第98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

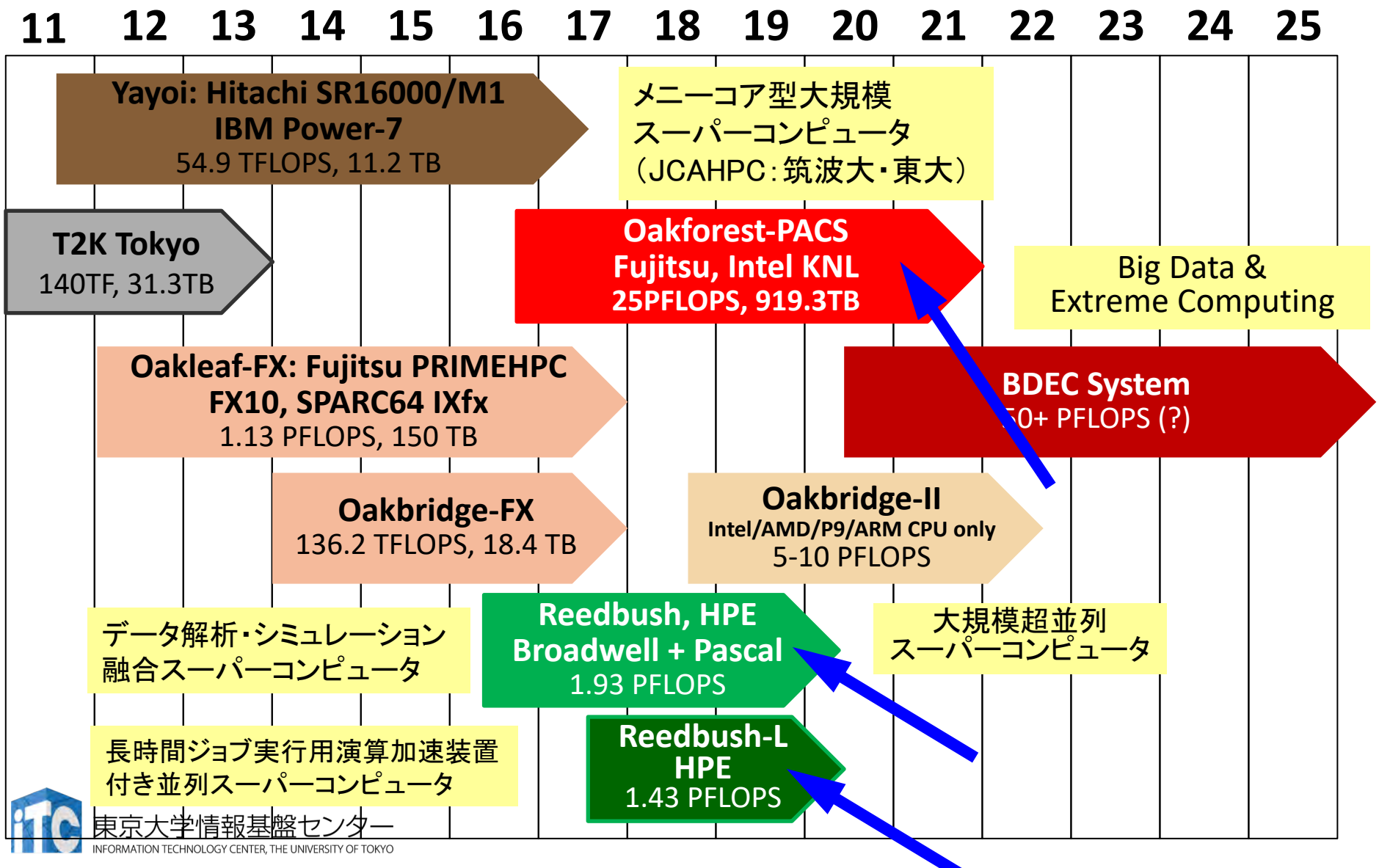
内容に関するご質問は
hanawa @ cc.u-tokyo.ac.jp
まで、お願いします。

講習会概略

- 開催日：
  2018年 4月26日（木） 10：30 - 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
  - 10：00 - 10：30 受付
  - 10：30 - 11：30 ノートパソコンの設定、テストプログラムの実行（演習）
  - 11：30 - 12：30 並列プログラミングの基本（座学）
    (12：30 - 14：00 昼休み)
  - 14：00 - 15：00 MPIプログラム実習Ⅰ（演習）
  - 15：10 - 16：00 MPIプログラム実習Ⅱ（演習）
  - 16：10 - 17：00 MPIプログラム実習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（だった）



2(または3,4)システム運用中

- 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 (HPE, 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
    - 2016-Jun.2016年7月～2020年6月
  - 東大情基セ初のGPU搭載システム
  - Reedbush-U: CPU only, 420 nodes, 508 TF (2016年7月)
  - Reedbush-H: 120 nodes, 2 GPUs/node: 1.42 PF (2017年3月)
  - Reedbush-L: 64 nodes, 4 GPUs/node: 1.43 PF (2017年10月)

- **Oakforest-PACS (OFP)** (富士通, Intel Xeon Phi (KNL))
  - JCAHPC (筑波大CCS＆東大ITC)
  - 25 PF, 世界第9位 (2017年11月) (日本第2位)
  - Omni-Path アーキテクチャ, DDN IME (Burst Buffer)

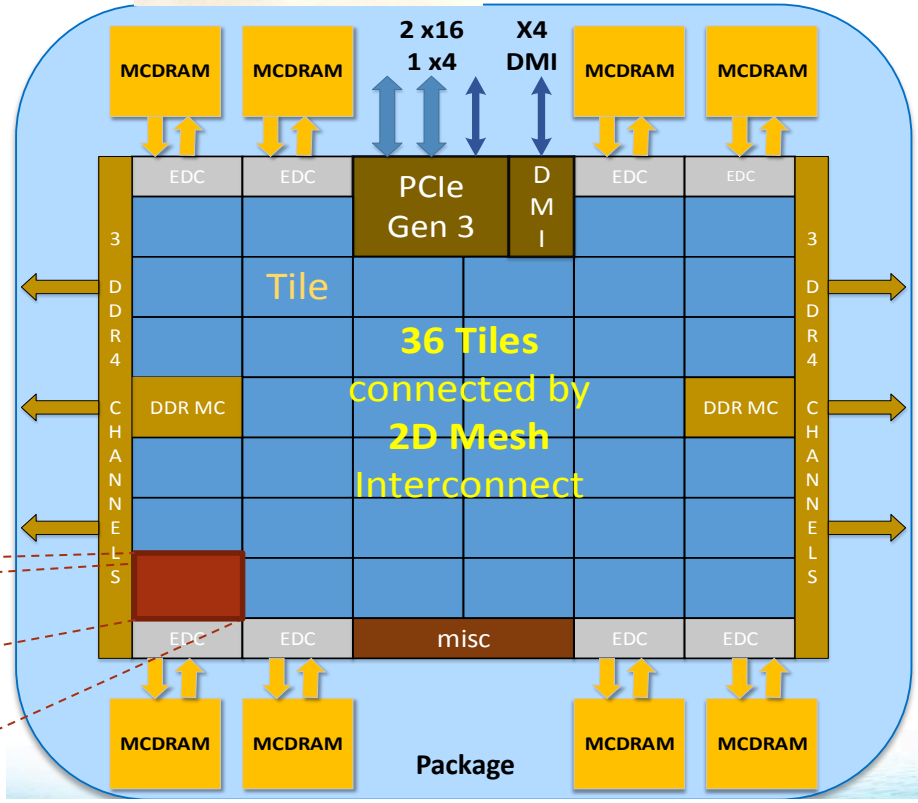
Oakforest-PACS (OFP)

- 2016年12月1日稼働開始
- 8,208 Intel Xeon/Phi (KNL)、ピーク性能25PFLOPS
  - 富士通が構築
- **TOP 500　9位（国内2位），HPCG 6位（国内2位）（2017年11月）**
- **最先端共同HPC 基盤施設(JCAHPC: Joint Center for Advanced High Performance Computing)**
  - 筑波大学計算科学研究センター
  - 東京大学情報基盤センター
  - 東京大学柏キャンパスの東京大学情報基盤センター内に、両機関の教職員が中心となって設計するスーパーコンピュータシステムを設置し，最先端の大規模高性能計算基盤を構築・運営するための組織
  - http://jcahpc.jp

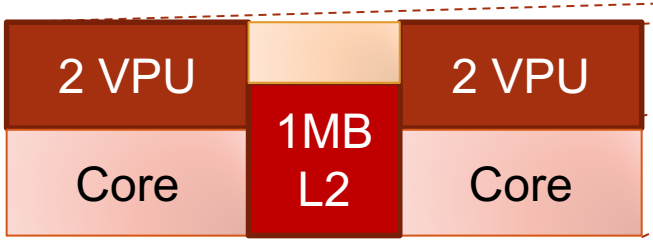
Oakforest-PACS 計算ノード

- Intel Xeon Phi (Knights Landing)
  - 1ノード1ソケット, 68コア
- MCDRAM: オンパッケージの高バンド幅メモリ16GB + DDR4メモリ 16GBx6 = 16 + 96 GB

HotChips27
KNLスライドより



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



スパコンへのログイン・テストプログラム起動

別紙：Oakforest-PACS利用の手引きを参照

ユーザアカウント

- 本講習会でのユーザ名
  - 利用者番号： t00871～
  - 利用グループ：gt00
- 利用期限
  - 5/26 17:00まで有効

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-ofp.tar.gz

- 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

  /work/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. cd コマンドを実行して Lustreファイルシステムに移動する
   $ cd /work/gt00/t008XX (下線部は自分のIDに変えること)
2. /work/gt00/z30105 にある Samples-ofp.tar.gz を
   自分のディレクトリにコピーする
   $ cp /work/gt00/z30105/Samples-ofp.tar.gz ./
3. Samples-ofp.tar を展開する
   $ tar xvfz Samples-ofp.tar.gz
4. Samplesディレクトリに入る
   $ cd Samples
5. C言語 : $ cd C
   Fortran90言語 : $ cd F
6. Helloディレクトリに入る
   $ 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 (Makefile_pure)を使ってmakeする
   ```
   $ make -f Makefile_pure
   ```
7. 実行ファイル(hello)ができていることを確認する
   ```
   $ ls
   ```

JOBスクリプトサンプルの説明（フラットMPI）

（hello-pure.bash, C言語、Fortran言語共通）

```
#!/bin/bash
#PJM -L rscgrp=lecture-flat
#PJM -L node=16
#PJM --mpi proc=1088
#PJM -L elapse=0:01:00
#PJM -g gt00

mpiexec.hydra -n
${PJM_MPI_PROC} ./hello
```

リソースグループ
：lecture-flat

利用ノード数、
MPIプロセス数

実行時間制限
：1分

利用グループ名
：gt00

MPIジョブを68*16 = 1088 プロセスで実行する。

ピュアMPIの実行状況（ノード内）

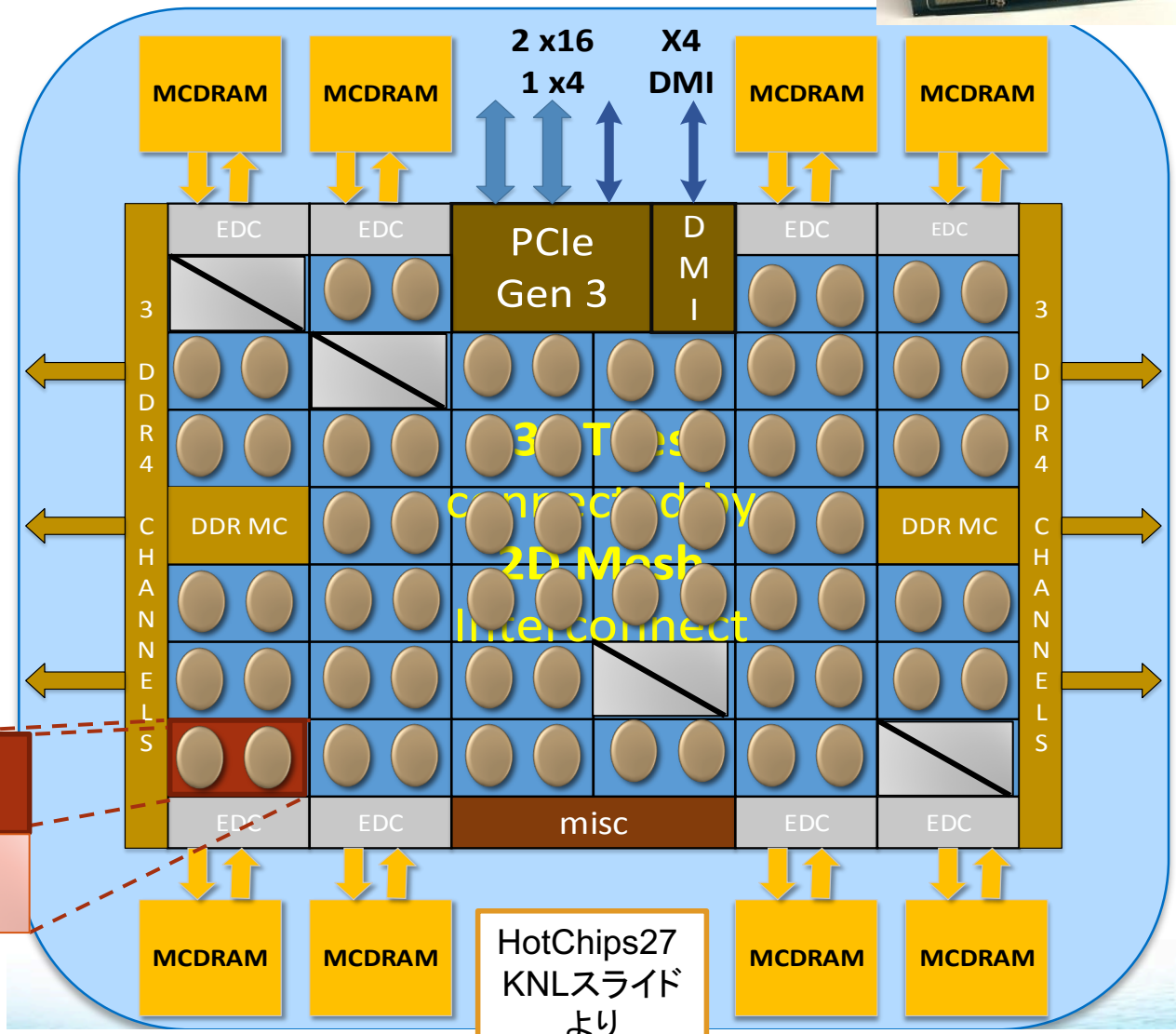
MPIプロセス

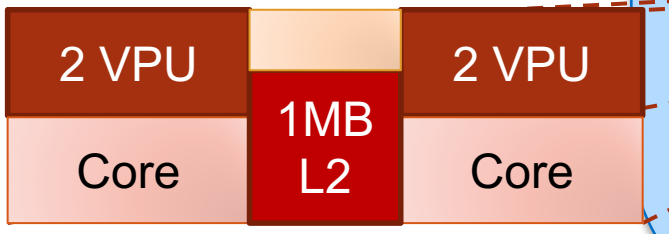
無効のタイル(例)

- Intel Xeon Phi (Knights Landing)
  - 1ノード1ソケット, 68コア

MCDRAM: オンパッケージの高バンド幅メモリ16GB
+ DDR4メモリ 16GBx6
= 16 + 96 GB

MCDRAM: 490GB/秒以上（実測）
DDR4: 115.2 GB/秒
=(8Byte×2400MHz×6 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”lecture-flat”になっています
- $ emacs hello-pure.bash で、“lecture-flat” → “tutorial-flat” 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、
   68プロセス*16ノード=1088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1088、もしくは16）

