save into a matrix A

The eq. then becomes a $Ax = b$ problem

## The A Matrix

- Review: The Poisson eq. can be made into an $Ax = b$ problem
- The A matrix is a $n^2 \times n^2$ matrix

Takes up too much memory (n = 256 takes 32 GiB)

Is made up of mostly zeroes: very sparse

Hard to parallelize

Big bottleneck for efficiency

```
def A_matrix(idx_cell):
    idx_neighbor_list = []
    if (idx_cell%N  != 0 ):
        idx_neighbor_list.append(idx_cell-1)
    if (idx_cell%N  != (N-1)):
        idx_neighbor_list.append(idx_cell+1)
    if (idx_cell//N != (N-1)):
        idx_neighbor_list.append(idx_cell+N)
    if (idx_cell//N != 0    ):
        idx_neighbor_list.append(idx_cell-N)
    return idx_neighbor_list
```

This is only n=3!

```
[-4.  1. -0.  1. -0. -0. -0. -0. -0.]
[ 1. -4.  1. -0.  1. -0. -0. -0. -0.]
[-0.  1. -4. -0. -0.  1. -0. -0. -0.]
[ 1. -0. -0. -4.  1. -0.  1. -0. -0.]
[-0.  1. -0.  1. -4.  1. -0.  1. -0.]
[-0. -0.  1. -0.  1. -4. -0. -0.  1.]
[-0. -0. -0.  1. -0. -0. -4.  1. -0.]
[-0. -0. -0. -0.  1. -0.  1. -4.  1.]
[-0. -0. -0. -0. -0.  1. -0.  1. -4.]
```

## The A matrix (cont.)

Instead, we tell the code how A interacts with d

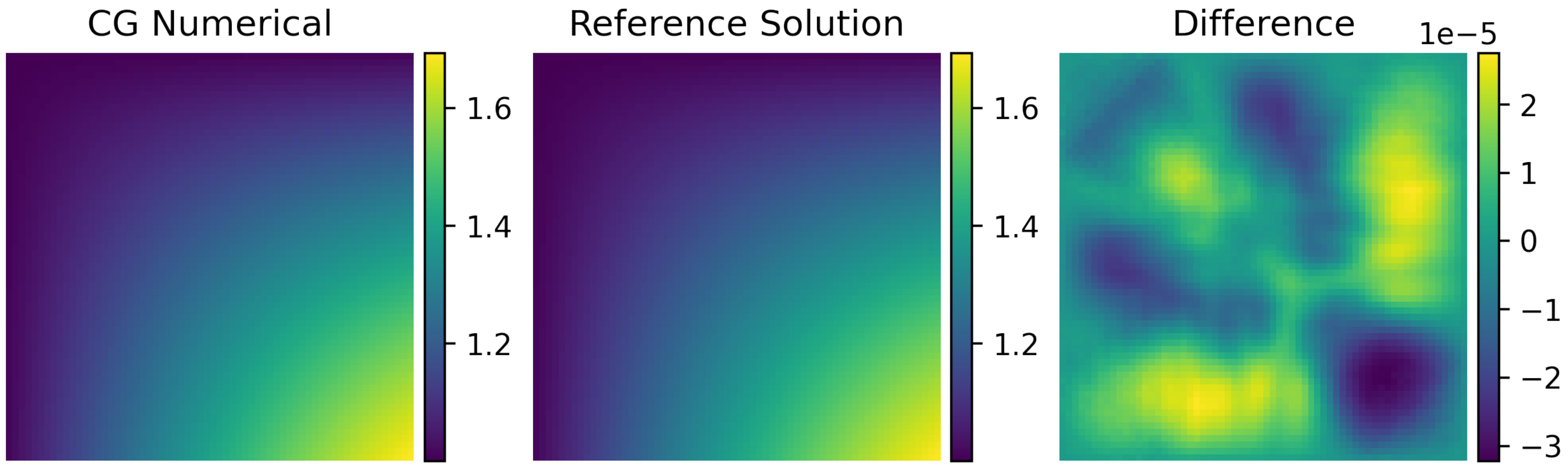
```
def Adotd(d):
       result = []
    for row in range(dimension):
        index_d = A_matrix(row)
        result.append(np.sum(d[index_d]) + d[row]*-4)
    return result
```

## CG Solver — final code

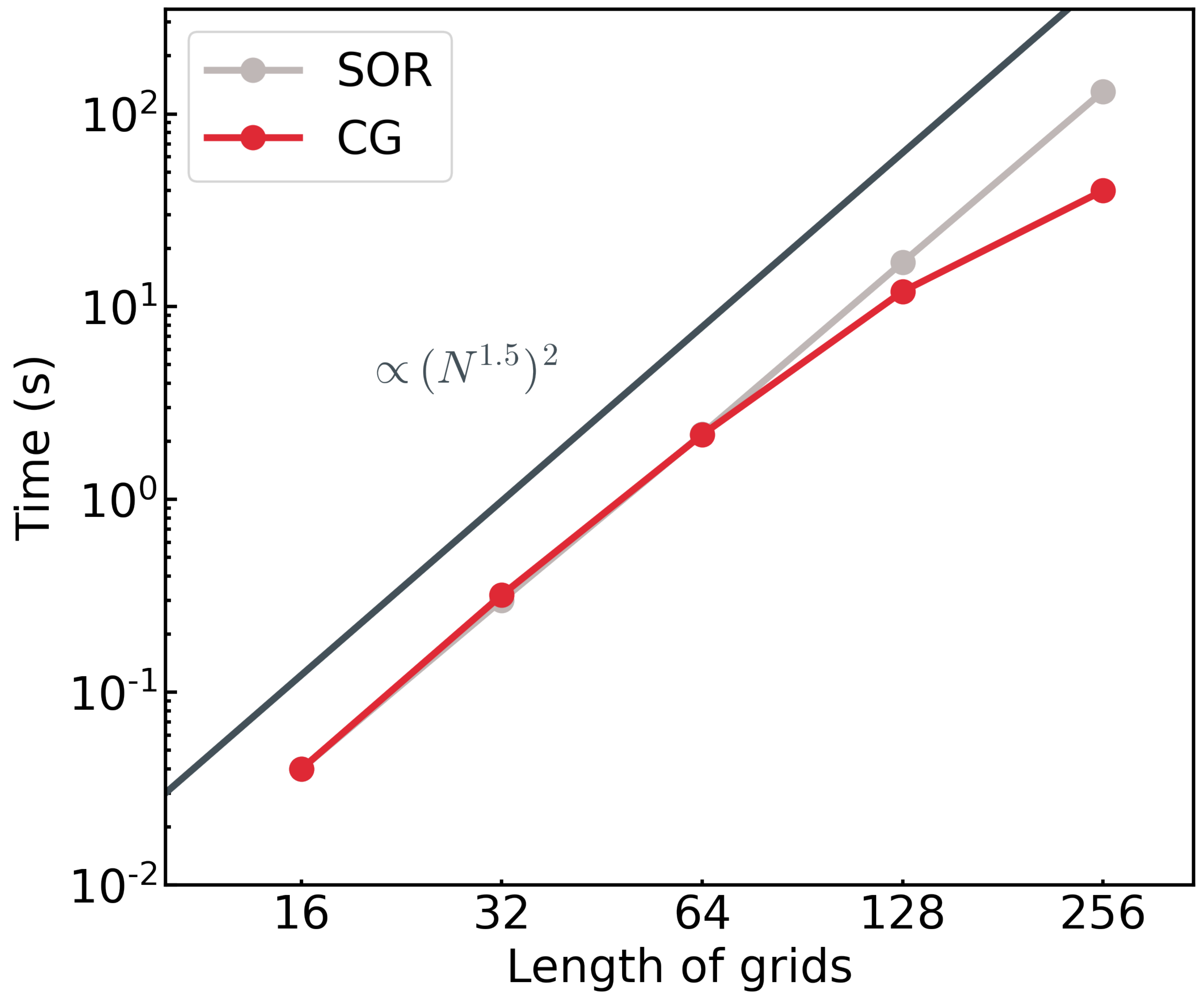
```
while (error > 1e-10) == True:
    Ad = np.array(Adotd(d))
    alpha = np.dot(r.T, r) / np.dot(d.T, Ad)
    x = x + alpha * d  # step to next guess

    rnew = r - alpha * Ad  # update residual r
    beta = np.dot(rnew.T, rnew) / np.dot(r.T, r) #correction
    r = rnew
    d = r + beta * d  # compute new search direction
    error = np.dot(r.T,r)/N**4
```

