JAPAN-KOREA HPC WINTER SCHOOL

JAPAN-KOREA HPC WINTER SCHOOL

- Parallel numerical algorithms -

Hiroto Tadano

tadano@cs.tsukuba.ac.jp

Center for Computational Sciences

University of Tsukuba

Contents

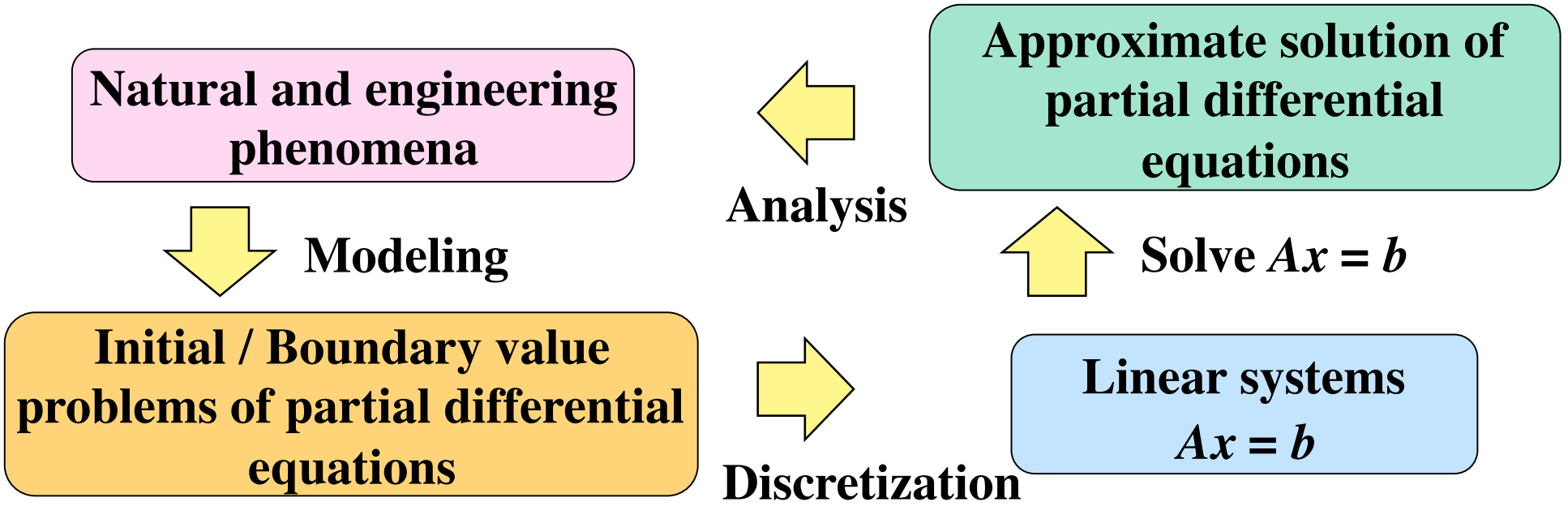
- **Methods for solving linear systems $Ax = b$**
  - Krylov subspace iterative methods
  - Storage formats for sparse matrices
  - Parallelization of basic linear algebra calculations
  - Preconditioning

- **Methods for solving linear systems with multiple right-hand sides $AX = B$**
  - Block Krylov subspace iterative methods
  - Parallelization with OpenMP

Methods for solving linear systems

$$Ax = b$$

Analysis of natural and engineering phenomena



Linear equations appears in many scientific applications.

However, the solution of linear systems is the most time-consuming part.

Linear systems

Linear systems : $Ax = b$

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Linear systems appear in many scientific applications.

However, the solution of linear systems is the most time-consuming part.

Direct methods and iterative methods

Direct methods

Gaussian elimination, LU factorization, etc.

1) We can always obtain solution in a finite number of operations.

2) Number of nonzero elements increases in transformation of coefficient matrix A.

We cannot utilize coefficient matrix sparsity.

Direct methods

● Gaussian elimination method

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

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

$$U\boldsymbol{x} = \boldsymbol{b}'$$

$$\begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ & u_{22} & \dots & u_{2n} \\ & & \ddots & \vdots \\ 0 & & & u_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{bmatrix}$$

● LU decomposition method

$$LU\boldsymbol{x} = \boldsymbol{b}$$

The coefficient matrix A is only transformed.

$$\begin{bmatrix} 1 & & & \\ l_{21} & 1 & & \\ \vdots & \vdots & \ddots & \\ l_{n1} & l_{n2} & \dots & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ & u_{22} & \dots & u_{2n} \\ & & \ddots & \vdots \\ 0 & & & u_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Direct methods: Gaussian Elimination

Step 1.

Transform the matrix A of the linear system $A\boldsymbol{x} = \boldsymbol{b}$ to an upper triangular matrix U.

- Computational complexity : $n^3 / 3$.

$$\underbrace{\begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & u_{nn} \end{bmatrix}}_{U} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}}_{\boldsymbol{x}} = \underbrace{\begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{bmatrix}}_{\boldsymbol{b}'}$$

Step 2.

Solve the linear system $U\boldsymbol{x} = \boldsymbol{b}'$ by backward substitution with the following recursion formula.

$$x_i = (b'_i - u_{i,i+1}x_{i+1} - \ldots - u_{i,n}x_n)/u_{i,i}, \quad i = n, n-1, \ldots, 1$$

- Computational complexity : $n^2 / 2$.

Direct methods: LU decomposition

Step 1.

Perform the LU decomposition of the coefficient matrix A.

$$A = LU$$

L : Lower triangular matrix, U : Upper triangular matrix.

- Computational complexity : $n^3 / 3$.

$$\underbrace{\begin{bmatrix} 1 & & 0 \\ l_{2,1} & 1 & \\ \vdots & \vdots & \ddots \\ l_{n,1} & l_{n,2} & \dots \quad 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} u_{1,1} & u_{1,2} & \dots & u_{1,n} \\ & u_{2,2} & \dots & u_{2,n} \\ & & \ddots & \vdots \\ 0 & & & u_{n,n} \end{bmatrix}}_{U} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Direct methods: LU decomposition

Step 2. Find $\boldsymbol{x}$ using forward / backward substitution.

1) Solve $L\boldsymbol{y} = \boldsymbol{b}$ for $\boldsymbol{y}$ by forward substitution. Here, $\boldsymbol{y} = U\boldsymbol{x}$.

$$\begin{bmatrix} 1 & & & 0 \\ l_{2,1} & 1 & & \\ \vdots & \vdots & \ddots & \\ l_{n,1} & l_{n,2} & \dots & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

2) Solve $U\boldsymbol{x} = \boldsymbol{y}$ for $\boldsymbol{x}$ by backward substitution.

$$\begin{bmatrix} u_{1,1} & u_{1,2} & \dots & u_{1,n} \\ & u_{2,2} & \dots & u_{2,n} \\ & & \ddots & \vdots \\ 0 & & & u_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

Once LU decomposition has been performed, solutions can be found for other right-hand vectors with a computational complexity of n^2.

