# 第84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入門」

東京大学情報基盤センター

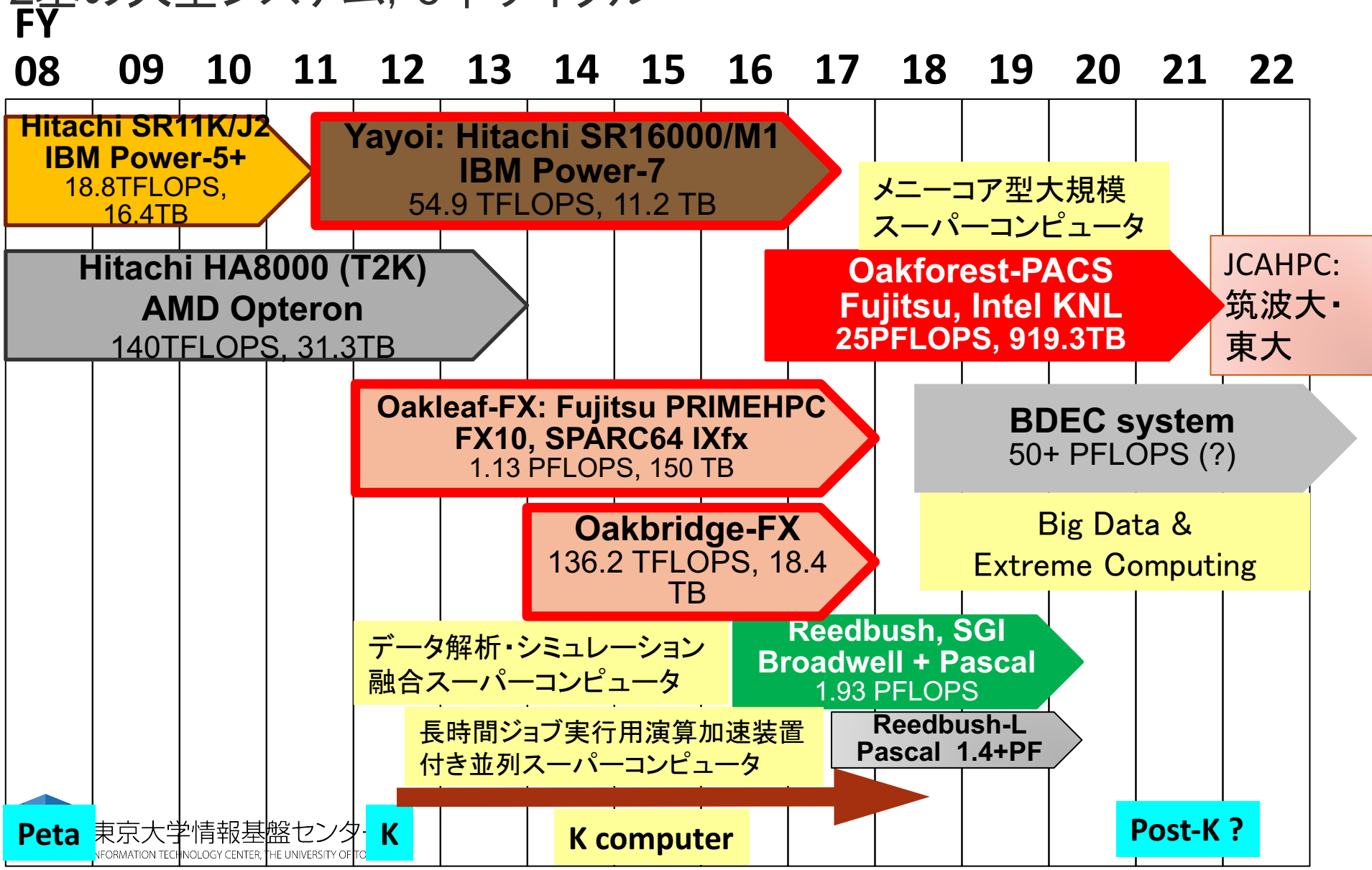
内容に関するご質問は
hanawa @ cc.u-tokyo.ac.jp
まで、お願いします。

# 講習会概略

- 開催日：
  2017年 9月4日（月） 10：30 - 17：00
  2017年 9月5日（火） 10：00 - 17：00
- 場所：東京大学情報基盤センター 4階 413遠隔会議室
  (昼食スペース：3階328 大会議室)
- 講習会プログラム： 講師：塙
- 9月4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）
  - 14：00 - 15：45 並列プログラミングの基本（座学）
  - 16：00 - 17：00 MPIプログラム実習Ⅰ（演習）
- 9月5日（火）
  - 10：00 - 12：30 ファイル入出力の基礎（演習）
  - 14：00 - 15：30 MPIプログラミング実習Ⅱ（演習）
  - 15：45 - 17：00 MPIプログラミング実習Ⅲ（演習）

# 東大センターのスパコン

## 2基の大型システム，6年サイクル



# 5システム運用中





- Yayoi (日立 SR16000, IBM Power7)
  - 54.9 TF, 2011年11月 ～ 2017年10月
- Oakleaf-FX (富士通 PRIMEHPC FX10)
  - 1.135 PF, 京コンピュータ商用版, 2012年4月 ～ 2018年3月
- Oakbridge-FX (富士通 PRIMEHPC FX10)
  - 136.2 TF, 長時間実行用（168時間）, 2014年4月 ～ 2018年3月
- Reedbush (SGI, Intel BDW + NVIDIA P100 (Pascal))
  - データ解析・シミュレーション融合スーパーコンピュータ
  - 1.93 PF, 2016年7月～ 2020年6月
  - 東大ITC初のGPUシステム (2017年3月より), DDN IME (Burst Buffer)
- Oakforest-PACS (OFP) (富士通、Intel Xeon Phi (KNL))
  - JCAHPC (筑波大CCS＆東大ITC)
  - 25 PF, TOP 500で6位 (2016年11月) (日本で1位)
  - Omni-Path アーキテクチャ, DDN IME (Burst Buffer)





# トライアルユース制度について

- 安価に当センターのOakleaf/Oakbridge-FX, Reedbush-U/H, Oakforest-PACS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」制度があります。
  - 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）（FX10: 最大24ノード、最大96ノード, RB-U: 最大16ノード, RB-H: 最大2ノード、OFP: 最大16ノード, 最大64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（FX10:最大1,440ノード、RB-U: 最大128ノード、RB-H: 最大32ノード、OFP: 最大2048ノード）
      - 有償トライアルユース：（1ヶ月～最大通算9ヶ月）、有償（計算資源は無償と同等）
      - 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

# ノートパソコンの設定：公開鍵の生成、登録

# 無線LAN（Wi-Fi）に接続する

- 接続先アクセスポイント　eduroam
- 利用に必要な情報　ユーザーIDとパスワード（配布しているもの）

- よくあるトラブルと対策など
  - 既にeduroamというネットワークに接続したことがある場合
    - 現在も有効なアカウントを持っている場合
      →持っているアカウント情報を使って接続して構いません
    - 現在は有効なアカウントを持っていない場合
      →接続情報を削除してから、配布した情報で接続してください

- 手持ちのノートPCで接続できることを確認してください

# 鍵の作成

1. ターミナルを起動する
2. 以下を入力する

```
$ ssh-keygen  –t  rsa
```

3. 鍵の収納先を聞かれるので、リターンを押す
4. 鍵を使うためのパスワードを聞かれるので、センターのパスワードではない、自分の好きなパスワードを入れる（パスフレーズとよぶ）
5. もう一度、上記のパスフレーズを入れる
6. 鍵が生成される

# 鍵の利用（1/2）

1. 生成した鍵は、以下に入っている
   .ssh/
2. 以下を入力する
   $ cd .ssh/
3. 以下を入力すると、ファイルが見える
   $ ls
   id_rsa id_rsa.pub known_hosts

- ここで、以下のファイルを区別する
  id_rsa : 秘密鍵
  id_rsa.pub：公開鍵

# 鍵の利用(2/2)

4. 以下を入力して、公開鍵を表示する
   $ cat  id_rsa.pub
   ＜公開鍵が表示される＞

5. “ssh-rsa …”で始まる部分を、マウスでカットアンドペーストし、公開鍵の登録に使う 。

# Reedbush-Uへの公開鍵の登録

- 以下をアクセスする。

**https://reedbush-www.cc.u-tokyo.ac.jp/**

- ユーザ名とパスワードを聞かれるので、センターから発行されたユーザ名とパスワードを入れる。
- 注意：記載されたパスワードそのままではNG!

# ユーザ名の確認

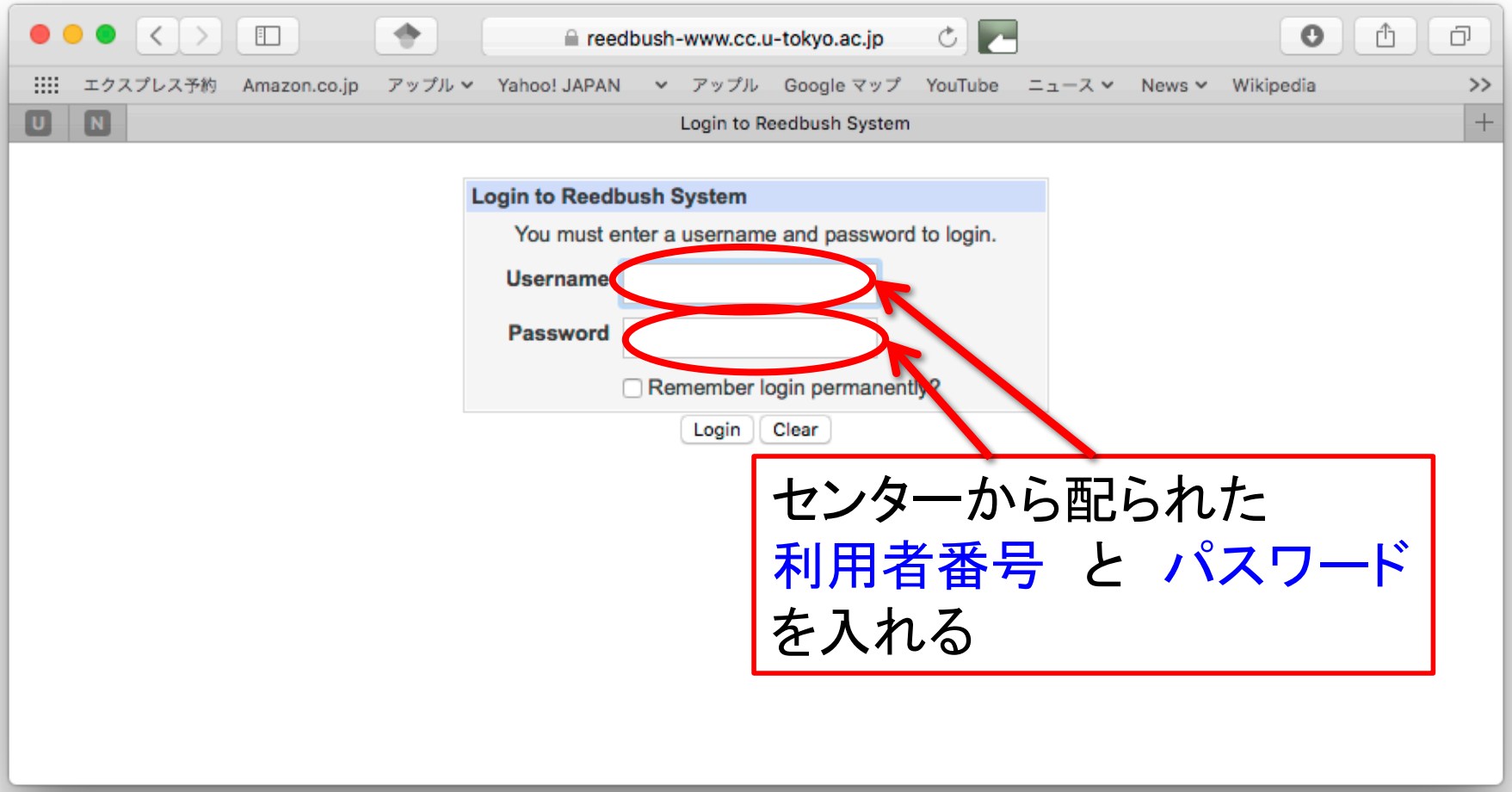
- 本講習会でのユーザー名

利用者番号： t00411～

利用グループ：gt00

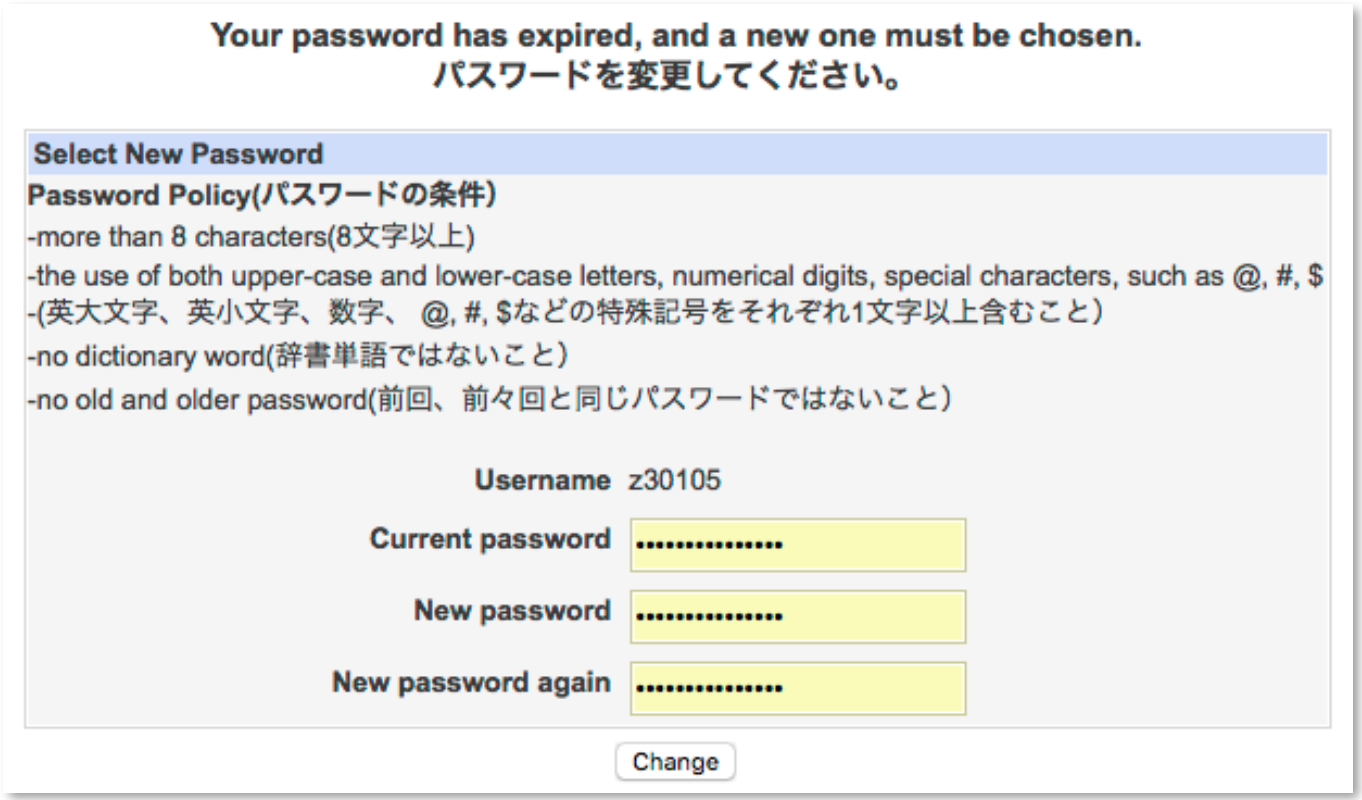
ここで、00411～は数字

# ポータル画面（ログイン前）



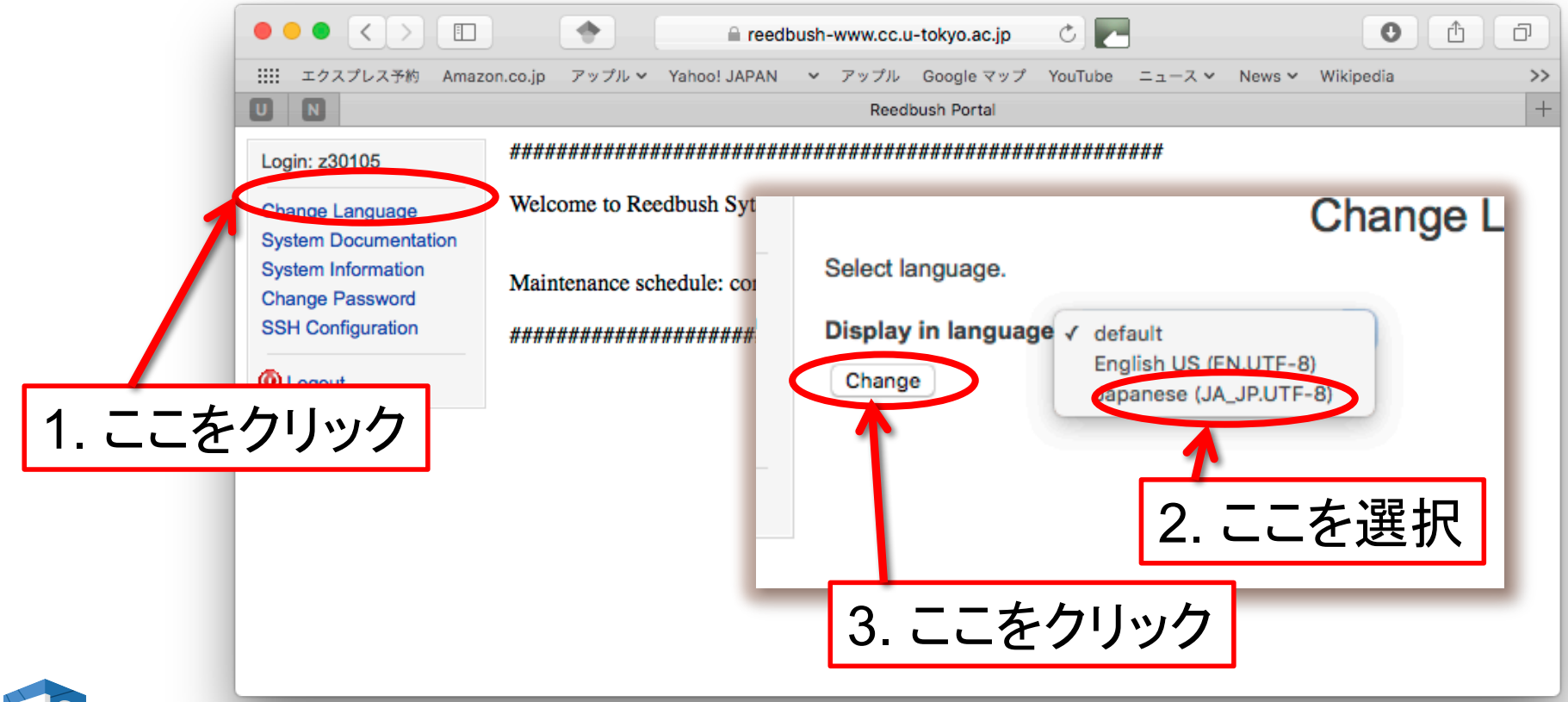
# パスワード変更

- 最初のログイン時にパスワード変更を求められるので、新しいパスワードを入力してください。



# 言語の変更

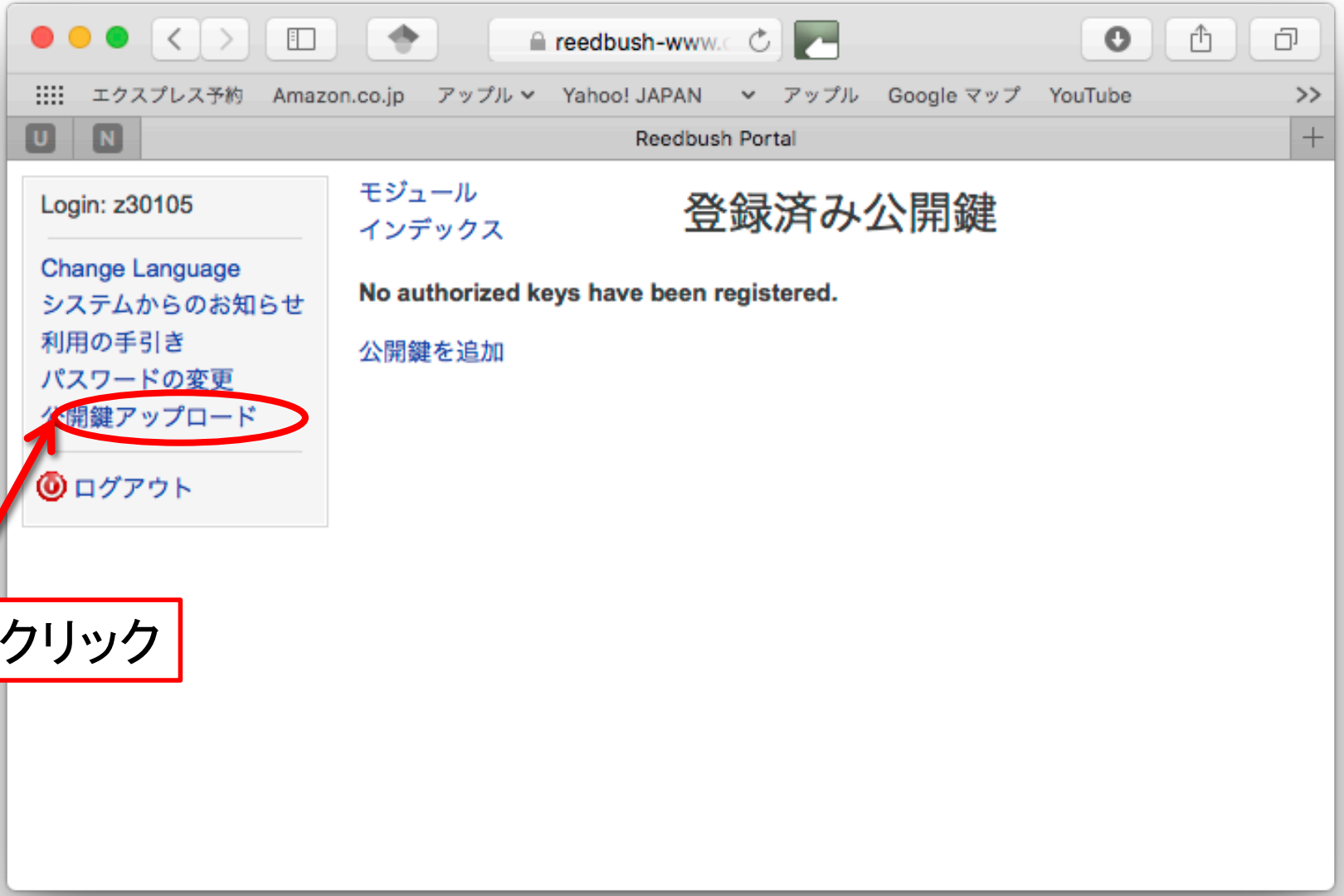
- “Change Language” で日本語に変更できます。
  - “Japanese (JA_JP.UTF-8)” を選んで “Change”を押す
  - 終わったら、ブラウザで再読み込み



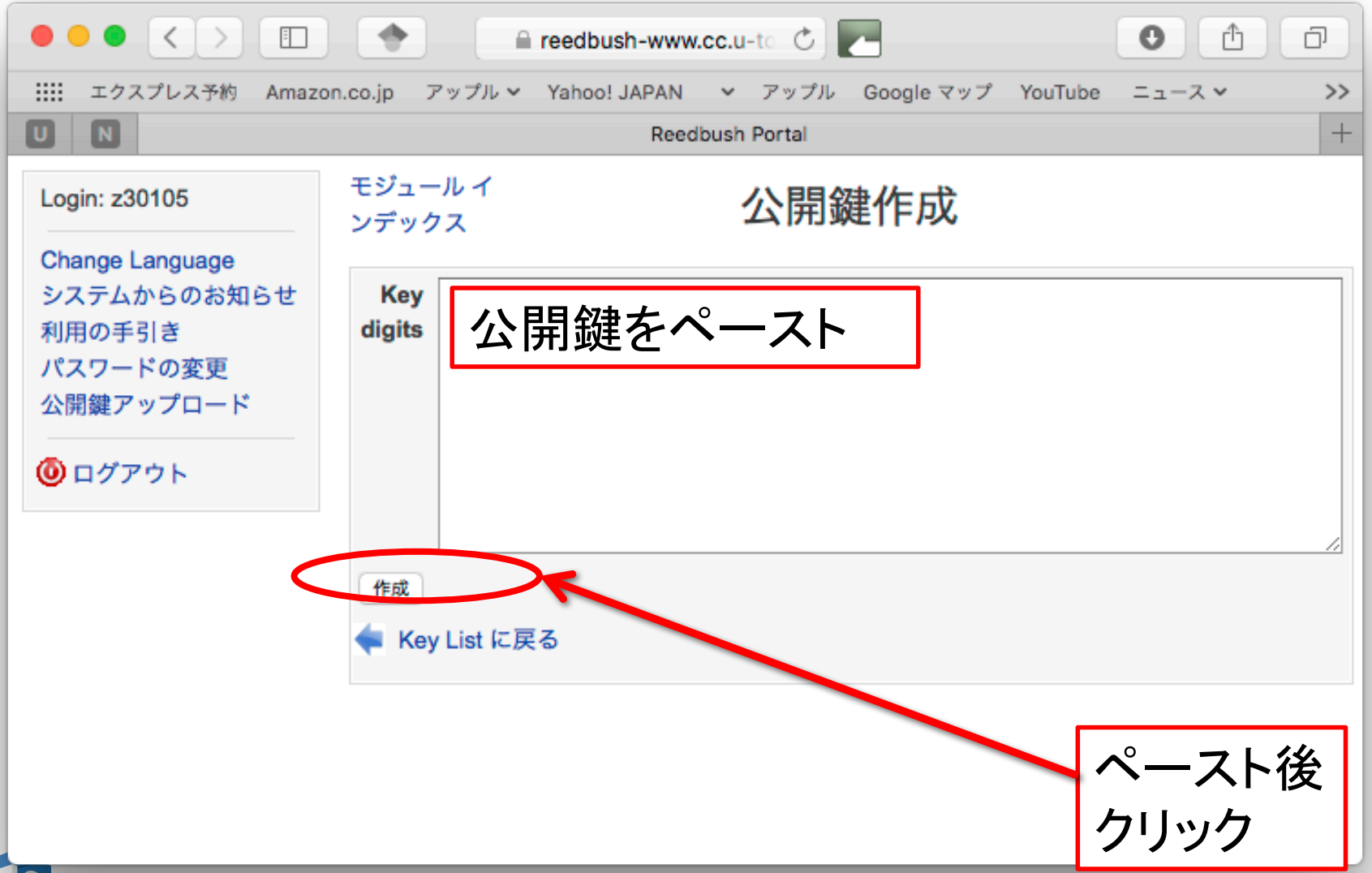
# 鍵の登録

1. 左側メニューの「公開鍵アップロード」をクリックする
2. 「公開鍵を追加」をクリックし、画面に、公開鍵をカットアンドペーストする
3. 「作成」ボタンを押す

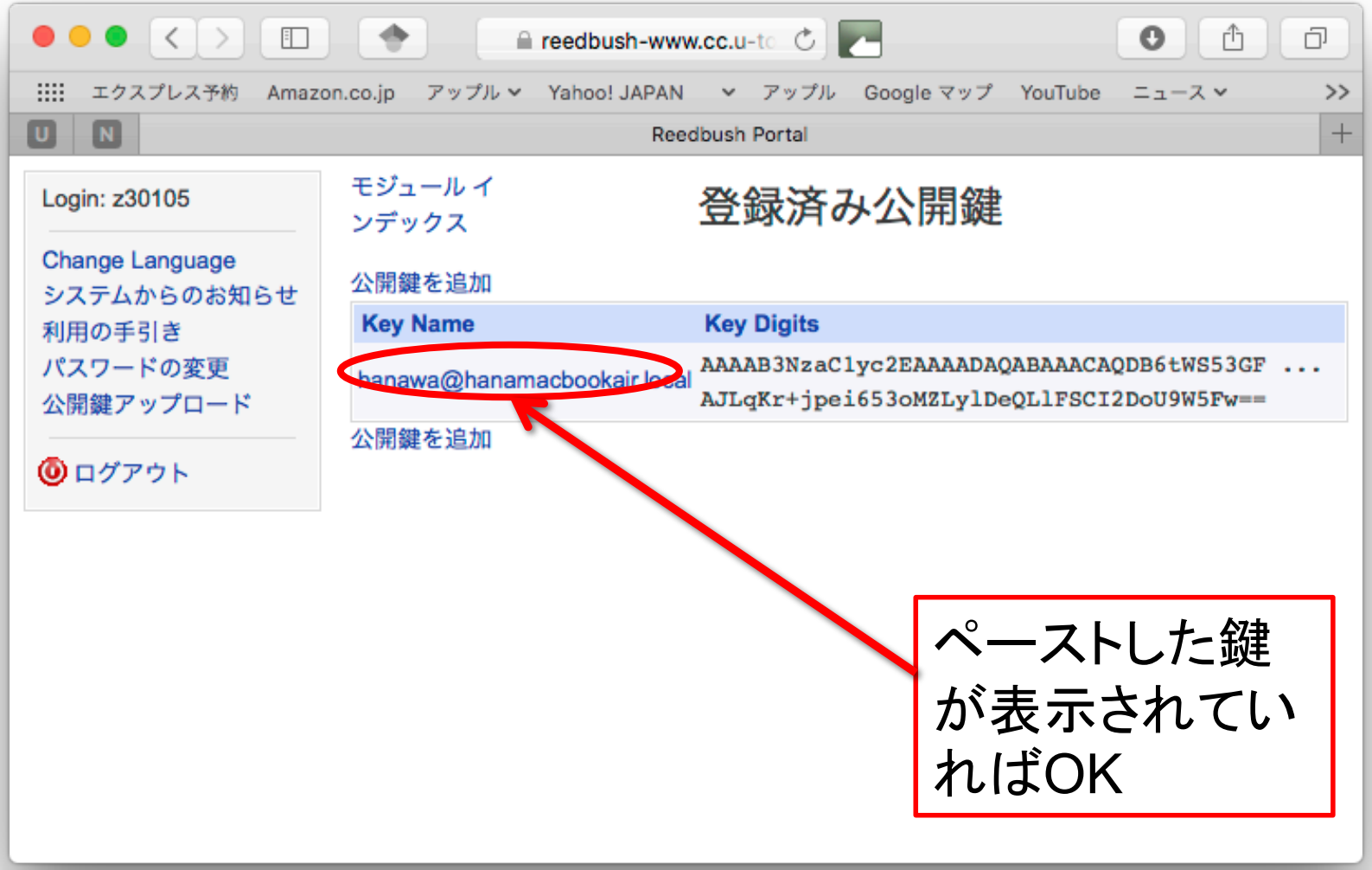
# ポータル画面（言語変更後）



# ポータル画面（公開鍵登録）



# ポータル画面（公開鍵登録成功）



# スパコンへのログイン・テストプログラム起動

# 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

# Reedbush-Uへログイン

- ターミナルから、以下を入力する
  $ ssh reedbush-u.cc.u-tokyo.ac.jp -l tYYxxx
  「-l」はハイフンと小文字のL、
  「tYYxxx」は利用者番号（数字）
  “tYYxxx”は、利用者番号を入れる
- 接続するかと聞かれるので、 yes を入れる
- 鍵の設定時に入れた
  自分が決めたパスワード（パスフレーズ）
  を入れる
- 成功すると、ログインができる

# Reedbush-UのデータをPCに取り込む

- ターミナルから、以下を入力する

$ scp tYYxxx@reedbush-u.cc.u-tokyo.ac.jp:~/a.f90 ./

「tYYxxx」は利用者番号（数字）
“tYYxxx”は、利用者番号を入れる

- Reedbush上のホームディレクトリにある”a.f90”を、PCのカレントディレクトリに取ってくる
- ディレクトリごと取ってくるには、”-r” を指定

$ scp -r tYYxxx@reedbush-u.cc.u-tokyo.ac.jp:~/SAMP ./

- Reedbush上のホームディレクトリにあるSAMPフォルダを、
その中身ごと、PCのカレントディレクトリに取ってくる

# PCのファイルをReedbush-Uに置く

- ターミナルから、以下を入力する

$ scp ./a.f90 tYYxxx@reedbush-u.cc.u-tokyo.ac.jp:

「tYYxxx」は利用者番号（数字）
“tYYxxx”は、利用者番号を入れる

- PCのカレントディレクトリにある”a.f90”を、Reedbush上のホームディレクトリに置く
- ディレクトリごと置くには、”-r” を指定

$ scp -r ./SAMP tYYxxx@reedbush-u.cc.u-tokyo.ac.jp:

- PCのカレントディレクトリにあるSAMPフォルダを、その中身ごと、Reedbush上のホームディレクトリに置く

# EmacsのTramp機能（必要な人のみ）

- emacs が自分のパソコンに入っている人は、Tramp機能により、遠隔のファイルが操作できます
- Reedbushの秘密鍵を、SSHに登録します
- emacs を立ち上げます
- ファイル検索モードにします
  ^x ^f （^はcontrol）
- “Find file: ”の現在のパス名部分を消し、以下を入れます（ただし、t～は自分のログインIDにする）
  Find file:/ssh:tYYxxx@reedbush-u.cc.u-tokyo.ac.jp:
- パスフレーズを入れると、ローカルファイルのようにReedbush上のファイルが編集できます。

# Reedbushにおける注意

- /home ファイルシステムは容量が小さく、ログインに必要なファイルだけを置くための場所です。
  - /home に置いたファイルは計算ノードから参照できません。ジョブの実行もできません。
- 転送が終わったら、/lustre ファイルシステムに移動(mv)してください。
- または、直接 /lustre ファイルシステムを指定して転送してください。

- ホームディレクトリ: /home/gt00/t009XX
  - cd コマンドで移動できます。
- Lustreディレクトリ: /lustre/gt00/t009XX
  - cdw コマンドで移動できます。

# UNIX備忘録 (1/3)

- 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 x goto-line： 指定した行まで移動する。

# UNIX備忘録 (2/3)

- 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。使う時は慎重に。*~ の間に空白が入ってしまうと、全てが消えます。
- 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）

# UNIX備忘録 (3/3)

- less ファイル名： ファイル名の中身を見る(catでは画面がいっぱいになってしまうとき）
  - スペースキー：1画面スクロール
  - / ：文字列の箇所まで移動する。
  - q ： 終了（訳がわからなくなったとき）