## Results (N=64)

$$\nabla^2\phi(x,y) = -2\sin x \sin y \quad ; \quad \phi(x,y) = \sin x + \sin y + 1$$



# CG vs. SOR (w/out parallelization)



Theory

Jin-Tong Feng

Implementation

Chih-Yuan Chang

## Parallelization

Yu Voon Ng

# Python (mpi4py)

|  |  |
|---|---|
| Model Name | **MacBook Air** |
| Chip | **Apple M1** |
| Total Number of Cores | **8**<br>**- 4 performance**<br>**- 4 efficiency** |

**Parallelization**

**MPI**

| Python (mpi4py) | C++ |
|---|---|
| `from mpi4py import MPI` | `#include <mpi.h>` |
| Default to call when import the module | `MPI_Init( &argc, &argv );`<br>`MPI_Finalize();` |
| `comm = MPI.COMM_WORLD`<br>`comm.Get_size()`<br>`comm.Get_rank()` | `MPI_Comm_rank( MPI_COMM_WORLD, &MyRank );`<br>`MPI_Comm_size( MPI_COMM_WORLD, &NRank );` |
| `comm.bcast()/`<br>`comm.Bcast()` | `MPI_Bcast()` |

## Which to parallelize?

```
while (error > 1e-10) == True:
    Ad = np.array(Adotd(d))
    alpha = np.dot(r.T, r) / np.dot(d.T, Ad)
    x = x + alpha * d  # step to next guess

    rnew = r - alpha * Ad  # update residual r
    beta = np.dot(rnew.T, rnew) / np.dot(r.T, r) #correction
    r = rnew
    d = r + beta * d  # compute new search direction
    error = np.dot(r.T,r)/N**4
```

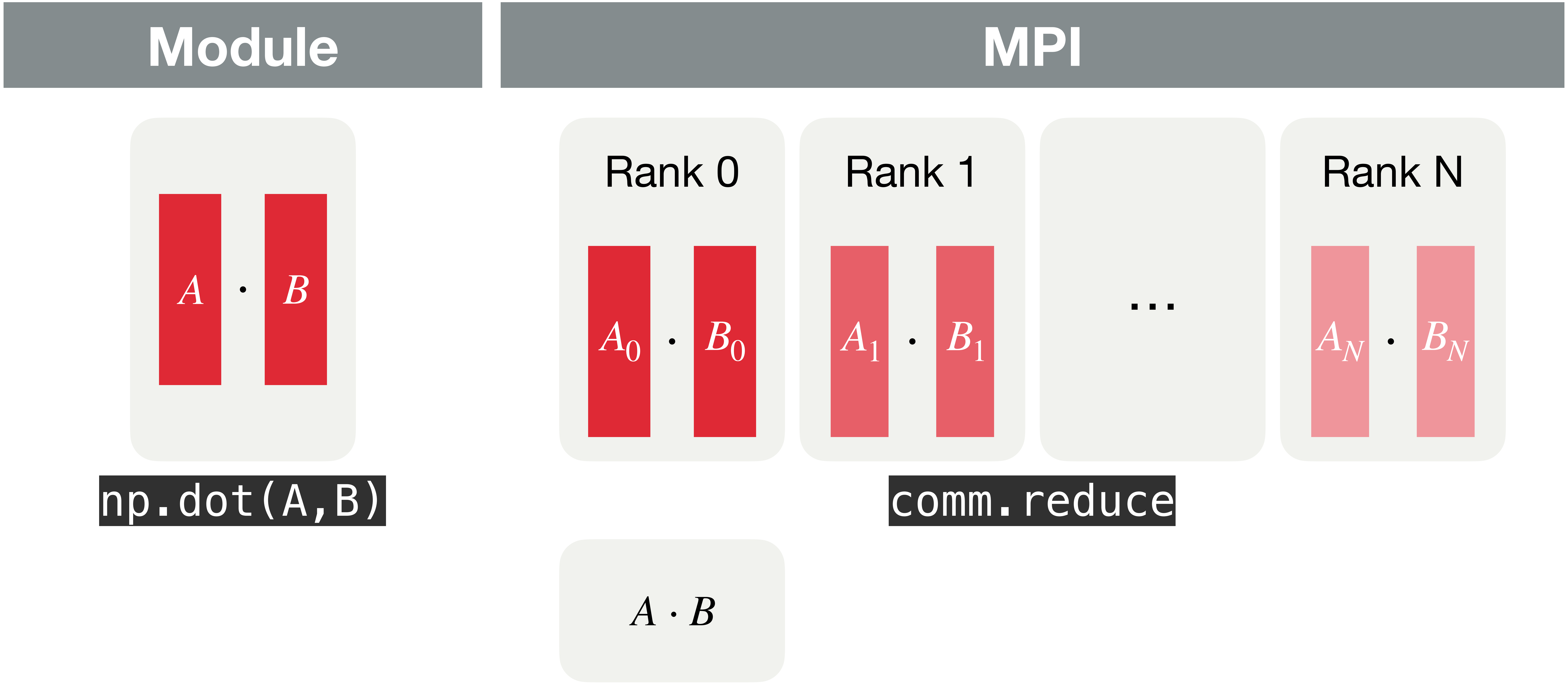
## $A \cdot d$**?**

```
while (error > 1e-10) == True:
    Ad = np.array(Adotd(d))
    alpha = np.dot(r.T, r) / np.dot(d.T, Ad)
    x = x + alpha * d  # step to next guess

    rnew = r - alpha * Ad  # update residual r
    beta = np.dot(rnew.T, rnew) / np.dot(r.T, r) #correction
    r = rnew
    d = r + beta * d  # compute new search direction
    error = np.dot(r.T,r)/N**4
```

## Dot product?

```
while (error > 1e-10) == True:
    Ad = np.array(Adotd(d))
    alpha = np.dot(r.T, r) / np.dot(d.T, Ad)
    x = x + alpha * d  # step to next guess

    rnew = r - alpha * Ad  # update residual r
    beta = np.dot(rnew.T, rnew) / np.dot(r.T, r) #correction
    r = rnew
    d = r + beta * d  # compute new search direction
    error = np.dot(r.T,r)/N**4
```