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
  use mpi
  implicit none
  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 program main
```

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
1088、もしくは16）

MPIの終了

参考：結果の確認方法

- 出力が多すぎて正しいか簡単にはわからない…

1. Hello parallel worldの個数を数える

   $ grep Hello hello-pure.bash.oXXXX | wc -l

   1088と表示されればOK!

2. myidが連続しているかどうか、ばらばらに出力されるのでわからない

   $ grep Hello hello-pure.bash.oXXXX | sort | less

   Myid: 0 から始まり、Myid: 1087で終われればよい

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。
- Oakforest-PACS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　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

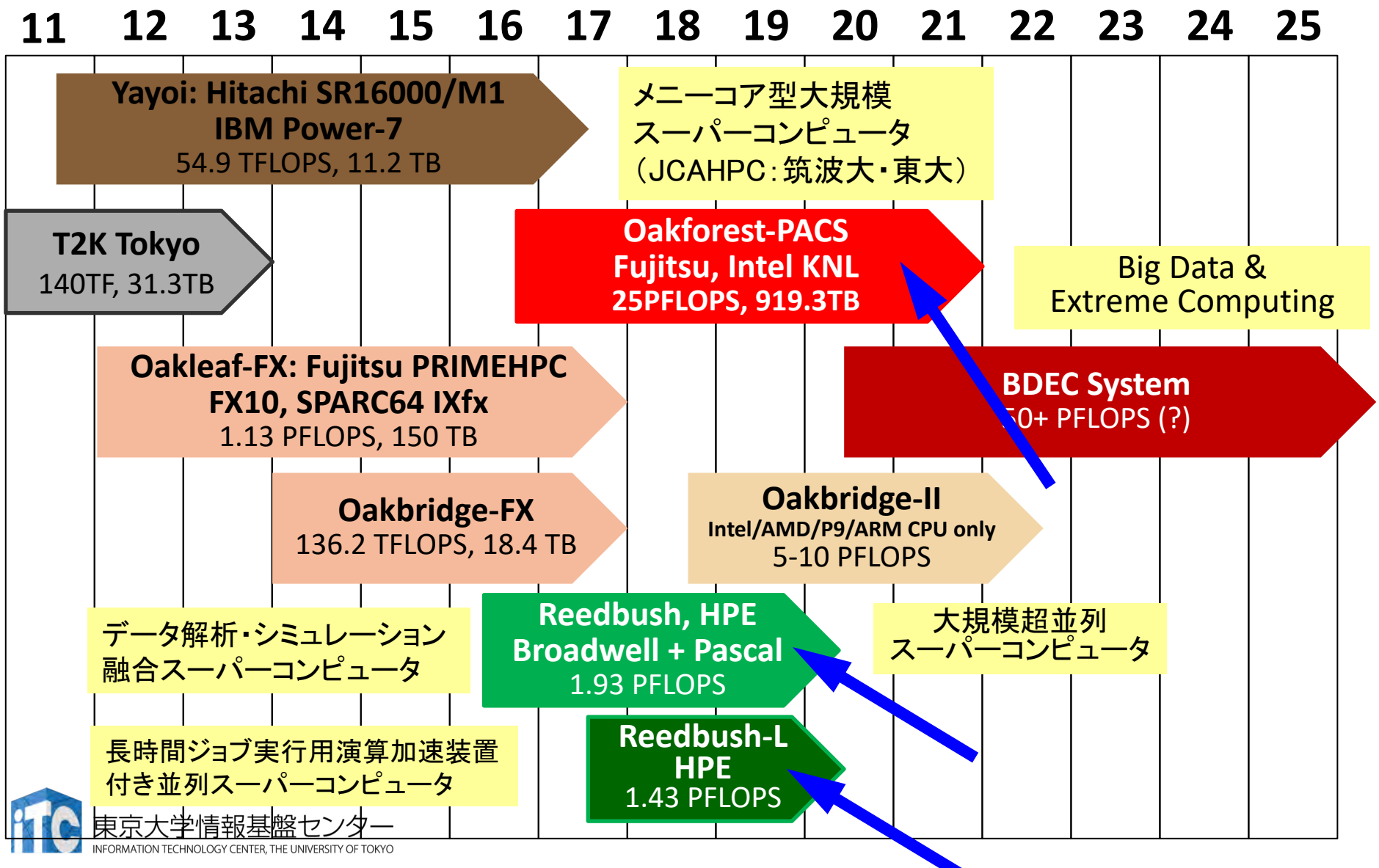
本講義の流れ

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

東大スーパーコンピュータの概略

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（だった）



2(または3,4)システム運用中

- 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 (HPE, 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
    - 2016-Jun.2016年7月～2020年6月
  - 東大情基セ初のGPU搭載システム
  - Reedbush-U: CPU only, 420 nodes, 508 TF (2016年7月)
  - Reedbush-H: 120 nodes, 2 GPUs/node: 1.42 PF (2017年3月)
  - Reedbush-L: 64 nodes, 4 GPUs/node: 1.43 PF (2017年10月)

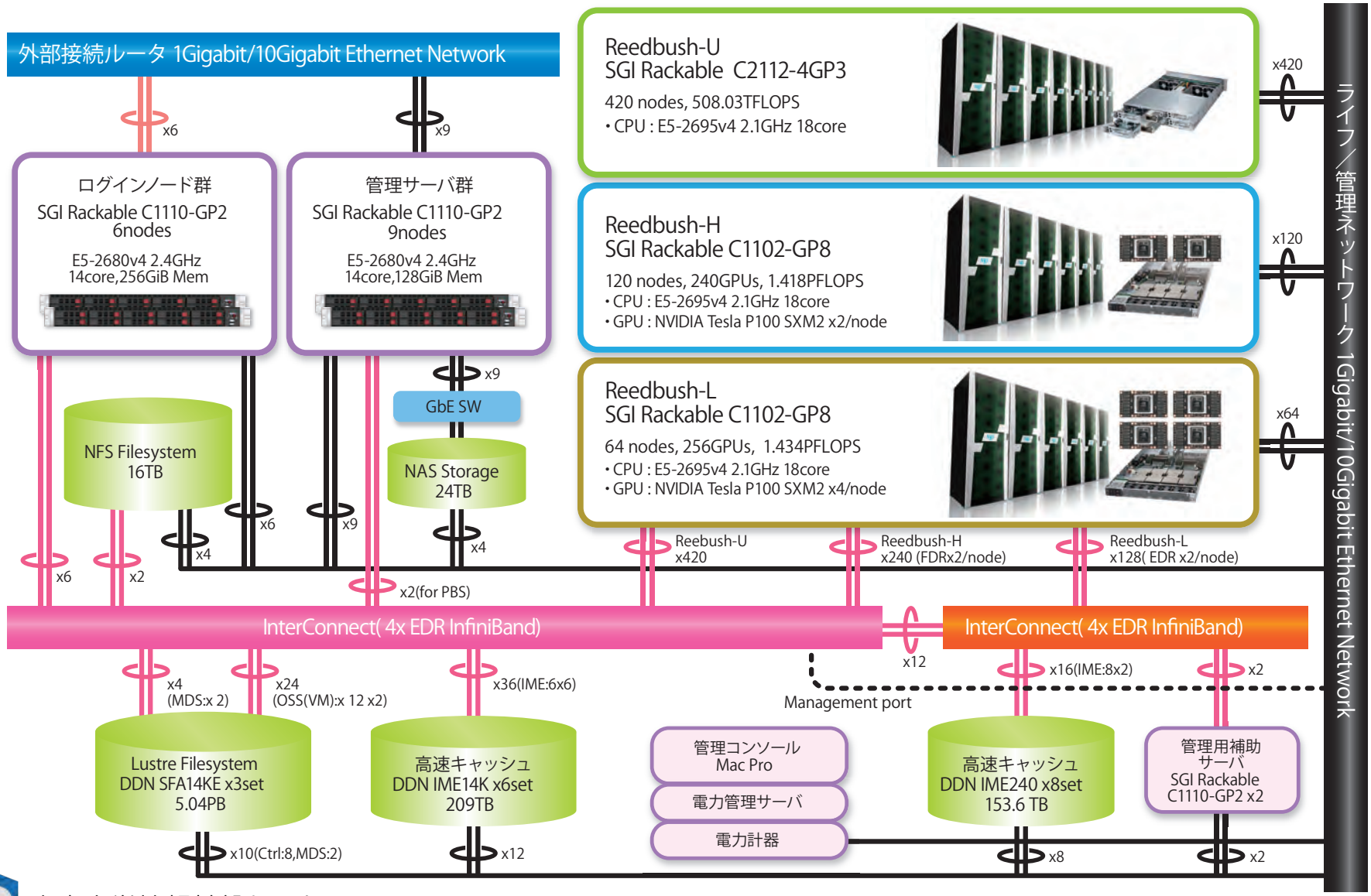
- **Oakforest-PACS (OFP)** (富士通, Intel Xeon Phi (KNL))
  - JCAHPC (筑波大CCS＆東大ITC)
  - 25 PF, 世界第9位 (2017年11月) (日本第2位)
  - Omni-Path アーキテクチャ, DDN IME (Burst Buffer)

Reedbushシステム

Top500: RB-L 291位@Nov. 2017
RB-H 203位@Jun. 2017
RB-U 361位@Nov. 2016
Green500: RB-L 11位@Nov. 2017
RB-H 11位@Jun. 2017

Reedbush-U
2016年7月1日 試験運転開始
2016年9月1日 正式運用開始
Reedbush-H
2017年3月1日 試験運転開始
2017年4月3日 正式運用開始
Reedbush-L
2017年10月2日 試験運転開始
2017年11月1日 正式運用開始

東京大学情報基盤センター　Reedbushシステム



Reedbushのサブシステム

	Reedbush-U	Reedbush-H	Reedbush-L
CPU/node	Intel Xeon E5-2695v4 (Broadwell-EP, 2.1GHz, 18core) x 2 sockets (1.210 TF), 256 GiB (153.6GB/sec)		
GPU	-	NVIDIA Tesla P100 (Pascal, 5.3TF, 720GB/sec, 16GiB)	
Infiniband	EDR	FDR×2ch	EDR×2ch
ノード数	420	120	64
GPU数	-	240 (=120×2)	256 (=64×4)
ピーク性能 (TFLOPS)	509	1,417 (145 + 1,272)	1,433 (76.8 + 1,358)
メモリバンド幅 (TB/sec)	64.5	191.2 (18.4+172.8)	194.2 (9.83+184.3)
運用開始	2016.07	2017.03	2017.10

Oakforest-PACS（OFP）

- 2016年12月1日稼働開始
- 8,208 Intel Xeon/Phi (KNL)，ピーク性能25PFLOPS
  - 富士通が構築
- **TOP 500　9位（国内2位），HPCG 6位（国内2位）（2017年11月）**
- **最先端共同HPC 基盤施設(JCAHPC: Joint Center for Advanced High Performance Computing)**
  - 筑波大学計算科学研究センター
  - 東京大学情報基盤センター
    - 東京大学柏キャンパスの東京大学情報基盤センター内に、両機関の教職員が中心となって設計するスーパーコンピュータシステムを設置し，最先端の大規模高性能計算基盤を構築・運営するための組織
  - http://jcahpc.jp

JCAHPC

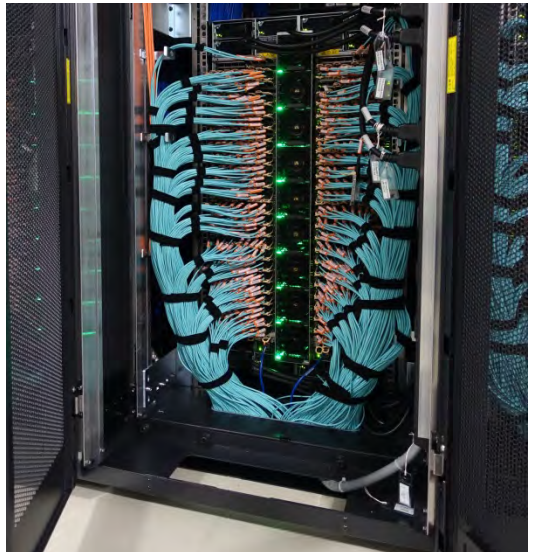
東京大学情報基盤センター
INFORMATION TECHNOLOGY CENTER, THE UNIVERSITY OF TOKYO

東京大学 THE UNIVERSITY OF TOKYO

筑波大学 University of Tsukuba

Oakforest-PACSの特徴 (1/2)

- 計算ノード
  - 1ノード 68コア，3TFLOPS×8,208ノード＝ 25 PFLOPS
  - メモリ（MCDRAM（高速，16GB）＋DDR4（低速，96GB））
- ノード間通信
  - フルバイセクションバンド幅を持つFat-Treeネットワーク
  - 全系運用時のアプリケーション性能に効果，多ジョブ運用
  - Intel Omni-Path Architecture

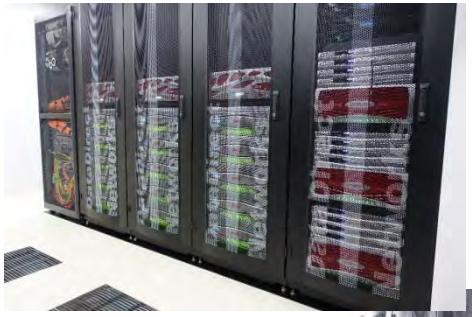


Oakforest-PACS の仕様

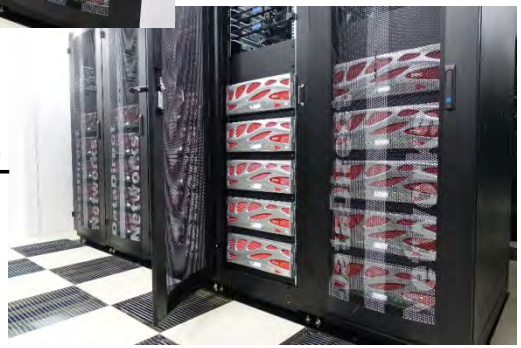
総ピーク演算性能			25 PFLOPS
ノード数			8,208
計算ノード	Product		富士通 PRIMERGY CX600 M1 (2U) + CX1640 M1 x 8node
	プロセッサ		Intel® Xeon Phi™ 7250（開発コード: Knights Landing）68 コア、1.4 GHz
	メモリ	高バンド幅	16 GB, MCDRAM, 実効 490 GB/sec
		低バンド幅	96 GB, DDR4-2400, ピーク 115.2 GB/sec
相互結合網	Product		Intel® Omni-Path Architecture
	リンク速度		100 Gbps
	トポロジ		フルバイセクションバンド幅Fat-tree網

Oakforest-PACS の特徴(2 / 2)

- ファイルI/O
  - 並列ファイルシステム:
    Lustre 26PB
  - ファイルキャッシュシステム
    (DDN IME):
    1TB/secを超える実効性能,
    約1PB
    - 計算科学・ビッグデータ解析・機械学習にも貢献
- 消費電力
  - Green 500でも世界6位
    (2016.11)
  - Linpack： 2.72 MW
    - 4,986 MFLOPS/W(OFP)
    - 830 MFLOPS/W(京)



並列ファイル
システム



ファイルキャッシュ
システム

ラック当たり120ノード
の高密度実装

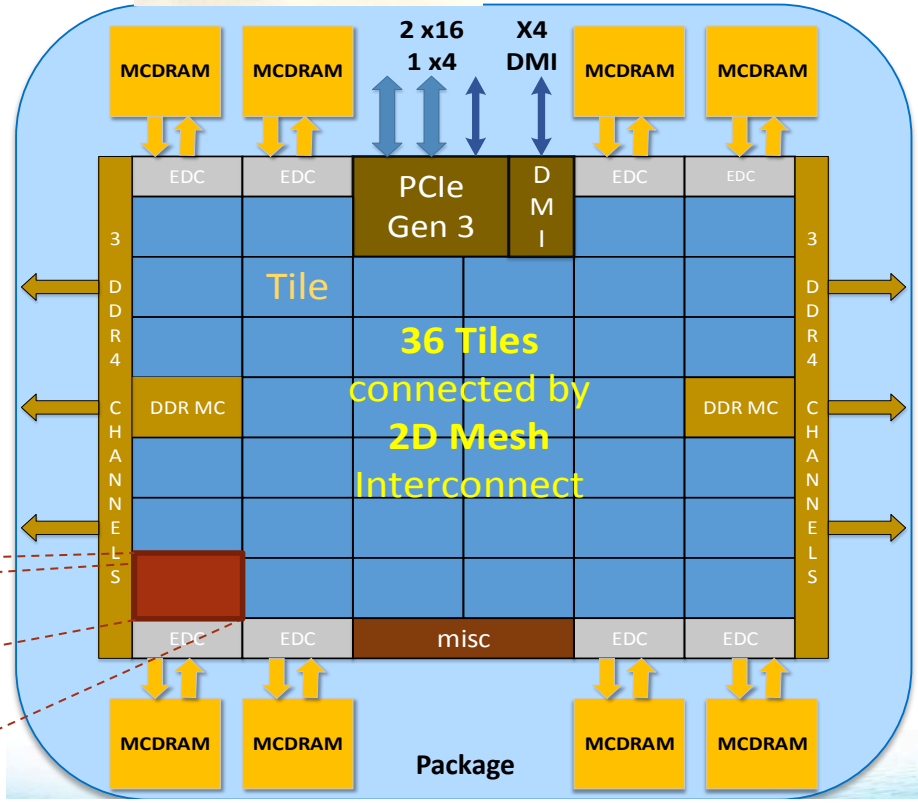
Oakforest-PACS の仕様（続き）

並列ファイルシステム	Type	Lustre File System
	総容量	26.2 PB
	Product	DataDirect Networks SFA14KE
	総バンド幅	500 GB/sec
高速ファイルキャッシュシステム	Type	Burst Buffer, Infinite Memory Engine (by DDN)
	総容量	940 TB (NVMe SSD, パリティを含む)
	Product	DataDirect Networks IME14K
	総バンド幅	1,560 GB/sec
総消費電力		4.2MW（冷却を含む）
総ラック数		102

Oakforest-PACS 計算ノード

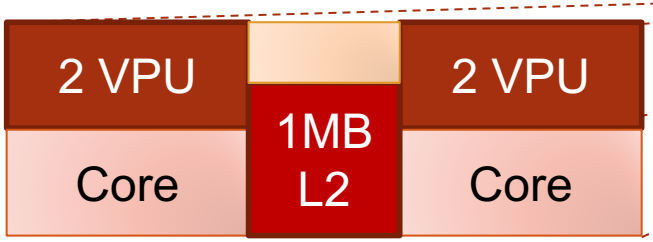
- Intel Xeon Phi (Knights Landing)
  - 1ノード1ソケット
- MCDRAM: オンパッケージの高バンド幅メモリ16GB + DDR4メモリ

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

50th TOP500 List (Nov., 2017)

	Site	Computer/Year Vendor	Cores	R_{max} (TFLOPS)	R_{peak} (TFLOPS)	Power (kW)
1	National Supercomputing Center in Wuxi, China	**Sunway TaihuLight**, Sunway MPP, Sunway SW26010 260C 1.45GHz, 2016 NRCPC	10,649,600	93,015 (= 93.0 PF)	125,436	15,371
2	National Supercomputing Center in Tianjin, China	**Tianhe-2**, Intel Xeon E5-2692, TH Express-2, Xeon Phi, 2013 NUDT	3,120,000	33,863 (= 33.9 PF)	54,902	17,808
3	Swiss Natl. Supercomputer Center, Switzerland	**Piz Daint** Cray XC30/NVIDIA P100, 2013 Cray	361,760	19,590	33,863	2,272
4	JAMSTEC	**Gyoukou** ZettaScaler-2.2, PEZY-SC2	19,860,000	19,135.8	28,192	1,350
5	Oak Ridge National Laboratory, USA	**Titan** Cray XK7/NVIDIA K20x, 2012 Cray	560,640	17,590	27,113	8,209
6	Lawrence Livermore National Laboratory, USA	**Sequoia** BlueGene/Q, 2011 IBM	1,572,864	17,173	20,133	7,890
7	DOE/NNSA/LANL/SNL, USA	**Trinity**, Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, 2017 Cray	979,968	14,137	43,902	3,844
8	DOE/SC/LBNL/NERSC USA	**Cori**, Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Cray Aries, 2016 Cray	632,400	14,015	27,881	3,939
9	Joint Center for Advanced High Performance Computing, Japan	**Oakforest-PACS**, PRIMERGY CX600 M1, Intel Xeon Phi Processor 7250 68C 1.4GHz, Intel Omni-Path, 2016 Fujitsu	557,056	13,555	24,914	2,719
10	RIKEN AICS, Japan	**K computer**, SPARC64 VIIIfx , 2011 Fujitsu	705,024	10,510	11,280	12,660

R_{max}: Performance of Linpack (TFLOPS)
R_{peak}: Peak Performance (TFLOPS), Power: kW

http://www.top500.org/

HPCG Ranking (Nov, 2017)

	Computer	Cores	HPL Rmax (Pflop/s)	TOP500 Rank	HPCG (Pflop/s)	Peak
1	**K computer**	705,024	10.510	10	0.6027	5.3%
2	**Tianhe-2 (MilkyWay-2)**	3,120,000	33.863	2	0.5801	1.1%
3	**Trinity**	979,072	14.137	7	0.546	1.8%
4	**Piz Daint**	361,760	19.590	3	0.486	1.9%
5	**Sunway TaihuLight**	10,649,600	93.015	1	0.4808	0.4%
6	**Oakforest-PACS**	557,056	13.555	9	0.3855	1.5%
7	**Cori**	632,400	13.832	8	0.3554	1.3%
8	**Sequoia**	1,572,864	17.173	6	0.3304	1.6%
9	**Titan**	560,640	17.590	5	0.3223	1.2%
10	**TSUBAME3.0**	136,080	8.125	13	0.189	1.6%

http://www.hpcg-benchmark.org/

Green 500 Ranking (SC16, November, 2016)

	Site	Computer	CPU	HPL Rmax (Pflop/s)	TOP500 Rank	Power (MW)	GFLOPS/W
1	NVIDIA Corporation	DGX SATURNV	NVIDIA DGX-1, Xeon E5-2698v4 20C 2.2GHz, Infiniband EDR, NVIDIA Tesla P100	3.307	28	0.350	9.462
2	Swiss National Supercomputing Centre (CSCS)	Piz Daint	Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100	9.779	8	1.312	7.454
3	RIKEN ACCS	Shoubu	ZettaScaler-1.6 etc.	1.001	116	0.150	6.674
4	National SC Center in Wuxi	Sunway TaihuLight	Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway	93.01	1	15.37	6.051
5	SFB/TR55 at Fujitsu Tech. Solutions GmbH	QPACE3	PRIMERGY CX1640 M1, Intel Xeon Phi 7210 64C 1.3GHz, Intel Omni-Path	0.447	375	0.077	5.806
6	JCAHPC	Oakforest-PACS	PRIMERGY CX1640 M1, Intel Xeon Phi 7250 68C 1.4GHz, Intel Omni-Path	1.355	6	2.719	4.986
7	DOE/SC/Argonne National Lab.	Theta	Cray XC40, Intel Xeon Phi 7230 64C 1.3GHz, Aries interconnect	5.096	18	1.087	4.688
8	Stanford Research Computing Center	XStream	Cray CS-Storm, Intel Xeon E5-2680v2 10C 2.8GHz, Infiniband FDR, Nvidia K80	0.781	162	0.190	4.112
9	ACCMS, Kyoto University	Camphor 2	Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect	3.057	33	0.748	4.087
10	Jefferson Natl. Accel. Facility	SciPhi XVI	KOI Cluster, Intel Xeon Phi 7230 64C 1.3GHz, Intel Omni-Path	0.426	397	0.111	3.837

http://www.top500.org/

IO 500 Ranking (Nov., 2017)

	Site	Computer	File system	Client nodes	IO500 Score	BW (GiB/s)	MD (kIOP/s)
1	JCAHPC, Japan	Oakforest-PACS	DDN IME	204	101.48	471.25	21.85
2	KAUST, Saudi	Shaheen2	Cray DataWarp	300	70.90	151.53	33.17
3	KAUST, Saudi	Shaheen2	Lustre	1000	41.00	54.17	31.03
4	JSC, Germany	JURON	BeeGFS	8	35.77	14.24	89.83
5	DKRZ, Germany	Mistral	Lustre	100	32.15	22.77	45.39
6	IBM, USA	Sonasad	Spectrum Scale	10	21.63	4.57	102.38
7	Fraunhofer, Germany	Seislab	BeeGFS	24	18.75	5.13	68.58
8	PNNL, USA	EMSL Cascade	Lustre	126	11.17	4.88	25.57
9	SNL, USA	Serrano	Spectrum Scale	16	4.25	0.65	27.98

http://www.io500.org/

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（2018年4月1日 ）

- パーソナルコース
  - 100,000円 ： 1口8ノード(基準) 3口まで、最大2048ノードまで
    トークン：2ノード x 24時間 x 360日分（1口）
- 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
    トークン：8ノード x 24時間 x 360日分（1口）
- 以上は、「トークン制」で運営
  - 基準ノード数までは、トークン消費係数が1.0
  - 基準ノード数を超えると、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（2018年4月1日 ）

- パーソナルコース
  - 150,000円 ： RB-U: 4ノード（基準）、最大128ノードまで
    RB-H: 1ノード（基準、係数はUの2.5倍）、最大 32ノードまで
    RB-L: 1ノード（基準、係数はUの4倍）、最大 16ノードまで
- グループコース
  - 300,000円: RB-U 1口 4ノード（基準）、最大128ノードまで、
    RB-H: 1ノード（基準、係数はUの2.5倍）、最大 32ノードまで
    RB-L: 1ノード（基準、係数はUの4倍）、最大 16ノードまで
  - 企業 RB-Uのみ 360,000円 ：1口 4ノード（基準）、最大128ノードまで
  - 企業 RB-Hのみ 216,000円 ：1口 1ノード（基準）、最大32ノードまで
  - 企業 RB-Lのみ 360,000円 ：1口 1ノード（基準）、最大16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 (Hはその2.5倍, Lは4倍)
  - 基準ノードを超えると、超えた分は、消費係数がさらに2倍になる
  - 大学等のユーザはOakforest-PACSとの相互トークン移行も可能
  - ノード固定もあり (Reedbush-U, L)

トライアルユース制度について

- 安価に当センターのReedbush-U/H/L, 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（1～3ヶ月）（RB-U: 最大16ノード, RB-H: 最大4ノード、RB-L: 最大4ノード、OFP: 最大16ノード）
    - グループコース（1～9ヶ月）（RB-U: 最大128ノード、RB-H: 最大32ノード、RB-L: 最大16ノード、OFP: 最大2048ノード）
  - 企業利用
    - パーソナルコース（1～3ヶ月）（RB-U: 最大16ノード, RB-H: 最大4ノード、RB-L: 最大4ノード、OFP: 最大16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（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

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

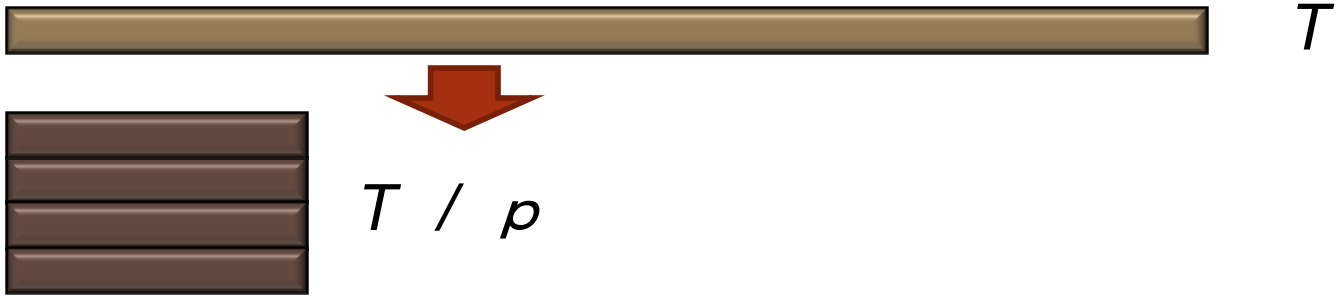
などがご覧になれます。

https://www.cc.u-tokyo.ac.jp/guide/

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

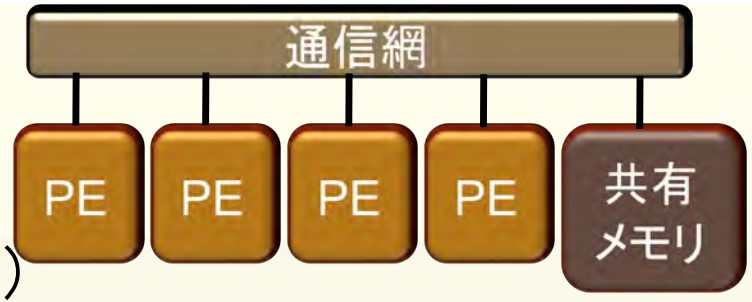
並列計算機の分類

- 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

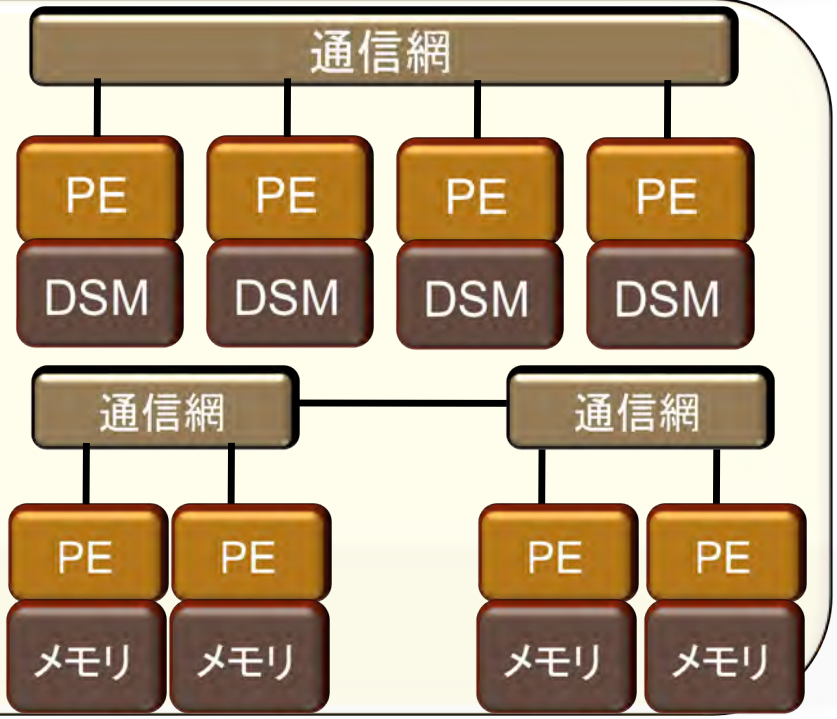
A) メモリアドレスを共有している：互いのメモリがアクセス可能

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）

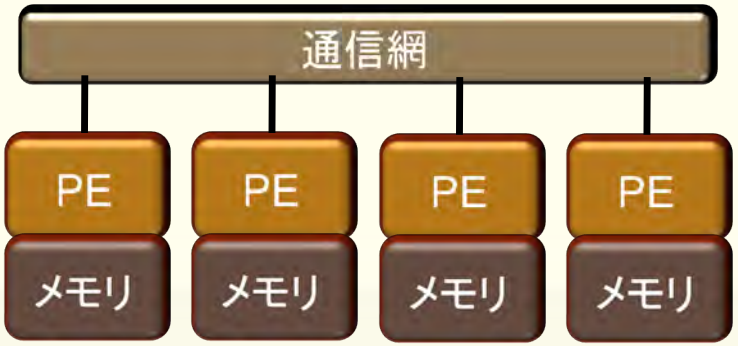
共有・非対称メモリ型
（ccNUMA、
Cache Coherent Non-Uniform
Memory Access）



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）



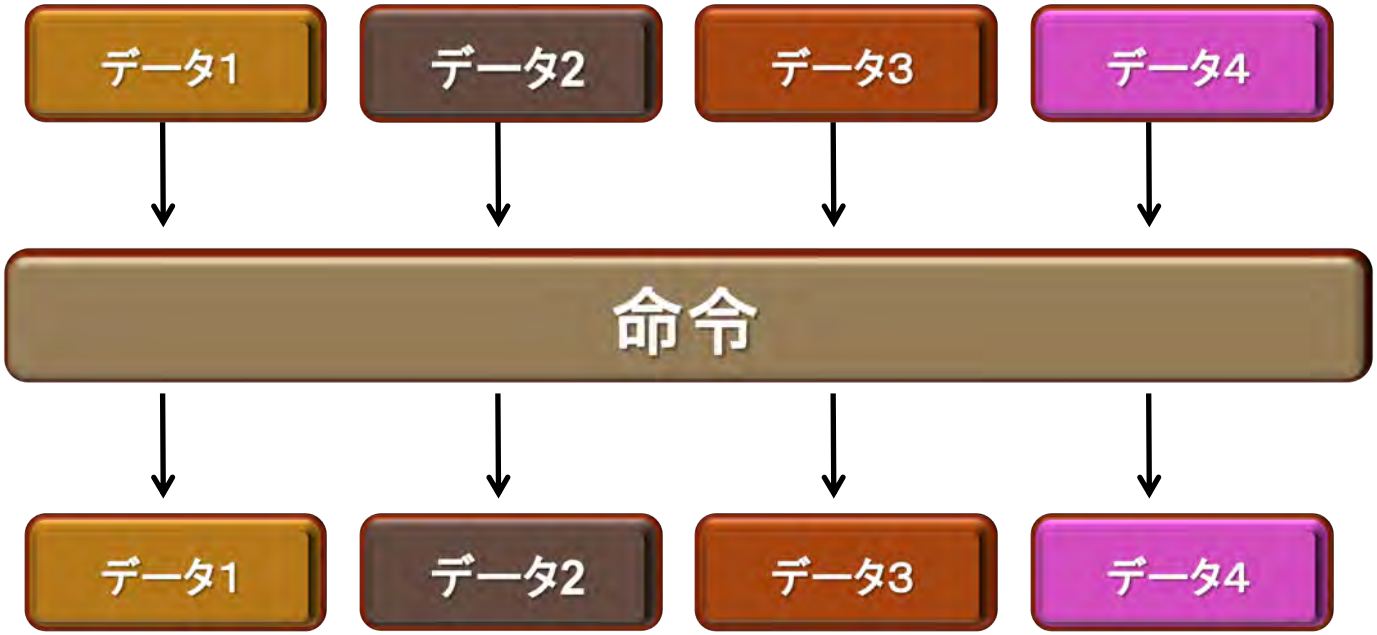
プログラミング手法から見た分類

1. マルチスレッド
   - Pthreads, …
2. データ並列
   - OpenMP
   - （最近の)Fortran
   - PGAS (Partitioned Global Address Space)言語: XcalableMP, UPC, Chapel, X10, Co-array Fortran, …
3. タスク並列
   - Cilk (Cilk plus), Thread Building Block (TBB), StackThreads, MassiveThreads, …
4. メッセージ通信
   - MPI

複数ノードにまたがる並列化に使える
他はメモリを共有していること
（共有メモリ）が前提

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

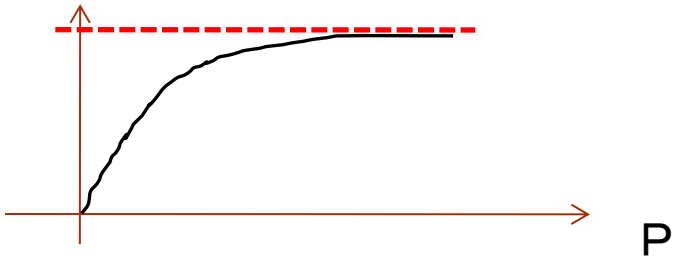
2. Master / Worker（Master / Slave）
   - 1つのプロセス（Master）が、複数のプロセス（Worker）を管理（生成、消去）する。

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 $S_P = T_S / T_P \quad (0 \leq S_p)$
  - 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： $E_P = S_P / P \times 100 \quad (0 \leq E_p)\ [\%]$
- 飽和性能
  - 速度向上の限界
  - Saturation、「さちる」



Weak ScalingとStrong Scaling

並列処理においてシステム規模を大きくする方法

- Weak Scaling: それぞれの問題サイズは変えず並列度をあげる
  - 全体の問題サイズが（並列数に比例して）大きくなる
  - 通信のオーバヘッドはあまり変わらないか、やや増加する
- Strong Scaling: 全体の問題サイズを変えずに並列度をあげる
  - 問題サイズが装置数に反比例して小さくなる
  - 通信のオーバヘッドは相対的に大きくなる

Weak Scaling

それまで解けなかった規模の問題が解ける

Strong Scalingも重要

同じ問題規模で、短時間に結果を得る（より難しい）

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。

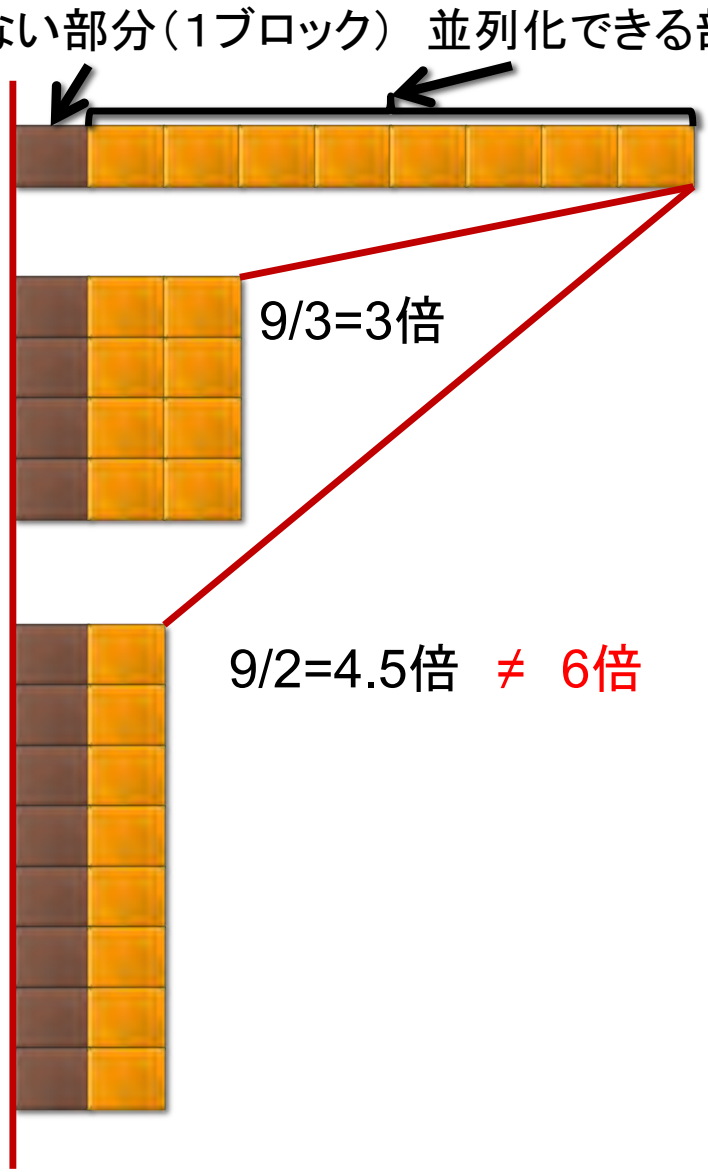
$$S_P = K/(K\alpha/P + K(1-\alpha))$$

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

- 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



Byte/Flop, B/F値

1. プログラム中で要求するメモリアクセスの割合