# 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-rb.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

  /lustre/gt00/z30105 (/homeでないので注意）

# 並列版Helloプログラムをコンパイルしよう(1/2)

1. cdw コマンドを実行して Lustreファイルシステムに移動する
2. /lustre/gt00/z30105 にある Samples-rb.tar を自分のディレクトリにコピーする
   $ cp /lustre/gt00/z30105/Samples-rb.tar ./
3. Samples-rb.tar を展開する
   $ tar xvf Samples-rb.tar
4. Samples フォルダに入る
   $ cd Samples
5. C言語 : $ cd C
   Fortran90言語 : $ cd F
6. 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
   ```

# Reedbush-Uスーパー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
  - Reedbush-Uでは、以下に限定
    - 1ノード（36コア）（30分まで）
    - 4ノード（144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
  - Reedbush-Uでは、最大128ノード（4,608コア）(24時間)

※講習会アカウントではバッチジョブ実行のみ、最大8ノードまで

# 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
    $ qsub -I -q u-interactive -l select=1 -l walltime=01:00 -W group_list=gt00
  - 4ノード実行用
    $ qsub -I -q u-interactive -l select=4 -l walltime=01:00 -W group_list=gt00

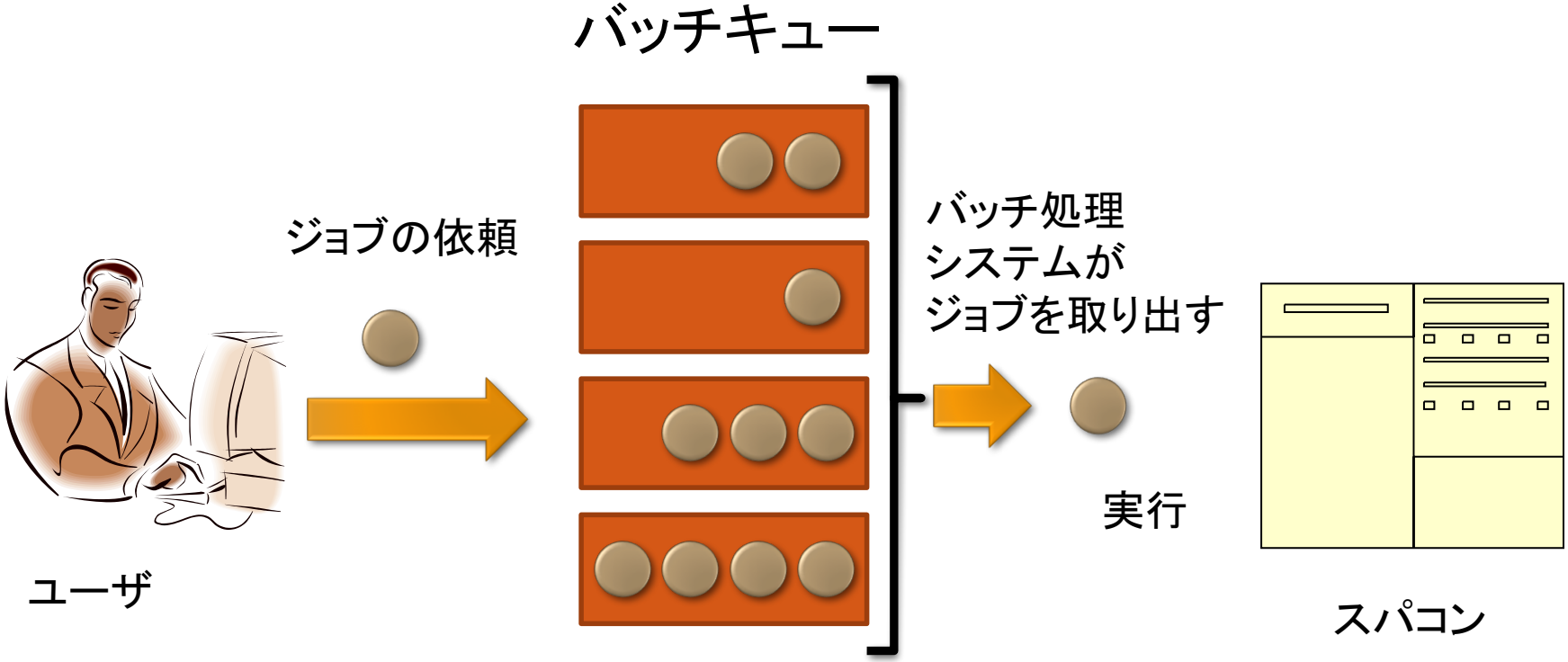
※インタラクティブ用のノードがすべて使われている場合、資源が空くまで、ログインできません。
※講習会アカウントでは使えません。

# 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

- Reedbush-Uでは、コンパイラはバッチ実行、インタラクティブ実行で共通に使えます。
- 例）Intelコンパイラ
  - Cコンパイラ: icc, mpiicc (Intel MPIを使う場合）
  - Fortran90コンパイラ: ifort, mpiifort (Intel MPIを使う場合）

# 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、Altair社のバッチシステム PBS Professional 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：
    qsub <ジョブスクリプトファイル名>
  - 自分が投入したジョブの状況確認： rbstat
  - 投入ジョブの削除： qdel <ジョブID>
  - バッチキューの状態を見る： rbstat --rsc
  - バッチキューの詳細構成を見る： rbstat --rsc -x
  - 投げられているジョブ数を見る： rbstat -b
  - 過去の投入履歴を見る： rbstat -H
  - 同時に投入できる数／実行できる数を見る：rbstat --limit

# 本お試し講習会でのキュー名

- 本演習中のキュー名：
  - u-tutorial
  - 最大10分まで
  - 最大ノード数は8ノード(288コア) まで
- 本演習時間以外（24時間）のキュー名：
  - u-lecture
- 利用条件は演習中のキュー名と同様

# rbstat --rsc の実行画面例

```
$ rbstat --rsc
QUEUE                    STATUS             NODE
u-debug                  [ENABLE ,START]      54
u-short                  [ENABLE ,START]      16
u-regular                [ENABLE ,START]
  |---- u-small          [ENABLE ,START]     288
  |---- u-medium         [ENABLE ,START]     288
  |---- u-large          [ENABLE ,START]     288
  |---- u-x-large        [ENABLE ,START]     288
u-interactive            [ENABLE ,START]
  |---- u-interactive_1  [ENABLE ,START]      54
  |---- u-interactive_4  [ENABLE ,START]      54
u-lecture                [ENABLE ,START]      54
u-lecture8               [DISABLE,START]      54
u-tutorial               [ENABLE ,START]      54
```

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
利用可能数

# rbstat --rsc -x の実行画面例

```
$ rbstat --rsc -x
QUEUE                 STATUS          MIN_NODE MAX_NODE MAX_ELAPSE REMAIN_ELAPSE MEM(GB)/NODE  PROJECT
u-debug               [ENABLE ,START]        1       24   00:30:00      00:30:00         244GB  pz0105,gcXX
u-short               [ENABLE ,START]        1        8   02:00:00      02:00:00         244GB  pz0105,gcXX
u-regular             [ENABLE ,START]
  |---- u-small        [ENABLE ,START]        4       16   12:00:00      12:00:00         244GB  gcXX,pz0105
  |---- u-medium       [ENABLE ,START]       17       32   12:00:00      12:00:00         244GB  gcXX
  |---- u-large        [ENABLE ,START]       33       64   12:00:00      12:00:00         244GB  gcXX
  |---- u-x-large      [ENABLE ,START]       65      128   06:00:00      06:00:00         244GB  gcXX
u-interactive         [ENABLE ,START]
  |---- u-interactive_1 [ENABLE ,START]       1        1   00:15:00      00:15:00         244GB  pz0105,gcXX
  |---- u-interactive_4 [ENABLE ,START]       2        4   00:05:00      00:05:00         244GB  pz0105,gcXX
u-lecture             [ENABLE ,START]        1        8   00:10:00      00:10:00         244GB  gt00,gtYY
u-lecture8            [DISABLE,START]        1        8   00:10:00      00:10:00         244GB  gtYY
u-tutorial            [ENABLE ,START]        1        8   00:10:00      00:10:00         244GB  gt00
```

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

# rbstat --rsc -b の実行画面例

```
$ rbstat --rsc -b
QUEUE                  STATUS          TOTAL RUNNING QUEUED HOLD BEGUN WAIT EXIT TRANSIT NODE
u-debug                [ENABLE ,START]     1       1      0    0     0    0    0       0   54
u-short                [ENABLE ,START]     9       3      5    1     0    0    0       0   16
u-regular              [ENABLE ,START]
  |---- u-small        [ENABLE ,START]    38      10      6   22     0    0    0       0  288
  |---- u-medium       [ENABLE ,START]     2       2      0    0     0    0    0       0  288
  |---- u-large        [ENABLE ,START]     4       2      0    2     0    0    0       0  288
  |---- u-x-large      [ENABLE ,START]     1       0      1    0     0    0    0       0  288
u-interactive          [ENABLE ,START]
  |---- u-interactive_1 [ENABLE ,START]     0       0      0    0     0    0    0       0   54
  |---- u-interactive_4 [ENABLE ,START]     0       0      0    0     0    0    0       0   54
u-lecture              [ENABLE ,START]     0       0      0    0     0    0    0       0   54
u-lecture8             [DISABLE,START]     0       0      0    0     0    0    0       0   54
u-tutorial             [ENABLE ,START]     0       0      0    0     0    0    0       0   54
```

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

ノードの利用可能数

# 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）

```
#!/bin/bash
#PBS -q u-lecture
#PBS -Wgroup_list=gt00
#PBS -l select=8:mpiprocs=36
#PBS -l walltime=00:01:00
cd $PBS_O_WORKDIR
. /etc/profile.d/modules.sh

mpirun ./hello
```

リソースグループ名
：u-lecture

利用グループ名
：gt00

利用ノード数

ノード内利用コア数
（MPIプロセス数）

実行時間制限
：1分

MPIジョブを8＊36 = 288 プロセスで実行する。

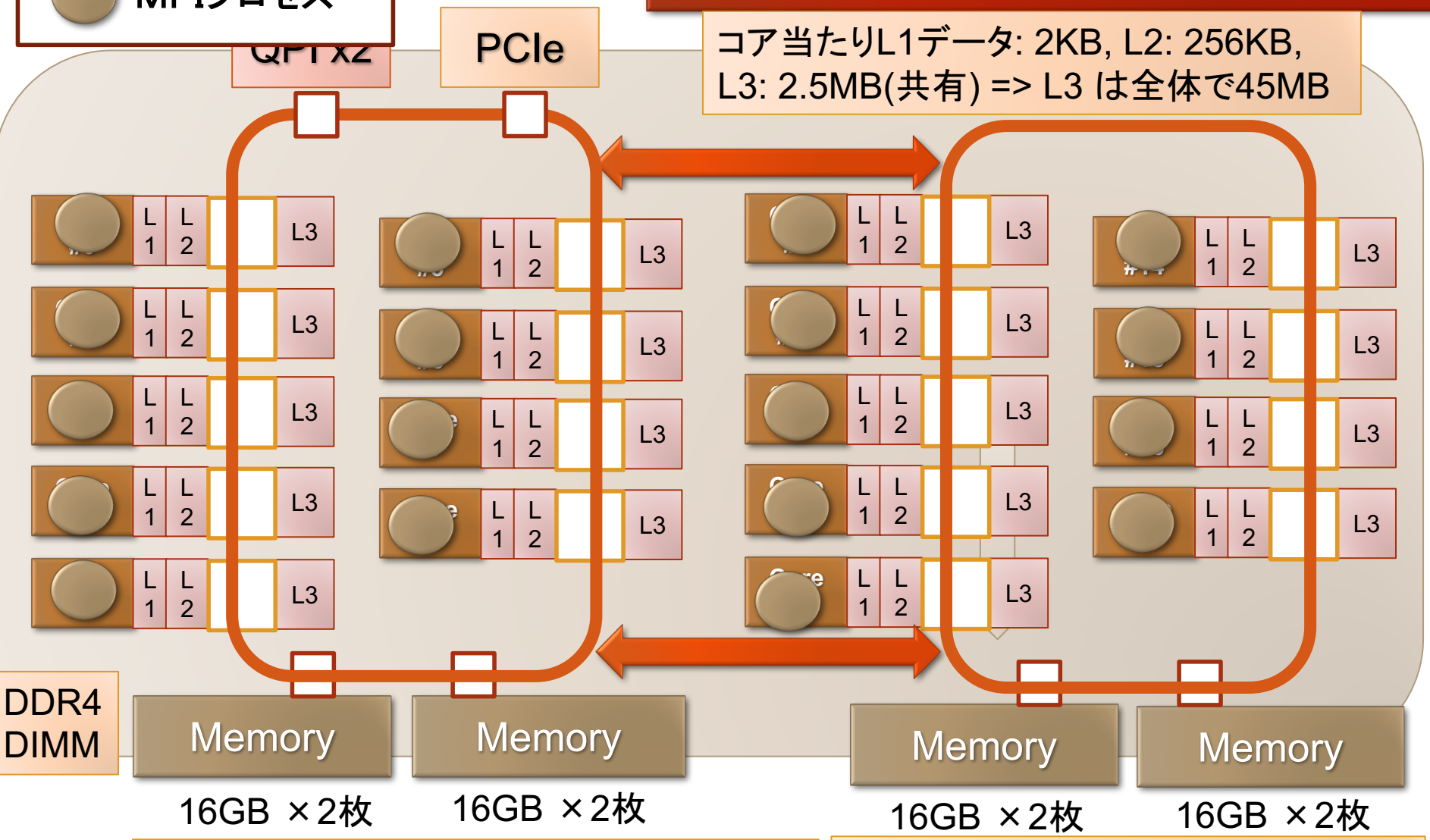
カレントディレクトリ設定、環境変数設定（必ず入れておく）

# ピュアMPIの実行状況（ノード内）

MPIプロセス

1ソケットのみを図示（もう1ソケットある）

コア当たりL1データ: 2KB, L2: 256KB,
L3: 2.5MB(共有) => L3 は全体で45MB




ソケット当たりメモリ量：16GB×8＝128GB

76.8 GB/秒
=(8Byte×2400MHz×4 channel)

# 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”u-lecture”になっています
- $ emacs hello-pure.bash
  で、“u-lecture” → “u-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
   $ qsub hello-pure.bash
2. 自分の導入されたジョブを確認する
   $ rbstat
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、
   36プロセス*8ノード=288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288、もしくは8）

MPIの終了

# 並列版Helloプログラムの説明（Fortran言語）

このプログラムは、全PEで起動される

```
program main

common /mpienv/myid,numprocs

integer  myid, numprocs
integer  ierr

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)

print *, "Hello parallel world!  Myid:", myid

call MPI_FINALIZE(ierr)