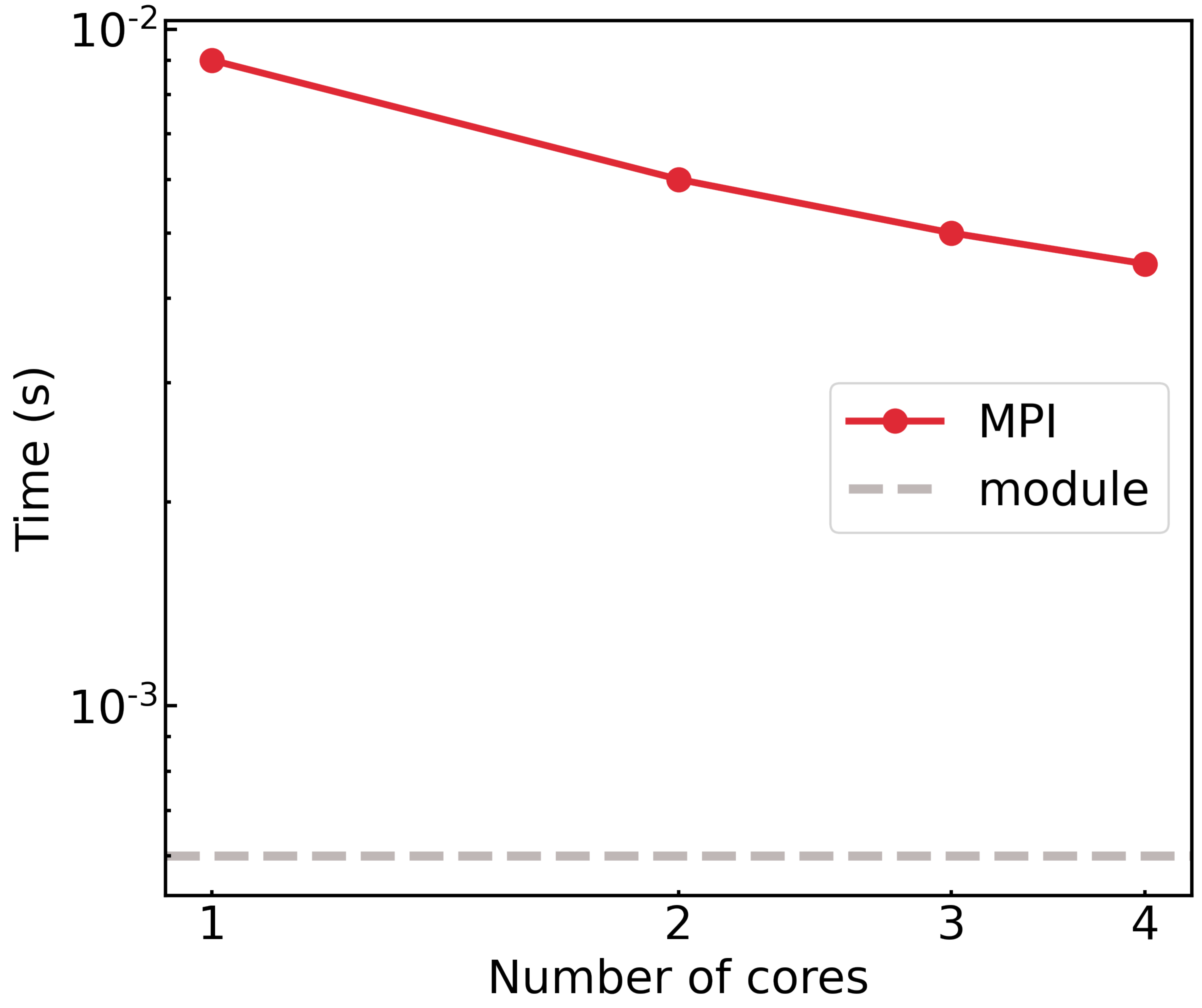
## Dot product?



`np.dot(A,B)`

`comm.reduce`

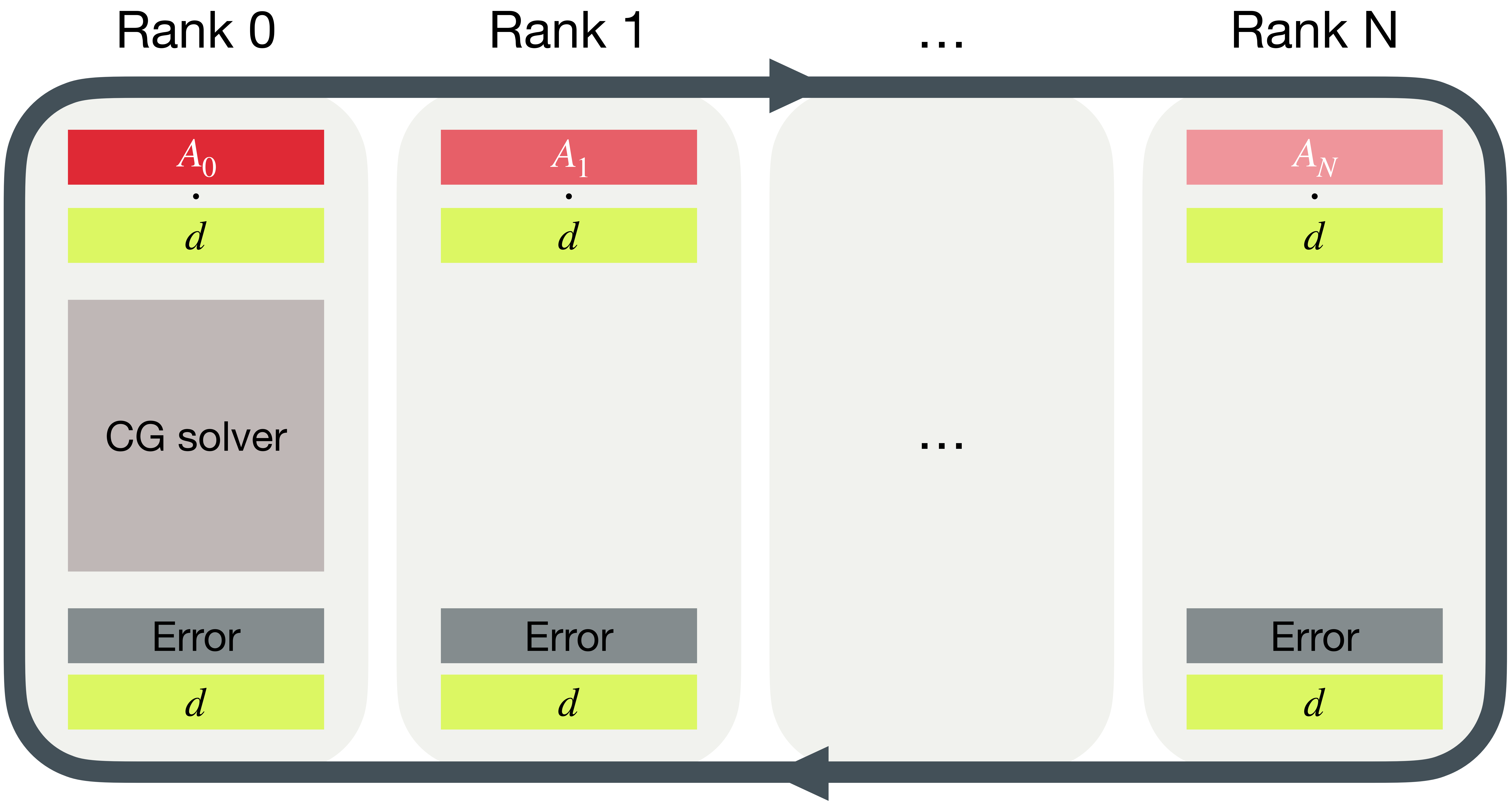
# Dot product?

