内容に関するご質問は
katagiri ＠ cc.u-tokyo.ac.jp
まで、お願いします。

第４３回　お試しアカウント付き
並列プログラミング講習会
「MPI基礎：並列プログラミング初級入門」

東京大学情報基盤センター

講習会概略

- 開催日：
  2015年 3月　9日（月）　10：30 - 17：00
  2015年 3月10日（火）　10：00 - 17：00
- 場所：東京大学情報基盤センター　4階　413遠隔会義室
- 講習会プログラム：
- 3月9日（月）
  - 10：00 - 10：30　受付
  - 10：30 - 12：30　ノートパソコンの設定、テストプログラムの実行など（演習）
    （講師：片桐）
  - 14：00 - 15：45　並列プログラミングの基本（座学）（講師：片桐）
  - 16：00 - 17：00　MPIプログラム実習 I（演習）（講師：片桐）
- 3月10日（火）
  - 10：00 - 12：30　プログラミングの基礎（分割コンパイル）（演習）（講師：大島）
  - 14：00 - 15：30　MPIプログラミング実習 II（演習）（講師：片桐）
  - 15：45 - 17：00　MPIプログラミング実習 III（演習）（講師：片桐）

有償トライアルユース制度の開始について

- 2012年4月より、安価に当センターのFX10スーパーコンピュータシステムが使える「無償トライアルユース、および、有償トライアルユース」制度が開始予定です。
  - アカデミック利用
    - パーソナルコース、グループコースの双方（1ヶ月～3ヶ月）
  - 企業利用
    - パーソナルコース（1ヶ月～3ヶ月）（最大24ノード、最大96ノード）
      本講習会の受講が必須、審査無
    - グループコース
      - 無償トライアルユース：（1ヶ月～3ヶ月）：無料（12ノード、1口）
      - 有償トライアルユース：（1ヶ月～最大通算9ヶ月）、有償（12ノード、1口）
      - スーパーコンピュータ利用資格者審査委員会の審査が必要（年2回実施）
    - 双方のコースともに、簡易な利用報告書の提出が必要
- 料金体系や利用条件の詳細は、以下のHPをご覧ください
  http://www.cc.u-tokyo.ac.jp/service/trial/fx10.html

スパコンへのログイン・テストプログラム起動

東京大学情報基盤センター　准教授　片桐孝洋

講義の流れ

1. スパコン利用の仕方
   - 単純な並列プログラムの実行
2. 総和演算

テストプログラム起動

ＵＮＩＸ備忘録

- Emacsの起動： emacs <編集ファイル名>
  - ^x ^s （^はcontrol）：テキストの保存
  - ^x ^c ： 終了
    （ ^z で終了すると、スパコンの負荷が上がる。絶対にしないこと。）
  - ^g ： 訳がわからなくなったとき。
  - ^k ：カーソルより行末まで消す。
    消した行は、一時的に記憶される。
  - ^y ：^kで消した行を、現在のカーソルの場所にコピーする。
  - ^s 文字列： 文字列の箇所まで移動する。
  - ^M goto-line： 指定した行まで移動する。
    (^MはESCキーを押す)

ＵＮＩＸ備忘録

- rm ファイル名： ファイル名のファイルを消す。
  - rm *~： test.c~ などの、~がついたバックアップファイルを消す。
- ls： 現在いるフォルダの中身を見る。
- cd フォルダ名： フォルダに移動する。
  - cd .. : 一つ上のフォルダに移動。
  - cd ~ : ホームディレクトリに行く。訳がわからなくなったとき。
- cat ファイル名： ファイル名の中身を見る
- make : 実行ファイルを作る
  （Makefile があるところでしか実行できない）
  - make clean : 実行ファイルを消す。
    （clean がMakefileで定義されていないと実行できない）

サンプルプログラムの実行

初めての並列プログラムの実行

サンプルプログラム名

- C言語版・Fortran90版共通ファイル：

  Samples-fx.tar

- tarで展開後、C言語とFortran90言語のディレクトリが作られる
  - C/ ：C言語用
  - F/ ：Fortran90言語用
- 上記のファイルが置いてある場所

  /home/z30082

並列版Helloプログラムをコンパイルしよう (1/2)

1. /home/z30082 にある Samples-fx.tar を
   自分のディレクトリにコピーする
   $ cp /home/z30082/Samples-fx.tar ./
2. Samples-fx.tar を展開する
   $ tar xvf Samples-fx.tar
3. Samples フォルダに入る
   $ cd Samples
4. C言語 : $ cd C
   Fortran90言語 : $ cd F
5. Hello フォルダに入る
   $ cd Hello

並列版Helloプログラムをコンパイルしよう (2/2)

6. ピュアMPI用のMakefileをコピーする
   ```
   $ cp Makefile_pure  Makefile
   ```
7. make する
   ```
   $  make
   ```
8. 実行ファイル(hello)ができていることを確認する
   ```
   $  ls
   ```

FX10スーパコンピュータシステムでのジョブ実行形態

- 以下の2通りがあります
- インタラクティブジョブ実行
  - PCでの実行のように、コマンドを入力して実行する方法
  - スパコン環境では、あまり一般的でない
  - デバック用、大規模実行はできない
  - FX10では、以下に限定
    - 1ノード（16コア）（2時間まで）
    - 8ノード（128コア）（10分まで）
- バッチジョブ実行
  - バッチジョブシステムに処理を依頼して実行する方法
  - スパコン環境で一般的
  - 大規模実行用
  - FX10では、最大1440ノード（23,040コア）(6時間)

インタラクティブ実行のやり方

- コマンドラインで以下を入力
  - 1ノード実行用

    $ pjsub --interact

  - 8ノード実行用

    $ pjsub --interact -L “node=8”

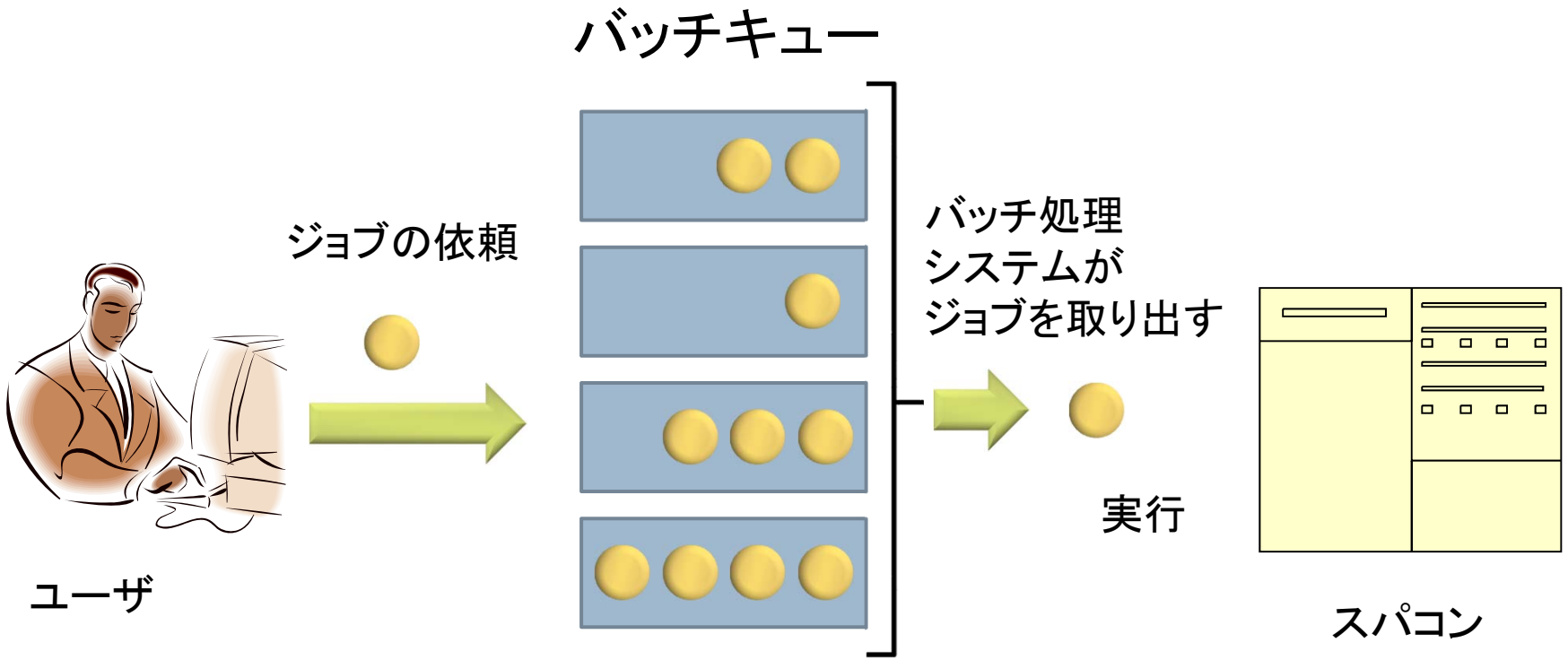
※インタラクティブ用のノード総数は50ノードです。
もしユーザにより50ノードすべて使われている場合、
資源が空くまで、ログインできません。

コンパイラの種類とインタラクティブ実行およびバッチ実行

- インタラクティブ実行、およびバッチ実行で、利用するコンパイラ（C言語、C＋＋言語、Fortran90言語）の種類が違います
- インタラクティブ実行では
  - オウンコンパイラ（そのノードで実行する実行ファイルを生成するコンパイラ）を使います
- バッチ実行では
  - クロスコンパイラ（そのノードでは実行できないが、バッチ実行する時のノードで実行できる実行ファイルを生成するコンパイラ）を使います
- それぞれの形式
  - オウンコンパイラ：＜コンパイラの種類名＞
  - クロスコンパイラ：＜コンパイラの種類名＞px
  - 例）富士通Fortran90コンパイラ
    - オウンコンパイラ：frt
    - クロスコンパイラ：frtpx

バッチ処理とは

- スパコン環境では、通常は、インタラクティブ実行（コマンドラインで実行すること）はできません。
- ジョブはバッチ処理で実行します。



バッチキューの設定のしかた

- バッチ処理は、富士通社のバッチシステムで管理されています。
- 以下、主要コマンドを説明します。
  - ジョブの投入：
    pjsub <ジョブスクリプトファイル名> -g <プロジェクトコード>
  - 自分が投入したジョブの状況確認： pjstat
  - 投入ジョブの削除： pjdel <ジョブID>
  - バッチキューの状態を見る： pjstat --rsc
  - バッチキューの詳細構成を見る： pjstat –rsc -x
  - 投げられているジョブ数を見る： pjstat -b
  - 過去の投入履歴を見る： pjstat --history
  - 同時に投入できる数／実行できる数を見る：pjstat --limit

本お試し講習会でのキュー名

- 本演習中のキュー名：
  - tutorial
  - 最大15分まで
  - 最大ノード数は12ノード(192コア) まで
- 本演習時間以外（24時間）のキュー名：
  - lecture
- 利用条件は演習中のキュー名と同様

pjstat --rsc の実行画面例

```
$ pjstat --rsc
RSCGRP                STATUS              NODE:COORD
debug                [ENABLE,START]     480:10x3x16
short                [ENABLE,START]     480:10x3x16
regular
  |---- small        [ENABLE,START]     3840:20x12x16
  |---- medium       [ENABLE,START]     3840:20x12x16
  |---- large        [ENABLE,START]     3840:20x12x16
  `---- x-large      [ENABLE,START]     3840:20x12x16
interactive
  |---- interactive_n1  [ENABLE,START]      50
  `---- interactive_n8  [ENABLE,START]      50
```

使える
キュー名
(リソース
グループ)

現在
使えるか

ノードの
物理構成情報

pjstat --rsc -x の実行画面例

```
$ pjstat --rsc -x
RSCGRP    STATUS    MIN_NODE  MAX_NODE  ELAPSE   MEM(GB)  PROJECT
debug        [ENABLE,START]      1         240      00:30:00     28     gcXX, gcYY
short        [ENABLE,START]      1          72      06:00:00     28     gcXX, gcYY
regular
  |---- small   [ENABLE,START]     12        216      48:00:00     28     gcXX, gcYY
  |---- medium   [ENABLE,START]  217      372      48:00:00     28     gcXX, gcYY
  |---- large      [ENABLE,START]   373      480      48:00:00     28     gcXX, gcYY
  `---- x-large  [ENABLE,START]    481     1440     24:00:00     28     gcXX, gcYY
interactive
  |---- interactive_n1   [ENABLE,START]   1   1      02:00:00     28     gcXX, gcYY
  `---- interactive_n8   [ENABLE,START]   2    8     00:10:00     28     gcXX, gcYY
```

使える
キュー名
(リソース
グループ)

現在
使えるか

ノードの
実行情報

課金情報(財布)
実習では1つのみ

pjstat -b の実行画面例

```
$ pjstat –b
RSCGRP        STATUS      TOTAL    RUNNING   QUEUED   HOLD   OTHER NODE:COORD
debug     [ENABLE,START]      3          2         0      0       1       480:10x3x16
short     [ENABLE,START]      1          1         0      0       0       480:10x3x16
regular
  |---- small  [ENABLE,START]   165        81        84      0       0       3840:20x12x16
  |---- medium  [ENABLE,START]   25         4        20      0       1       3840:20x12x16
  |---- large    [ENABLE,START]    0         0         0      0       0       3840:20x12x16
  `---- x-large   [ENABLE,START]    4         0         4      0       0       3840:20x12x16
interactive
  |---- interactive_n1   [ENABLE,START]   2     2          0      0       0       50
  `---- interactive_n8   [ENABLE,START]   1     1          0      0       0       50
```

使えるキュー名(リソースグループ)

現在使えるか

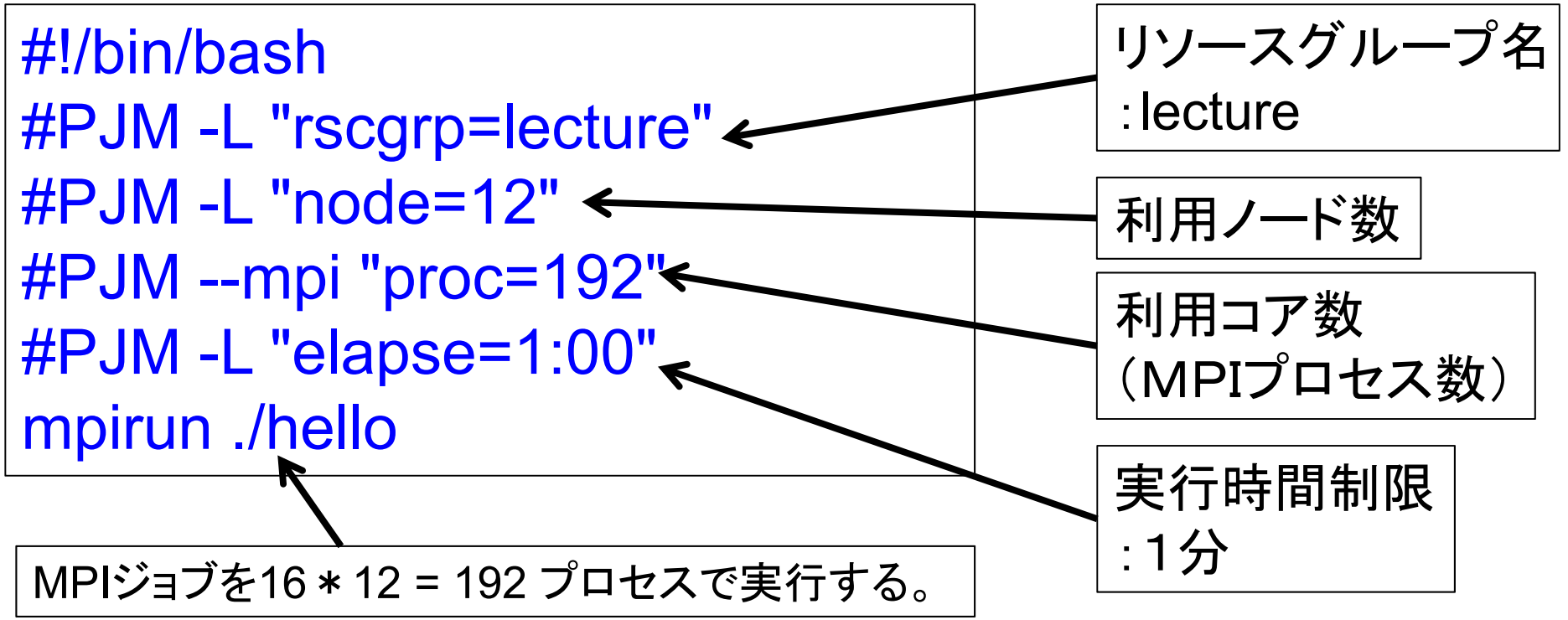
ジョブの総数

実行しているジョブの数

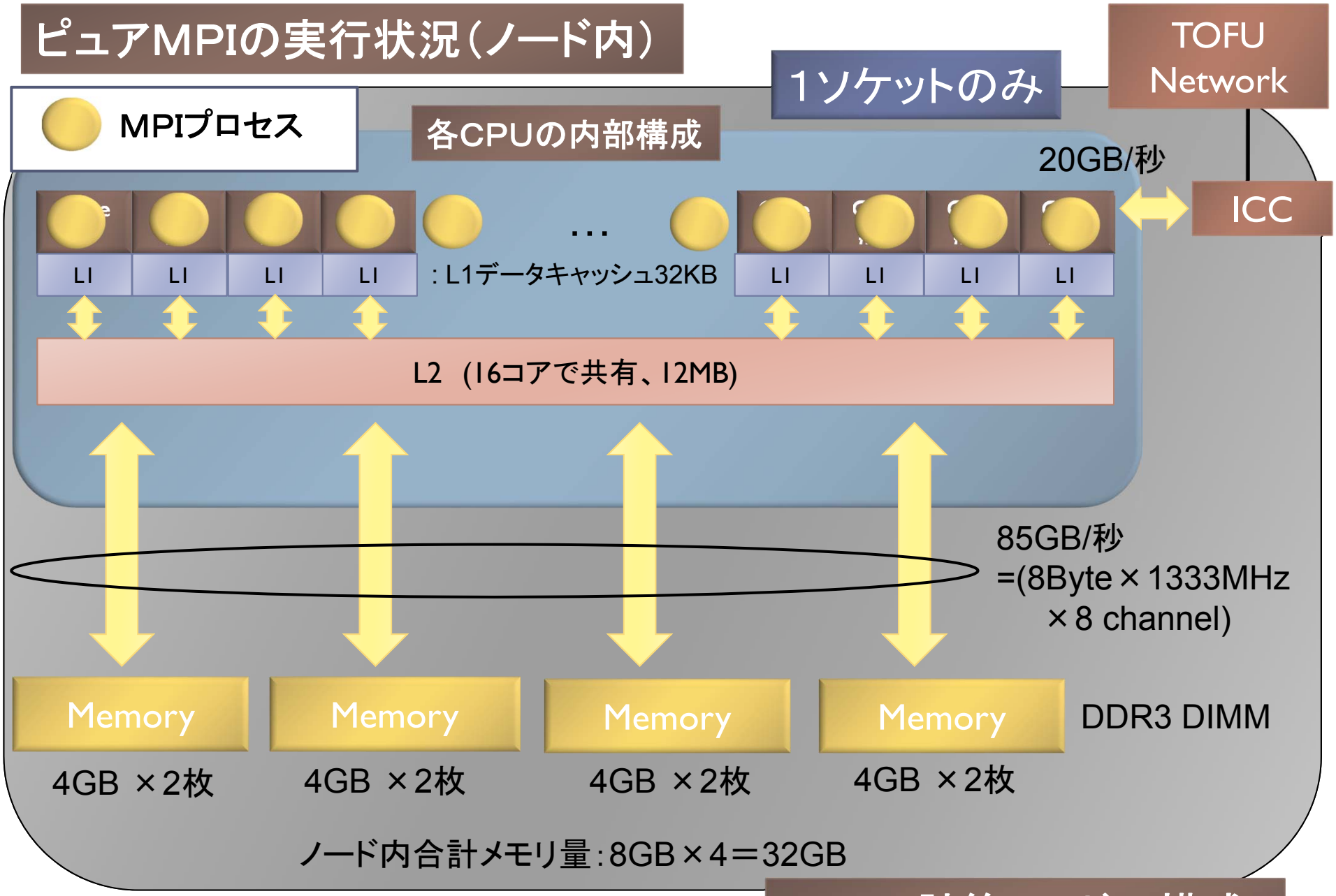
待たされているジョブの数

ノードの物理構成情報

JOBスクリプトサンプルの説明（ピュアＭＰＩ）（hello-pure.bash, C言語、Fortran言語共通）



ピュアMPIの実行状況（ノード内）



FX10計算ノードの構成

並列版Helloプログラムを実行しよう（ピュアMPI）

- このサンプルのJOBスクリプトは
  hello-pure.bash
  です。
- 配布のサンプルでは、キュー名が”lecture”になっています
- $ emacs hello-pure.bash
  で、“lecture” → “tutorial”
  に変更してください

並列版Helloプログラムを実行しよう（ピュアMPI）

1. Helloフォルダ中で以下を実行する
   $ pjsub hello-pure.bash
2. 自分の導入されたジョブを確認する
   $ pjstat
3. 実行が終了すると、以下のファイルが生成される
   hello-pure.bash.eXXXXXX
   hello-pure.bash.oXXXXXX （XXXXXXは数字）
4. 上記の標準出力ファイルの中身を見てみる
   $ cat hello-pure.bash.oXXXXXX
5. “Hello parallel world!”が、
   16プロセス*12ノード=192表示されていたら成功。

バッチジョブ実行による標準出力、標準エラー出力

- バッチジョブの実行が終了すると、標準出力ファイルと標準エラー出力ファイルが、ジョブ投入時のディレクトリに作成されます。
- 標準出力ファイルにはジョブ実行中の標準出力、標準エラー出力ファイルにはジョブ実行中のエラーメッセージが出力されます。

ジョブ名.oXXXXX --- 標準出力ファイル
ジョブ名.eXXXXX --- 標準エラー出力ファイル
(XXXXX はジョブ投入時に表示されるジョブのジョブID)

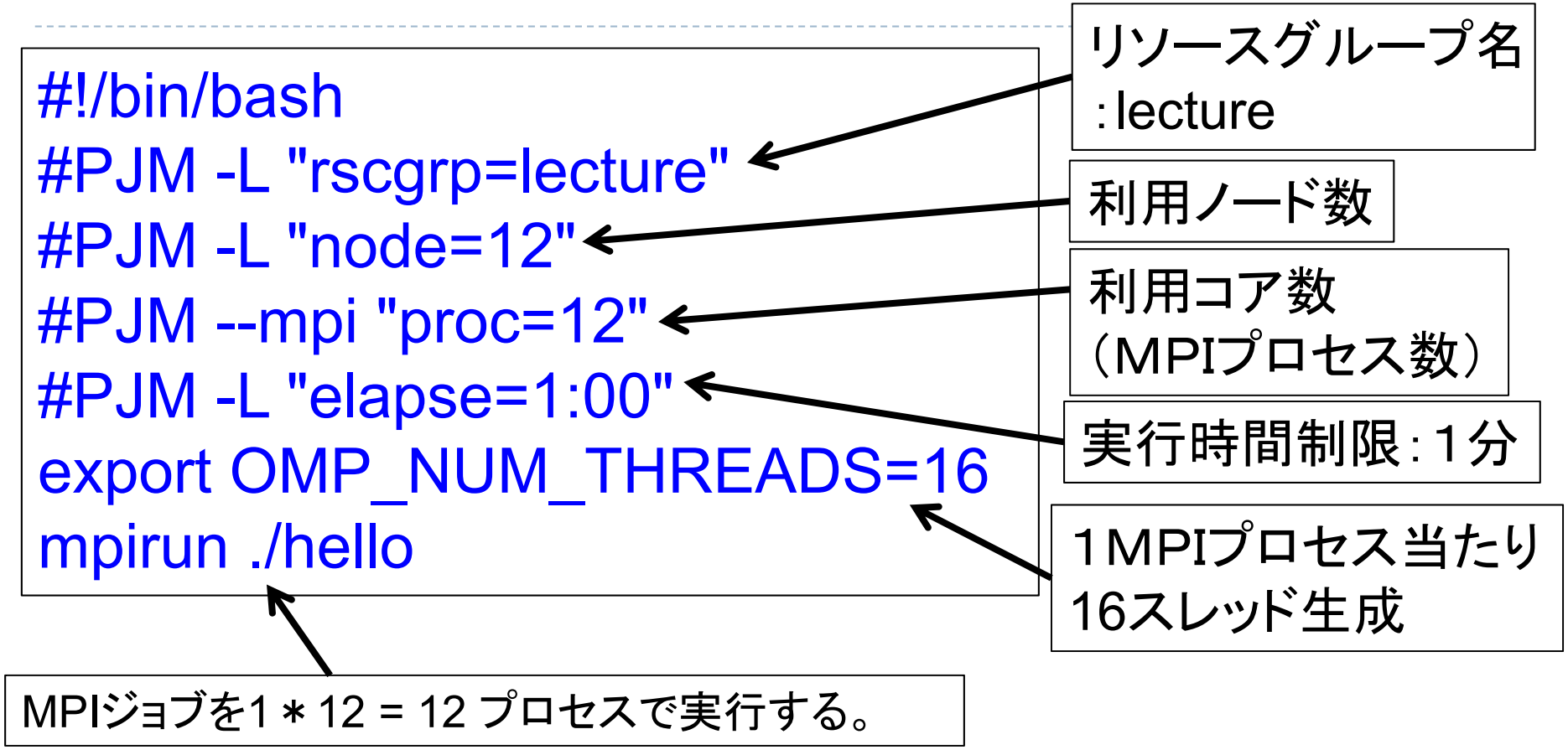
並列版Helloプログラムをコンパイルしよう

1. ハイブリッドMPI用の Makefile をコピーする。
   $ cp Makefile_hy16 Makefile
2. make する。
   $ make clean
   $ make
3. 実行ファイル（hello）ができていることを確認する。
   $ ls
4. JOBスクリプト中（hello-hy16.bash）のキュー名を変更する。“lecture”　→“tutorial”に変更する。
   $ emacs hello-hy16.bash

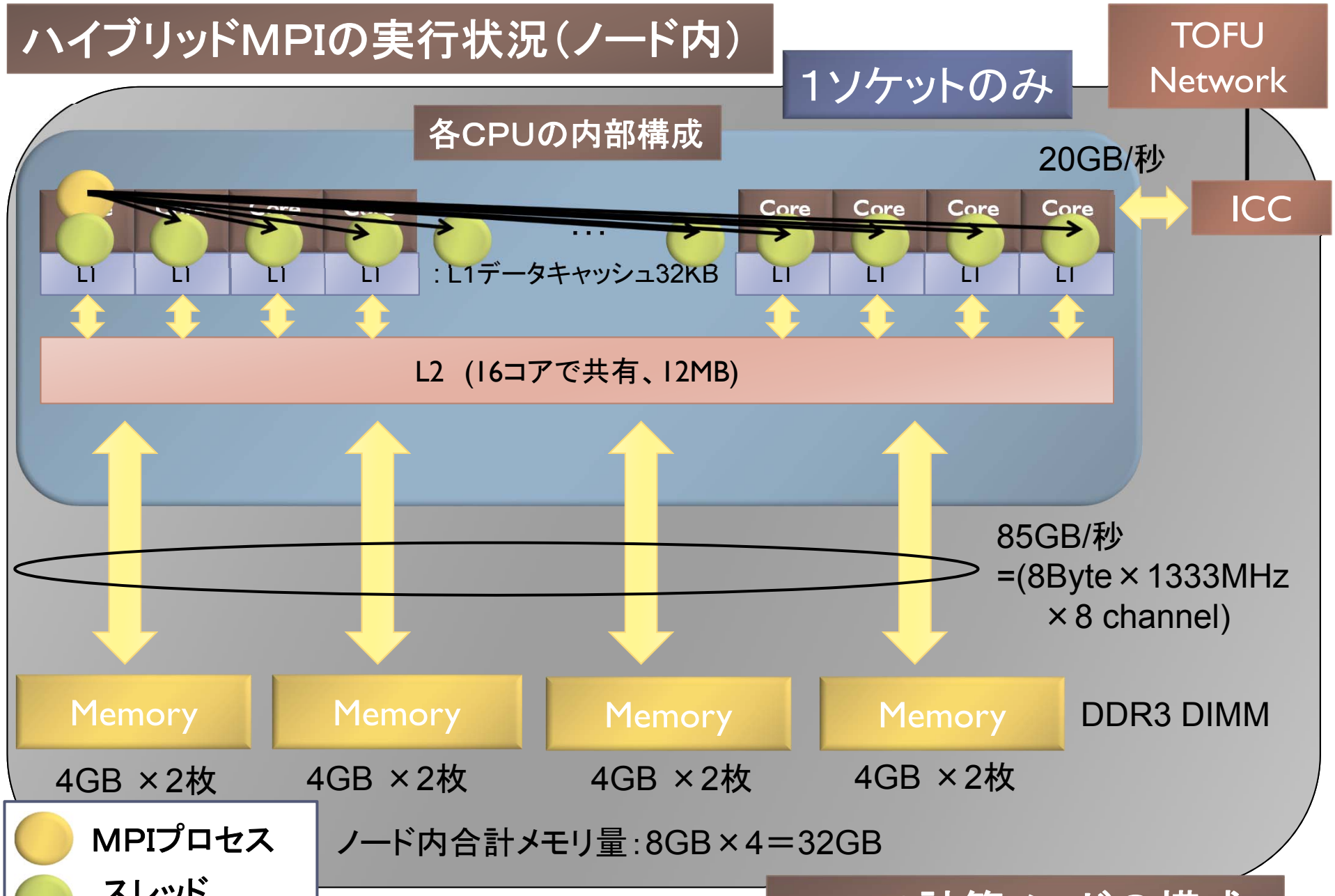
並列版Helloプログラムを実行しよう（ハイブリッドMPI）

1. Helloフォルダ中で以下を実行する
   $ pjsub hello-hy16.bash
2. 自分の導入されたジョブを確認する
   $ pjstat
3. 実行が終了すると、以下のファイルが生成される
   hello-hy16.bash.eXXXXXX
   hello-hy16.bash.oXXXXXX （XXXXXXは数字）
4. 上記標準出力ファイルの中身を見てみる
   $ cat hello-hy16.bash.oXXXXXX
5. “Hello parallel world!”が、
   1プロセス*12ノード=12 個表示されていたら成功。

JOBスクリプトサンプルの説明（ハイブリッドMPI）
（hello-hy16.bash, C言語、Fortran言語共通）



ハイブリッドMPIの実行状況（ノード内）



並列版Helloプログラムの説明（C言語）

このプログラムは、全PEで起動される

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[]) {

    int    myid, numprocs;
    int    ierr, rc;

    ierr = MPI_Init(&argc, &argv);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

    printf("Hello parallel world!  Myid:%d ¥n", myid);

    rc = MPI_Finalize();

    exit(0);
}
```

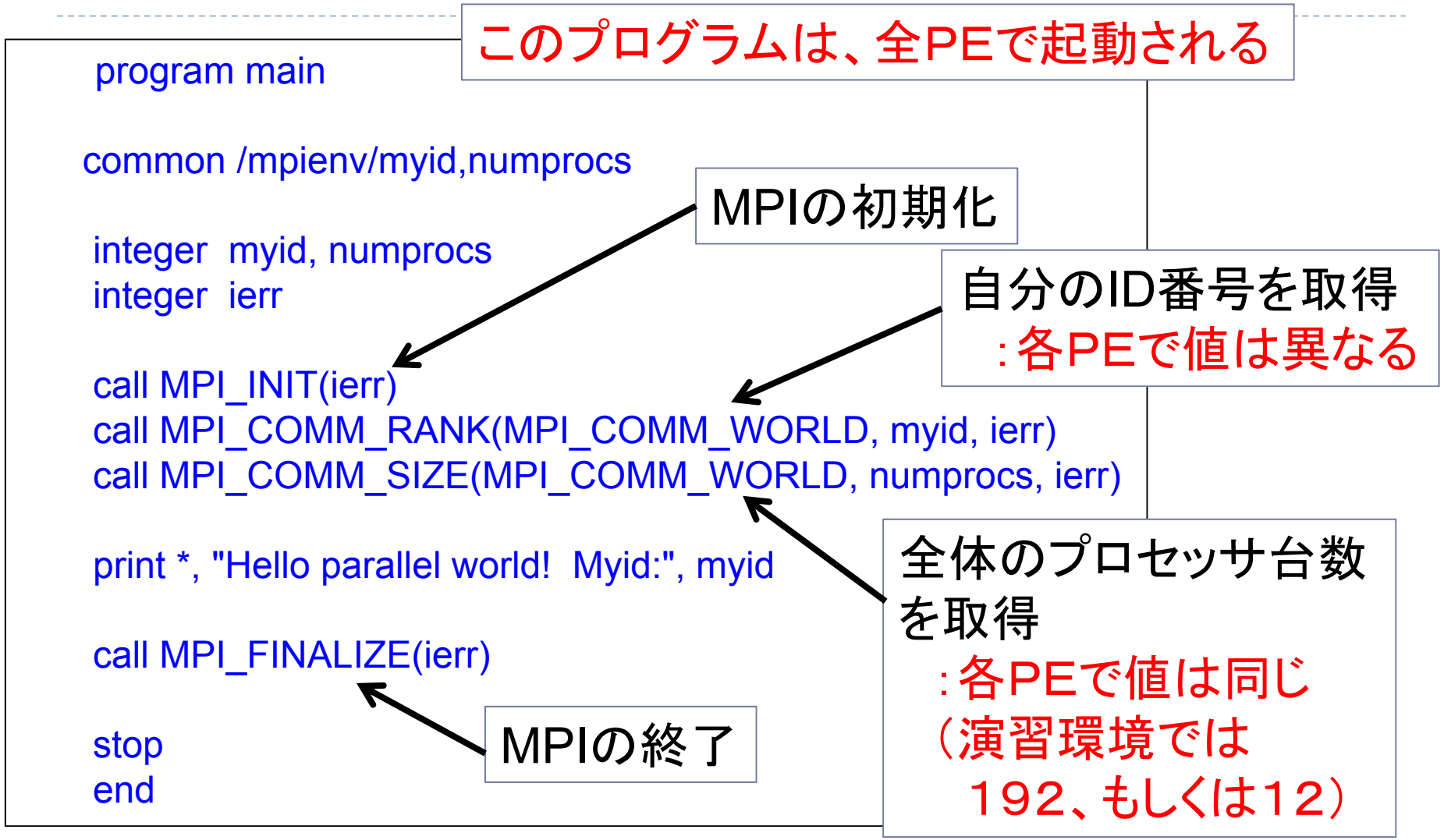
MPIの初期化

自分のID番号を取得
：各PEで値は異なる

全体のプロセッサ台数を取得
：各PEで値は同じ
（演習環境では192、もしくは12）

MPIの終了

並列版Helloプログラムの説明（Fortran言語）



時間計測方法（C言語）