stop
end
```

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
288、もしくは8）

MPIの終了

# 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＞

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＞

  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

- Reedbush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す。
  - 例）mpirun ./a.out < in.txt > out.txt

# 依存関係のあるジョブの投げ方（ステップジョブ、チェーンジョブ）

- 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、ステップジョブまたはチェーンジョブという。
- Reedbushにおけるステップジョブの投げ方
  1. $qsub go1.sh
     12345.reedbush-pbsadmin0
  2. 上記のジョブ番号12345を覚えておき、以下の入力をする
     $qsub -W depend=afterok:12345 go2.sh
     12346.reedbush-pbsadmin0
  3. 以下同様
     $qsub -W depend=afterok:12346 go3.sh
     12347.reedbush-pbsadmin0

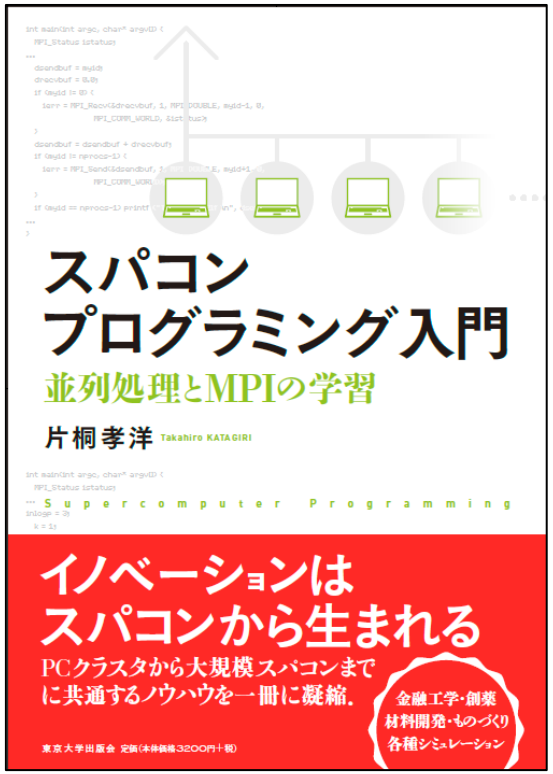
  afterok: 前のジョブが正常に終了したら実行する
  afternotok: 前のジョブが正常終了しなかった場合に実行する
  afterany: どのような状態でも実行する

# 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　塙　敏博

# 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



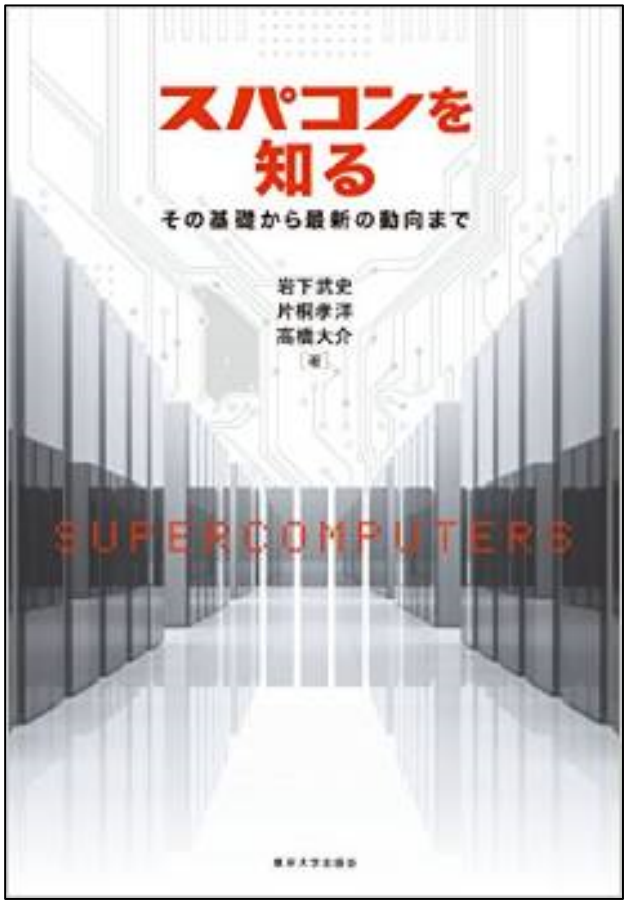
# 教科書（演習書）




- 「並列プログラミング入門：
  サンプルプログラムで学ぶOpenMPとOpenACC」（仮題）
  - 片桐 孝洋　著
  - 東大出版会、ISBN-10: 4130624563、
    ISBN-13: 978-4130624565、発売日： 2015年5月25日
  - 【本書の特徴】
    - C言語、Fortran90言語で解説
    - C言語、Fortran90言語の複数のサンプルプログラムが入手可能（ダウンロード形式）
    - 本講義の内容を全てカバー
    - Windows PC演習可能(Cygwin利用)。スパコンでも演習可能。
    - 内容は初級。初めて並列プログラミングを学ぶ人向けの
      入門書


東京大学情報基盤センター
INFORMATION TECHNOLOGY CENTER, THE UNIVERSITY OF TOKYO

# 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（1/3）

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7月~2018年3月（予定）




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）

# 東京大学情報基盤センター スパコン（2/3-1）

Reedbush-U （SGI Rackable クラスタシステム ）

| | |
|---|---|
| Total Peak performance | : 508 TFLOPS |
| Total number of nodes | : 420 |
| Total memory | : 105 TB |
| Peak performance per node | : 1209.6 GFLOPS |
| Main memory per node | : 256 GB |
| Disk capacity | : 5.04 PB |
| File Cache system (SSD) | : 230 TB |
| **Intel Xeon E5-2695v4 2.1GHz 18 core x2 socket** | |

2016年7月1日試験運転開始

2016年9月1日正式運用開始






# 東京大学情報基盤センター スパコン（2/3-2）

Reedbush-H （SGI Rackable クラスタシステム ）

| | |
|---|---|
| Total Peak performance | : 145 TFLOPS<br>+ 1272 TFLOPS |
| Total number of nodes | : 120 |
| Total memory | : 30 TB + 3.75 TB |
| Peak performance per node | : 1209.6 GFLOPS<br>+ 10.6 TFLOPS |
| Main memory per node | : 256 GB<br>+ 32 GB |
| Disk capacity (shared w/ U) | : 5.04 PB |
| File Cache system (SSD, (shared w/ U) | : 230 TB |

**Intel Xeon E5-2695v4 2.1GHz 18 core x2 socket**
**+ NVIDIA Tesla P100 with NVLink x 2**

2017年3月1日試験運転開始

2017年4月1日正式運用開始



# 東京大学情報基盤センター スパコン（2/3-3）New!!

Reedbush-L （SGI Rackable クラスタシステム ）

| | |
|---|---|
| Total Peak performance | : 77.4 TFLOPS<br>+ 1356 TFLOPS |
| Total number of nodes | : 64 |
| Total memory | : 16TB + 4 TB |
| Peak performance per node | : 1209.6 GFLOPS<br>+ 21.2 TFLOPS |
| Main memory per node | : 256 GB<br>+ 64 GB |
| Disk capacity (shared w/ U,H) | : 5.04 PB |
| File Cache system (SSD, separately) | : 153.6 TB |

**Intel Xeon E5-2695v4 2.1GHz 18 core x2 socket**
**+ NVIDIA Tesla P100 with NVLink x 4**

2017年10月2日試験運転開始（予定）

2017年11月1日正式運用開始（予定）



# 東京大学情報基盤センター スパコン（3/3）

筑波大学計算科学研究センターと共同運用

Oakforest-PACS （Fujitsu PRIMERGY CX600）

| | |
|---|---|
| Total Peak performance | : 25 PFLOPS |
| Total number of nodes | : 8,208 |
| Total memory | : 897.7 TB |
| Peak performance per node | : 3.046 TFLOPS |
| Main memory per node | : 96 GB (DDR4) |
| | + 16 GB(MCDRAM) |
| Disk capacity | : 26.2 PB |
| File Cache system (SSD) | : 960 TB |

**Intel Xeon Phi 7250 1.4 GHz 68 core x1 socket**

2016年12月1日試験運転開始

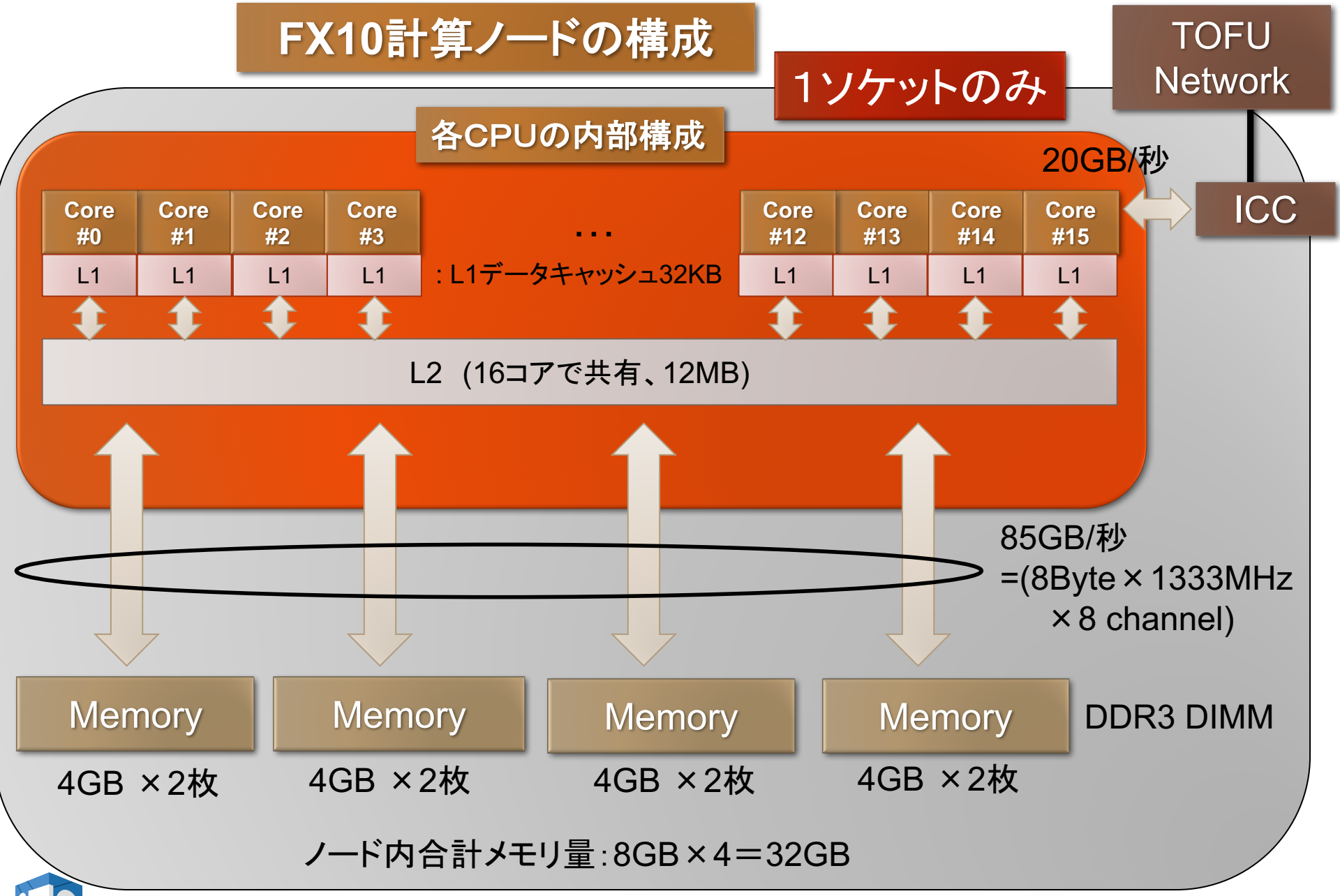
2017年4月3日正式運用開始



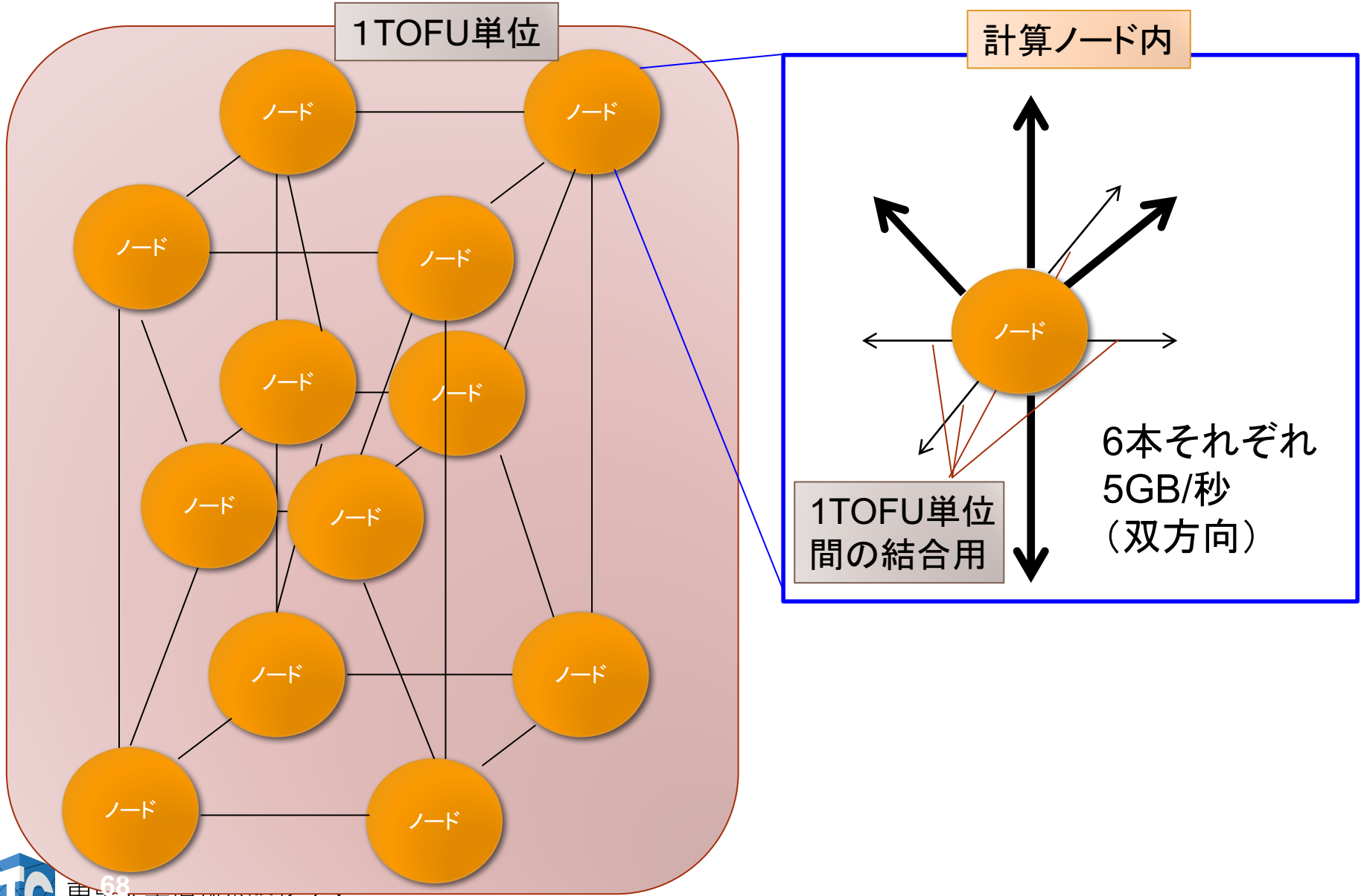




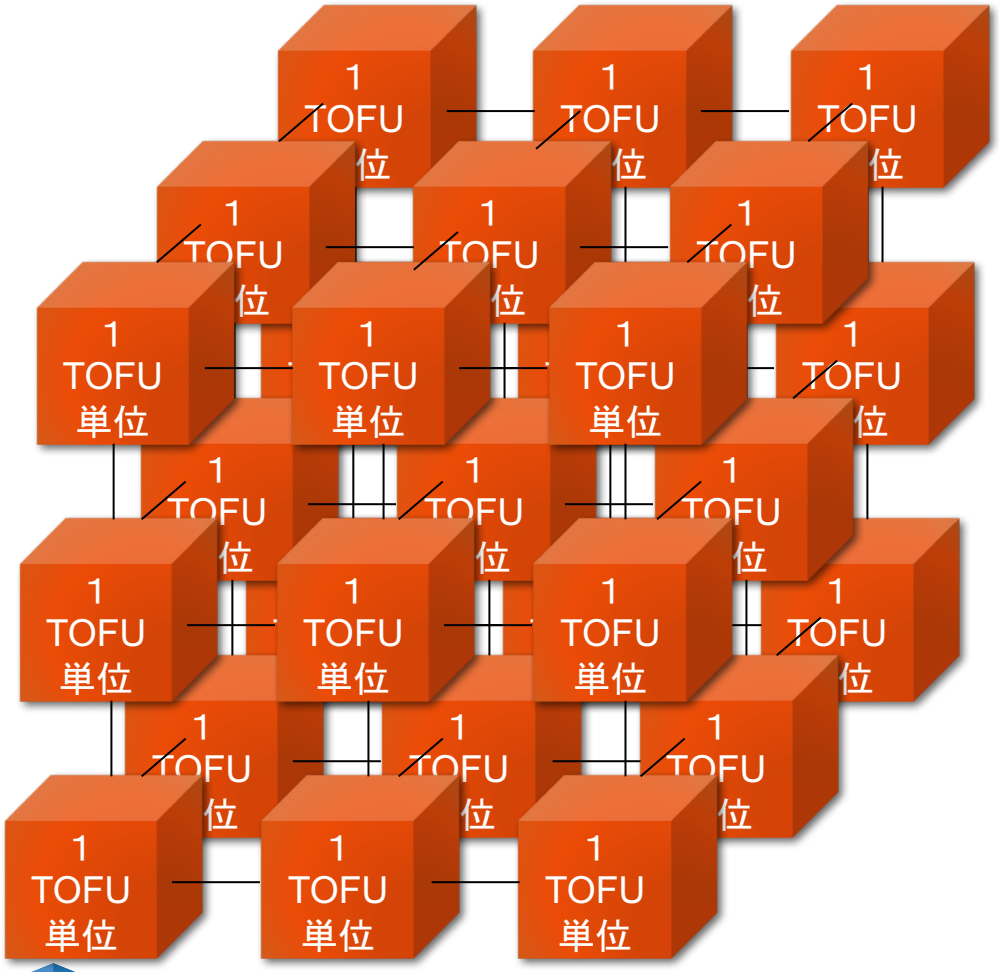
# FX10計算ノードの構成



# FX10の通信網（1TOFU単位）



# FX10の通信網（1TOFU単位間の結合）

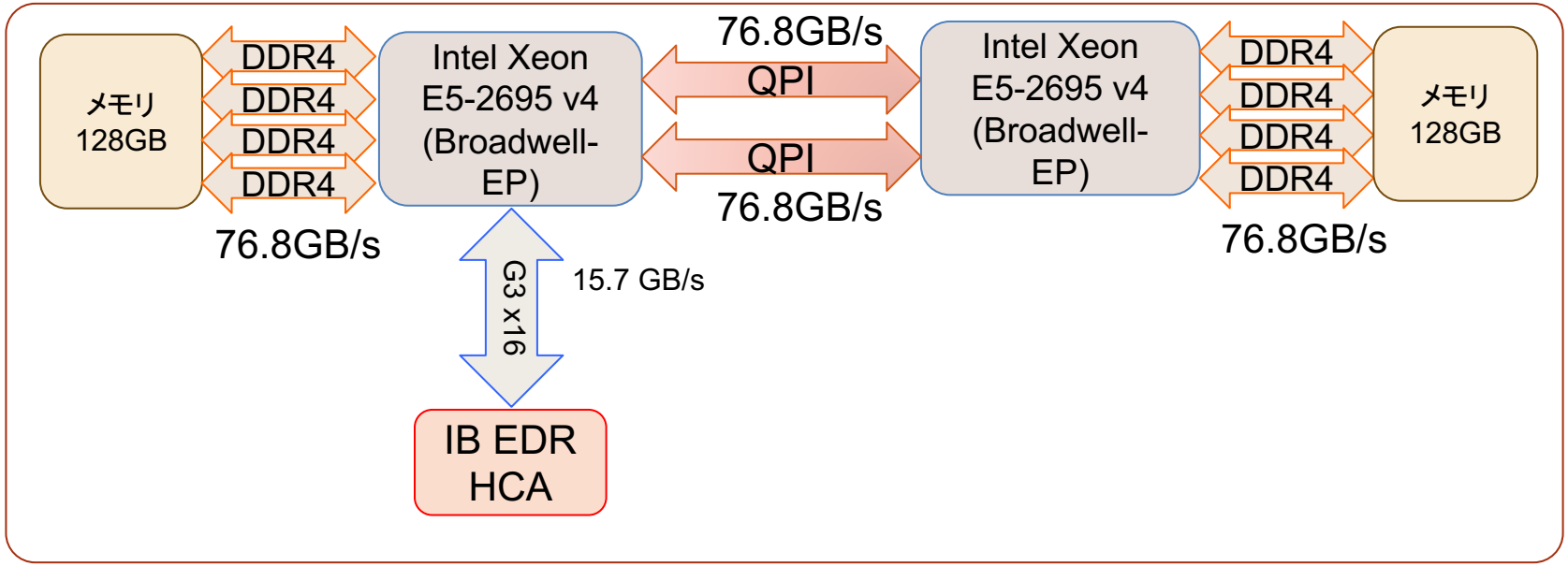



- ユーザから見ると、
  X軸、Y軸、Z軸について、
  奥の1TOFUと、手前の
  1TOFUは、繋がってみえます
（3次元トーラス接続）
- ただし物理結線では
  - X軸はトーラス
  - Y軸はメッシュ
  - Z軸はメッシュまたは、
    トーラス

になっています

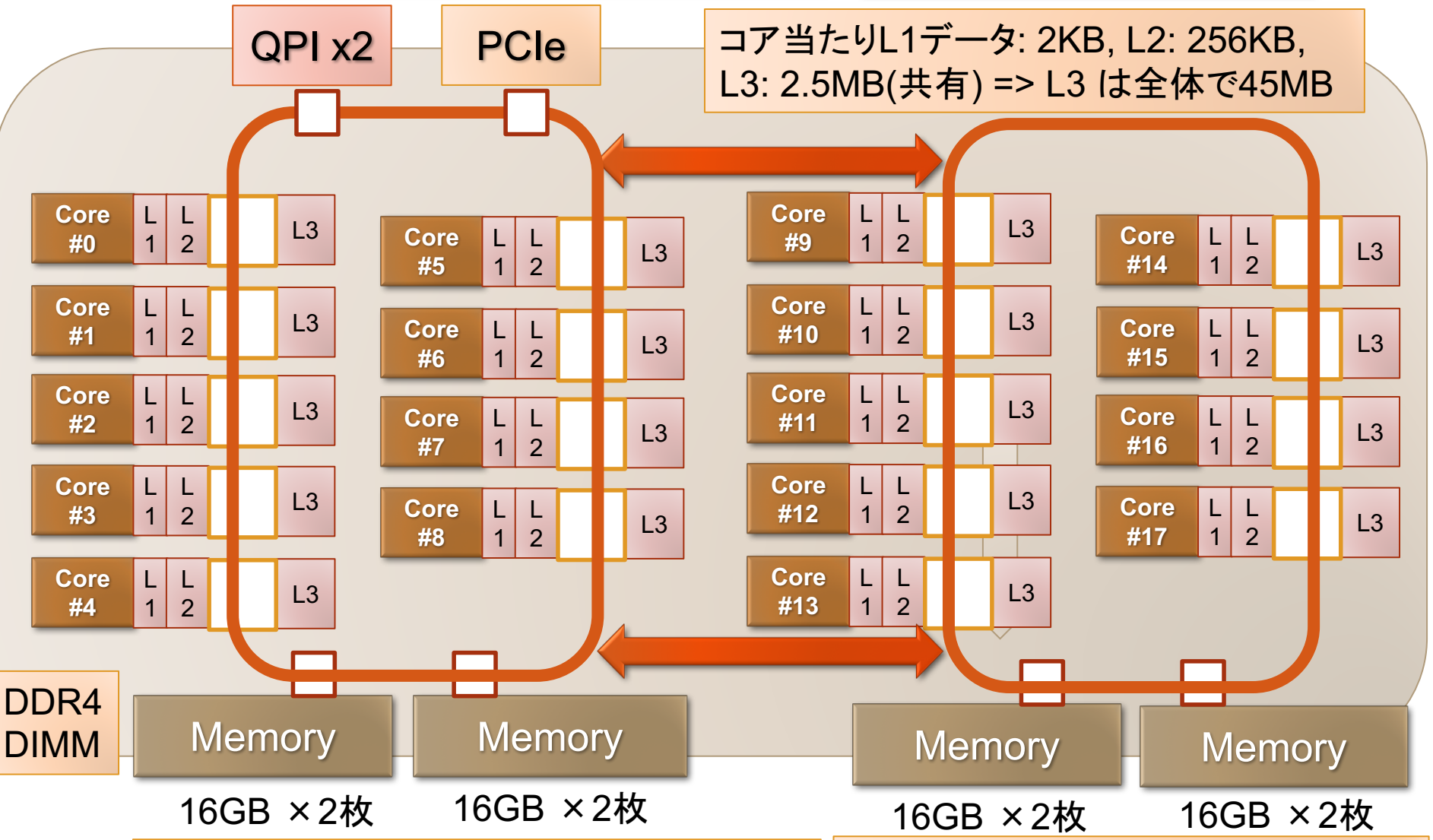
# Reedbush-Uノードのブロック図

- メモリのうち、「近い」メモリと「遠い」メモリがある
  => NUMA (Non-Uniform Memory Access)
  (FX10はフラット)



# Broadwell-EPの構成

## 1ソケットのみを図示

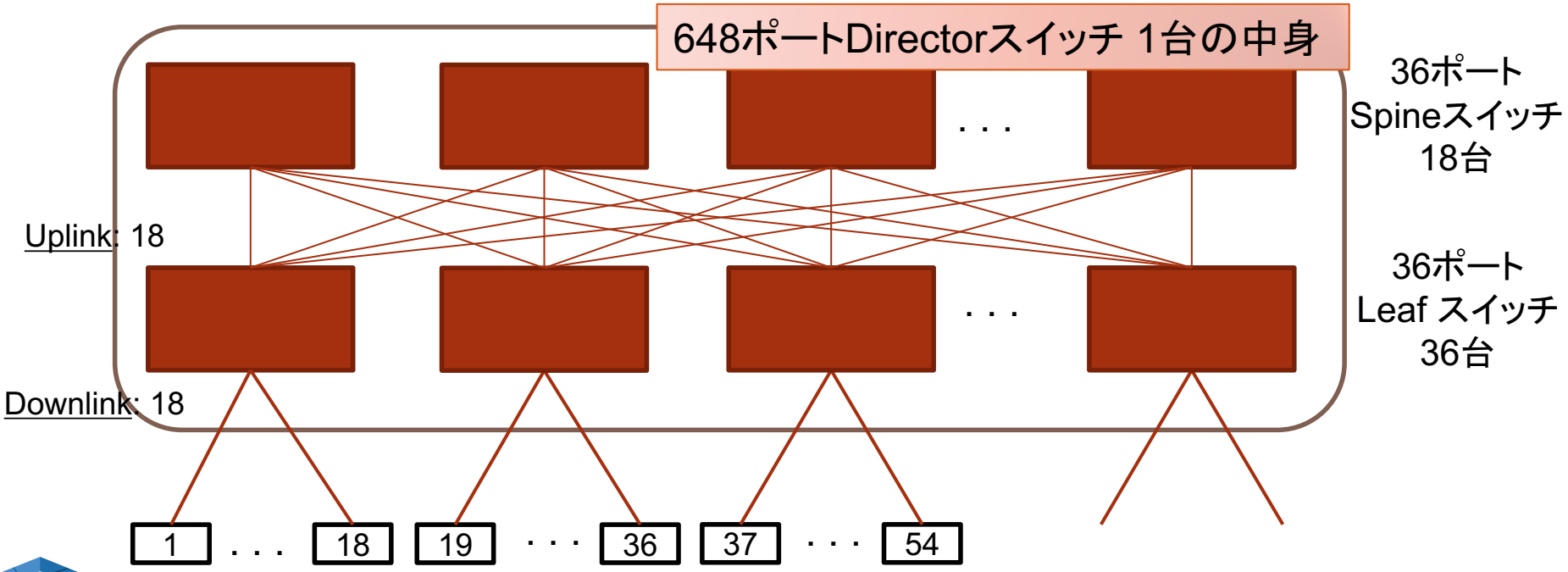



ソケット当たりメモリ量：16GB×8＝128GB

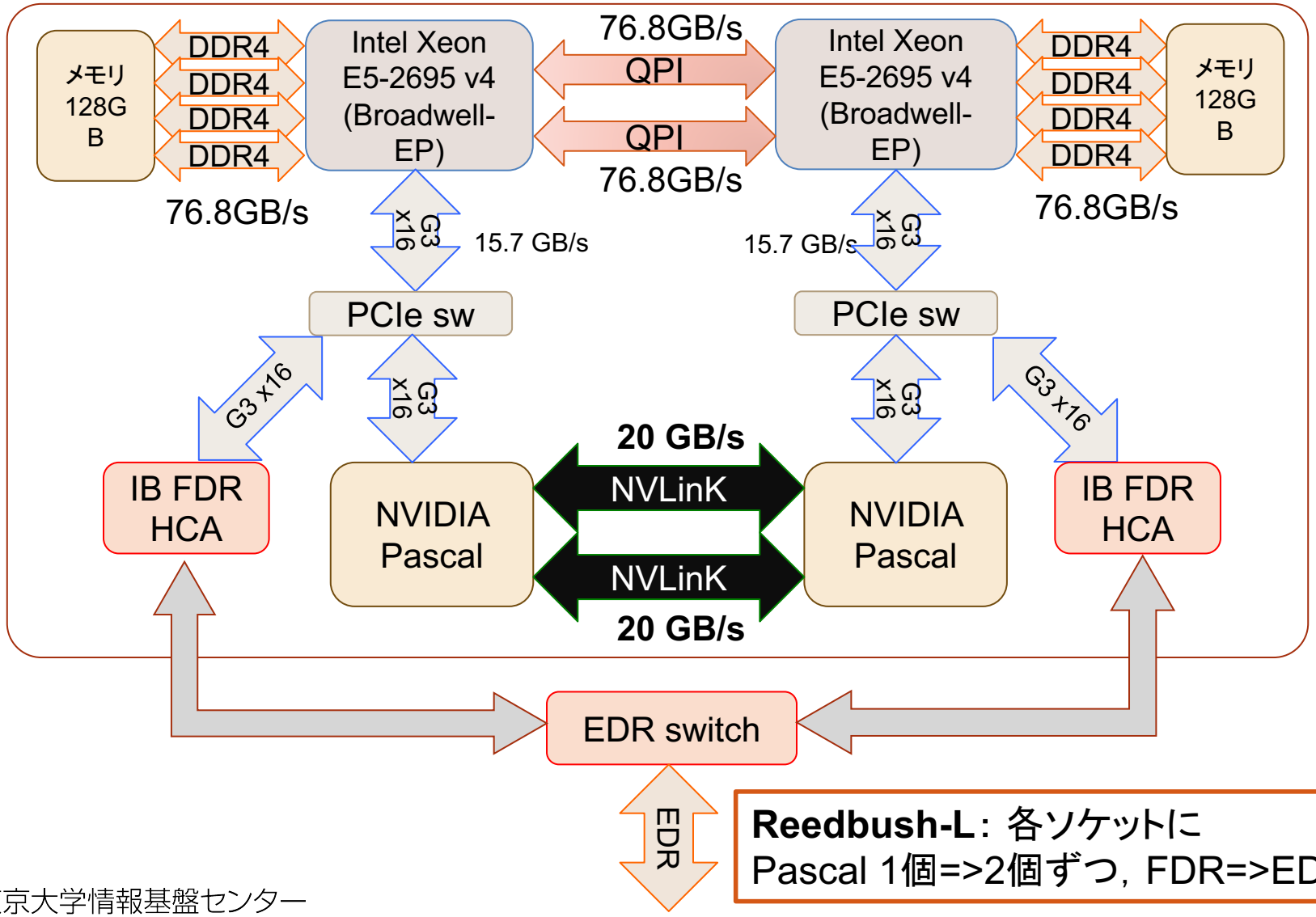
76.8 GB/秒
=(8Byte×2400MHz×4 channel)

# Reedbush-Uの通信網

- フルバイセクションバンド幅を持つFat Tree網
  - どのように計算ノードを選んでも互いに無衝突で通信が可能
- Mellanox InfiniBand EDR 4x CS7500: 648ポート
  - 内部は36ポートスイッチ (SB7800)を (36+18)台組み合わせたものと等価



# Reedbush-Hノードのブロック図



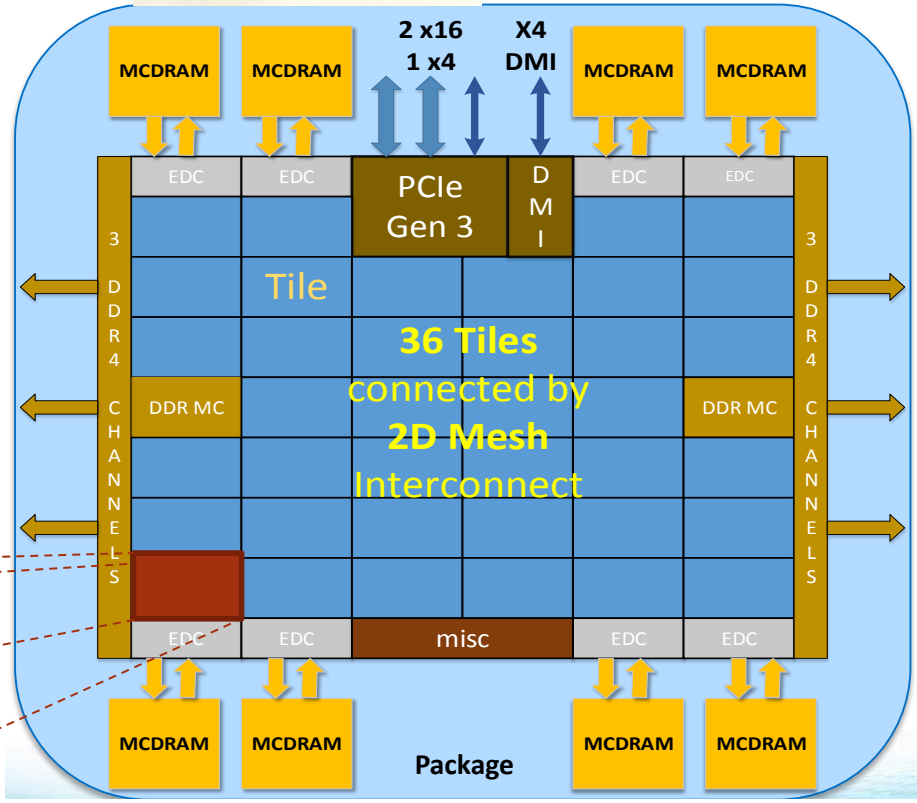

**Reedbush-L**：各ソケットに
Pascal 1個=>2個ずつ，FDR=>EDR

# Oakforest-PACS 計算ノード

- Intel Xeon Phi (Knights Landing)
  - 1ノード1ソケット
- MCDRAM: オンパッケージの高バンド幅メモリ16GB + DDR4メモリ

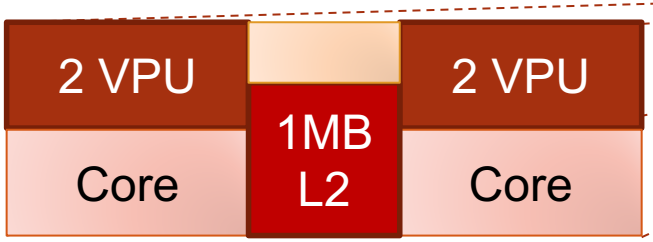


HotChips27
KNLスライドより



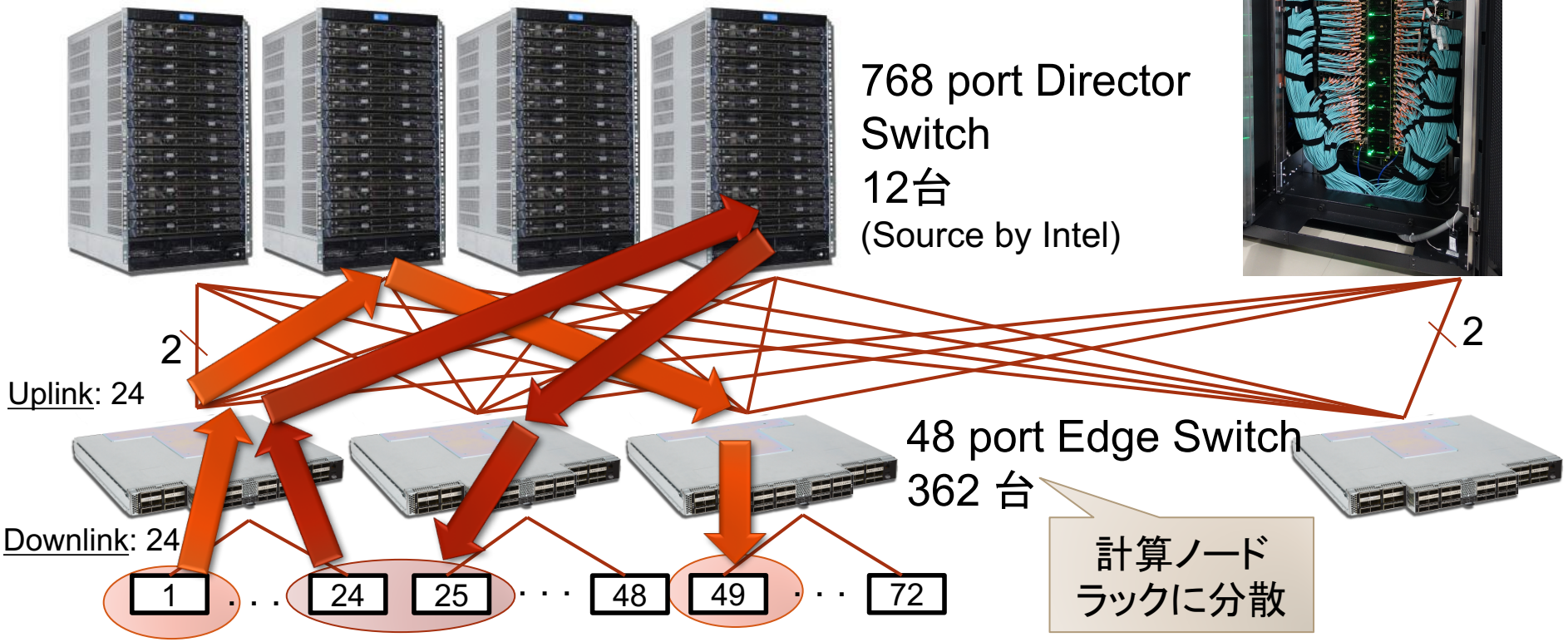

ソケット当たりメモリ量：16GB×6＝96GB

MCDRAM: 490GB/秒以上（実測）
DDR4: 115.2 GB/秒
=(8Byte×2400MHz×6 channel)



# **Oakforest-PACS:** Intel Omni-Path Architecture による フルバイセクションバンド幅Fat-tree網






コストはかかるがフルバイセクションバンド幅を維持
- システム全系使用時にも高い並列性能を実現
- 柔軟な運用：ジョブに対する計算ノード割り当ての自由度が高い

# 東大情報基盤センターOakforest-PACSスーパーコンピュータシステムの料金表（2017年4月1日 ）

- パーソナルコース（年間）
  - コース1： 100,000円 ： 8ノード(基準)、最大16ノードまで
  - コース2： 200,000円 ： 16ノード(基準)、最大64ノードまで
- グループコース
  - 400,000円 (企業 480,000円) ：1口 8ノード（基準）、最大2048ノードまで
- 以上は、「トークン制」で運営