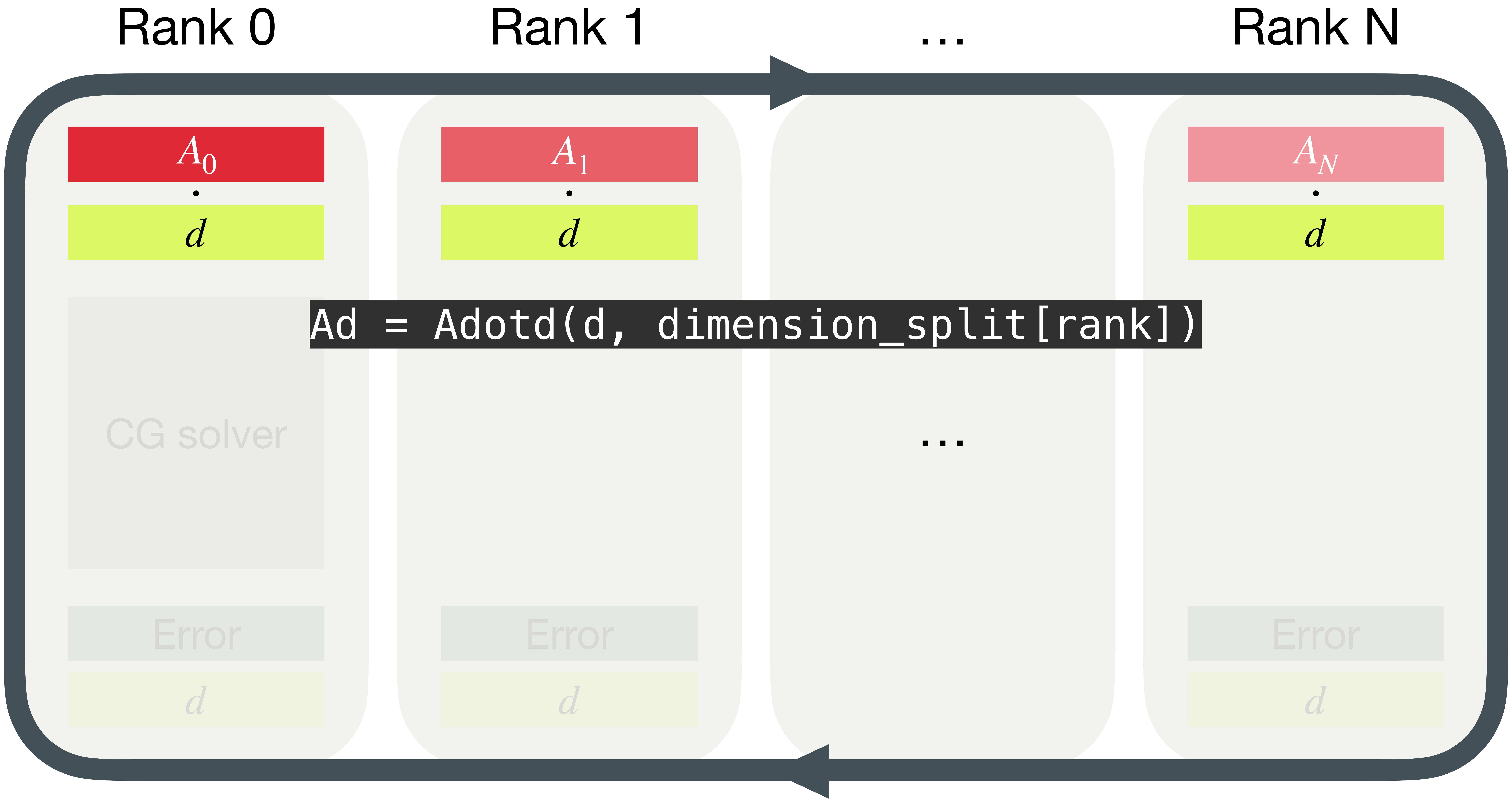


- Length = 1,000,000
- **MPI** much lower efficiency
- **Module** uses *optimized BLAS library*

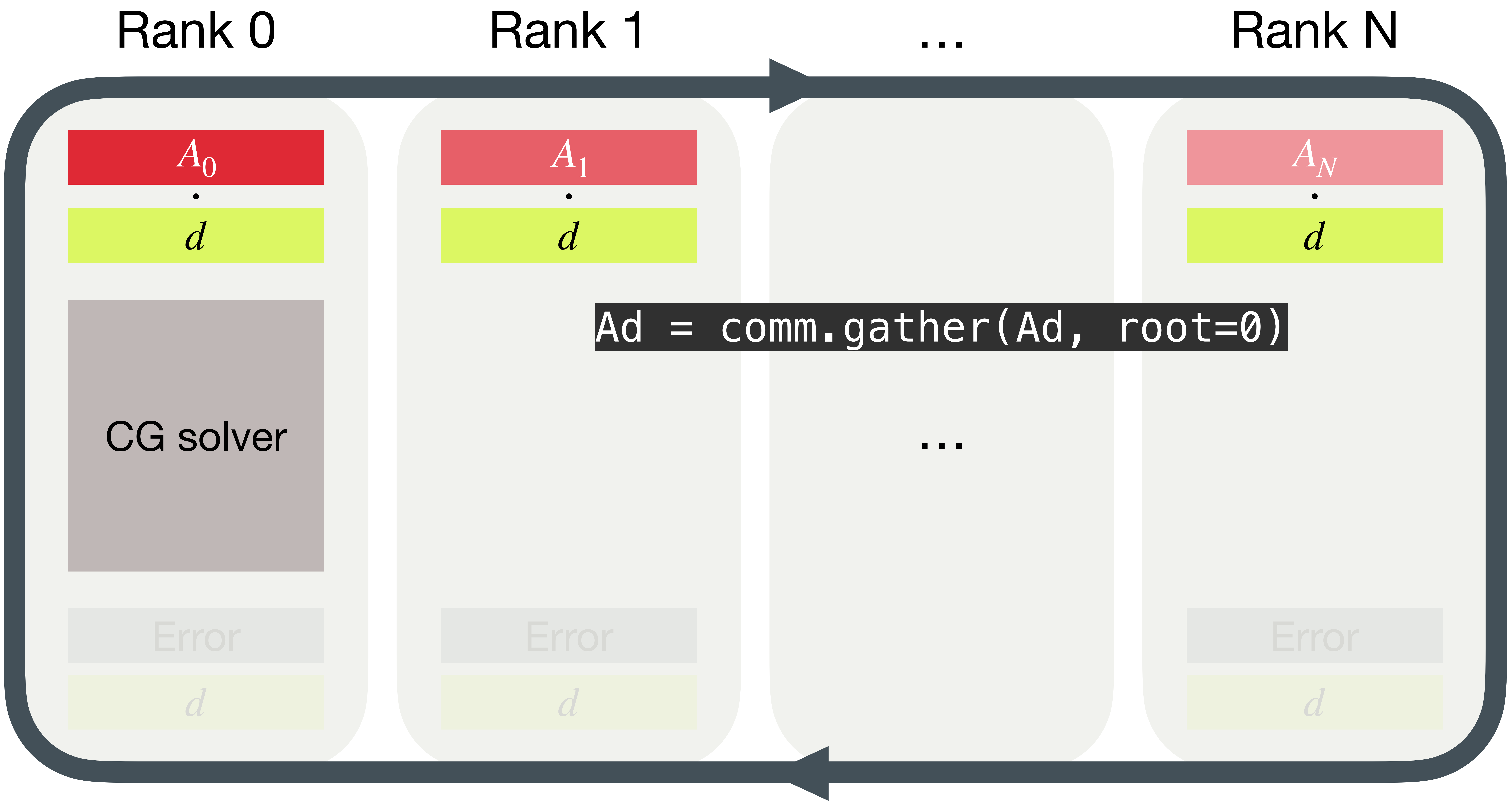
$A \cdot d$

```
while (error > 1e-10) == True:
    Ad = np.array(Adotd(d))
    alpha = np.dot(r.T, r) / np.dot(d.T, Ad)
    x = x + alpha * d  # step to next guess

    rnew = r - alpha * Ad  # update residual r
    beta = np.dot(rnew.T, rnew) / np.dot(r.T, r) #correction
    r = rnew
    d = r + beta * d  # compute new search direction
    error = np.dot(r.T,r)/N**4
```