```
double  t0, t1, t2, t_w;
..
ierr = MPI_Barrier(MPI_COMM_WORLD);
t1 = MPI_Wtime();

＜ここに測定したいプログラムを書く＞

ierr = MPI_Barrier(MPI_COMM_WORLD);
t2 = MPI_Wtime();

t0 =  t2 - t1;
ierr = MPI_Reduce(&t0, &t_w, 1,
       MPI_DOUBLE,MPI_MAX, 0,
       MPI_COMM_WORLD);
```

バリア同期後
時間を習得し保存

各プロセッサーで、t0の値は異なる。
この場合は、最も遅いものの値をプロセッサ0番が受け取る

時間計測方法（Fortran言語）

```
  double precision t0, t1, t2, t_w
  double precision MPI_WTIME
..
  call MPI_BARRIER(MPI_COMM_WORLD, ierr)
  t1 = MPI_WTIME(ierr)

  ＜ここに測定したいプログラムを書く＞

  call MPI_BARRIER(MPI_COMM_WORLD, ierr)
  t2 = MPI_WTIME(ierr)

  t0 =  t2 - t1
  call  MPI_REDUCE(t0, t_w, 1,
&       MPI_DOUBLE_PRECISION,
&       MPI_MAX, 0, MPI_COMM_WORLD, ierr)
```

バリア同期後
時間を習得し保存

各プロセッサーで、t0の値は異なる。
この場合は、最も遅いものの値をプロセッサ0番が受け取る

MPI実行時のリダイレクトについて

- FX10スーパーコンピュータシステムでは、
  MPI実行時の入出力のリダイレクトができません
  - ×例）mpirun ./a.out < ./in.txt > ./out.txt

- リダイレクトを行う場合、以下のオプションを指定してください
  - ○例) mpirun --stdin ./in.txt --stdout ./out.txt ./a.out

依存関係のあるジョブの投げ方（ステップジョブ）

- あるジョブスクリプト go1.sh　の後に、go2.sh を投げたい
- さらに、go2.shの後に、go3.shを投げたい、ということがある
- 以上を、ステップジョブという。
- FX10におけるステップジョブの投げ方

1. $pjsub --step go1.sh

   [INFO] PJM 0000 pjsub Job 800967_0 submitted.
2. 上記のジョブ番号800967を覚えておき、以下の入力をする

   $pjsub --step --sparam jid=800967 go2.sh

   [INFO] PJM 0000 pjsub Job 800967_1 submitted
3. 以下同様

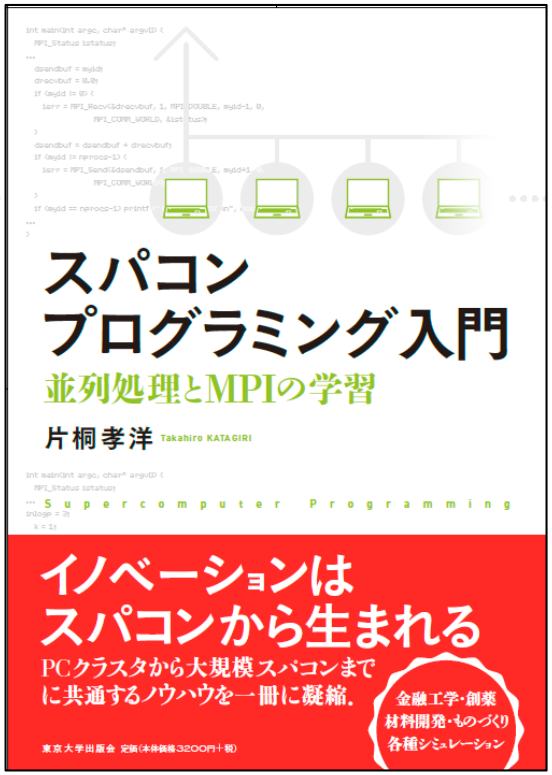
   $pjsub --step --sparam jid=800967 go3.sh

   [INFO] PJM 0000 pjsub Job 800967_2 submitted

並列プログラミングの基礎（座学）

東京大学情報基盤センター　准教授　片桐孝洋

教科書（演習書）



- 「スパコンプログラミング入門
  ー並列処理とMPIの学習ー」
  - 片桐 孝洋　著、
  - 東大出版会、ISBN978-4-13-062453-4、
    発売日：2013年3月12日、判型:A5, 200頁
  - 【本書の特徴】
    - C言語で解説
    - C言語、Fortran90言語のサンプルプログラムが付属
    - 数値アルゴリズムは、図でわかりやすく説明
    - 本講義の内容を全てカバー
    - 内容は初級。初めて並列数値計算を学ぶ人向けの入門書

参考書

- 「スパコンを知る:
  その基礎から最新の動向まで」
  - 岩下武史、片桐孝洋、高橋大介 著
  - 東大出版会、ISBN-10: 4130634550、
    ISBN-13: 978-4130634557、
    発売日：2015年2月20日、176頁
  - 【本書の特徴】
    - スパコンの解説書です。以下を
      分かりやすく解説します。
      - スパコンは何に使えるか
      - スパコンはどんな仕組みで、なぜ速く計算できるのか
      - 最新技術、今後の課題と将来展望、など

参考書

- 「並列数値処理 – 高速化と性能向上のために –」
  - 金田康正 東大教授 理博 編著、
    片桐孝洋 東大特任准教授 博士(理学) 著、黒田久泰 愛媛大准教授 博士(理学) 著、山本有作 神戸大教授 博士(工学) 著、五百木伸洋 ㈱日立製作所 著、
  - コロナ社、発行年月日:2010/04/30 , 判 型: A5, ページ数:272頁、ISBN:978-4-339-02589-7, 定価:3,990円 (本体3,800円＋税5%)
  - 【本書の特徴】
    - Fortran言語で解説
    - 数値アルゴリズムは、数式などで厳密に説明
    - 本講義の内容に加えて、固有値問題の解法、疎行列反復解法、FFT、ソート、など、主要な数値計算アルゴリズムをカバー
    - 内容は中級～上級。専門として並列数値計算を学びたい人向き

本講義の流れ

1. 東大スーパーコンピュータの概略
2. 並列プログラミングの基礎
3. 性能評価指標
4. 基礎的なMPI関数
5. データ分散方式
6. ベクトルどうしの演算
7. ベクトル-行列積
8. リダクション演算

東大スーパーコンピュータの概略

東京大学情報基盤センター スパコン（１／２）

2014年3月10日　運用終了

HITACHI SR16000

Total Peak performance	: 54.9 TFLOPS
Total number of nodes	: 56
Total memory	: 11200 GB
Peak performance per node	: 980.4 GFLOPS
Main memory per node	: 200GB
Disk capacity	: 556 TB

IBM POWER7　3.83GHz (30.64GFLOPS)

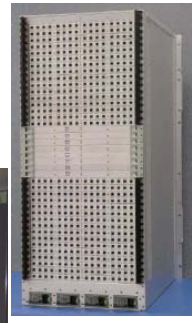
2011年10月～試験運用開始

T2K東大（HA8000クラスタシステム）

Total Peak performance	: 140 TFLOPS
Total number of nodes	: 952
Total memory	: 32000 GB
Peak performance per node	: 147.2 GFLOPS
Main memory per node	: 32 GB, 128 GB
Disk capacity	: 1 PB

AMD Quad Core Opteron 2.3GHz (9.2GFLOPS)

ノード製品名：HITACHI HA8000-tc/RS425



東京大学情報基盤センター スパコン（２／２）

Fujitsu PRIMEHPC FX10 （FX10スーパーコンピュータシステム）

Total Peak performance	: 1.13 PFLOPS
Total number of nodes	: 4,800
Total memory	: 150TB
Peak performance per node	: 236.5 GFLOPS
Main memory per node	: 32 GB
Disk capacity	: 2.1 PB
SPARC64 IXfx　1.848GHz	

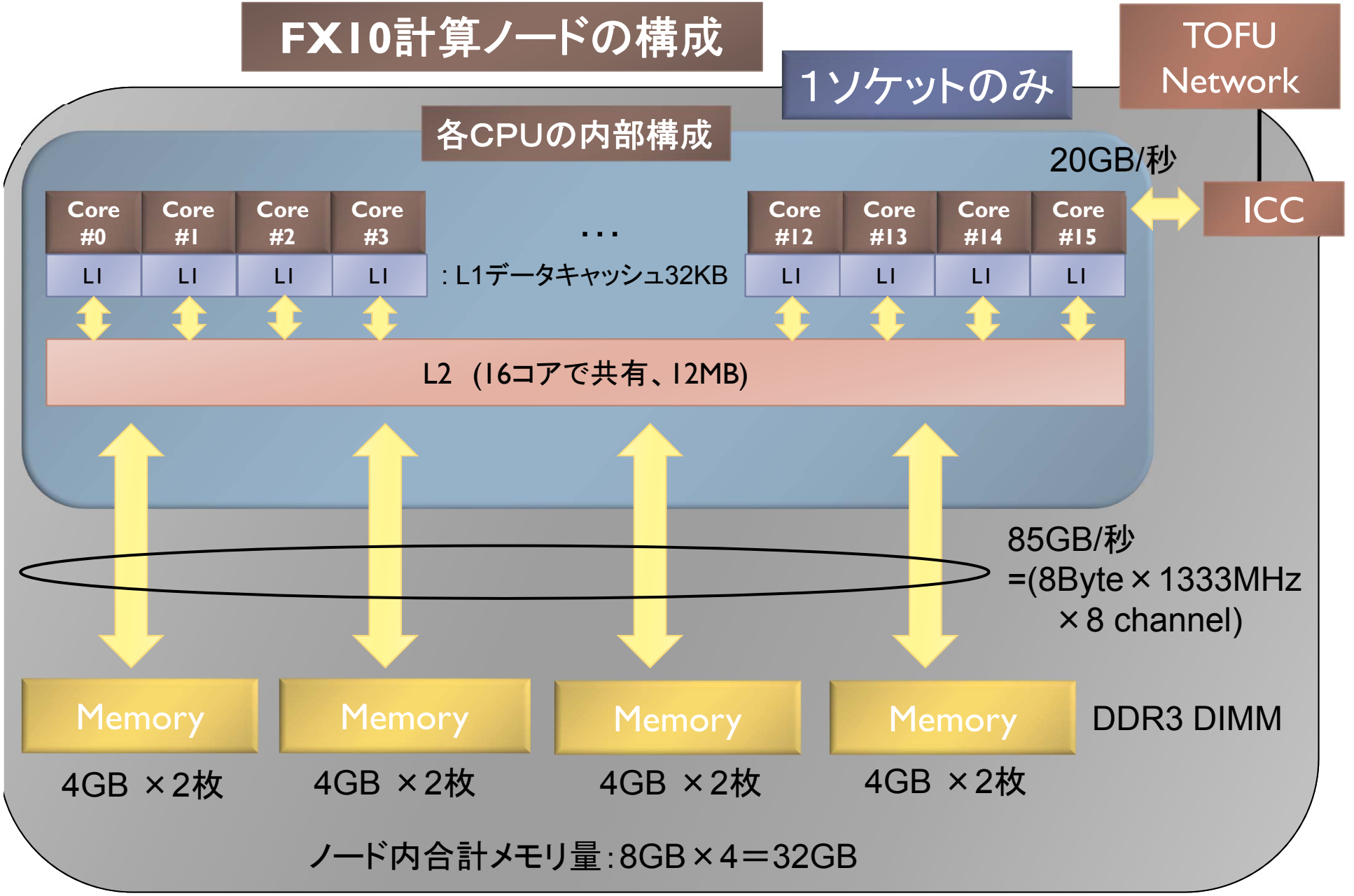
2012年4月2日試験運転開始

2012年7月2日正式運用開始

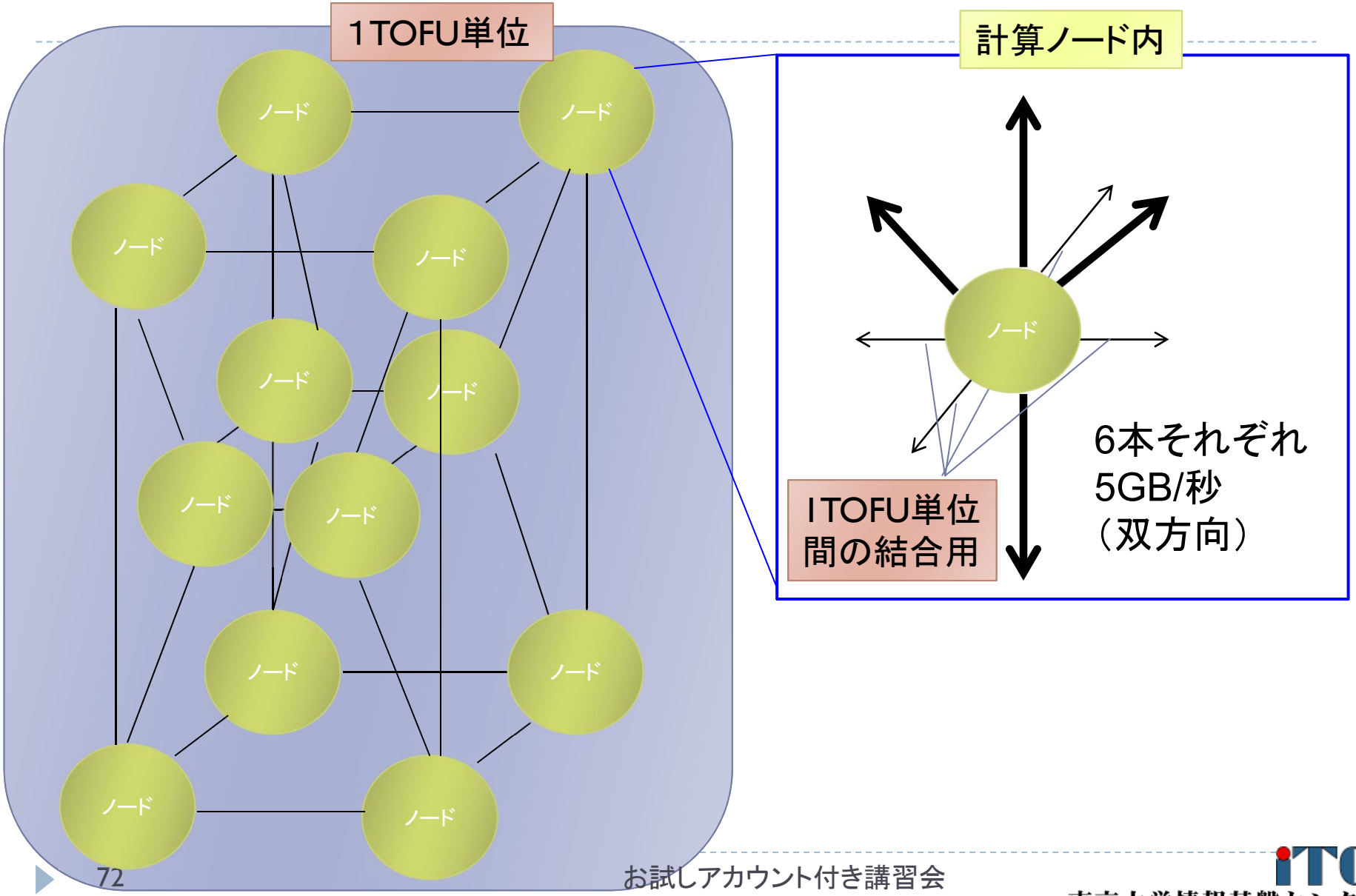
Oakbridge-FX

：長時間ジョブ用のFX10。
ノード数：24～576
制限時間：最大168時間
（１週間）

FX10計算ノードの構成

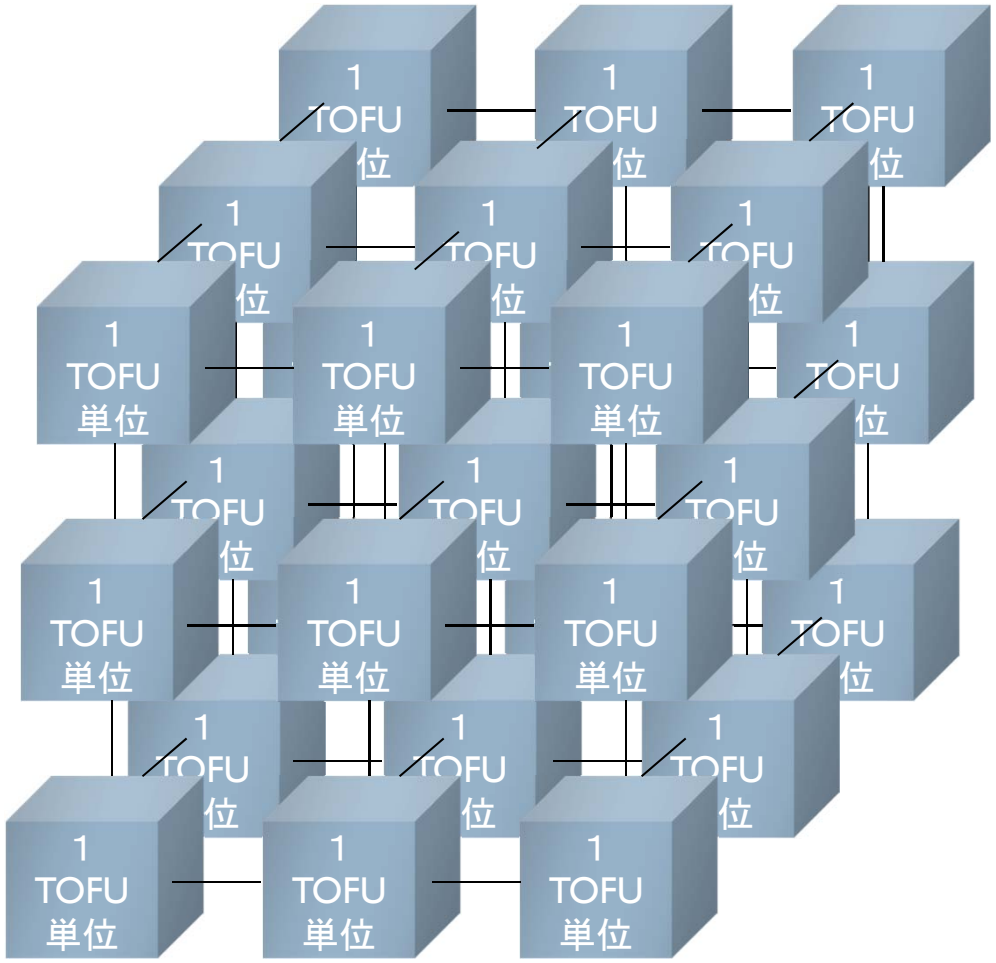


FX10の通信網（１TOFU単位）



FX10の通信網（１TOFU単位間の結合）

３次元接続



- ユーザから見ると、
  X軸、Y軸、Z軸について、
  奥の1TOFUと、手前の
  1TOFUは、繋がってみえます
（３次元トーラス接続）
- ただし物理結線では
  - X軸はトーラス
  - Y軸はメッシュ
  - Z軸はメッシュまたは、
    トーラス

になっています

東大情報基盤センターFX10スーパーコンピュータシステムの料金表（2011年4月1日）

- パーソナルコース（年間）
  - コース1： 120，000円 ： 12ノード（優先）、最大24ノードまで
  - コース2： 250，000円 ： 24ノード（優先）、最大96ノードまで
- グループコース
  - 500，000円 ：1口、12ノード（優先）、最大1440ノードまで
- 以上は、「トークン制」で運営
  - 申し込みノード（優先ノード）×360日×24時間の「トークン」が与えられる
  - 優先ノードまでは、トークン消費係数が1.0
  - 優先ノードを超えると、超えた分は、消費係数が2.0になる

FX10スーパーコンピュータシステムの詳細

- 以下のページをご参照ください
  - 利用申請方法
  - 運営体系
  - 料金体系
  - 利用の手引

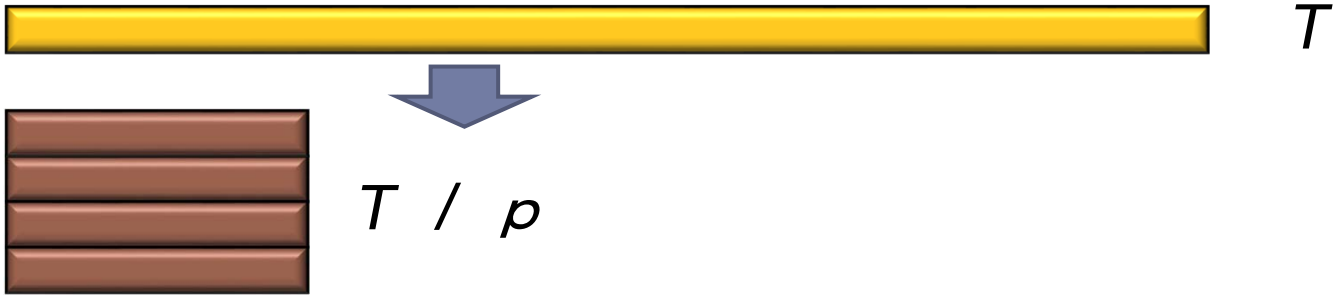
などがご覧になれます。

http://www.cc.u-tokyo.ac.jp/system/fx10/

並列プログラミングの基礎

並列プログラミングとは何か？

- 逐次実行のプログラム（実行時間　T）を、p台の計算機を使って、T / p にすること.



- 素人考えでは自明。
- 実際は、できるかどうかは、対象処理の内容（アルゴリズム）で　大きく　難しさが違う
  - アルゴリズム上、絶対に並列化できない部分の存在
  - 通信のためのオーバヘッドの存在
    - 通信立ち上がり時間
    - データ転送時間

並列と並行

- 並列（Parallel）
  - 物理的に並列（時間的に独立）
  - ある時間に実行されるものは多数

- 並行（Concurrent）
  - 論理的に並列（時間的に依存）
  - ある時間に実行されるものは1つ（＝1プロセッサで実行）

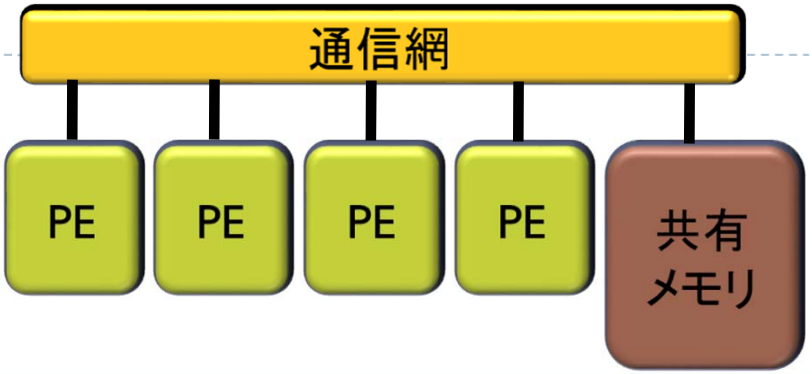
  - 時分割多重、疑似並列
  - OSによるプロセス実行スケジューリング（ラウンドロビン方式）

並列計算機の分類

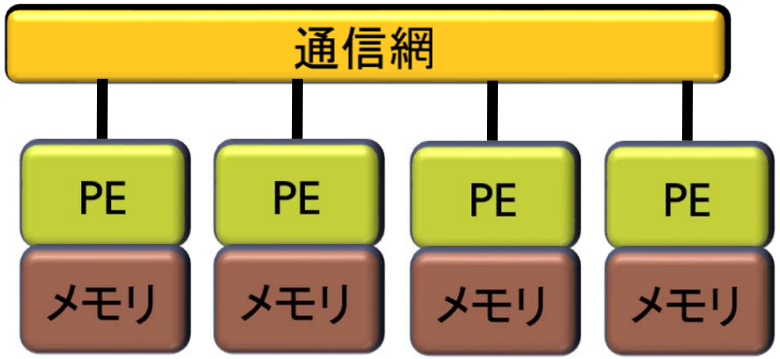
- Michael J. Flynn教授（スタンフォード大）の分類（1966）
- 単一命令・単一データ流
  （SISD, Single Instruction Single Data Stream）
- 単一命令・複数データ流
  （SIMD, Single Instruction Multiple Data Stream）
- 複数命令・単一データ流
  （MISD, Multiple Instruction Single Data Stream）
- 複数命令・複数データ流
  （MIMD, Multiple Instruction Multiple Data Stream）

並列計算機のメモリ型による分類

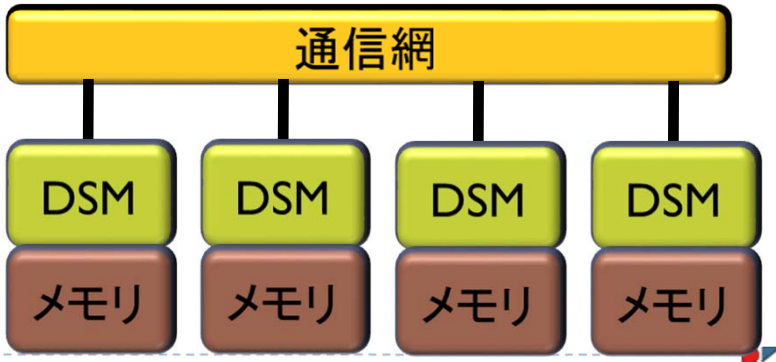
1. 共有メモリ型
（SMP、
Symmetric Multiprocessor）



2. 分散メモリ型
（メッセージパッシング）

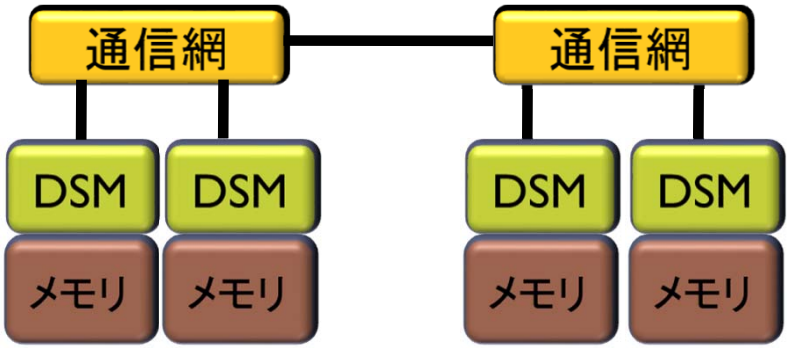


3. 分散共有メモリ型
（DSM、
Distributed Shared Memory）



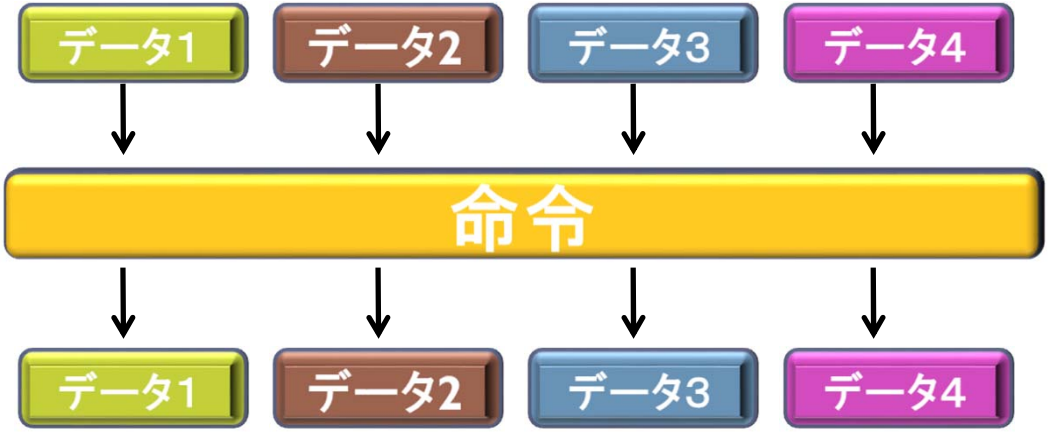
並列計算機のメモリ型による分類

4. 共有・非対称メモリ型
（ccNUMA、
Cache Coherent Non-
Uniform Memory Access）



並列プログラミングのモデル

- 実際の並列プログラムの挙動はMIMD
- アルゴリズムを考えるときは＜SIMDが基本＞
  - 複雑な挙動は理解できないので



並列プログラミングのモデル

- MIMD上での並列プログラミングのモデル
  1. SPMD（Single Program Multiple Data）
     - 1つの共通のプログラムが、並列処理開始時に、全プロセッサ上で起動する
     - MPI（バージョン1）のモデル

  2. Master / Worker（Master / Slave）
     - 1つのプロセス（Master）が、複数のプロセス（Worker）を管理（生成、消去）する。

並列プログラムの種類

- マルチプロセス
  - MPI（Message Passing Interface）
  - HPF（High Performance Fortran）
    - 自動並列化Fortranコンパイラ
    - ユーザがデータ分割方法を明示的に記述
- マルチスレッド
  - Pthread (POSIX スレッド)
  - Solaris Thread (Sun Solaris OS用)
  - NT thread (Windows NT系、Windows95以降)
    - スレッドの Fork（分離）と Join（融合）を明示的に記述
  - Java
    - 言語仕様としてスレッドを規定
  - Open MP
    - ユーザが並列化指示行を記述

プロセスとスレッドの違い
•メモリを意識するかどうかの違い
　•別メモリは「プロセス」
　•同一メモリは「スレッド」

並列処理の実行形態（1）

- データ並列
  - データを分割することで並列化する。
  - データの操作（＝演算）は同一となる。
  - データ並列の例：行列ー行列積

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1*9+2*6+3*3 & 1*8+2*5+3*2 & 1*7+2*4+3*1 \\ 4*9+5*6+6*3 & 4*8+5*5+6*2 & 4*7+5*4+6*1 \\ 7*9+8*6+9*3 & 7*8+8*5+9*2 & 7*7+8*4+9*1 \end{pmatrix}$$

●並列化

全CPUで共有

CPU0 $\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$

CPU1 $\begin{pmatrix} 4 & 5 & 6 \end{pmatrix}$

CPU2 $\begin{pmatrix} 7 & 8 & 9 \end{pmatrix}$

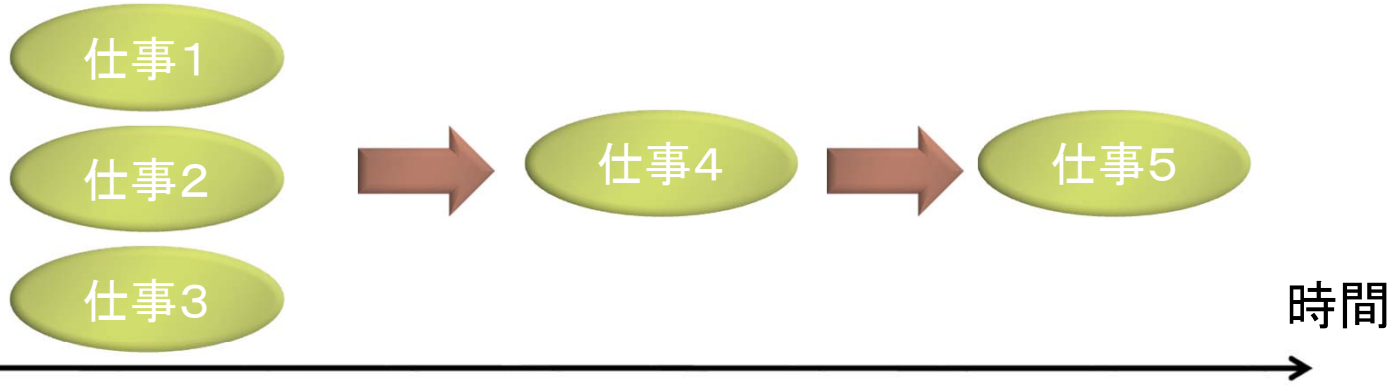
$\begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1*9+2*6+3*3 & 1*8+2*5+3*2 & 1*7+2*4+3*1 \\ 4*9+5*6+6*3 & 4*8+5*5+6*2 & 4*7+5*4+6*1 \\ 7*9+8*6+9*3 & 7*8+8*5+9*2 & 7*7+8*4+9*1 \end{pmatrix}$

並列に計算：初期データは異なるが演算は同一

並列処理の実行形態（２）

- タスク並列
  - タスク（ジョブ）を分割することで並列化する。
  - データの操作（＝演算）は異なるかもしれない。
  - タスク並列の例：カレーを作る
    - 仕事１：野菜を切る
    - 仕事２：肉を切る
    - 仕事３：水を沸騰させる
    - 仕事４：野菜・肉を入れて煮込む
    - 仕事５：カレールゥを入れる

●並列化



MPIの特徴

- メッセージパッシング用のライブラリ規格の1つ
  - メッセージパッシングのモデルである
  - コンパイラの規格、特定のソフトウエアやライブラリを指すものではない！
- 分散メモリ型並列計算機で並列実行に向く
- 大規模計算が可能
  - 1プロセッサにおけるメモリサイズやファイルサイズの制約を打破可能
  - プロセッサ台数の多い並列システム（MPPシステム、Massively Parallel Processingシステム）を用いる実行に向く
    - 1プロセッサ換算で膨大な実行時間の計算を、短時間で処理可能
  - 移植が容易
    - API（Application Programming Interface）の標準化
- スケーラビリティ、性能が高い
  - 通信処理をユーザが記述することによるアルゴリズムの最適化が可能
  - プログラミングが難しい（敷居が高い）

MPIの経緯（1/2）

- MPIフォーラム（http://www.mpi-forum.org/）が仕様策定
  - 1994年5月1.0版（MPI-1）
  - 1995年6月1.1版
  - 1997年7月1.2版、および 2.0版（MPI-2）
- 米国アルゴンヌ国立研究所、およびミシシッピ州立大学で開発
- MPI-2 では、以下を強化：
  - 並列I/O
  - C++、Fortran 90用インターフェース
  - 動的プロセス生成/消滅
    - 主に、並列探索処理などの用途
  - 片方向通信、RMA (Remote Memory Access)

MPIの経緯（2/2）MPI3.0策定

- 以下のページで経緯が公開中
  - http://meetings.mpi-forum.org/MPI_3.0_main_page.php
- 注目すべき機能（検討中）
  - RMA (Remote Memory Access) サポート
  - FT (Fault Tolerant) stabilization、FT を意識したMPIプログラム作成のAPI群とセマンティクス
  - MPIT (Performance Tool)、デバッガやパフォーマンスモニタ、開発環境といったツール群とのインターフェース仕様
  - Neighborhood collectives(物理的に近いノードを対象にした専用コレクティブ通信), 粗通信モデルの為の集団通信API
  - Fortran bindings - improved Fortran bindings, taking into account Fortran 2003 and 2008 features.
  - Non-blocking collective I/O - extend non-blocking collective support to include MPI-I/O
  - Hybrid: Shared memory communicator, Threads, and Endpoint proposals

MPIの実装

- 主要なもの
  - MPICH（エム・ピッチ）
    - 米国アルゴンヌ国立研究所が開発
  - LAM（Local Area Multicomputer）
    - ノートルダム大学が開発
  - その他
    - OpenMPI （FT-MPI, LA-MPI, LAM/MPI, PACX-MPIの統合プロジェクト）
    - YAMPII（東大・石川研究室）（SCore通信機構をサポート）
  - 注意点
    - ヘッダファイル定義の違いにより動作が異なることがある
    - メーカ独自機能の拡張がなされていることがある

MPIによる通信

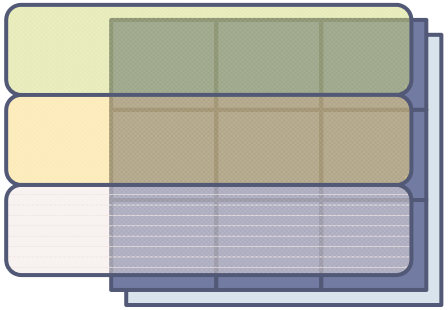
- 郵便物の郵送に同じ
- 郵送に必要な情報：
  1. 自分の住所、送り先の住所
  2. 中に入っているものはどこにあるか
  3. 中に入っているものの分類
  4. 中に入っているものの量
  5. （荷物を複数同時に送る場合の）認識方法（タグ）
- MPIでは：
  1. 自分の認識ID、および、送り先の認識ID
  2. データ格納先のアドレス
  3. データ型
  4. データ量
  5. タグ番号

MPI関数

- システム関数
  - MPI_Init；MPI_Comm_rank；MPI_Comm_size；MPI_Finalize；
- １対１通信関数
  - ブロッキング型
    - MPI_Send； MPI_Recv；
  - ノンブロッキング型
    - MPI_Isend； MPI_Irecv；
- １対全通信関数
  - MPI_Bcast
- 集団通信関数
  - MPI_Reduce； MPI_Allreduce； MPI_Barrier；
- 時間計測関数
  - MPI_Wtime

コミュニケータ

- MPI_COMM_WORLDは、コミュニケータとよばれる概念を保存する変数
- コミュニケータは、操作を行う対象のプロセッサ群を定める
- 初期状態では、0番～numprocs –1番までのプロセッサが、1つのコミュニケータに割り当てられる
  - この名前が、“MPI_COMM_WORLD”
- プロセッサ群を分割したい場合、MPI_Comm_split 関数を利用
  - メッセージを、一部のプロセッサ群に放送するときに利用
  - “マルチキャスト”で利用

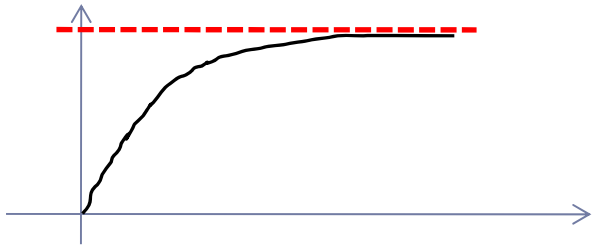


性能評価指標

並列化の尺度

性能評価指標－台数効果

- 台数効果
  - 式: $S_P = T_S / T_P \quad (0 \leq S_p)$
  - T_S :逐次の実行時間、 T_P :P台での実行時間
  - P台用いて $S_P = P$ のとき、理想的な(ideal)速度向上
  - P台用いて $S_P > P$ のとき、スーパリニア・スピードアップ
    - 主な原因は、並列化により、データアクセスが局所化されて、キャッシュヒット率が向上することによる高速化
- 並列化効率
  - 式: $E_P = S_P / P \times 100 \ (0 \leq E_p) \ [\%]$
- 飽和性能
  - 速度向上の限界
  - Saturation、「さちる」



アムダールの法則

- 逐次実行時間を　K　とする。
  そのうち、並列化ができる割合を　α　とする。
- このとき、台数効果は以下のようになる。

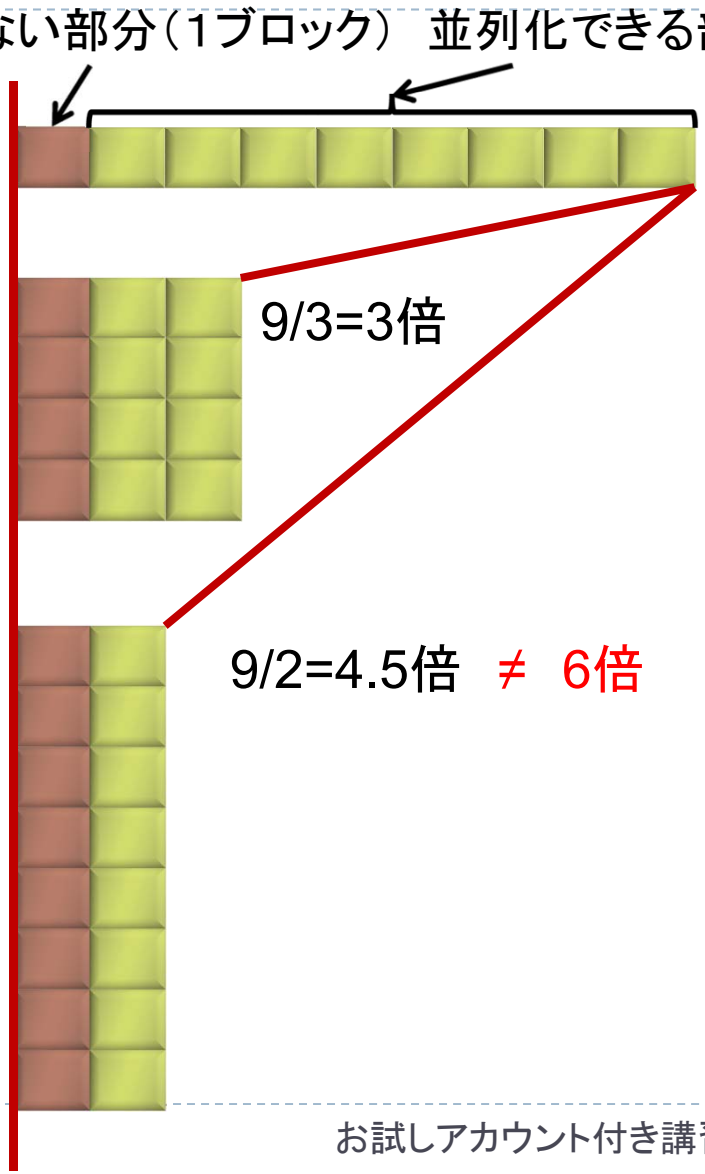
$$S_P = K/(K\alpha/P + K(1-\alpha))$$

$$= 1/(\alpha/P + (1-\alpha)) = 1/(\alpha(1/P-1)+1)$$

- 上記の式から、たとえ無限大の数のプロセッサを使っても(P→∞)、台数効果は、高々　1／(1－α)　である。
  (アムダールの法則)
  - 全体の90%が並列化できたとしても、無限大の数のプロセッサをつかっても、1/(1-0.9) = 10 倍　にしかならない！

  →高性能を達成するためには、少しでも並列化効率を上げる実装をすることがとても重要である

アムダールの法則の直観例



●逐次実行

＝88.8%が並列化可能

●並列実行（4並列）

9/3=3倍

●並列実行（8並列）

9/2=4.5倍 ≠ 6倍

略語とMPI用語

- MPIは「プロセス」間の通信を行います。プロセスは（普通は）「プロセッサ」（もしくは、コア）に一対一で割り当てられます。
- 今後、「MPIプロセス」と書くのは長いので、ここではPE（Processer Elementsの略）と書きます。
  - ただし用語として「PE」は現在はあまり使われていません。
- ランク（Rank）
  - 各「MPIプロセス」の「識別番号」のこと。
  - 通常MPIでは、MPI_Comm_rank関数で設定される変数（サンプルプログラムではmyid）に、0～全PE数－1　の数値が入る
  - 世の中の全MPIプロセス数を知るには、MPI_Comm_size関数を使う。

  （サンプルプログラムでは、numprocs に、この数値が入る）

基本的なMPI関数

送信、受信のためのインタフェース

C言語インターフェースと Fortranインターフェースの違い

- C版は、 整数変数ierr が戻り値

  ierr = MPI_Xxxx(….);

- Fortran版は、最後に整数変数ierrが引数

  call MPI_XXXX(…., ierr)

- システム用配列の確保の仕方
  - C言語

    MPI_Status istatus;
  - Fortran言語

    integer istatus(MPI_STATUS_SIZE)

C言語インターフェースと Fortranインターフェースの違い

- MPIにおける、データ型の指定
  - C言語

    MPI_CHAR (文字型) 、MPI_INT (整数型)、MPI_FLOAT (実数型)、 MPI_DOUBLE(倍精度実数型)
  - Fortran言語

    MPI_CHARACTER (文字型) 、MPI_INTEGER (整数型)、MPI_REAL (実数型)、MPI_DOUBLE_PRECISION(倍精度実数型) 、MPI_COMPLEX(複素数型)
- 以降は、C言語インタフェースで説明する

基礎的なMPI関数—MPI_Recv（１／２）

- ierr = MPI_Recv(recvbuf, icount, idatatype, isource, itag, icomm, **istatus**);

  - recvbuf : 受信領域の先頭番地を指定する。
  - icount : 整数型。受信領域のデータ要素数を指定する。
  - idatatype : 整数型。受信領域のデータの型を指定する。
    - MPI_CHAR (文字型) 、MPI_INT (整数型)、MPI_FLOAT (実数型)、 MPI_DOUBLE(倍精度実数型)
  - isource : 整数型。受信したいメッセージを送信するPEのランクを指定する。
    - 任意のPEから受信したいときは、MPI_ANY_SOURCE を指定する。

基礎的なMPI関数—MPI_Recv（２／２）

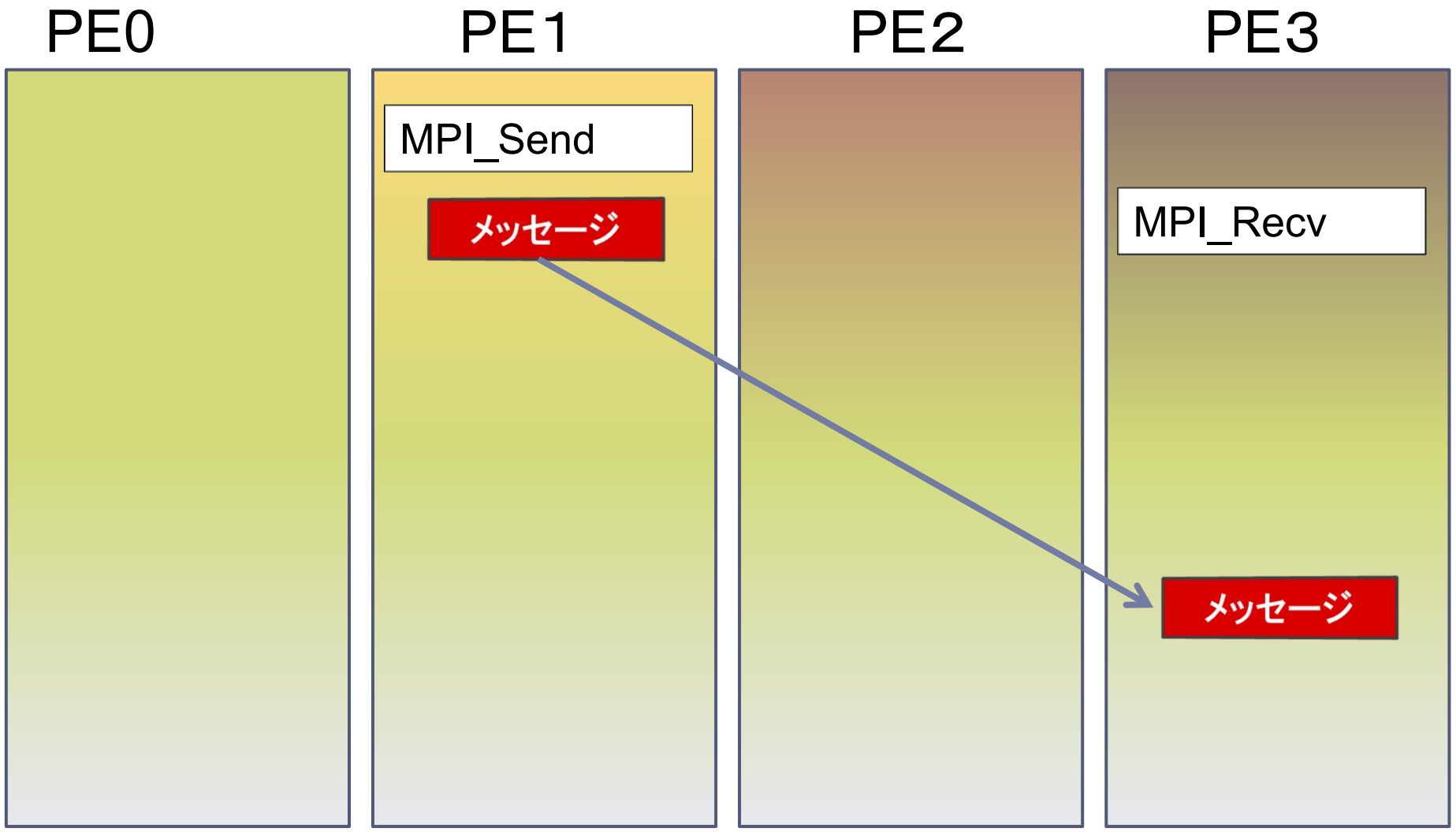
- itag： 整数型。受信したいメッセージに付いているタグの値を指定する。
  - 任意のタグ値のメッセージを受信したいときは、MPI_ANY_TAG を指定する。
- icomm： 整数型。PE集団を認識する番号であるコミュニケータを指定する。
  - 通常ではMPI_COMM_WORLD を指定すればよい。
- istatus： MPI_Status型（整数型の配列）。受信状況に関する情報が入る。かならず専用の型宣言をした配列を確保すること。
  - 要素数がMPI_STATUS_SIZEの整数配列が宣言される。
  - 受信したメッセージの送信元のランクが istatus[MPI_SOURCE]、タグが istatus[MPI_TAG] に代入される。
- ierr(戻り値)： 整数型。エラーコードが入る。

基礎的なMPI関数—MPI_Send

