Parallelization of Power Method

名古屋大学情報基盤中心　教授　片桐孝洋

Takahiro Katagiri, Professor,
Information Technology Center, Nagoya University

台大数学科学中心　科学計算冬季学校

Agenda

1. Power Method
2. Execute sample program of power method
3. Explanation of sample program
4. Lecture of parallelization
5. Homework

Power Method

What is power method?

- <Maximum absolute eigenvalue> and <corresponding eigenvector> of standard eigenproblem can be calculated by using power method.
  - Standard Eigenproblem : $Ax = \lambda x$
  - An Eigenvalue : λ An Eigenvector : x

, where a matrix A be a $n \times n$ matrix.

- Let sorted of eigenvalues of A from large part of its absolute, and with no deflation be $\lambda_1, \lambda_2, \cdots, \lambda_n$.
- Let corresponding eigenvectors with normalized and orthogonalized be $x_1, x_2, \cdots, x_n$.
- We can describe arbitrary vector with a linear combination:

$$u = c_1 x_1 + c_2 x_2 + ,\ldots, + c_n x_n$$

What is power method?

- By applying *A* to left hand side, we obtain;

$$Au = A(c_1 x_1 + c_2 x_2 +,\ldots,+c_n x_n)$$

- With respect to formula of standard eigenvalue problem, we obtain:

$$Au = c_1 \lambda_1 x_1 + c_2 \lambda_2 x_2 +,\ldots,+c_n \lambda_n x_n$$

$$= \lambda_1 \left[c_1 x_1 + c_2 \frac{\lambda_2}{\lambda_1} x_2 +,\ldots,+c_n \frac{\lambda_n}{\lambda_1} x_n \right]$$

What is power method?

- By applying *Au* with *n*-times, we obtain:

$$A^k u = \lambda_1^{\,k} \left[c_1 x_1 + c_2 \left[\frac{\lambda_2}{\lambda_1} \right]^k x_2 +, \ldots, + c_n \left[\frac{\lambda_n}{\lambda_1} \right]^k x_n \right]$$

- This implies that coefficients of the vectors are reducing except for x_1 when *k* is increasing.

→It converges with a maximum eigenvalue and a corresponding eigenvector.

What is power method?

- We denote (x, y) for dot products. Consider the following formula:

$$\frac{(A^{k+1}u,\ A^{k+1}u)}{(A^{k+1}u,\ A^{k}u)} = \frac{\sum_{i=1}^{n}\sum_{j=1}^{n} c_i c_j \lambda_i^{k+1} \lambda_j^{k+1} (x_i, x_j)}{\sum_{i=1}^{n}\sum_{j=1}^{n} c_i c_j \lambda_i^{k+1} \lambda_j^{k} (x_i, x_j)}$$

$$= \frac{\lambda_1^{2k+2}\left[c_1^{2} |x_1|^2 + \sum_{i=2}^{n} c_i^{2} \left[\frac{\lambda_i}{\lambda_1}\right]^{2k+2} |x_i|^2 \right]}{\lambda_1^{2k+1}\left[c_1^{2} |x_1|^2 + \sum_{i=2}^{n} c_i^{2} \left[\frac{\lambda_i}{\lambda_1}\right]^{2k+1} |x_i|^2 \right]} \approx \underset{(k \to \infty)}{\lambda_1}$$

Algorithm of Power Method

- Do the following until converge:

1. Make an initial guess x and normalize it;
2. $\lambda\_0 = 0.0; \quad i = 1;$
3. Compute a matrix-vector multiplication: $y = A\ x$;
4. Compute an approximate eigenvalue
   $\lambda\_i = (y,\ y)\ /\ (y,\ x)$;
5. If $|\lambda\_i - \lambda\_\{ i -1 \}|$ is small enough:
   - It converges, and exit;
6. Otherwise :
   - Normalize x and $x = y$;
   - $i = i + 1$; go to 3;

Execute sample program (Power Method)

Note: Sample program of power method

- File name of C/Fortran codes:

  PowM-fx.tar
- Change queue name from lecture to lecture7 in job script file pown.bash .
- Submit the job with "pjsub".
  - lecture : Queue in out of time for the lesson.
  - lecture7: Queue in time for the lesson.

Execute sample program of power method

- Type the followings in command line.

```
$ cp /home/z30082/PowM-fx.tar ./
$ tar xvf PowM-fx.tar
$ cd PowM
```

- Choose the follows:

$ cd C : For C language.
$ cd F : For Fortran language.

- Type the follows:

```
$ make
$ pjsub powm.bash
```

- After finishing execution, type the follow:

```
$ cat powm.bash.oXXXXXX
```

Output for sample program of power method (C Language)

- The follows can be seen if execution is successfully ended.