  - 申し込みノード数×360日×24時間の「トークン」が与えられる
  - 基準ノードまでは、トークン消費係数が1.0
  - 基準ノードを超えると、超えた分は、消費係数が2.0になる
  - 大学等のユーザはFX10、Reedbushとの相互トークン移行も可能

# 東大情報基盤センターReedbushスーパーコンピュータシステムの料金表（2017年4月1日 ）

- パーソナルコース（年間）
  - 150,000円 ： RB-U: 4ノード（基準）、最大16ノードまで
    RB-H: 1ノード（基準）、最大2ノードまで
- グループコース
  - 300,000円：1口 4ノード（基準）、最大128ノードまで、
    RB-H：1ノード（基準）、最大32ノードまで（トークン係数はUの2.5倍）
  - RB-Uのみ 企業 360,000円 ：1口 4ノード（基準）、最大128ノードまで
  - RB-Hのみ 企業 216,000円 ：1口 1ノード（基準）、最大32ノードまで
- 以上は、「トークン制」で運営
  - 申し込みノード数×360日×24時間の「トークン」が与えられる
  - 基準ノードまでは、トークン消費係数が1.0
  - 基準ノードを超えると、超えた分は、消費係数が2.0になる
  - 大学等のユーザはFX10, Oakforest-PACSとの相互トークン移行も可能
  - ノード固定もあり
  - RB-Lは詳細検討中

# 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（2017年4月1日）

- パーソナルコース（年間）
  - コース1： 90,000円 ： 12ノード(基準)、最大24ノードまで
  - コース2： 180,000円 ： 24ノード(基準)、最大96ノードまで
- グループコース
  - 360,000円 (企業 432,000円) ：1口、12ノード、最大1440ノードまで
- 以上は、「トークン制」で運営
  - 申し込みノード数×360日×24時間の「トークン」が与えられる
  - 基準ノードまでは、トークン消費係数が1.0
  - 基準ノードを超えると、超えた分は、消費係数が2.0になる
  - 大学等のユーザはReedbush, Oakforest-PACSとの相互トークン移行も可能

# 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

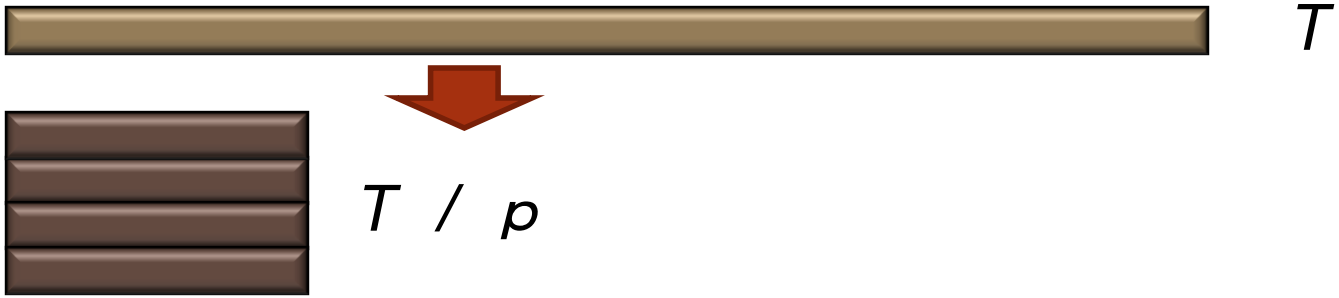
などがご覧になれます。

http://www.cc.u-tokyo.ac.jp/system/ofp/
http://www.cc.u-tokyo.ac.jp/system/reedbush/
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）

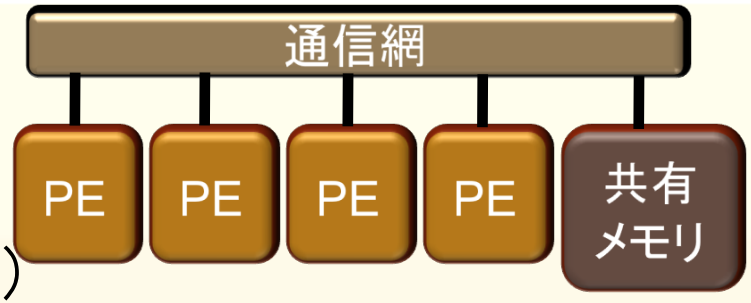
# 並列計算機の分類

- 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

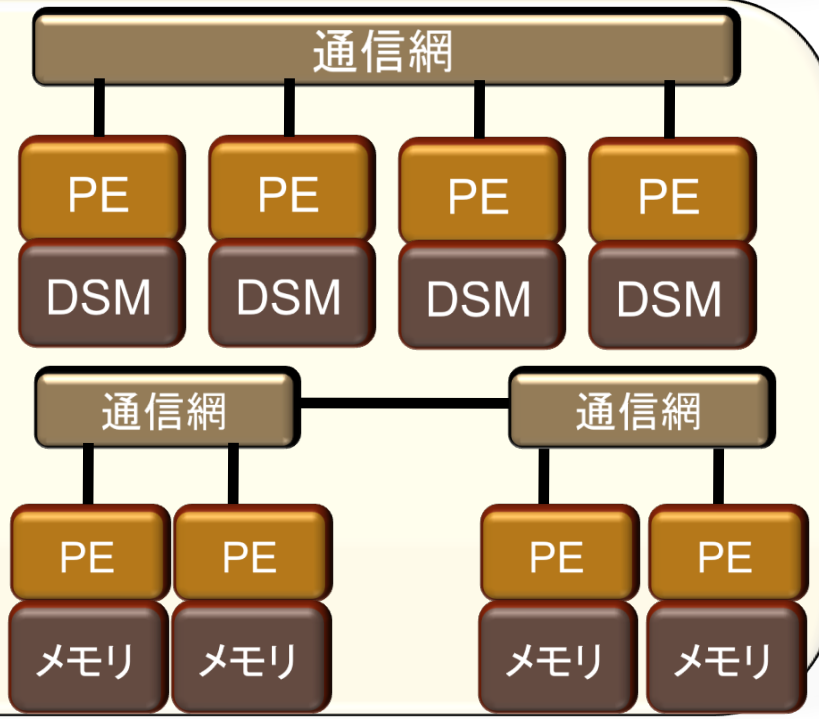
A) メモリアドレスを共有している：互いのメモリがアクセス可能

1. 共有メモリ型
（SMP:
Symmetric Multiprocessor,
UMA: Uniform Memory Access）




2. 分散共有メモリ型
（DSM:
Distributed Shared Memory）

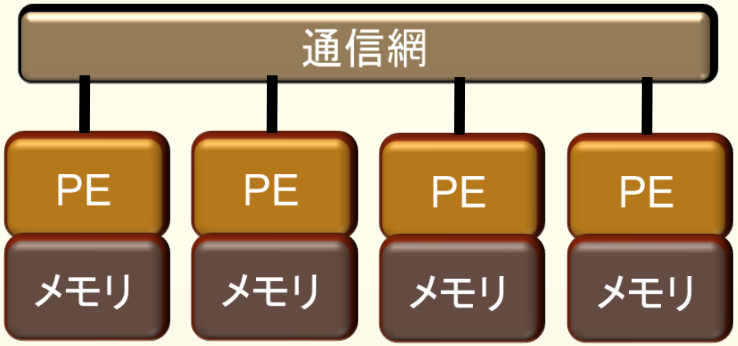
共有・非対称メモリ型
（ccNUMA、
Cache Coherent Non-Uniform
Memory Access）



# 並列計算機のメモリ型による分類

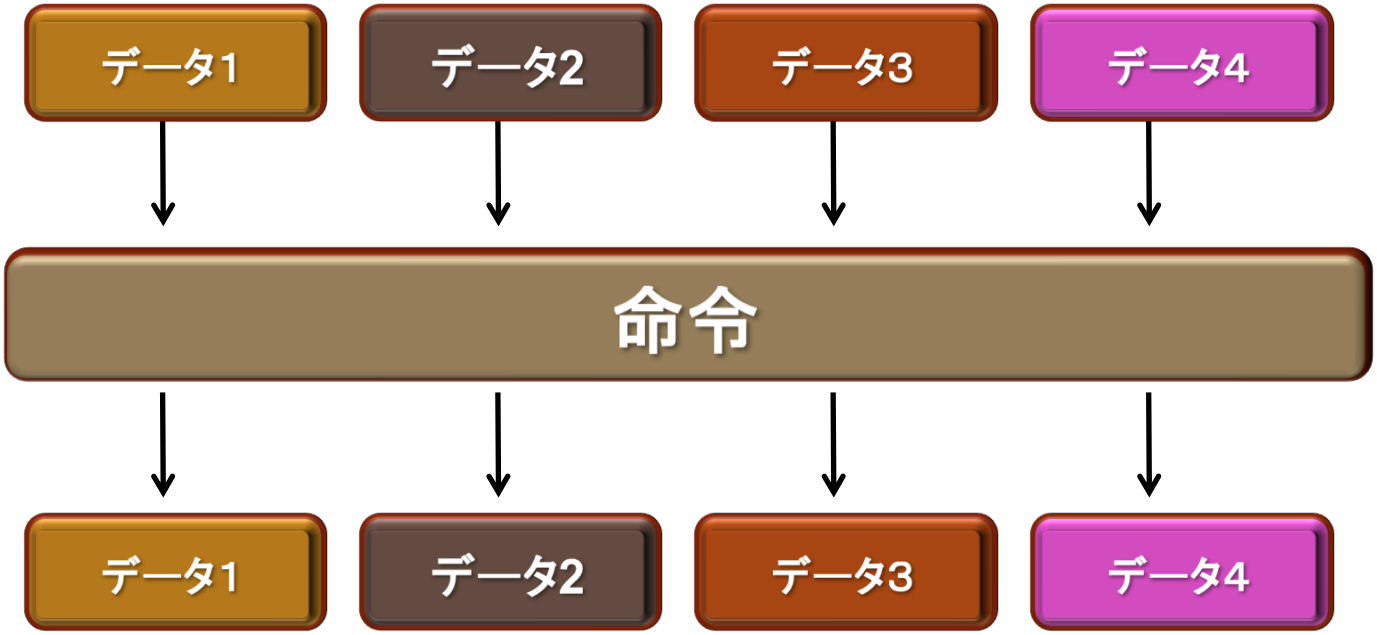
B) メモリアドレスは独立：互いのメモリはアクセス不可

3. 分散メモリ型
（メッセージパッシング）



# 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＞
  - 複雑な挙動は人間には想定し難い



# 並列プログラミングのモデル

- 多くの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
  - OpenMP
    - 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」

マルチプロセスとマルチスレッドは共存可能
→ハイブリッドMPI/OpenMP実行

# 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

SIMDの考え方と同じ

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

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

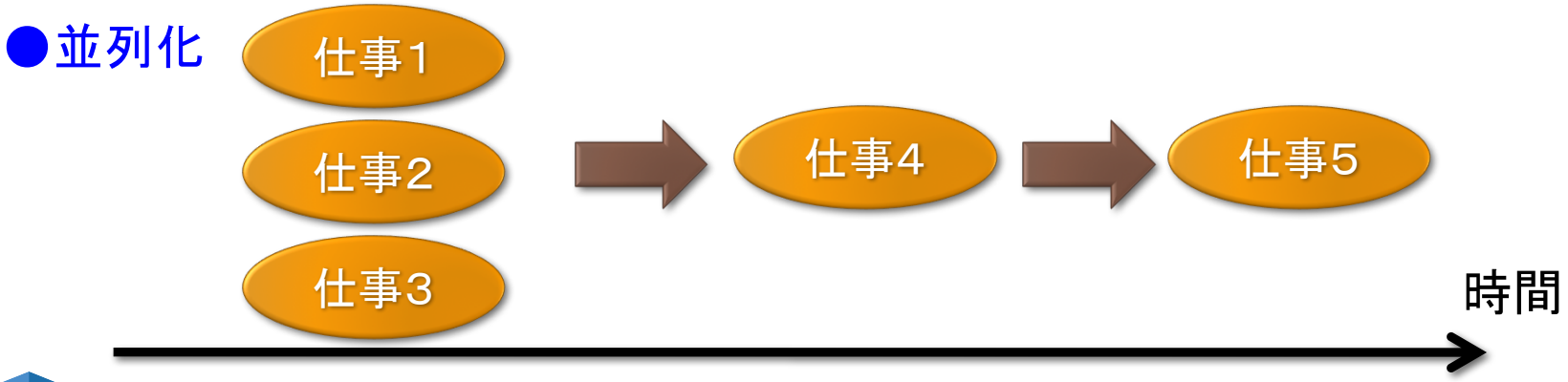
$\begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix} =$

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

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



# 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（Massively Parallel Processing (MPP)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（これまで）

- 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）
  - 2008年5月 1.3版、2008年6月 2.1版
  - 2009年9月 2.2版
    - 日本語版 http://www.pccluster.org/ja/mpi.html
- 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

# MPIの経緯 MPI-3.1

- MPI-3.0 2012年9月
- MPI-3.1 2015年6月
- 以下のページで現状・ドキュメントを公開中
  - http://mpi-forum.org/docs/docs.html
  - http://meetings.mpi-forum.org
  - http://meetings.mpi-forum.org/mpi31-impl-status-Nov15.pdf
- 注目すべき機能
  - ノン・ブロッキング集団通信機能（MPI_IALLREDUCE、など）
  - 高性能な片方向通信（RMA、Remote Memory Access)
  - Fortran2008 対応、など

# MPIの経緯 MPI-4.0策定中

- 以下のページで経緯・ドキュメントを公開
  - http://meetings.mpi-forum.org/MPI_4.0_main_page.php
- 検討されている機能
  - ハイブリッドプログラミングへの対応
  - MPIアプリケーションの耐故障性（Fault Tolerance, FT）
  - いくつかのアイデアを検討中
    - Active Messages (メッセージ通信のプロトコル)
      - 計算と通信のオーバラップ
      - 最低限の同期を用いた非同期通信
      - 低いオーバーヘッド、パイプライン転送
      - バッファリングなしで、インタラプトハンドラで動く
    - Stream Messaging
    - 新プロファイル・インターフェース

# MPIの実装

- MPICH（エム・ピッチ）
  - 米国アルゴンヌ国立研究所が開発
- MVAPICH (エムヴァピッチ)
  - 米国オハイオ州立大学で開発、MPICHをベース
  - InfiniBand向けの優れた実装
- OpenMPI
  - オープンソース
- ベンダMPI
  - 大抵、上のどれかがベースになっている
    例: 富士通「京」、FX10用のMPI: Open-MPIベース
    Intel MPI: MPICH、MVAPICHベース
  - 注意点:メーカ独自機能拡張がなされていることがある

# MPIによる通信

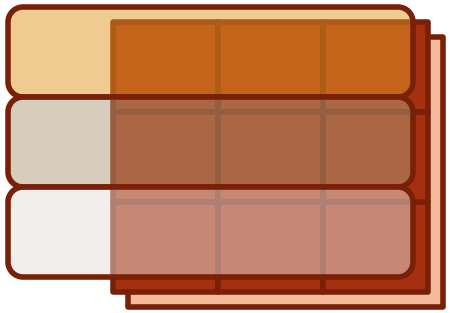
- 郵便物の郵送に同じ
- 郵送に必要な情報：
  1. 自分の住所、送り先の住所
  2. 中に入っているものはどこにあるか
  3. 中に入っているものの分類
  4. 中に入っているものの量
  5. （荷物を複数同時に送る場合の）認識方法（タグ）
- MPIでは：
  1. 自分の認識ID、および、送り先の認識ID
  2. データ格納先のアドレス
  3. データ型
  4. データ量
  5. タグ番号

# 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

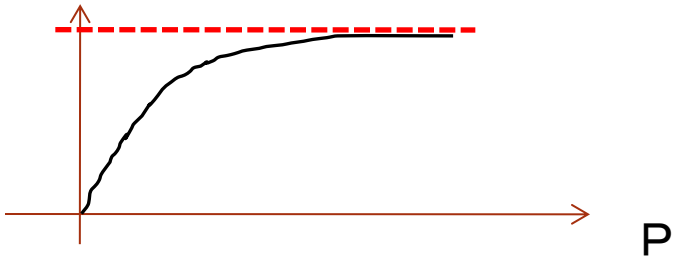


# 性能評価指標

## 並列化の尺度

# 性能評価指標ー台数効果

- 台数効果 $S_P = T_S / T_P \quad (0 \le S_p)$
  - 式：
  - $T_S$ ：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： $E_P = S_P / P \times 100 \quad (0 \le E_p)\ [\%]$
- 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。

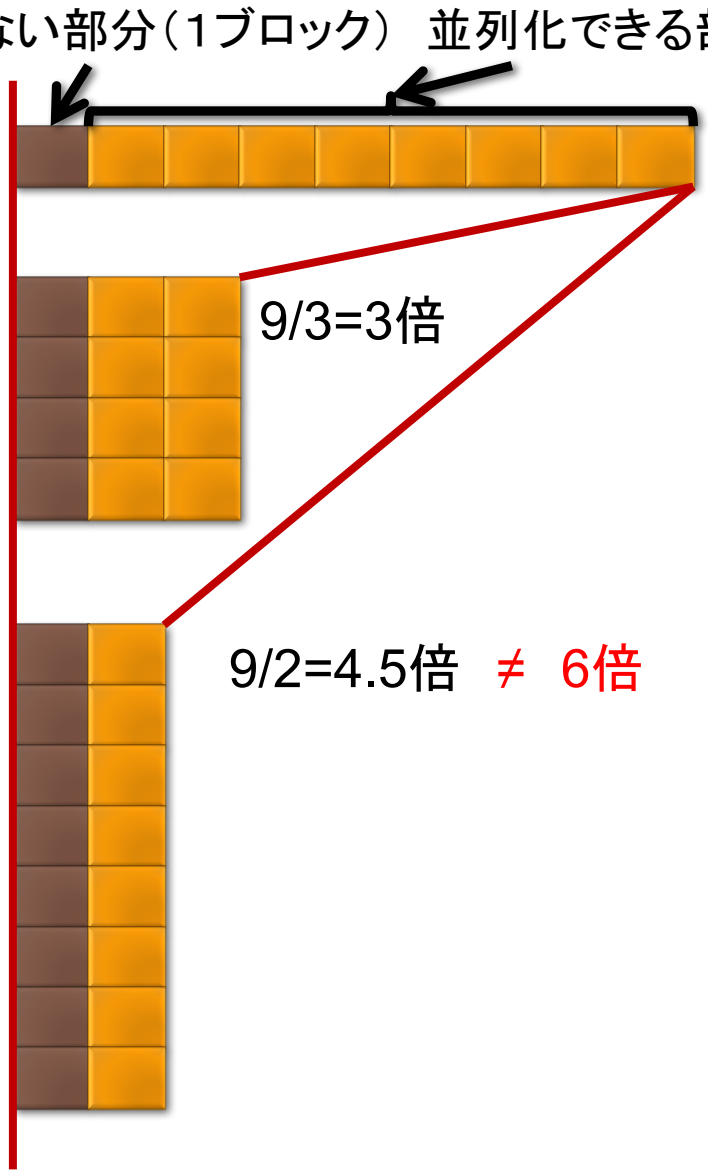
$$S_P = K/(K\alpha/P + K(1-\alpha))$$

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

- 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



# 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）

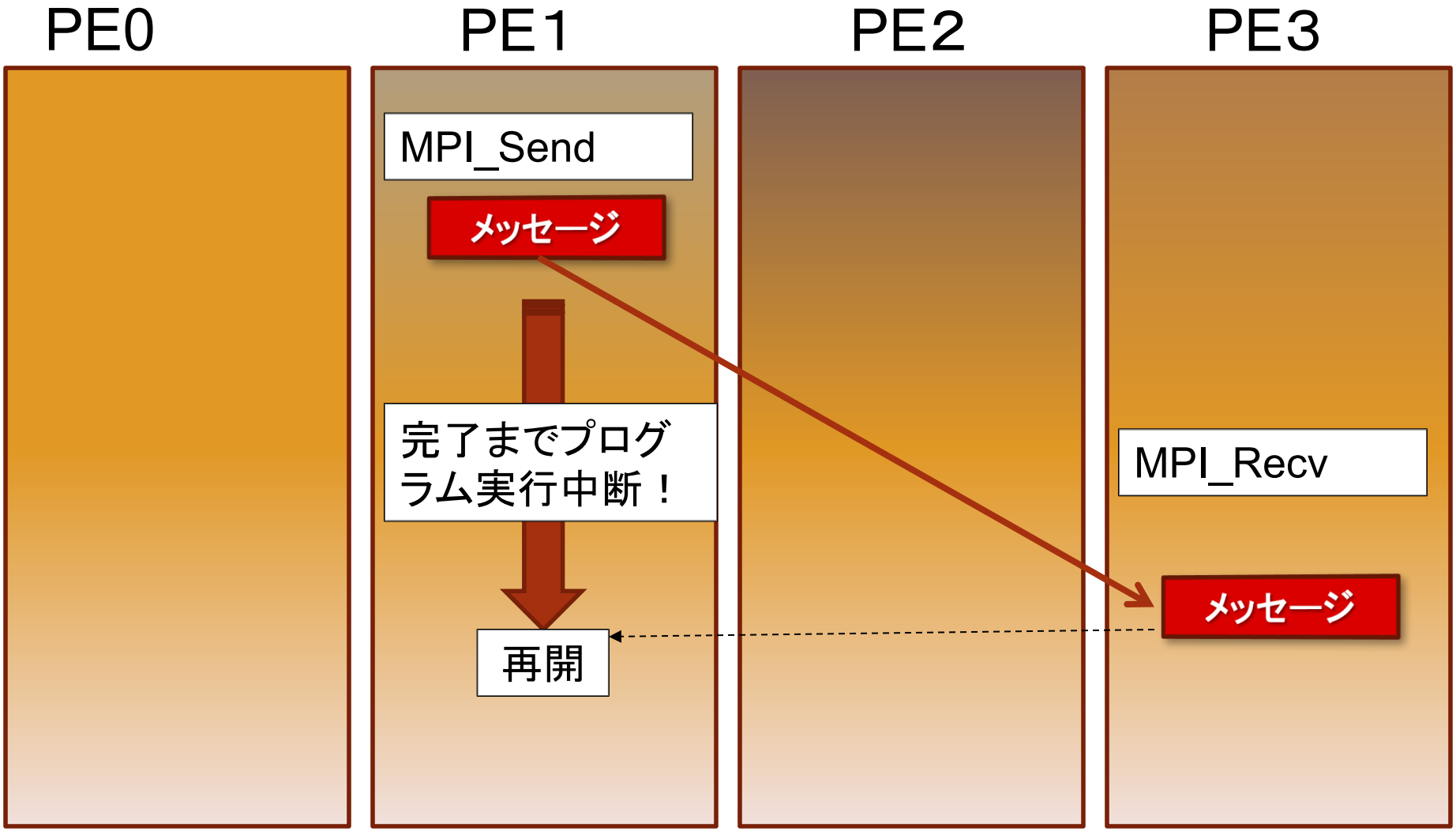
- itag： 整数型。受信したいメッセージに付いているタグの値を指定。
  - 任意のタグ値のメッセージを受信したいときは、MPI_ANY_TAG を指定。
- icomm： 整数型。PE集団を認識する番号であるコミュニケータを指定。
  - 通常ではMPI_COMM_WORLD を指定すればよい。
- istatus： MPI_Status型（整数型の配列）。受信状況に関する情報が入る。かならず専用の型宣言をした配列を確保すること。
  - 要素数がMPI_STATUS_SIZEの整数配列が宣言される。
  - 受信したメッセージの送信元のランクが istatus[MPI_SOURCE]、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

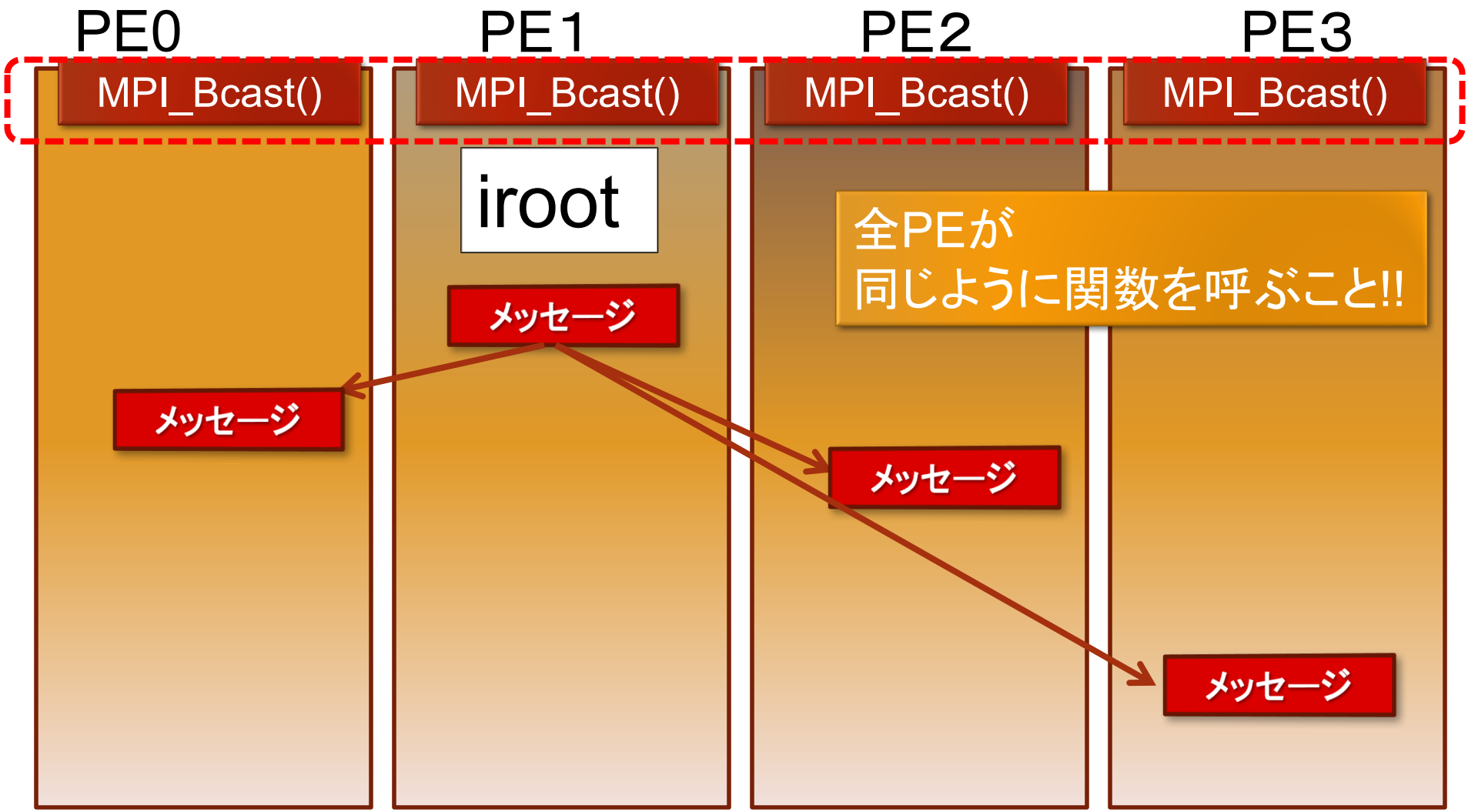


# 基礎的なMPI関数—MPI_Bcast

- 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。

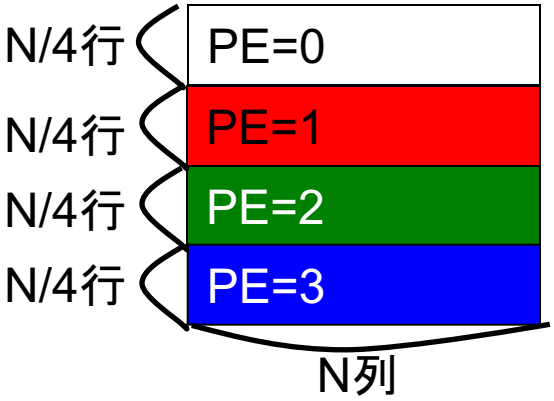
# MPI_Bcastの概念（集団通信）



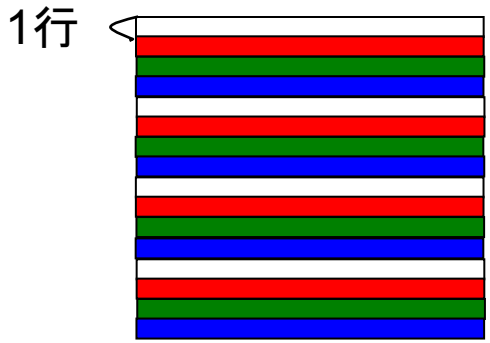
# 基本演算

- 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

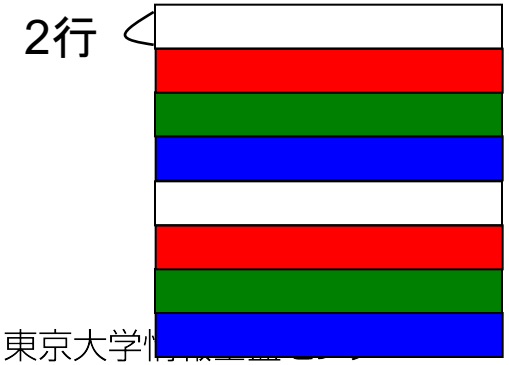