```
ierr = MPI_Send(sendbuf, icount, idatatype, idest,
       itag, icomm);
```

- sendbuf : 送信領域の先頭番地を指定する
- icount : 整数型。送信領域のデータ要素数を指定する
- idatatype : 整数型。送信領域のデータの型を指定する
- idest : 整数型。送信したいPEのicomm内でのランクを指定する。
- itag : 整数型。受信したいメッセージに付けられたタグの値を指定する。
- icomm : 整数型。プロセッサー集団を認識する番号であるコミュニケータを指定する。
- ierr (戻り値) : 整数型。エラーコードが入る。

Send－Recvの概念（１対１通信）

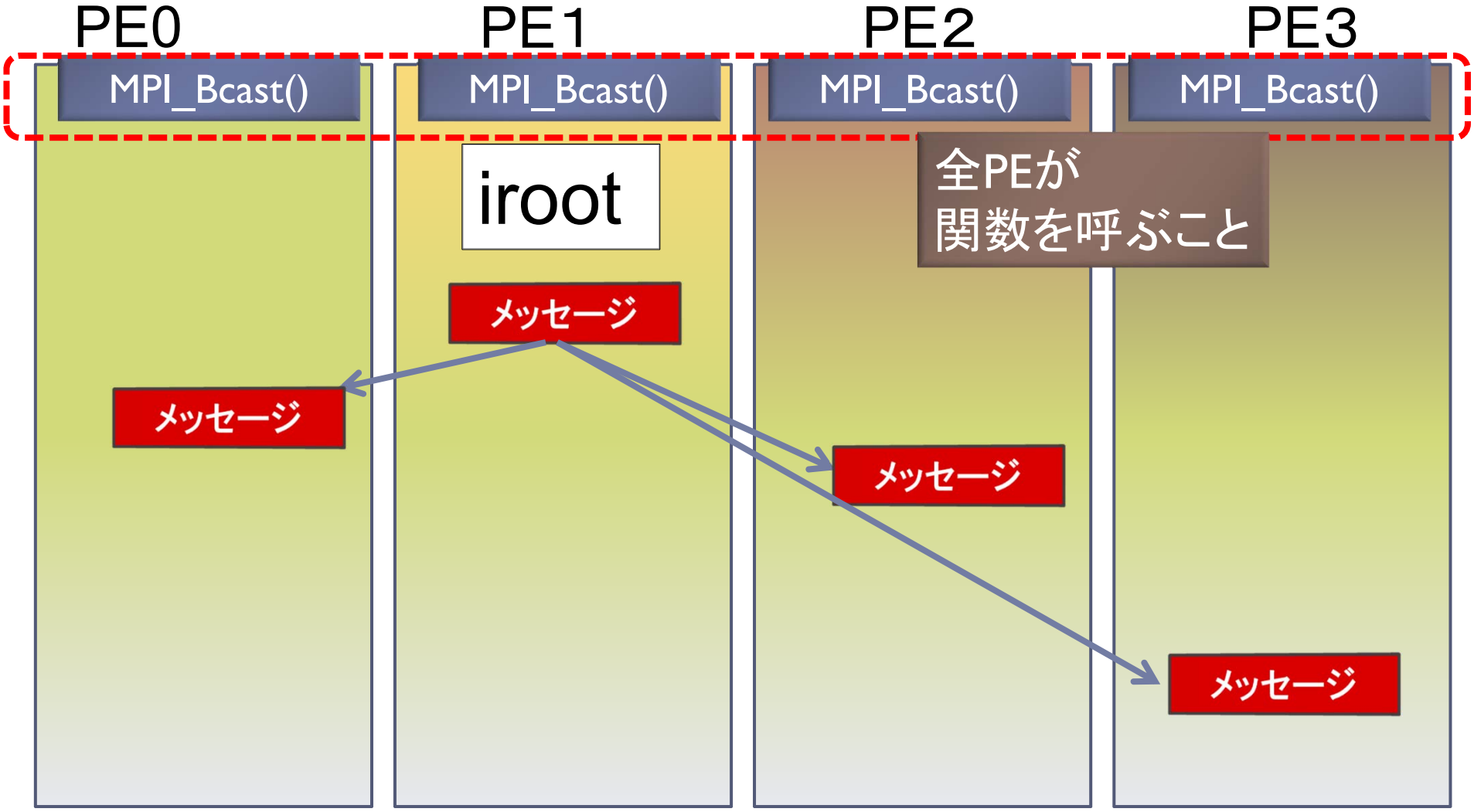


基礎的なMPI関数—MPI_Bcast

```
ierr = MPI_Bcast(sendbuf,  icount,  idatatype,
       iroot,  icomm);
```

- sendbuf： 送信および受信領域の先頭番地を指定する。
- icount： 整数型。送信領域のデータ要素数を指定する。
- idatatype： 整数型。送信領域のデータの型を指定する。
- iroot： 整数型。送信したいメッセージがあるPEの番号を指定する。全PEで同じ値を指定する必要がある。
- icomm： 整数型。PE集団を認識する番号であるコミュニケータを指定する。
- ierr (戻り値)： 整数型。エラーコードが入る。

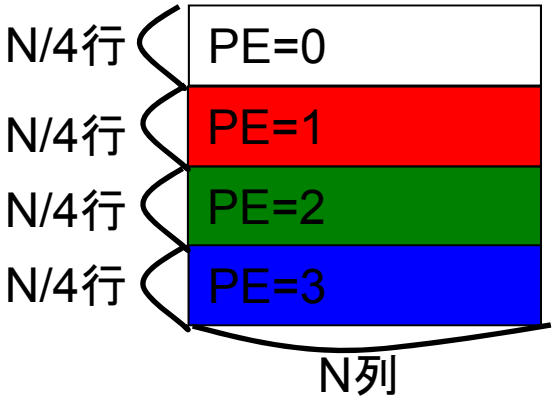
MPI_Bcastの概念（集団通信）



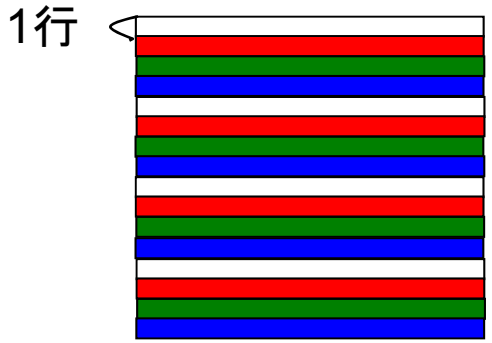
1．基本演算

- 逐次処理では、「データ構造」が重要
- 並列処理においては、「データ分散方法」が重要になる！
  1. 各PEの「演算負荷」を均等にする
     - ロード・バランシング：　並列処理の基本操作の一つ
     - 粒度調整
  2. 各PEの「利用メモリ量」を均等にする
  3. 演算に伴う通信時間を短縮する
  4. 各PEの「データ・アクセスパターン」を高速な方式にする
     （＝逐次処理におけるデータ構造と同じ）
- 行列データの分散方法
  - ＜次元レベル＞：　1次元分散方式、2次元分散方式
  - ＜分割レベル＞：　ブロック分割方式、サイクリック（循環）分割方式

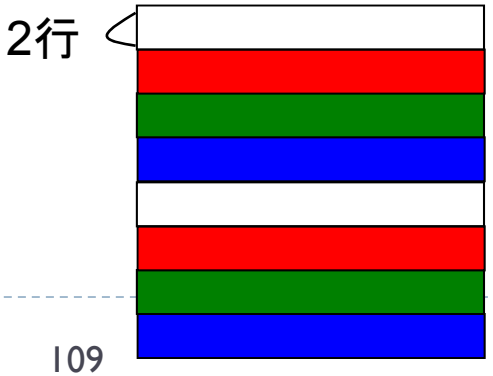
1.1.1 1次元分散



- （行方向） ブロック分割方式
- (Block, *) 分散方式



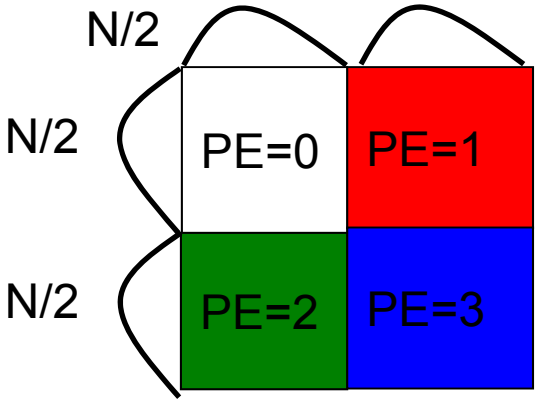
- (行方向) サイクリック分割方式
- (Cyclic, *) 分散方式



- (行方向)ブロック・サイクリック分割方式
- (Cyclic(2), *) 分散方式

この例の「2」: ＜ブロック幅＞とよぶ

1.1.2　2次元分散



- ブロック・ブロック分割方式
- (Block, Block)分散方式

0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3

- サイクリック・サイクリック分割方式
- (Cyclic, Cyclic)分散方式

0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1
2	2	3	3	2	2	3	3
2	2	3	3	2	2	3	3
0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1
2	2	3	3	2	2	3	3
2	2	3	3	2	2	3	3

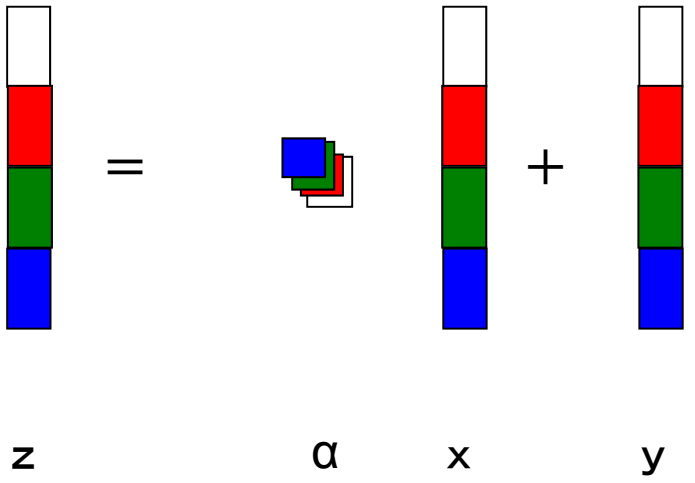
- 二次元ブロック・サイクリック分割方式
- (Cyclic(2), Cyclic(2))分散方式

1.2 ベクトルどうしの演算

- 以下の演算

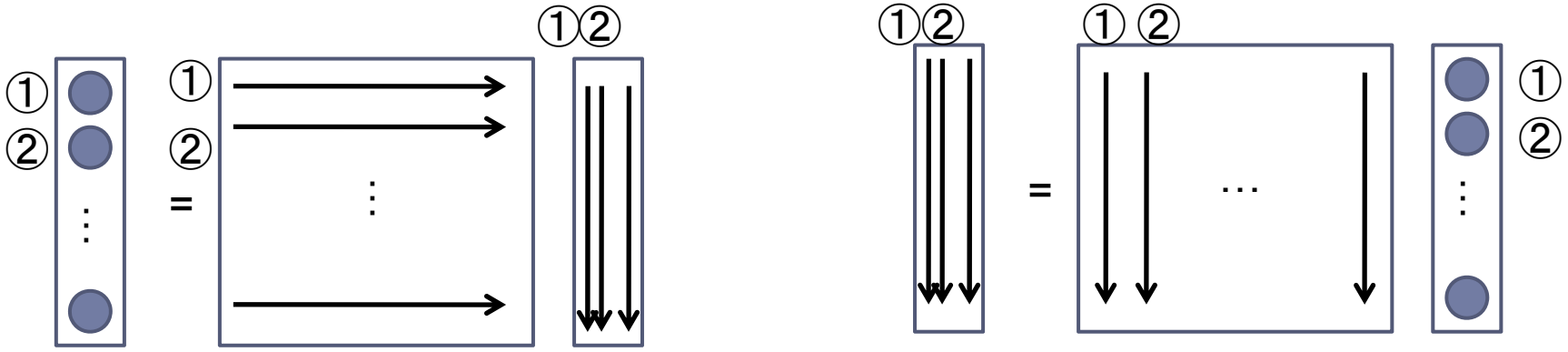
$$z = ax + y$$

  - ここで、αはスカラ、z、x、y　はベクトル
- どのようなデータ分散方式でも並列処理が可能
  - ただし、スカラ α は全PEで所有する。
  - ベクトルはO(n)のメモリ領域が必要なのに対し、スカラはO(1)のメモリ領域で大丈夫。
    →スカラメモリ領域は無視可能
  - 計算量：O(N/P)
  - あまり面白くない



1.3 行列とベクトルの積

- ＜行方式＞と＜列方式＞がある。
  - ＜データ分散方式＞と＜方式＞組のみ合わせがあり、少し面白い