N = 4000

Power Method time = 0.472348 [sec.]

Eigenvalue = 2.000342e+03

Iteration Number: 7

Residual 2-Norm ||A x - lambda x||_2 = 7.656578e-09

Output for sample program of power method (Fortran Language)

- The follows can be seen if execution is successfully ended.

N = 4000

Power Method time[sec.] = 0.32137653301469982

Eigenvalue = 2000.30672121217447

Iteration Number: 6

Residual 2-Norm ||A x - lambda x||_2 =
4.681124813641846E-07

Explanation of sample program

- You can change size of matrix to modify the following number of:

  #define N 4000

- Specification of PowM function
  - Maximum eigenvalue with double precision is returned.
  - Eigenvector corresponding to maximum eigenvalue is stored in array of x with double precision
  - Iteration count when it converges is stored in argument n_iter.
    - If it returns "-1", then this means that no convergence is happen until maximum iteration MAX_ITER.

Note: sample program of Fortran

- Declaration of size of matrix NN and MAX_ITER is in:

  pown.inc

- The size of matrix is defined by variable NN:

```
integer  NN
parameter (NN=4000)
```

Overview of sample program (in function of PowM)

```
/* Normizeation of x */
  d_tmp1 = 0.0;
  for(i=0; i<n; i++) {
    d_tmp1 += x[i] * x[i];
  }
  d_tmp1 = 1.0 / sqrt(d_tmp1);
  for(i=0; i<n; i++) {
    x[i] = x[i] * d_tmp1;
  }

  /* Main iteration loop ---------------------- */
  for(i_loop=1; i_loop<MAX_ITER; i_loop++)  {

     /* Matrix Vector Product */
     MyMatVec(y, A, x, n);

     /* innner products */
       d_tmp1 = 0.0;
       d_tmp2 = 0.0;
       for (i=0; i<n; i++) {
       d_tmp1 += y[i] * y[i];
       d_tmp2 += y[i] * x[i];
     }

     /* current approximately eigenvalue */
     dlambda = d_tmp1 / d_tmp2;
```

Normalization of vector x

Matrix-vector Multiplications

Dot product with vectors x and y.

```
       /* Convergence test*/
       if (fabs(d_before-dlambda) < EPS ) {
         *n_iter = i_loop;
          return dlambda;
       }

     /* keep current value */
     d_before = dlambda;

  /* Normalization and set new x */
     d_tmp1 = 1.0 / sqrt(d_tmp1);
     for(i=0; i<n; i++)
       x[i] = y[i] * d_tmp1;

  }  /* end of i_loop ------------------------
```

Normalization and setting of new vector x.

Homework 3

- Parallelize function (procedure) of PowM.
  - For debugging, set #define N 192 .
  - Use parallel matrix-vector code in previous lesson.
- In the sample program, 2-norm of residual vector $Ax\text{-}\lambda x$ is calculated. Use the calculated value for debugging.
  - If you found big value of this, it means a bug in program.
  - The parallelization of computation of 2-norm may be needed if you choose "perfect" distribution of vector x. This explains later.
- By parallelization, number of iteration and execution time may change.

Hints for parallelization

- As same as previous lesson, one of easy ways to parallelize the code is allocating redundant matrix A with NxN, vectors x and y with N, for each processes.
- Use following distributions. This is as same as previous lesson for matrix-vector multiplication.
  - Matrix *A*:
    Row-wise block distribution with one dimensional.
  - Vector *x*:
    Allocate redundant vector with N dimension for all processes.
  - Vector *y*:
    Block distribution.