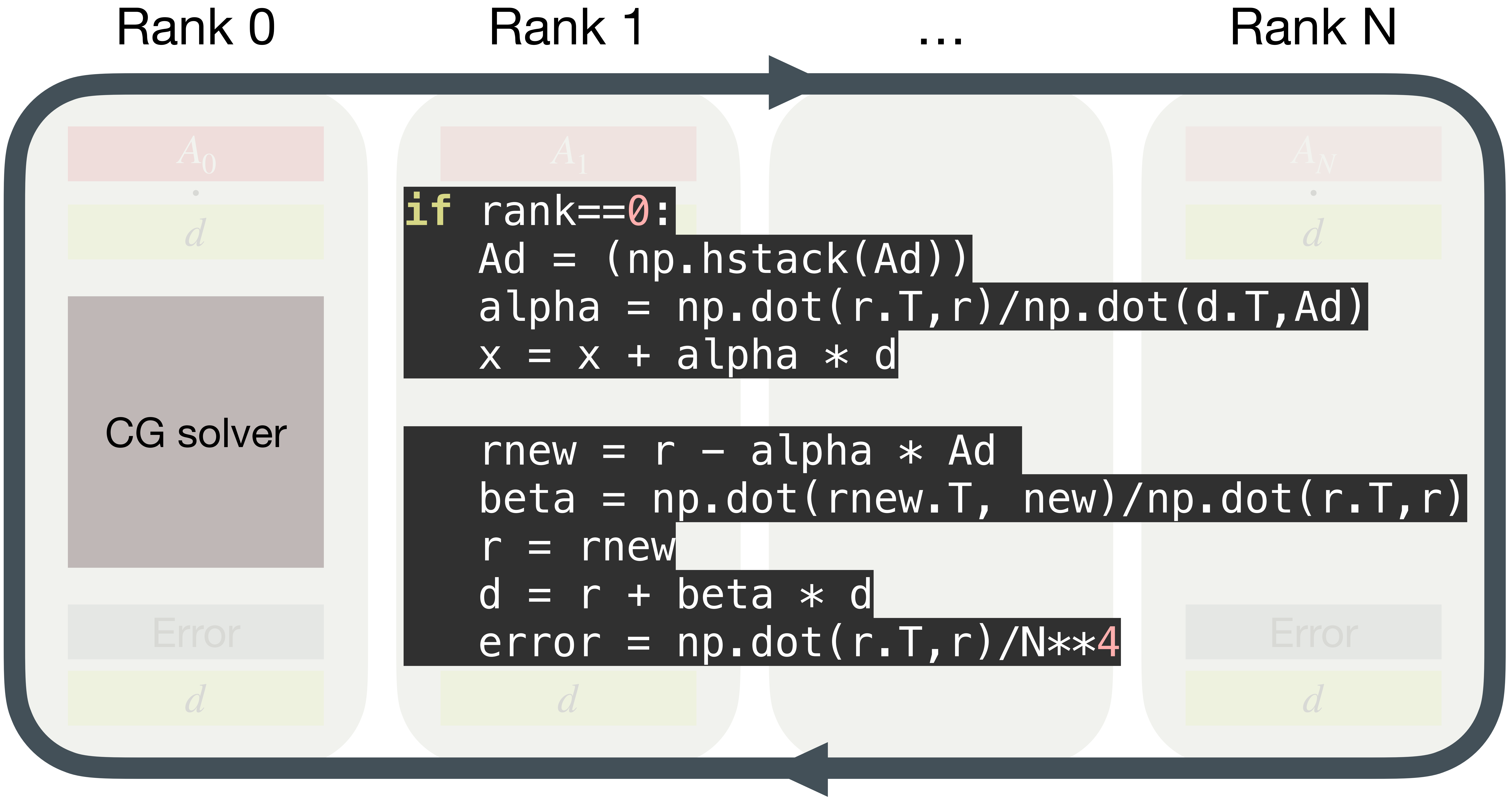

Rank 0
Rank 1
…
Rank N
$A_0$
·
$d$
CG solver
Error
$d$
$A_1$
·
$d$
Error
$d$
…
$A_N$
·
$d$
Error
$d$



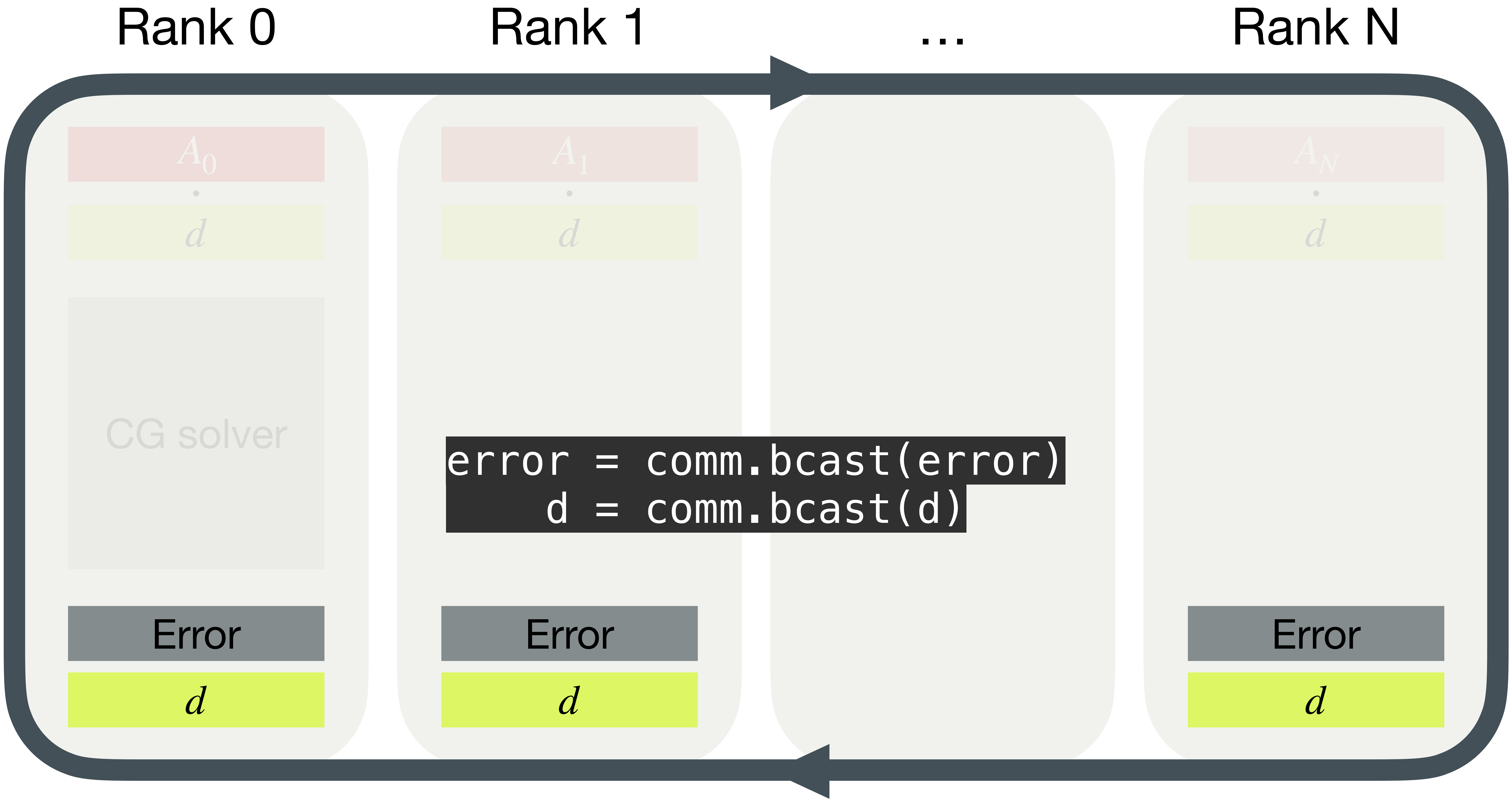
```
Ad = Adotd(d, dimension_split[rank])
```