```
for(i=0;i<n;i++){
    y[i]=0.0;
    for(j=0;j<n;j++){
        y[i] += a[i][j]*x[j];
    }
}
```

＜行方式＞： 自然な実装
C言語向き

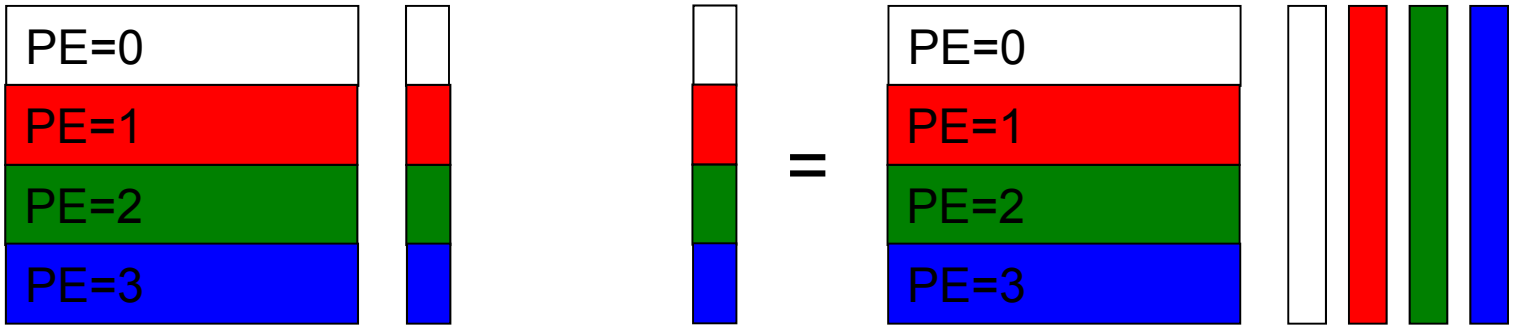
```
for(j=0; j<n; j++) y[j]=0.0;
for(j=0; j<n; j++) {
   for (i=0; i<n; i++) {
      y[i] += a[i][j]*x[j];
   }
}
```

＜列方式＞： Fortran言語向き

1.3 行列とベクトルの積

＜行方式の場合＞

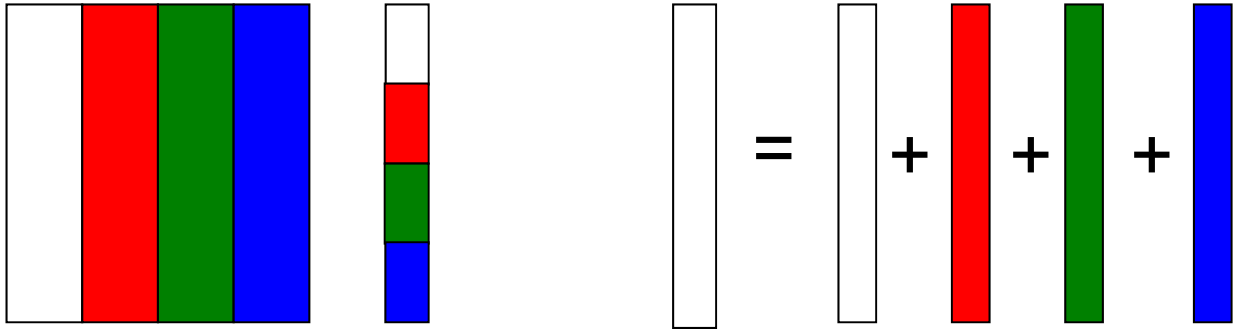
＜行方向分散方式＞　：行方式に向く分散方式



右辺ベクトルを MPI_Allgather関数を利用し、全PEで所有する

各PE内で行列ベクトル積を行う

＜列方向分散方式＞　：ベクトルの要素すべてがほしいときに向く



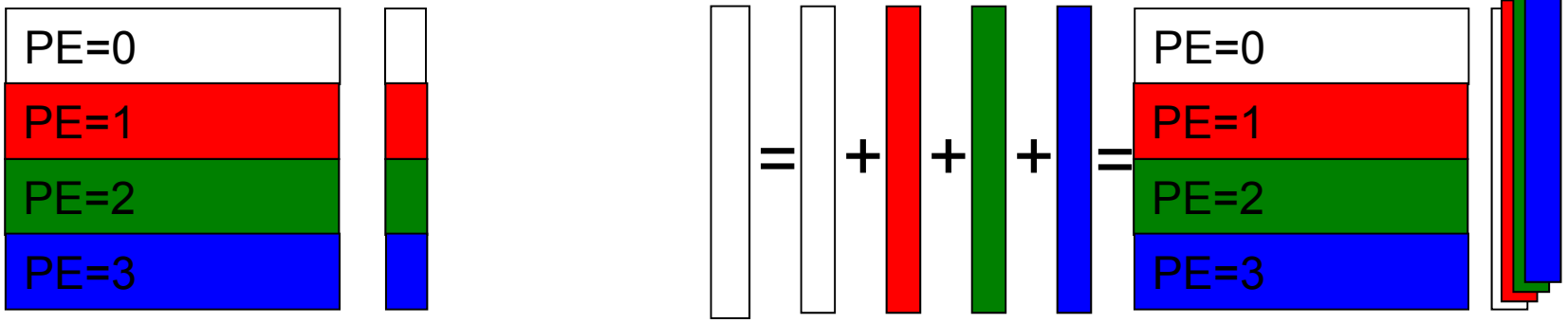
各PE内で行列-ベクトル積を行う

MPI_Reduce関数で総和を求める
（※ある1PEにベクトルすべてが集まる）

1.3 行列とベクトルの積

＜列方式の場合＞

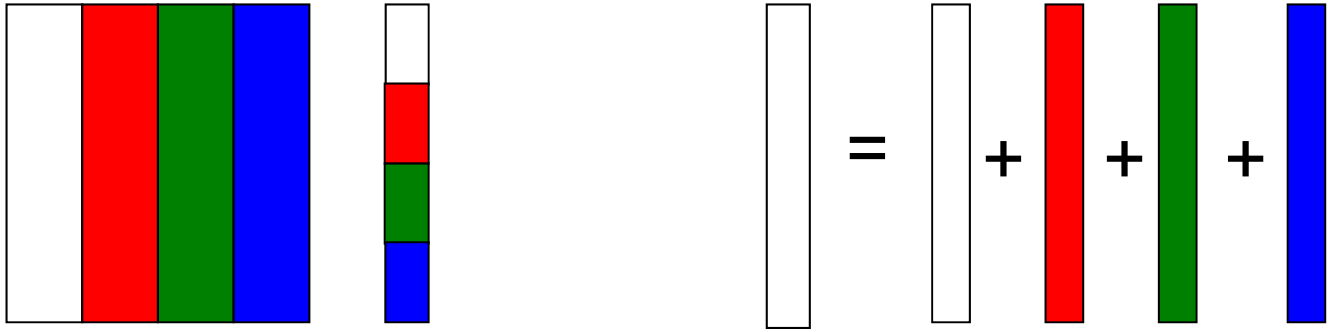
＜行方向分散方式＞　：無駄が多く使われない



右辺ベクトルを MPI_Allgather関数を利用して、全PEで所有する

結果をMPI_Reduce関数により総和を求める

＜列方向分散方式＞　：列方式に向く分散方式



各PE内で行列-ベクトル積を行う

MPI_Reduce関数で総和を求める
（※ある1PEにベクトルすべてが集まる）

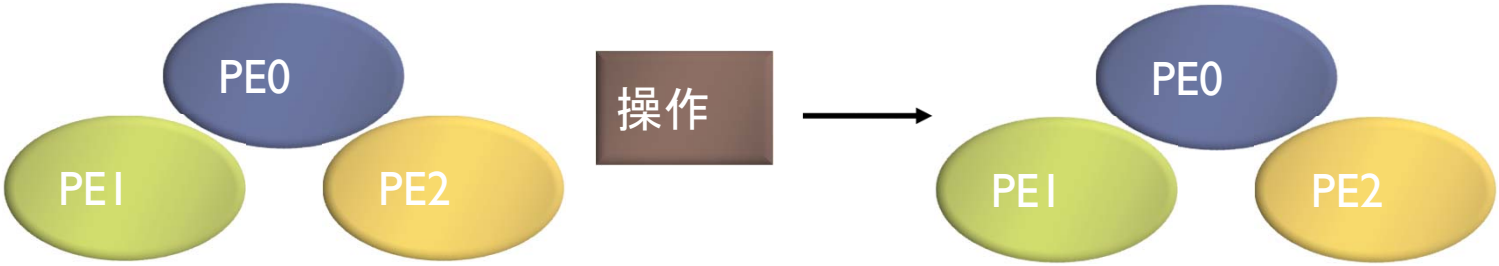
1.7　リダクション演算

- ＜操作＞によって＜次元＞を減少（リダクション）させる処理
  - 例：　内積演算
    ベクトル（n次元空間）　→　スカラ（1次元空間）
- リダクション演算は、通信と計算を必要とする
  - 集団通信演算（collective communication operation）と呼ばれる
- 演算結果の持ち方の違いで、2種のインタフェースが存在する

1.7　リダクション演算

- 演算結果に対する所有PEの違い
  - MPI_Reduce関数
    - リダクション演算の結果を、ある一つのPEに所有させる

  - MPI_Allreduce関数
    - リダクション演算の結果を、全てのPEに所有させる



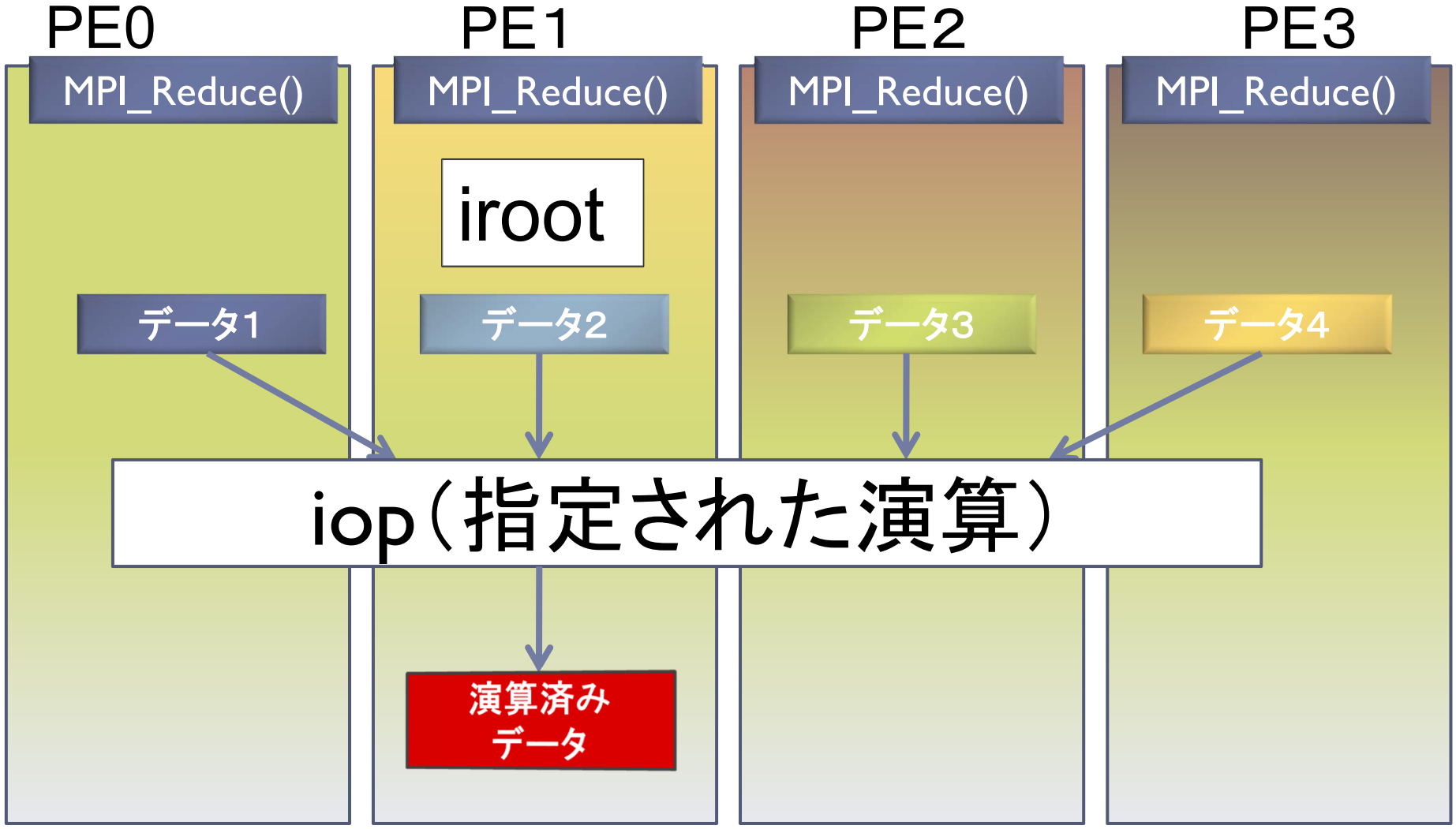
基礎的なMPI関数—MPI_Reduce

- ierr = MPI_Reduce(sendbuf, recvbuf, icount,
  idatatype, iop, iroot, icomm);
  - sendbuf : 送信領域の先頭番地を指定する。
  - recvbuf : 受信領域の先頭番地を指定する。iroot で指定したPEのみで書き込みがなされる。
    領域と受信領域は、同一であってはならない。
    すなわち、異なる配列を確保しなくてはならない。
  - icount : 整数型。送信領域のデータ要素数を指定する。
  - idatatype : 整数型。送信領域のデータの型を指定する。
    - ＜最小／最大値と位置＞を返す演算を指定する場合は、MPI_2INT(整数型)、 MPI_2FLOAT (単精度型)、MPI_2DOUBLE(倍精度型) 、 を指定する。

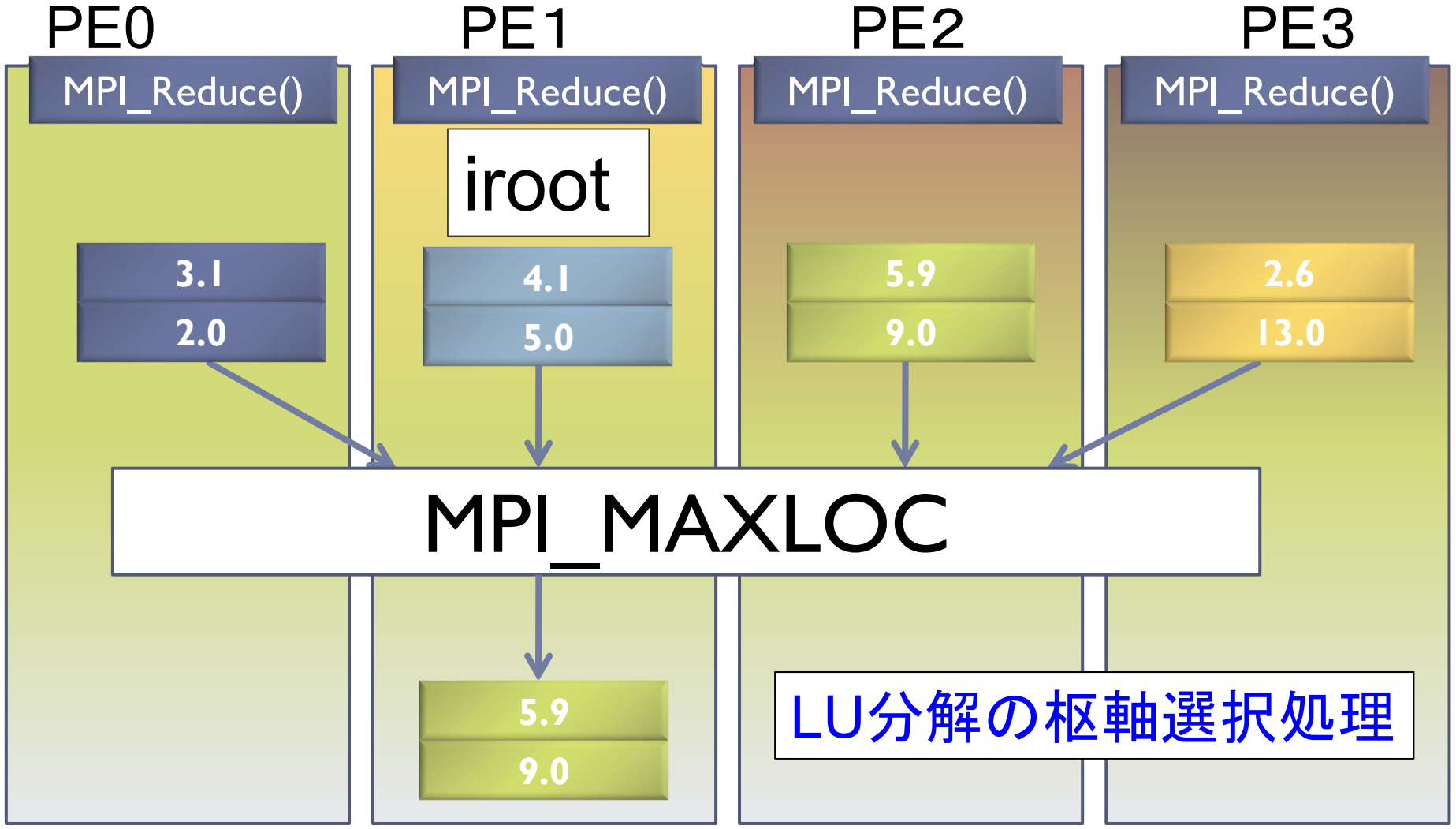
基礎的なMPI関数—MPI_Reduce

- iop :　整数型。演算の種類を指定する。
  - MPI_SUM (総和)、MPI_PROD (積)、MPI_MAX (最大)、MPI_MIN (最小)、 MPI_MAXLOC (最大と位置)、MPI_MINLOC (最小と位置) など。
- iroot :　整数型。結果を受け取るPEのicomm 内でのランクを指定する。全てのicomm 内のPEで同じ値を指定する必要がある。
- icomm :　整数型。PE集団を認識する番号であるコミュニケータを指定する。
- ierr : 整数型。 エラーコードが入る。

MPI_Reduceの概念（集団通信）



MPI_Reduceによる２リスト処理例（MPI_2DOUBLEとMPI_MAXLOC）



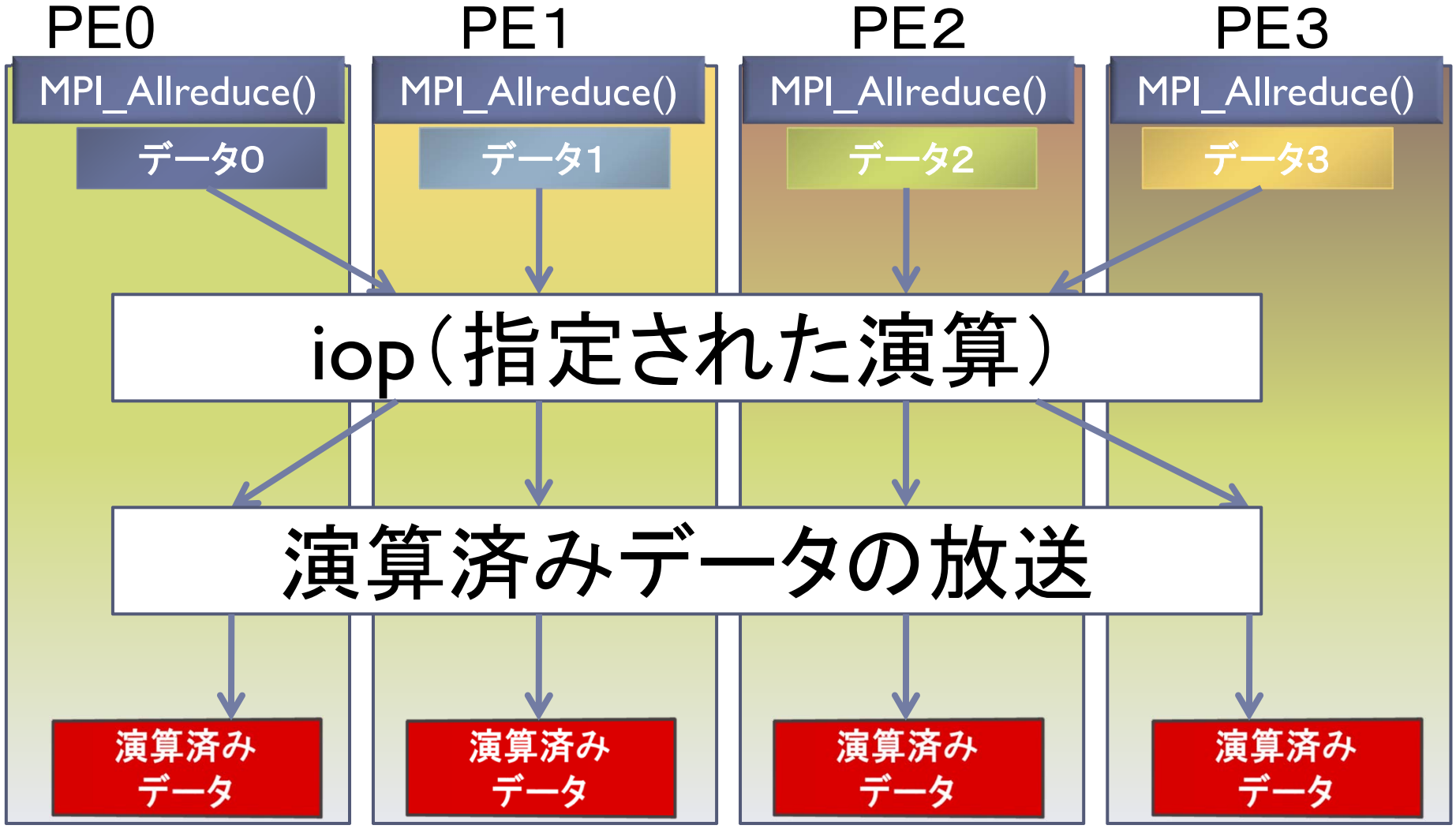
基礎的なMPI関数—MPI_Allreduce

- ierr = MPI_Allreduce(sendbuf, recvbuf, icount, idatatype, iop, icomm);
  - sendbuf : 送信領域の先頭番地を指定する。
  - recvbuf : 受信領域の先頭番地を指定する。iroot で指定したPEのみで書き込みがなされる。
    領域と受信領域は、同一であってはならない。すなわち、異なる配列を確保しなくてはならない。
  - icount : 整数型。送信領域のデータ要素数を指定する。
  - idatatype : 整数型。送信領域のデータの型を指定する。
    - 最小値や最大値と位置を返す演算を指定する場合は、MPI_2INT(整数型)、MPI_2FLOAT (単精度型)、MPI_2DOUBLE(倍精度型) を指定する。

基礎的なMPI関数—MPI_Allreduce

- iop :　整数型。演算の種類を指定する。
  - MPI_SUM (総和)、MPI_PROD (積)、MPI_MAX (最大)、MPI_MIN (最小)、 MPI_MAXLOC (最大と位置)、MPI_MINLOC (最小と位置) など。
- icomm :　整数型。PE集団を認識する番号であるコミュニケータを指定する。
- ierr : 整数型。 エラーコードが入る。

MPI_Allreduceの概念（集団通信）

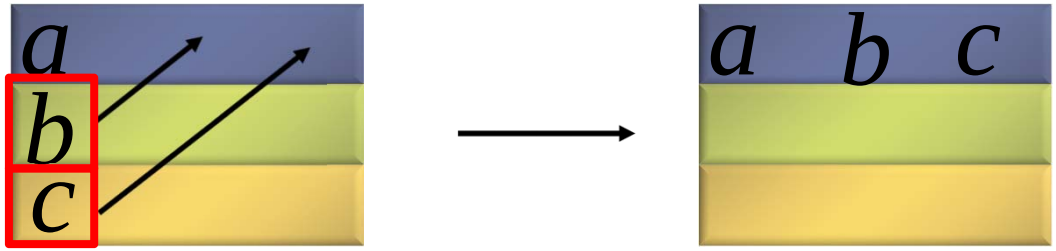


1.7　リダクション演算

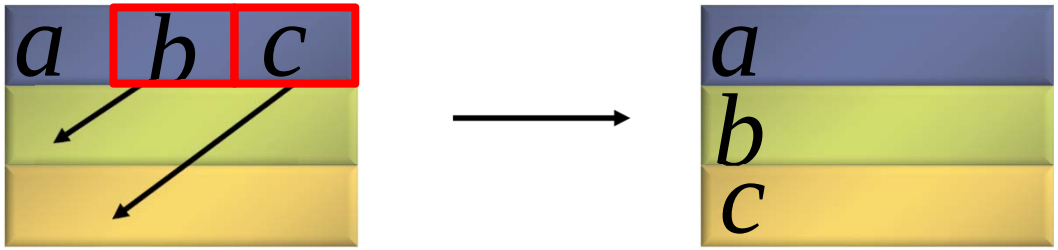
- 性能について
  - リダクション演算は、1対1通信に比べ遅い
    - プログラム中で多用すべきでない！
  - MPI_Allreduce は MPI_Reduce に比べ遅い
    - MPI_Allreduce は、放送処理が入る。
    - なるべく、MPI_Reduce を使う。

1.4　行列の転置

- 行列 A が（Block, ＊）分散されているとする。
- 行列 A の転置行列 A^T を作るには、MPIでは次の2通りの関数を用いる
  - MPI_Gather関数



  - MPI_Scatter関数



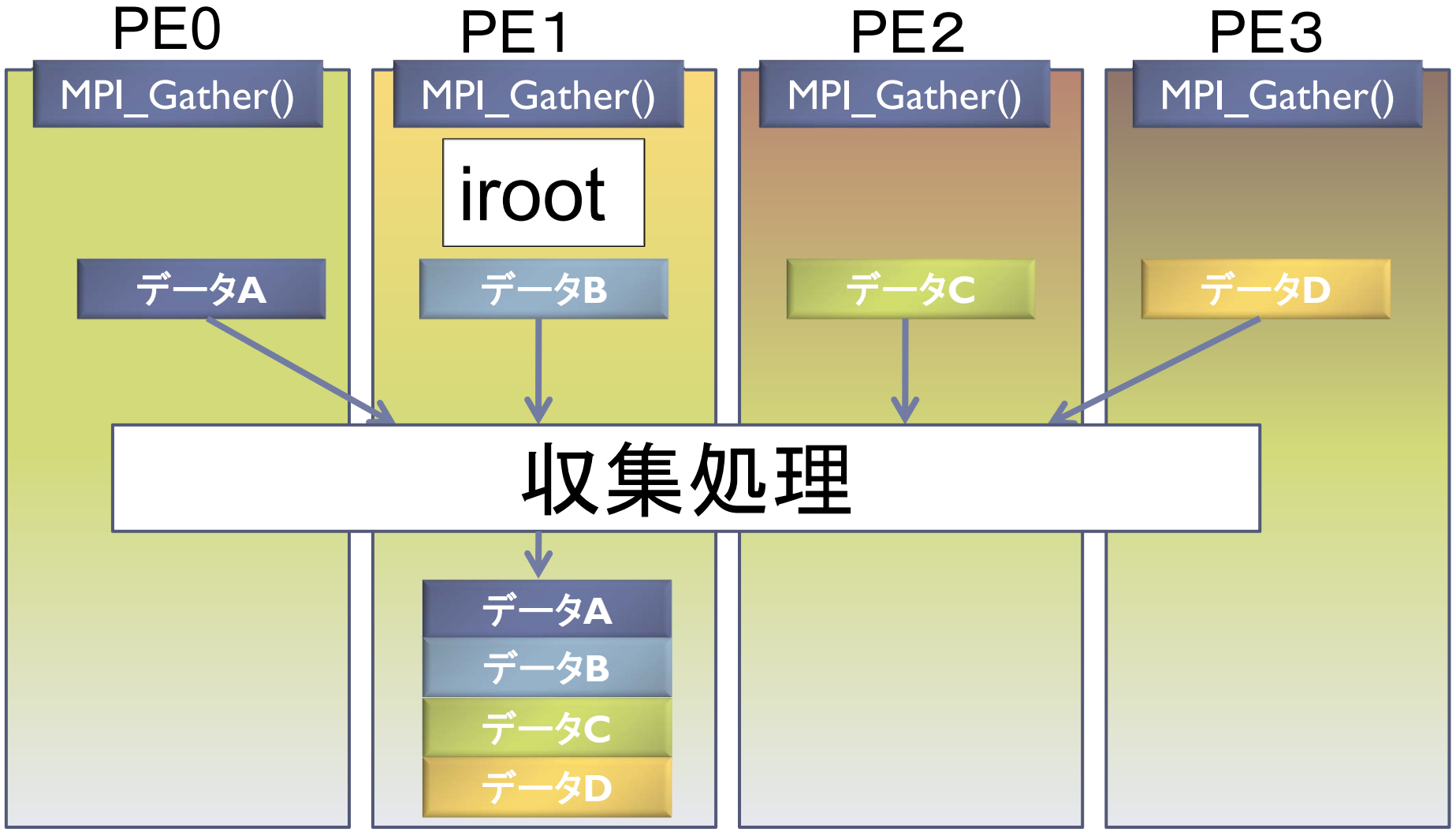
基礎的なMPI関数—MPI_Gather

- ierr = MPI_Gather(sendbuf, isendcount, isendtype,
  recvbuf, irecvcount, irecvtype, iroot, icomm);
  - sendbuf : 送信領域の先頭番地を指定する。
  - isendcount: 整数型。送信領域のデータ要素数を指定する。
  - isendtype : 整数型。送信領域のデータの型を指定する。
  - recvbuf : 受信領域の先頭番地を指定する。iroot で指定したPEのみで書き込みがなされる。
    - なお原則として、送信領域と受信領域は、同一であってはならない。すなわち、異なる配列を確保しなくてはならない。
  - irecvcount: 整数型。受信領域のデータ要素数を指定する。
    - この要素数は、1PE当たりの送信データ数を指定すること。
    - MPI_Gather 関数では各PEで異なる数のデータを収集することはできないので、同じ値を指定すること。

基礎的なMPI関数—MPI_Gather

- irecvtype： 整数型。受信領域のデータ型を指定する。
- iroot： 整数型。収集データを受け取るPEのicomm 内でのランクを指定する。
  - 全てのicomm 内のPEで同じ値を指定する必要がある。
- icomm： 整数型。PE集団を認識する番号であるコミュニケータを指定する。
- ierr： 整数型。エラーコードが入る。

MPI_Gatherの概念（集団通信）



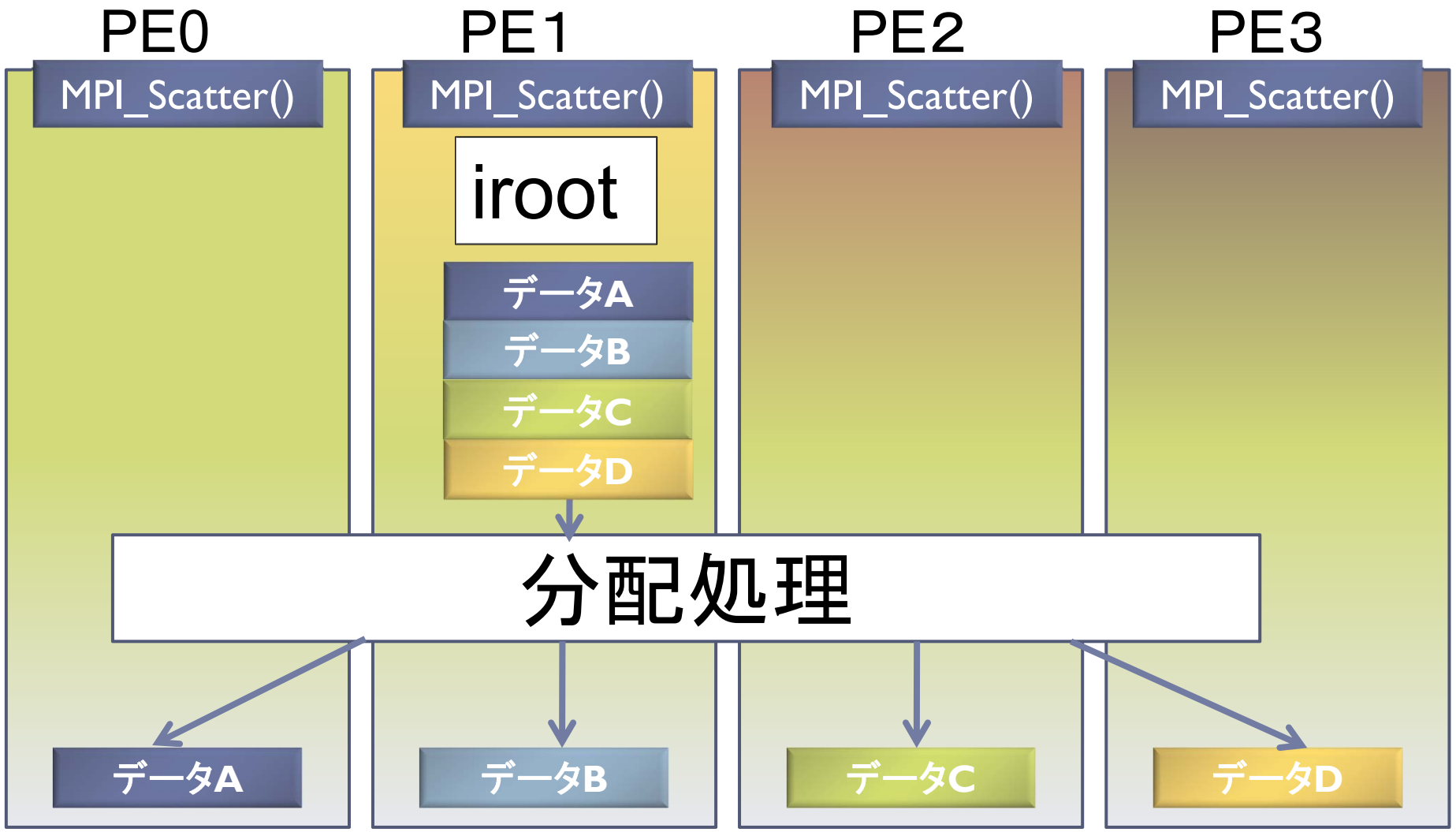
基礎的なMPI関数—MPI_Scatter

- ierr = MPI_Scatter (sendbuf, isendcount, isendtype, recvbuf, irecvcount, irecvtype, iroot, icomm);
  - sendbuf : 送信領域の先頭番地を指定する。
  - isendcount: 整数型。送信領域のデータ要素数を指定する。
    - この要素数は、1PE当たりに送られる送信データ数を指定すること。
    - MPI_Scatter 関数では各PEで異なる数のデータを分散することはできないので、同じ値を指定すること 。
  - isendtype : 整数型。送信領域のデータの型を指定する。 iroot で指定したPEのみ有効となる。
  - recvbuf : 受信領域の先頭番地を指定する。
    - なお原則として、送信領域と受信領域は、同一であってはならない。すなわち、異なる配列を確保しなくてはならない。
  - irecvcount: 整数型。受信領域のデータ要素数を指定する。

基礎的なMPI関数—MPI_Scatter

- irecvtype： 整数型。受信領域のデータ型を指定する。
- iroot： 整数型。収集データを受け取るPEのicomm 内でのランクを指定する。
  - 全てのicomm 内のPEで同じ値を指定する必要がある。
- icomm： 整数型。PE集団を認識する番号であるコミュニケータを指定する。
- ierr： 整数型。エラーコードが入る。

MPI_Scatterの概念（集団通信）



参考文献

1. MPI並列プログラミング、P.パチェコ 著 ／ 秋葉 博 訳
2. 並列プログラミング虎の巻MPI版、青山幸也 著、
   理化学研究所情報基盤センタ
   （ http://accc.riken.jp/HPC/training/text.html ）
3. Message Passing Interface Forum
   （ http://www.mpi-forum.org/ ）
4. MPI-Jメーリングリスト
   （ http://phase.hpcc.jp/phase/mpi-j/ml/ ）
5. 並列コンピュータ工学、富田眞治著、昭晃堂（1996）

MPIプログラム実習Ⅰ（演習）

東京大学情報基盤センター　准教授　片桐孝洋

実習課題

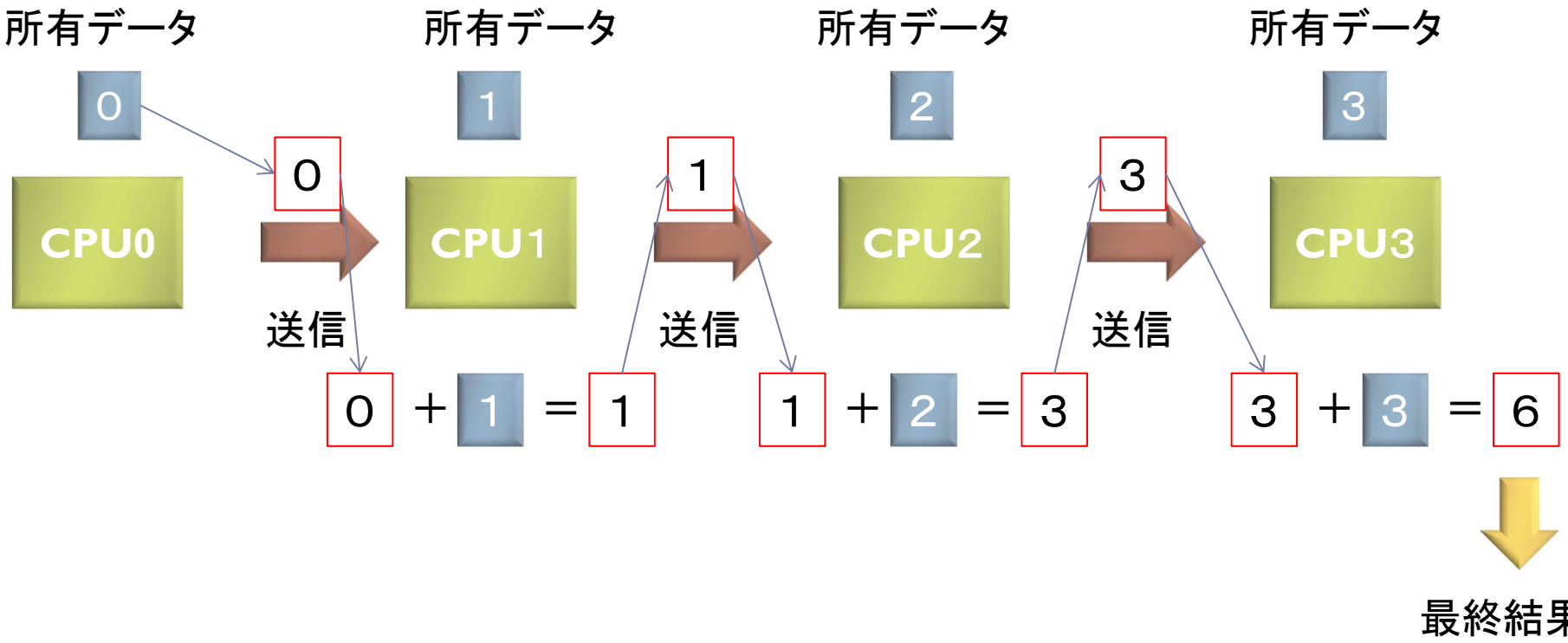
サンプルプログラムの説明

- Hello/
  - 並列版Helloプログラム
  - hello-pure.bash, hello-hy16.bash : NQSジョブスクリプトファイル
- Cpi/
  - 円周率計算プログラム
  - cpi-pure.bash NQSジョブスクリプトファイル
- Wa1/
  - 逐次転送方式による総和演算
  - wa1-pure.bash NQSジョブスクリプトファイル
- Wa2/
  - 二分木通信方式による総和演算
  - wa2-pure.bash NQSジョブスクリプトファイル
- Cpi_m/
  - 円周率計算プログラムに時間計測ルーチンを追加したもの
  - cpi_m-pure.bash NQSジョブスクリプトファイル

総和演算プログラム（逐次転送方式）

- 各プロセスが所有するデータを、全プロセスで加算し、
  あるプロセス1つが結果を所有する演算を考える。
- 素朴な方法（逐次転送方式）
  1. （0番でなければ）左隣のプロセスからデータを受信する；
  2. 左隣のプロセスからデータが来ていたら；
     1. 受信する;
     2. ＜自分のデータ＞と＜受信データ＞を加算する;
     3. （191番でなければ）右隣のプロセスに＜2の加算した結果を＞送信する;
     4. 処理を終了する;
- 実装上の注意
  - 左隣りとは、（myid-1）のIDをもつプロセス
  - 右隣りとは、（myid+1）のIDをもつプロセス
    - myid=0のプロセスは、左隣りはないので、受信しない
    - myid=p-1のプロセスは、右隣りはないので、送信しない

逐次転送方式（バケツリレー方式）による加算



１対１通信利用例（逐次転送方式、C言語）