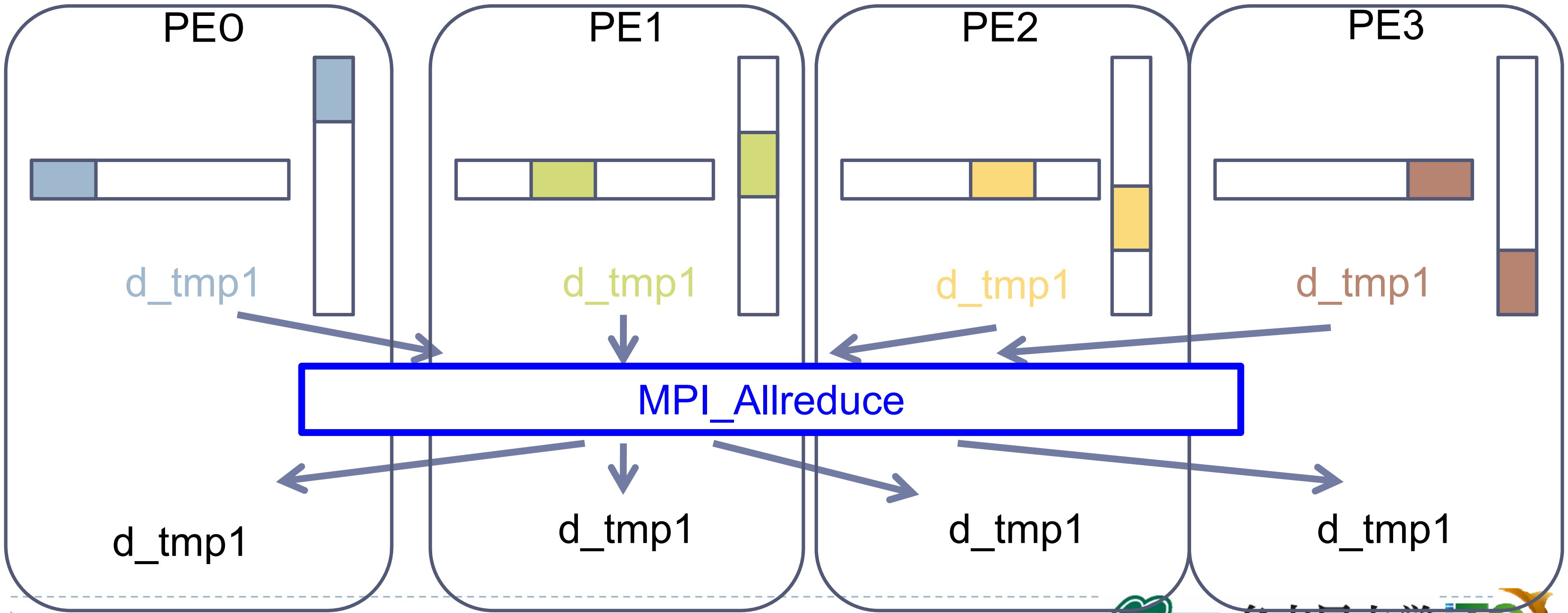
Hints of parallelization (Strategy)

- There are two ways to parallelize the code:
  - Way 1 : Only parallelization for part of "matrix-vector multiplication"
  - Way 2 : Parallelization of all routines.
- Easy way is 1 (But parallel efficiency is limited). The follows is procedure.

1. Use developed "parallel matrix-vector multiplication".
2. Since y of y = Ax is retuned by distributed manner, it cannot continue the following computations. Hence to match sequential result, we need a communication such that:
   - By using an MPI function just after part of calling MyMatVec() in PowM function to gather all distributed elements of y.
   - There are many ways to implement it. The easiest way is implementation with MPI_Allreduce().
3. To use MPI_Allreduce(), initialization of array, such as fill on 0, is needed. This will be explained later.

Hints of parallelization (Way 2. Parallelization of all routines)

- Parallelize processes in function PowM with the following:
  1. For the part of normalization of vector x
     - After finishing local computations of dot product with block distribution, call function of MPI_Allreduce, which as shown as the follow.
     - Gather all elements of vector for partially calculated in each PE with MPI_Allreduce function. This will be explained later.



Hints of parallelization
(Way 2. Parallelization of all routines)