Direct methods and iterative methods

Iterative methods

Krylov subspace methods

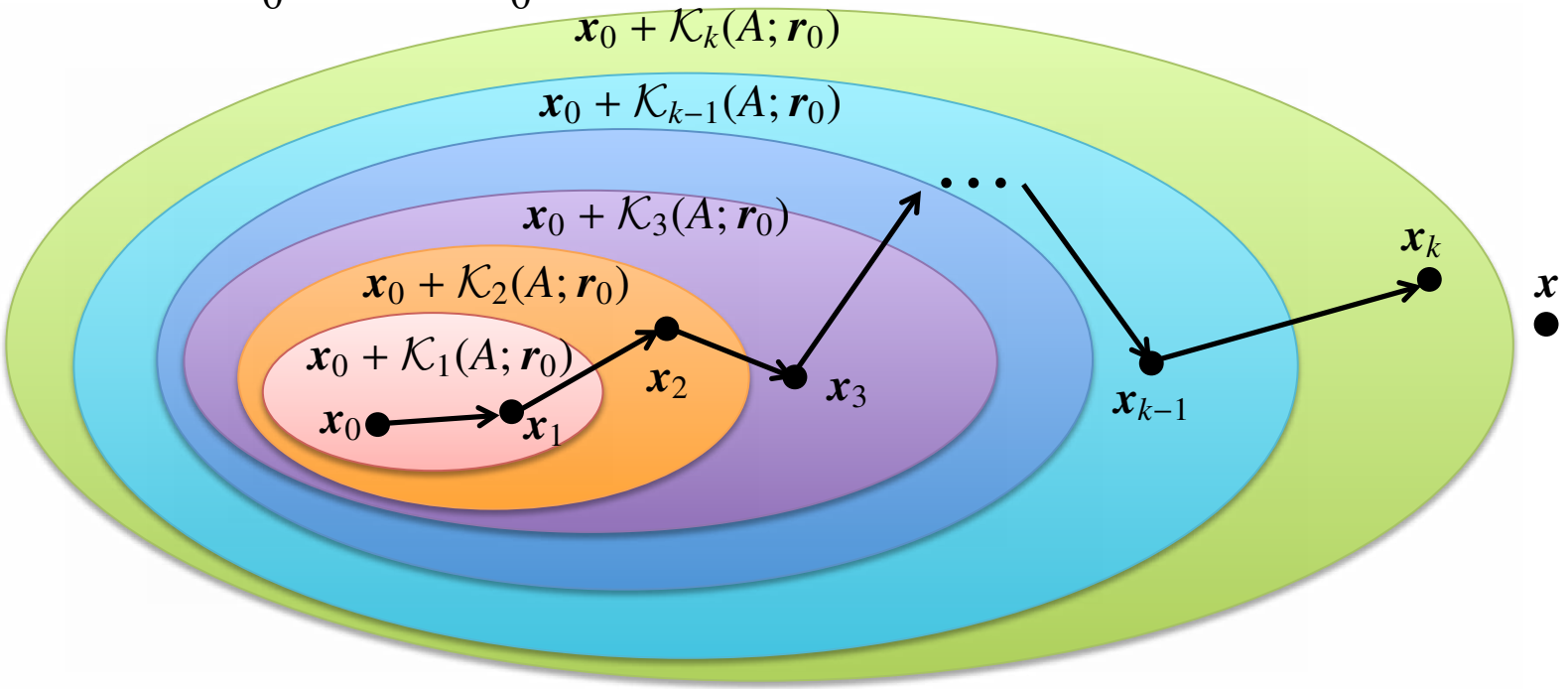
1) Required operations are
   - Multiplication of a coefficient matrix and a vector : $A\boldsymbol{u}$
   - Inner product of vectors : $(\boldsymbol{u}, \boldsymbol{v}) = \boldsymbol{u}^{\mathrm{H}}\boldsymbol{v}$
   - Constant times a vector plus a vector : $a\boldsymbol{u} + \boldsymbol{v}$

   ⇒ We can utilize coefficient matrix sparsity.

2) Some problems may require many number of iterations

Krylov subspace methods

- $\boldsymbol{x}_0$ is an initial guess. The vector $\boldsymbol{x}_k$ is k-th approximate solution of the linear system $A\boldsymbol{x} = \boldsymbol{b}$. $\boldsymbol{x}_k$ is updated by the iteration process.
- $\mathcal{K}_j(A; \boldsymbol{r}_0)$ is called a Krylov subspace. This subspace is spanned by the vectors $\boldsymbol{r}_0, A\boldsymbol{r}_0, \ldots, A^{j-1}\boldsymbol{r}_0$.
- The vector $\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$ is called an initial residual vector.



Sketch of Krylov subspace methods.

Methods for Hermitian matrix

1. Coefficient matrix is an Hermitian matrix ($A = A^{\mathrm{H}}$)

- Conjugate **G**radient (**CG**) method
- Conjugate **R**esidual (**CR**) method
- **Min**imal **Res**idual (**MINRES**) method

Use of Hermitian property of the coefficient matrix enables derivation of algorithms with short recurrence formula (low computational complexity).

Note : Hermitian matrix

$$A = A^{\mathrm{H}} = \bar{A}^{\mathrm{T}}$$

$$(a_{ij} = \bar{a}_{ji})$$

Methods for Hermitian matrix

$\boldsymbol{x}_0$ is an initial guess,

Compute $\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$,

Set $\boldsymbol{p}_0 = \boldsymbol{r}_0$,

For $k = 0, 1, \ldots,$ **until** $\|\boldsymbol{r}_k\|_2 \leq \varepsilon_{\mathrm{TOL}} \|\boldsymbol{b}\|_2$ **do** :

$$\boldsymbol{q}_k = A\boldsymbol{p}_k,$$ ← **Matrix-vector multiplication**

$$\alpha_k = \frac{(\boldsymbol{r}_k, \boldsymbol{r}_k)}{(\boldsymbol{p}_k, \boldsymbol{q}_k)},$$ ← **Inner product**

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k,$$

$$\boldsymbol{r}_{k+1} = \boldsymbol{r}_k - \alpha_k \boldsymbol{q}_k,$$ ← **Constant times a vector plus a vector**

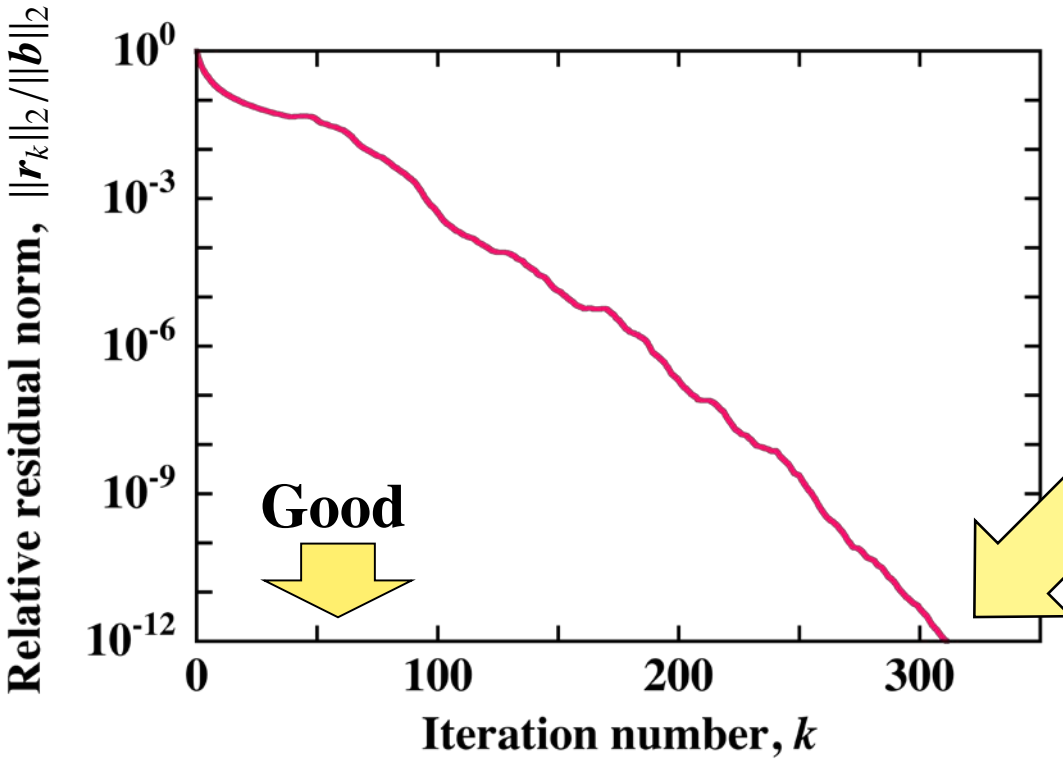
$$\beta_k = \frac{(\boldsymbol{r}_{k+1}, \boldsymbol{r}_{k+1})}{(\boldsymbol{r}_k, \boldsymbol{r}_k)},$$ ← **Inner product**

$$\boldsymbol{p}_{k+1} = \boldsymbol{r}_{k+1} + \beta_k \boldsymbol{p}_k,$$ ← **Constant times a vector plus a vector**

End For

Algorithm of the Conjugate Gradient (CG) method.

Relative residual history of the CG method



In this figure, the iteration is stopped when the condition

$$\|r_k\|_2/\|b\|_2 \leq 10^{-12}$$

is satisfied.