```
void main(int argc, char* argv[]) {
  MPI_Status istatus;
....
  dsendbuf = myid;
  drecvbuf = 0.0;
  if (myid != 0) {
     ierr = MPI_Recv(&drecvbuf, 1, MPI_DOUBLE, myid-1, 0,
                  MPI_COMM_WORLD, &istatus);
  }
  dsendbuf = dsendbuf + drecvbuf;
  if (myid != nprocs-1) {
    ierr = MPI_Send(&dsendbuf, 1, MPI_DOUBLE, myid+1, 0,
                MPI_COMM_WORLD);
  }
  if (myid == nprocs-1) printf ("Total = %4.2lf ¥n", dsendbuf);
....
}
```

受信用システム配列の確保

自分より一つ少ないID番号（myid-1）から、double型データ1つを受信しdrecvbuf変数に代入

自分より一つ多いID番号（myid+1）に、dsendbuf変数に入っているdouble型データ1つを送信

1対1通信利用例（逐次転送方式、Fortran言語）

```
      program main
      integer  istatus(MPI_STATUS_SIZE)
      ....
      dsendbuf = myid
      drecvbuf = 0.0
      if (myid .ne. 0) then
        call MPI_RECV(drecvbuf, 1, MPI_DOUBLE_PRECISION,
     &              myid-1, 0, MPI_COMM_WORLD, istatus, ierr)
      endif
      dsendbuf = dsendbuf + drecvbuf
      if (myid .ne. numprocs-1) then
        call MPI_SEND(dsendbuf, 1, MPI_DOUBLE_PRECISION,
     &          myid+1, 0, MPI_COMM_WORLD, ierr)
      endif
     if (myid .eq. numprocs-1) then
        print *, "Total = ", dsendbuf
      endif
      ....
      stop
      end
```

受信用システム配列の確保

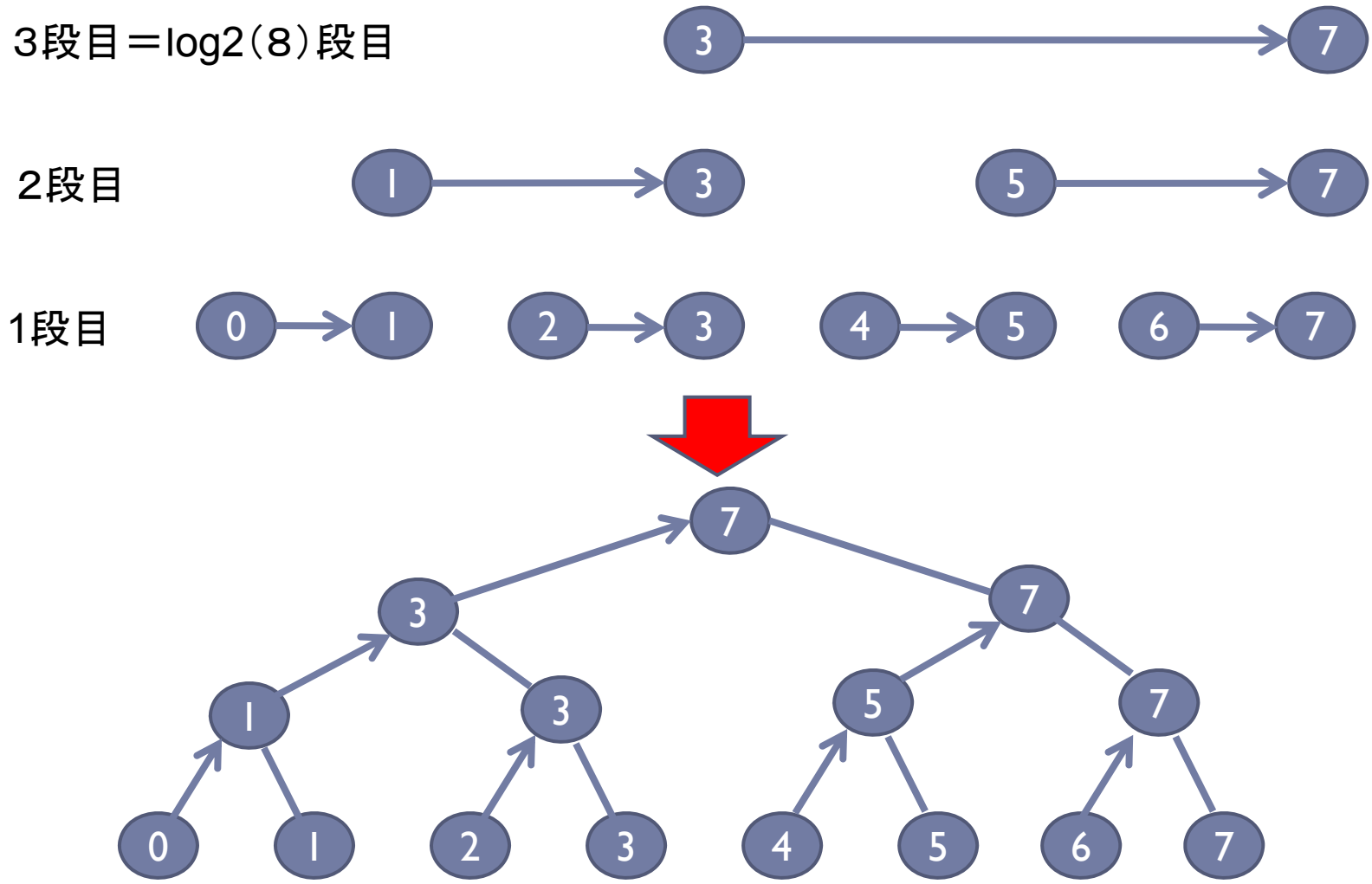
自分より一つ少ないID番号（myid-1）から、double型データ1つを受信しdrecvbuf変数に代入

自分より一つ多いID番号（myid+1）に、dsendbuf変数に入っているdouble型データ1つを送信

総和演算プログラム（二分木通信方式）

- 二分木通信方式
  1. k = 1;
  2. for (i=0; i < log2(nprocs); i++)
  3. if ((myid & k) == k)
     - (myid – k)番プロセスからデータを受信;
     - 自分のデータと、受信データを加算する;
     - k = k * 2;
  4. else
     - (myid + k)番プロセスに、データを転送する;
     - 処理を終了する;

総和演算プログラム（二分木通信方式）



総和演算プログラム（二分木通信方式）

- 実装上の工夫
  - 要点：プロセス番号の2進数表記の情報を利用する
  - 第i段において、受信するプロセスの条件は、以下で書ける：

    myid & k が k と一致
    - ここで、k = 2^(i-1) 。
    - つまり、プロセス番号の2進数表記で右からi番目のビットが立っているプロセスが、送信することにする
  - また、送信元のプロセス番号は、以下で書ける：

    myid + k
    - つまり 、通信が成立するプロセス番号の間隔は2^(i-1)←二分木なので
  - 送信プロセスについては、上記の逆が成り立つ。

総和演算プログラム（二分木通信方式）

- 逐次転送方式の通信回数
  - 明らかに、nprocs－1 回
- 二分木通信方式の通信回数
  - 見積もりの前提
    - 各段で行われる通信は、完全に並列で行われる
      （通信の衝突は発生しない）
  - 段数の分の通信回数となる
  - つまり、log2(nprocs) 回
- 両者の通信回数の比較
  - プロセッサ台数が増すと、通信回数の差（＝実行時間）が
    とても大きくなる
  - 1024プロセス構成では、1023回　対　10回！
  - でも、必ずしも二分木通信方式がよいとは限らない（通信衝突の多発）

性能プロファイラ

- 富士通コンパイラには、性能プロファイラ機能がある
- 富士通コンパイラでコンパイル後、実行コマンドで指定し利用する
- 以下の2種類があります
- 基本プロファイラ
  - 主な用途：プログラム全体で、最も時間のかかっている関数を同定する
- 詳細プロファイラ
  - 主な用途：最も時間のかかっている関数内の特定部分において、メモリアクセス効率、キャッシュヒット率、スレッド実行効率、MPI通信頻度解析、を行う

性能プロファイラの種類の詳細

- 基本プロファイラ
  - コマンド例：fipp –C
  - 表示コマンド：fipppx、GUI（WEB経由）
  - ユーザプログラムに対し一定間隔(デフォルト時10 0 ミリ秒間隔)毎に割り込みをかけ情報を収集する。
  - 収集した情報を基に、コスト情報等の分析結果を表示。
- 詳細プロファイラ
  - コマンド例：fapp –C
  - 表示コマンド：GUI(WEB経由)
  - ユーザプログラムの中に測定範囲を設定し、測定範囲のハードウェアカウンタの値を収集。
  - 収集した情報を基に、MFLOPS、MIPS、各種命令比率、キャッシュミス等の詳細な分析結果を表示。

基本プロファイラ利用例

- プロファイラデータ用の空のディレクトリがないとダメ
- /Wa2 に Profディレクトリを作成
  $ mkdir Prof
- Wa2 の wa2-pure.bash 中に以下を記載
  fipp -C -d Prof mpirun ./wa2
- 実行する
  $ pjsub wa2-pure.bash
- テキストプロファイラを起動
  $ fipppx –A -d Prof

基本プロファイラ出力例（1/2）

```
----------------------------------------------------------------------------------------
Fujitsu Instant Profiler Version 1.2.0
  Measured time             : Thu Apr 19 09:32:18 2012
  CPU frequency             : Process     0 -    127  1848 (MHz)
  Type of program           : MPI
  Average at sampling interval  : 100.0 (ms)
  Measured range            : All ranges
  Virtual coordinate        : (12, 0, 0)
----------------------------------------------------------------------------------------
_____________________________________________________________________________________

Time statistics
      Elapsed(s)      User(s)     System(s)
  ---------------------------------------------
        2.1684       53.9800       87.0800   Application
  ---------------------------------------------
        2.1684        0.5100        0.6400   Process     11
        2.1588        0.4600        0.6800   Process     88
        2.1580        0.5000        0.6400   Process     99
        2.1568        0.6600        1.4200   Process    111
...
```

基本プロファイラ出力例（2/2）

```
Procedures profile
  ********************************************************************************************
     Application - procedures
  ********************************************************************************************
          Cost       %         Mpi       %   Start     End
  ----------------------------------------------------------------
           475  100.0000         312   65.6842      --      --  Application
  ----------------------------------------------------------------
           312   65.6842         312  100.0000       1      45  MAIN__
            82   17.2632           0    0.0000      --      --  __GI___sched_yield
            80   16.8421           0    0.0000      --      --  __libc_poll
             1    0.2105           0    0.0000      --      --  __pthread_mutex_unlock_usercnt
  ********************************************************************************************
     Process    11 - procedures
  ********************************************************************************************
          Cost       %         Mpi       %   Start     End
  ----------------------------------------------------------------
             5  100.0000           4   80.0000      --      --  Process    11
  ----------------------------------------------------------------
             4   80.0000           4  100.0000       1      45  MAIN__
             1   20.0000           0    0.0000      --      --  __GI___sched_yeld
....
```

詳細プロファイラ利用例

- 測定したい対象に、以下のコマンドを挿入
- Fortran言語の場合
  - ヘッダファイル：なし
  - 測定開始　手続き名： call fapp_start(name, number, level)
  - 測定終了　手続き名： call fapp_stop(name, number, level)
  - 利用例： call fapp_start(“region1”,1,1)
- C/C++言語の場合
  - ヘッダファイル： fj_tool/fapp.h
  - 測定開始　関数名： void fapp_start(const char *name, int number, int level)
  - 測定終了　関数名： void fapp_stop(const char *name, int number, int level)
  - 利用例： fapp_start(“region1”,1,1);

詳細プロファイラ利用例

- 空のディレクトリがないとダメなので、/Wa2 に Profディレクトリを作成
  $ mkdir Prof
- Wa2のwa2-pure.bash中に以下を記載
  (キャッシュ情報取得時)
  fapp -C -d Prof –L 1 –lhwm –Hevent=Cache mpirun ./wa2
- 実行する
  $ pjsub wa2-pure.bash

詳細プロファイラGUIによる表示例

- プログラミング支援ツール(FUJITSU Software Development Tools Version 1.2.1 for Windows) をインストール
  - 以下をアクセス

https://oakleaf-fx-1.cc.u-tokyo.ac.jp/fsdtfx10tx/install/index.html

- 「ダウンロード」をクリック
- Serverに、oakleaf-fx-1.cc.u-tokyo.ac.jp
- Nameと passwordはセンターから配布したものを入れる
- うまくいくと、右のボックスがでる

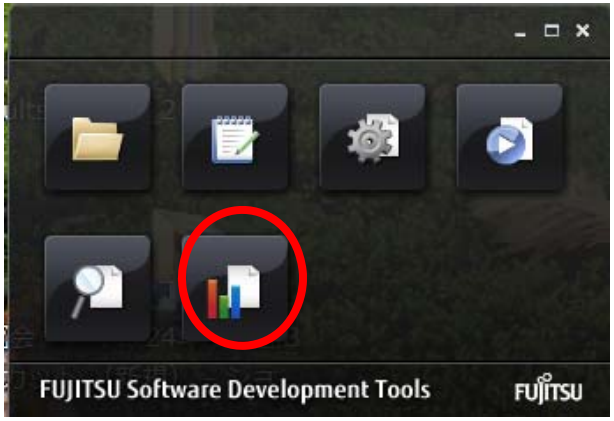
詳細プロファイラGUIによる表示例

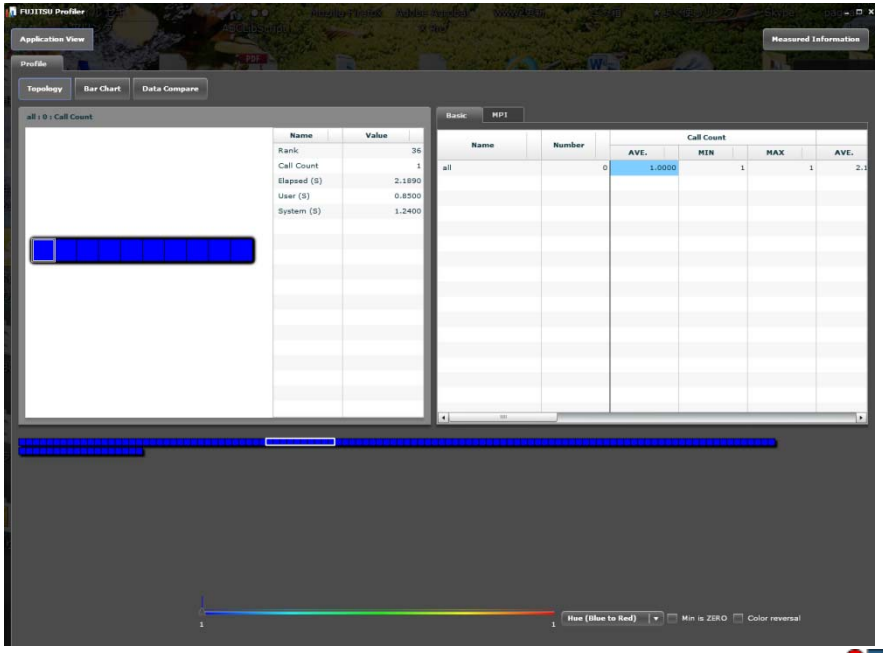
- MACの場合は、以下をアクセス

https://oakleaf-fx-1.cc.u-tokyo.ac.jp/fsdtfx10tx/install/mac/index.html

詳細プロファイラGUIによる表示例

- 右のボックスで、プロファイラ部分をクリック
- プロファイルデータがあるフォルダを指定する
- うまくいくと、右のような解析データが見える





詳細プロファイラで取れるデータ

- プロセス間の通信頻度情報
  （GUI上で色で表示）
- 各MPIプロセスにおける以下の情報
  - Cache: キャッシュミス率
  - Instructions: 実行命令詳細
  - Mem_access: メモリアクセス状況
  - Performance: 命令実行効率
  - Statistics: CPU core 動作状況

精密ＰＡ可視化機能（エクセル形式）

- 性能プロファイルは見にくい
- ７回程度実行しないといけないが、
  性能プロファイルデータ（マシン語命令の種類や、実行時間に占める割合など）を、Excelで可視化してくれるツール（精密PA可視化ツール）が、東京大学情報基盤センターのFX10では、提供されている
- 手順
  1. 対象箇所（ループ）を、専用のAPIで指定する
  2. プロファイルを入れるフォルダ7か所をつくる
  3. プロファイルのためのコマンドで7回実行する
  4. エクセル形式に変換する
  5. 4のエクセル形式を手元のパソコンに持ってくる
  6. 5のファイルを、指定のエクセルと同一のフォルダに入れてから、指定のエクセルを開く

精密ＰＡ可視化のための指示ＡＰＩ

- 以下のAPIで、対象となるループを挟む（Fortranの場合）

```
call start_collection(“region”)
    <対象となるループ>
call stop_collection(“region”)
```

- “region”は、対象となる場所の名前なので、任意の名前を付けることが可能
（後で、専用エクセルを開くときに使う）

実行のさせ方

- a.out という実行ファイルの場合以下のように7回実行する
- 実行するディレクトリに、pa1、pa2、…、pa7という、ディレクトリを作っておく必要がある
- 以下のように実行する

```
fapp -C –d pa1 -Hpa=1 mpiexec a.out
fapp -C –d pa2 -Hpa=2 mpiexec a.out
fapp -C –d pa3 -Hpa=3 mpiexec a.out
fapp -C –d pa4 -Hpa=4 mpiexec a.out
fapp -C –d pa5 -Hpa=5 mpiexec a.out
fapp -C –d pa6 -Hpa=6 mpiexec a.out
fapp -C –d pa7 -Hpa=7 mpiexec a.out
```

エクセルデータへの変換

- pa1、pa2、…、pa7の中に、プロファイルデータがあることを確認する
- 以下のコマンドを実行する

```
fapppx -A -d pa1 -o output_prof_1.csv -tcsv -Hpa
fapppx -A -d pa2 -o output_prof_2.csv -tcsv -Hpa
fapppx -A -d pa3 -o output_prof_3.csv -tcsv -Hpa
fapppx -A -d pa4 -o output_prof_4.csv -tcsv -Hpa
fapppx -A -d pa5 -o output_prof_5.csv -tcsv -Hpa
fapppx -A -d pa6 -o output_prof_6.csv -tcsv -Hpa
fapppx -A -d pa7 -o output_prof_7.csv -tcsv -Hpa
```

エクセルデータを手元のＰＣに転送

- 以下のFX10上のエクセルデータを、手元のPCに転送
  - output_prof_1.csv、output_prof_2.csv、…、output_prof_7.csv
- 上記のエクセルデータが入ったフォルダで、ポータル上で公開されている専用エクセルを開く
- 詳細なエクセルデータの利用法、分析されたデータの見方は、マニュアル参照
  - J2UL-1490-04Z0(01)　2013年6月、Technical Computing Suite V1.0、プロファイラ使用手引書(PRIMEHPC FX10用)

性能最適化情報の提示（1/2）

- C、Fortranを問わず、コンパイラが行った最適化情報を知ることは性能最適化で重要である
- 各ファイルのソースコードごとに、どのような最適化が行われたか出力する、コンパイラオプションがある
- C、Fortranで共通の翻訳情報
  - -Nlst=p：標準の最適化情報を出力(デフォルト)
  - -Nlst=t：詳細な最適化情報を出力

性能最適化情報の提示（2/2）

- Fortran のみ、p、t以外も指定可能
  - -Nlst=a ：名前の属性情報を出力
  - -Nlst=d ：派生型の構成情報を出力
  - -Nlst=I ：インクルードされたファイルのプログラムリストおよびインクルードファイル名一覧を出力を出力
  - -Nlst=m ：自動並列化の状況をOpenMP指示文によって表現した原始プログラムを出力
  - -Nlst=x ：名前および文番号の相互参照情報を出力
  - 詳細は、オンラインマニュアルの以下を参照のこと
    - C言語使用手引書： P.26
    - C++言語使用手引書： P.28
    - Fortran使用手引書： P.46 P.52 P.53

演習課題

1. 逐次転送方式のプログラムを実行
   - Wa1 のプログラム
2. 二分木通信方式のプログラムを実行
   - Wa2のプログラム
3. 時間計測プログラムを実行
   - Cpi_mのプログラム
4. プロセス数を変化させて、サンプルプログラムを実行
5. Helloプログラムを、以下のように改良
   - MPI_Sendを用いて、プロセス0からChar型のデータ“Hello World!!”を、その他のプロセスに送信する
   - その他のプロセスでは、MPI_Recvで受信して表示する

MPIプログラミング実習Ⅱ（演習）

東京大学情報基盤センター　准教授　片桐孝洋

講義の流れ

1. 行列-行列とは（30分）
2. 行列-行列積のサンプルプログラムの実行
3. サンプルプログラムの説明
4. 演習課題（1）：簡単なもの

行列 - 行列積の演習の流れ

- 演習課題（Ⅱ）
  - 簡単なもの（30分程度で並列化）
  - 通信関数が一切不要
- 演習課題（Ⅲ）
  - ちょっと難しい（1時間以上で並列化）
  - 1対1通信関数が必要
  - 演習課題（Ⅰ）が早く終わってしまった方は、やってみてください。

行列-行列積とは

実装により性能向上が見込める基本演算

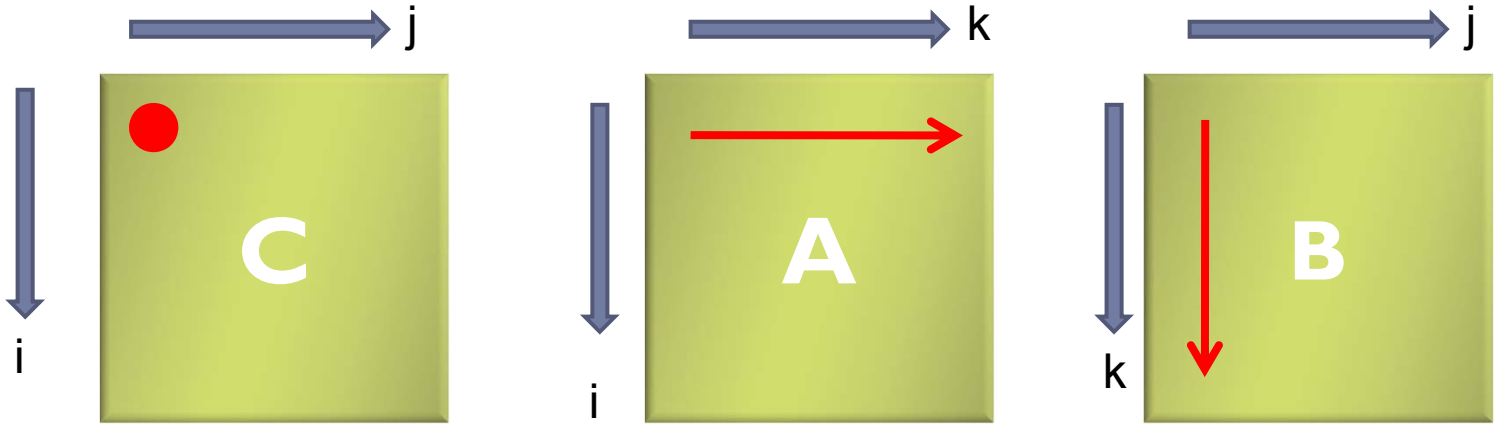
1.5　行列の積

- 行列積　C＝A・B は、コンパイラや計算機のベンチマークに使われることが多い
  - 理由1：実装方式の違いで性能に大きな差がでる
  - 理由2：手ごろな問題である（プログラムし易い）
  - 理由3：科学技術計算の特徴がよく出ている
    1. 非常に長い＜連続アクセス＞がある
    2. キャッシュに乗り切らない＜大規模なデータ＞に対する演算である
    3. メモリバンド幅を食う演算（メモリ・インテンシブ）な処理である

行列積コード例（C言語）

- コード例

```
for (i=0; i<n;  i++)
  for (j=0; j<n; j++)
    for (k=0; k<n; k++)
      C[i][j] += A[i][k] *B[k][j];
```



1.5　行列の積

- 行列積　$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \ (i, j = 1, 2, \ldots, n)$

  の実装法は、次の二通りが知られている:

1. ループ交換法
   - 連続アクセスの方向を変える目的で、行列-行列積を実現する3重ループの順番を交換する
2. ブロック化（タイリング）法
   - キャッシュにあるデータを再利用する目的で、あるまとまった行列の部分データを、何度もアクセスするように実装する

1.5　行列の積

- ループ交換法
  - 行列積のコードは、以下のような3重ループになる（C言語）

```
for(i=0; i<n; i++) {
  for(j=0; j<n; j++) {
    for(k=0; k<n; k++) {
      c[ i ][ j ] = c[ i ][ j ] + a[ i ][ k ] * b[ k][ j ];
    }
  }
}
```

  - 最内部の演算は、外側の3ループを交換しても、計算結果が変わらない
    → 6通りの実現の方法がある

1.5　行列の積

- ループ交換法
  - 行列積のコードは、以下のような3重ループになる（Fortran言語）

```
do i=1, n
  do j=1, n
    do k=1, n
      c( i , j ) = c( i, j) + a( i , k ) * b( k , j )
    enddo
  enddo
enddo
```

  - 最内部の演算は、外側の3ループを交換しても、計算結果が変わらない
    → 6通りの実現の方法がある

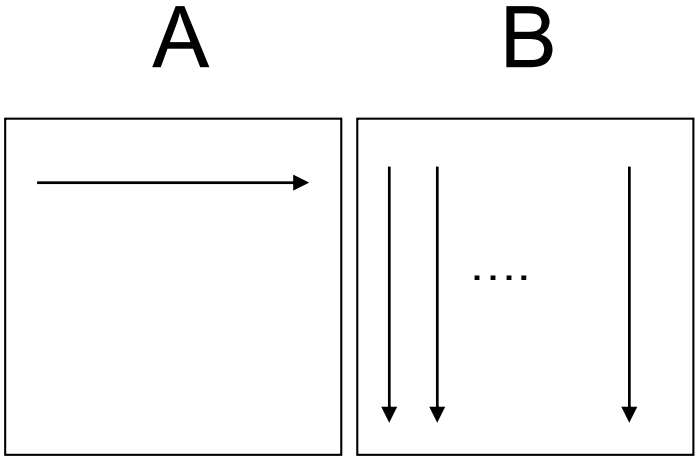
1.5　行列の積

- 行列データへのアクセスパターンから、
  以下の3種類に分類できる
  1. 内積形式 (inner-product form)
     最内ループのアクセスパタンが
     ＜ベクトルの内積＞と同等
  2. 外積形式 (outer-product form)
     最内ループのアクセスパタンが
     ＜ベクトルの外積＞と同等
  3. 中間積形式 (middle-product form)
     内積と外積の中間

1.5　行列の積

- 内積形式 (inner-product form)
  - ijk, jikループによる実現(C言語)

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    dc = 0.0;
    for (k=0; k<n; k++){
      dc = dc + A[ i ][ k ] * B[ k ][ j ];
    }
    C[ i ][ j ]= dc;
  }
}
```