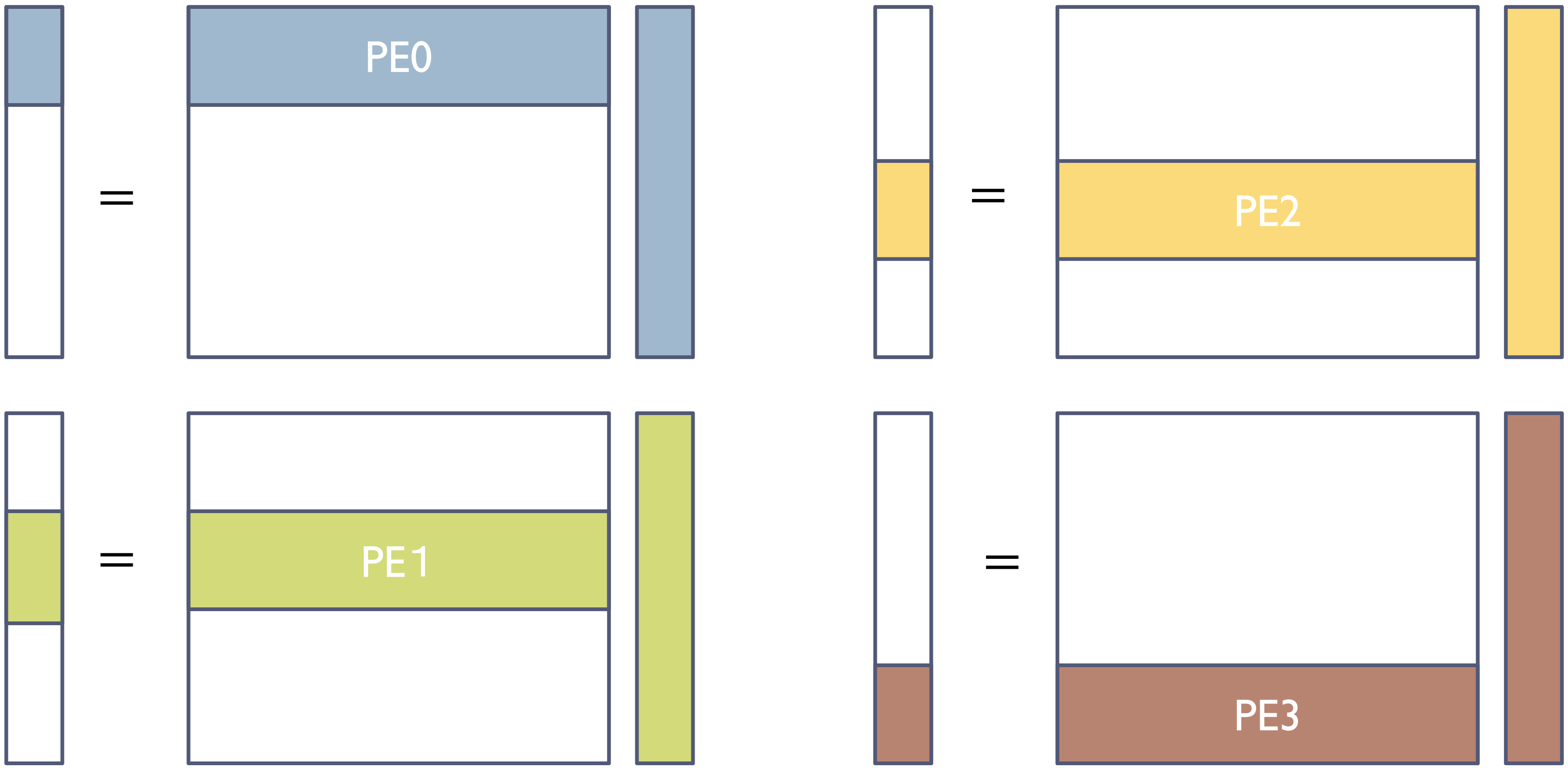
- The follows is an implementation:

```
/* Normizeation of x */
…
   d_tmp1_t = 0.0;
   for(i=myid*ib; i<i_end; i++) {
      d_tmp1_t += x[i] * x[i];
   }
   MPI_Allreduce(&d_tmp1_t,  &d_tmp1, 1, MPI_DOUBLE,
       MPI_SUM, MPI_COMM_WORLD);

   d_tmp1 = 1.0 / sqrt(d_tmp1);
   for(i=myid*ib; i<i_end; i++) {
      x_t[i] = x[i] * d_tmp1;
   }
    /* x_t[ ] is set to 0 in initial state. */
  MPI_Allreduce(x_t, x, n, MPI_DOUBLE, MPI_SUM,
      MPI_COMM_WORLD);
 ….
```

Hints of parallelization
(Both way 1 and way 2)

2. Part of matrix-vector multiplication. (In MyMatVec Function)
   - Use parallel code in previous lesson.



Hints of parallelization
(Way 2. Parallelization of all routines)

3. Dot product of vectors *x* and *y*.
   - Compute with respect to block distribution.
   - To obtain correct answer, do not forget to use MPI_Allreduce function.

Hints of parallelization (Way 2. Parallelization of all routines)

4. Part of normalization and set new x:
   - x : Allocated redundant vector with N-dimensional; y: Block distribution;
   - Computations of normalization are performed with local data, and set result to x.
   - Elements of x are distributed. Hence calculated x is stored in block distribution manner.
   - All elements of x need since next computation of matrix-vector multiplication is needed with the whole elements of x
   - To gather distributed data, we use MPI_Allreduce.
     - To use MPI_Allreduce, we allocate a buffer array x_t with zero cleared for distributed part. This can be used as:
       MPI_Allreduce(x_t, x, n, MPI_DOUBLE, MPI_SUM,
       MPI_COMM_WORLD);

Confirmation of MPI_Allreduce function (C Language)

- MPI_Allreduce

```
(x_t, x,  n,  MPI_DOUBLE,  MPI_SUM, MPI_COMM_WORLD);
```

Input vector. Each PE has different elements.