# １次元分散




- （行方向）ブロック分割方式
- (Block, *) 分散方式



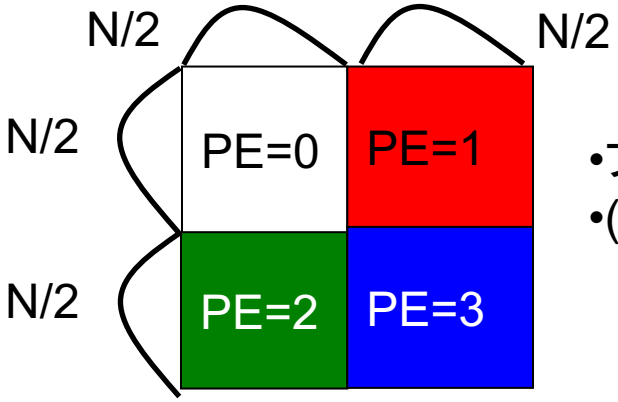

- (行方向) サイクリック分割方式
- (Cyclic, *) 分散方式




- (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

# 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 |

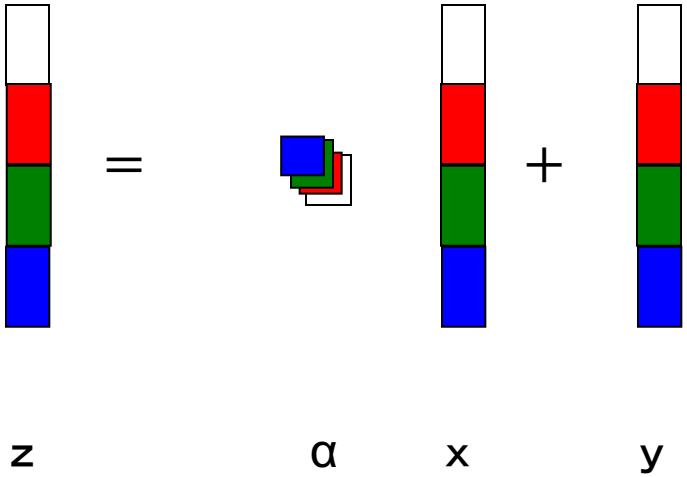
- 二次元ブロック・サイクリック分割方式
- (Cyclic(2), Cyclic(2))分散方式

# ベクトルどうしの演算

- 以下の演算

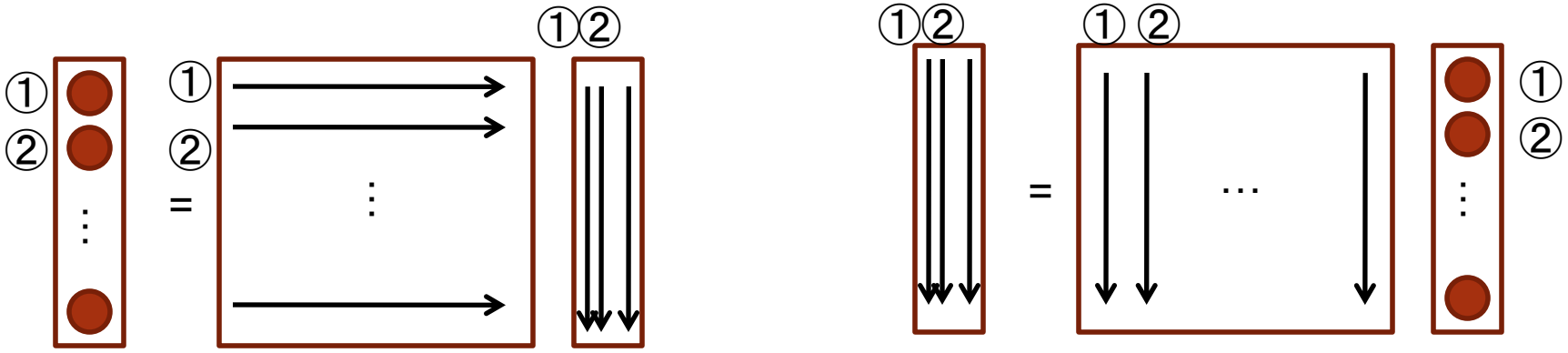
$$z = a\,x + y$$

  - 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



# 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

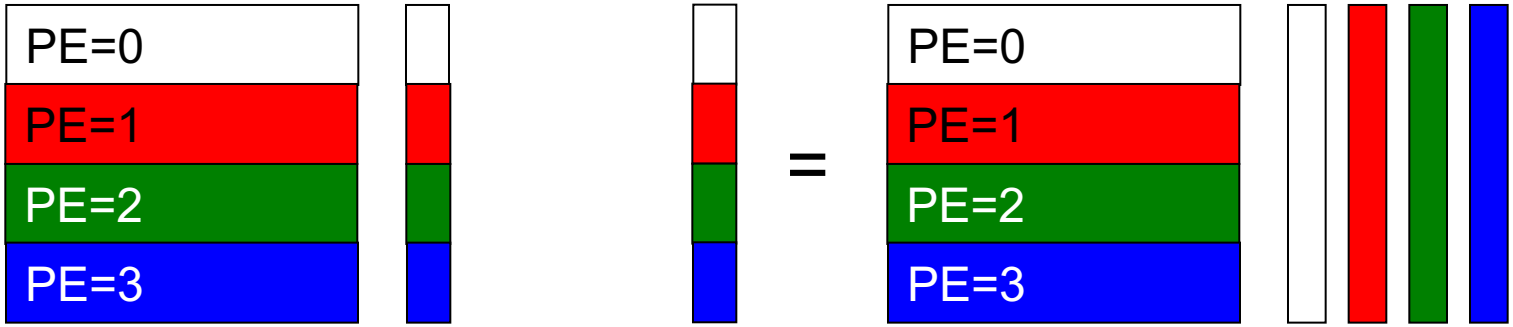
```
for(j=0; j<n; j++) y[j]=0.0;
for(j=0; j<n; j++) {
   for (i=0; i<n; i++) {
      y[i] += a[i][j]*x[j];
   }
}
```

＜列方式＞： Fortran言語向き

# 行列とベクトルの積

## ＜行方式の場合＞

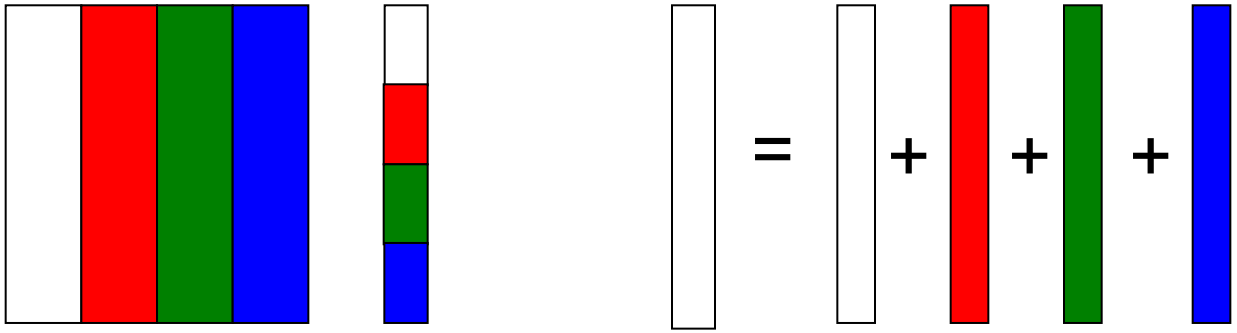
＜行方向分散方式＞ ：行方式に向く分散方式




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



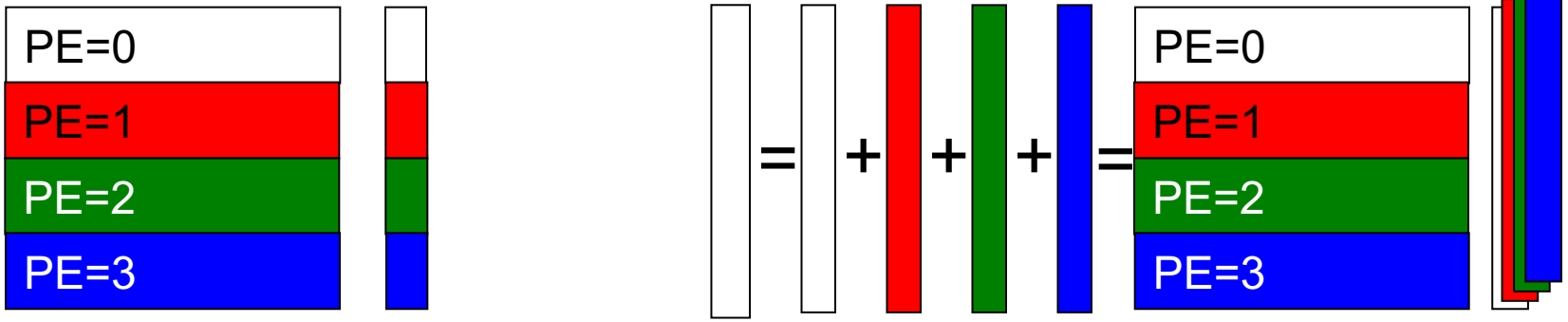

各PE内で行列-ベクトル積を行う

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）

# 行列とベクトルの積

## ＜列方式の場合＞

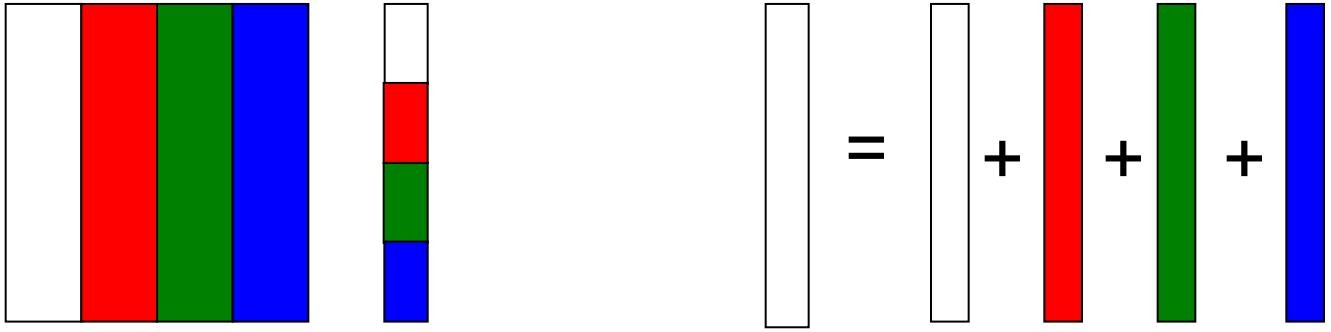
＜行方向分散方式＞ ：無駄が多く使われない




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

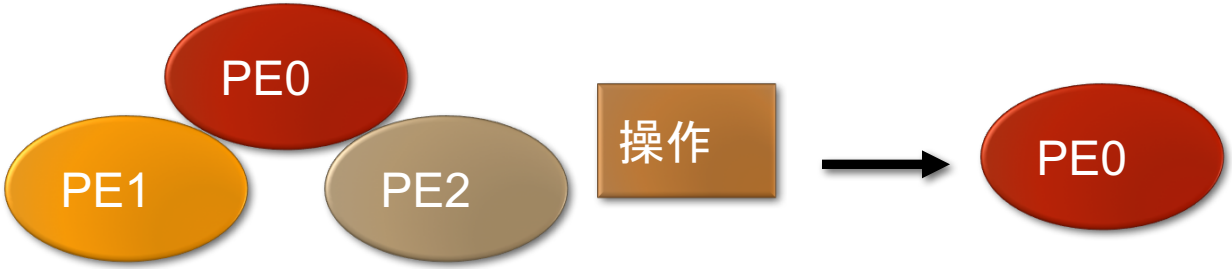
MPI_Reduce関数で総和を求める
（※ある1PEにベクトルすべてが集まる）

# リダクション演算

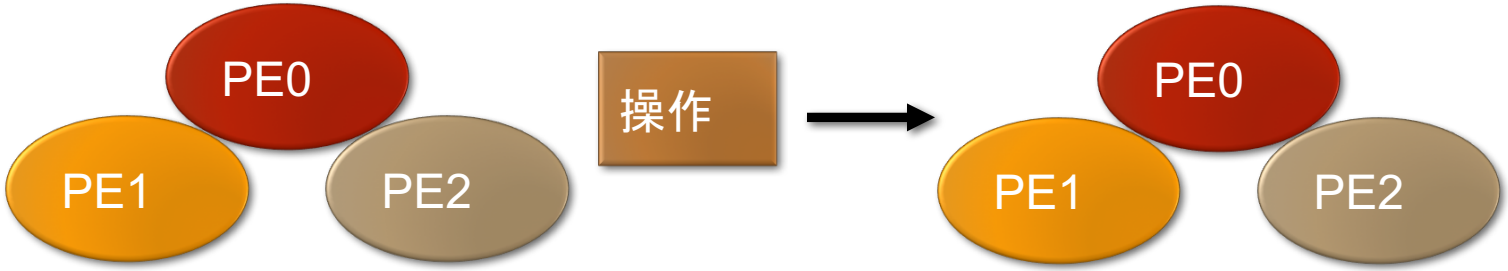
- ＜操作＞によって＜次元＞を減少（リダクション）させる処理
  - 例： 内積演算
    ベクトル（n次元空間） → スカラ（1次元空間）
- リダクション演算は、通信と計算を必要とする
  - 集団通信演算（collective communication operation）
    と呼ばれる
- 演算結果の持ち方の違いで、2種のインタフェースが存在する

# 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



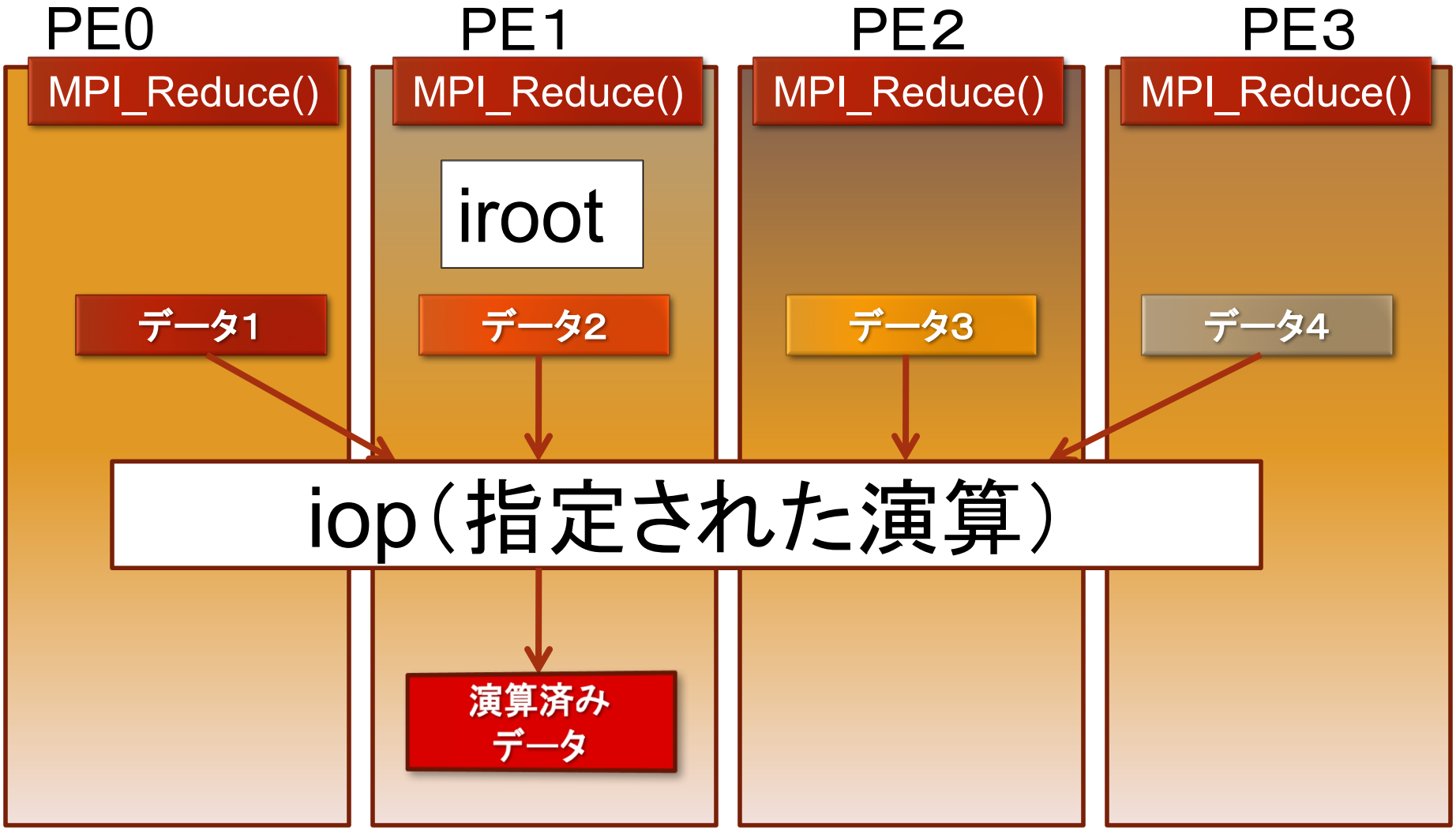
# 基礎的なMPI関数—MPI_Reduce

- 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。
    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。
  - 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。
    - （Fortran）＜最小／最大値と位置＞を返す演算を指定する場合は、MPI_2INTEGER(整数型)、 MPI_2REAL (単精度型)、MPI_2DOUBLE_PRECISION(倍精度型) 、を指定する。

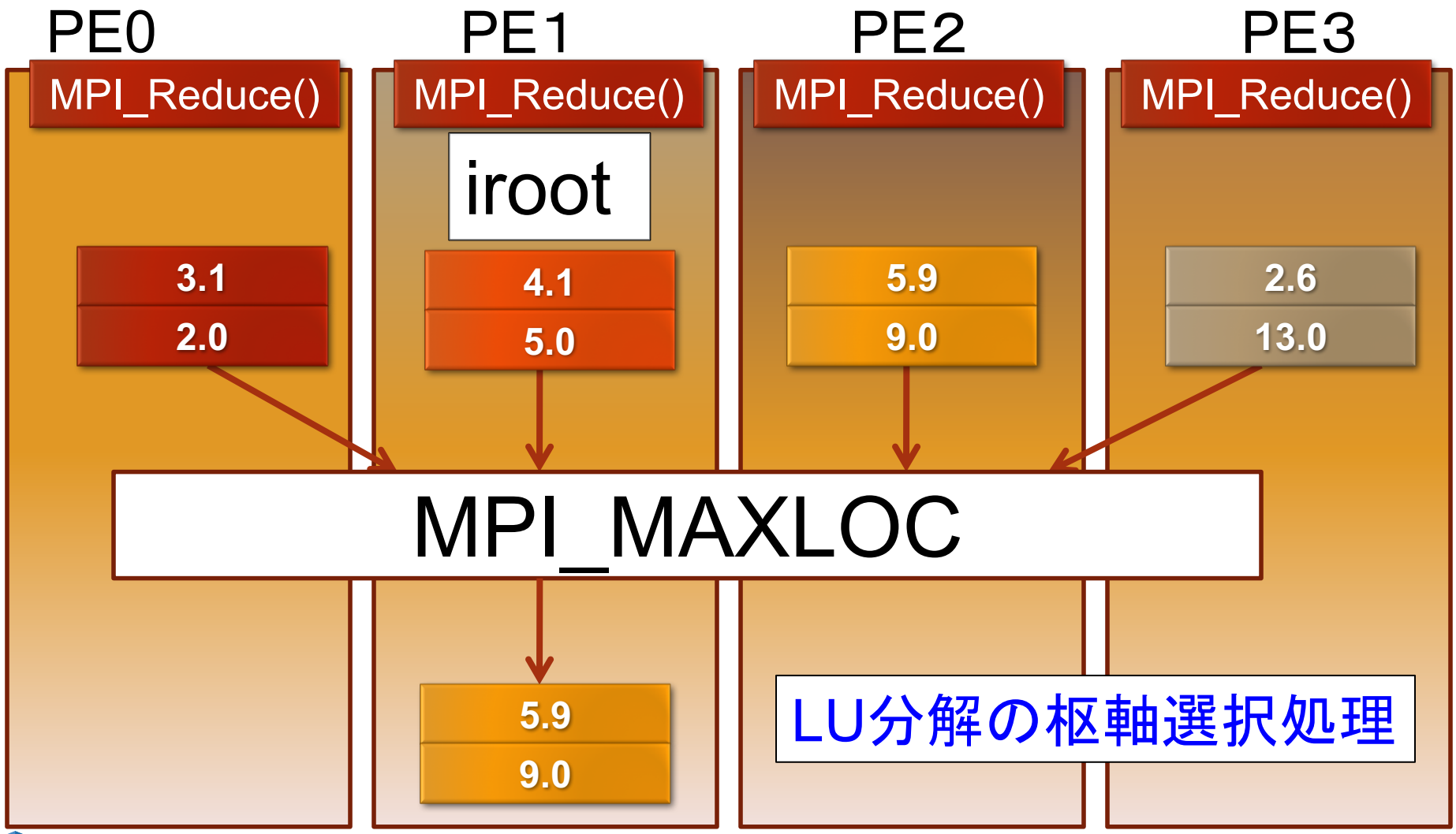
# 基礎的なMPI関数—MPI_Reduce

- 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 (最大とその位置)、MPI_MINLOC (最小とその位置) 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_PRECISIONとMPI_MAXLOC）



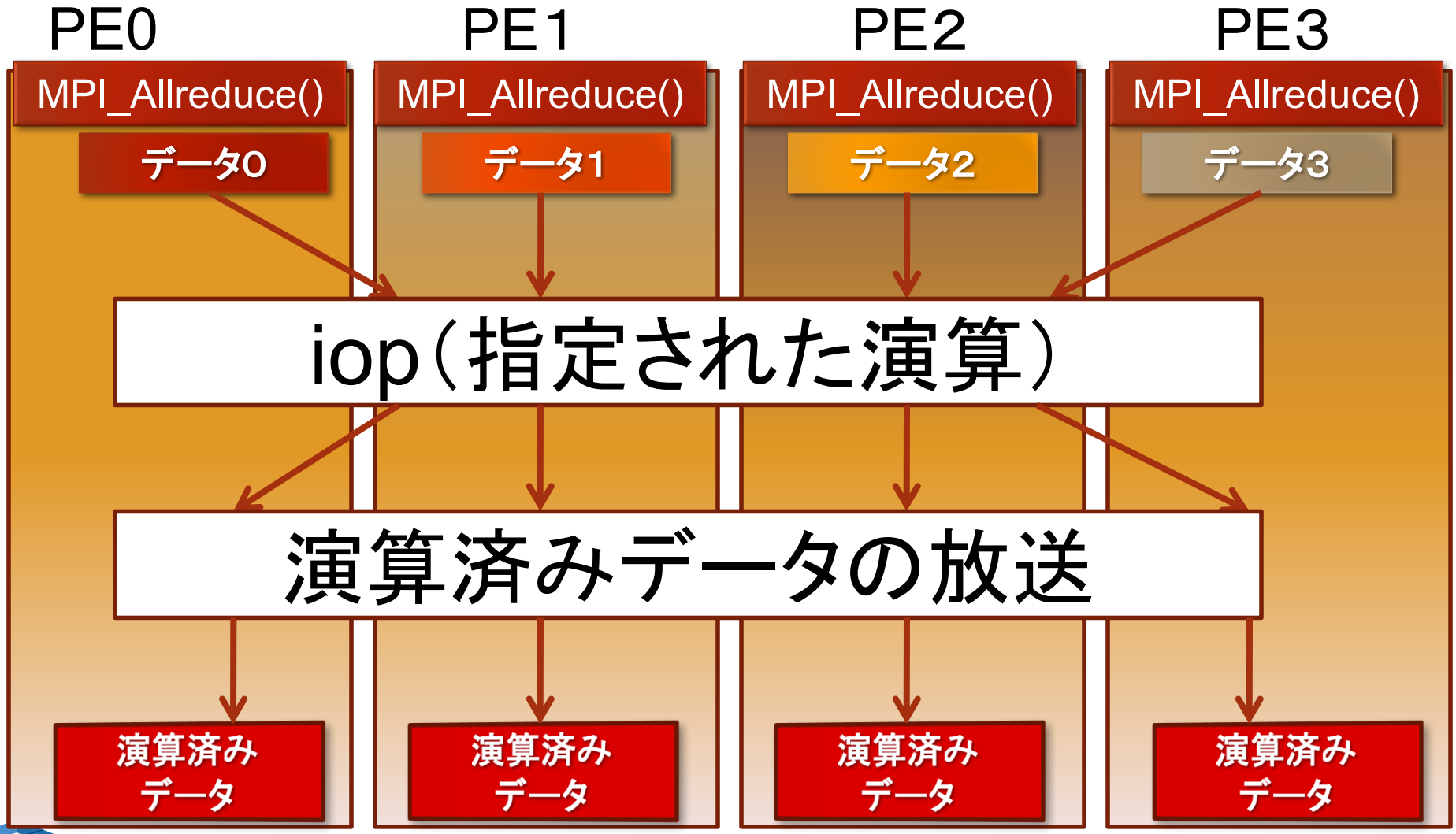
# 基礎的なMPI関数—MPI_Allreduce

- 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。
    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。
  - 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）

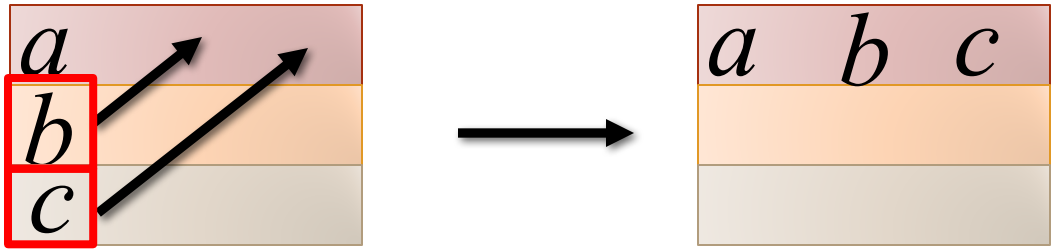


# リダクション演算

- 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。

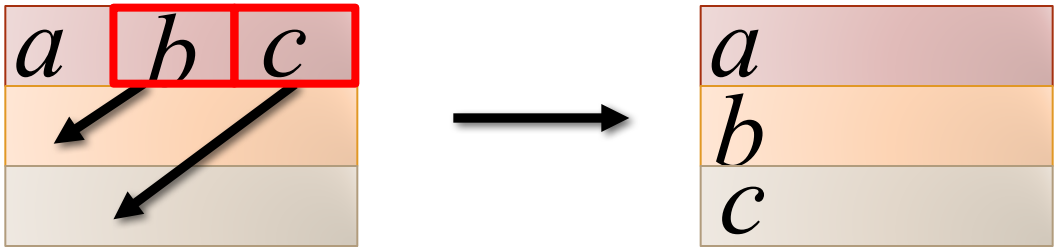
# 行列の転置

- 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




集めるメッセージサイズが各PEで均一のとき使う

  - MPI_Scatter関数




集めるサイズが各PEで均一でないときは：
**MPI_GatherV関数**
**MPI_ScatterV関数**

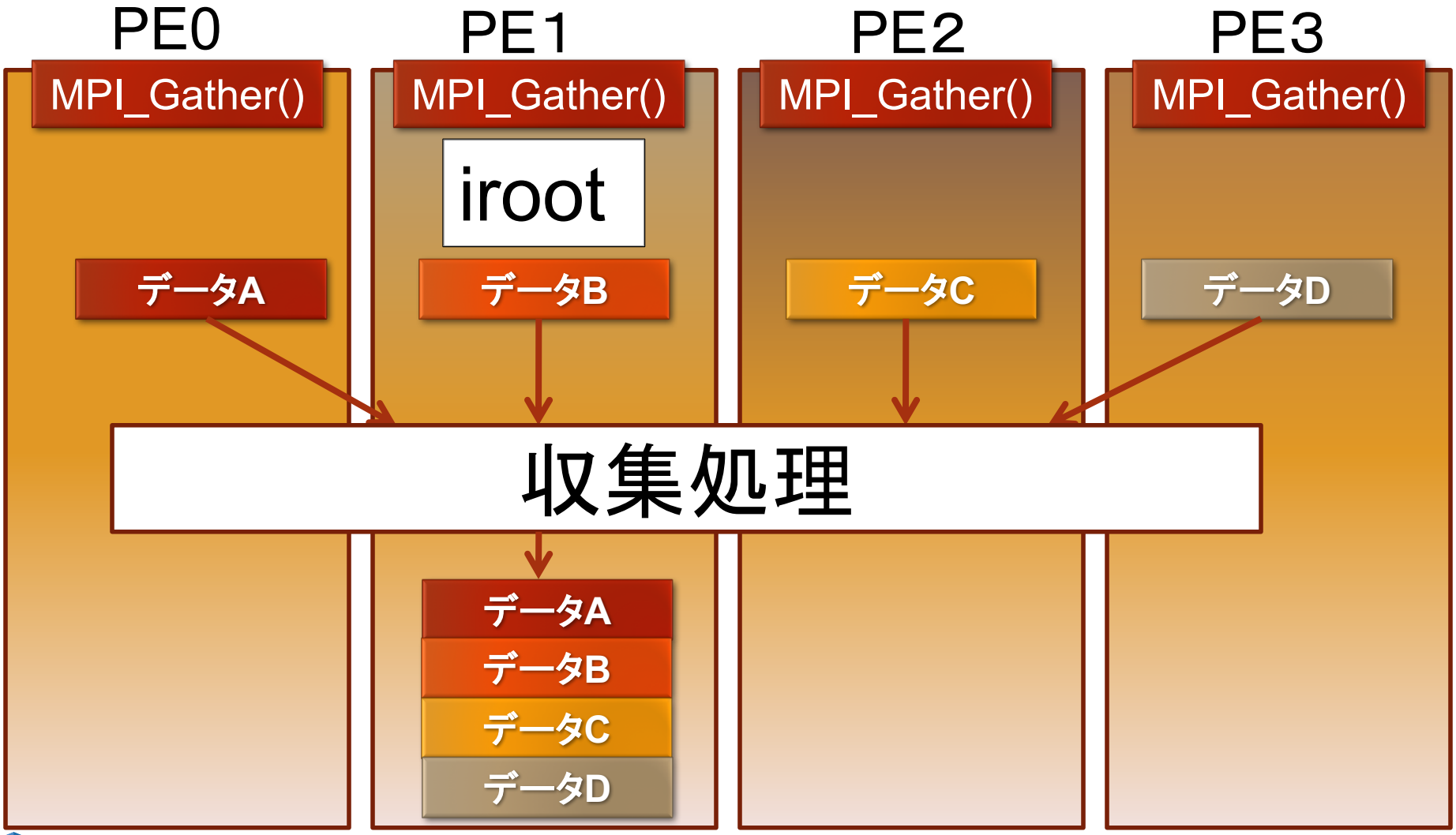
# 基礎的なMPI関数—MPI_Gather

- 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）



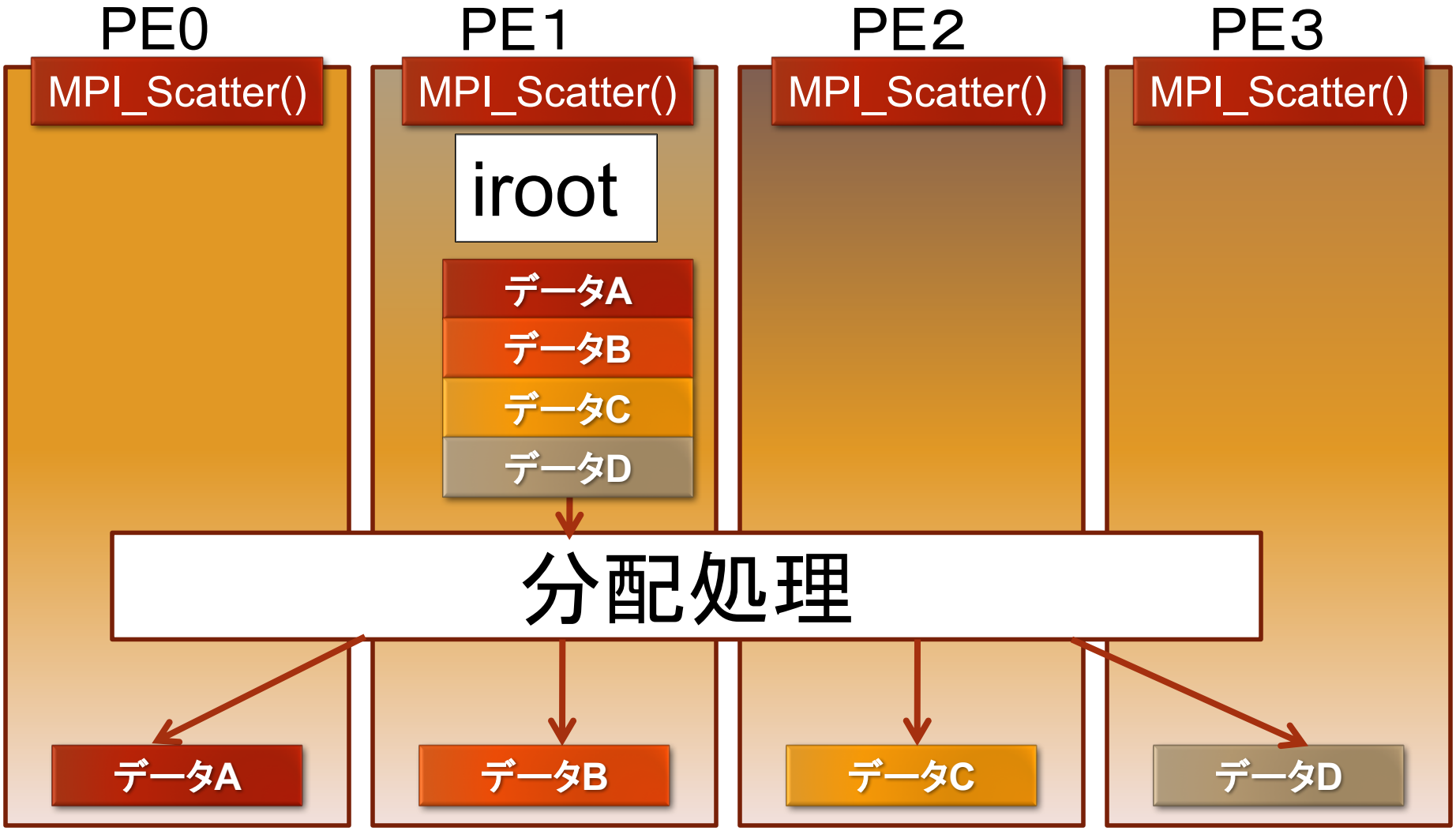
# 基礎的なMPI関数—MPI_Scatter

- 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。

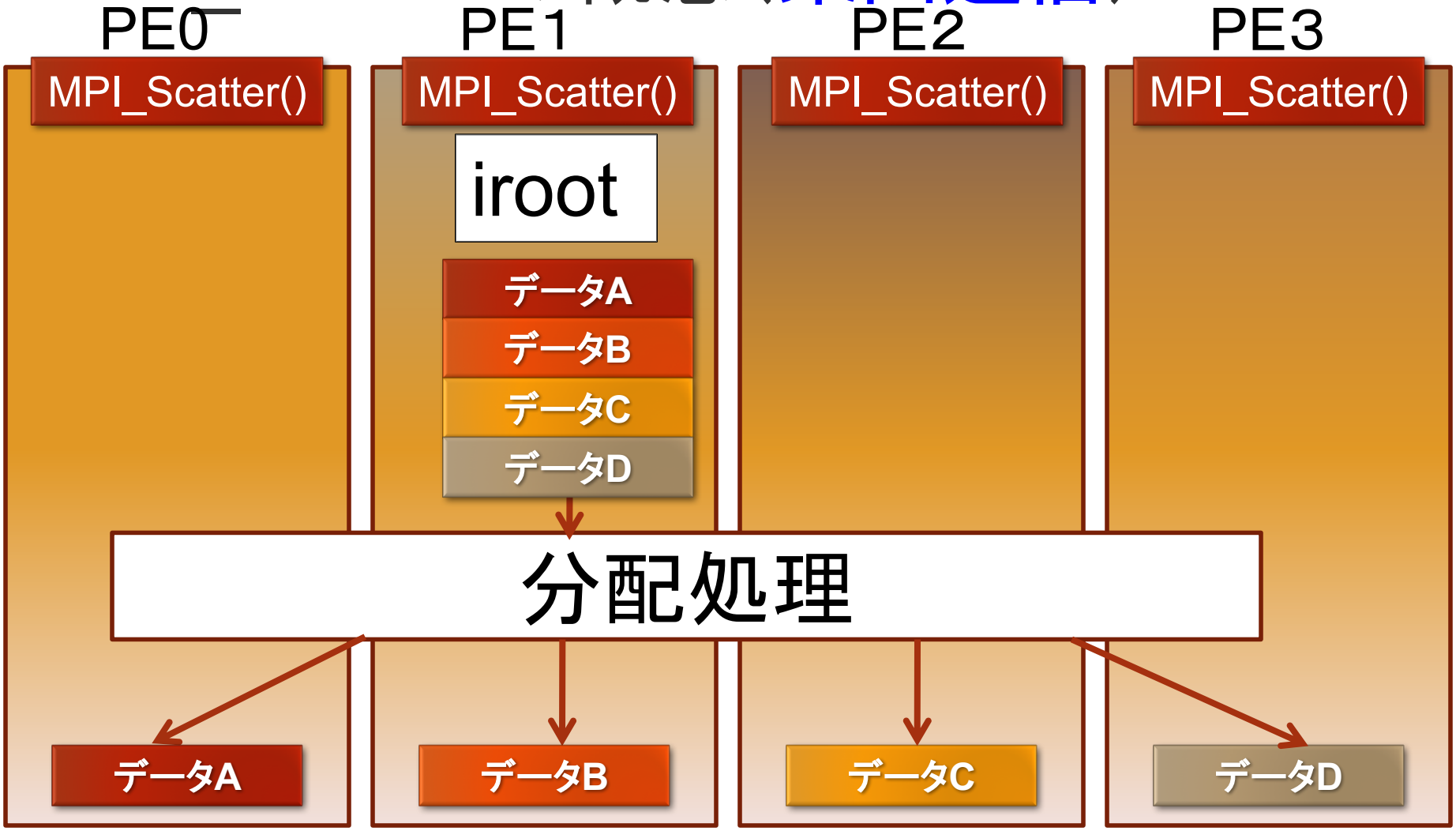
# 基礎的なMPI関数—MPI_Scatter

- 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）



# MPI_Scatterの概念（集団通信）



# ブロッキング、ノンブロッキング

## 1. ブロッキング

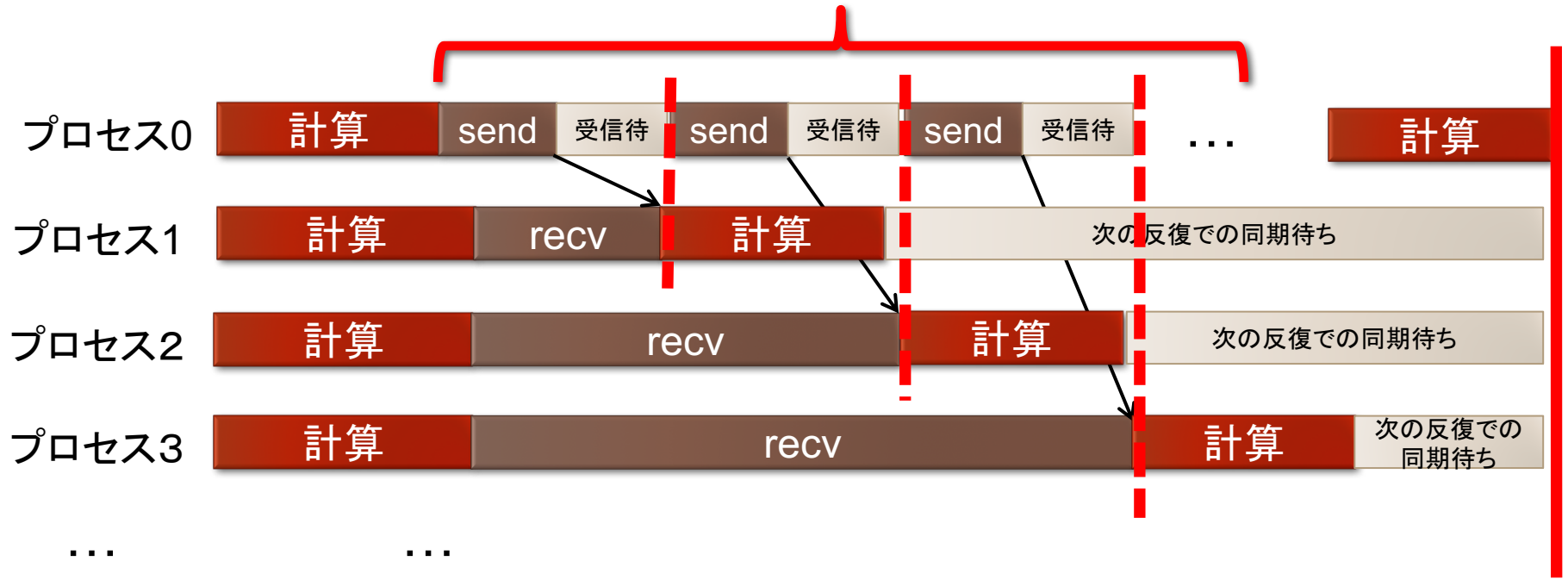
- 送信／受信側のバッファ領域にメッセージが格納され、受信／送信側のバッファ領域が自由にアクセス・上書きできるまで、呼び出しが戻らない
- バッファ領域上のデータの一貫性を保障
- MPI_Send, MPI_Bcastなど

## 2. ノンブロッキング

- 送信／受信側のバッファ領域のデータを保障せずすぐに呼び出しが戻る
- バッファ領域上のデータの一貫性を保障せず
  - 一貫性の保証はユーザの責任

# ブロッキング通信で効率の悪い例

- プロセス0が必要なデータを持っている場合
  連続するsendで、効率の悪い受信待ち時間が多発



# ノンブロッキング通信関数

- ierr = MPI_Isend(sendbuf, icount, datatype, idest, itag, icomm, irequest);

  - 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
  - datatype： 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

# ノンブロッキング通信関数

- 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。
- irequest： MPI_Request型（整数型の配列）。送信を要求したメッセージにつけられた識別子が戻る。
- ierr： 整数型。エラーコードが入る。

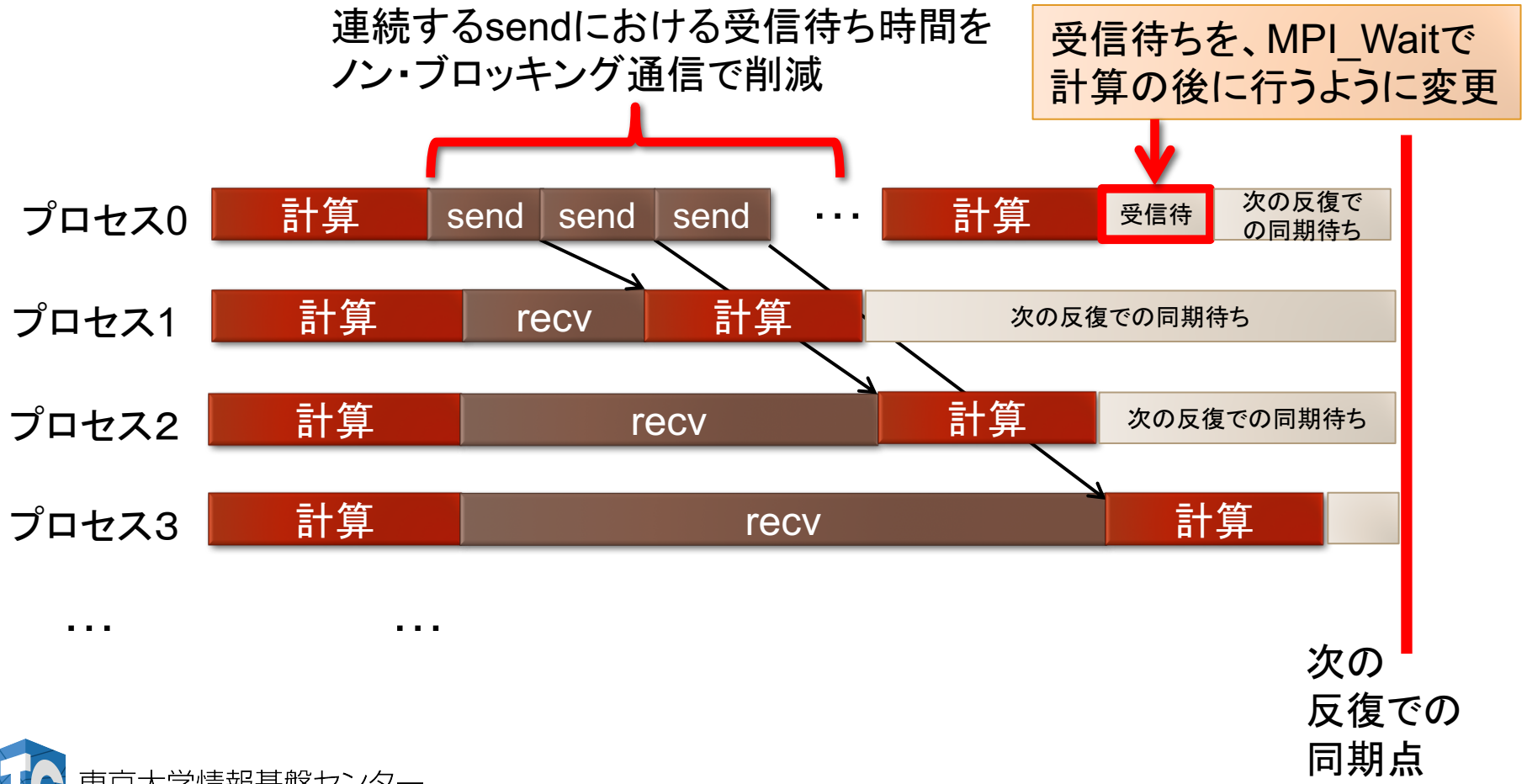
# 同期待ち関数

- ierr = MPI_Wait(irequest, istatus);

  - irequest： MPI_Request型（整数型配列）。
    送信を要求したメッセージにつけられた識別子。
  - istatus： MPI_Status型（整数型配列）。
    受信状況に関する情報が入る。
    - 要素数がMPI_STATUS_SIZEの整数配列を宣言して指定する。
    - 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。
  - 送信データを変更する前・受信データを読み出す前には必ず呼ぶこと

# ノン・ブロッキング通信による改善

- プロセス0が必要なデータを持っている場合



# 注意点

- 以下のように解釈してください：
  - MPI_Send関数
    - 関数中にMPI_Wait関数が入っている；
  - MPI_Isend関数
    - 関数中にMPI_Wait関数が入っていない；
    - かつ、すぐにユーザプログラム戻る；

# 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　塙　敏博

# 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-hy??.bash：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 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 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 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 ジョブスクリプトファイル

# ハイブリッド版Helloプログラム

1. ハイブリッドMPI用の Makefile をコピーする。
   $ cp Makefile_hy36  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-hy36.bash）のキュー名を変更する。“u-lecture”　→ “u-tutorial”に変更する。
   $  emacs hello-hy36.bash

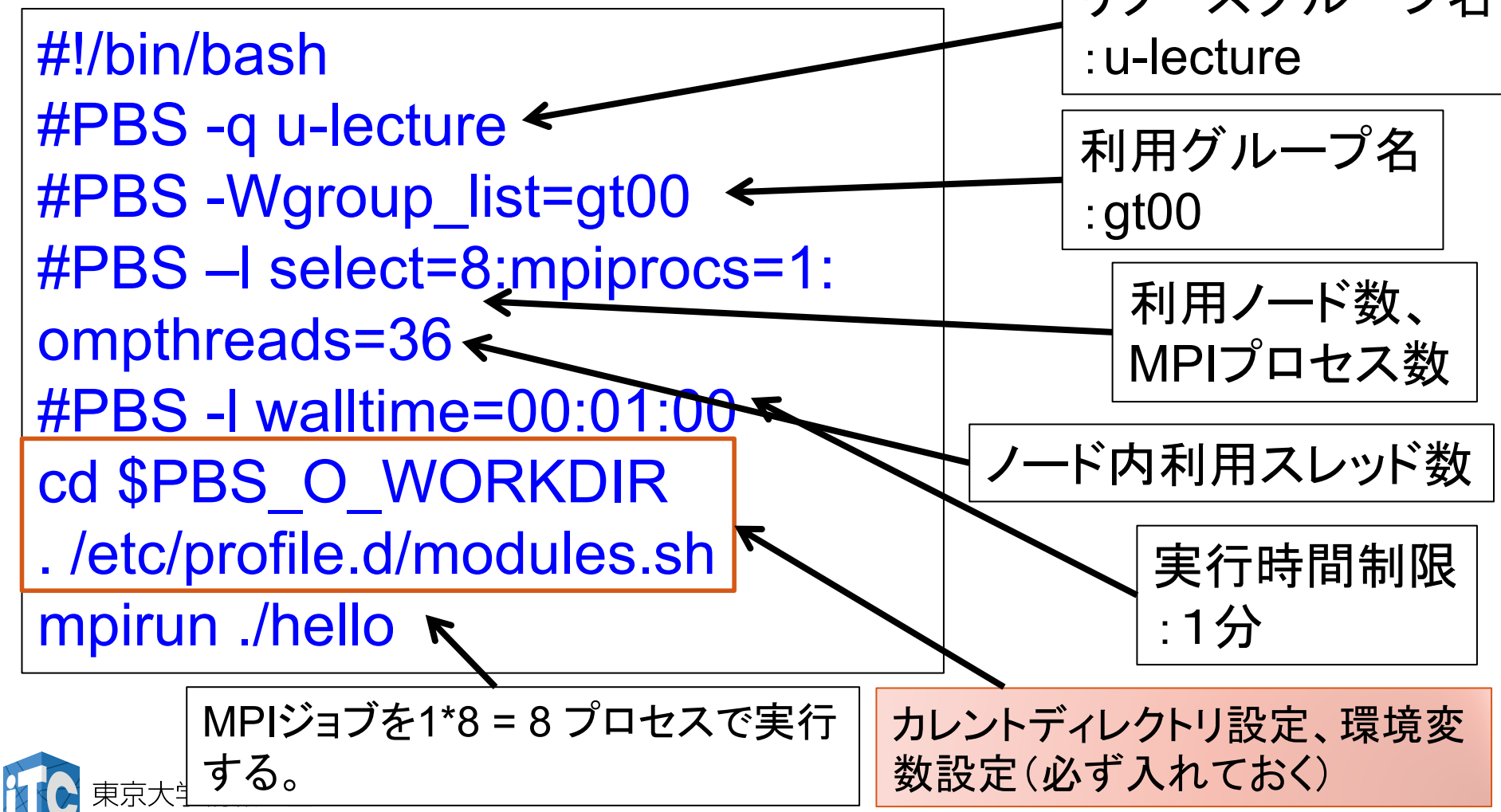
# 並列版Helloプログラムを実行しよう（ハイブリッドMPI）

1. Helloフォルダ中で以下を実行する
   $ qsub hello-hy36.bash
2. 自分の導入されたジョブを確認する
   $ rbstat
3. 実行が終了すると、以下のファイルが生成される
   hello-hy36.bash.eXXXXXX
   hello-hy36.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-hy36.bash.oXXXXXX
5. “Hello parallel world!”が、
   1プロセス*8ノード*36スレッド=288 個表示されていたら成功。

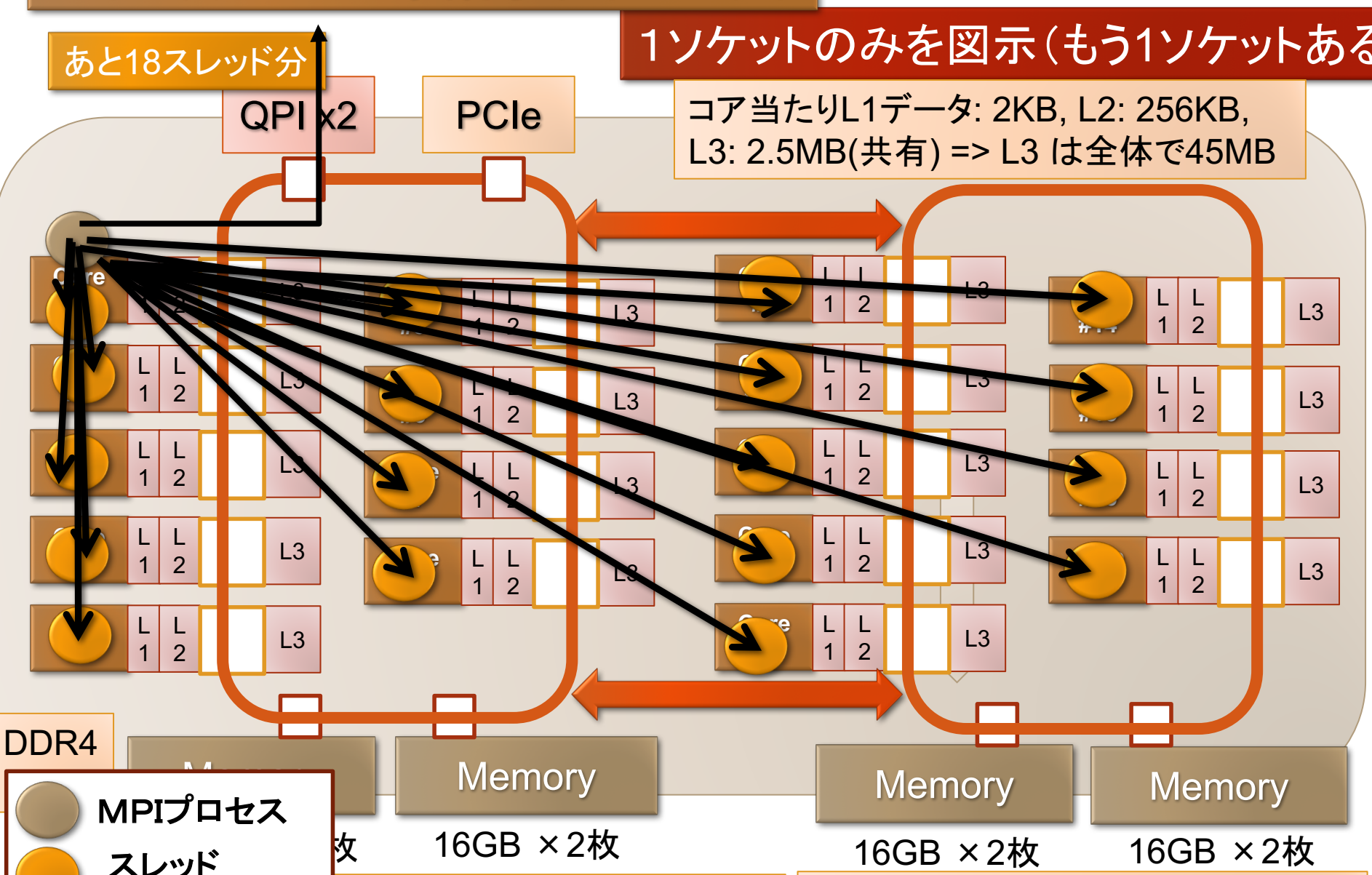
# JOBスクリプトサンプルの説明（OpenMP+MPIハイブリッド）

（hello-hy36.bash, C言語、Fortran言語共通）