The relative residual norm $\|r_k\|_2/\|b\|_2$ is monitored during the iterations.
If the condition $\|r_k\|_2/\|b\|_2 \leq \varepsilon_{\mathrm{TOL}}$ is satisfied, the iteration is stopped.
Then, the approximate solution x_k is employed as the solution.

Methods for non-Hermitian matrix

2. Coefficient matrix is a non-Hermitian matrix ($A \neq A^{H}$)

Methods derived from residual bi-orthobonality condition

- **Bi**-Conjugate **G**radient (**BiCG**) method
- Conjugate **G**radient **S**quared (**CGS**) method
- **BiCG Stab**ilization (**BiCGSTAB**) method

Computational complexity is low, but decrease of residual norm is non-monotonic.

Methods derived from residual norm minimization condition

- **G**eneralized **C**onjugate **R**esidual (**GCR**) method
- **G**eneralized **M**inimal **Res**idual (**GMRES**) method

Residual norm decreases monotonically, but long recurrence formula requires.

Methods for non-Hermitian matrix

$\boldsymbol{x}_0$ is an initial guess,

Compute $\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$,

Choose $\boldsymbol{r}_0^*$ such that $(\boldsymbol{r}_0^*, \boldsymbol{r}_0) \neq 0$,

Set $\boldsymbol{p}_0 = \boldsymbol{r}_0$ and $\boldsymbol{p}_0^* = \boldsymbol{r}_0^*$,

For $k = 0, 1, \ldots,$ **until** $\|\boldsymbol{r}_k\|_2 \leq \varepsilon_{\mathrm{TOL}} \|\boldsymbol{b}\|_2$ **do** :

$$\boldsymbol{q}_k = A\boldsymbol{p}_k, \qquad \boldsymbol{q}_k^* = A^{\mathrm{H}}\boldsymbol{p}_k^*,$$

$$\alpha_k = \frac{(\boldsymbol{r}_k^*, \boldsymbol{r}_k)}{(\boldsymbol{p}_k^*, \boldsymbol{q}_k)},$$

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k,$$

$$\boldsymbol{r}_{k+1} = \boldsymbol{r}_k - \alpha_k \boldsymbol{q}_k, \qquad \boldsymbol{r}_{k+1}^* = \boldsymbol{r}_k^* - \bar{\alpha}_k \boldsymbol{q}_k^*,$$

$$\beta_k = \frac{(\boldsymbol{r}_{k+1}^*, \boldsymbol{r}_{k+1})}{(\boldsymbol{r}_k^*, \boldsymbol{r}_k)},$$

$$\boldsymbol{p}_{k+1} = \boldsymbol{r}_{k+1} + \beta_k \boldsymbol{p}_k, \qquad \boldsymbol{p}_{k+1}^* = \boldsymbol{r}_{k+1}^* + \bar{\beta}_k \boldsymbol{p}_k^*,$$

End For

Matrix-vector multiplication

Inner product

Constant times a vector plus a vector

Algorithm of the Bi-Conjugate Gradient (BiCG) method

Methods for non-Hermitian matrix

$\boldsymbol{x}_0$ is an initial guess,
Compute $\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$,
Set $\boldsymbol{p}_0 = \boldsymbol{r}_0$ and $\boldsymbol{q}_0 = \boldsymbol{s}_0 = A\boldsymbol{r}_0$,
For $k = 0, 1, \ldots,$ **until** $\|\boldsymbol{r}_k\|_2 \leq \varepsilon_{\mathrm{TOL}}\|\boldsymbol{b}\|_2$ **do** :

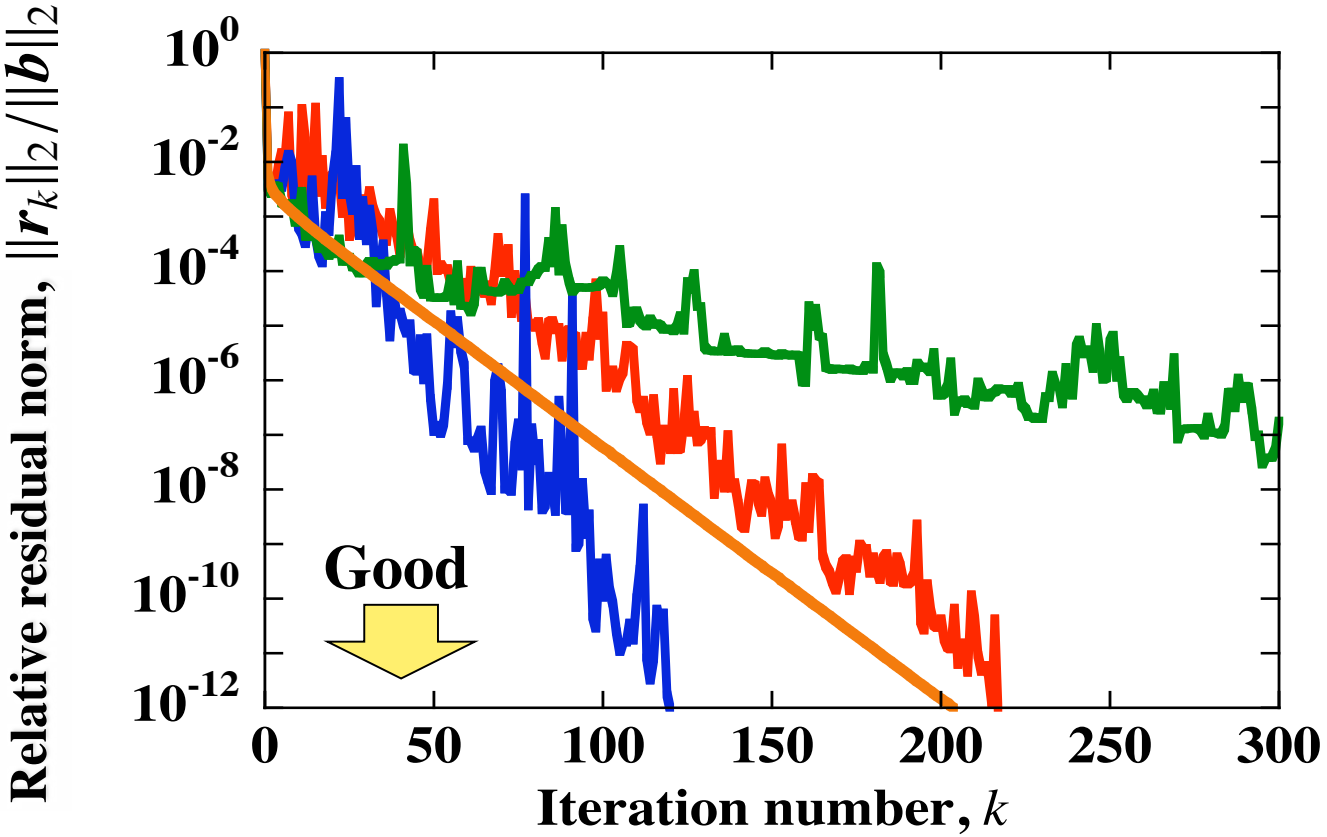
$$
\begin{aligned}
\alpha_k &= \frac{(\boldsymbol{q}_k, \boldsymbol{r}_k)}{(\boldsymbol{q}_k, \boldsymbol{q}_k)}, \\
\boldsymbol{x}_{k+1} &= \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k, \\
\boldsymbol{r}_{k+1} &= \boldsymbol{r}_k - \alpha_k \boldsymbol{q}_k, \\
\boldsymbol{s}_{k+1} &= A\boldsymbol{r}_{k+1}, \\
\beta_{k,i} &= -\frac{(\boldsymbol{q}_i, \boldsymbol{s}_{k+1})}{(\boldsymbol{q}_i, \boldsymbol{q}_i)}, \quad (i = 0, 1, \ldots, k) \\
\boldsymbol{p}_{k+1} &= \boldsymbol{r}_{k+1} + \sum_{i=0}^{k} \beta_{k,i} \boldsymbol{p}_i, \\
\boldsymbol{q}_{k+1} &= \boldsymbol{s}_{k+1} + \sum_{i=0}^{k} \beta_{k,i} \boldsymbol{q}_i,
\end{aligned}
$$

End For

- **One matrix-vector multiplication per iteration**
- **Large computational complexity and memory requirement due to long recurrence formulae**
- **Computational complexity and memory requirement can be reduced by Restart**

Algorithm of the Generalized Conjugate Residual (GCR) method

Convergence properties of iterative methods



Relative residual histories of iterative methods.