```
double A[N][N];
…
A[i][j] = 0.25 * (A[i][j] + A[i][j-1] + A[i][j+1] + A[i-1][j] + A[i+1][j]);
```

- メモリアクセス：8byte * 5回のロード + 8byte * 1回のストア = 48byte
- 演算：加算4回、乗算1回 = 5 FLOP　　B/F = 9.6

2. メモリシステムがデータを演算コアに供給する能力、供給できた割合

- 通常は 0.1以下、よくても0.5未満
  - B/F=0.5のシステムで上の計算をすると、96回分の計算ができるところを、5回しか動かない => ピークの5%しか使えない
- ➡ B/F値の不足をキャッシュによって補う
- ベクトル機: 伝統的にはB/F = 1くらいだった、今は 0.5とか

どちらのコンテキストで話しているかに注意！

メッセージ通信ライブラリ MPI

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

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

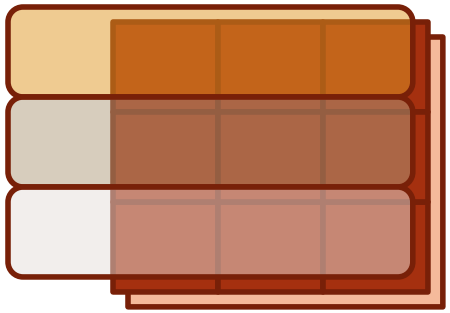
- 郵便物の郵送に同じ
- 郵送に必要な情報：
  1. 自分の住所、送り先の住所
  2. 中に入っているものはどこにあるか
  3. 中に入っているものの分類
  4. 中に入っているものの量
  5. （荷物を複数同時に送る場合の）認識方法（タグ）
- MPIでは：
  1. 自分の認識ID、および、送り先の認識ID
  2. データ格納先のアドレス
  3. データ型
  4. データ量
  5. タグ番号

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



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_REAL8 (=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）

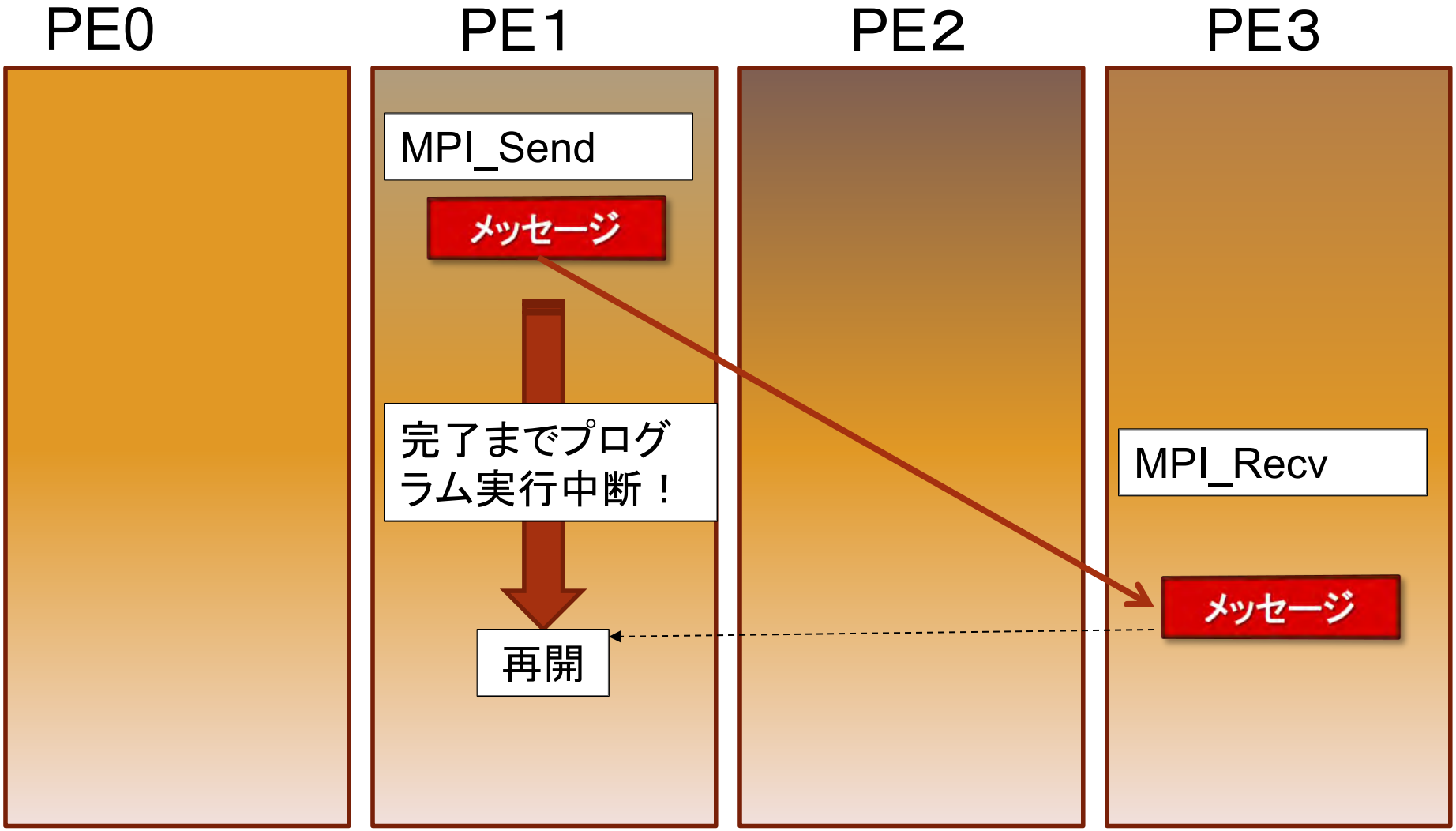
- itag： 整数型。受信したいメッセージに付いているタグの値を指定。
  - 任意のタグ値のメッセージを受信したいときは、MPI_ANY_TAG を指定。
- icomm： 整数型。PE集団を認識する番号であるコミュニケータを指定。
  - 通常ではMPI_COMM_WORLD を指定すればよい。
- istatus： MPI_Status型（整数型の配列）。受信状況に関する情報が入る。かならず専用の型宣言をした配列を確保すること。
  - 要素数がMPI_STATUS_SIZEの整数配列が宣言される。
  - 受信したメッセージの送信元のランクが istatus[MPI_SOURCE]、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

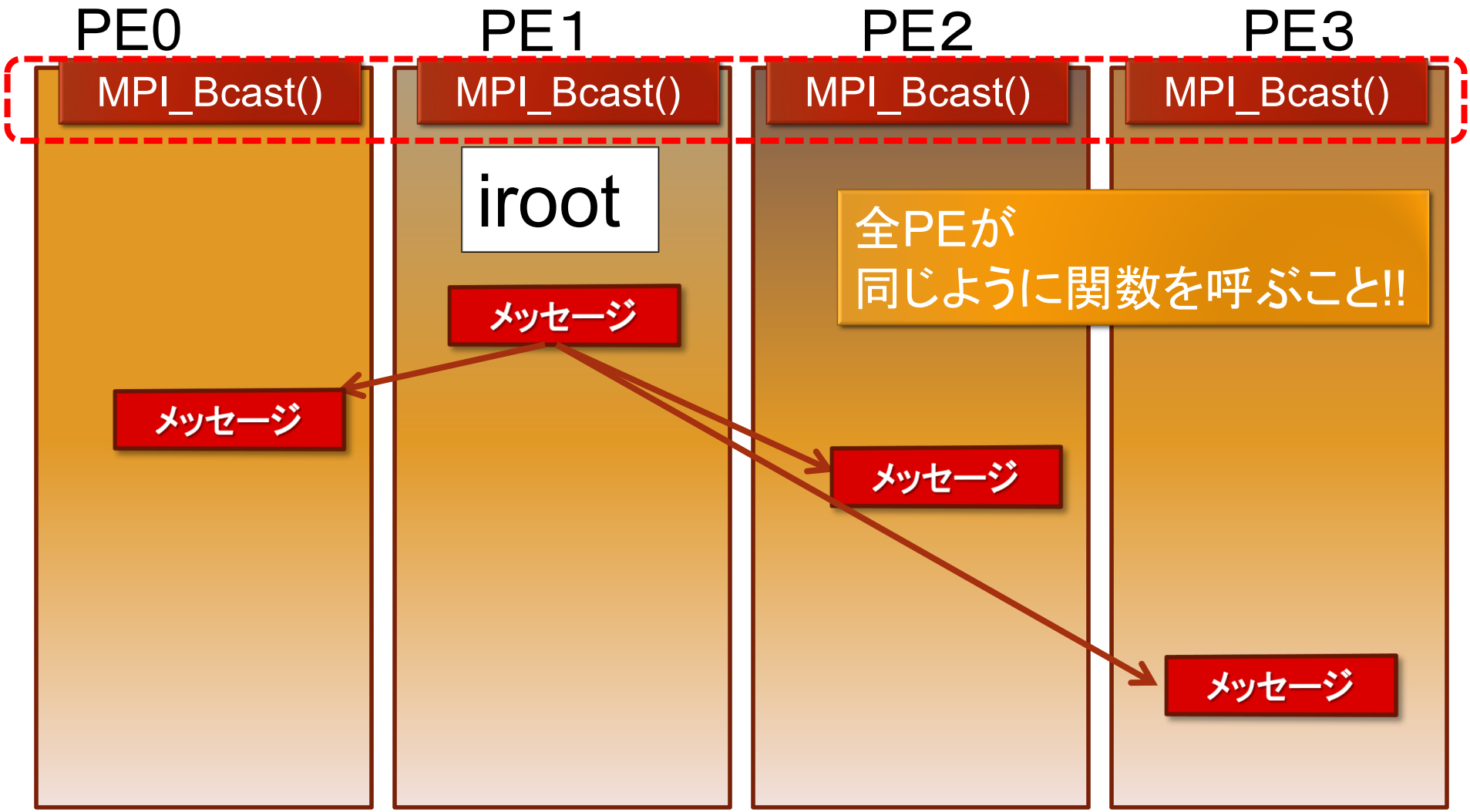


基礎的なMPI関数—MPI_Bcast

- 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）

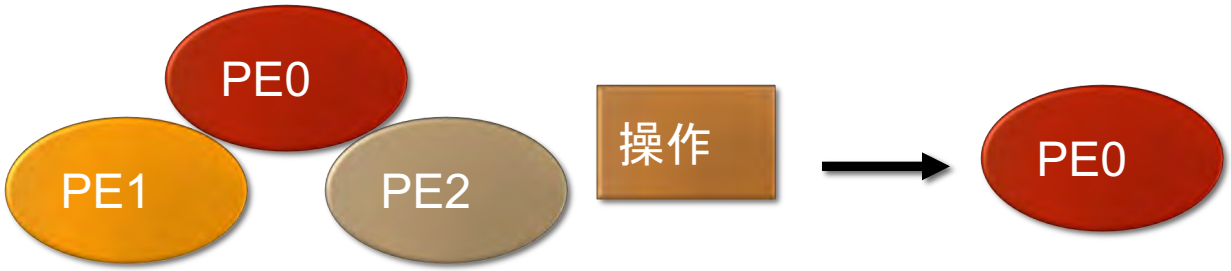


リダクション演算

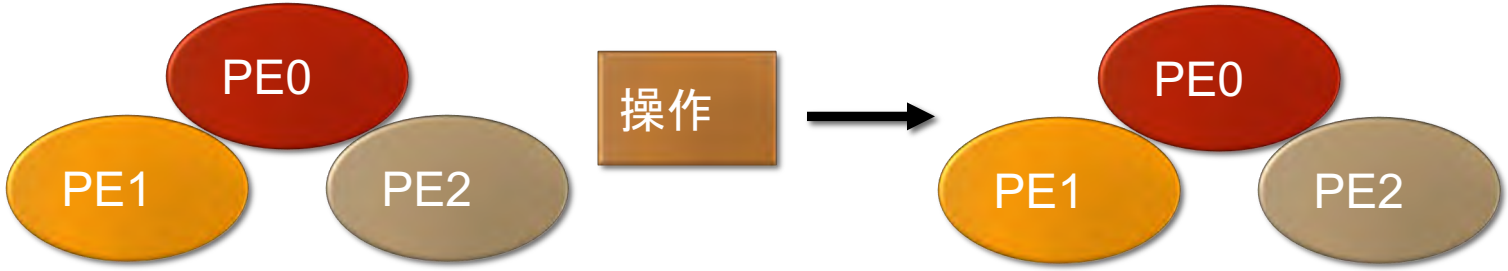
- ＜操作＞によって＜次元＞を減少（リダクション）させる処理
  - 例： 内積演算
    ベクトル（n次元空間） → スカラ（1次元空間）
- リダクション演算は、通信と計算を必要とする
  - 集団通信演算（collective communication operation）
    と呼ばれる
- 演算結果の持ち方の違いで、2種のインタフェースが存在する

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



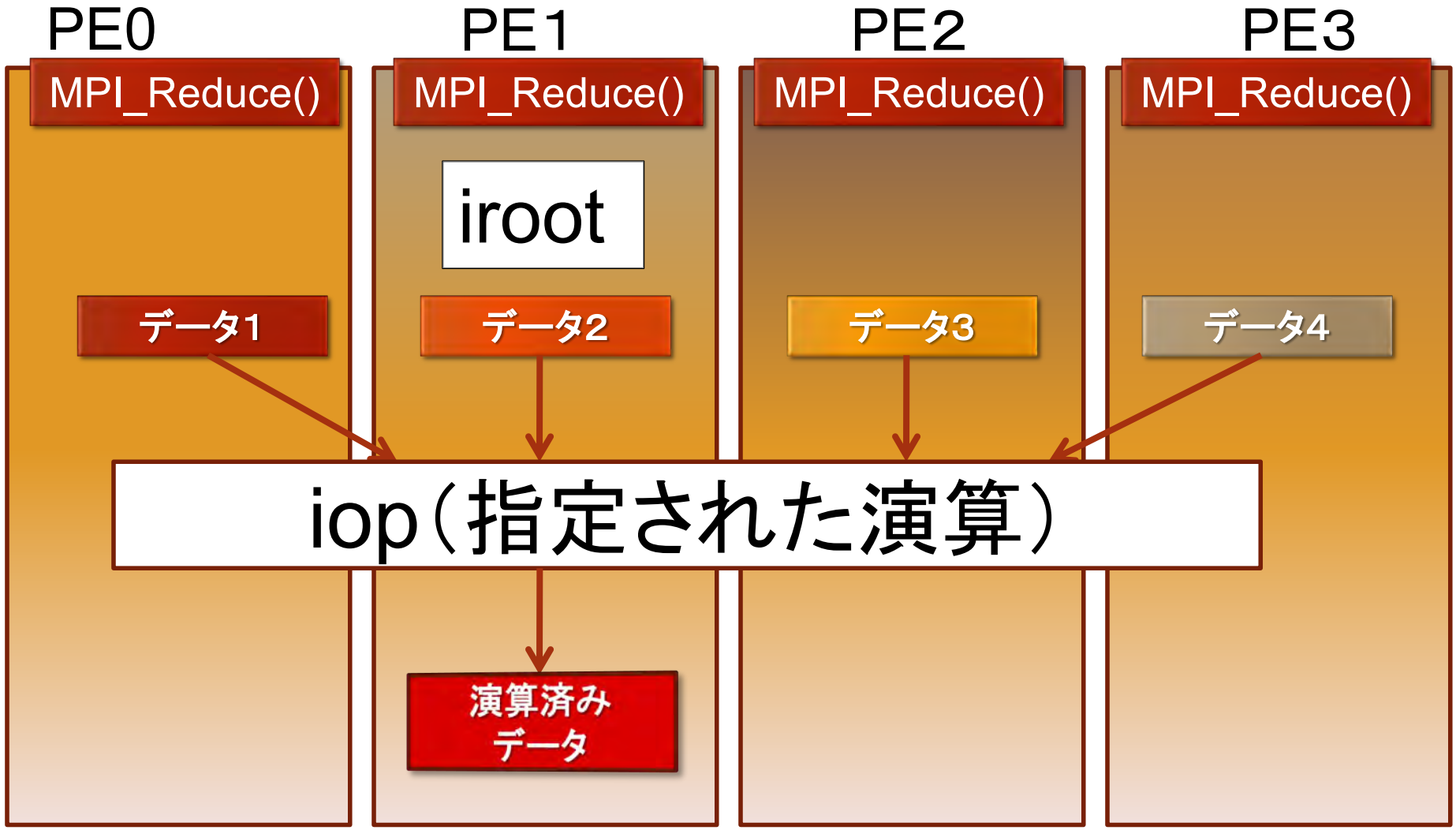
基礎的なMPI関数—MPI_Reduce

- 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_2REAL8 (=MPI_2DOUBLE_PRECISION) (倍精度型) 、を指定する。

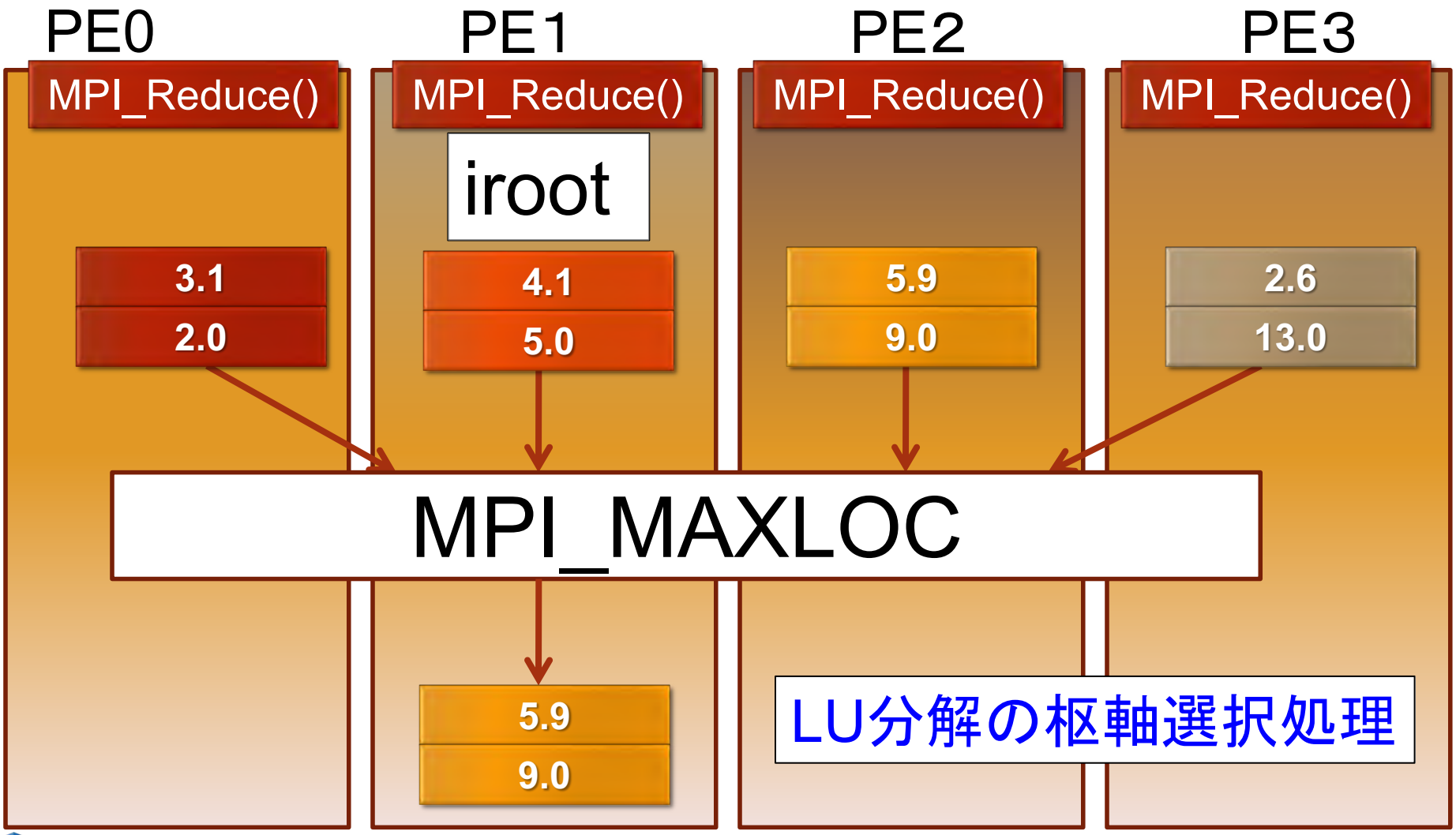
基礎的なMPI関数—MPI_Reduce

- 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）



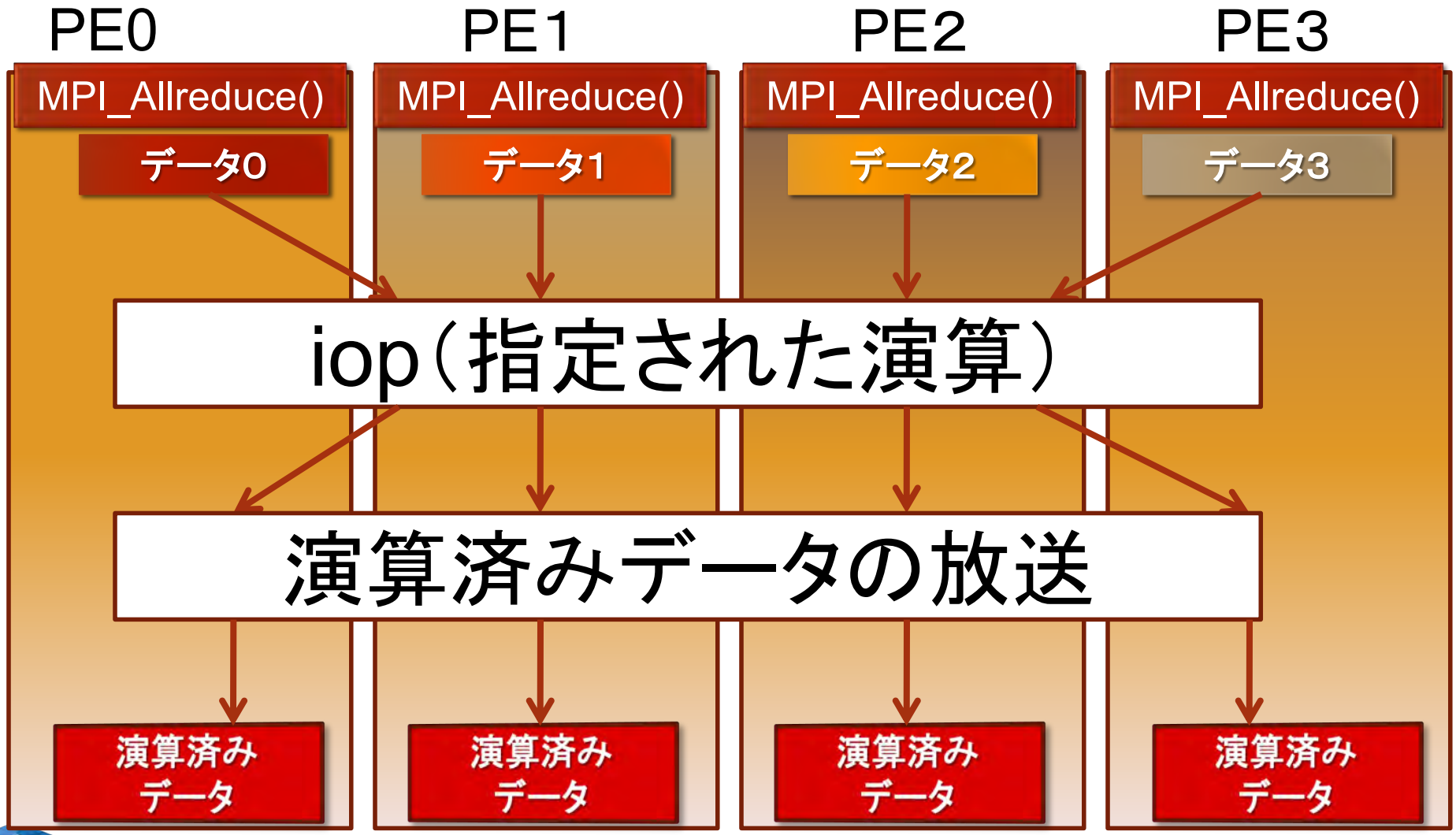
基礎的なMPI関数—MPI_Allreduce

- 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。
    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）

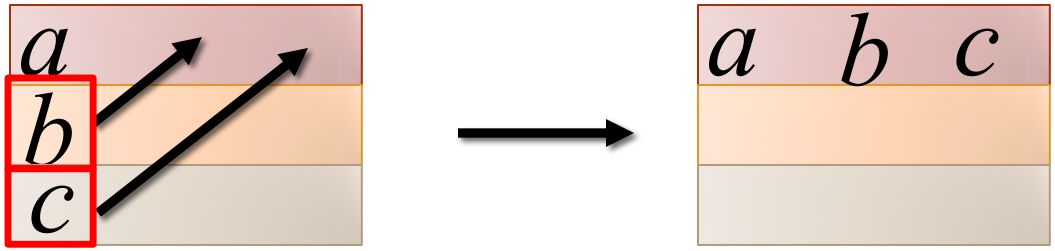


リダクション演算

- 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。

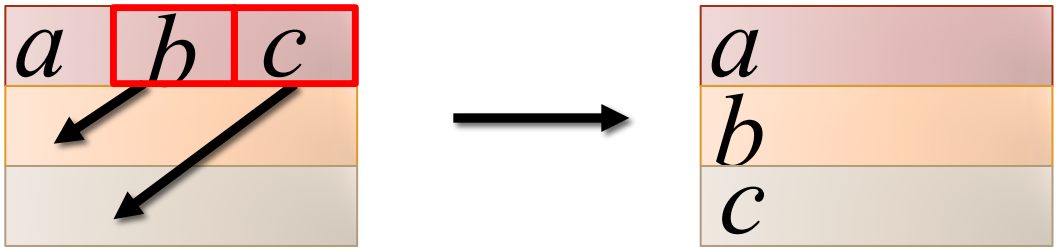
行列の転置

- 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

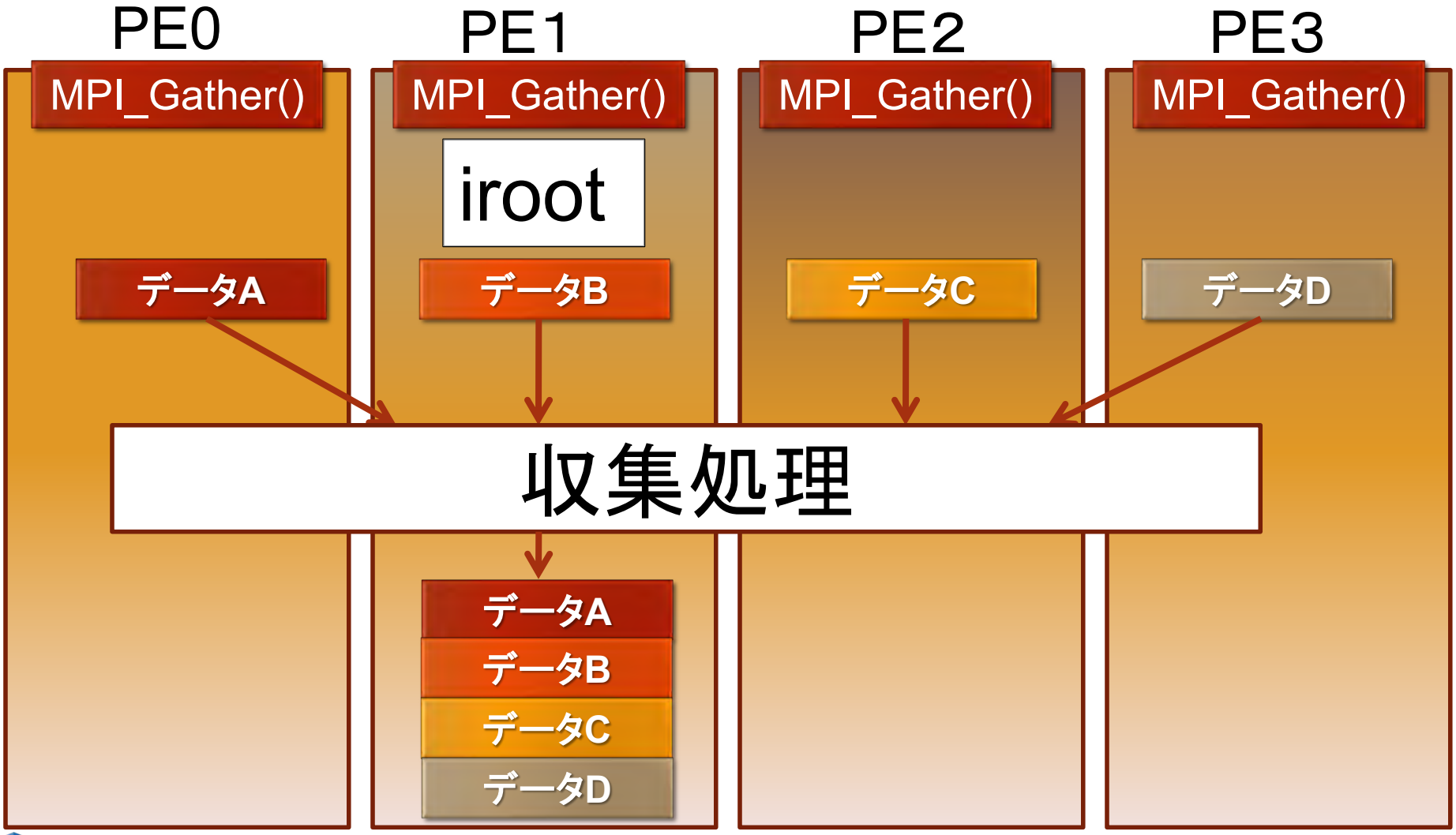
基礎的なMPI関数—MPI_Gather

- 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）



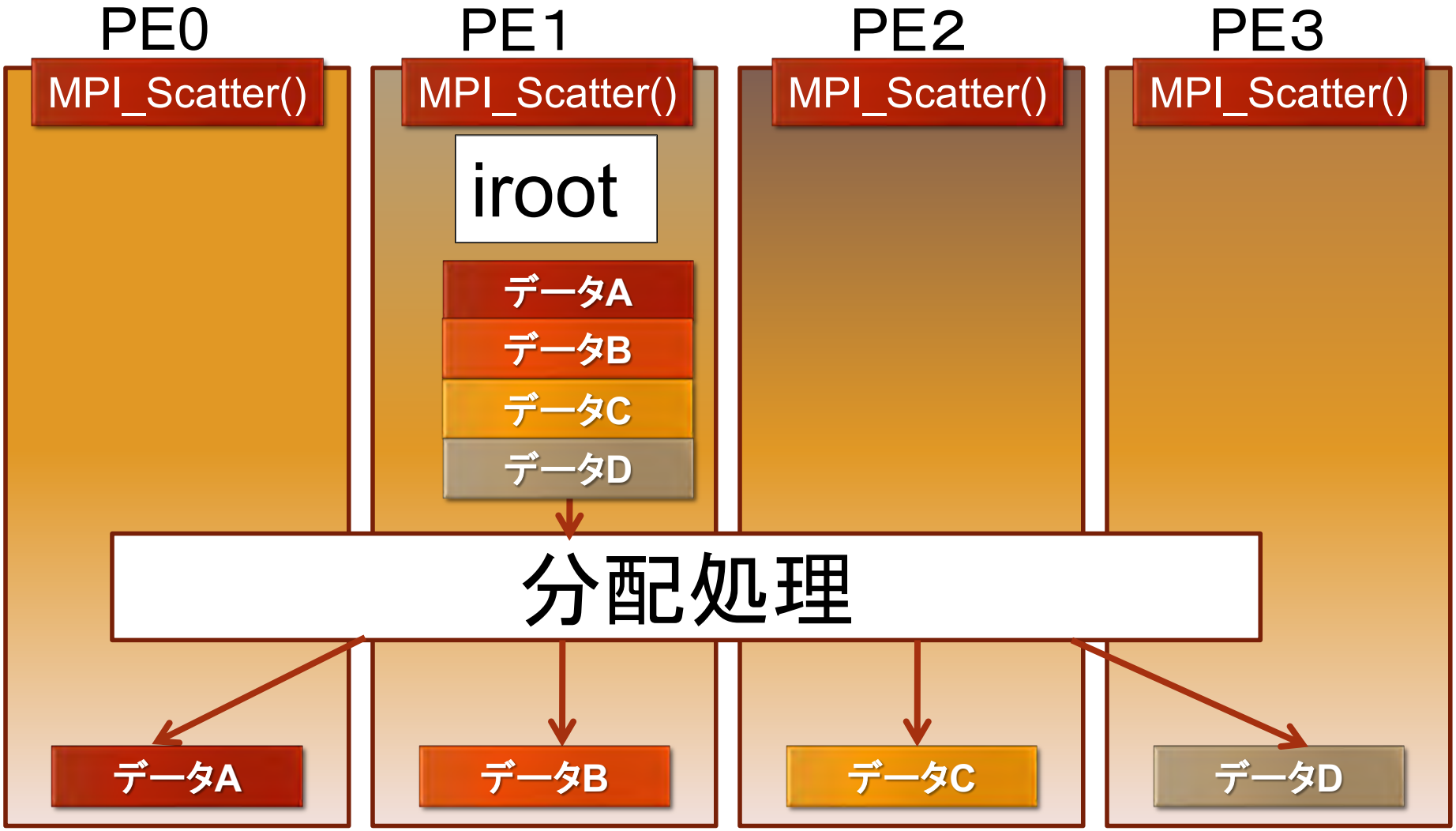
基礎的なMPI関数—MPI_Scatter

- 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。

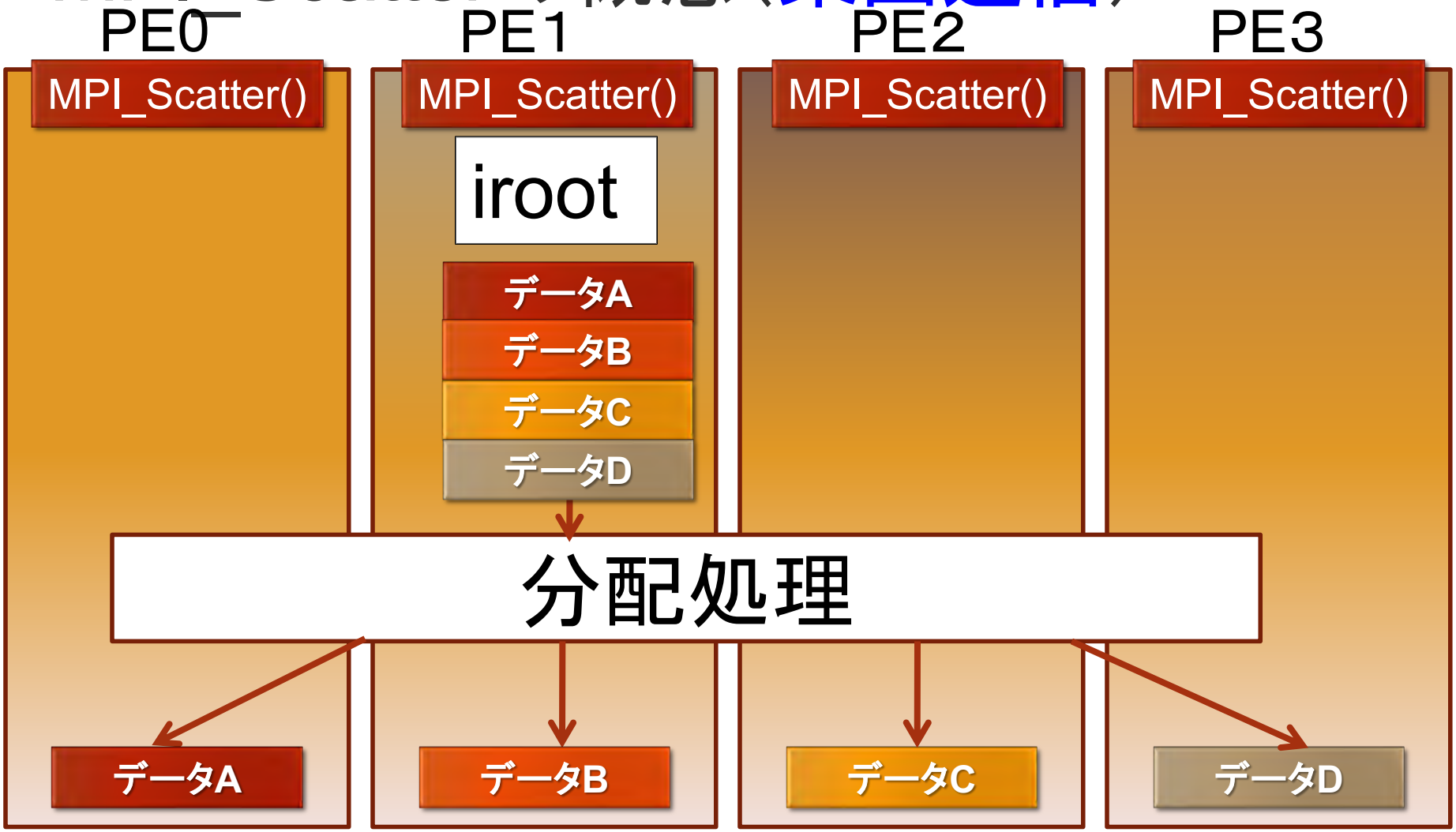
基礎的なMPI関数—MPI_Scatter

- 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