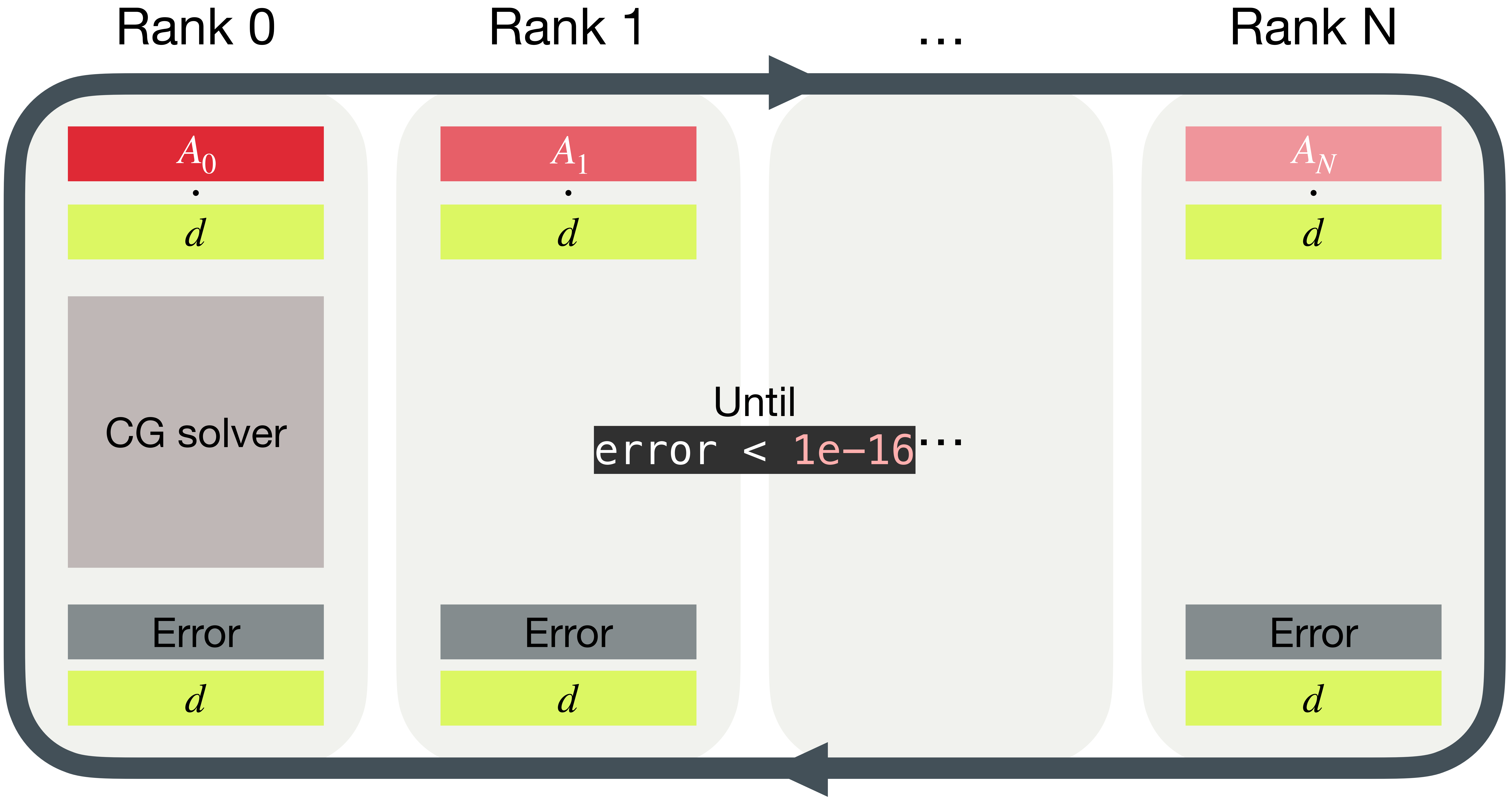
```
Ad = comm.gather(Ad, root=0)
```



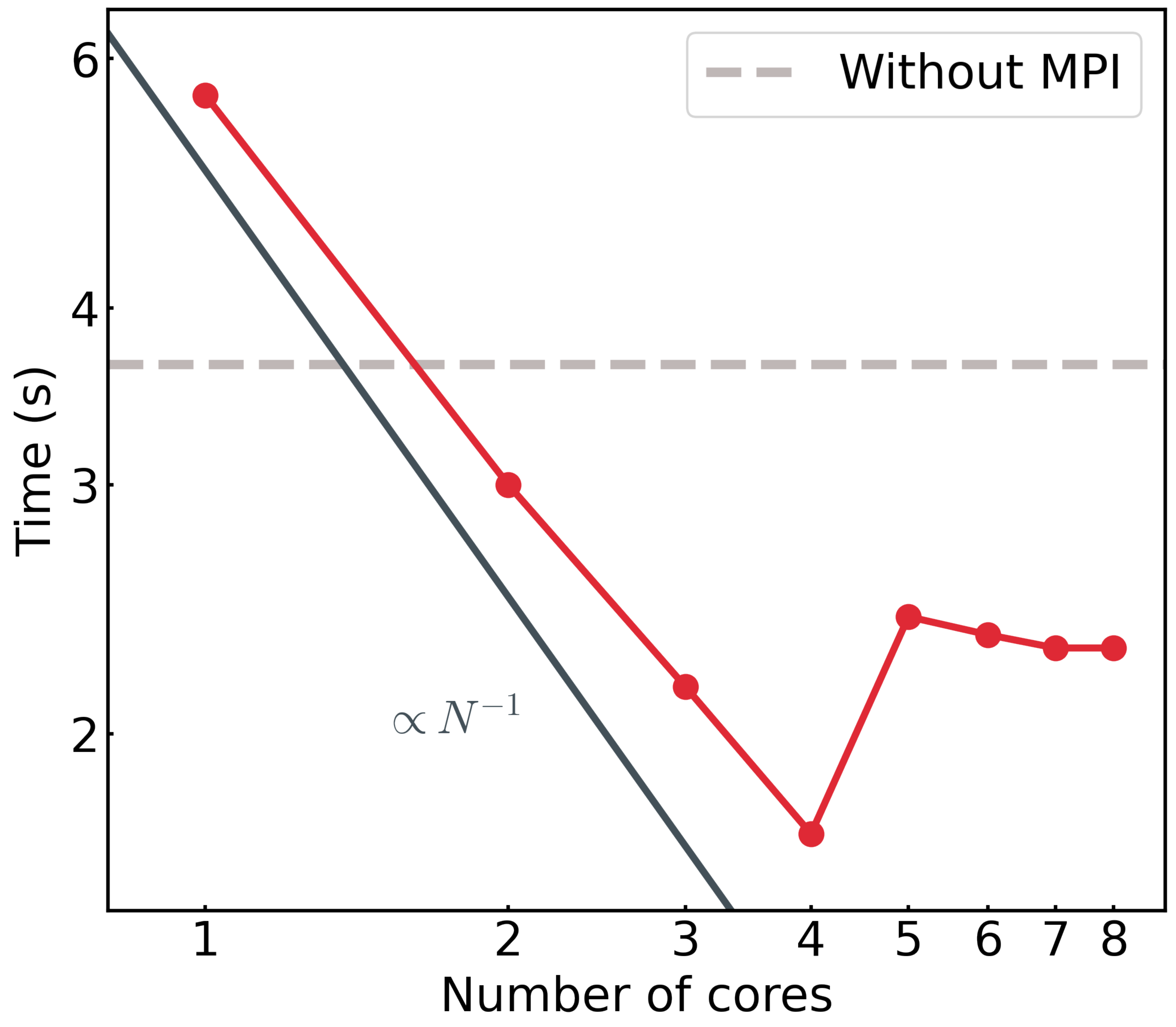
```
if rank==0:
   Ad = (np.hstack(Ad))
   alpha = np.dot(r.T,r)/np.dot(d.T,Ad)
   x = x + alpha * d

   rnew = r - alpha * Ad
   beta = np.dot(rnew.T, new)/np.dot(r.T,r)
   r = rnew