– : BiCG, – : CGS, – : BiCGSTAB, – : GCR.

Method for complex symmetric matrix

3. Coefficient matrix is a complex symmetric matrix ($A = A^{\mathrm{T}} \neq A^{\mathrm{H}}$)

- Conjugate **O**rthogonal **C**onjugate **G**radient (**COCG**) method

If the coefficient matrix is a complex symmetric matrix, computation can be performed with one matrix-vector multiplication per iteration and a short recurrence formula.

Note : Complex symmetric matrix

$$A = A^{\mathrm{T}} \neq A^{\mathrm{H}}$$

$$(a_{ij} = a_{ji} \neq \bar{a}_{ji})$$

Method for complex symmetric matrix

$\boldsymbol{x}_0$ is an initial guess,

Compute $\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$,

Set $\boldsymbol{p}_0 = \boldsymbol{r}_0$,

For $k = 0, 1, \ldots,$ **until** $\|\boldsymbol{r}_k\|_2 \leq \varepsilon_{\mathrm{TOL}} \|\boldsymbol{b}\|_2$ **do** :

$$\boldsymbol{q}_k = A\boldsymbol{p}_k,$$ ← **Matrix-vector multiplication**

$$\alpha_k = \frac{(\bar{\boldsymbol{r}}_k, \boldsymbol{r}_k)}{(\bar{\boldsymbol{p}}_k, \boldsymbol{q}_k)},$$ ← **Inner product**

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k,$$

$$\boldsymbol{r}_{k+1} = \boldsymbol{r}_k - \alpha_k \boldsymbol{q}_k,$$ ← **Constant times a vector and plus a vector**

$$\beta_k = \frac{(\bar{\boldsymbol{r}}_{k+1}, \boldsymbol{r}_{k+1})}{(\bar{\boldsymbol{r}}_k, \boldsymbol{r}_k)},$$ ← **Inner product**

$$\boldsymbol{p}_{k+1} = \boldsymbol{r}_{k+1} + \beta_k \boldsymbol{p}_k,$$ ← **Constant times a vector and plus a vector**

End For

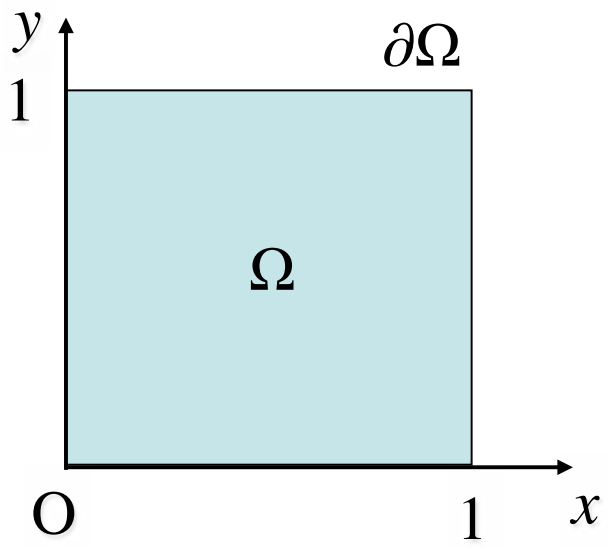
Algorithm of the **COCG** method

Example of sparse matrix

2D Poisson problem

$$\begin{cases} \dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2} = f, & \text{in } \Omega \\ u = \bar{u}, & \text{on } \partial\Omega \end{cases}$$

f, $\bar{u}$ are given functions



Ω is divided into $(M+1)$ equal parts in x, y directions and discretized by central difference with 5-points.

We obtain a linear system with matrix of order $M \times M$

Total number of elements in matrix : M^4

Number of nonzero elements : $5M^2 - 4M$

Sparse matrix storage format

Compressed **R**ow **S**torage (**CRS**) format
Search row-wise for nonzero elements

$$A = \begin{bmatrix} a_{11} & 0 & a_{13} & 0 & a_{15} \\ 0 & a_{22} & 0 & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & 0 & 0 \\ 0 & 0 & a_{43} & a_{44} & 0 \\ 0 & a_{52} & 0 & a_{54} & a_{55} \end{bmatrix}$$

`val` stores nonzero elements of A.

`col_ind` stores column number of nonzero elements of A.

`row_ptr` stores location of first nonzero element in each row.

`val:`

a_{11}	a_{13}	a_{15}	a_{22}	a_{24}	a_{25}	a_{31}	a_{32}	a_{33}	a_{43}	a_{44}	a_{52}	a_{54}	a_{55}

`col_ind:`

1	3	5	2	4	5	1	2	3	3	4	2	4	5

`row_ptr:`

1	4	7	10	12	15

The last entry is the number of nonzero elements + 1

Sparse matrix storage format

Compressed **C**olumn **S**torage (**CCS**) format
Search column-wise for nonzero elements

$$A = \begin{bmatrix} a_{11} & 0 & a_{13} & 0 & a_{15} \\ 0 & a_{22} & 0 & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & 0 & 0 \\ 0 & 0 & a_{43} & a_{44} & 0 \\ 0 & a_{52} & 0 & a_{54} & a_{55} \end{bmatrix}$$

`val` stores nonzero elements of A.

`row_ind` stores row number of nonzero elements of A.

`col_ptr` stores location of first nonzero element in each column.

`val:`

a_{11}	a_{31}	a_{22}	a_{32}	a_{52}	a_{13}	a_{33}	a_{43}	a_{24}	a_{44}	a_{54}	a_{15}	a_{25}	a_{55}

`row_ind:`

1	3	2	3	5	1	3	4	2	4	5	1	2	5

`col_ptr:`

1	3	6	9	12	15

The last entry is the number of nonzero elements + 1.

Matrix-vector multiplication CRS format

Multiplication of matrix A and vector x for $y = Ax$

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

Fortran Code

```
do i=1,n
  y(i) = 0.0D0
  do j=row_ptr(i), row_ptr(i+1)-1
    y(i) = y(i) + val(j) * x(col_ind(j))
  end do
end do
```

Matrix-vector multiplication CCS format

Multiplication of matrix A and vector x for $y = Ax$

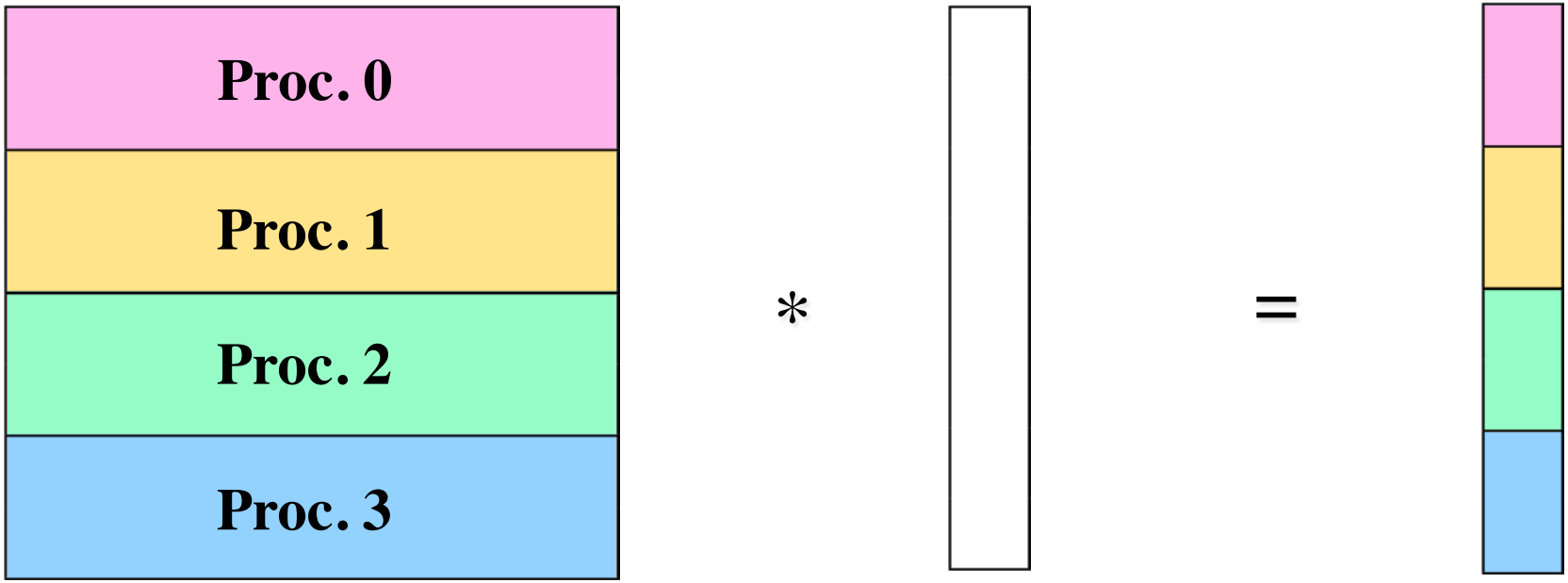
$$y = [a_1, a_2, \ldots, a_n] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \sum_{i=1}^{n} a_i x_i$$