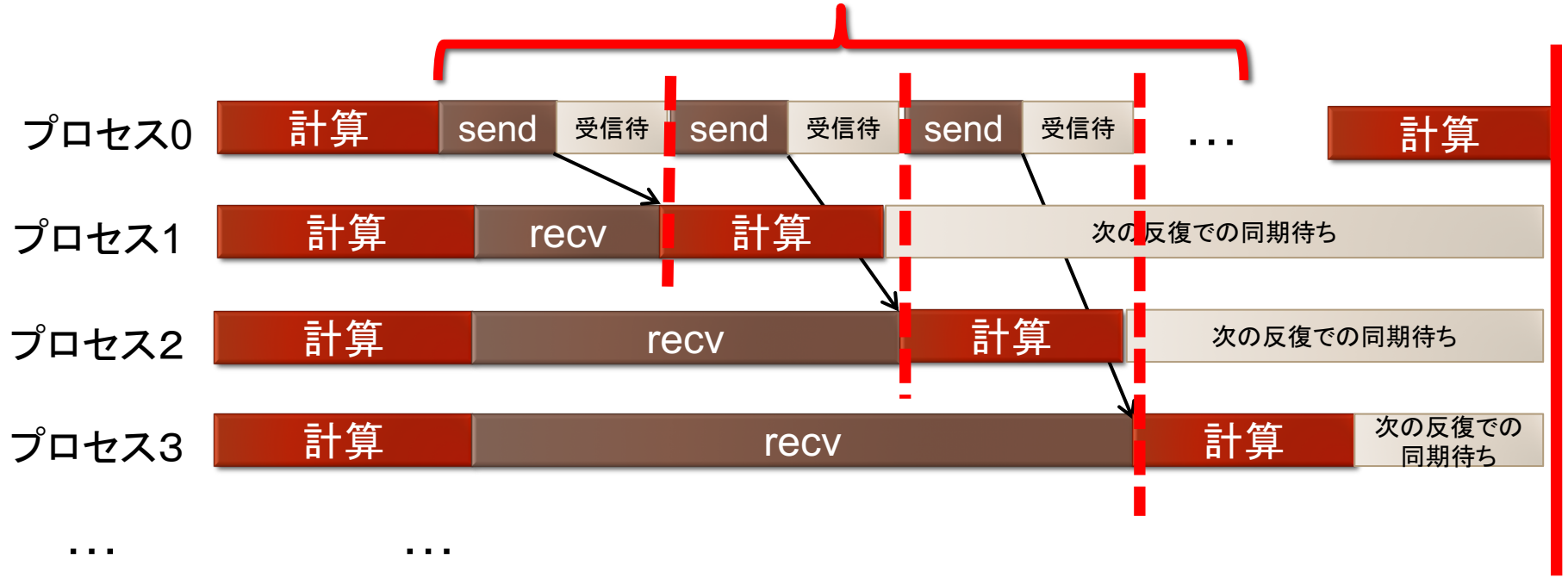
- 送信／受信側のバッファ領域にメッセージが格納され、受信／送信側のバッファ領域が自由にアクセス・上書きできるまで、呼び出しが戻らない
- バッファ領域上のデータの一貫性を保障
- MPI_Send, MPI_Bcastなど

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。

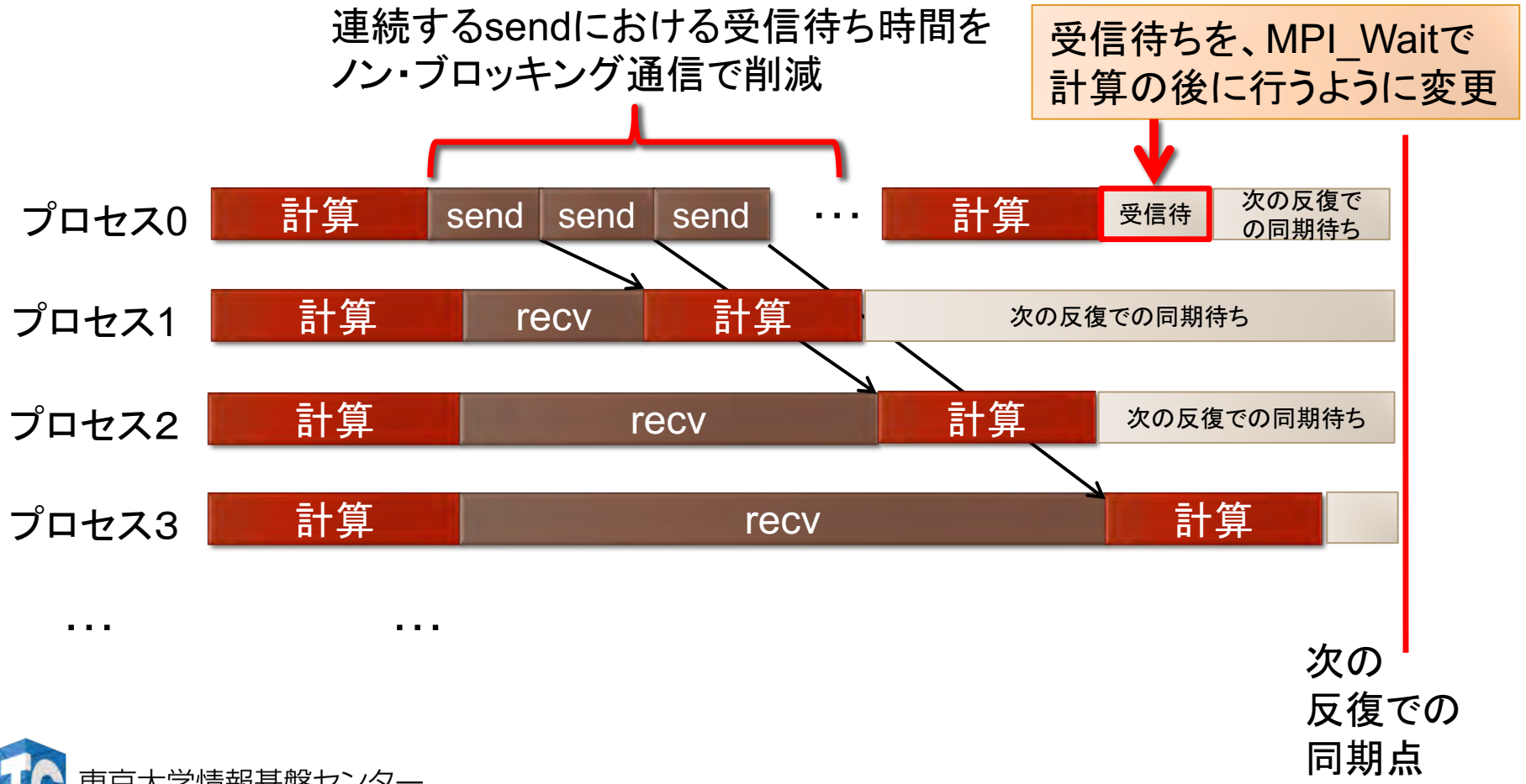
同期待ち関数

- 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）

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

並列化の考え方

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

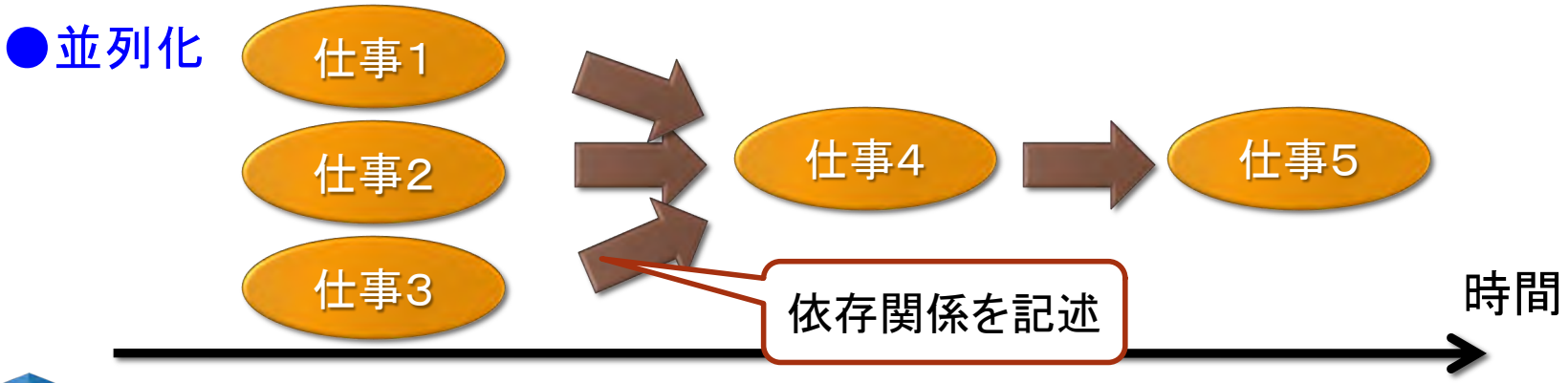
CPU2 $\begin{pmatrix} 7 & 8 & 9 \end{pmatrix}$

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

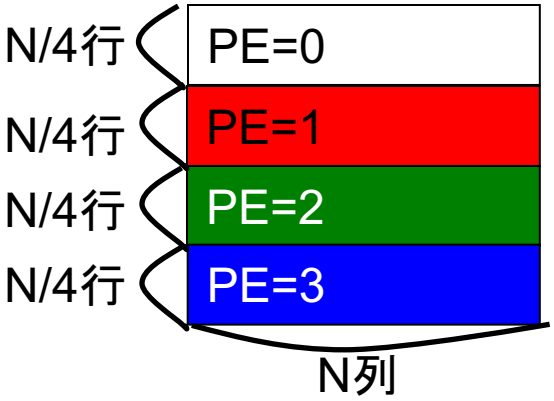
並列に計算：初期データは異なるが演算は同一

その他の並列化手法

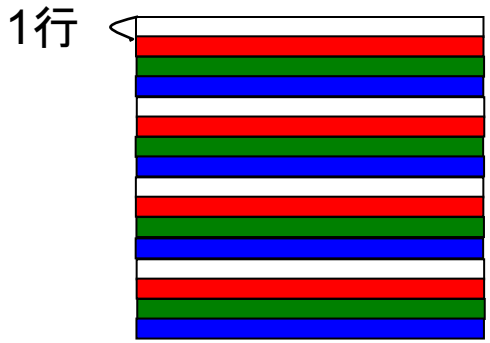
- タスク並列
  - タスク（ジョブ）を分割することで並列化する。
  - データの操作（＝演算）は異なるかもしれない。
  - タスク並列の例：カレーを作る
    - 仕事１：野菜を切る
    - 仕事２：肉を切る
    - 仕事３：水を沸騰させる
    - 仕事４：野菜・肉を入れて煮込む
    - 仕事５：カレールゥを入れる



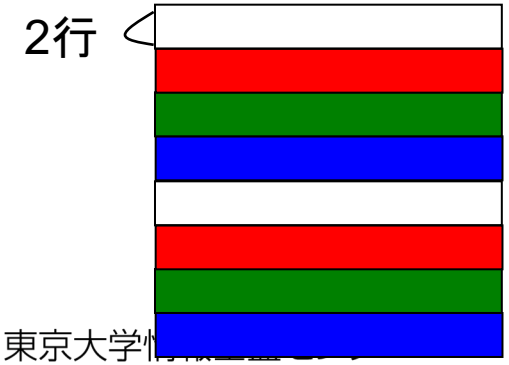
1次元分散



- （行方向）ブロック分割方式
- (Block, *) 分散方式



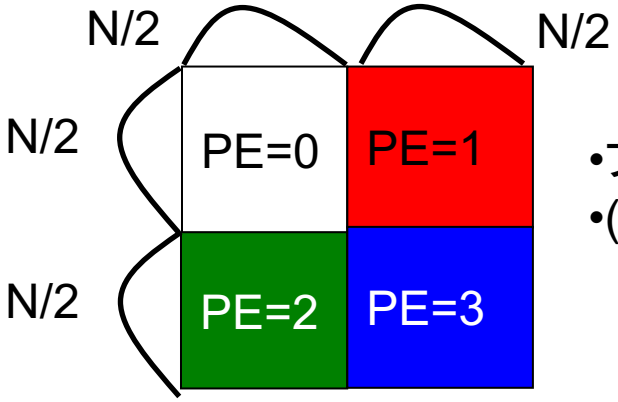
- (行方向) サイクリック分割方式
- (Cyclic, *) 分散方式



- (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

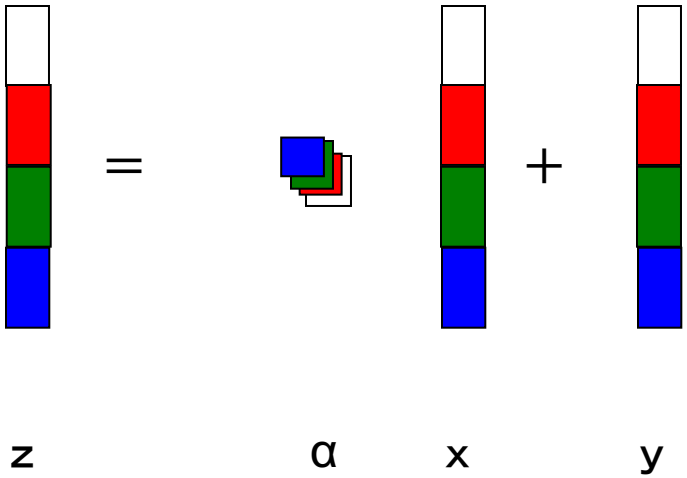
- 二次元ブロック・サイクリック分割方式
- (Cyclic(2), Cyclic(2))分散方式

ベクトルどうしの演算

- 以下の演算

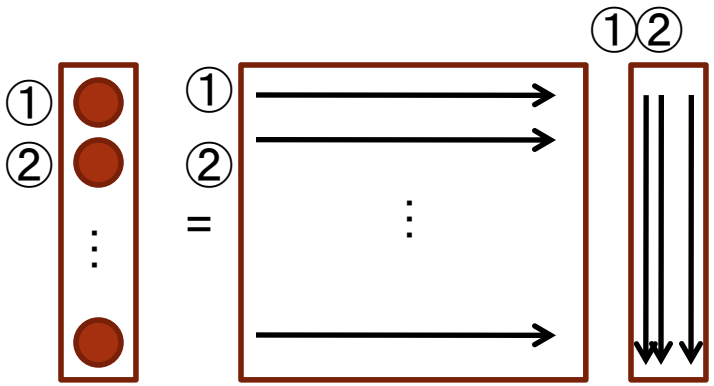
$$z = a\,x + y$$

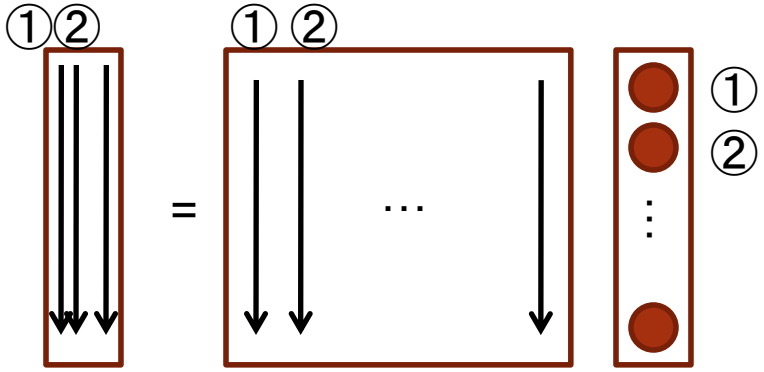
  - ここで、αはスカラ、z、x、y　はベクトル
- どのようなデータ分散方式でも並列処理が可能
  - ただし、スカラ α は全PEで所有する。
  - ベクトルはO(n)のメモリ領域が必要なのに対し、スカラはO(1)のメモリ領域で大丈夫。
    →スカラメモリ領域は無視可能
  - 計算量：O(N/P)
  - あまり面白くない



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

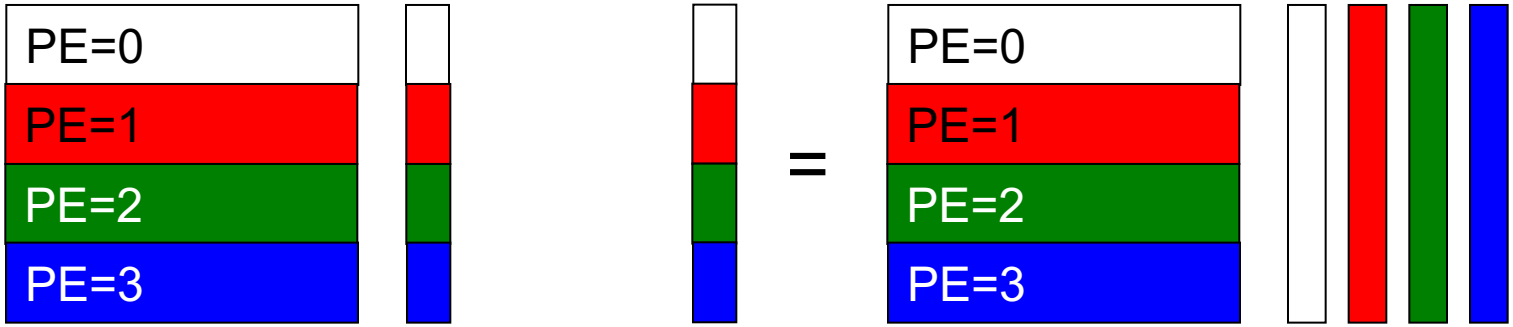
```
for(j=0; j<n; j++) y[j]=0.0;
for(j=0; j<n; j++) {
   for (i=0; i<n; i++) {
      y[i] += a(I,j)*x[j];
   }
}
```

＜列方式＞： Fortran言語向き

行列とベクトルの積

＜行方式の場合＞

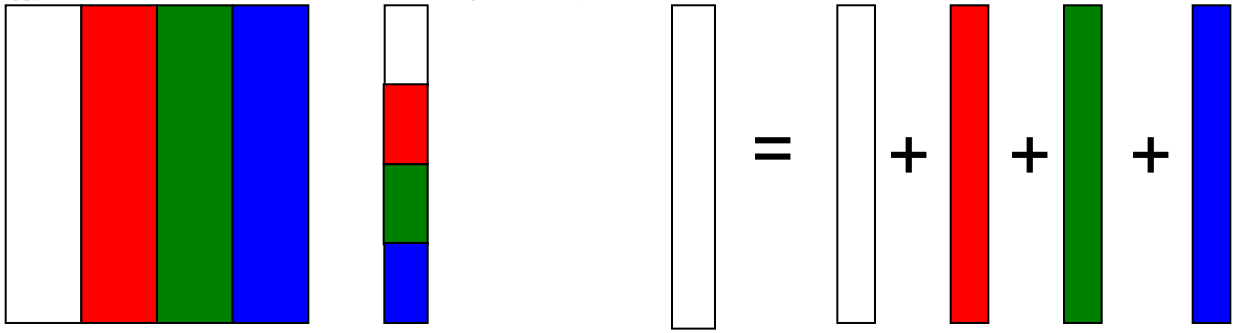
＜行方向分散方式＞ ：行方式に向く分散方式



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



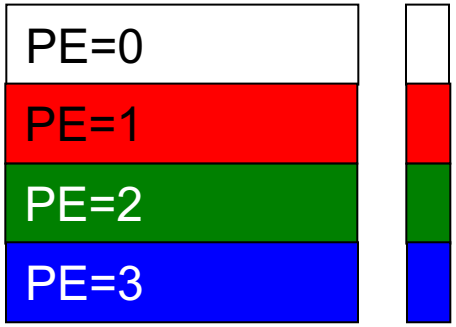
各PE内で行列-ベクトル積を行う

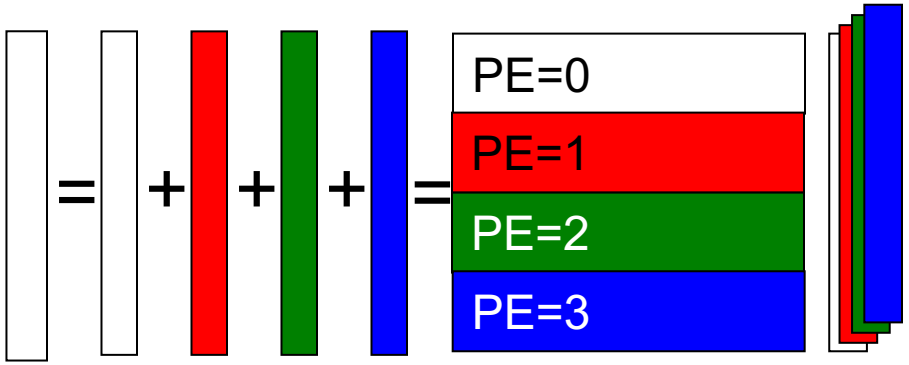
MPI_Reduce関数で総和を求める
（※ある1PEにベクトルすべてが集まる）

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

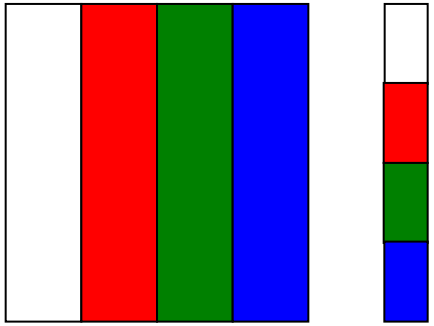


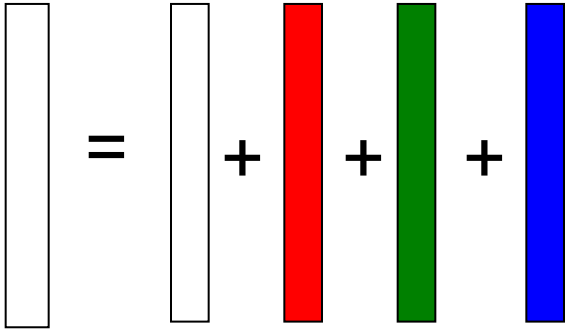


右辺ベクトルを MPI_Allgather関数を利用して、全PEで所有する

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）

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_hy68 を使って make する。
   $ make -f Makefile_hy68 clean
   $ make -f Makefile_hy68
2. 実行ファイル（hello_omp）ができていることを確認する。
   $ ls
4. JOBスクリプト中（hello-hy68.bash）のキュー名を変更する。“lecture-flat” → “tutorial-flat”に変更する。
   $ emacs hello-hy68.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-hy68.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-hy68.bash.eXXXXXX
   hello-hy68.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-hy68.bash.oXXXXXX
5. “Hello parallel world!”が、
   1プロセス*16ノード*68スレッド=1088 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-hy68.bash, C言語、Fortran言語共通）