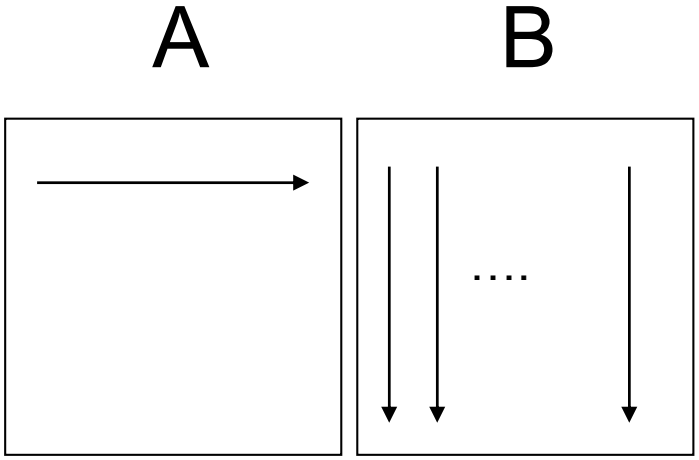
●行方向と列方向のアクセスあり
→行方向・列方向格納言語の
両方で性能低下要因
解決法:
A, Bどちらか一方を転置しておく

※以降、最外のループからの変数の順番で実装法を呼ぶ。たとえば上記のコードは<ijkループ>。

1.5　行列の積

- 内積形式 (inner-product form)
  - ijk, jikループによる実現(Fortran言語)

```
do i=1, n
  do j=1, n
    dc = 0.0d0
    do k=1, n
      dc = dc + A( i , k ) * B( k , j )
    enddo
    C( i , j ) = dc
  enddo
enddo
```



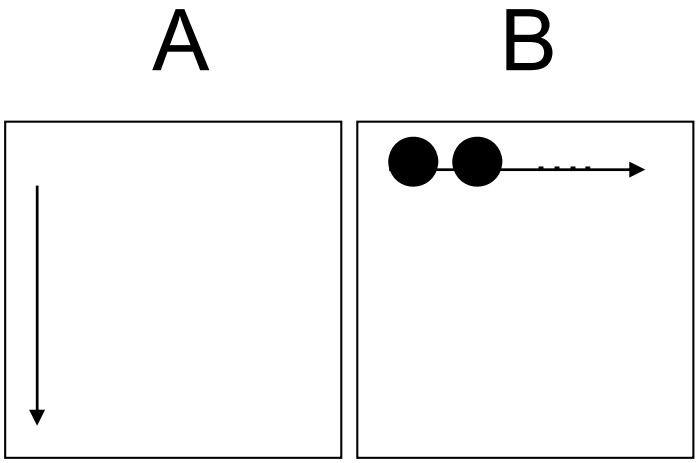
●行方向と列方向のアクセスあり
→行方向・列方向格納言語の
両方で性能低下要因
解決法:
A, Bどちらか一方を転置しておく

※以降、最外のループからの変数の順番で実装法を呼ぶ。たとえば上記のコードは<ijkループ>。

1.5　行列の積

- 外積形式 (outer-product form)
  - kij, kjiループによる実現（C言語）

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    C[ i ][ j ] = 0.0;
  }
}
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    db = B[ k ][ j ];
    for (i=0; i<n; i++) {
      C[ i ][ j ]= C[ i ][ j ]+ A[ i ][ k ]* db;
    }
  }
}
```

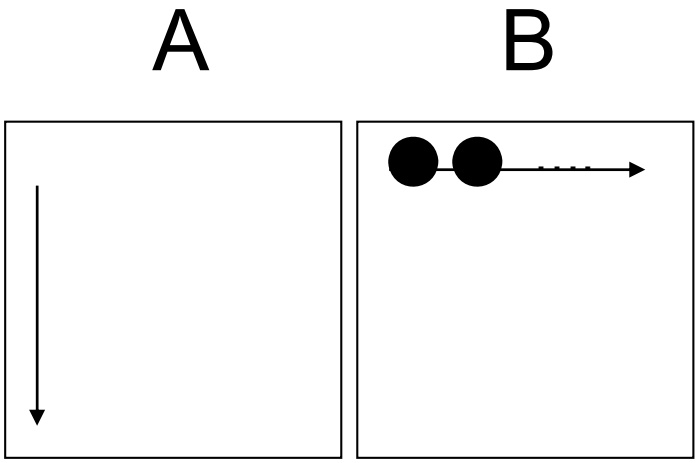


●kjiループでは
列方向アクセスがメイン
→列方向格納言語向き
（Fortran言語）

1.5　行列の積

- 外積形式 (outer-product form)
  - kij, kjiループによる実現（Fortran言語）

```
do i=1, n
  do j=1, n
    C( i , j ) = 0.0d0
  enddo
enddo
do k=1, n
  do j=1, n
    db = B( k , j )
    do i=1, n
      C( i , j ) = C( i , j )+ A( i , k ) * db
    enddo
  enddo
enddo
```

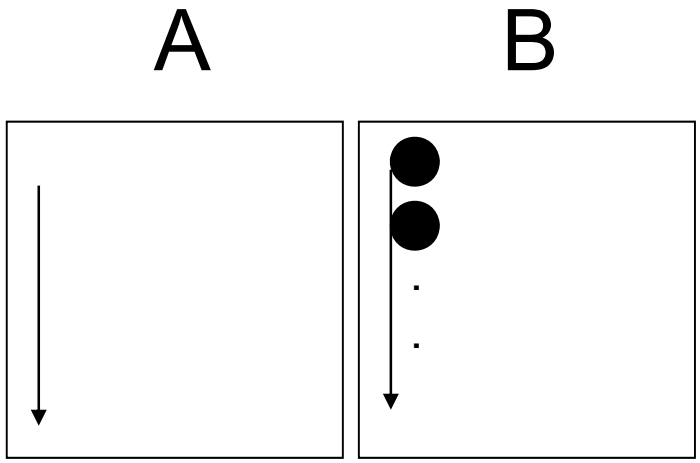


●kjiループでは
列方向アクセスがメイン
→列方向格納言語向き
（Fortran言語）

1.5　行列の積

- 中間積形式 (middle-product form)
  - ikj, jkiループによる実現（C言語）

```
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    C[ i ][ j ] = 0.0;
  }
  for (k=0; k<n; k++) {
    db = B[ k ][ j ];
    for (i=0; i<n; i++) {
      C[ i ][ j ] = C[ i ][ j ] + A[ i ][ k ] * db;
    }
  }
}
```

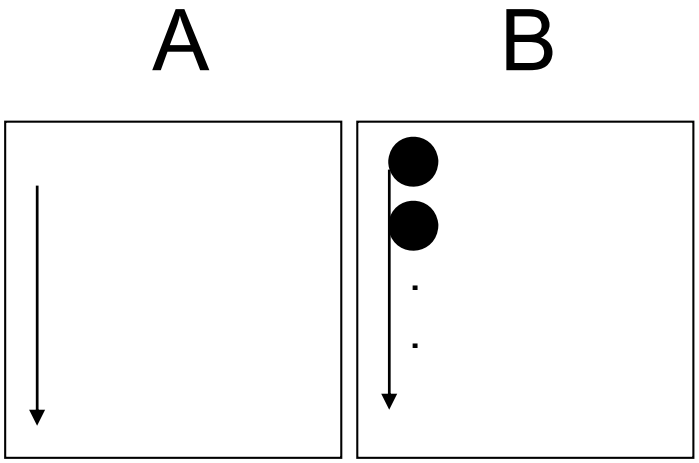


●jkiループでは
全て列方向アクセス
→列方向格納言語に
最も向いている
（Fortran言語）

1.5　行列の積

- 中間積形式 (middle-product form)
  - ikj, jkiループによる実現（Fortran言語）

```
do j=1, n
  do i=1, n
    C( i , j ) = 0.0d0
  enddo
  do k=1, n
    db = B( k , j )
    do i=1, n
      C( i , j ) = C( i , j ) + A( i , k ) * db
    enddo
  enddo
enddo
```

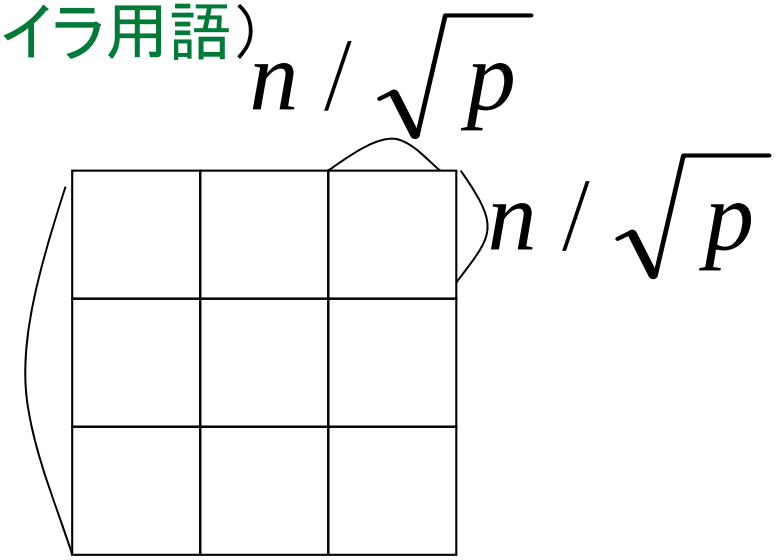


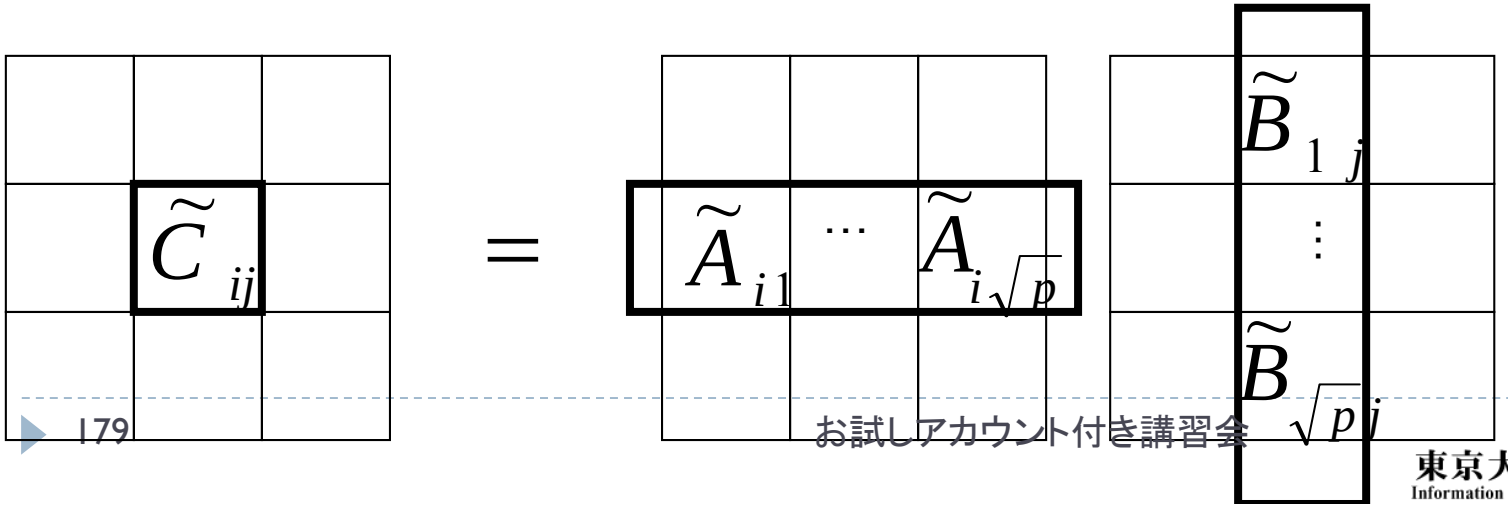
●jkiループでは
全て列方向アクセス
→列方向格納言語に
最も向いている
（Fortran言語）

1.5　行列の積

- 小行列ごとの計算に分けて（配列を用意し）計算
  （＜ブロック化＞, ＜タイリング＞:コンパイラ用語）
- 以下の計算

$$\tilde{C}_{ij} = \sum_{k=1}^{\sqrt{n}} \tilde{A}_{ik} \tilde{B}_{kj}$$





1.5　行列の積

- 各小行列をキャッシュに収まるサイズにする。
  1. ブロック単位で高速な演算が行える
  2. 並列アルゴリズムの変種が構築できる
- 並列行列積アルゴリズムは、データ通信の形態から、以下の2種に分類可能：
  1. セミ・シストリック方式
     - 行列A、Bの小行列の一部をデータ移動（Cannonのアルゴリズム）
  2. フル・シストリック方式
     - 行列A、Bの小行列のすべてをデータ移動（Foxのアルゴリズム）

サンプルプログラムの実行（行列-行列積）

行列-行列積のサンプルプログラムの注意点

- C言語版/Fortran言語版の共通ファイル名

  Mat-Mat-fx.tar

- ジョブスクリプトファイルmat-mat.bash　中のキュー名を

  lecture から tutorial に変更してから

  pjsub　してください。

  - lecture : 実習時間外のキュー
  - tutorial :実習時間内のキュー

行列-行列積のサンプルプログラムの実行

- 以下のコマンドを実行する

  $ cp /home/z30082/Mat-Mat-fx.tar ./

  $ tar xvf Mat-Mat-fx.tar

  $ cd Mat-Mat

- 以下のどちらかを実行

  $ cd C :C言語を使う人

  $ cd F :Fortran言語を使う人

- 以下共通

  $ make

  $ pjsub mat-mat.bash

- 実行が終了したら、以下を実行する

  $ cat mat-mat.bash.oXXXXXX

行列-行列積のサンプルプログラムの実行（C言語）

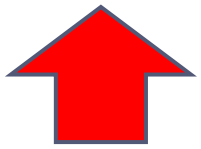
- 以下のような結果が見えれば成功

N = 1000

Mat-Mat time = 0.209609 [sec.]

9541.570931 [MFLOPS]

OK!



1コアのみで、9.5GFLOPSの性能

行列-行列積のサンプルプログラムの実行（Fortran言語）

- 以下のような結果が見えれば成功

NN = 1000

Mat-Mat time[sec.] = 0.2047346729959827

MFLOPS = 9768.741003580422

OK!

1コアのみで、9.7GFLOPSの性能

サンプルプログラムの説明

- #define N 1000
  の、数字を変更すると、行列サイズが変更できます
- #define DEBUG 0
  の「0」を「1」にすると、行列-行列積の演算結果が検証できます。
- MyMatMat関数の仕様
  - Double型N×N行列AとBの行列積をおこない、Double型N×N行列Cにその結果が入ります

Fortran言語のサンプルプログラムの注意

- 行列サイズNの宣言は、以下のファイルにあります。

  mat-mat.inc

- 行列サイズ変数が、NNとなっています。

```
integer  NN
parameter (NN=1000)
```

演習課題（１）

- MyMatMat関数を並列化してください。
  - #define　N　　192
  - #define　DEBUG　1

  として、デバッグをしてください。
- 行列A、B、Cは、各PEで重複して、かつ全部（N×N）所有してよいです。

演習課題（１）

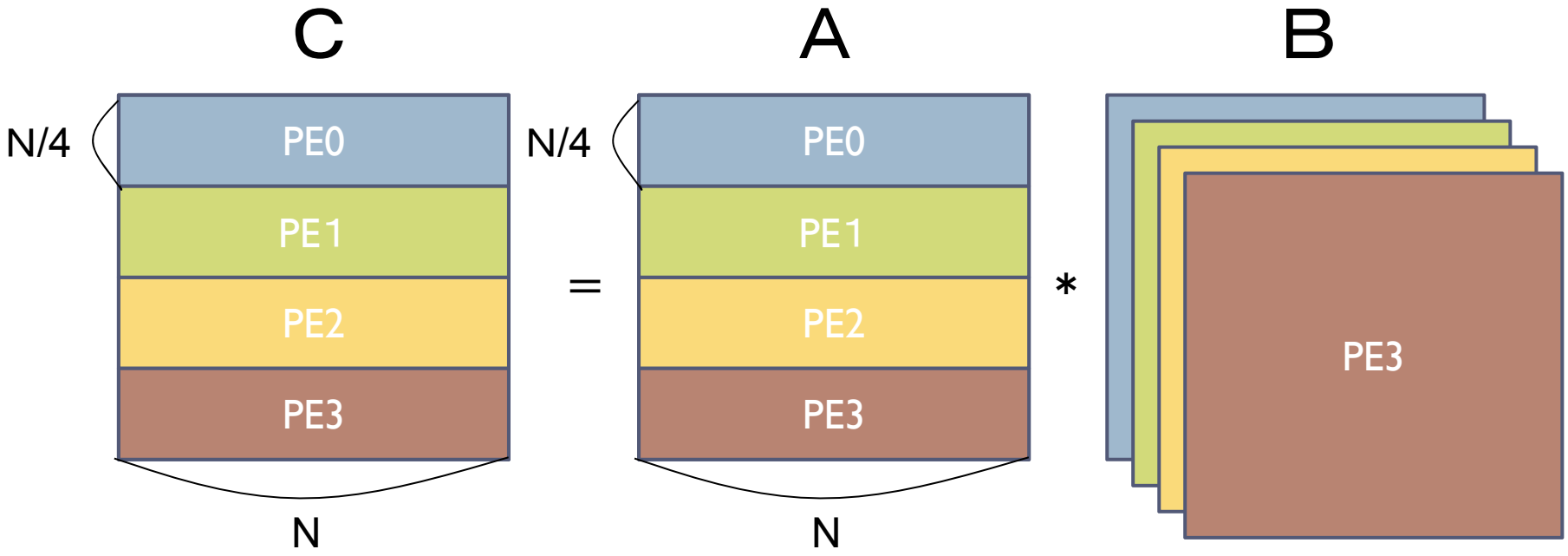
- サンプルプログラムでは、行列A、Bの要素を全部1として、行列-行列積の結果をもつ行列Cの全要素がNであるか調べ、結果検証しています。デバックに活用してください。
  - 行列Cの分散方式により、

演算結果チェックルーチンの並列化が必要

になります。注意してください。

並列化のヒント

- 以下のようなデータ分割にすると、とても簡単です。



- 通信関数は一切不要です。

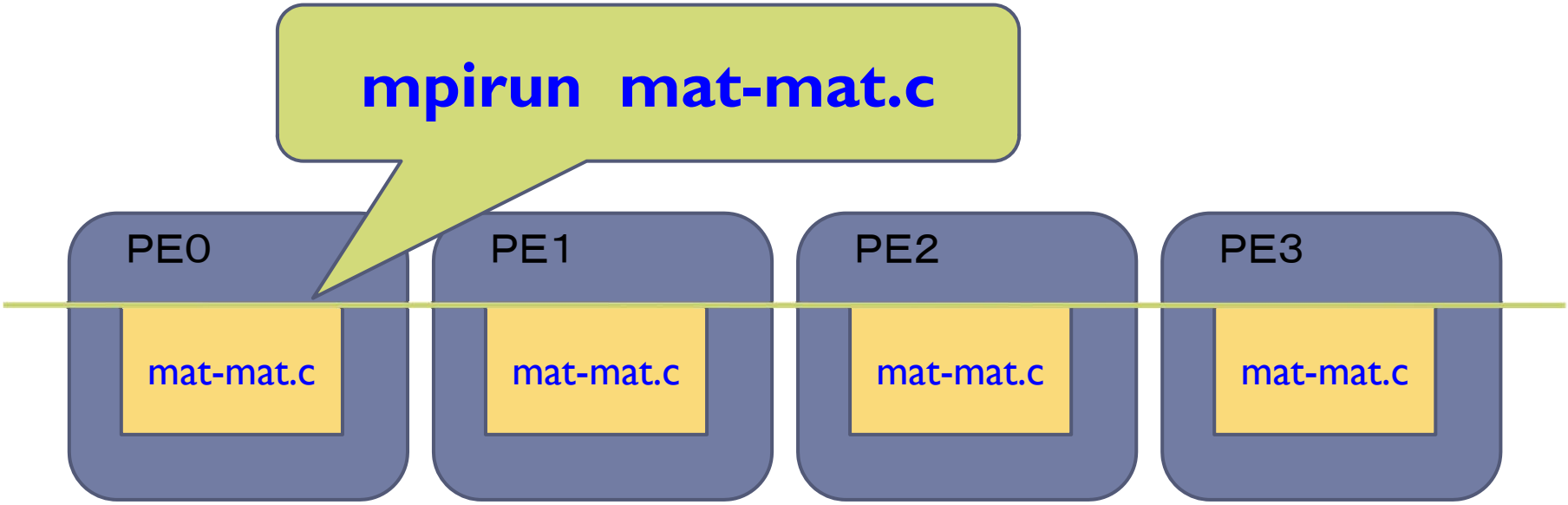
MPI並列化の大前提（再確認）

- SPMD
  - 対象のプログラム（mat-mat.c, mat-mat.f）は、
    - すべてのPEで、かつ、
    - 同時に起動された状態

    から処理が始まる。
- 分散メモリ型並列計算機
  - 各PEは、完全に独立したメモリを持っている。他のPEからは、通信なしには参照できない。（共有メモリではない）

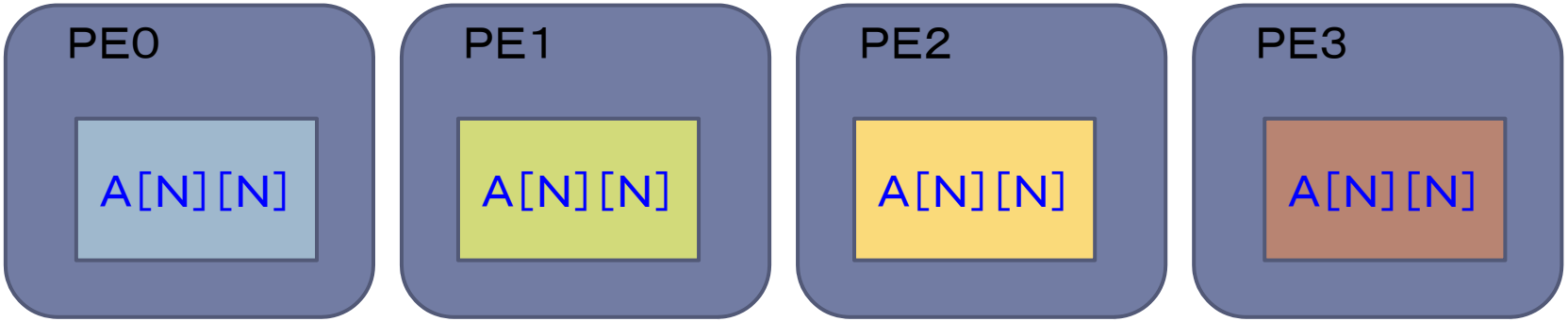
MPI並列化の大前提（再確認）

- 各PEでは、＜同じプログラムが同時に起動＞されて開始されます。

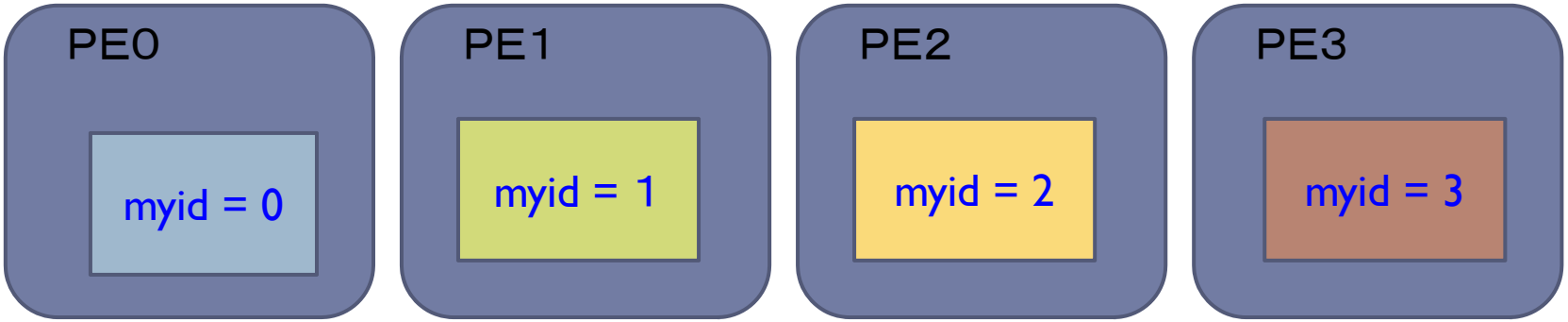


MPI並列化の大前提（再確認）

- 各PEでは、＜別配列が個別に確保＞されます。

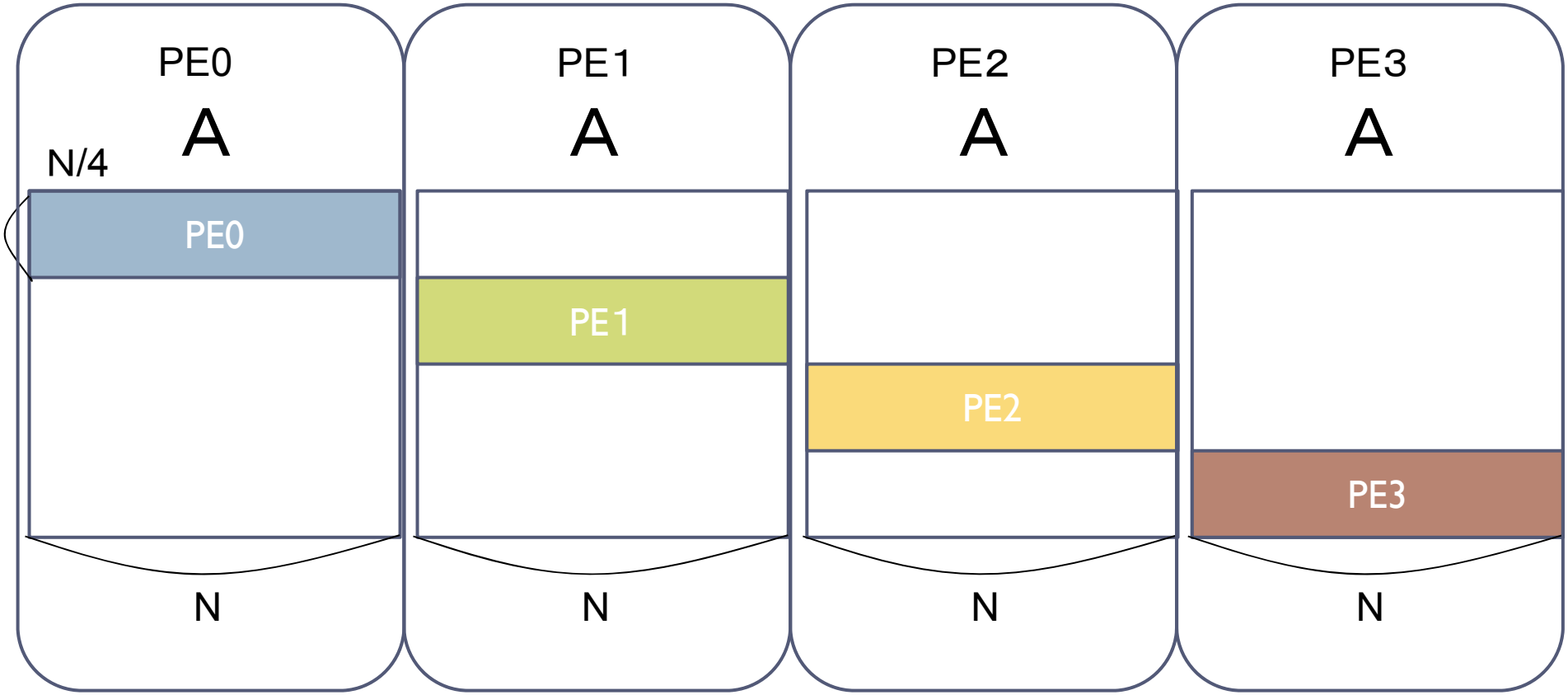


- myid変数は、MPI_Init()関数（もしくは、サブルーチン）が呼ばれた段階で、＜各PE固有の値＞になっています。



各PEでの配列の確保状況

- 実際は、以下のように配列が確保されていて、
  部分的に使うだけになります

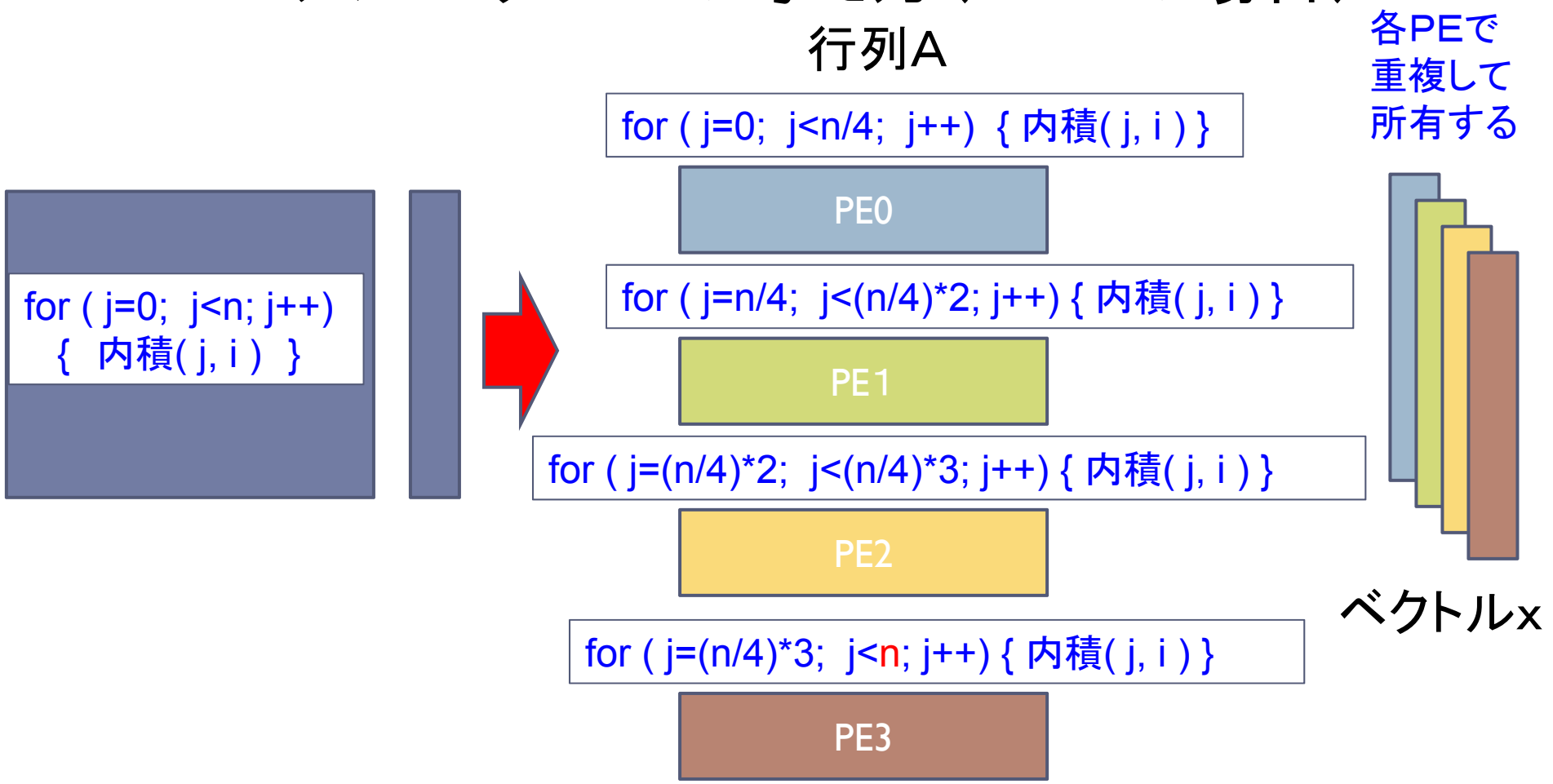


本実習プログラムのTIPS

- myid, numprocs は大域変数です
  - myid (=自分のID)、および、numprocs(=世の中のPE台数)の変数は大域変数です。MyMatVec関数内で、引数設定や宣言なしに、参照できます。
- myid, numprocs の変数を使う必要があります
  - MyMatMat関数を並列化するには、myid 、および、numprocs変数を利用しないと、並列化ができません。

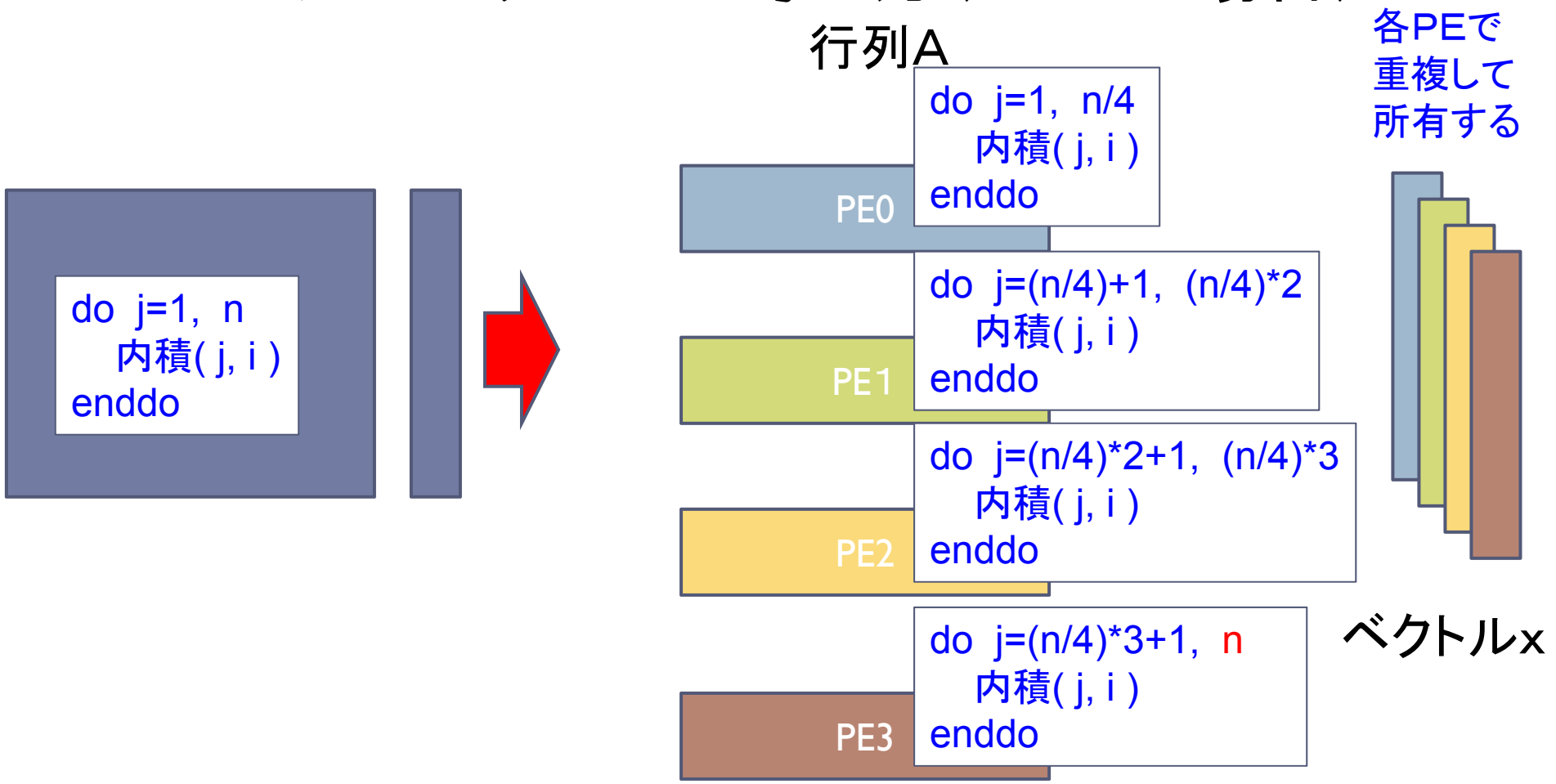
並列化の考え方 （行列-ベクトル積の場合、Ｃ言語）

- SIMDアルゴリズムの考え方（4PEの場合）



並列化の考え方（行列-ベクトル積の場合、Fortran言語）

- SIMDアルゴリズムの考え方（4PEの場合）



並列化の方針（C言語）

1. 全PEで行列AをN×Nの大きさ、ベクトルx、yをNの大きさ、確保してよいとする。
2. 各PEは、担当の範囲のみ計算するように、ループの開始値と終了値を変更する。
   - ブロック分散方式では、以下になる
     （n が numprocs で割り切れる場合）