   d = r + beta * d
   error = np.dot(r.T,r)/N**4
```



```
error = comm.bcast(error)
    d = comm.bcast(d)
```

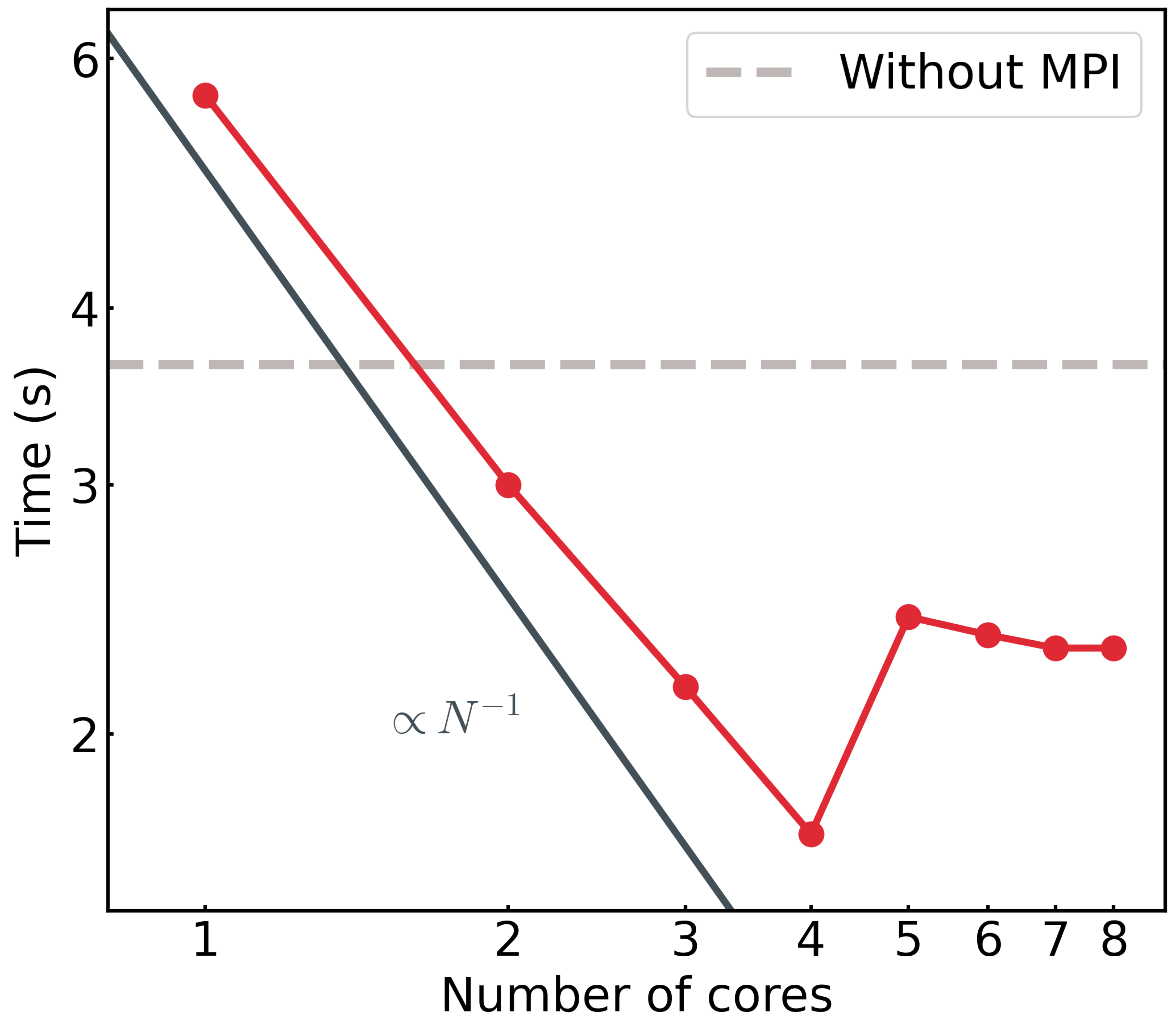

Rank 0
Rank 1
...
Rank N
$A_0$
$\cdot$
$d$
$A_1$
$\cdot$
$d$
$A_N$
$\cdot$
$d$
CG solver
Until
error < 1e-16
...
Error
$d$
Error
$d$
Error
$d$