```
#!/bin/bash
#PJM -L rscgrp=lecture-flat
#PJM -L node=16
#PJM --mpi proc=16
#PJM --omp thread=68
#PJM -L elapse=0:01:00
#PJM -g gt00

mpiexec.hydra -n
${PJM_MPI_PROC} ./hello_omp
```

リソースグループ名
：lecture-flat

利用ノード数、
MPIプロセス数

ノード内利用スレッド数

実行時間制限
：1分

利用グループ名
：gt00

MPIジョブを1*16 = 16 プロセスで実行する。

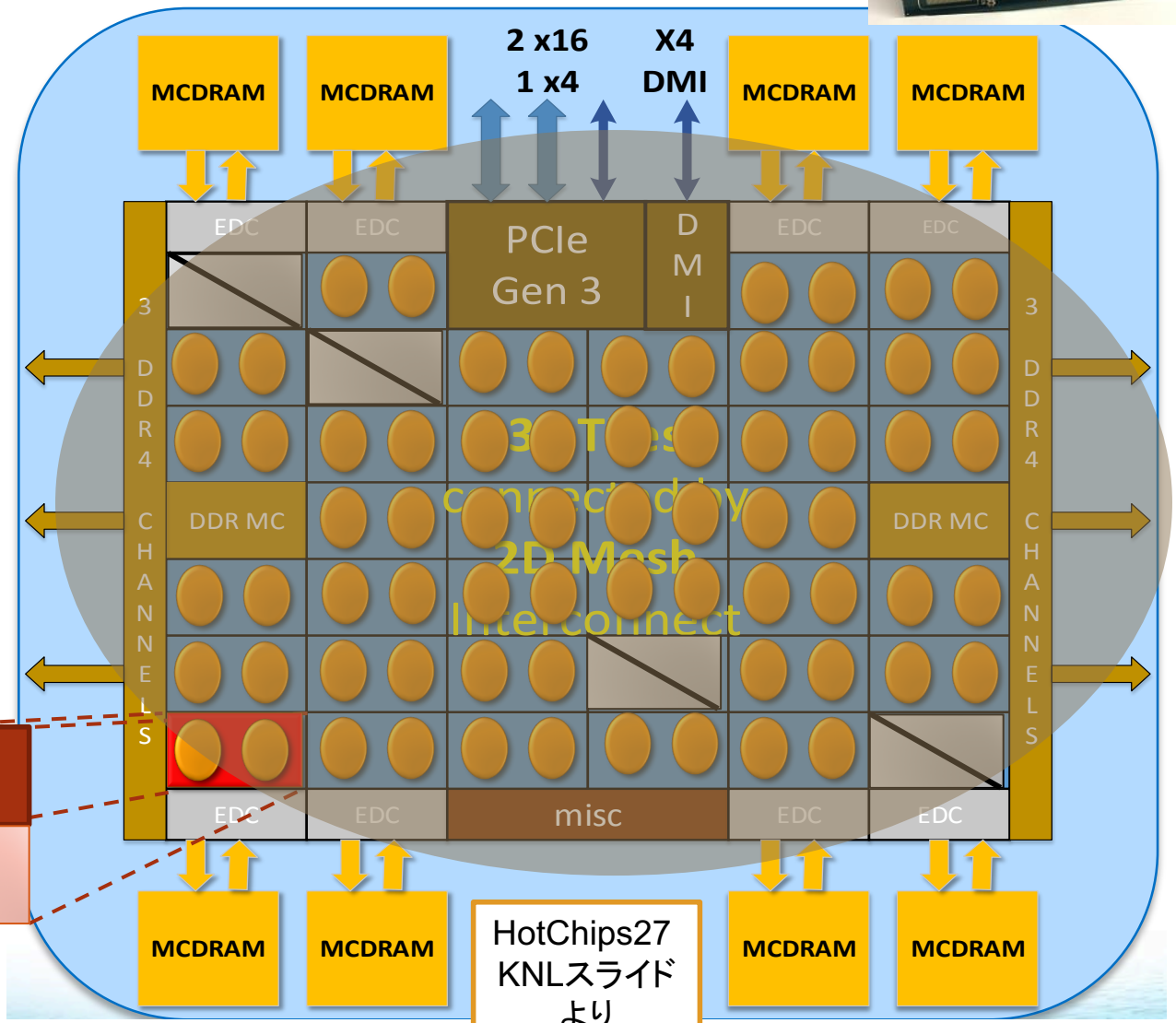
ハイブリッドMPIの実行状況（ノード内）

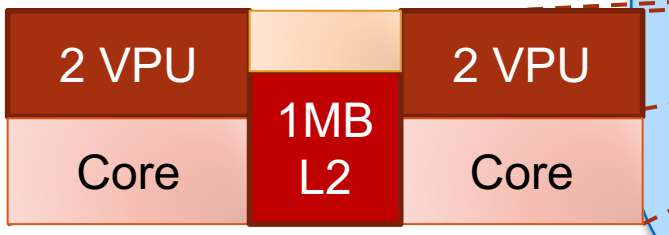
MPIプロセス
スレッド
無効のタイル(例)

- Intel Xeon Phi (Knights Landing)
  - 1ノード1ソケット, 68コア

MCDRAM: オンパッケージの高バンド幅メモリ16GB
+ DDR4メモリ 16GBx6
= 16 + 96 GB

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



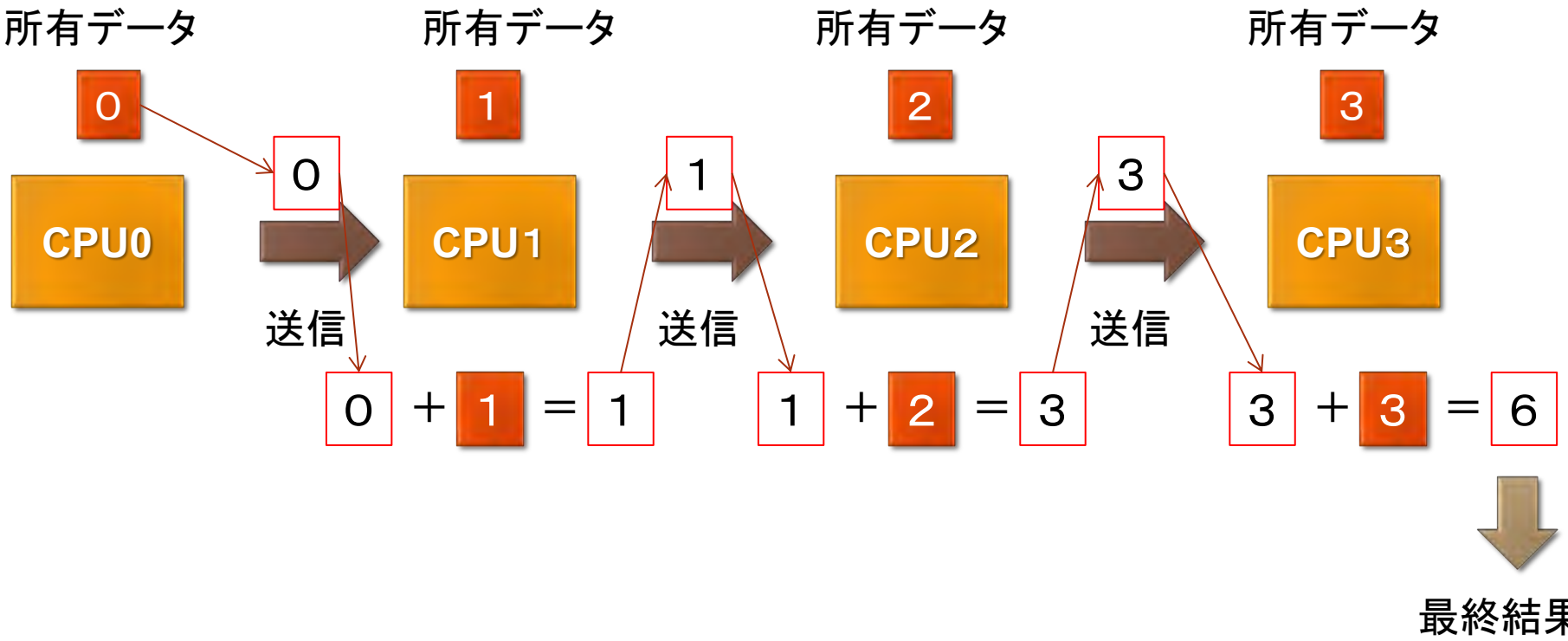


HotChips27 KNLスライドより

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. （1023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_REAL8, &
             myid-1, 0, MPI_COMM_WORLD, istatus, ierr)
       endif
       dsendbuf = dsendbuf + drecvbuf
       if (myid .ne. numprocs-1) then
         call MPI_SEND(dsendbuf, 1, MPI_REAL8, &
             myid+1, 0, MPI_COMM_WORLD, ierr)
       endif
      if (myid .eq. numprocs-1) then
         print *, "Total = ", dsendbuf
       endif
       ....
       stop
end program main
```