Fortran Code

```
do i=1,n
  y(i) = 0.0D0
end do
do j=1,n
  do i=col_ptr(j), col_ptr(j+1)-1
    y(row_ind(i)) = y(row_ind(i)) + val(i) * x(j)
  end do
end do
```

Parallelization of matrix-vector multiplication

- $y = Ax$ in CRS format



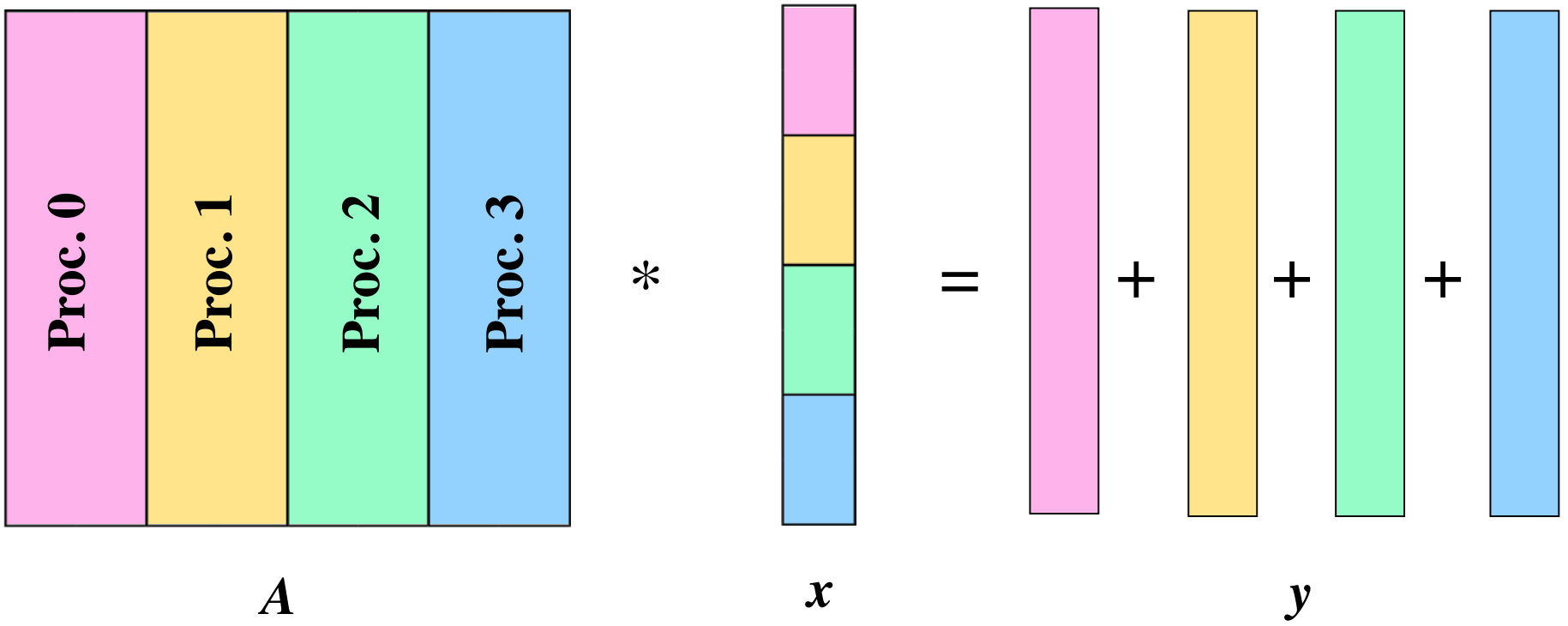
A x y

x is stored in all processes

Gather to Proc. 0 by MPI_Gather

Parallelization of matrix-vector multiplication

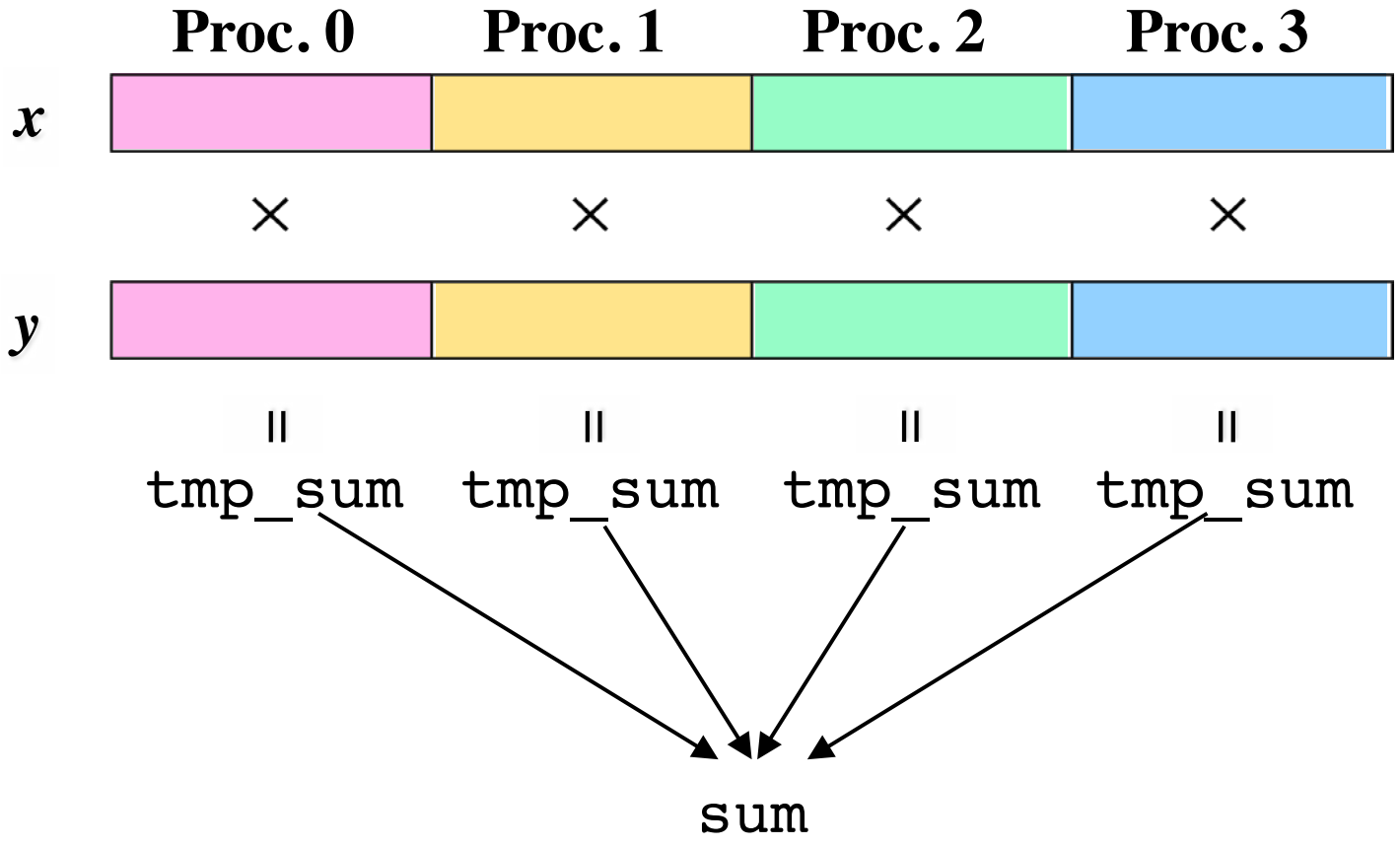
- $y = Ax$ in CCS format



Sum results by MPI_Reduce and send to Proc. 0

Parallelization of inner products

$$(x, y) = \sum_{j=1}^{n} \bar{x}_j y_j$$



Gather to Proc. 0 by MPI_Reduce

Example of MPI code

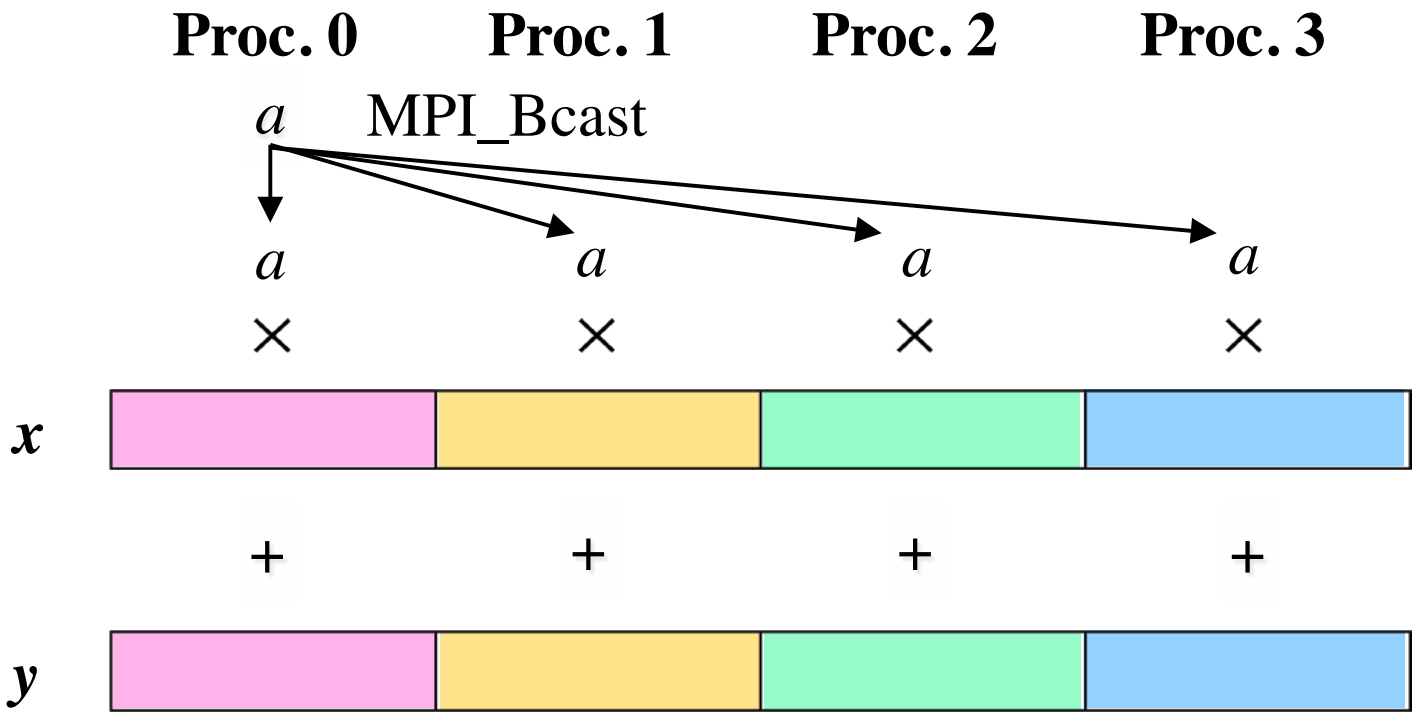
$$(x, y) = \sum_{j=1}^{n} \bar{x}_j y_j$$

```
program main
include 'mpif.h'
...
call mpi_init(ierr)
call mpi_comm_size(mpi_comm_world, npu, ierr)
call mpi_comm_rank(mpi_comm_world, mype, ierr)
...
tmp_sum = (0.0D0, 0.0D0)
do i=istart(mype+1), iend(mype+1)
  tmp_sum = tmp_sum + conj(x(i)) * y(i)
end do
call mpi_reduce(tmp_sum, sum, 1, mpi_double_complex,
mpi_sum, 0, mpi_comm_world, ierr)
...
call mpi_finalize(ierr)
```

Parallelization of constant times a vector plus a vector

$y = y + ax$, a : scalar, x, y : vector.

Send a scalar *a* to all processes by MPI_Bcast