Output vector. Each PE has same element.

Length of vectors.

Type of elements of vector.

Specifying operations. MPI_SUM: summation of elements of vectors in each PE.

Communicator.

Confirmation of MPI_Allreduce function (Fortran Language)

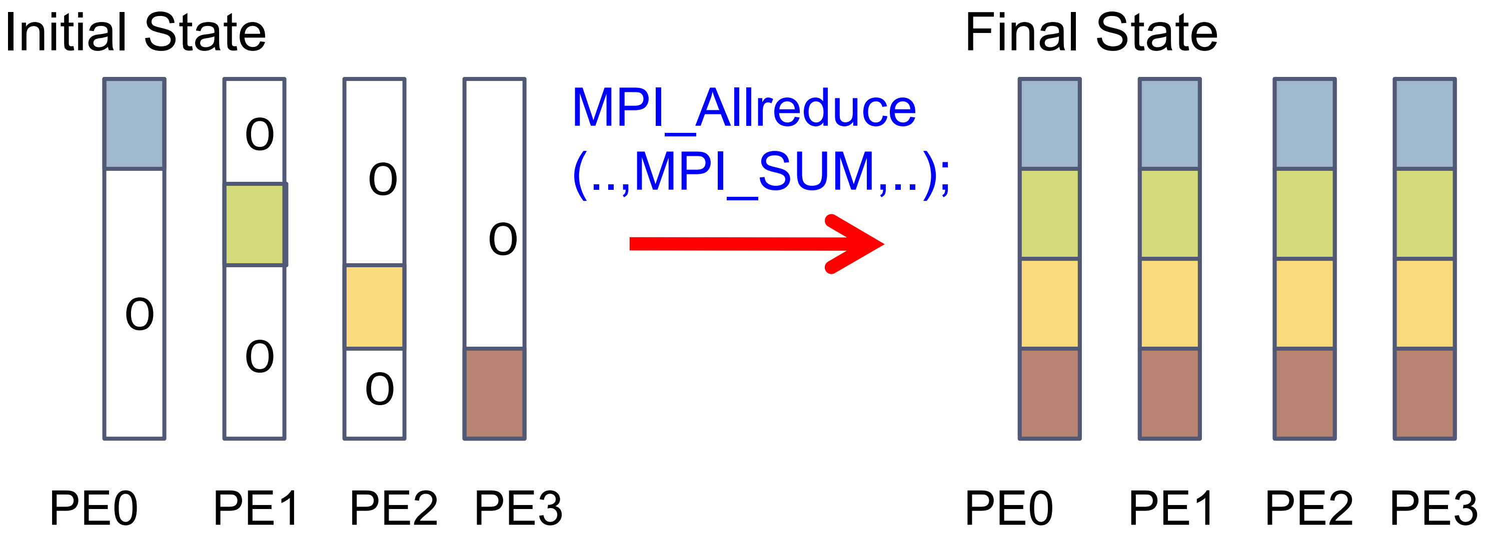
- ## MPI_ALLREDUCE

(x_t, x, n, MPI_DOUBLE_PRECISON, MPI_SUM, MPI_COMM_WORLD, ierr)

- Input vector. Each PE has different elements.
- Output vector. Each PE has same element.
- Length of vectors.
- Type of elements of vector
- Specifying operations. MPI_SUM: summation of elements of vectors in each PE.
- Communicator.

A technique of MPI (Gather vectors with MPI_Allreduce)

- Gather distributed data with MPI_Allreduce function, then it owns redundant elements between all PEs.
  - Write **MPI_SUM** in **iop**
  - Initialize elements of own part with 0.
  - Consider the following process.



It can also be implemented with **MPI_gather**.

Homework 3

- (Standard level) For the first step, implement
  Way 1： Only parallelization for part of "matrix-vector multiplication"

- (High level) After finishing the way 1, implement
  Way 2： Parallelization of all routines.

Lessons

1. Homework 3
2. Parallelize the sample program and evaluate it. Only allocations of required size of arrays of matrix A and vectors x and y for each PE are allowed. Compare performance to 1.

Lessons (Cont'd)

3. Evaluate number of iterations when options of compiler are changed. Compute execution time per iteration to evaluate it.
4. Improve performance of the sample programs with non-blocking communications. Evaluate program with several sizes of matrices.
5. Parallelize the program with hybrid MPI/OpenMP execution. Evaluate the program with several combinations of execution, such as P8T16, P16T8, and so on.
   Find condition that pure MPI execution is the fastest to other hybrid MPI/OpenMP execution.