受信用システム配列の確保

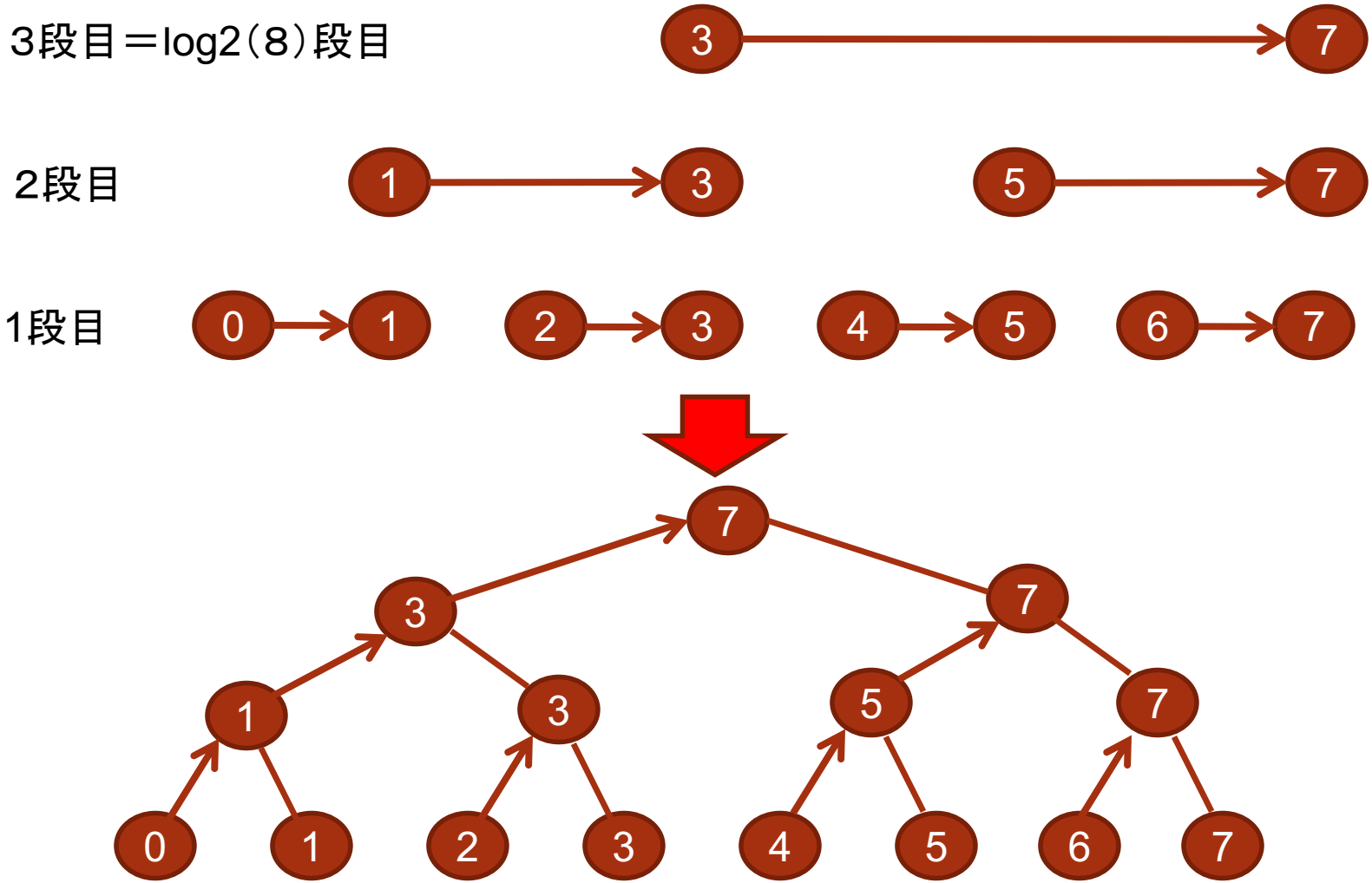
自分より一つ少ないID番号（myid-1）から、double型データ1つを受信しdrecvbuf変数に代入

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）

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）

```
  real(8) :: t0, t1, t2, t_w
..
  call MPI_BARRIER(MPI_COMM_WORLD, ierr)
  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
  call  MPI_REDUCE(t0, t_w, 1, MPI_REAL8, &
      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

- Oakforest-PACS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）mpiexec.hydra ./a.out < in.txt > out.txt

性能プロファイラ

性能プロファイラ

- Oakforest-PACS
  - Intel VTune Amplifier
  - PAPI (Performance API)
  - Oakforest-PACS PAライブラリ

- Webポータルから「**ドキュメント閲覧**」⇨
  Oakforest-PACS システム利用手引書
  7.1. パフォーマンス分析ツール
  または
  Oakforest-PACS PAライブラリ利用ガイド
  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

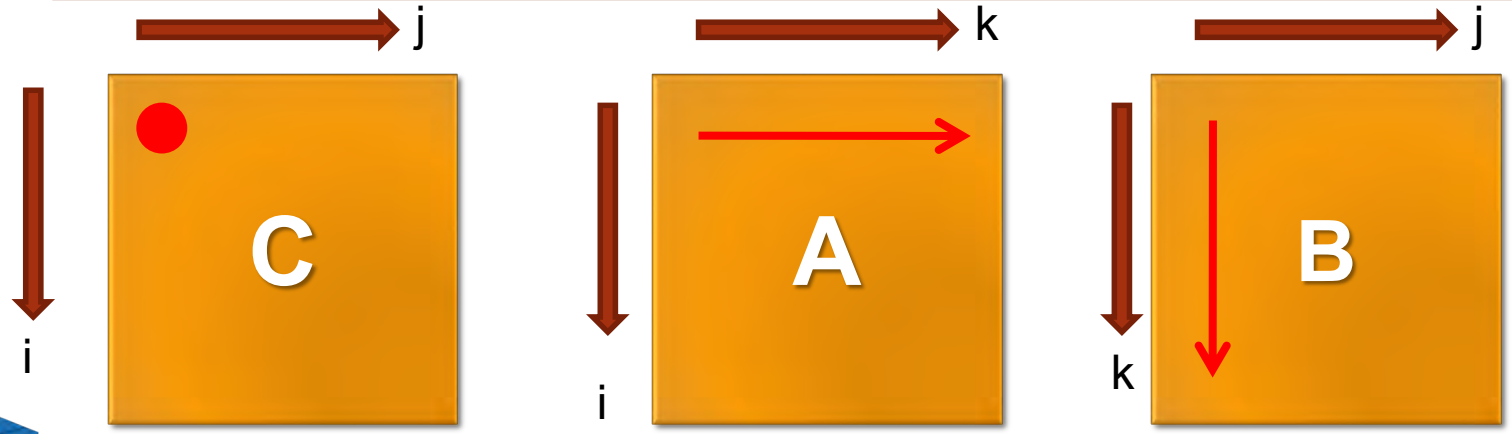
行列の積

- 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
```

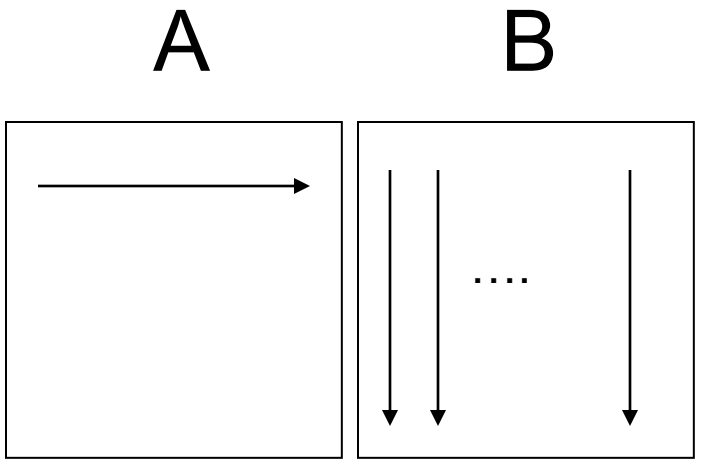
  - 最内部の演算は、外側の3ループを交換しても、計算結果が変わらない
    → 6通りの実現の方法がある

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;
   }
}
```



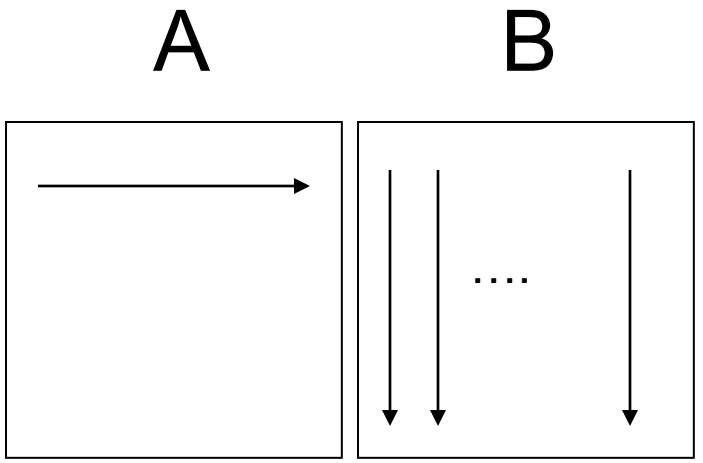
●行方向と列方向のアクセスあり
→行方向・列方向格納言語の
両方で性能低下要因
解決法：
A, Bどちらか一方を転置しておく

※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
```

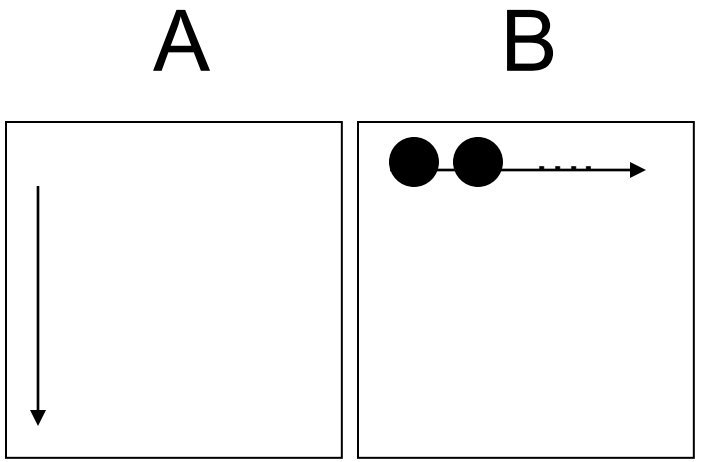


●行方向と列方向のアクセスあり
→行方向・列方向格納言語の
両方で性能低下要因
解決法：
A, Bどちらか一方を転置しておく

※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;
     }
  }
}
```

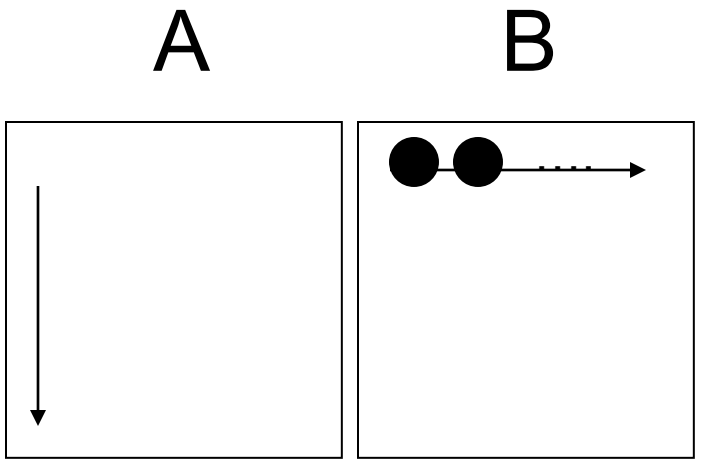


●kjiループでは
列方向アクセスがメイン
→列方向格納言語向き
（Fortran言語）

行列の積（Fortran言語）

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

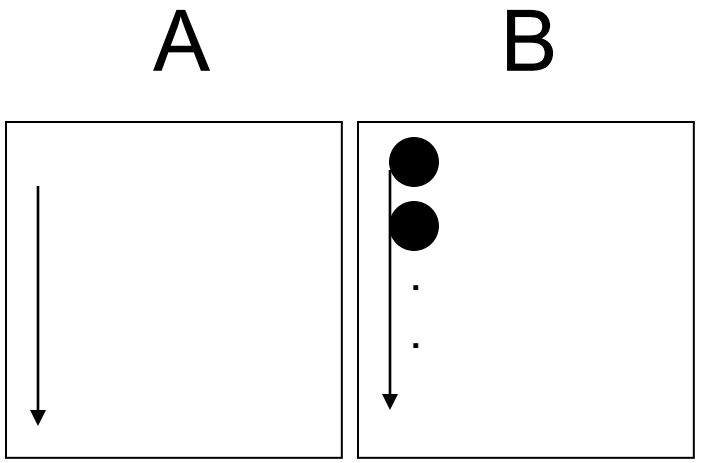
```
do i=1, n
  do j=1, n
    C( i , j ) = 0.0d0
  enddo
enddo
do k=1, n
  do j=1, n
    db = B( k , j )
    do i=1, n
      C( i , j ) = C( i , j )+ A( i , k ) * db
    enddo
  enddo
enddo
```



●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）

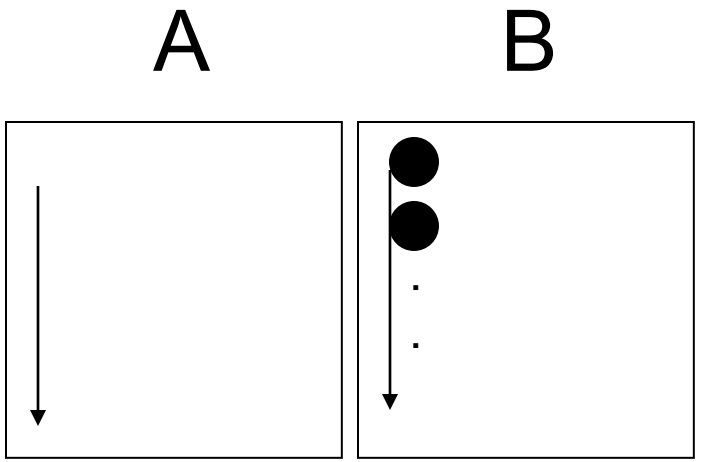
- 中間積形式 (middle-product form)
  - ikj, jkiループによる実現
  -
```
for (j=0;  j<n;  j++)  {
    for (i=0;  i<n;  i++) {
       C[ i ][ j ] = 0.0;
    }
    for (k=0;  k<n;  k++) {
      db = B[ k ][ j ];
      for (i=0;  i<n;  i++) {
       C[ i ][ j ] = C[ i ][ j ] + A[ i ][ k ] * db;
      }
    }
}
```



●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
```