Preconditioning of linear systems

In Krylov subspace methods, the residual sometimes fails to converge

Characteristics of Krylov subspace methods

If the coefficient matrix is close to an identity matrix,
the residual converges in a small number of iterations

Linear system
$Ax = b$

Preconditioning

Preconditioned linear system
$A'x' = b'$

Transformation to a coefficient matrix A' that is close to the identity matrix I !

Preconditioning of linear systems

Coefficient matrix A approximation preconditioning

$$A \approx K_1 K_2 \Longleftrightarrow K_1^{-1} A K_2^{-1} \approx I$$

$$A\boldsymbol{x} = \boldsymbol{b} \Longleftrightarrow (K_1^{-1} A K_2^{-1})(K_2 \boldsymbol{x}) = K_1^{-1} \boldsymbol{b}$$

Inverse matrix A^{-1} approximation preconditioning

$$A \approx M^{-1} \Longleftrightarrow AM \approx I,\ MA \approx I$$

$$A\boldsymbol{x} = \boldsymbol{b} \Longleftrightarrow MA\boldsymbol{x} = M\boldsymbol{b}$$

or $$A\boldsymbol{x} = \boldsymbol{b} \Longleftrightarrow (AM)(M^{-1}\boldsymbol{x}) = \boldsymbol{b}$$

Sparse approximate inverse preconditioning

Generate an approximate inverse matrix *M*

Determine *M* such that $\min_{M} \|I - AM\|_{\mathrm{F}}^2$

$$\|I - AM\|_{\mathrm{F}}^2 = \sum_{j=1}^{n} \|\boldsymbol{e}_j - A\boldsymbol{m}_j\|_2^2$$