# Cores vs. Time

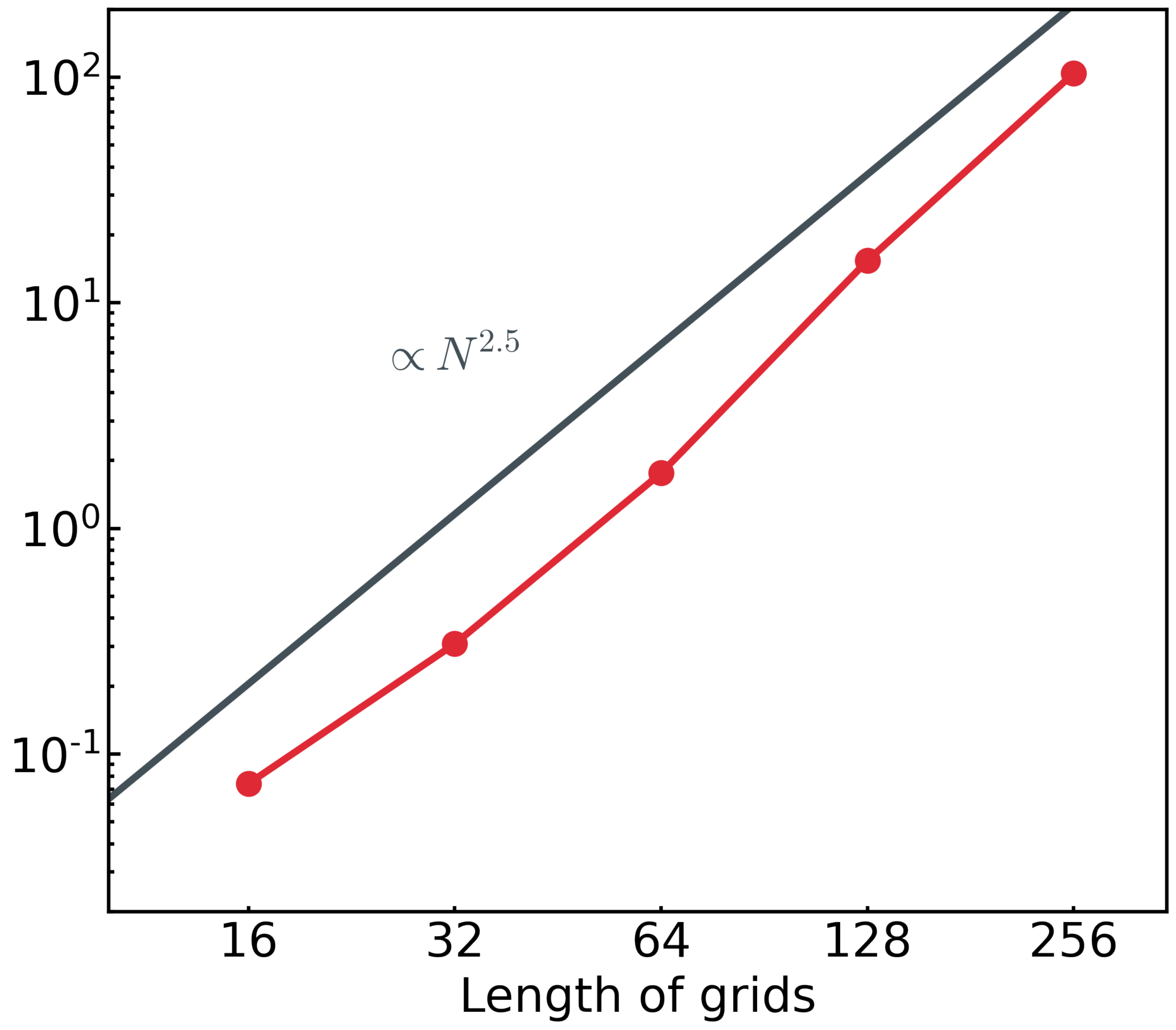


- 64×64 grids
- Saturated after 4 cores
- Spend time at **gathering** and **broadcasting**

## Cores vs. Time

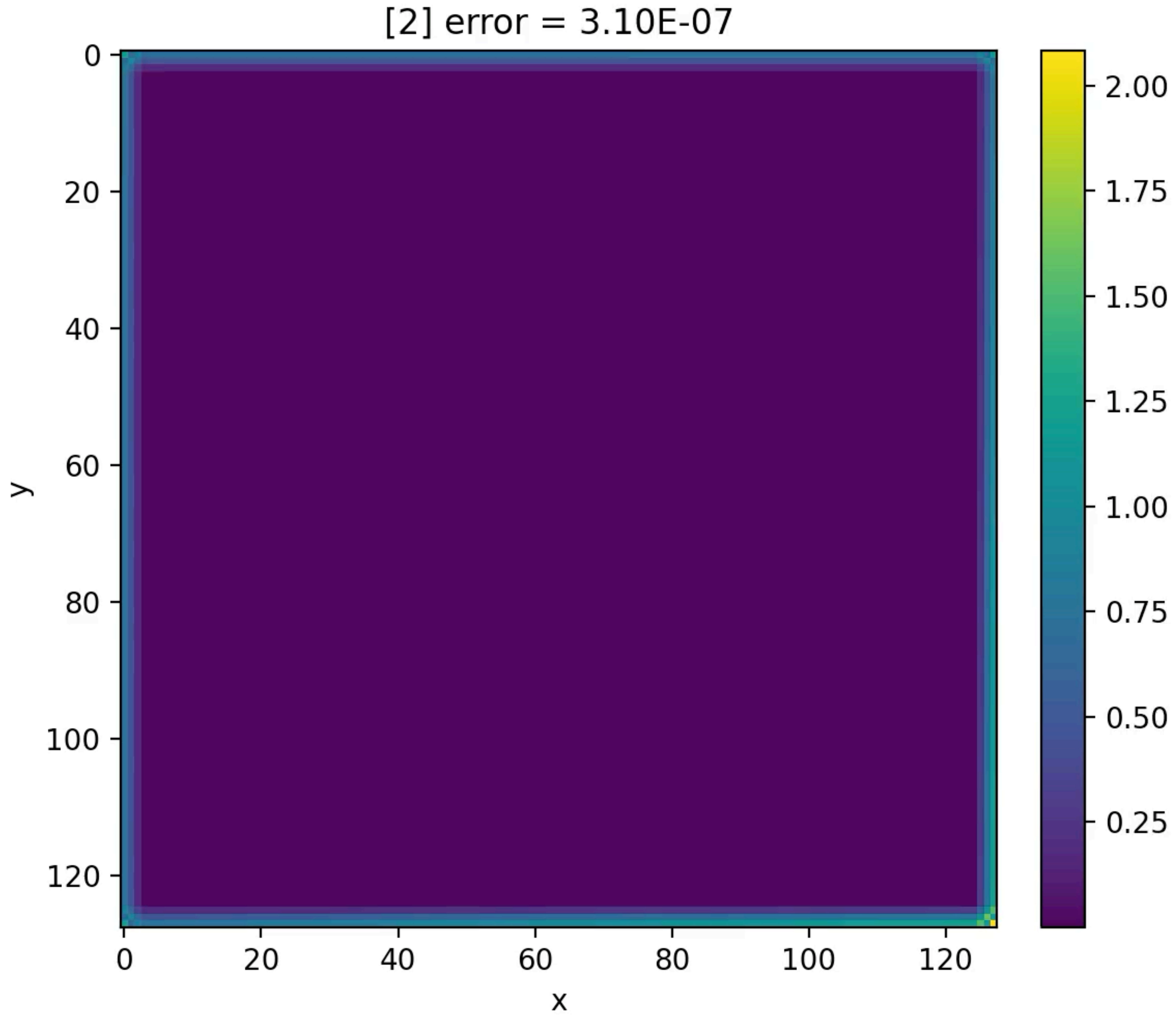


## Grids vs. Time



## Conclusion

- Conjugate gradient method is fast
- With MPI parallelization, it is faster



## Future work

### Precondition

- For a normal matrix, the condition number is defined as

$$\kappa(A) = \frac{|\lambda_{max}(A)|}{|\lambda_{min}(A)|}$$

which can be used to evaluate the sensitivity of a function to a small change.

- Suppose that we have a matrix M with

$$\kappa(M^{-1}A) \ll \kappa(A)$$

$\Rightarrow$

So, we can turn to solve

$$M^{-1}A\vec{x} = M^{-1}\vec{b}$$

## Future work

### Precondition

- Jacobi precondition uses

$$M = diag(A)$$

- Cholesky precondition uses

$$M = L\text{, where } A = LL^T$$