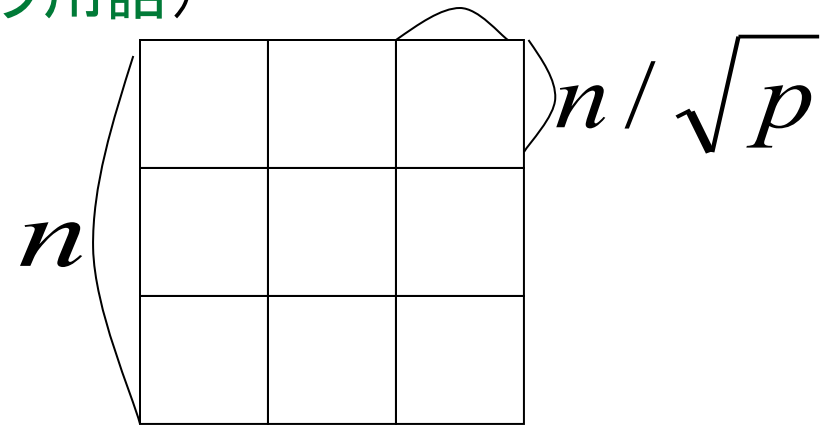


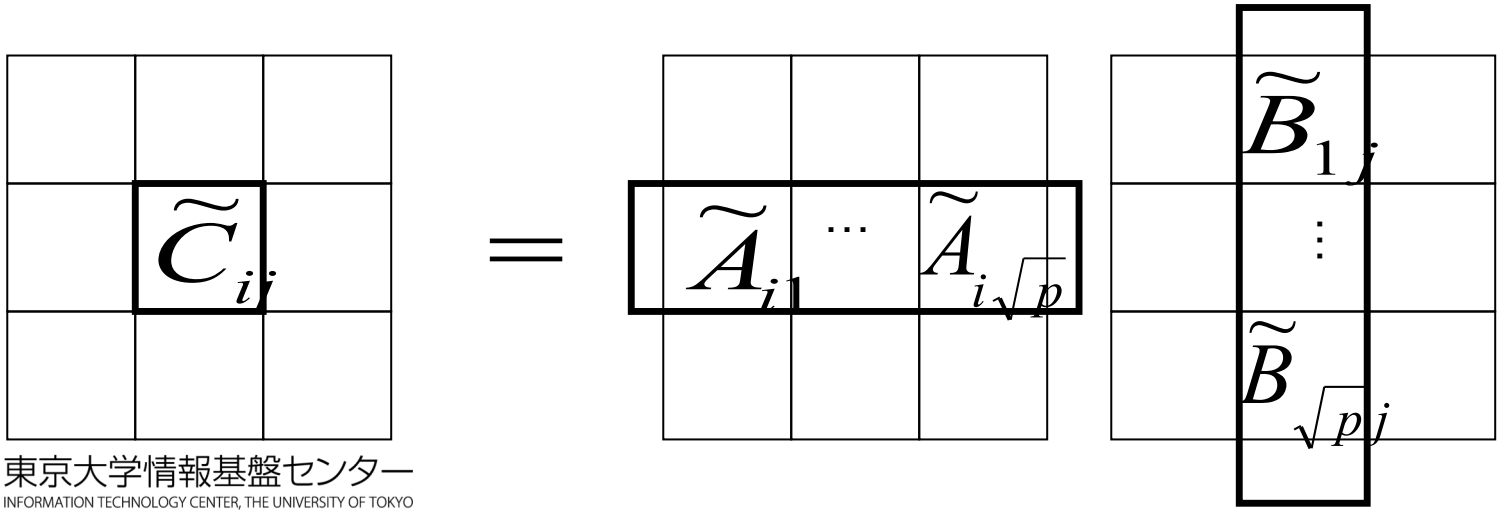
●jkiループでは
全て列方向アクセス
→列方向格納言語に
最も向いている
（Fortran言語）

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・システリック方式
     - 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・システリック方式
     - 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-ofp.tar.gz
- ジョブスクリプトファイルmat-mat.bash　中のキュー名を
  lecture-flat から tutorial-flat に変更してから
  pjsub　してください。
  - lecture-flat : 実習時間外のキュー
  - tutorial-flat :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 /work/gt00/z30105/Mat-Mat-ofp.tar.gz ./
  $ tar xvfz Mat-Mat-ofp.tar.gz
  $ 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  = 1088
Mat-Mat time  = 2.822111 [sec.]
 912.730515 [MFLOPS]
 OK!
N  = 1088
Mat-Mat time  = 0.567127 [sec.]
 4541.887430 [MFLOPS]
 OK!
N  = 1088
Mat-Mat time  = 0.302566 [sec.]
 8513.271503 [MFLOPS]
 OK!
```

コアの割り当てが偏っている（デフォルト）

コアの最適割り当て

export I_MPI_PIN_DOMAIN=4

MCDRAMの利用

export I_MPI_HBW_POLICY=hbw_bind

1コアのみで、8.5GFLOPS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  =        1088
 Mat-Mat time[sec.] =    2.93095993995667
 MFLOPS =    878.833894104587
  OK!
 NN  =        1088
 Mat-Mat time[sec.] =   0.556317090988159
 MFLOPS =    4630.14165702036
  OK!
 NN  =        1088
 Mat-Mat time[sec.] =   0.307068109512329
 MFLOPS =    8388.45473594594
  OK!
```

コアの割り当てが偏っている(デフォルト)

コアの最適割り当て

```
export I_MPI_PIN_DOMAIN=4
```

MCDRAMの利用

```
export I_MPI_HBW_POLICY
=hbw_bind
```

1コアのみで、8.5GFLOPSの性能

サンプルプログラムの説明

- #define N 1088
  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

- 行列サイズ変数が、NNとなっています。
  integer, parameter :: NN=1088

演習課題（1）

- MyMatMat関数を並列化してください。
  - #define N 1088
  - #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）

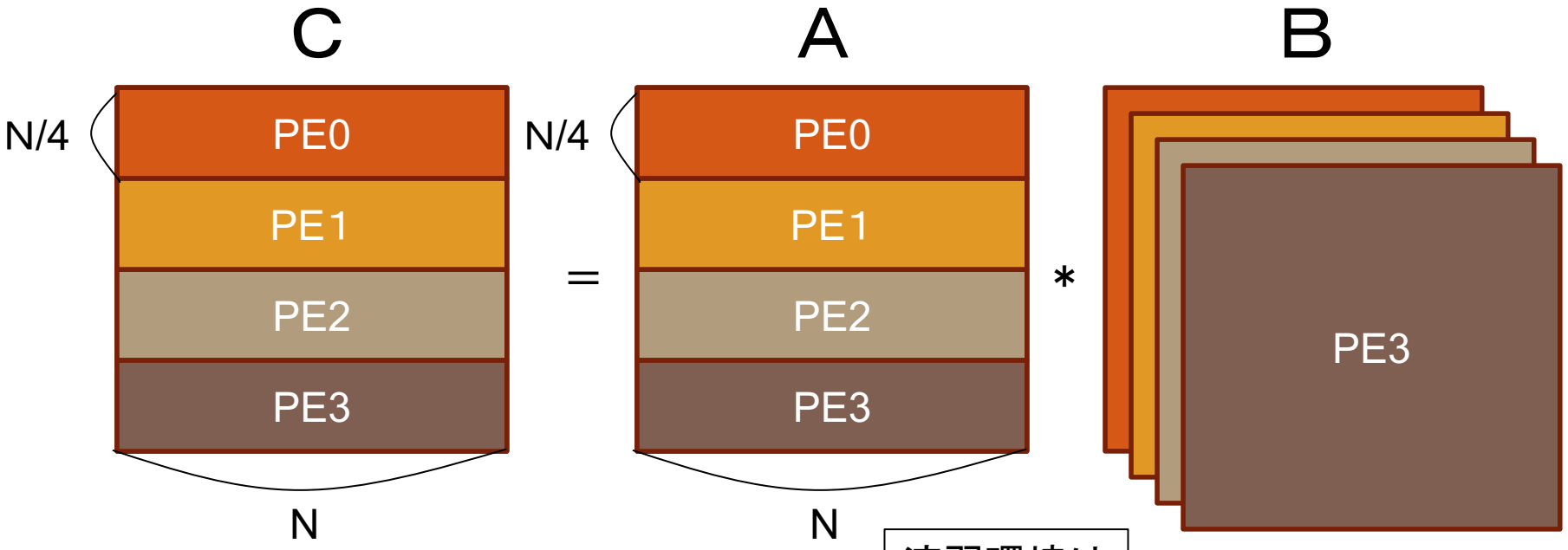
- サンプルプログラムでは、行列A、Bの要素を全部1として、行列-行列積の結果をもつ行列Cの全要素がNであるか調べ、結果検証しています。デバックに活用してください。
  - 行列Cの分散方式により、

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。

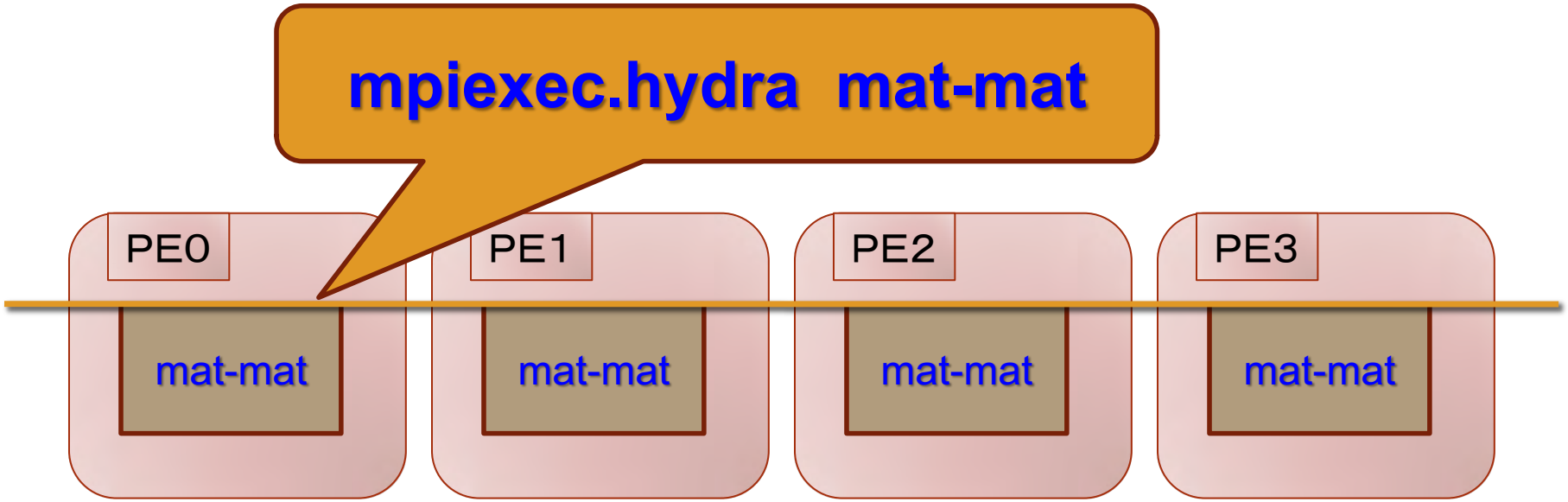
MPI並列化の大前提（再確認）

- 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

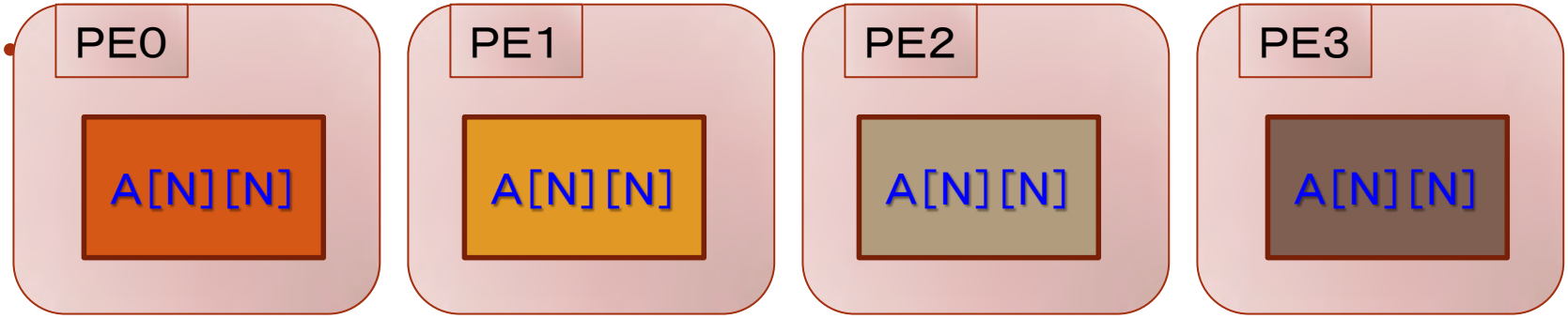
    から処理が始まる。
- 分散メモリ型並列計算機
  - 各PEは、完全に独立したメモリを持っている。（共有メモリではない）

MPI並列化の大前提（再確認）

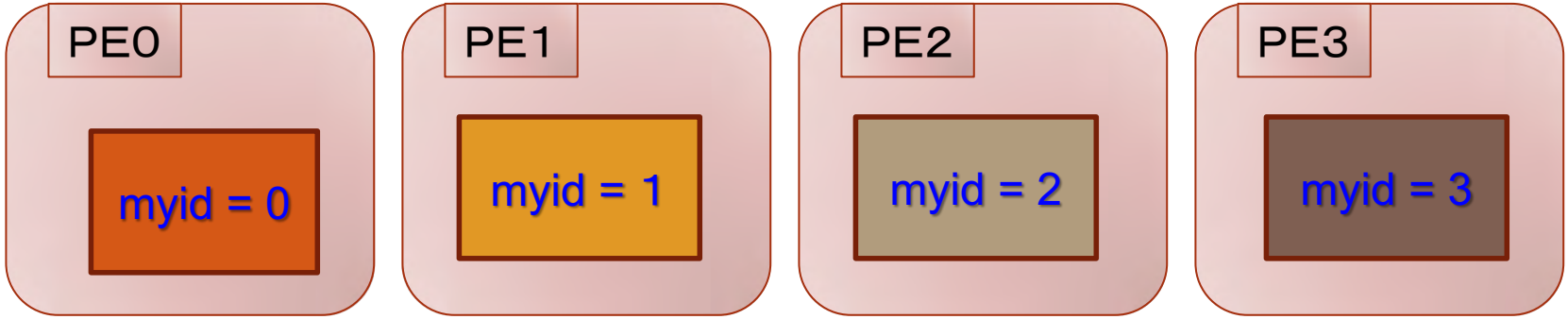
- 各PEでは、＜同じプログラムが同時に起動＞されて開始されます。



MPI並列化の大前提（再確認）

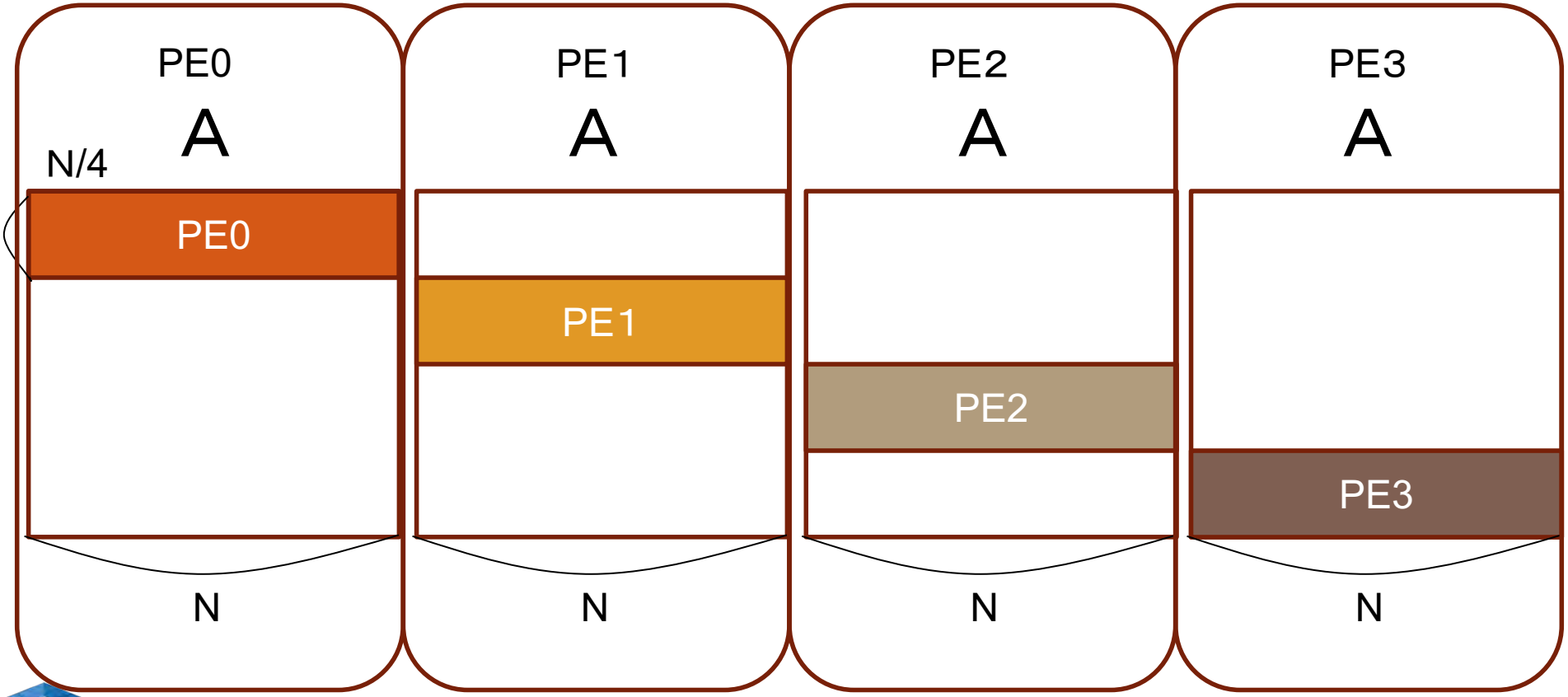
- 各PEでは、＜別配列が個別に確保＞されます。
- 

- 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

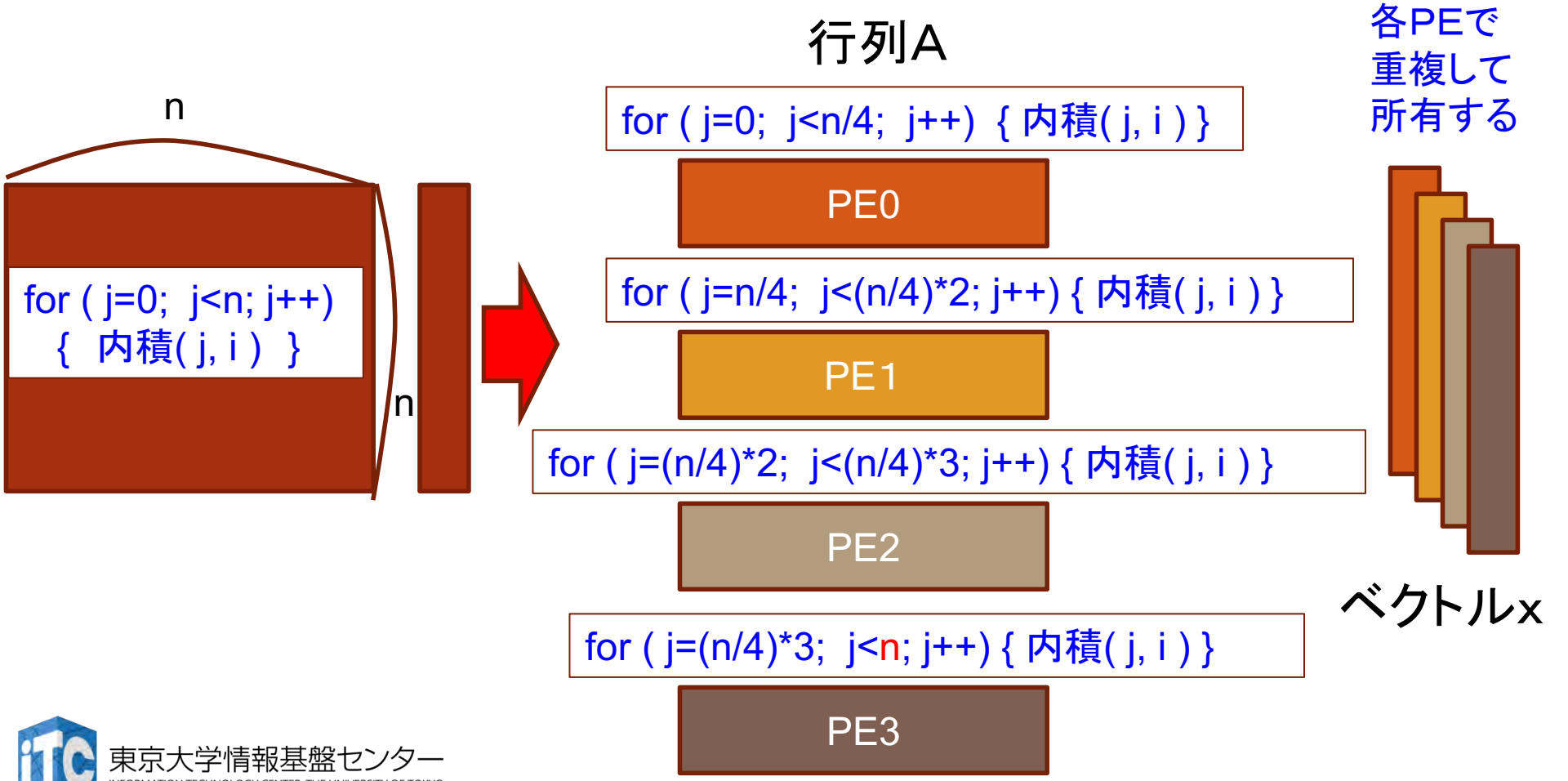


本実習プログラムのTIPS

- 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関数およびmain内で、引数設定や宣言なしに、参照できます。
  - （Fortranではmodule MyCalcの中にあります）
- 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。