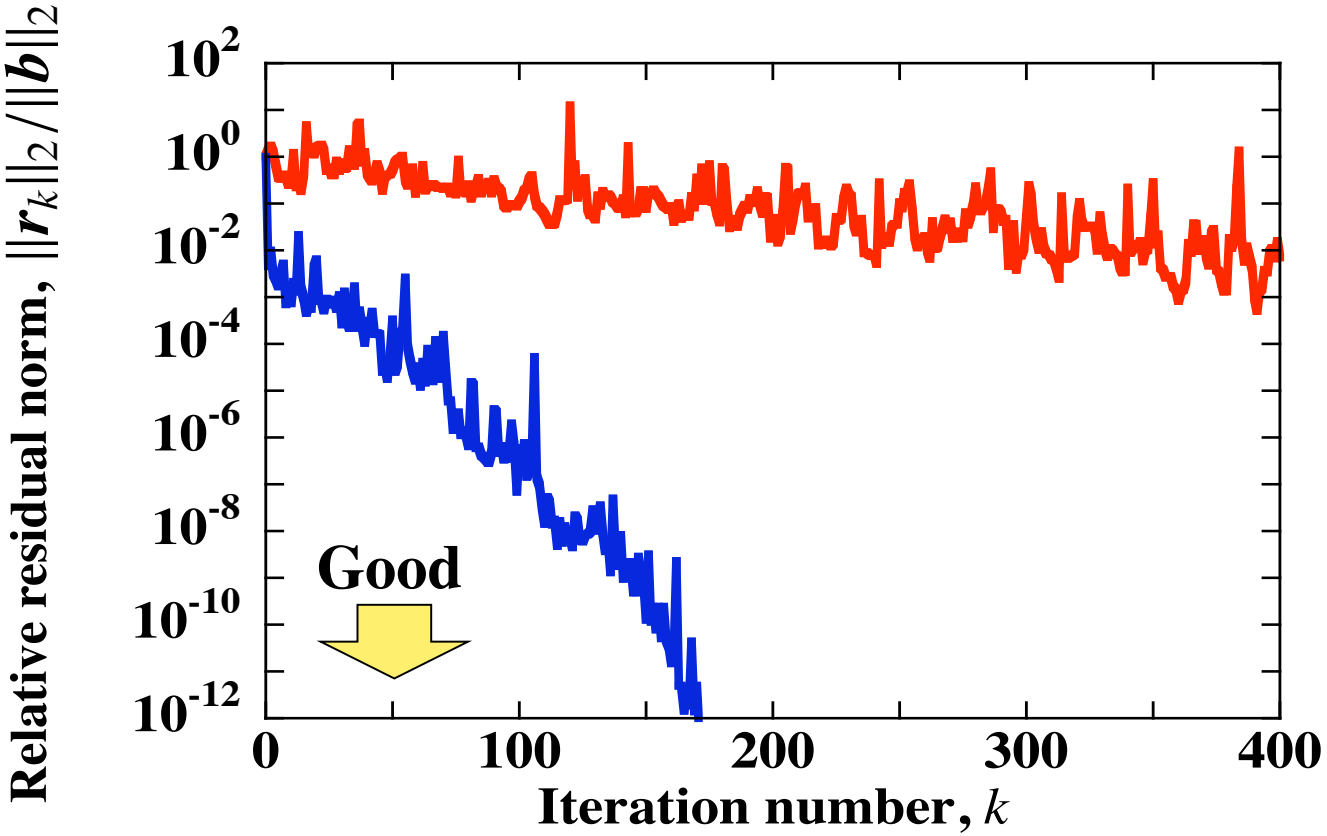
- Nonzero structure of M can be arbitrary selected
- If M is a dense matrix, then we have $M = A^{-1}$

1) n least square problems need to be solved.
2) We can solve these problems in parallel because these problems are independent.

Note : Frobenius norm

$$\|A\|_{\mathrm{F}} = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} |a_{ij}|^2}$$

Convergence property of preconditioned iterative methods



Relative residual histories of iterative methods.

– : BiCG, – : BiCC with sparse approximate inverse preconditioning

Methods for linear systems with multiple right-hand sides

$$AX = B$$

Linear systems with multiple right-hand sides

Linear systems with L right-hand sides

$$AX = B$$

where, A is a matrix of order n and

$$X = \left[\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(L)}\right], \quad B = \left[\boldsymbol{b}^{(1)}, \boldsymbol{b}^{(2)}, \ldots, \boldsymbol{b}^{(L)}\right]$$

Solution by Direct methods

- Complete factorization (e.g., $A = LU$) of the matrix A is required.
- If complete factorization is possible, then we can solve the system by L forward and backward substitutions.
- Large computational complexity and memory usage are required for complete factorization.

Block Krylov subspace methods

Types of Block Krylov subspace methods

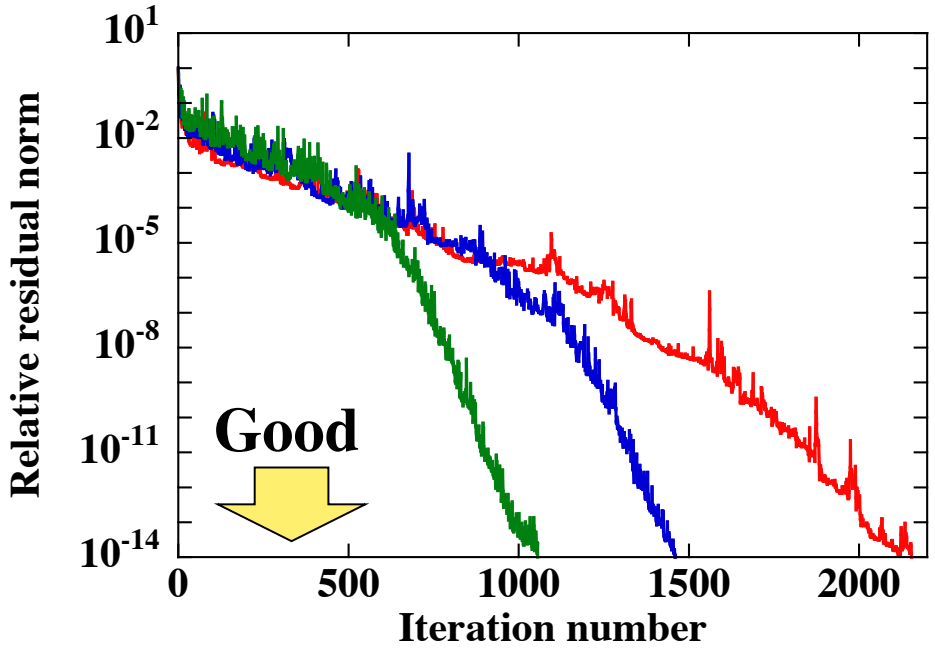
• Block BiCG	O'Leary (1980)
• Block GMRES	Vital (1990)
• Block QMR	Freund (1997)
• Block BiCGSTAB	Guennouni (2003)
• Block BiCGGR	Tadano (2009)

We can efficiently obtain solution vectors by using Block Krylov subspace methods.

Block Krylov subspace methods

What is the meaning of "**good efficiency**" ?

Residual may converge in fewer iterations than Krylov subspace methods for single right-hand side.



Relatrive residual histories of the Block BiCGSTAB methods.

■ : $L = 1$, ■ : $L = 2$, ■ : $L = 4$.

Block BiCGSTAB

$X_0 \in \mathbb{C}^{n\times L}$ is an initial guess,
Compute $R_0 = B - AX_0$,
Set $P_0 = R_0$,
Choose $\tilde{R}_0 \in \mathbb{C}^{n\times L}$,
For $k = 0, 1, \ldots,$ **until** $\|R_k\|_{\mathrm{F}} \le \varepsilon \|B\|_{\mathrm{F}}$ **do:**

$$V_k = AP_k,$$
Solve $(\tilde{R}_0^{\mathrm{H}} V_k)\alpha_k = \tilde{R}_0^{\mathrm{H}} R_k$ for α_k,
$$T_k = R_k - V_k\alpha_k,$$
$$Z_k = AT_k,$$
$$\zeta_k = \mathrm{Tr}\left[Z_k^{\mathrm{H}} T_k\right] / \mathrm{Tr}\left[Z_k^{\mathrm{H}} Z_k\right],$$
$$X_{k+1} = X_k + P_k\alpha_k + \zeta_k T_k,$$
$$R_{k+1} = T_k - \zeta_k Z_k,$$
Solve $(\tilde{R}_0^{\mathrm{H}} V_k)\beta_k = -\tilde{R}_0^{\mathrm{H}} Z_k$ for β_k,
$$P_{k+1} = R_{k+1} + (P_k - \zeta_k V_k)\beta_k,$$
End

Differences from BiCGSTAB

1. Number of matrix-vector mult. increases from 1 to L.
2. α_k and β_k become matrices of order L.
3. Computation of constatnt times vector becomes matrix-matrix mult..
4. To compute ζ_k, matrix trace Tr[ ] becomes necessary.

Trace: Sum of diagonal elements.

Efficient matrix-vector multiplication

- Let the matrix A be stored in CRS format.
- Compute $Y = AX$. Y and X are n-row L-column arrays.

```
do k=1,L
 do i=1,n
  do j=row_ptr(i), row_ptr(i+1)-1
   Y(i,k)=Y(i,k)+A(j)*X(col_ind(j),k)
  end do
 end do
end do
```

[**Problems**]

- Continuous memory access for X is not available.

  (In Fortran, arrays are stored in column major order.)
- Coefficient matrix data must be read L times from memory.