```
#!/bin/bash
#PBS -q u-lecture
#PBS -Wgroup_list=gt00
#PBS –l select=8:mpiprocs=1:
ompthreads=36
#PBS -l walltime=00:01:00
cd $PBS_O_WORKDIR
. /etc/profile.d/modules.sh
mpirun ./hello
```



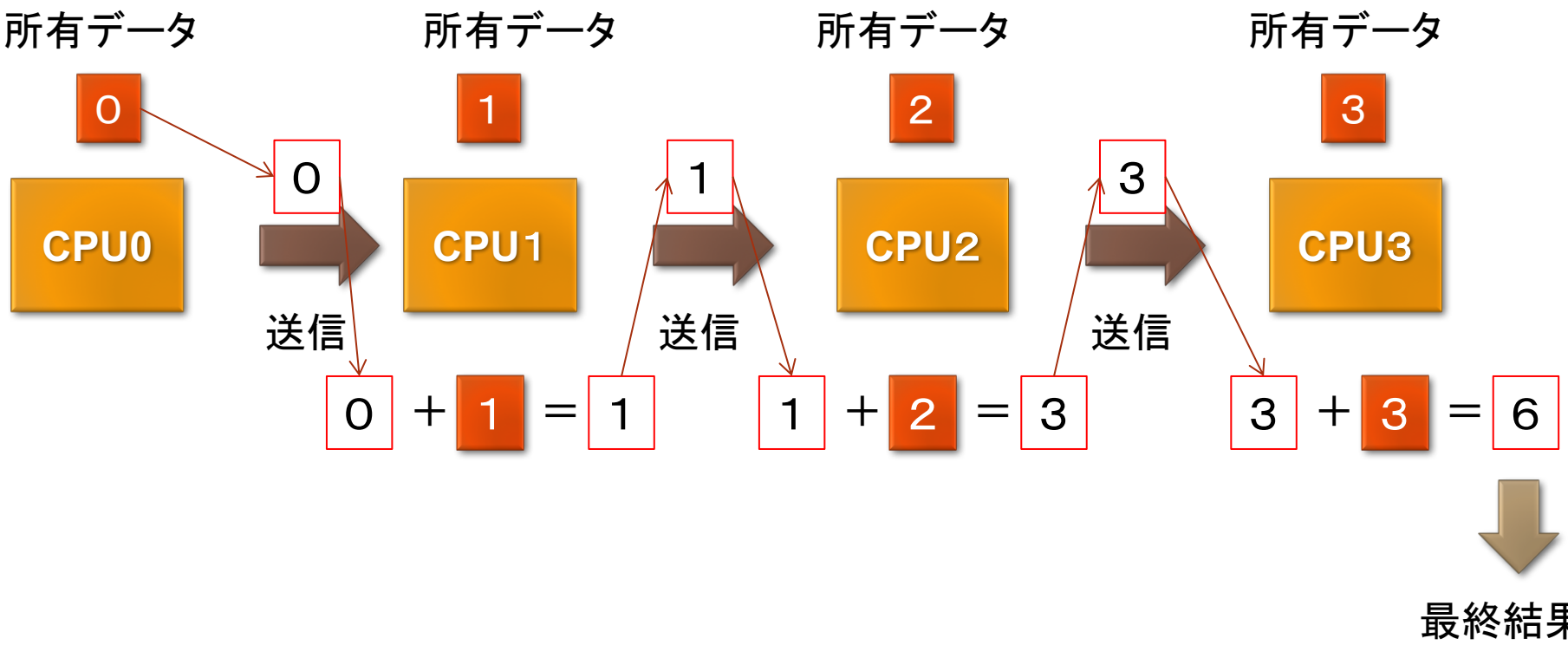
# ハイブリッドMPIの実行状況（ノード内）



# 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. （255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

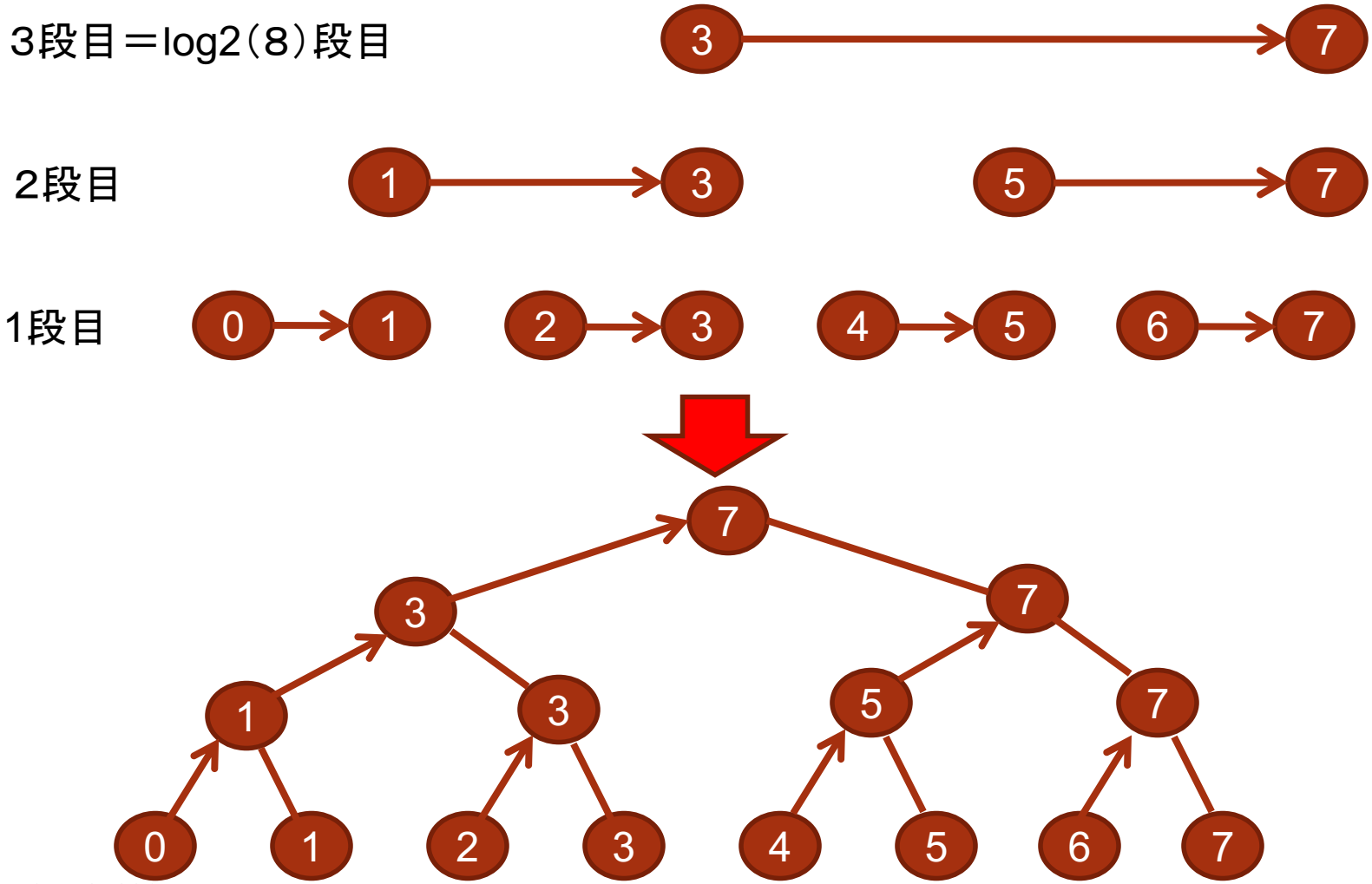
自分より一つ少ないID番号（myid-1）から、double型データ1つを受信しdrecvbuf変数に代入

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（通信衝突が多発する可能性）

# 性能プロファイラ

# 性能プロファイラ

- Reedbush-U
  - Intel VTune Amplifier
  - PAPI (Performance API)

- Webポータルから「利用の手引き」⇨
  「チューニングガイド　性能評価ツール編」
  を参照してください。

# 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　塙　敏博

# 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（Ⅱ）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

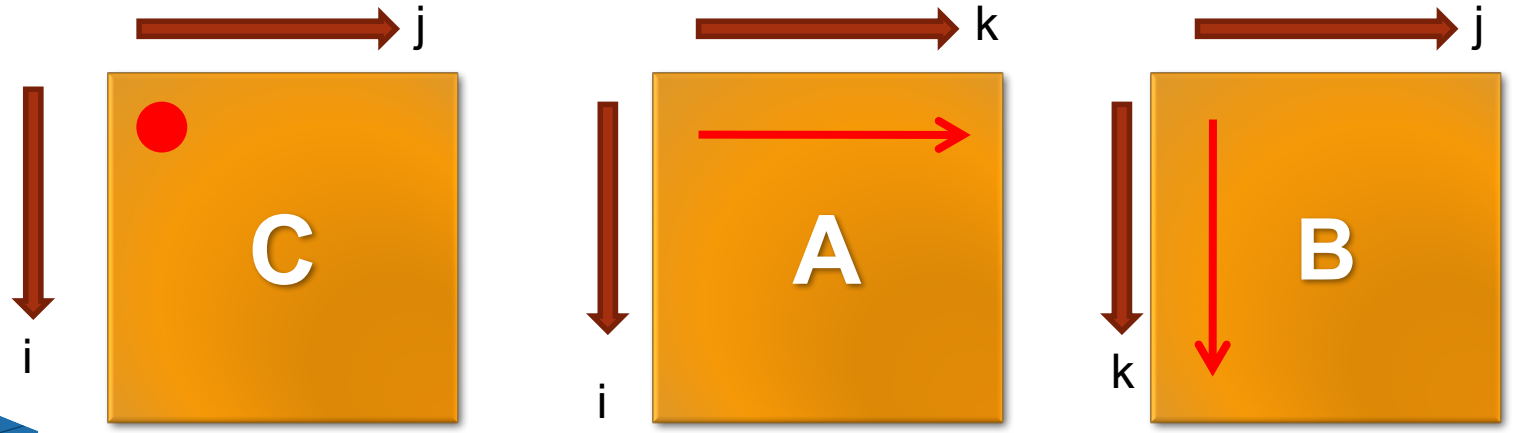
# 行列の積

- 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];
```