```
ib = n / numprocs;
for ( j=myid*ib;  j<(myid+1)*ib;  j++)    { … }
```

3. （2の並列化が完全に終了したら）各PEで担当のデータ部分しか行列を確保しないように変更する。
   - 上記のループは、以下のようになる。

```
for ( j=0;  j<ib;  j++)  { … }
```

並列化の方針（Fortran言語）

1. 全PEで行列AをN×Nの大きさ、ベクトルx、yをNの大きさ、確保してよいとする。
2. 各PEは、担当の範囲のみ計算するように、ループの開始値と終了値を変更する。
   - ブロック分散方式では、以下になる
     （n が numprocs で割り切れる場合）

```
ib = n / numprocs
do j = myid*ib+1, (myid+1)*ib
    …..
enddo
```

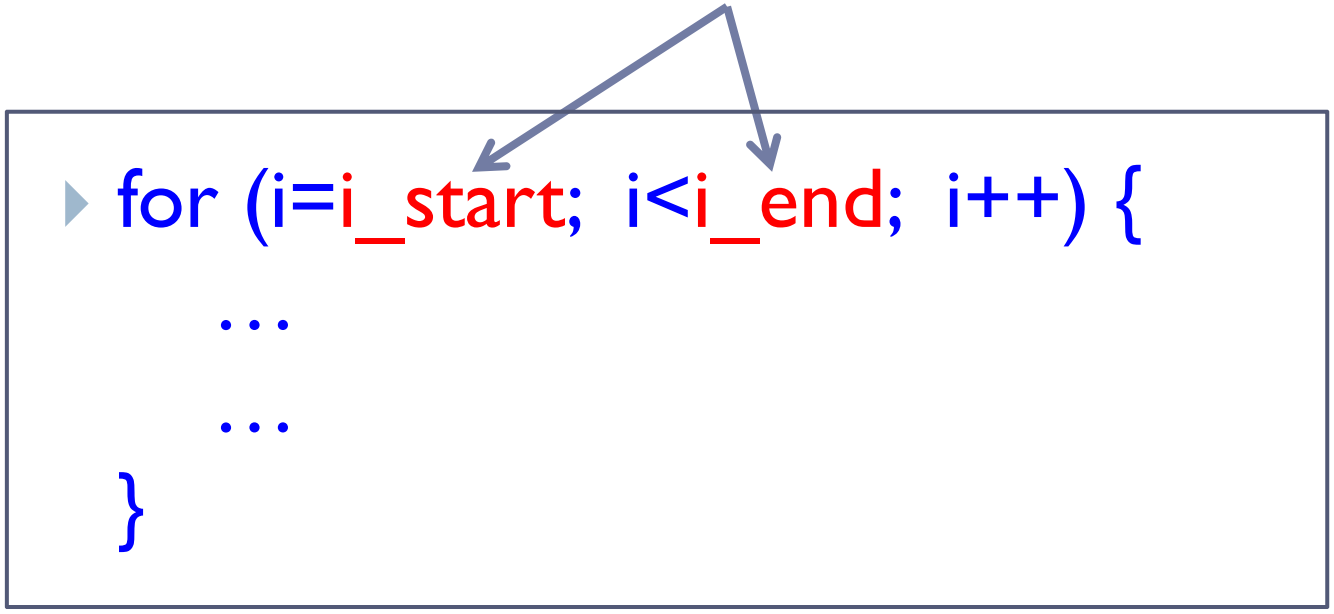
3. （2の並列化が完全に終了したら）各PEで担当のデータ部分しか行列を確保しないように変更する。
   - 上記のループは、以下のようになる。

```
do j=1, ib
    …
enddo
```

実装上の注意

- ループ変数をグローバル変数にすると、性能が出ません。
  必ずローカル変数か、定数（　2　など）にしてください。

ローカル変数にすること



MPIプログラミング実習(Ⅲ)(演習)

実習（Ⅱ）が早く終わってしまった方のための演習です

講義の流れ

1. 行列-行列積（2）のサンプルプログラムの実行
2. サンプルプログラムの説明
3. 演習課題（2）：ちょっと難しい完全分散版
4. 並列化のヒント

サンプルプログラムの実行（行列-行列積（２））

行列-行列積のサンプルプログラムの注意点

- C言語版/Fortran言語版のファイル名

  Mat-Mat-d-fx.tar

- ジョブスクリプトファイルmat-mat-d.bash　中のキュー名を

  lecture から　tutorial　に変更してから

  qsub　してください。
  - lecture : 実習時間外のキュー
  - tutorial: 実習時間内のキュー

行列-行列積(2)のサンプルプログラムの実行

- 以下のコマンドを実行する

  $ cp /home/z30082/Mat-Mat-d-fx.tar ./
  $ tar xvf Mat-Mat-d-fx.tar
  $ cd Mat-Mat-d

- 以下のどちらかを実行

  $ cd C :C言語を使う人
  $ cd F :Fortran言語を使う人

- 以下共通

  $ make
  $ pjsub mat-mat-d.bash

- 実行が終了したら、以下を実行する

  $ cat mat-mat-d.bash.oXXXXXX

行列-行列積のサンプルプログラムの実行（C言語版）

- 以下のような結果が見えれば成功

```
N  = 384
Mat-Mat time  = 0.000135 [sec.]
 841973.194818 [MFLOPS]
 Error! in ( 0 , 2 )-th argument in PE 0
 Error! in ( 0 , 2 )-th argument in PE 61
 Error! in ( 0 , 2 )-th argument in PE 51
 Error! in ( 0 , 2 )-th argument in PE 59
 Error! in ( 0 , 2 )-th argument in PE 50
 Error! in ( 0 , 2 )-th argument in PE 58
....
```

並列化が完成していないのでエラーが出ます。ですが、これは正しい動作です

行列-行列積のサンプルプログラムの実行（Fortran言語）

- 以下のような結果が見えれば成功

```
NN  =  384
Mat-Mat time  =  1.29550899146124 7E-03
MFLOPS =  87414.45135502046
 Error! in ( 1 , 3 )-th argument in PE  0
 Error! in ( 1 , 3 )-th argument in PE  61
 Error! in ( 1 , 3 )-th argument in PE  51
 Error! in ( 1 , 3 )-th argument in PE  58
 Error! in ( 1 , 3 )-th argument in PE  55
 Error! in ( 1 , 3 )-th argument in PE  63
 Error! in ( 1 , 3 )-th argument in PE  60
```

並列化が
完成して
いないので
エラーが出ます。
ですが、
これは正しい
動作です。

サンプルプログラムの説明

- #define N 384
  - 数字を変更すると、行列サイズが変更できます
- #define DEBUG 1
  - 「0」を「1」にすると、行列-行列積の演算結果が検証できます。
- MyMatMat関数の仕様
  - Double型の行列A((N/NPROCS)×N行列)とB((N×(N/NPROCS)行列))の行列積をおこない、Double型の(N/NPROCS)×N行列Cに、その結果が入ります。

Fortran言語のサンプルプログラムの注意

- 行列サイズNの宣言は、以下のファイルにあります。

  mat-mat-d.inc

- 行列サイズ変数が、NNとなっています。

  integer NN

  parameter (NN=384)

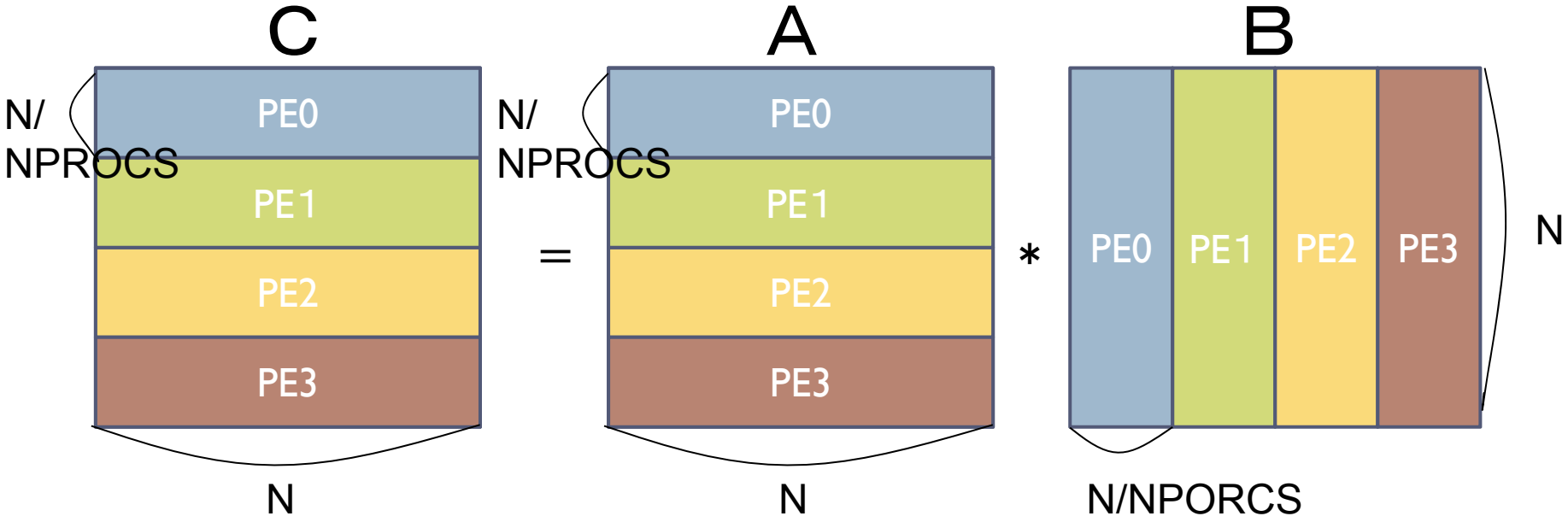
演習課題（１）

- MyMatMat関数（手続き）を並列化してください。
  - デバック時は
    #define　N　384
    としてください。

- 行列A、B、Cの初期配置（データ分散）を、十分に考慮してください。

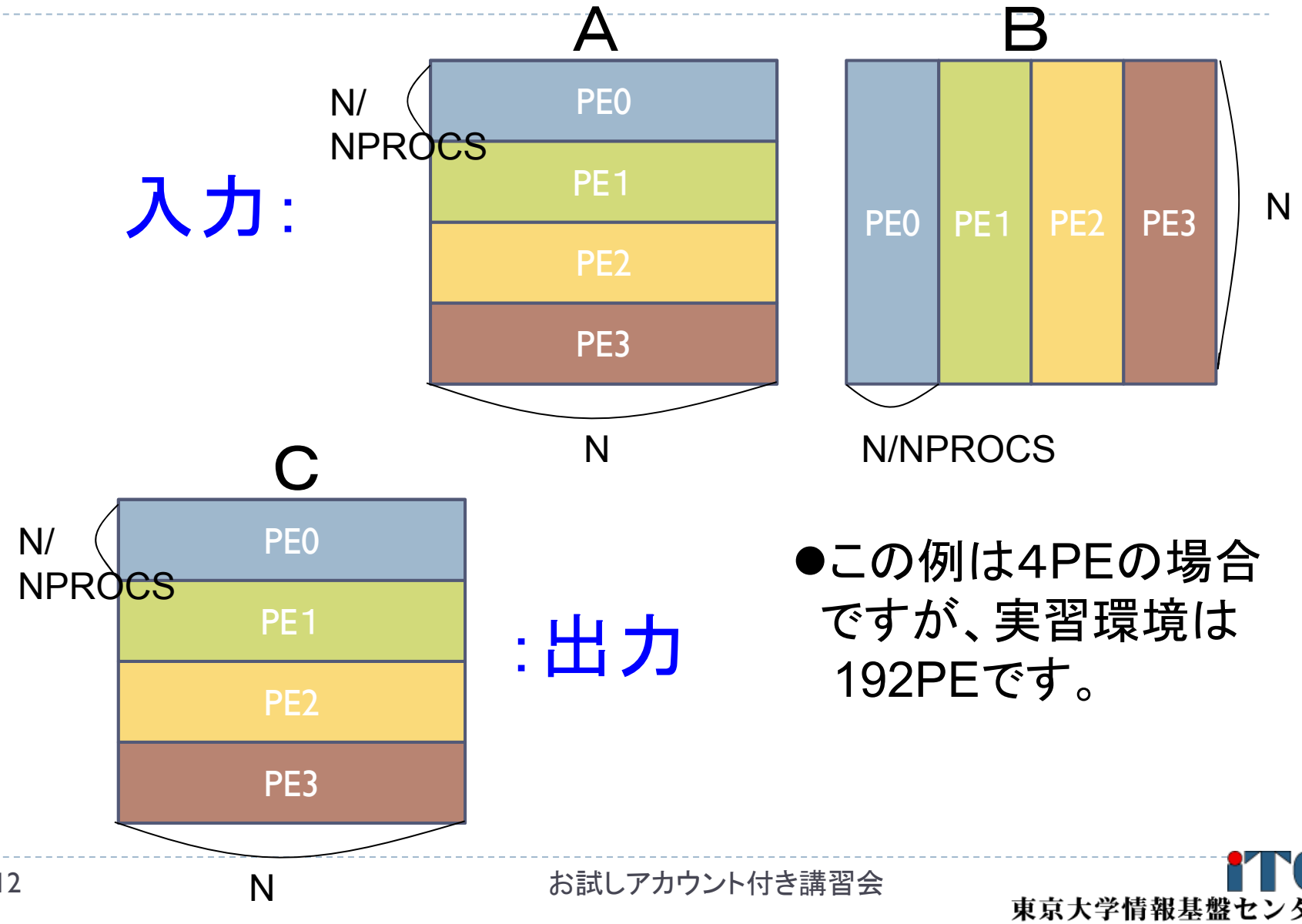
行列A、B、Cの初期配置

- 行列A、B、Cの配置は以下のようになっています。
  （ただし以下は4PEの場合で、実習環境は192PEです。）



- １対１通信関数が必要です。
- 行列A、B、Cの配列のほかに、受信用バッファの配列が必要です。

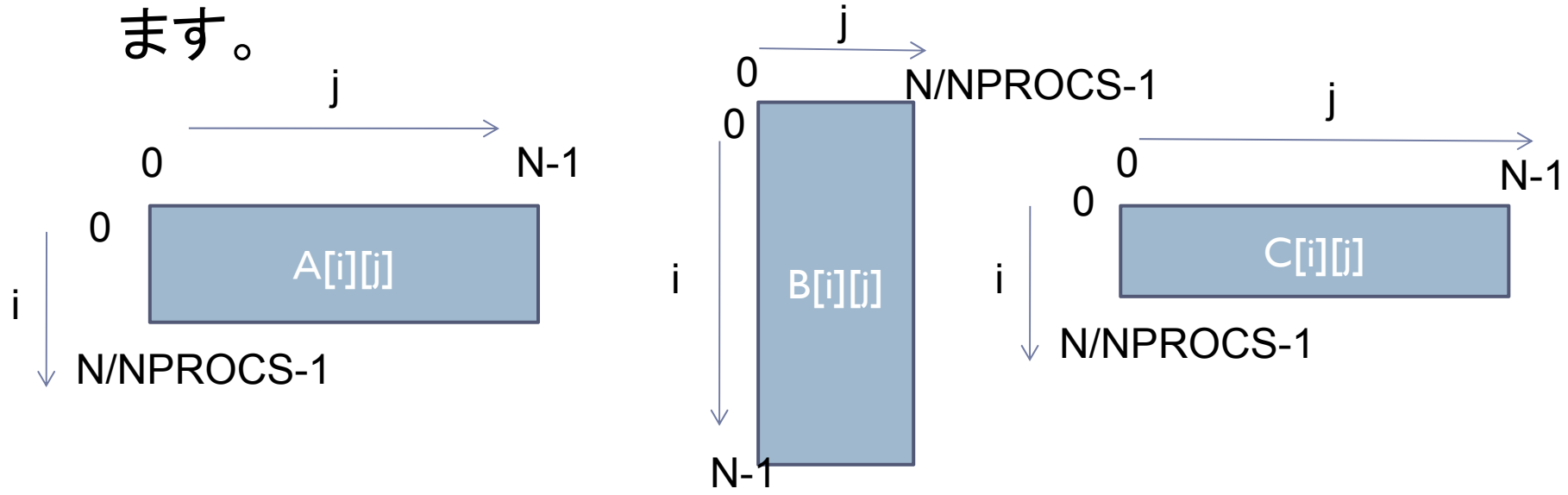
入力と出力仕様



●この例は4PEの場合ですが、実習環境は192PEです。

並列化の注意（C言語）

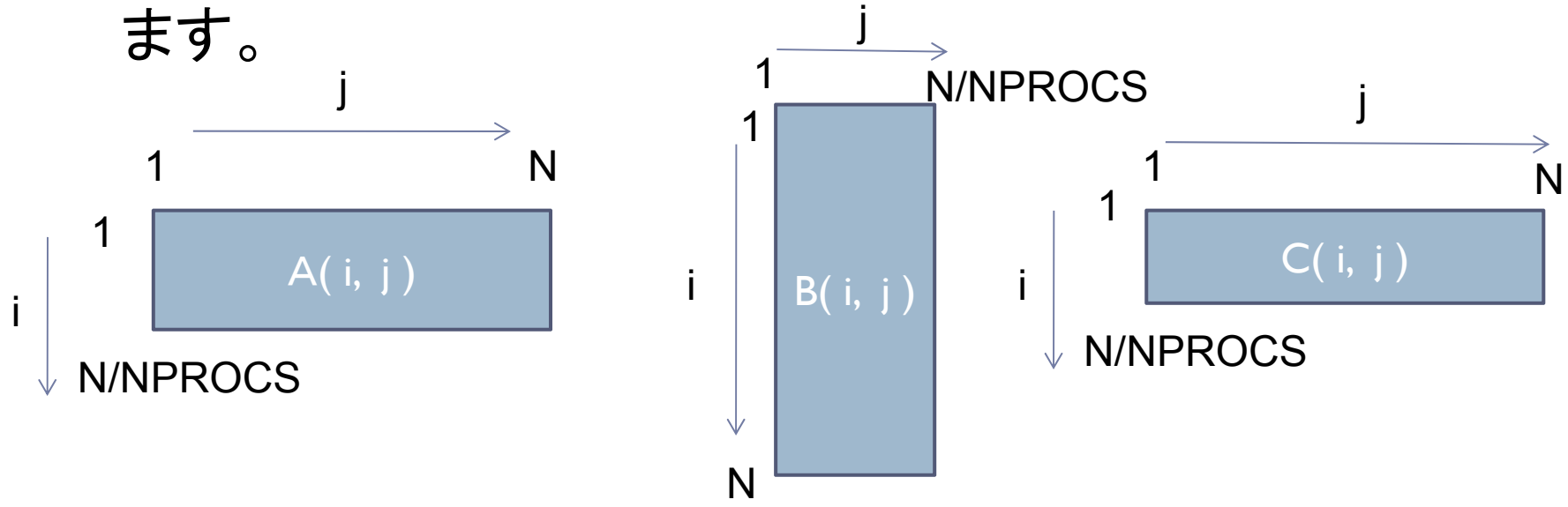
- 各配列は、完全に分散されています。
- 各PEでは、以下のようなインデックスの配列となっています。



- 各PEで行う、ローカルな行列-行列積演算時のインデックス指定に注意してください。

並列化の注意（Fortran言語）

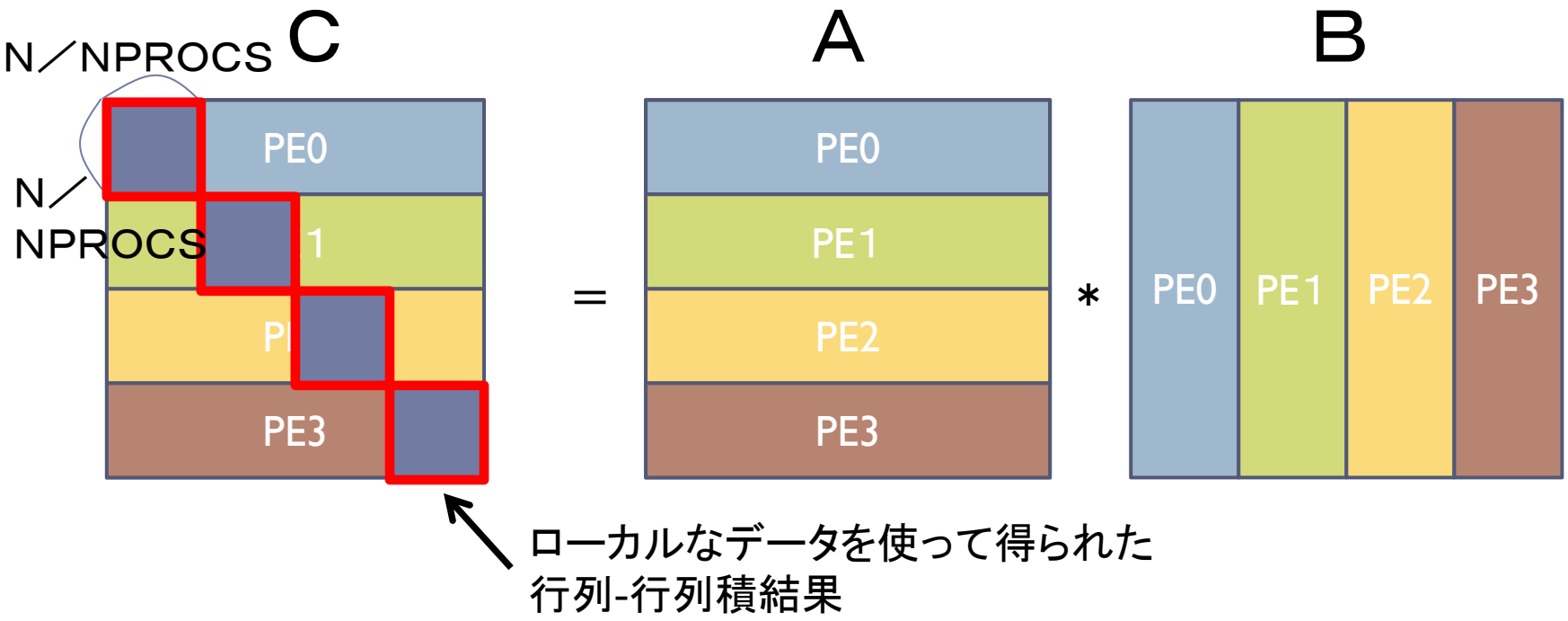
- 各配列は、完全に分散されています。
- 各PEでは、以下のようなインデックスの配列となっています。



- 各PEで行う、ローカルな行列-行列積演算時のインデックス指定に注意してください。

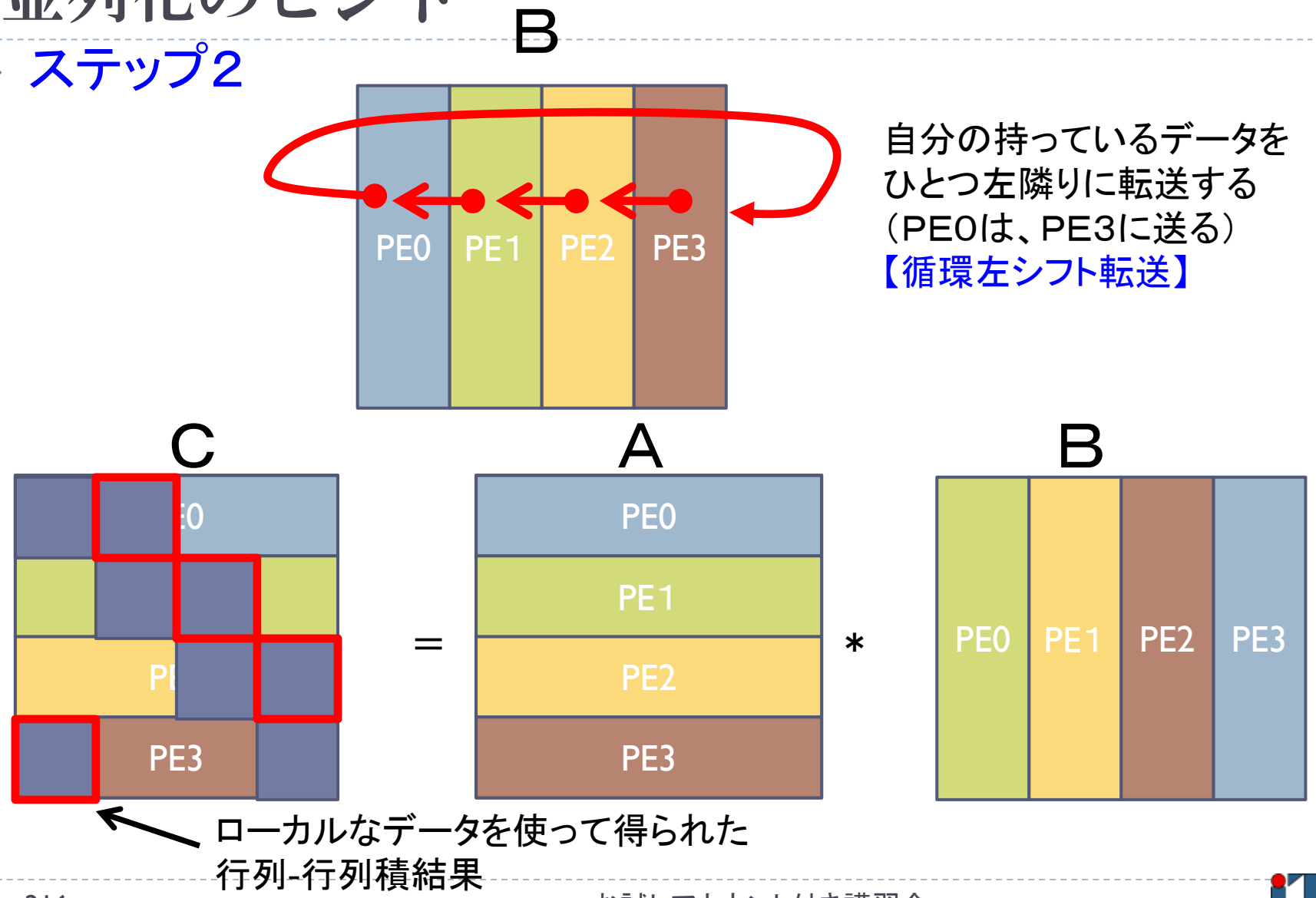
並列化のヒント

- 行列積を計算するには、各PEで完全な行列Bのデータがないので、行列Bのデータについて通信が必要です。
- たとえば、以下のように計算する方法があります。
- ステップ1



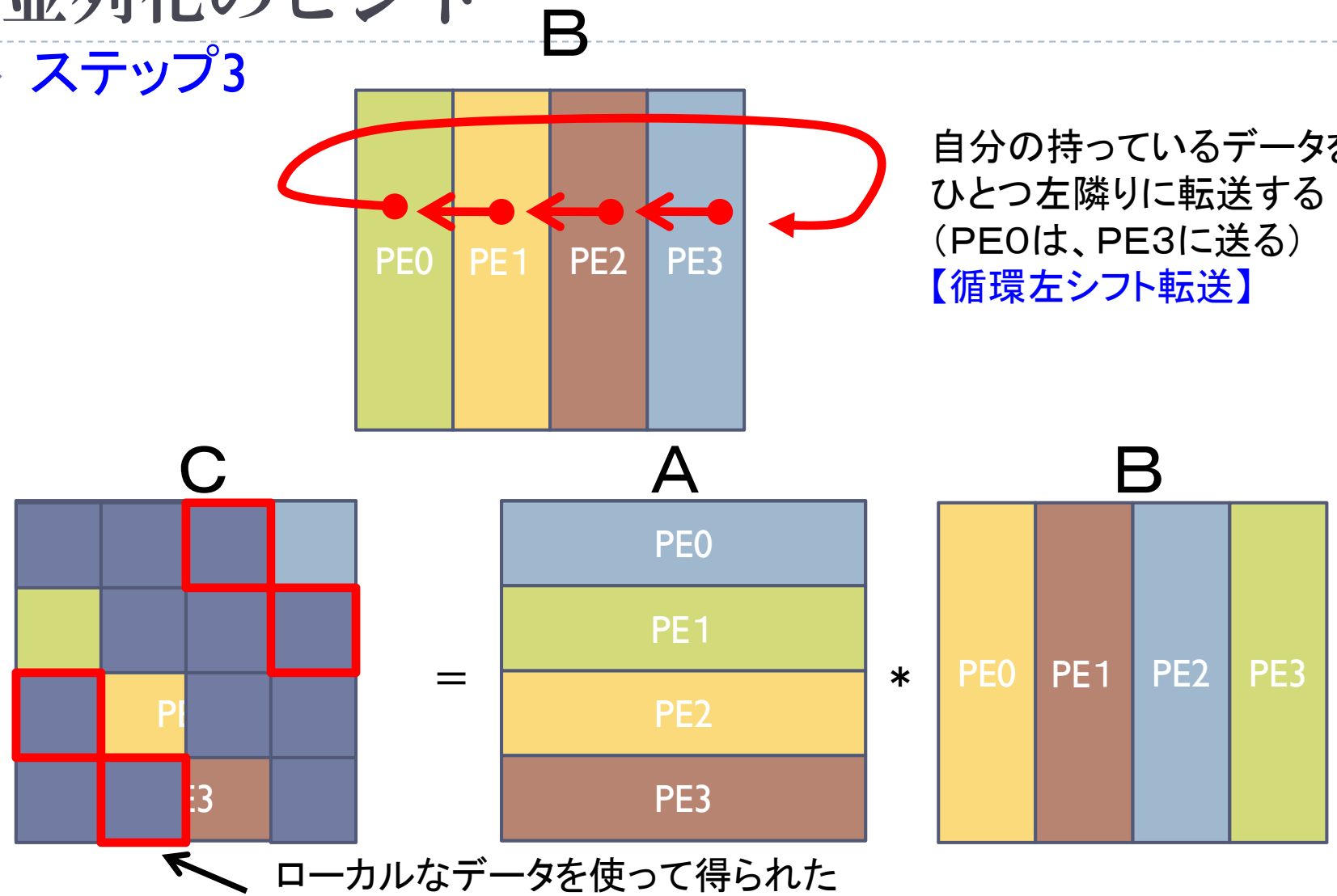
並列化のヒント

- ステップ2



並列化のヒント

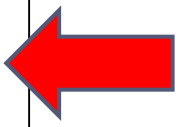
- ステップ3



並列化の注意（１／３）

- 循環左シフト転送を実装する際、全員がMPI_Sendを先に発行すると、その場所で処理が止まる。
（正確には、動いたり、動かなかったり、する）