Efficient matrix-vector multiplication

[**Solution strategy**]

- We store *X* and *Y* in transposed form. (*L*-row *n*-column array).

```
do i=1,n
 do j=row_ptr(i), row_ptr(i+1)-1
  do k=1,L
   Y(k,i)=Y(k,i)+A(j)*X(k,col_ind(j))
  end do
 end do
end do
```

- Continuous access (at least *L* times) can be provided for *X*.
- Matrix data are read in just once from memory.
- Continuous access can also be provided for *Y*.

Computation of $n \times L$ matrix by $L \times L$ matrix multiplication

- The vectors are transposed, for efficient matrix-vector multiplication.
- Some device is also necessary to compute $n \times L$ by $L \times L$ matrix mult..

Transposition

$$T_k = R_k - V_k \alpha_k \quad \Rightarrow \quad T_k^{\mathrm{T}} = R_k^{\mathrm{T}} - \alpha_k^{\mathrm{T}} V_k^{\mathrm{T}}$$

```
do j=1,n
 do i=1,L
  T(i,j)=R(i,j)
 end do
end do
do j=1,n
 do i=1,L
  do k=1,L
   T(k,j)=T(k,j)- Alpha(k,i)*V(i,j)
  end do
 end do
end do
```

The matrix `Alpha` is transposed in advance.

Continuous access is enabled by transposing.

Computation of $L \times n$ matrix by $n \times L$ matrix multiplication

- This computation is required to compute α_k and β_k.
- Let us consider the computation of $C_k = \tilde{R}_0^{\mathrm{H}} V_k$.

```
do j=1,n
 do i=1,L
  do k=1,L
   C(k,i)=C(k,i)+R0(k,j)*V(i,j)
  end do
 end do
end do
```

- We can also maintain continuous memory access in computation of C_k.

Parallelization with OpenMP

- Parallelization interface for shared memory.
- Parallelization can be obtained simply by adding a few lines to the exist program.

```
!$OMP PARALLEL
      [  program  ]
!$OMP END PARALLEL
```

Writing as above enables thread start and separate processing in each thread.
(We assume that the following codes are enclosed by `!$OMP PARALLEL` and `!$OMP END PARALLEL` directives.)

Parallelization with OpenMP

1. Parallelization of matrix-vector multiplication

```
!$OMP DO PRIVATE(j,k)
do i=1,n
 do j=row_ptr(i), row_ptr(i+1)-1
  do k=1,L
   Y(k,i)=Y(k,i)+A(j)*X(k,col_ind(j))
  end do
 end do
end do
```

Simply add `!$OMP DO` before the first `do` loop.

Parallelization with OpenMP

2. Parallelization of $n \times L$ matrix by $L \times L$ matrix multiplication

```
!$OMP DO PRIVATE(i)
do j=1,n
 do i=1,L
  T(i,j)=R(i,j)
 end do
end do
!$OMP DO PRIVATE(i,k)
do j=1,n
 do i=1,L
  do k=1,L
   T(k,j)=T(k,j)- Alpha(k,i)*V(i,j)
  end do
 end do
end do
```

Simply add two lines for parallelization.

Parallelization with OpenMP

3. Parallelization of $L \times n$ matrix by $n \times L$ matrix multiplication

Execute the following at the beginning of the code

```
NTH  = OMP_GET_NUM_THREADS()
MYID = OMP_GET_THREAD_NUM()+1
!$OMP SINGLE
allocate( TMP(L,L,NTH) )
!$OMP END SINGLE
```

`NTH` : Number of threads

`MYID` : Thread number

`TMP` : Temporary array for parallel computation

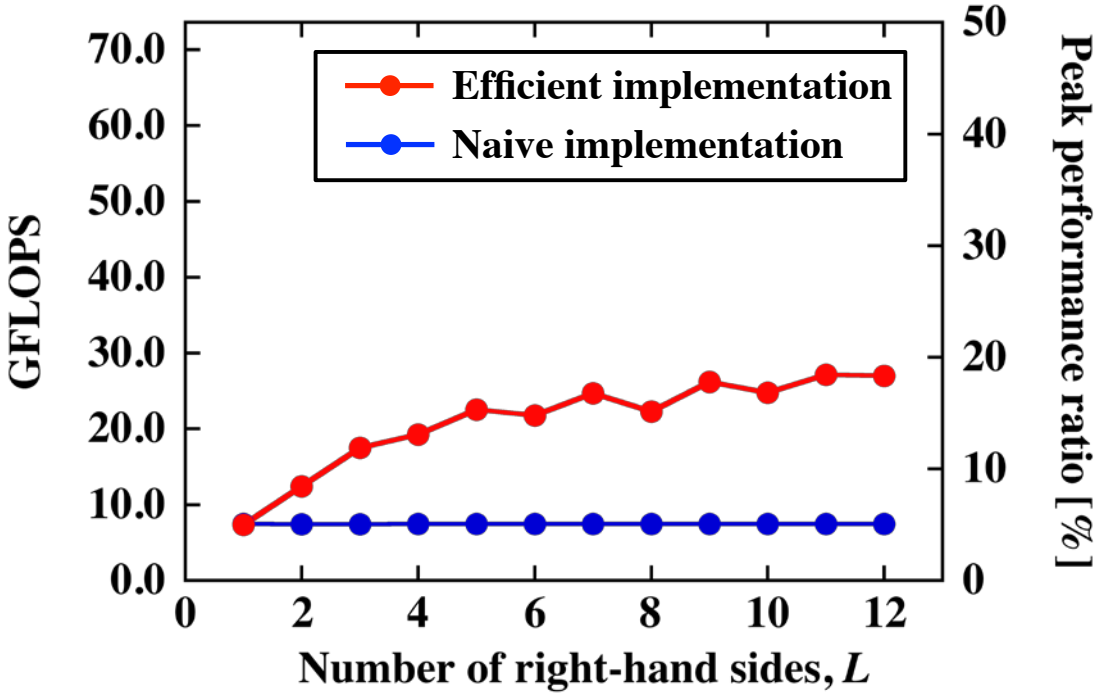
Parallelization with OpenMP

Then, execute the following code to compute $C_k = \tilde{R}_0^{\mathrm{H}} V_k$.

```
!$OMP DO PRIVATE(i,k,MYID)
do j=1,n
 do i=1,L
  do k=1,L
   TMP(k,i,MYID) = TMP(k,i,MYID)+R0(k,j)*V(i,j)
  end do
 end do
end do
!$OMP BARRIER
!$OMP SINGLE
do k=1,NTH
 do j=1,L
  do i=1,L
   C(i,j) = C(i,j) + TMP(i,j,k)
  end do
 end do
end do
!$OMP END SINGLE
```

By using this code, we can execute array reduction processing.

Performance of Matrix-vector multiplication



Relation between GFLOPS count, peak performance ratio and #RHS L.

Matrix size : 1,572,864, #nonzero elements : 80,216,064.

Experimental environment : 1 node of T2K-Tsukuba (Peak : 147.2 GFLOPS)

CPU : AMD Opteron 2.3GHz × 4. Parallelization : 4 OpenMP x 4 MPI.

Parallelization with OpenMP

[**Test linear system**]
Size : 1, 572, 864
#nonzero elements : 80, 216, 064
#right-hand sides : 4
This linear system derived from Quantum Chromodynamics (QCD).

[**Computing environment**]
CPU: Intel Xeon X5550 2.67GHz × 2
Mem: 48GBytes
OS: Cent OS 5.3
Compiler：Intel Fortran ver. 11.1
Option：`-fast -openmp`

#Threads	Time [sec] (#Iterations)	Time / #Iterations	Speedup
1	303.49 (179)	1.6955	1.00
2	183.07 (179)	1.0227	1.66
3	138.07 (179)	0.7713	2.20
4	104.61 (181)	0.5749	2.95
5	80 57 (181)	0.4451	3.81
6	78.56 (181)	0.4340	3.91
7	74.96 (181)	0.4141	4.09
8	68.18 (181)	0.3767	4.50

Summary

In this lecture, we have considered in particular

- Krylov subspace methods for solving linear systems.
- Methods of implementing and parallelizing matrix-vector multiplication for sparse matrices.
- Block Krylov subspace methods, code optimization, and parallelization with OpenMP.