# 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

# 行列の積（C言語）

- 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

```
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

# 行列の積（Fortran言語）

- 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

```
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
```

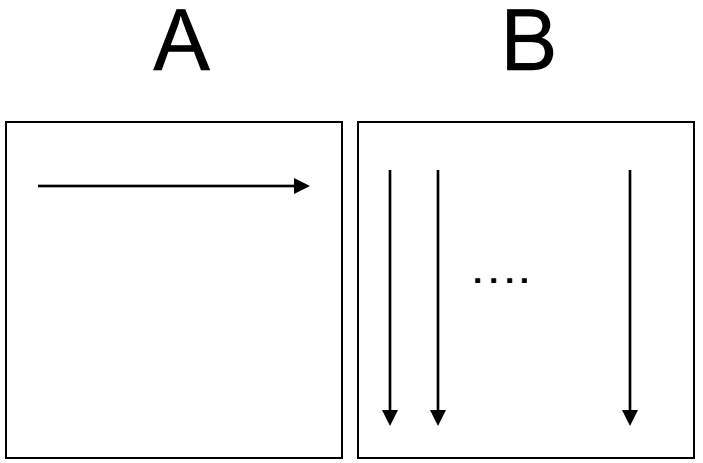
  - 最内部の演算は、外側の3ループを交換しても、計算結果が変わらない
    → 6通りの実現の方法がある

# 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ーンが＜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ーンが＜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

# 行列の積（C言語）

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



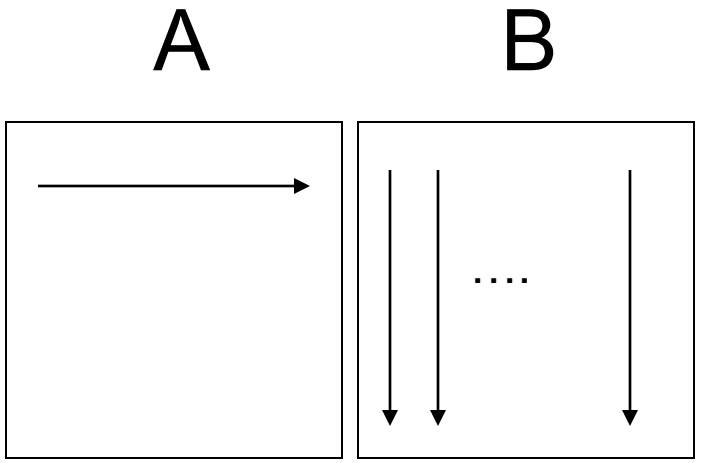

●行方向と列方向のアクセスあり
→行方向・列方向格納言語の
両方で性能低下要因
解決法：
A, Bどちらか一方を転置しておく

※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＞。

# 行列の積（Fortran言語）

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

```
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
```

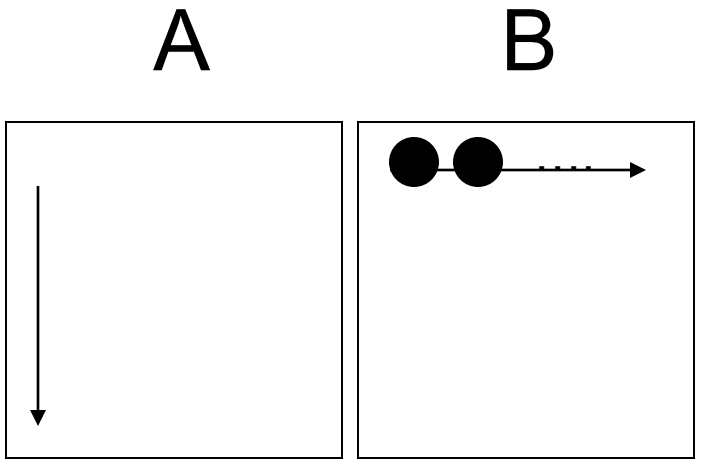



●行方向と列方向のアクセスあり
→行方向・列方向格納言語の
両方で性能低下要因
解決法：
A, Bどちらか一方を転置しておく

※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＞。

# 行列の積（C言語）

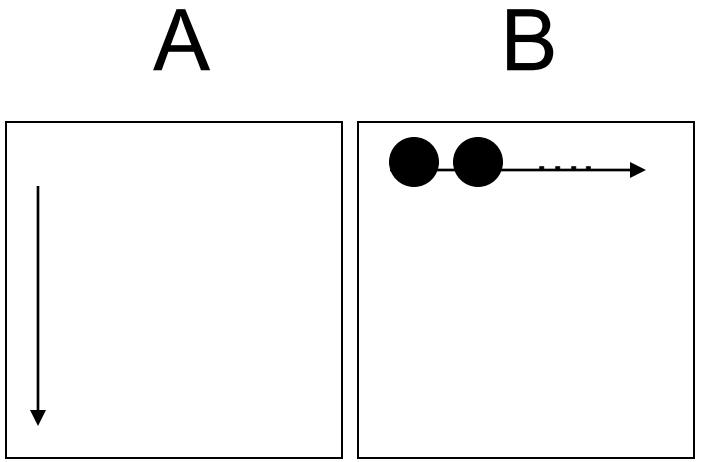
- 外積形式 (outer-product form)
  - kij, kjiループによる実現
  - 
```
for (i=0;  i<n;  i++) {
   for (j=0;  j<n;  j++) {
      C[ i ][ j ] = 0.0;
   }
}
for (k=0;  k<n;  k++) {
  for (j=0;  j<n;  j++) {
     db = B[ k ][ j ];
      for (i=0;  i<n;  i++) {
      C[ i ][ j ]= C[ i ][ j ]+ A[ i ][ k ]* db;
     }
  }
}
```




●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）

# 行列の積（Fortran言語）

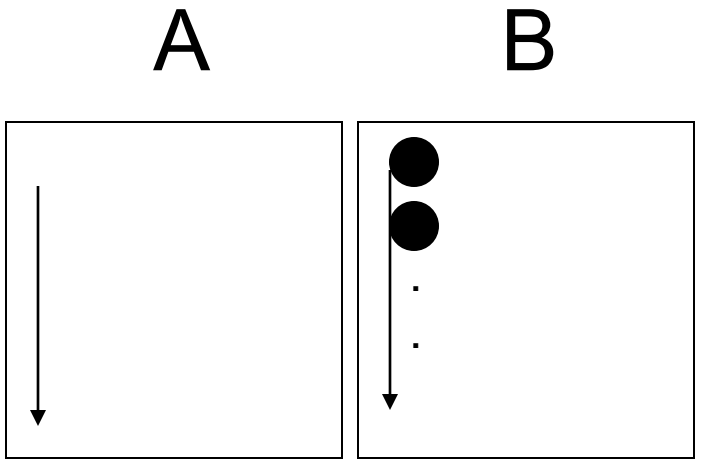
- 外積形式 (outer-product form)
  - kij, kjiループによる実現
  - ```
    do i=1, n
      do j=1, n
        C( i , j ) = 0.0d0
      enddo
    enddo
    do k=1, n
      do j=1, n
        db = B( k , j )
        do i=1, n
          C( i , j ) = C( i , j )+ A( i , k ) * db
        enddo
      enddo
    enddo
    ```




●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）
    ```

# 行列の積（C言語）

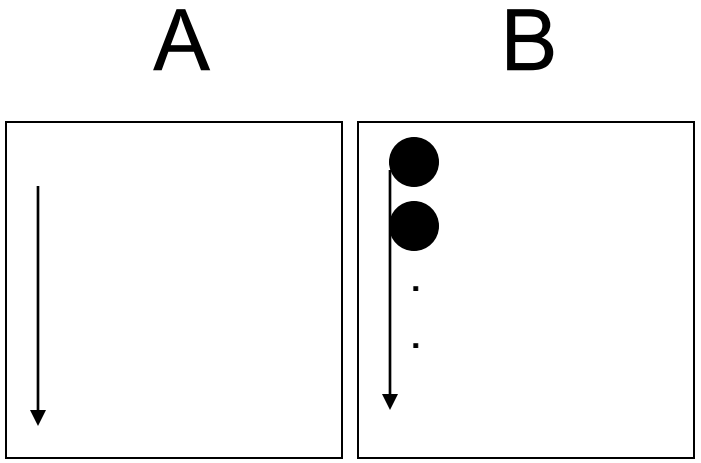
- 中間積形式 (middle-product form)
  - ikj, jkiループによる実現
  - 
```
for (j=0;  j<n;  j++)  {
    for (i=0;  i<n;  i++) {
       C[ i ][ j ] = 0.0;
    }
    for (k=0;  k<n;  k++) {
      db = B[ k ][ j ];
      for (i=0;  i<n;  i++) {
       C[ i ][ j ] = C[ i ][ j ] + A[ i ][ k ] * db;
      }
    }
}
```




●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）

# 行列の積（Fortran言語）

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

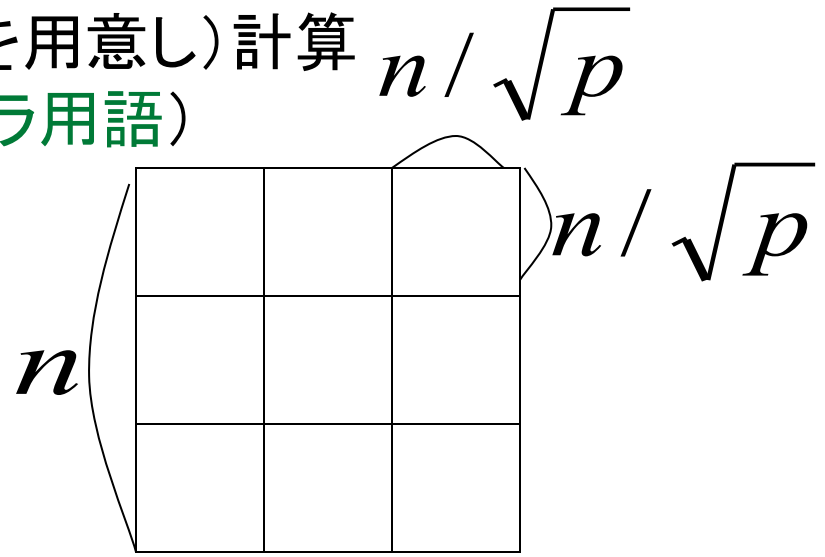


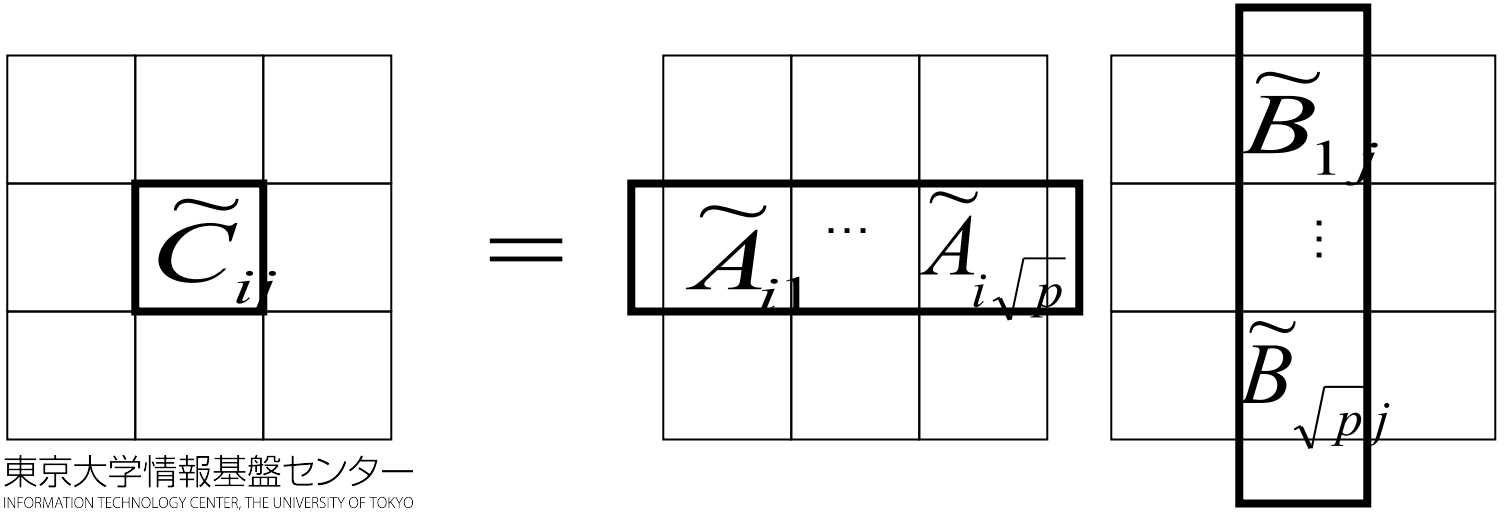

●jkiループでは
全て列方向アクセス
→列方向格納言語に
最も向いている
（Fortran言語）

# 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

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



# 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-rb.tar
- ジョブスクリプトファイルmat-mat.bash　中のキュー名を
  u-lecture から u-tutorial に変更してから
  qsub　してください。
  - u-lecture : 実習時間外のキュー
  - u-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 /lustre/gt00/z30105/Mat-Mat-rb.tar ./
  $ tar xvf Mat-Mat-rb.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
  $ qsub 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 = 0.100450 [sec.]

19910.395473 [MFLOPS]

OK!



1コアのみで、20GFLOPS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.] =   0.101383924484253
MFLOPS =    19726.9932597759
 OK!
```



1コアのみで、20GFLOPS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 288
  - #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）

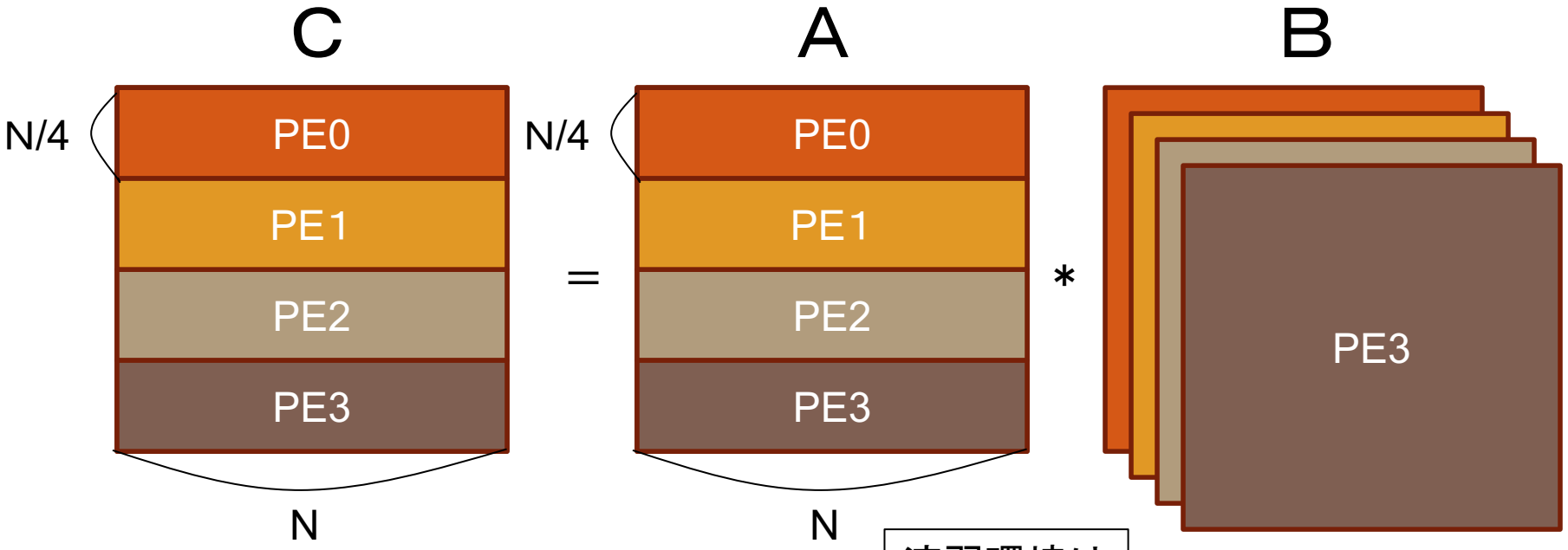
- サンプルプログラムでは、行列A、Bの要素を全部1として、行列-行列積の結果をもつ行列Cの全要素がNであるか調べ、結果検証しています。デバックに活用してください。
  - 行列Cの分散方式により、

**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。
- 行列-ベクトル積の演習と同じ方法で並列化できます。

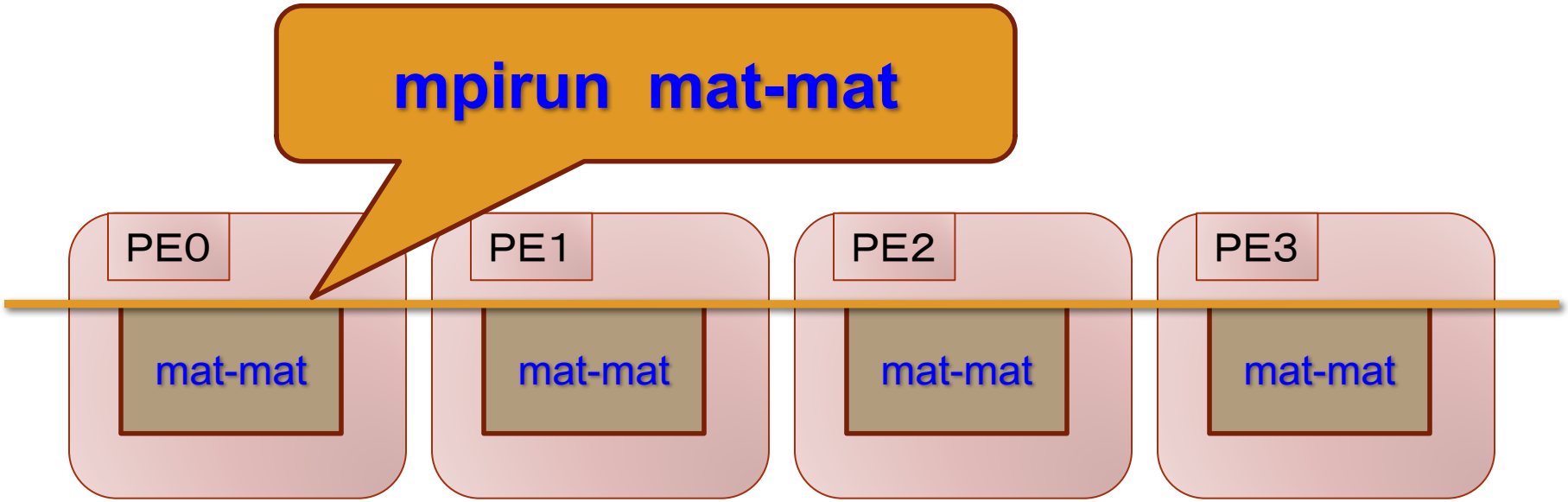
# MPI並列化の大前提（再確認）

- SPMD
  - 対象のメインプログラム（mat-mat）は、
    - すべてのPEで、かつ、
    - 同時に起動された状態

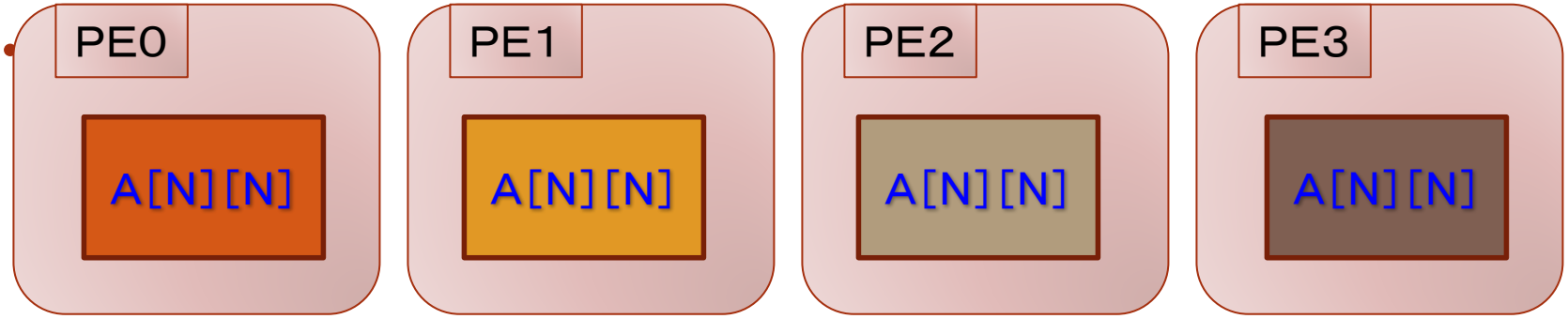
    から処理が始まる。
- 分散メモリ型並列計算機
  - 各PEは、完全に独立したメモリを持っている。（共有メモリではない）

# MPI並列化の大前提（再確認）

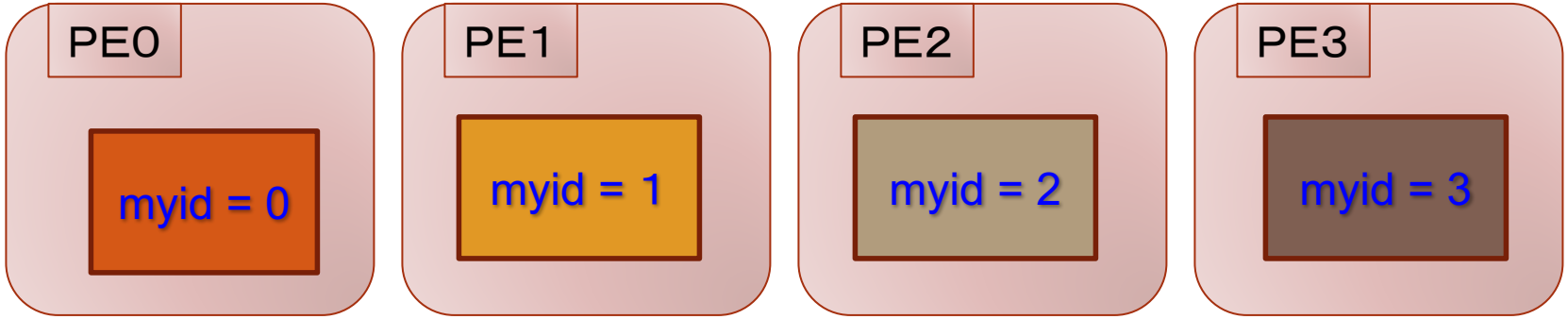
- 各PEでは、＜同じプログラムが同時に起動＞されて開始されます。



# MPI並列化の大前提（再確認）

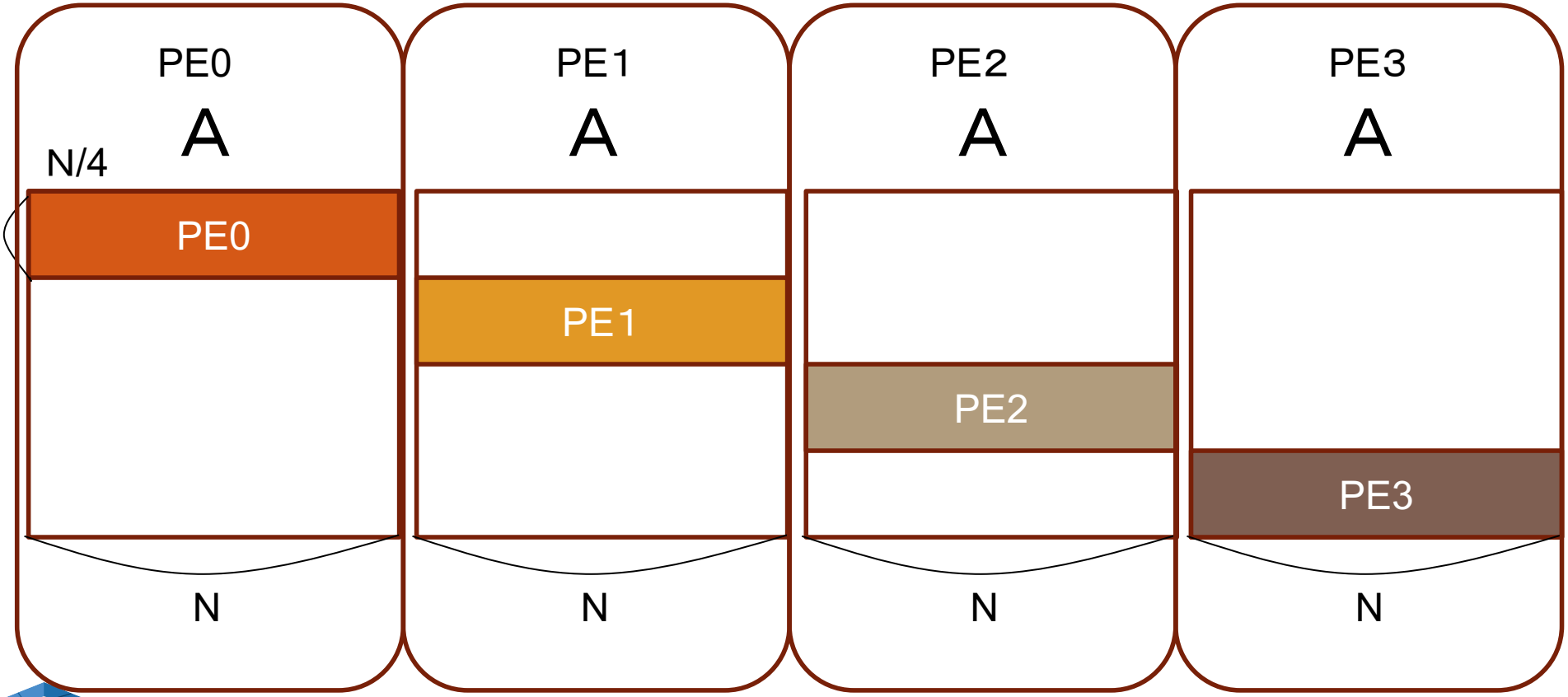
- 各PEでは、＜別配列が個別に確保＞されます。
- 


- 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

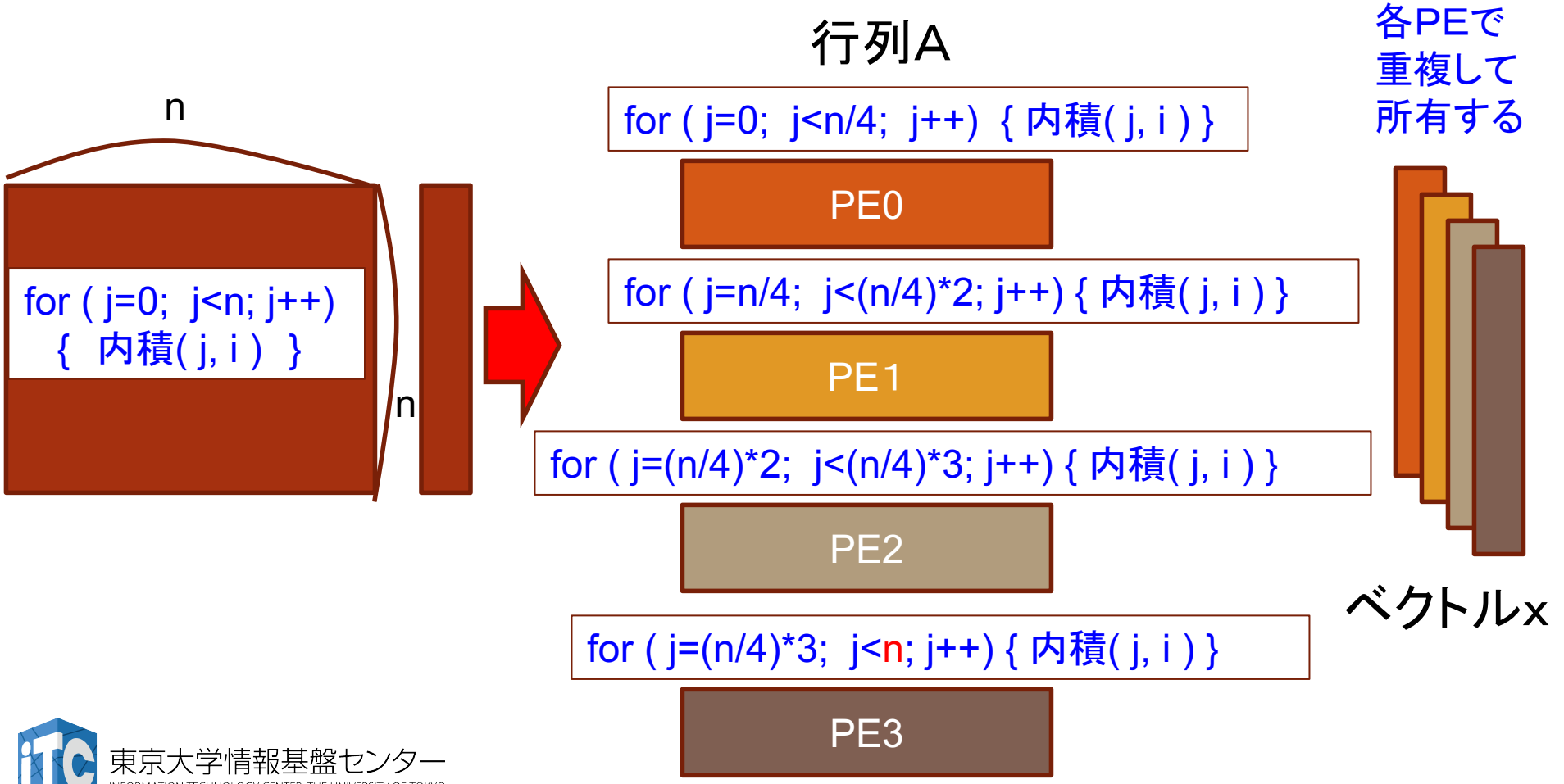


# 本実習プログラムのTIPS

- 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。

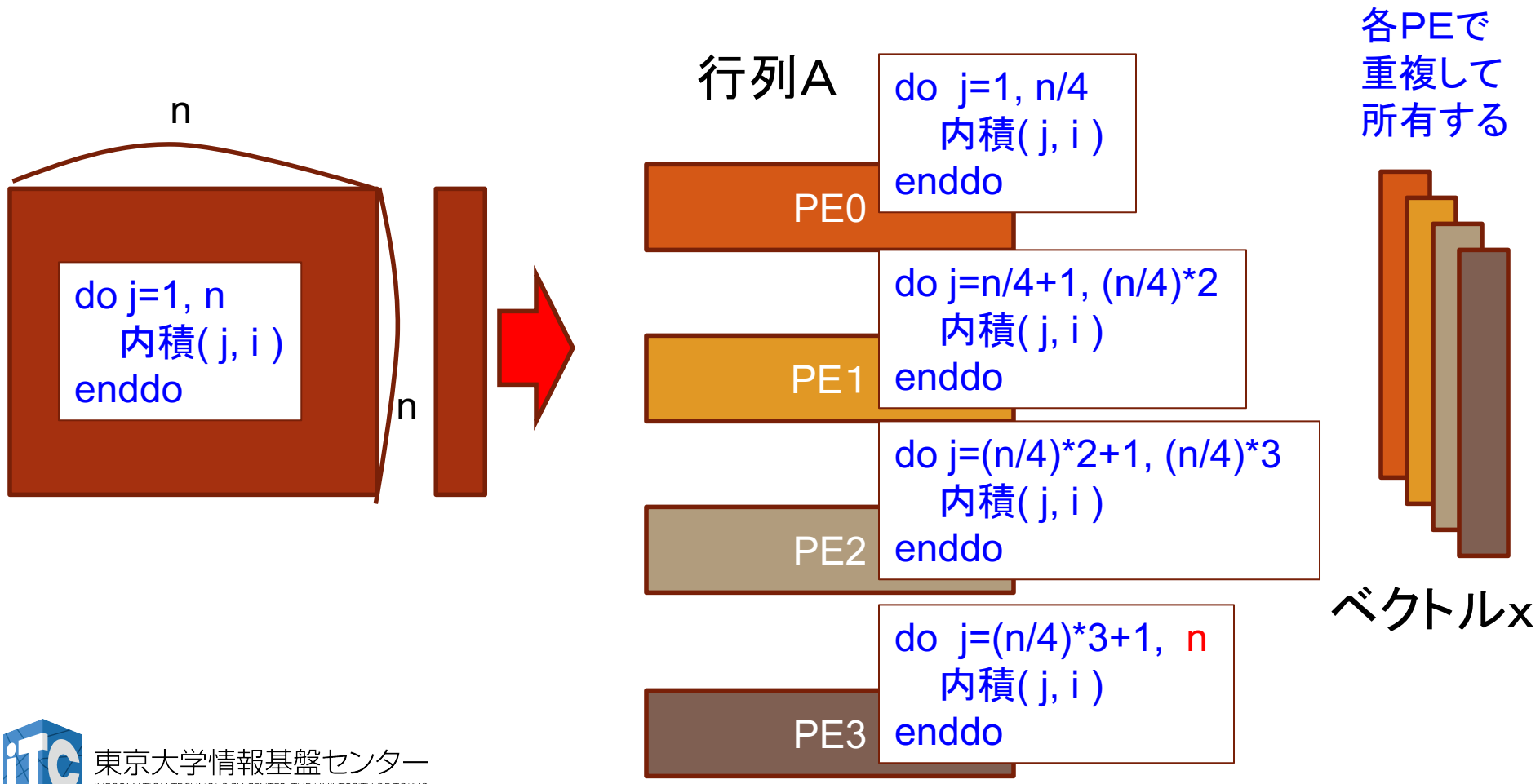
# 並列化の考え方（行列-ベクトル積の場合、C言語）

- 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
```