```
…
  MPI_Send(…);
  MPI_Recv(…);
…
```



このMPI_Sendで止まる

並列化の注意（２／３）

- MPI_Send で止まる理由
  1. MPI_Sendの処理中で、大きいメッセージを送るとき、システムのバッファ領域がなくなる。
  2. バッファ領域が空くまで待つ（スピンウェイトする）。
  3. バッファ領域が空くには、相手の受信が呼ばれる必要がある。
  4. しかし、世の中に受信（MPI_Recv）をコールする人はいない。
  5. バッファ領域が、永遠に空かない。
     →一生、スピンウェイトから脱出できない。
     （MPI_Sendの箇所でずっと止まる）

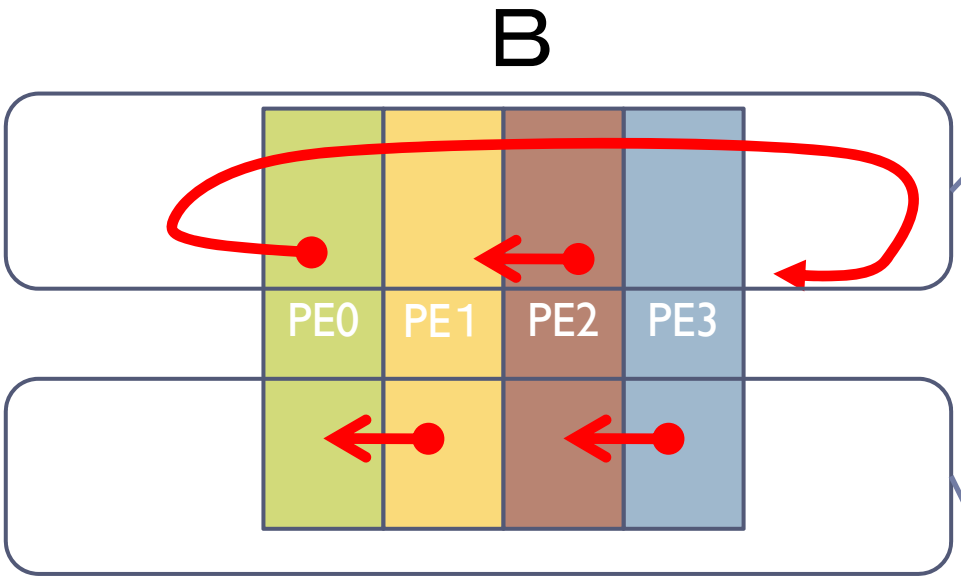
並列化の注意（３／３）

- これ（デットロック）を回避するため、以下の実装を行う。
  - PE番号が2で割り切れるPE：
    - MPI_Send();
    - MPI_Recv();
  - それ以外のPE：
    - MPI_Recv();
    - MPI_Send();

それぞれに対応

デットロック回避の通信パターン

- 以下の2ステップで、循環左シフト通信をする



ステップ1：
2で割り切れるPEが
データを送る

ステップ2：
2で割り切れないPEが
データを送る

基礎的なMPI関数—MPI_Recv（１／２）

- ierr = MPI_Recv(recvbuf, icount, idatatype, isource, itag, icomm, istatus);

  - recvbuf : 受信領域の先頭番地を指定する。
  - icount : 整数型。受信領域のデータ要素数を指定する。
  - idatatype : 整数型。受信領域のデータの型を指定する。
    - MPI_CHAR (文字型) 、MPI_INT (整数型)、MPI_FLOAT (実数型)、 MPI_DOUBLE(倍精度実数型)
  - isource : 整数型。受信したいメッセージを送信するPEのランクを指定する。
    - 任意のPEから受信したいときは、MPI_ANY_SOURCE を指定する。

基礎的なMPI関数—MPI_Recv（２／２）

- itag： 整数型。受信したいメッセージに付いているタグの値を指定する。
  - 任意のタグ値のメッセージを受信したいときは、MPI_ANY_TAG を指定する。
- icomm： 整数型。PE集団を認識する番号であるコミュニケータを指定する。
  - 通常ではMPI_COMM_WORLD を指定すればよい。
- istatus： MPI_Status型（整数型の配列）。受信状況に関する情報が入る。専用の配列を宣言すること。
  - 要素数がMPI_STATUS_SIZEの整数配列が宣言される。
  - 受信したメッセージの送信元のランクが istatus[MPI_SOURCE]、タグが istatus[MPI_TAG] に代入される。
- ierr(戻り値)： 整数型。エラーコードが入る。

実装上の注意

- タグ(itag)について
  - MPI_Send(), MPI_Recv()で現れるタグ（itag）は、任意のint型の数字を指定してよいです。
  - ただし、同じ値（0など）を指定すると、どの通信に対応するかわからなくなり、誤った通信が行われるかもしれません。
  - 循環左シフト通信では、MPI_Send()と MPI_Recv()の対が、2つでてきます。これらを別のタグにした方が、より安全です。
  - たとえば、一方は最外ループの値iloopとして、もう一方をiloop+NPROCSとすれば、全ループ中でタグがぶつかることがなく、安全です。

さらなる並列化のヒント

以降、本当にわからない人のための資料です。
ほぼ回答が載っています。

並列化のヒント

1. 循環左シフトは、PE総数-1回 必要
2. 行列Bのデータを受け取るため、行列B[][]に関するバッファ行列B_T[][]が必要
3. 受け取ったB_T[][] を、ローカルな行列-行列積で使うため、B[][]へコピーする。
4. ローカルな行列-行列積をする場合の、対角ブロックの初期値：ブロック幅*myid。
   ループ毎にブロック幅だけ増やしていくが、Nを超えたら0に戻さなくてはいけない。

並列化のヒント（ほぼ回答，C言語）

- 以下のようなコードになる。

```
ib = n/numprocs;
for (iloop=0;  iloop<NPROCS;  iloop++ )  {
    ローカルな行列-行列積  C  =  A * B;
    if (iloop != (numprocs-1) ) {
      if (myid % 2 == 0 ) {
          MPI_Send(B, ib*n, MPI_DOUBLE, isendPE,
                iloop, MPI_COMM_WORLD);
          MPI_Recv(B_T, ib*n, MPI_DOUBLE, irecvPE,
                iloop+numprocs, MPI_COMM_WORLD, &istatus);
      } else {
          MPI_Recv(B_T, ib*n, MPI_DOUBLE, irecvPE,
                iloop, MPI_COMM_WORLD, &istatus);
          MPI_Send(B, ib*n, MPI_DOUBLE, isendPE,
                iloop+numprocs, MPI_COMM_WORLD);
      }
      B[ ][ ]  へ  B_T[ ][ ]  をコピーする;
    }
  }
```

並列化のヒント（ほぼ回答，C言語）

- ローカルな行列-行列積は、以下のようなコードになる。

```
jstart=ib*( (myid+iloop)%NPROCS );
for (i=0;  i<ib;  i++) {
   for(j=0;  j<ib;  j++) {
      for(k=0;  k<n;  k++) {
         C[ i ][ jstart + j ] += A[ i ][ k ] * B[ k ][ j ];
      }
   }
}
```

並列化のヒント（ほぼ回答，Fortran言語）

- 以下のようなコードになる。

```
ib = n/numprocs
do iloop=0,  NPROCS-1
    ローカルな行列-行列積  C  =  A * B
    if (iloop .ne. (numprocs-1) ) then
      if (mod(myid, 2)  .eq. 0 ) then
         call MPI_SEND(B, ib*n, MPI_DOUBLE_PRECISION,  isendPE,
&                iloop, MPI_COMM_WORLD,  ierr)
         call MPI_RECV(B_T,  ib*n,  MPI_DOUBLE_PRECISION,  irecvPE,
&                iloop+numprocs, MPI_COMM_WORLD,  istatus,  ierr)
      else
         call MPI_RECV(B_T,  ib*n,  MPI_DOUBLE_PRECISION,  irecvPE,
&                iloop,  MPI_COMM_WORLD,  istatus,  ierr)
         call MPI_SEND(B,  ib*n,  MPI_DOUBLE_PRECISION,  isendPE,
&                iloop+numprocs,  MPI_COMM_WORLD,  ierr)
      endif
      B  へ  B_T  をコピーする
    endif
  enddo
```

並列化のヒント（ほぼ回答，Fortran言語）

- ローカルな行列-行列積は、以下のようなコードになる。

```
imod = mod( (myid+iloop), NPROCS )
jstart = ib* imod
do i=1, ib
  do j=1, ib
    do k=1, n
      C( i , jstart + j ) = C( i , jstart + j ) +  A( i , k ) * B( k , j )
    enddo
  enddo
enddo
```

おわり

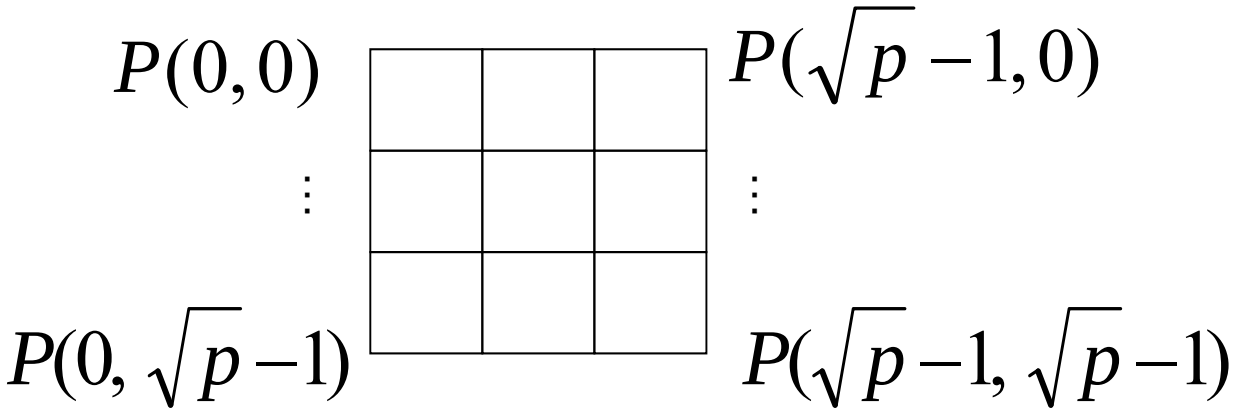
お疲れさまでした

添付資料

行列積の並列アルゴリズムに興味のある方はご覧ください

1.5.1 Cannonのアルゴリズム

- データ分散方式の仮定
  - プロセッサ・グリッドは＜二次元正方＞



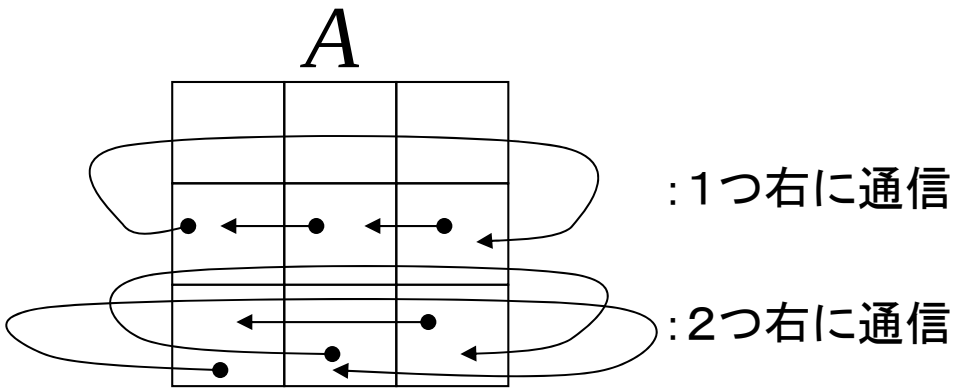
  - PE数が、2のべき乗数しかとれない
  - 各PEは、行列A、B、Cの対応する各小行列を、1つづつ所有
  - 行列A、Bの小行列と同じ大きさの作業領域を所有

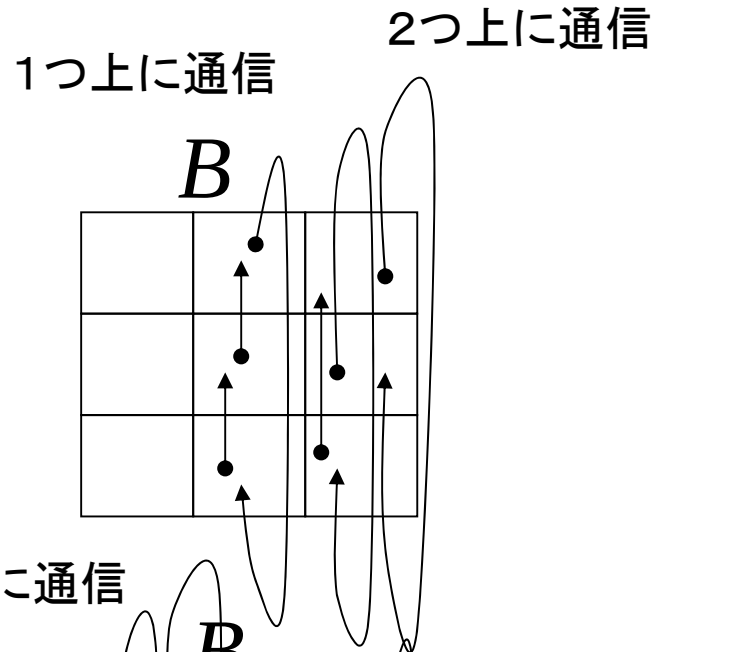
言葉の定義－放送と通信

- 通信
  - ＜通信＞とは、1つのメッセージを1つのPEに送ることである
  - MPI_Send関数、MPI_Recv関数で記述できる処理のこと
  - 1対1通信ともいう
- 放送
  - ＜放送＞とは、同一メッセージを複数のPEに（同時に）通信することである
  - MPI_Bcast関数で記述できる処理のこと
  - 1対多通信ともいう
  - 通信の特殊な場合と考えられる

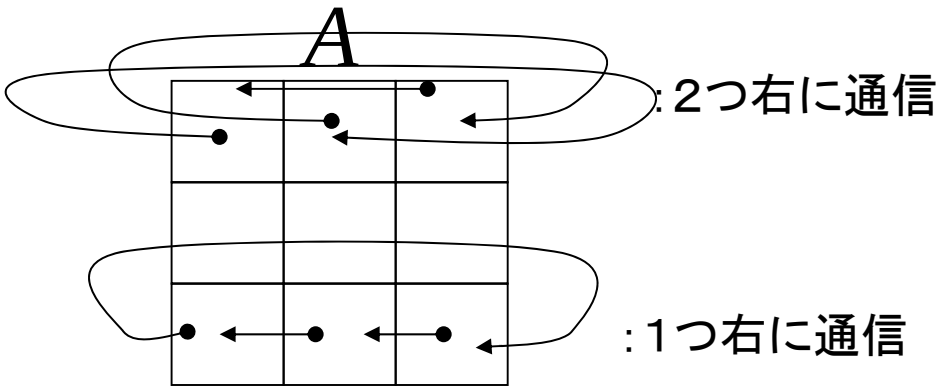
1.5.1 Cannonのアルゴリズム

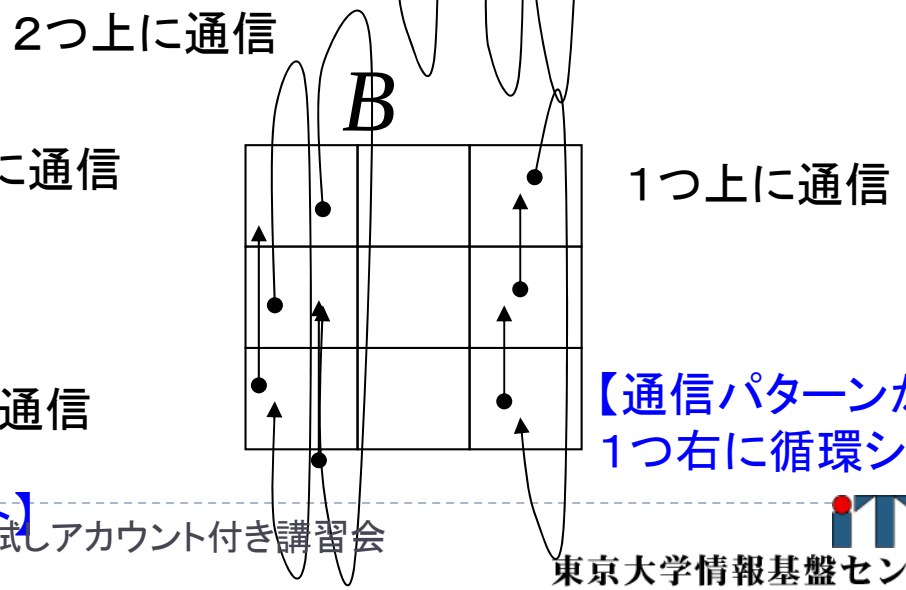
- アルゴリズムの概略
  - 第一ステップ





  - 第二ステップ





【通信パターンが1つ下に循環シフト】

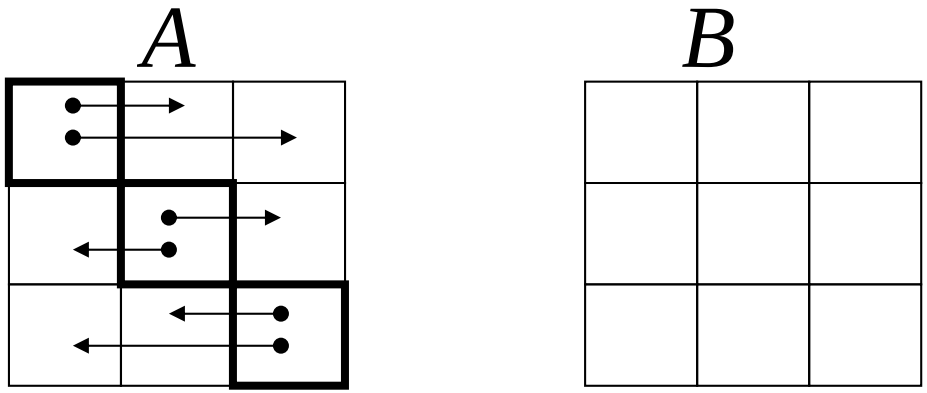
【通信パターンが
1つ右に循環シフト】

1.5.1　Cannonのアルゴリズム

- まとめ
  - ＜循環シフト通信＞のみで実現可能
  - １対１通信（隣接通信）のみで実現可能
  - 物理的に隣接PEにしかつながっていないネットワーク（例えば二次元メッシュ）向き
  - 放送処理がハードウエアでできるネットワークをもつ計算機では、遅くなることも

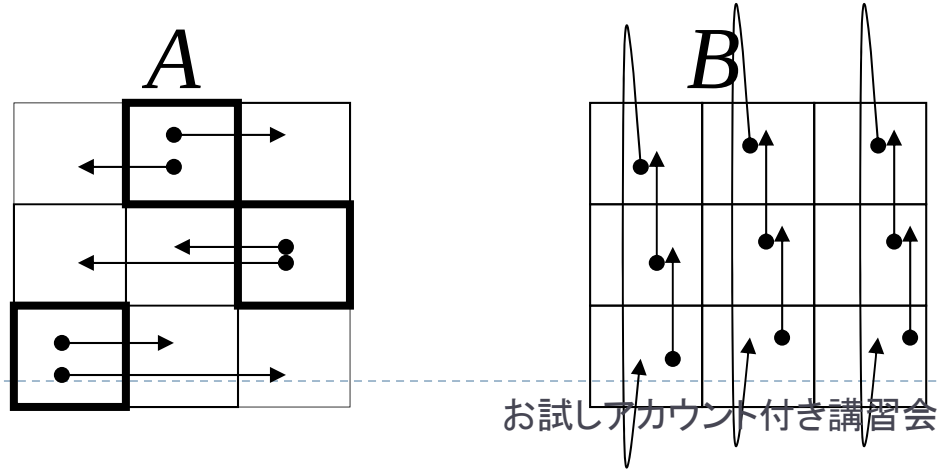
1.5.2　Foxのアルゴリズム

- アルゴリズムの概要
  - 第一ステップ



  - 第二ステップ

【放送PEが
1つ右に
循環シフト】



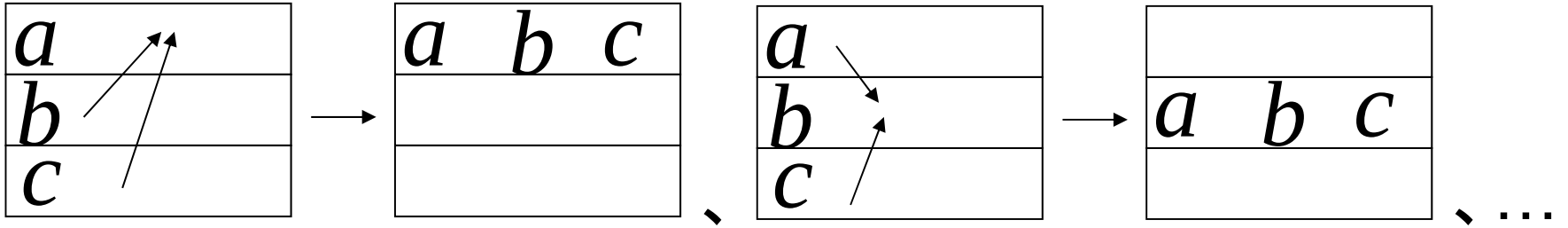
1つ上に通信

1.5.2　Foxのアルゴリズム

- まとめ
  - ＜同時放送（マルチキャスト）＞が必要
  - 物理的に隣接PEにしかつながっていないネットワーク（例えば二次元メッシュ）で性能が悪い（通信衝突が多発）
  - 同時放送がハードウエアでできるネットワークをもつ計算機では、Cannonのアルゴリズムに比べ高速となる

1.5.3　転置を行った後での行列積

- 仮定
  1. データ分散方式：
     行列A、B、C：行方向ブロック分散方式（Block，＊）
  2. メモリに十分な余裕があること：
     分散された行列Bを各PEに全部収集できること
- どうやって、行列Bを収集するか？
  - 2.4節の行列転置の操作をプロセッサ台数回実行



1.5.3　転置を行った後での行列積

- 特徴
  - 一度、行列Bの転置行列が得られれば、*一切通信は不要*
  - 行列Bの転置行列が得られているので、たとえば行方向連続アクセスのみで行列積が実現できる（行列転置の処理が不要）

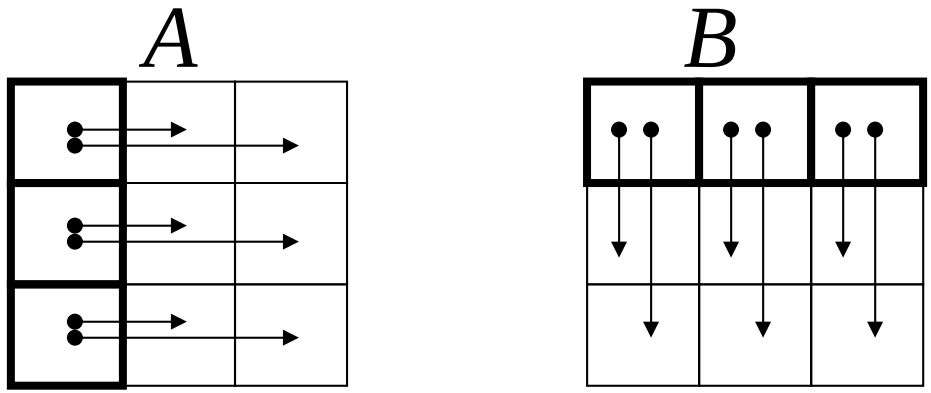
1.5.4　SUMMA、PUMMA

- 近年提案された並列アルゴリズム

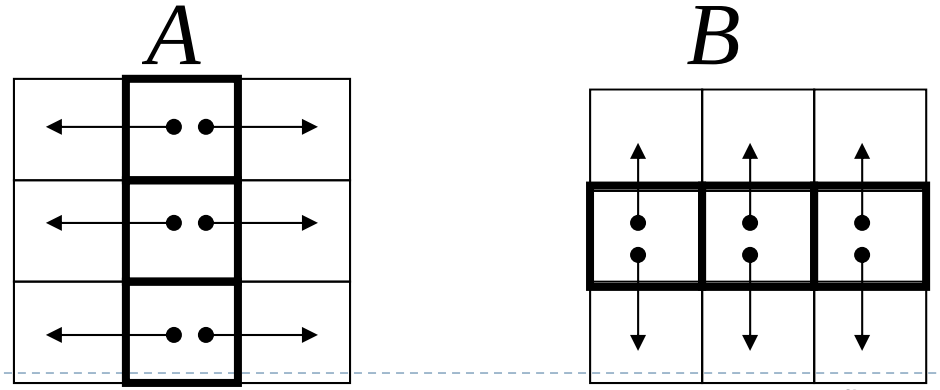
1. SUMMA（Scalable Universal Matrix Multiplication Algorithm）
   - R.Van de Geijinほか、1997年
   - 同時放送（マルチキャスト）のみで実現
2. PUMMA（Parallel Universal Matrix Multiplication Algorithms）
   - Choiほか、1994年
   - 二次元ブロックサイクリック分散方式むきのFoxのアルゴリズム

1.5.4 SUMMA

- アルゴリズムの概略
  - 第一ステップ

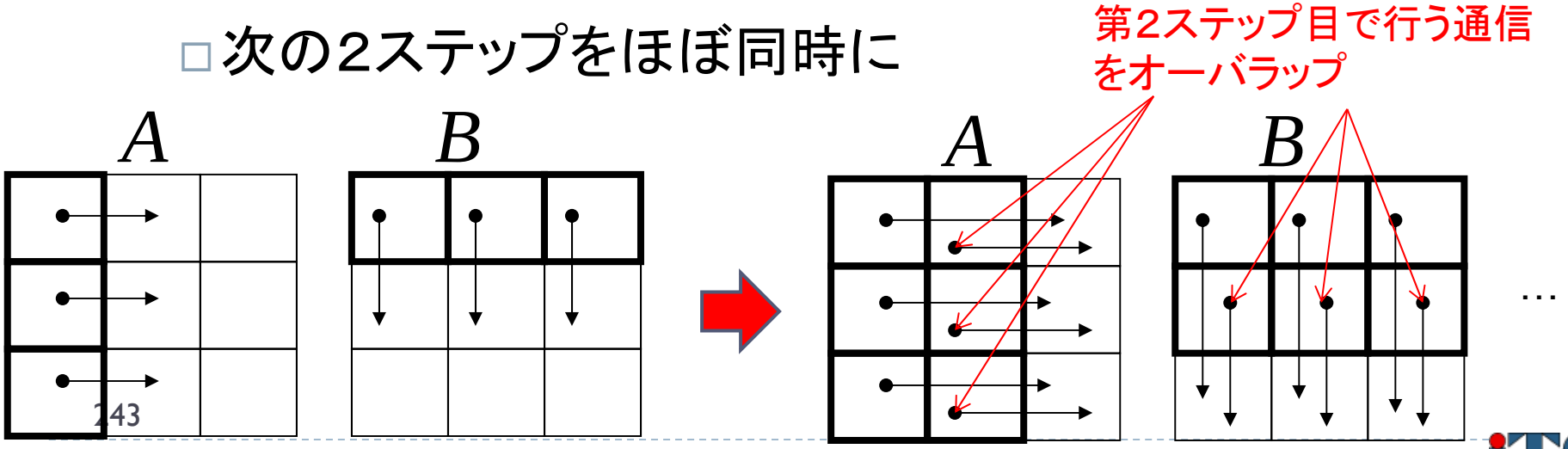


  - 第二ステップ



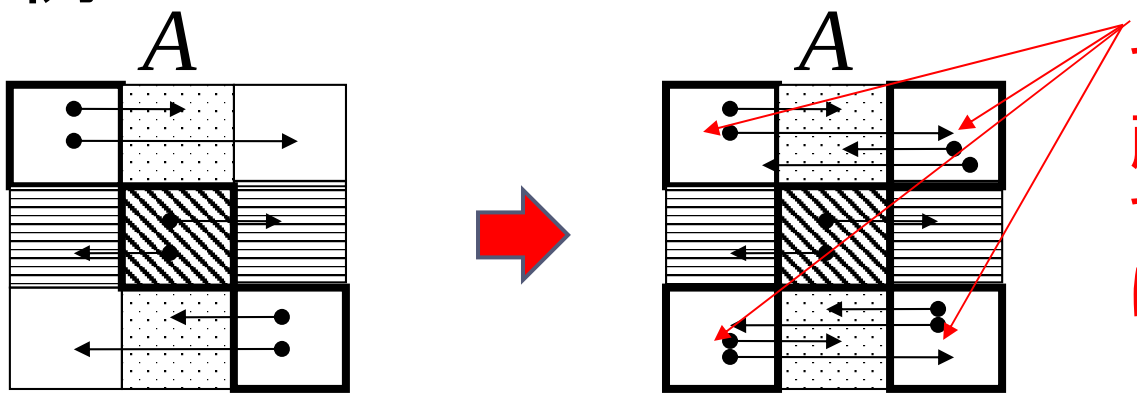
1.5.4 SUMMA

- 特徴
  - 同時放送をブロッキング関数（例. MPI_Bcast）で実装すると、同期回数が多くなり性能低下の要因になる
  - SUMMAにおけるマルチキャストは、非同期通信の1対1通信（例. MPI_Isend）で実装することで、通信と計算のオーバラップ（ ***通信隠蔽*** ）可能
    - 次の2ステップをほぼ同時に



1.5.4 PUMMA

- 概略
  - 二次元ブロックサイクリック分散方式用のFoxアルゴリズム
  - ScaLAPACKが二次元ブロックサイクリック分散を採用していることから開発された
  - 例：



＜同じPE＞が所有しているデータだから、所有データをまとめて＜同一宛先PE＞に一度に送る

1.5.5　Strassenのアルゴリズム

- 素朴な行列積：n^3 の乗算と $(n-1)^3$ の加算
- Strassenのアルゴリズムでは $n^{\log 7}$ の演算
- アイデア：　＜分割統治法＞
  - 行列を小行列に分割して、計算を分割
- 実際の性能
  - 再帰処理や行列のコピーが必要
  - 素朴な実装法より遅くなることがある
  - 再帰の打ち切り、再帰処理展開などの工夫をすれば、（nが大きい範囲で）効率の良い実装が可能

1.5.5　Strassenのアルゴリズム

- 並列化の注意
  - アルゴリズムを単純に分散メモリ型並列計算機に実装すると通信が多発
    - 性能がでない
  - PE内の行列積をStrassenで行い、PE間をSUMMAなどで実装すると効率的な実装が可能