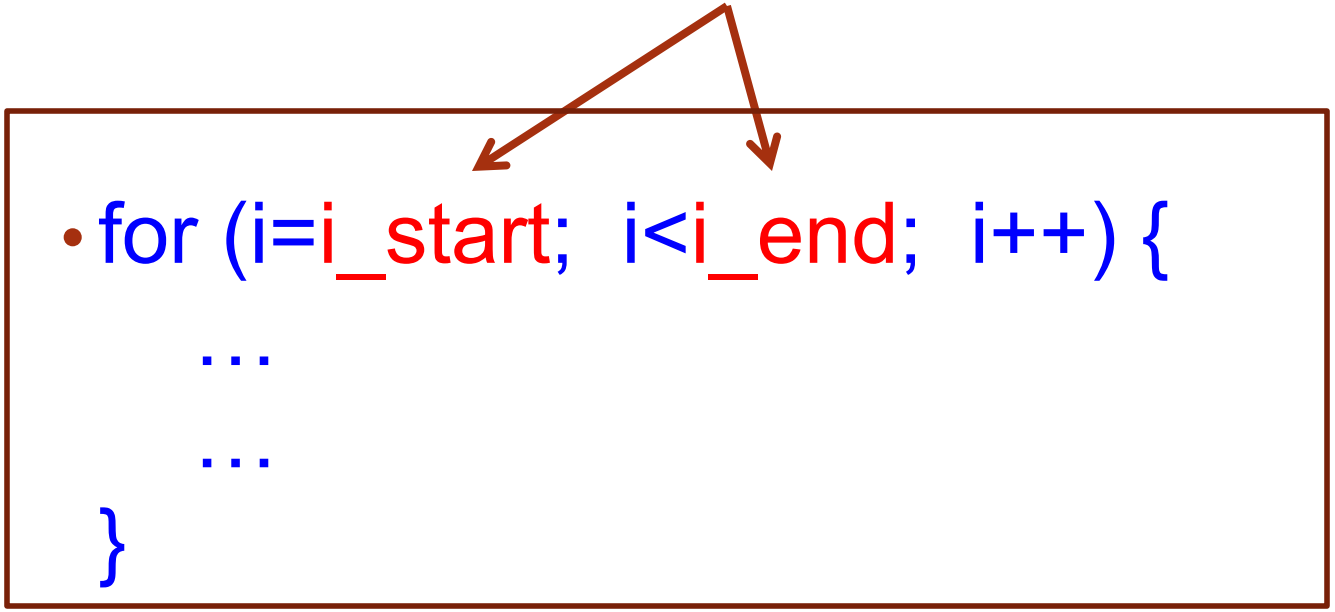
3. （2の並列化が完全に終了したら）各PEで担当のデータ部分しか行列を確保しないように変更する。
   - 上記のループは、以下のようになる。

```
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-rb.tar
- ジョブスクリプトファイルmat-mat-d.bash 中のキュー名を
  u-lecture から u-tutorial に変更してから
  qsub してください。
  - u-lecture : 実習時間外のキュー
  - u-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 /lustre/gt00/z30105/Mat-Mat-d-rb.tar ./
  $ tar xvf Mat-Mat-d-rb.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
  $ qsub 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  = 576
Mat-Mat time  = 0.000074 [sec.]
 5154623.644043 [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  =         576
Mat-Mat time  =   6.604909896850586E-003
MFLOPS =    57866.9439862109
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 576
  - 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=576)

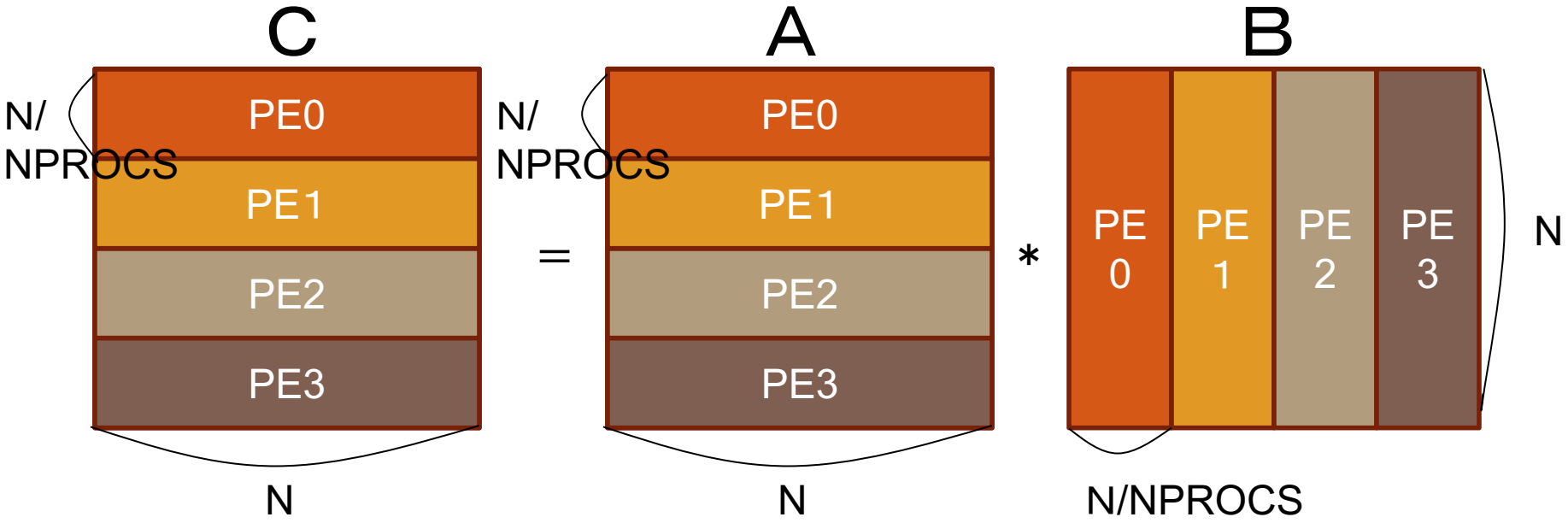
# 演習課題（1）

- MyMatMat関数（手続き）を並列化してください。
  - デバック時は
    #define N 576
    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。

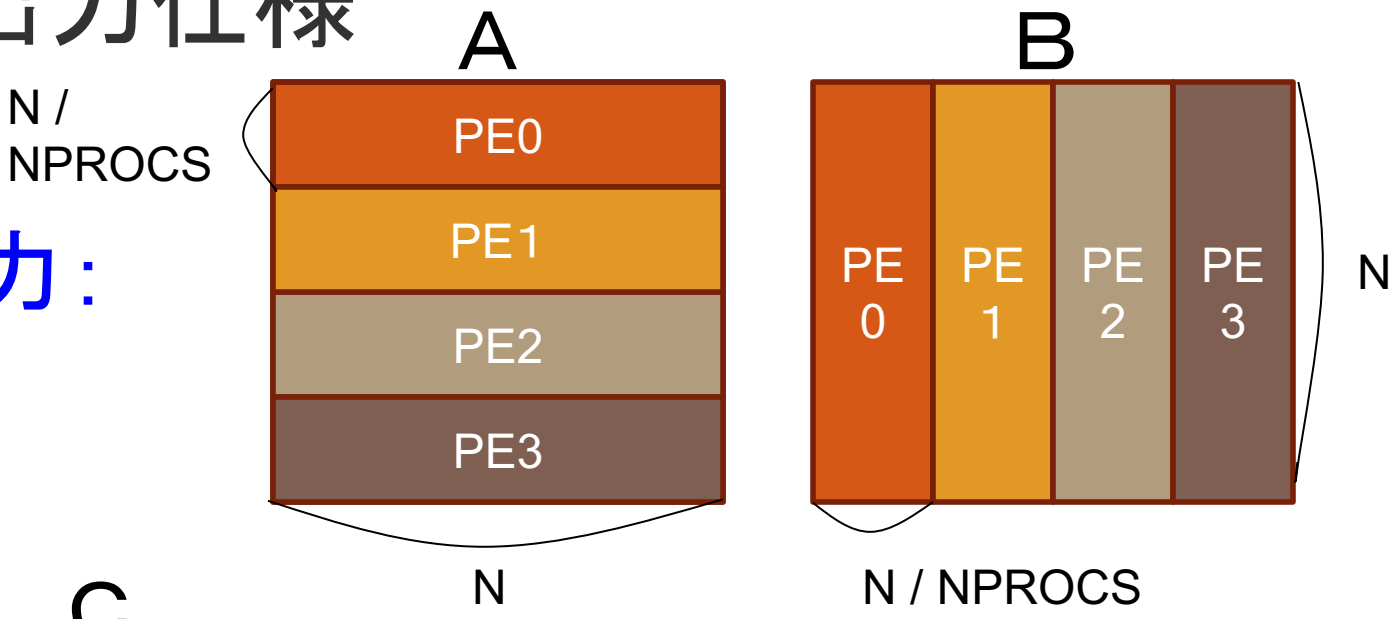
# 行列A、B、Cの初期配置

- 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、実習環境は異なります。）

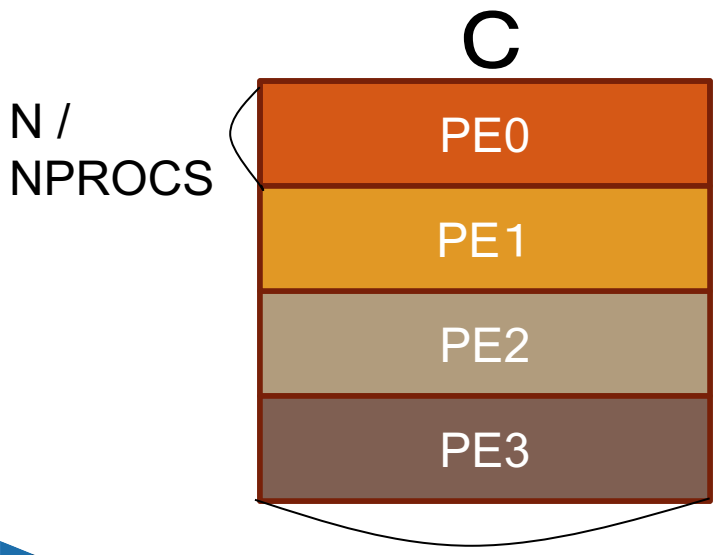



- 1対1通信関数が必要です。
- 行列A、B、Cの配列のほかに、受信用バッファの配列が必要です。

# 入力と出力仕様




入力：

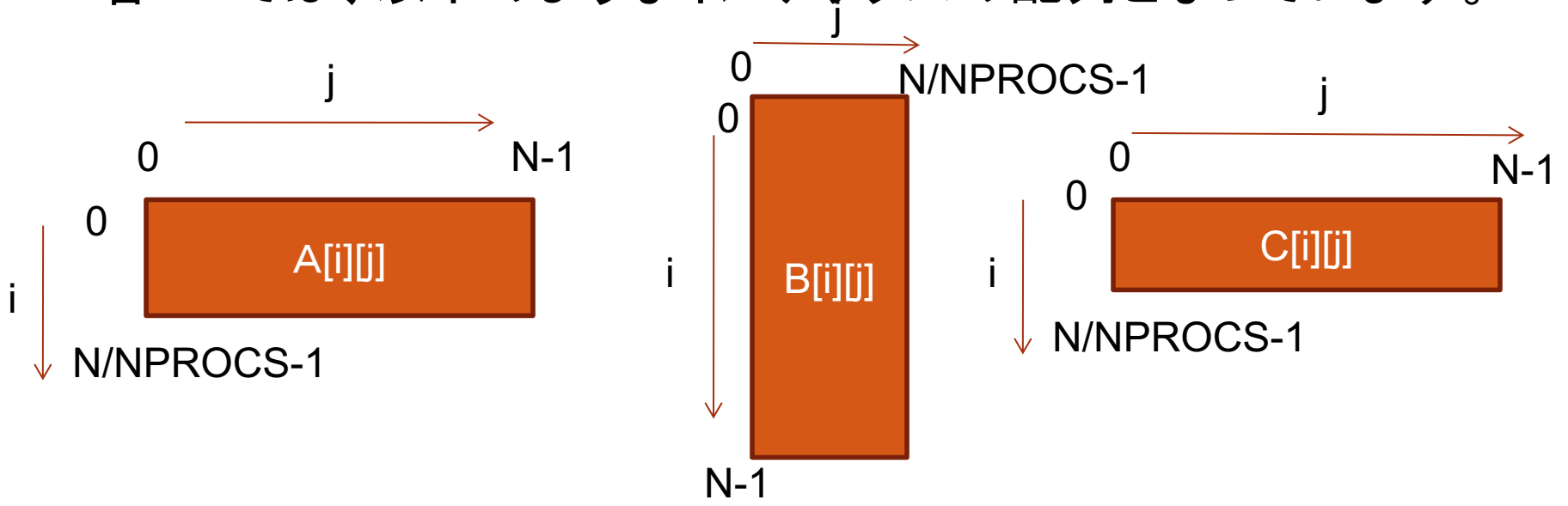



：出力

●この例は4PEの場合ですが、実習環境は異なります。

# 並列化の注意（C言語）

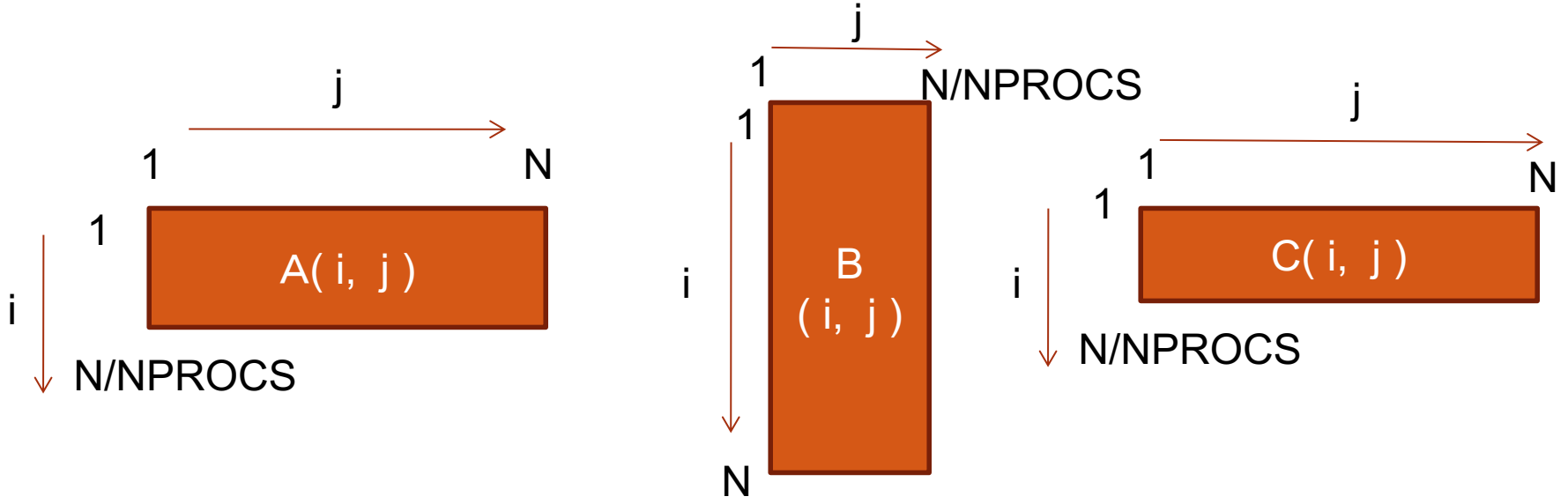
- 各配列は、完全に分散されています。
- 各PEでは、以下のようなインデックスの配列となっています。




- 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）

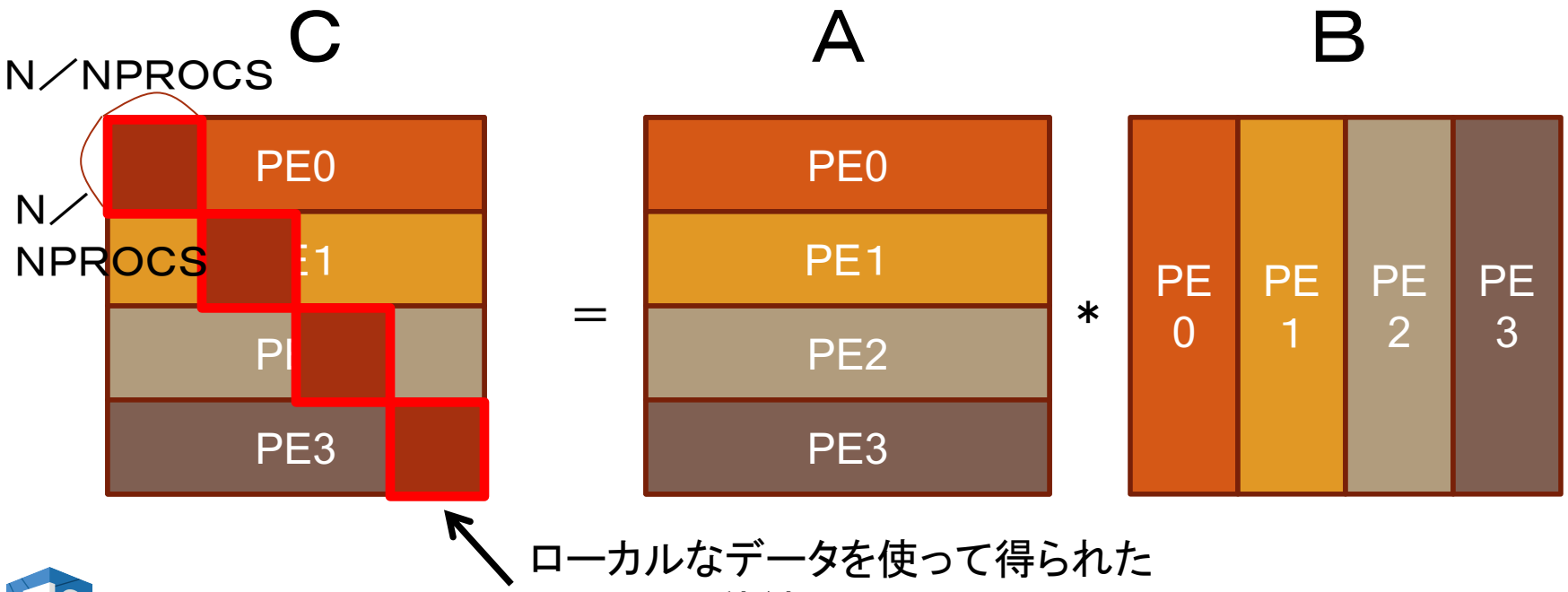
- 各配列は、完全に分散されています。
- 各PEでは、以下のようなインデックスの配列となっています。




- 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。

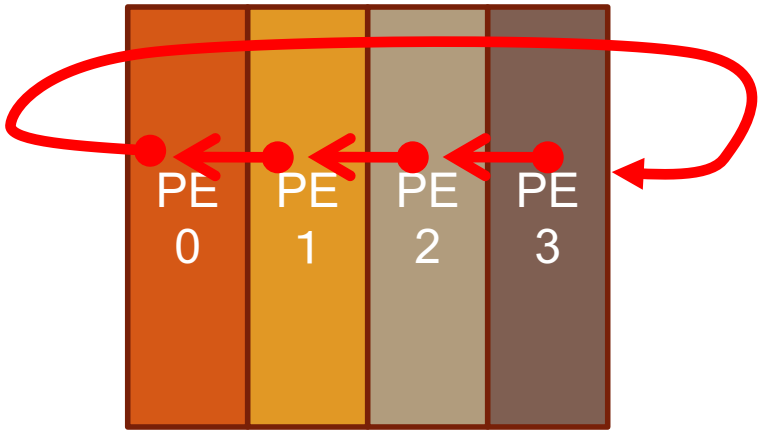
# 並列化のヒント

- 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

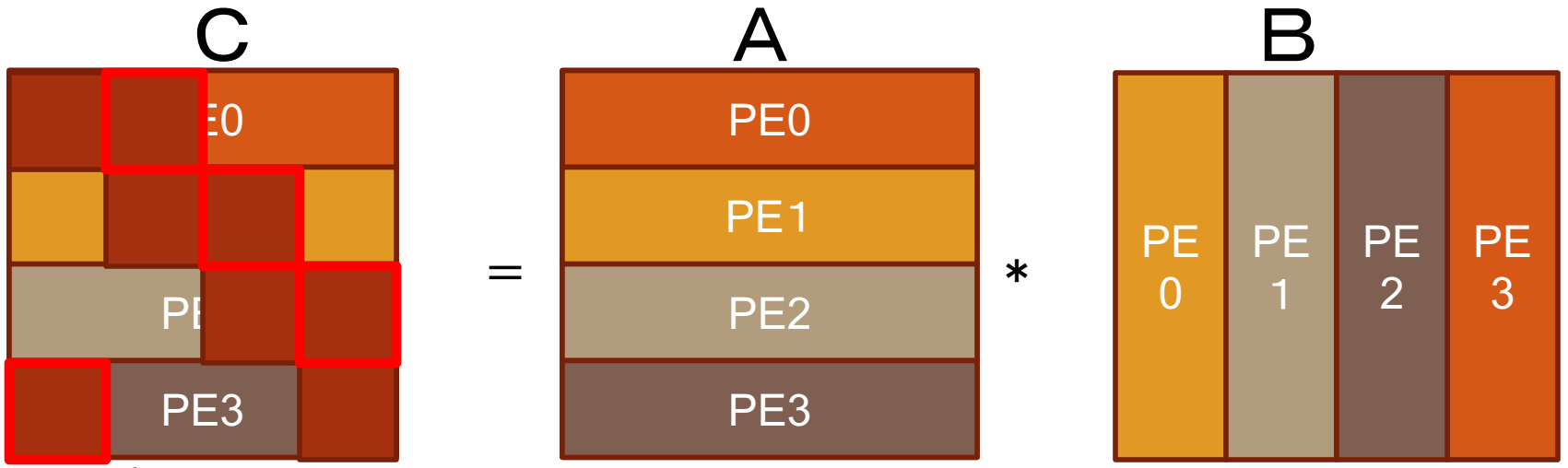


# 並列化のヒントB

- ステップ2



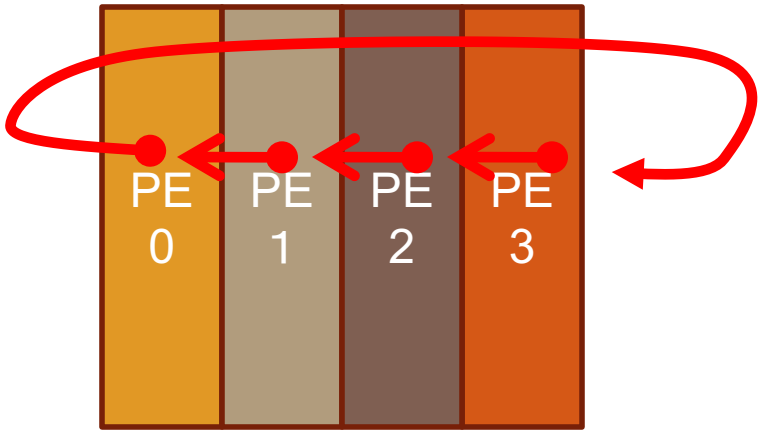

自分の持っているデータを
ひとつ左隣りに転送する
（PE0は、PE3に送る）
【循環左シフト転送】



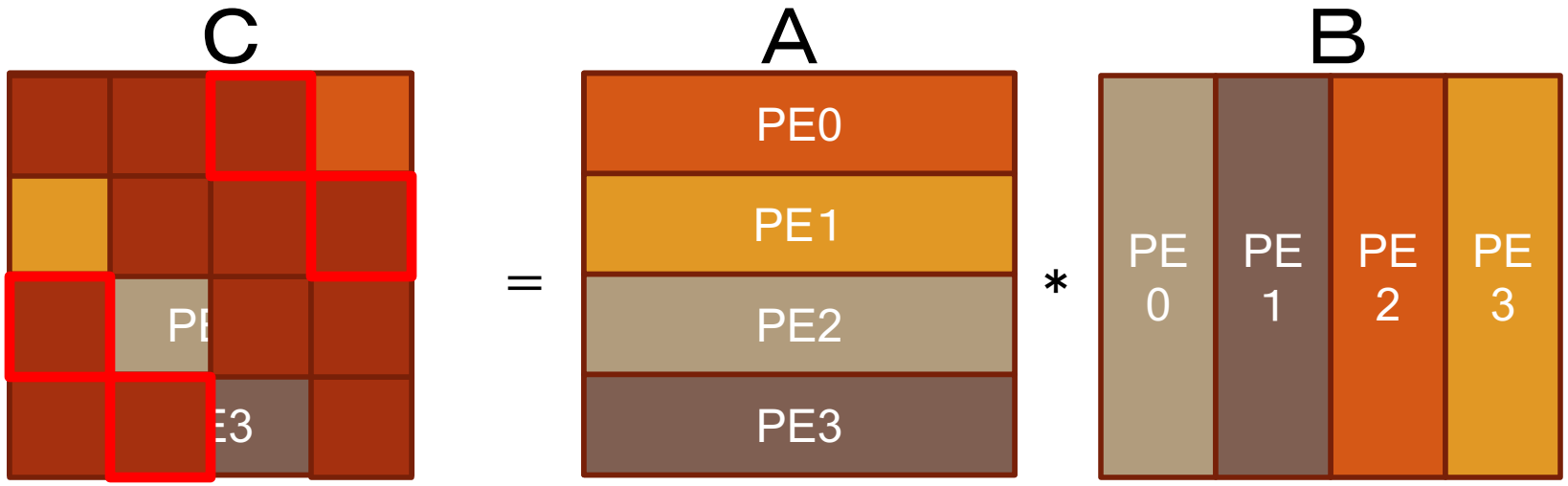

ローカルなデータを使って得られた
行列-行列積結果

# 並列化のヒントB

- ステップ3




自分の持っているデータを
ひとつ左隣りに転送する
（PE0は、PE3に送る）
【循環左シフト転送】




ローカルなデータを使って得られた
行列-行列積結果

# 並列化の注意

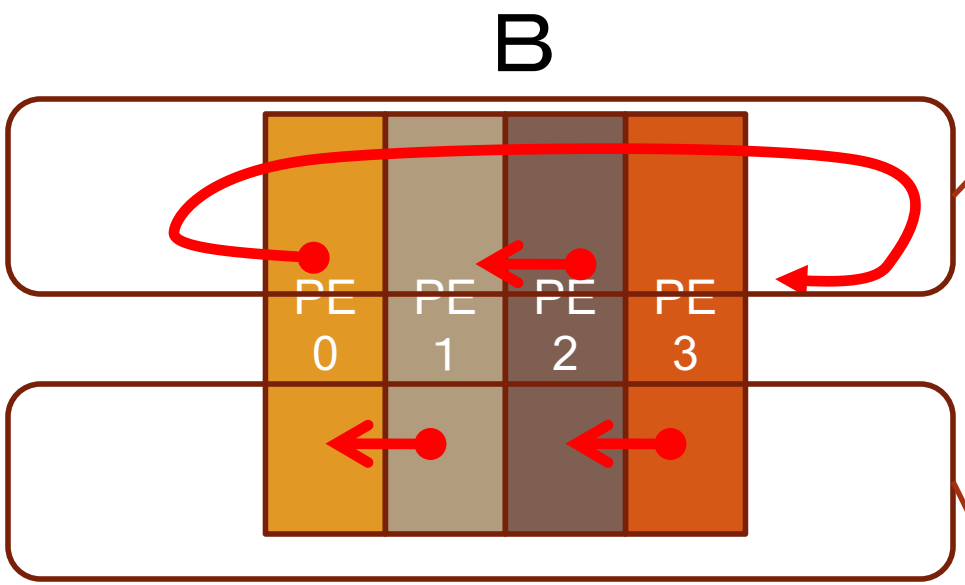
- 循環左シフト転送を実装する際、全員がMPI_Sendを先に発行すると、その場所で処理が止まる。
  （正確には、動いたり、動かなかったり、する）
  - MPI_Sendの処理中で、場合により、バッファ領域がなくなる。
  - バッファ領域が空くまで待つ（スピンウェイトする）。
  - しかし、バッファ領域不足から、永遠に空かない。
- これを回避するため、以下の実装を行う。
  - PE番号が2で割り切れるPE：
    - MPI_Send();
    - MPI_Recv();
  - それ以外のPE：
    - MPI_Recv();
    - MPI_Send();



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。
  - 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

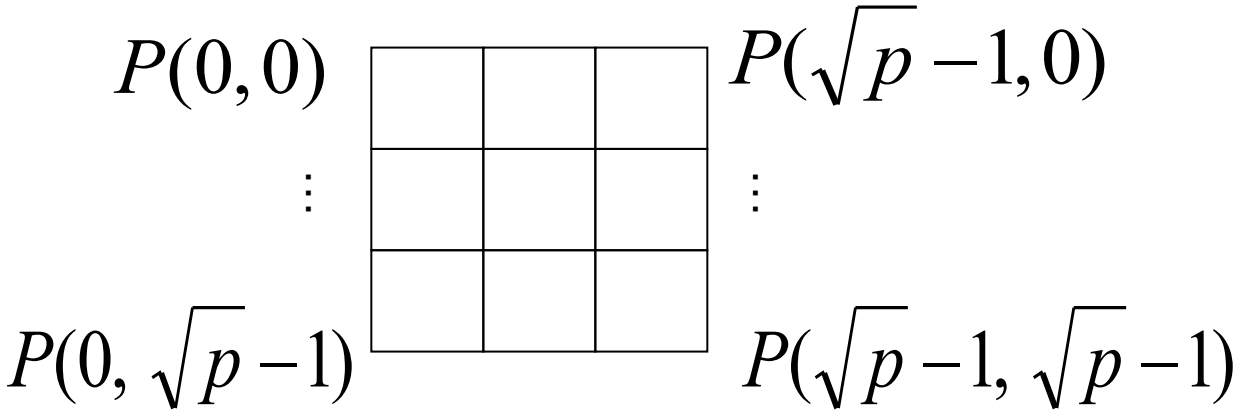
お疲れさまでした

# 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

# 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



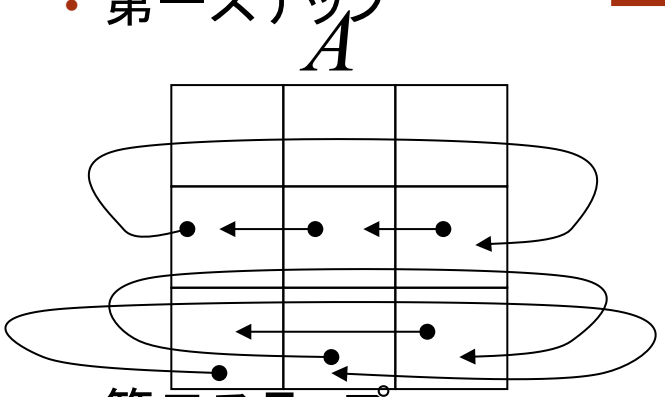

  - PE数が、2のべき乗数しかとれない
  - 各PEは、行列A、B、Cの対応する各小行列を、1つづつ所有
  - 行列A、Bの小行列と同じ大きさの作業領域を所有

# 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

# Cannonのアルゴリズム

- アルゴリズムの概略
  - 第一ステップ



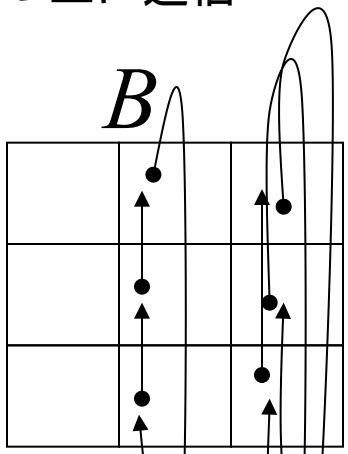

ローカルな
行列-行列積の後

：1つ右に通信

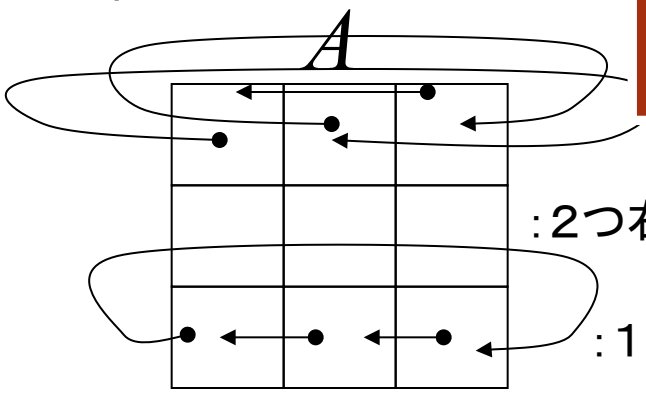
：2つ右に通信

1つ上に通信

2つ上に通信




  - 第二ステップ

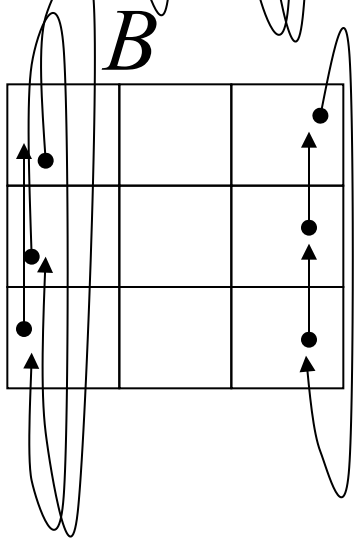



ローカルな
行列-行列積の後

：2つ右に通信

：1つ右に通信

2つ上に通信




1つ上に通信

【通信パターンが1つ下に循環シフト】

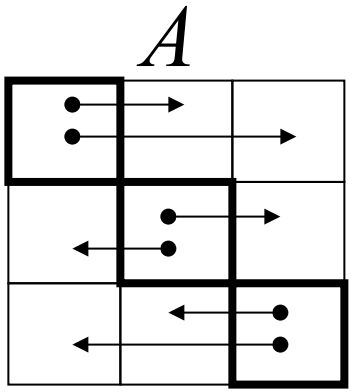
【通信パターンが
1つ右に循環シフト】

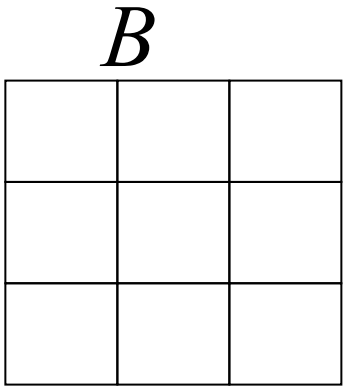
# Cannonのアルゴリズム

- 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

# Foxのアルゴリズム

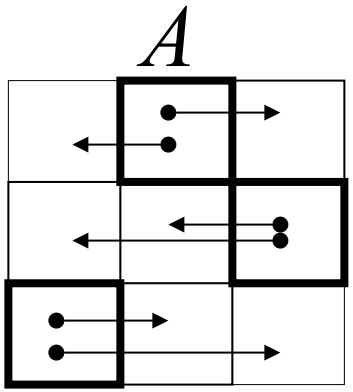
- アルゴリズムの概要
  - 第一ステップ

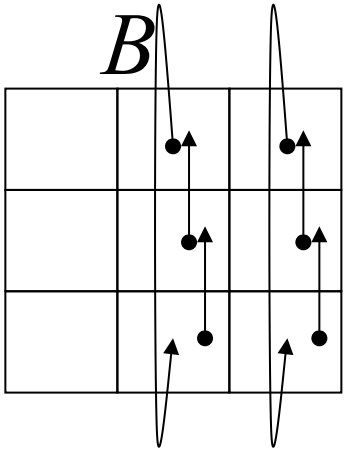



  - 第二ステップ

【放送PEが
1つ右に
循環シフト】



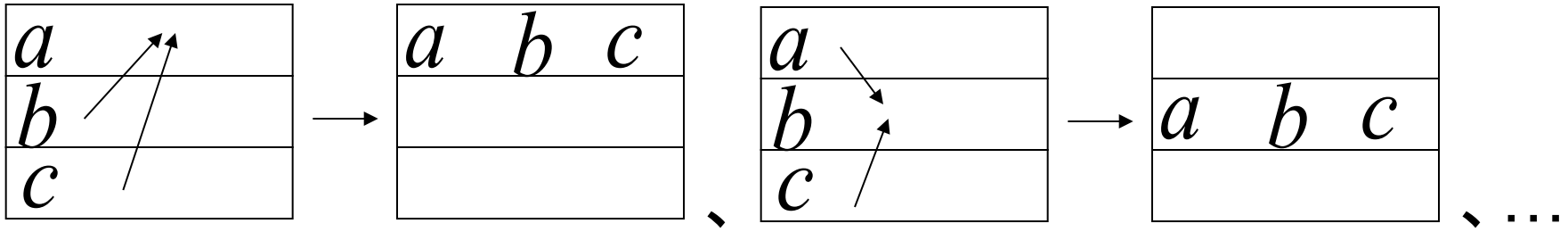

1つ上に通信

# 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

# 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？
  - 行列転置の操作をプロセッサ台数回実行



# 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）

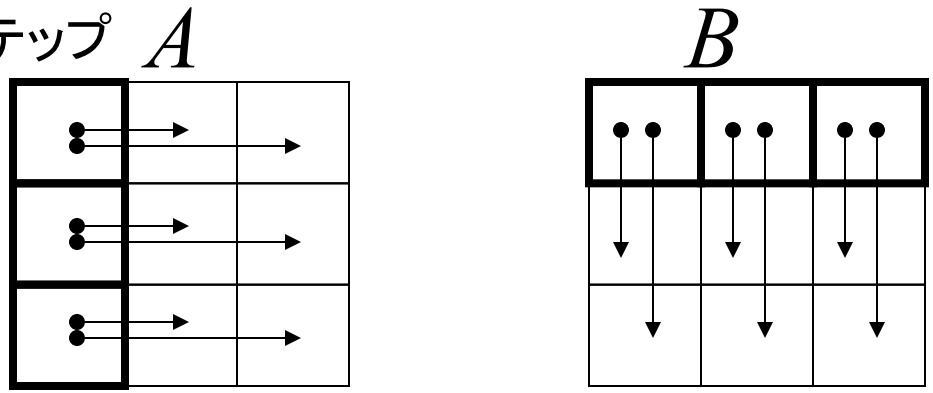
# SUMMA、PUMMA

- 近年提案された並列アルゴリズム

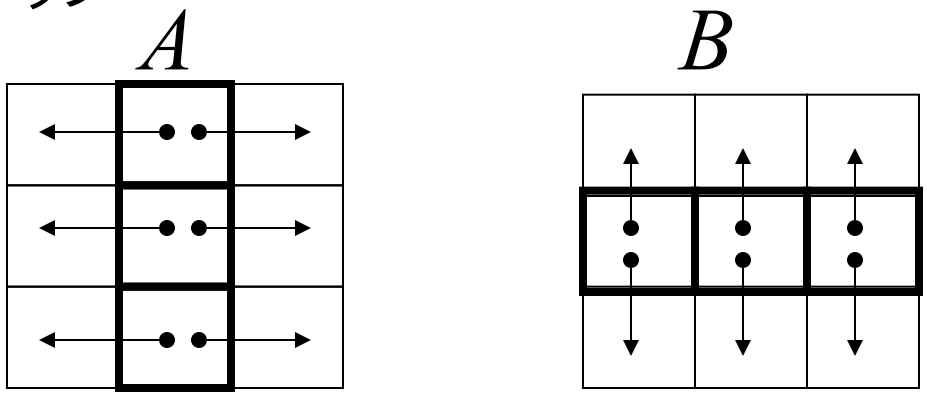
1. SUMMA（Scalable Universal Matrix Multiplication Algorithm）
   - R.Van de Geijinほか、1997年
   - 同時放送（マルチキャスト）のみで実現
2. PUMMA（Parallel Universal Matrix Multiplication Algorithms）
   - Choiほか、1994年
   - 二次元ブロックサイクリック分散方式むきのFoxのアルゴリズム

# SUMMA

- アルゴリズムの概略
  - 第一ステップ

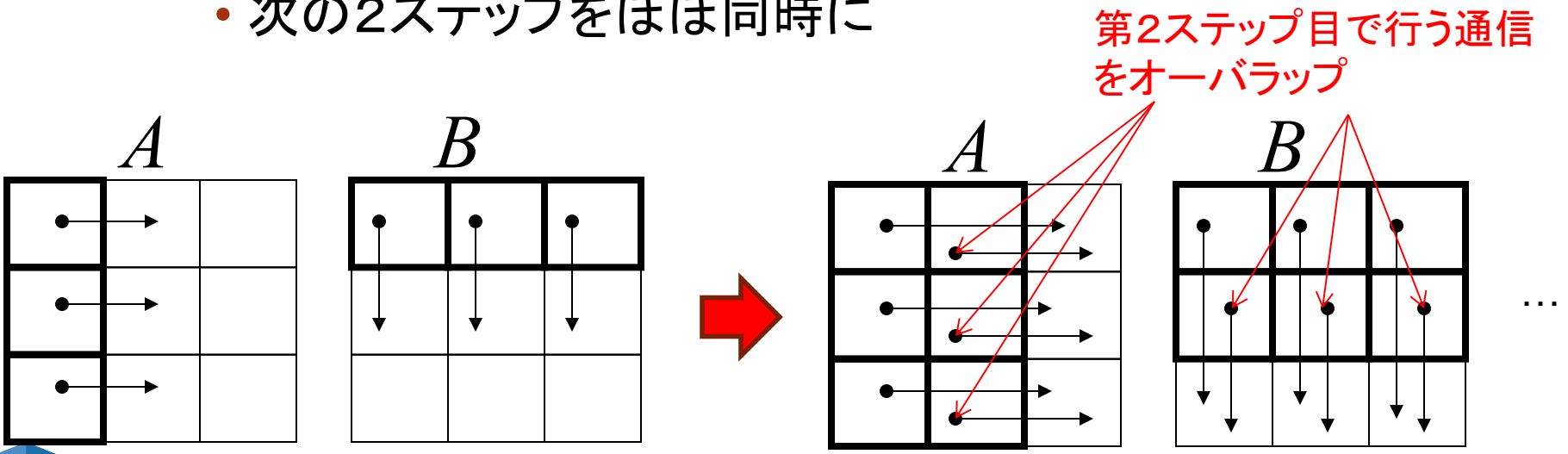



  - 第二ステップ



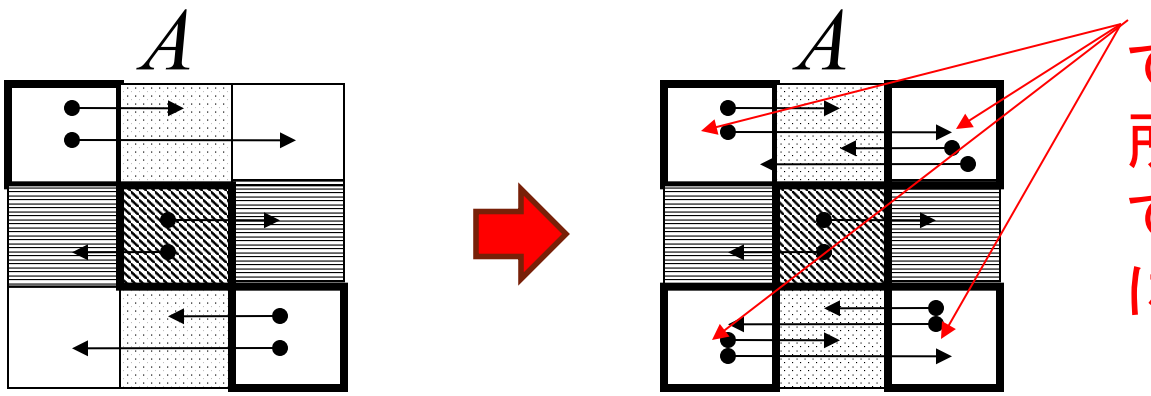
# SUMMA

- 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



# 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

# 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

# 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
- ところが通信量は、アルゴリズムの性質から、通常の行列-行列積アルゴリズムに対して減少する。この性質を利用して、近年、Strassenを用いた通信回避アルゴリズムが研究されている。

# 参考資料
# PUTTYでの使い方

# PuTTYの入手

（注：2007/06/03 Version 0.60-jp20070603 用）

- 以下をアクセスする
  http://hp.vector.co.jp/authors/VA024651/PuTTYkj.html
- putty-0.60-jp20070603.zip をダウンロードして、zipファイルを展開する。
  - Zipファイルを展開するソフトが必要です。
    たとえば、Lhaplus
    http://www.vector.co.jp/soft/win95/util/se169348.html

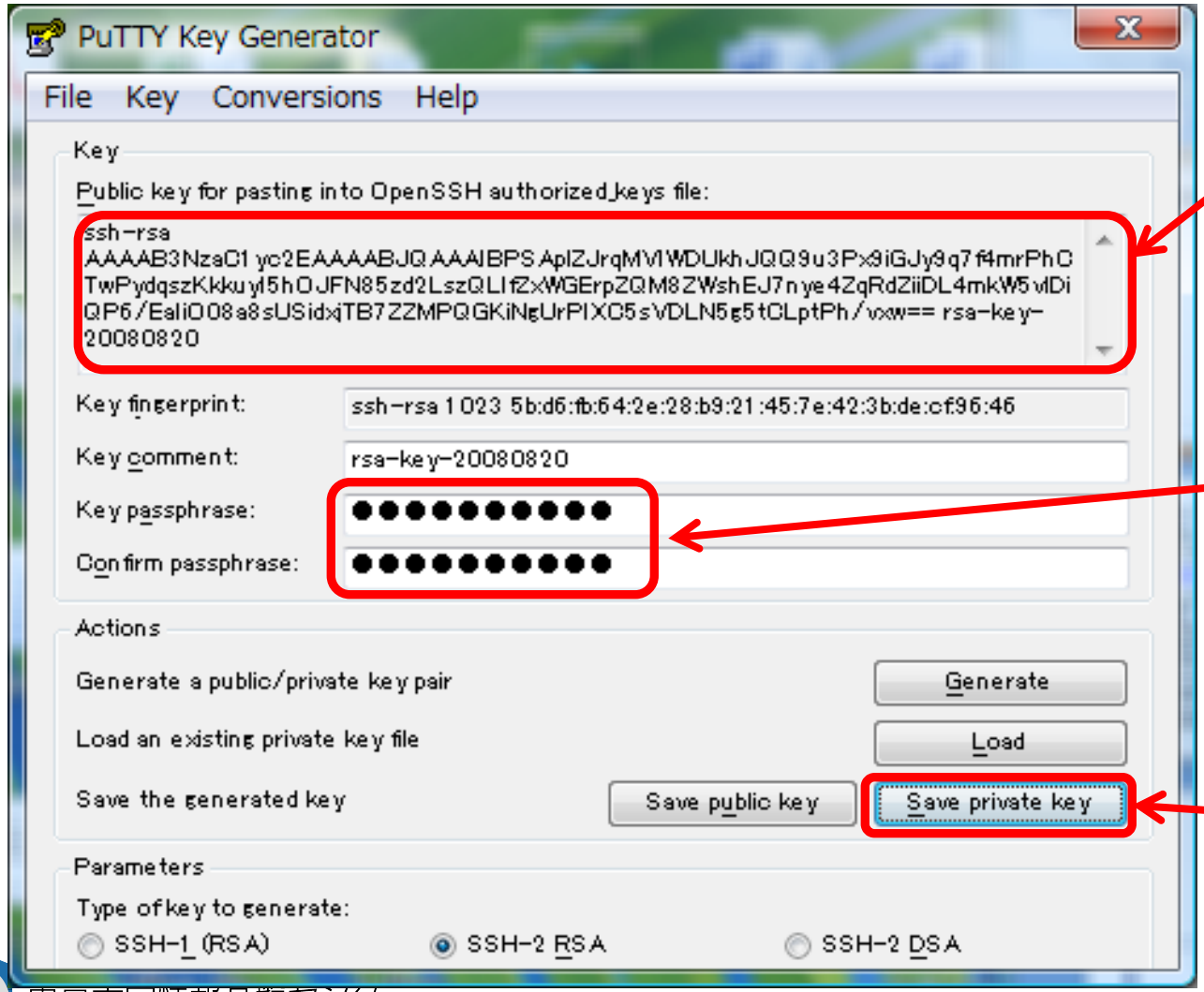
# 鍵の設定

（注：2007/06/03 Version 0.60-jp20070603 用）

- PuTTYの配布ファイルを展開すると、puttygen.exe という実行ファイルがあるので、それをクリックする
- 右側にある、“Generate”ボタンをクリックする
- バーが出る。バーが右端に行くまで待つ。
  - この間、空白エリアにマウスを移動させる。そうしないと、いつまでたっても、鍵が生成されない。
- “Key passphrase”に、ログインするとき（この鍵を使うとき）に使うパスワードを入れる。
  - 注意：センターから配られたパスワードではない。センターから配られたパスワードを入れないこと。
- “Save private key”をクリックし、適切な場所に鍵を保存する。

# 現在の状況（以下の画面は消さないこと）

（注：2007/06/03 Version 0.60-jp20070603 用）




これが「公開鍵」。
FX10に登録する

これが、ログインするときに使うパスワード。センターから配られたパスワードを入れないこと！

これでセーブされるのが「秘密鍵」。
他人に見せてはいけない。

# Reedbush-Uへのログイン

（注：2007/06/03 Version 0.60-jp20070603 用）

- “puttyjp.exe”をクリックする
- “ホスト名（またはIPアドレス）（N）”に、以下を入力する

- reedbush-u.cc.u-tokyo.ac.jp

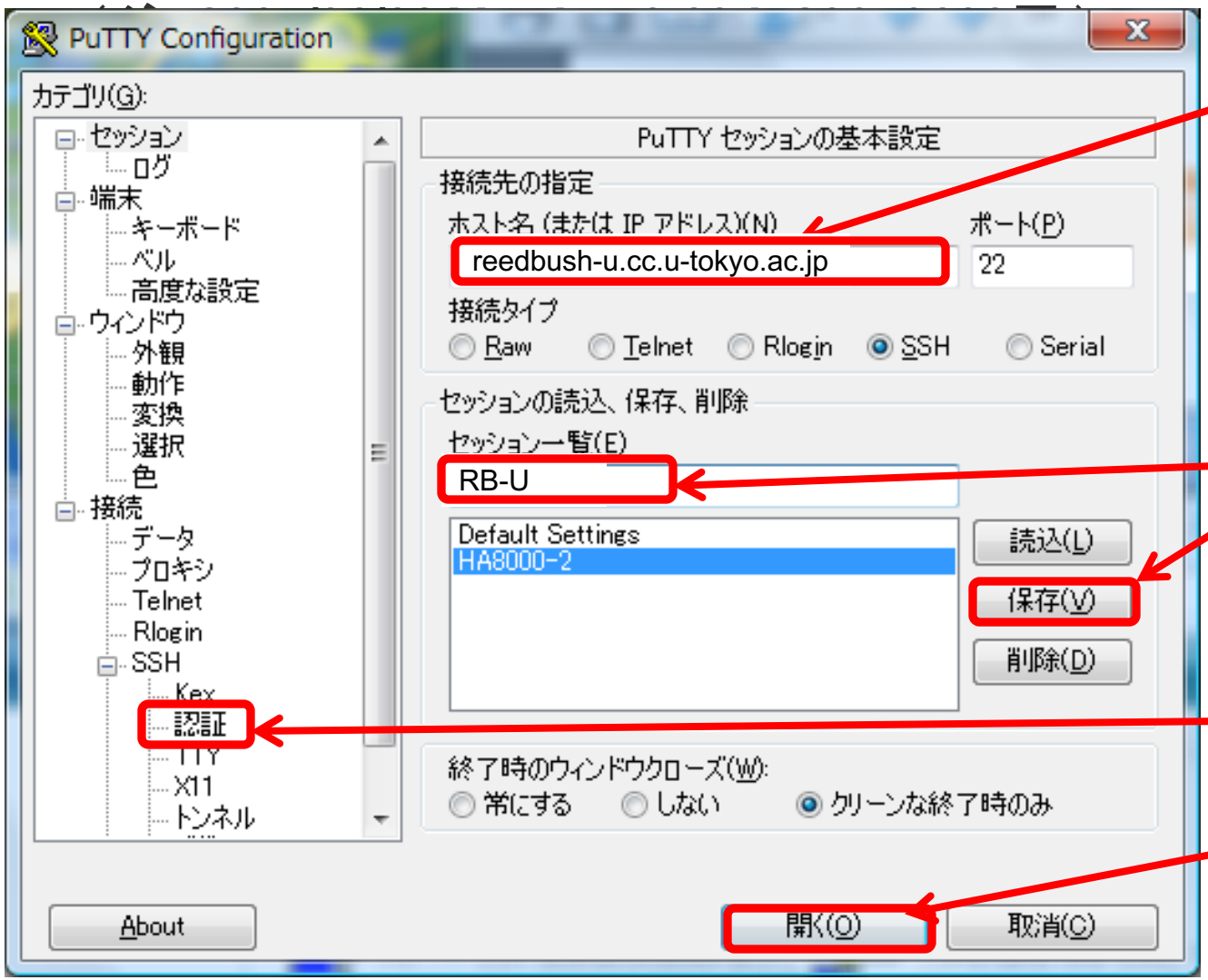
- 右側のカテゴリにある、“SSH”をクリックし、“認証”をクリックする。“認証パラメータ”の中の、“認証のためのプライベートキーファイル”に、先ほど保存した「秘密鍵」のファイルを登録する。
- 左側のカテゴリにある、“セッション”をクリックする

# Reedbush-Uへのログイン

（注：2007/06/03 Version 0.60-jp20070603 用）

- “セッション一覧”に、適当な名前（FX10）を入れ、“保存”をクリック
- 下の、“開く”をクリック
- “login as:”が出るので、センターの利用者番号を入力
- Authenticating with public key ”rsa-key-20080820“ Passphrase for key ”rsa-key-20080820“ のようなメッセージが出るので、自分で設定した鍵を利用するためのパスワードを入力
  （注：センターが配布したパスワードではない）
- 以上が正しければ、ログインができる

# 現在の手順




手順1

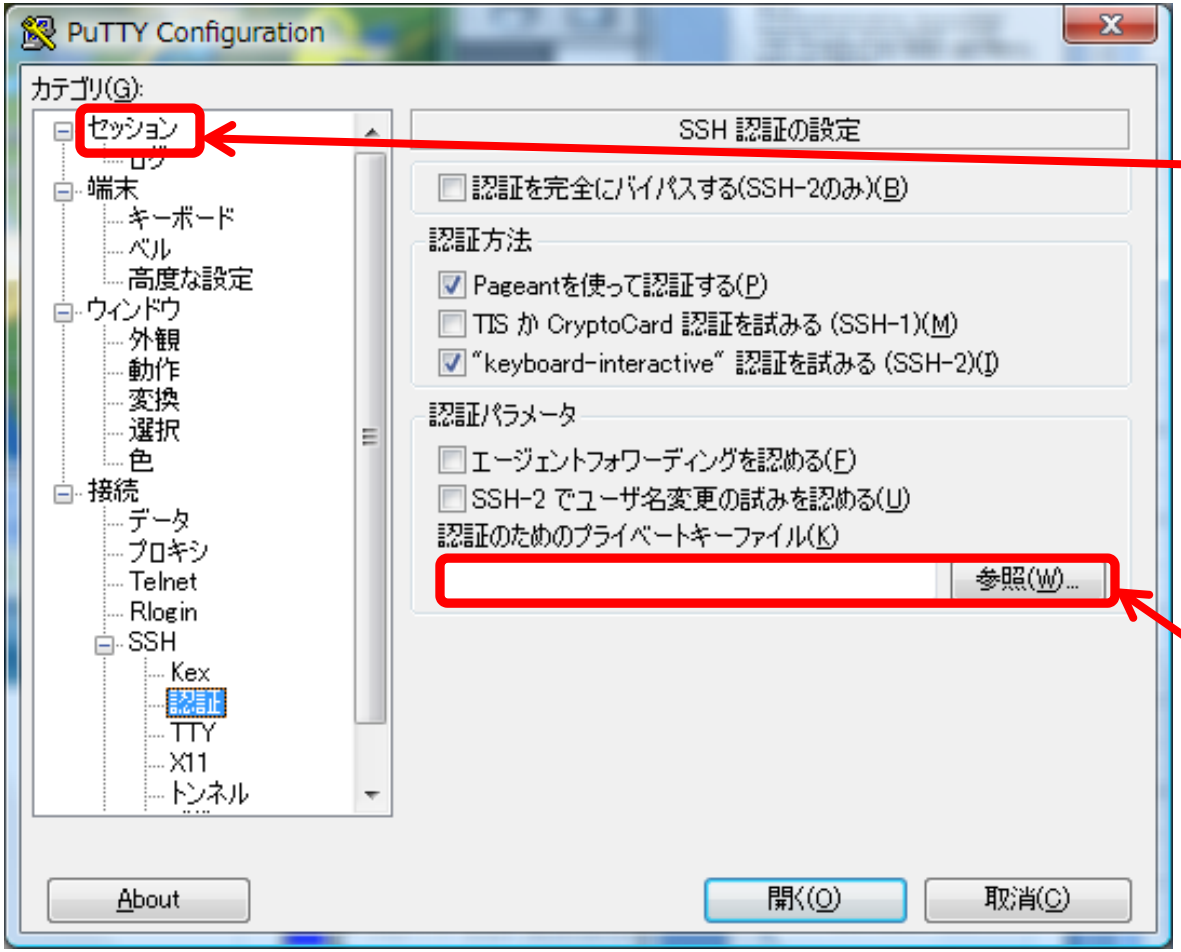
手順5

手順2

手順6：
ログインへ

# 現在の手順

## （注：2007/06/03 Version 0.60-jp20070603用）




手順４

手順３

# Reedbush-Uへのログイン（以降）

（注：2007/06/03 Version 0.60-jp20070603 用）

- 以降、“セッション読込”にある、設定ファイル（先ほど例では、“RB-U”）を指定して、“読込”ボタンをクリックすると、先ほどの設定が復元できる
- 同様に、“開く”をクリックすると、ログインできる
- 画面が小さいという人は、、、
  - “ウインドウ”—“外観”をクリックし、フォントの設定にある“変更”をクリックする
  - サイズから、適切なフォントサイズを指定する
  - 設定ファイルに、この設定を保存する。（保存を忘れると、次回にこの環境が復元できない。）

# Reedbush-Uへログイン（SSHが使える人）

- ターミナルから、以下を入力する
  $ ssh reedbush-u.cc.u-tokyo.ac.jp -l tYYxxx
  「-l」はハイフンと小文字のL、
  「tYYxxx」は利用者番号（数字）
  “tYYxxx”は、利用者番号を入れる
- 接続するかと聞かれるので、 yes を入れる
- 鍵の設定時に入れた
  自分が決めたパスワード（パスフレーズ）
  を入れる
- 成功すると、ログインができる

# WinSCPのインストール

- 以下のサイトへ行く

  http://winscp.net/eng/download.php
- 適当なバージョンをダウンロード
- インストーラで日本語を設定すると、日本語版がダウンロードできる
- アイコンをクリックする
  - ホスト名に、reedbush-u.cc.u-tokyo.ac.jp
  - ユーザ名に、センター発行の利用者番号
  - 秘密鍵に、生成した秘密鍵ファイルを指定する
  - プロトコルを、“SCP”にする
- “ログイン”を押すと、ファイル転送モードになる

# TeraTerm Proのインストール

## （利用したい人のみ。Puttyの利用を推奨します）

注意：以下の利用法は、puttyで生成したppk形式の秘密鍵では、そのままでは使えません。ssh-keygenで生成した鍵を持つ人が利用する方法です。

- 以下のアドレスから、UTF-8 TeraTerm Pro with TTSSH2 をインストールしてください：

  http://sourceforge.jp/projects/ttssh2/

1. 「パッケージ」の「UTF-8 TeraTerm Pro with TTSSH2」をクリック
2. 「4.56」の「teraterm_utf8-4.56.exe」をクリック
3. ブラウザに「セキュリティ保護のため…」が出る場合、クリックしてブロックを解除
4. 「このファイルを実行または保存しますか？」が出るので、「実行」をクリック
5. 言語（Japanse）や「同意」を確認して、インストール

# TeraTerm Proの実行（必要な人のみ）

- TeraTerm Proを実行したら、「秘密鍵」を登録してログインする