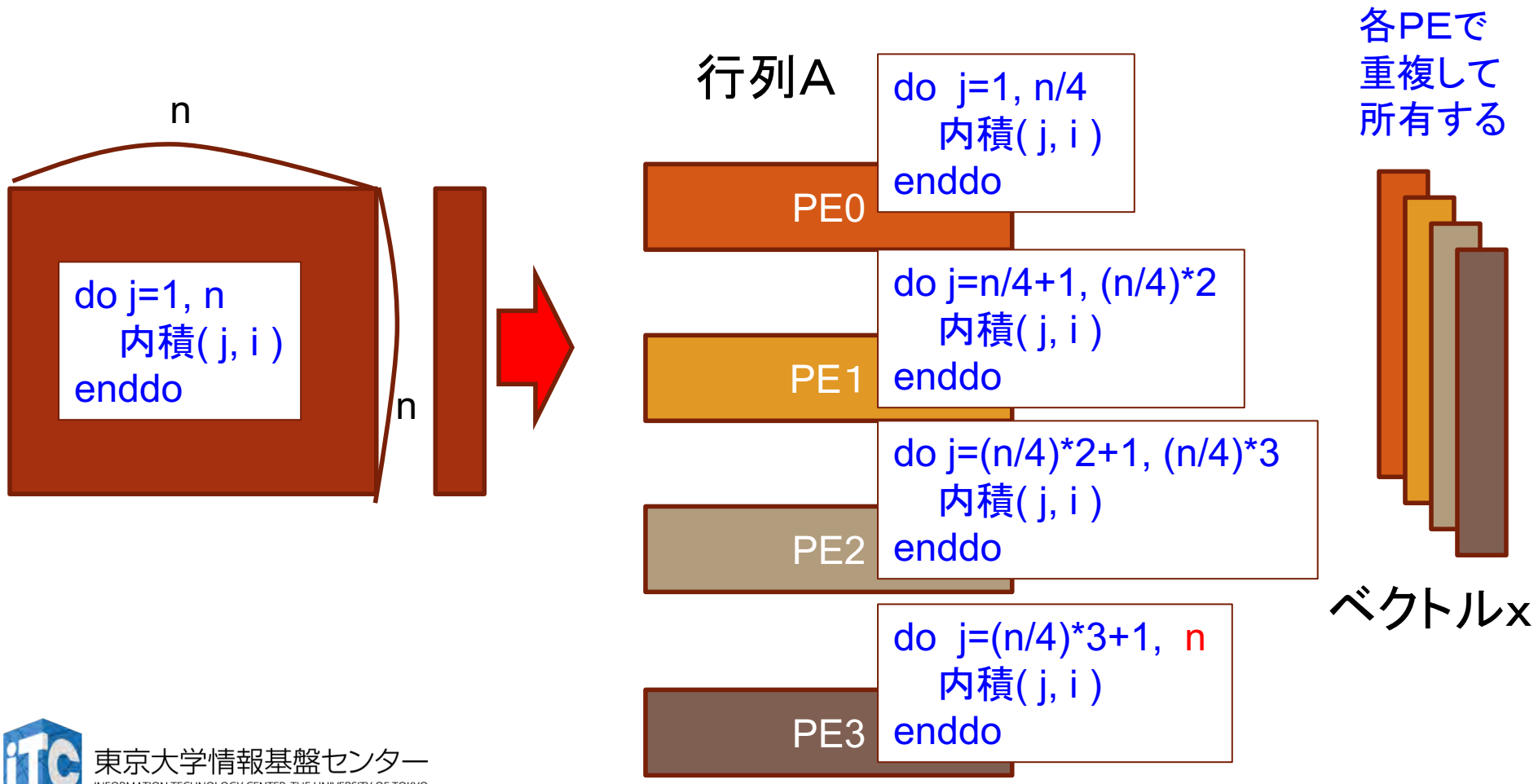
並列化の考え方（行列-ベクトル積の場合、C言語）

- 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
```

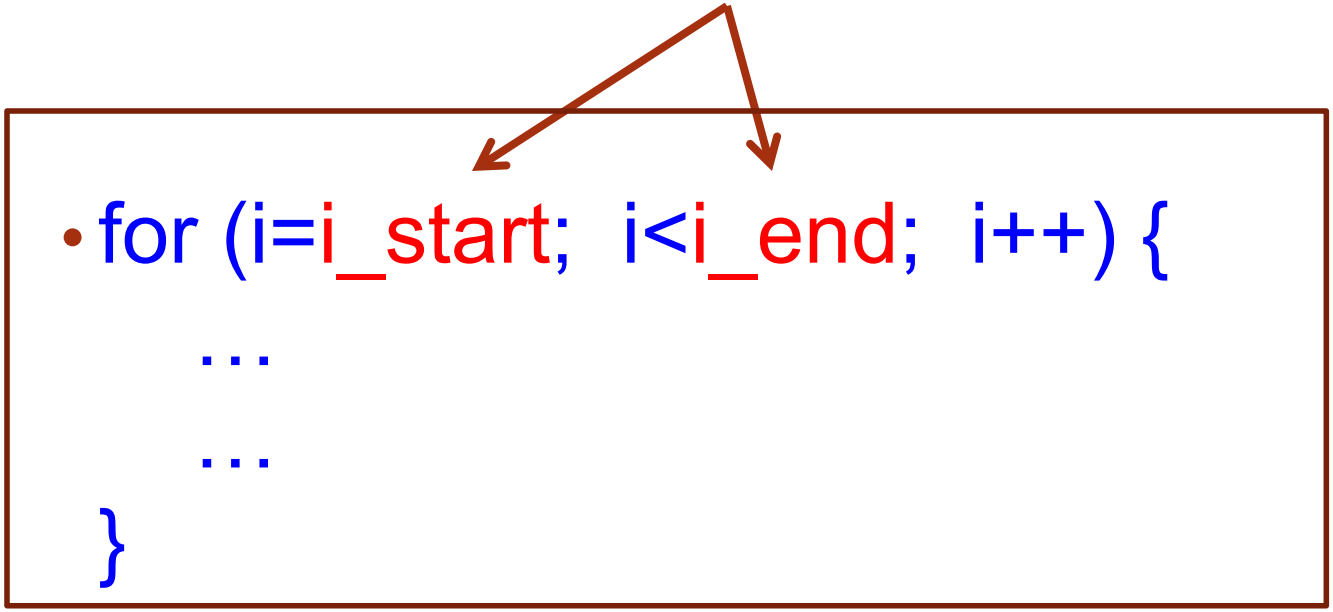
3. （2の並列化が完全に終了したら）各PEで担当のデータ部分しか行列を確保しないように変更する。
   - 上記のループは、以下のようになる。

```
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-ofp.tar.gz
- ジョブスクリプトファイルmat-mat-d.bash 中のキュー名を
  lecture-flat から tutorial-flat に変更してから qsub してください。
  - lecture-flat: 実習時間外のキュー
  - tutorial-flat: 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 /work/gt00/z30105/Mat-Mat-d-ofp.tar.gz ./
  $ tar xvfz Mat-Mat-d-ofp.tar.gz
  $ 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

```
Error! in ( 0 , 2 )-th argument in PE 0
Error! in ( 0 , 2 )-th argument in PE 61
 …
N  = 2176
Mat-Mat time  = 0.000896 [sec.]
 22999044.714267 [MFLOPS]
...
N  = 2176
Mat-Mat time  = 0.000285 [sec.]
 72326702.959176 [MFLOPS]
...
N  = 2176
Mat-Mat time  = 0.000237 [sec.]
 86952122.772853 [MFLOPS]
```

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

```
Error! in ( 1 , 3 )-th argument in PE 0
Error! in ( 1 , 3 )-th argument in PE 1033
...
 NN  =        2176
 Mat-Mat time  =   1.497983932495117E-003
 MFLOPS =    13756232.6624224
...
 NN  =        2176
 Mat-Mat time  =   1.003026962280273E-003
 MFLOPS =    20544428.2904683
...
 NN  =        2176
 Mat-Mat time  =   1.110076904296875E-003
 MFLOPS =    18563232.3492268
...
```

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 2176
  - 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

- 行列サイズ変数が、NNとなっています。
  integer, parameter :: NN=2176

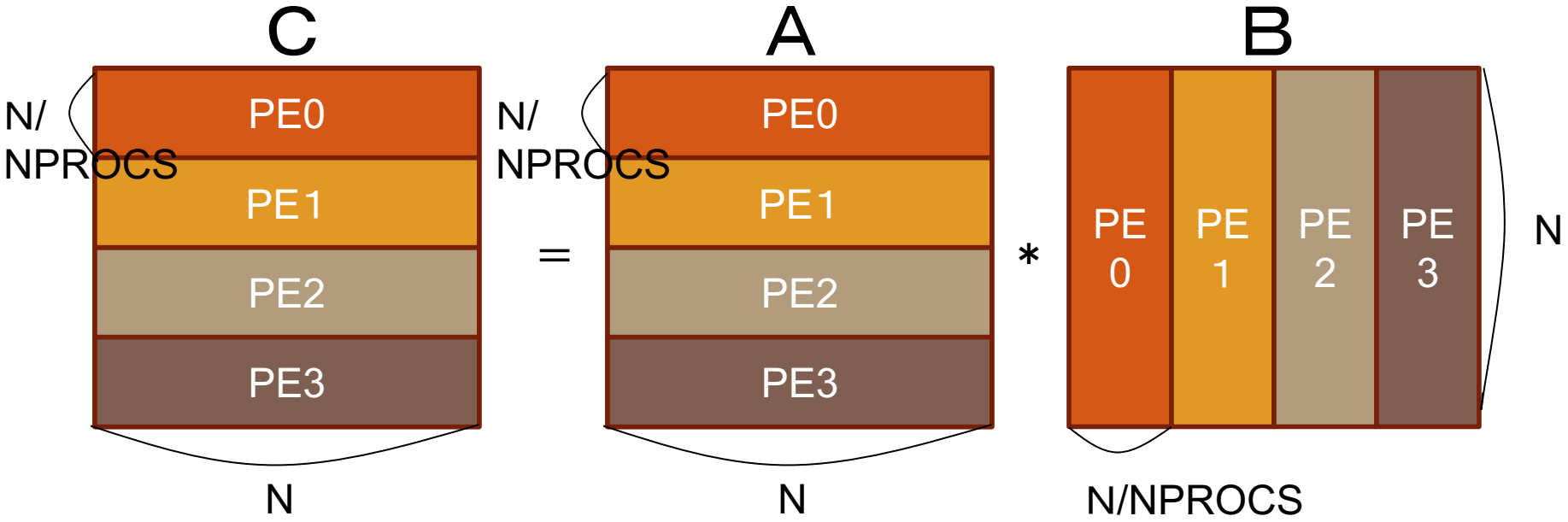
演習課題（1）

- MyMatMat関数（手続き）を並列化してください。
  - デバック時は
    #define N 2176
    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。

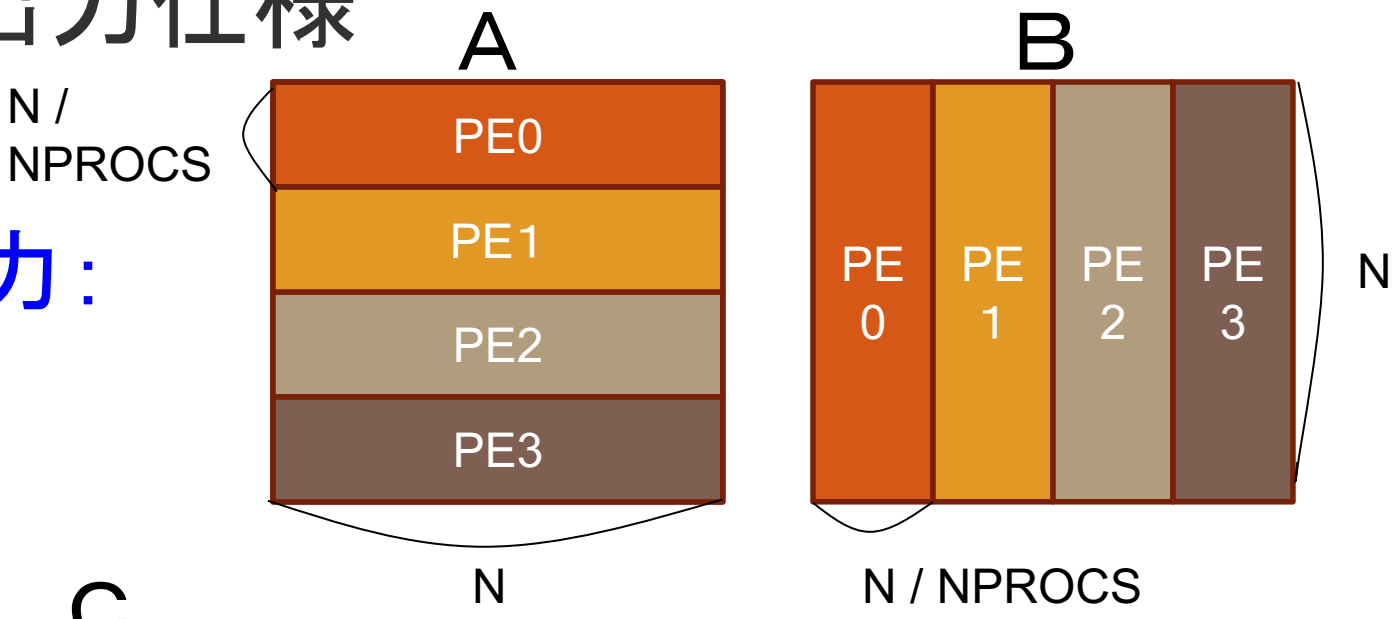
行列A、B、Cの初期配置

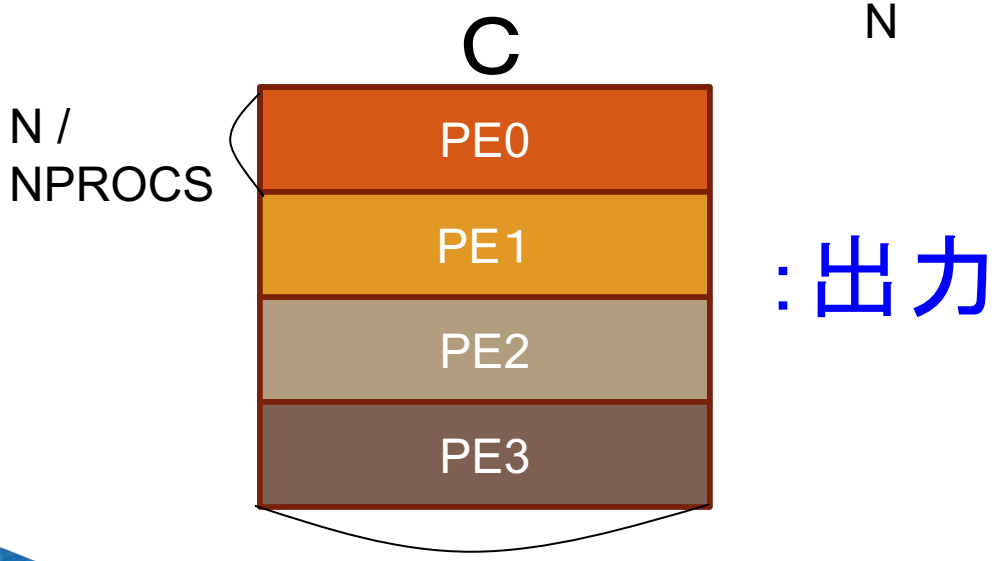
- 行列A、B、Cの配置は以下のようになっています。
  （ただし以下は4PEの場合で、実習環境は異なります。）



- 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

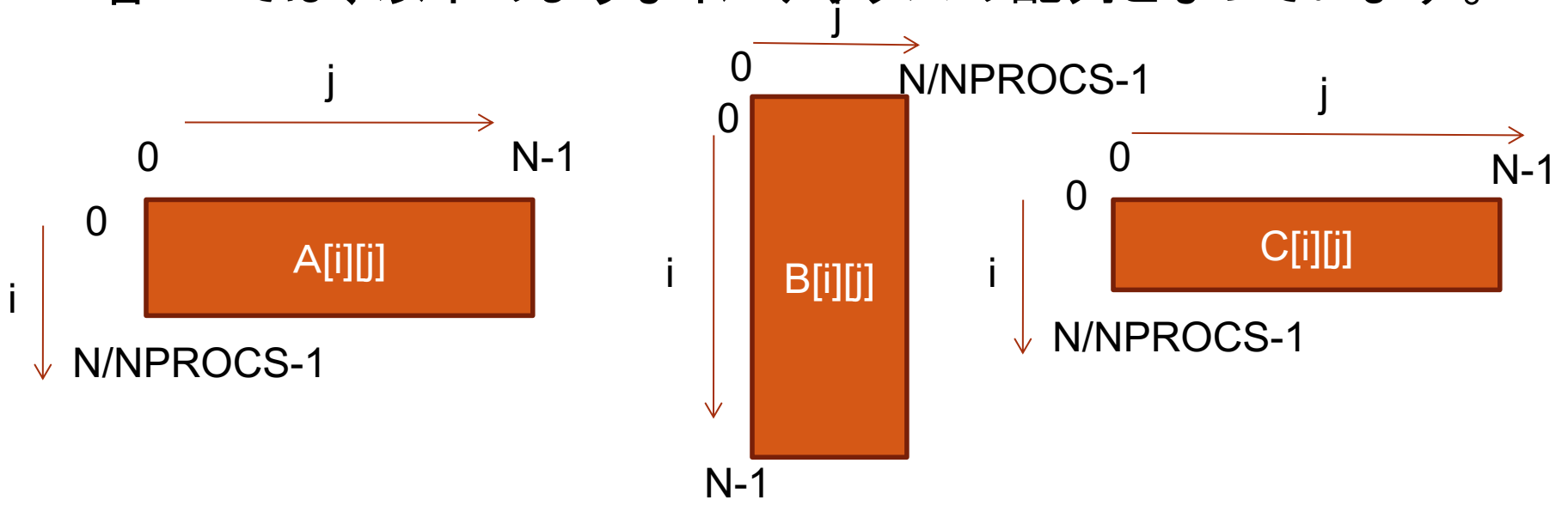




●この例は4PEの場合ですが、実習環境は異なります。

並列化の注意（C言語）

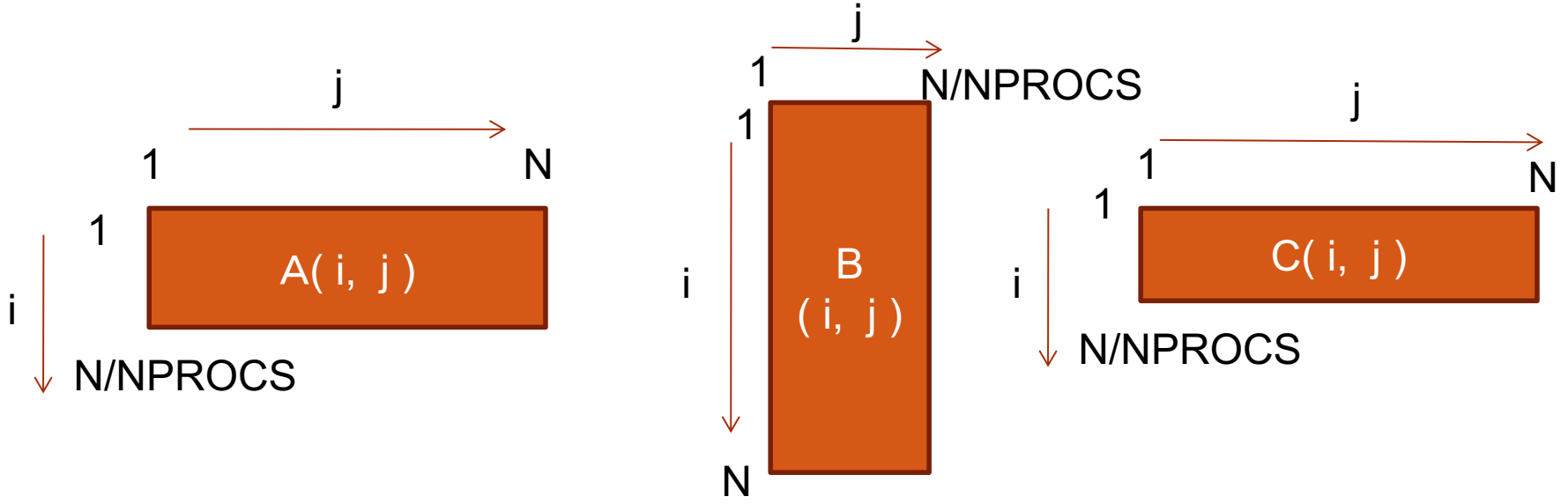
- 各配列は、完全に分散されています。
- 各PEでは、以下のようなインデックスの配列となっています。



- 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）

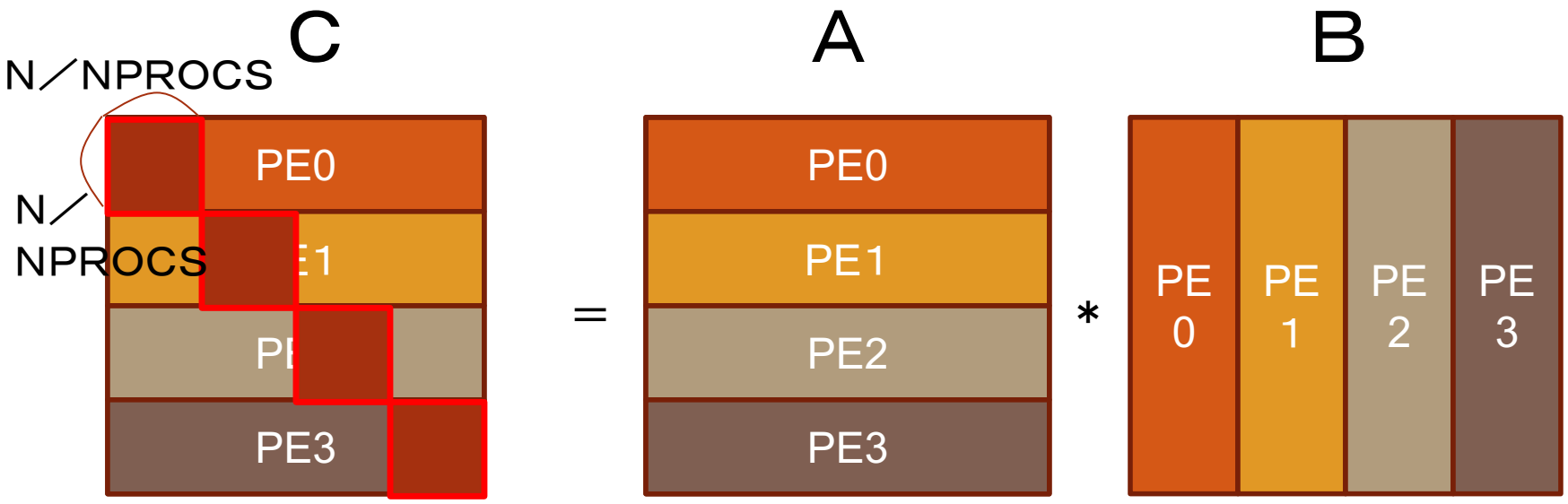
- 各配列は、完全に分散されています。
- 各PEでは、以下のようなインデックスの配列となっています。



- 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

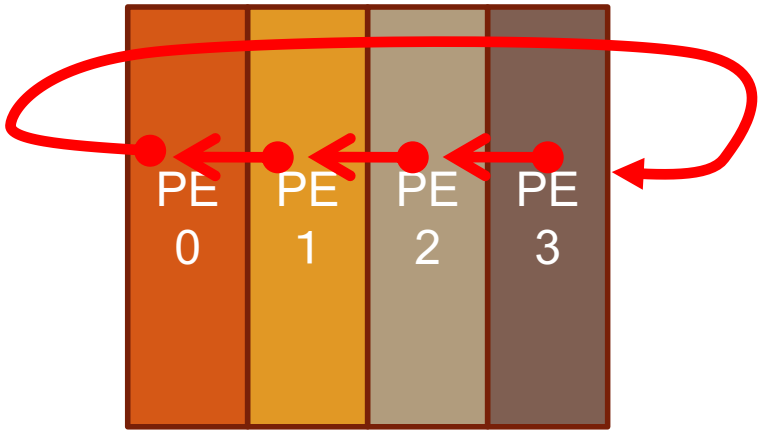
- 行列積を計算するには、各PEで完全な行列Bのデータがないので、行列Bのデータについて通信が必要です。
- たとえば、以下のように計算する方法があります。
- ステップ1



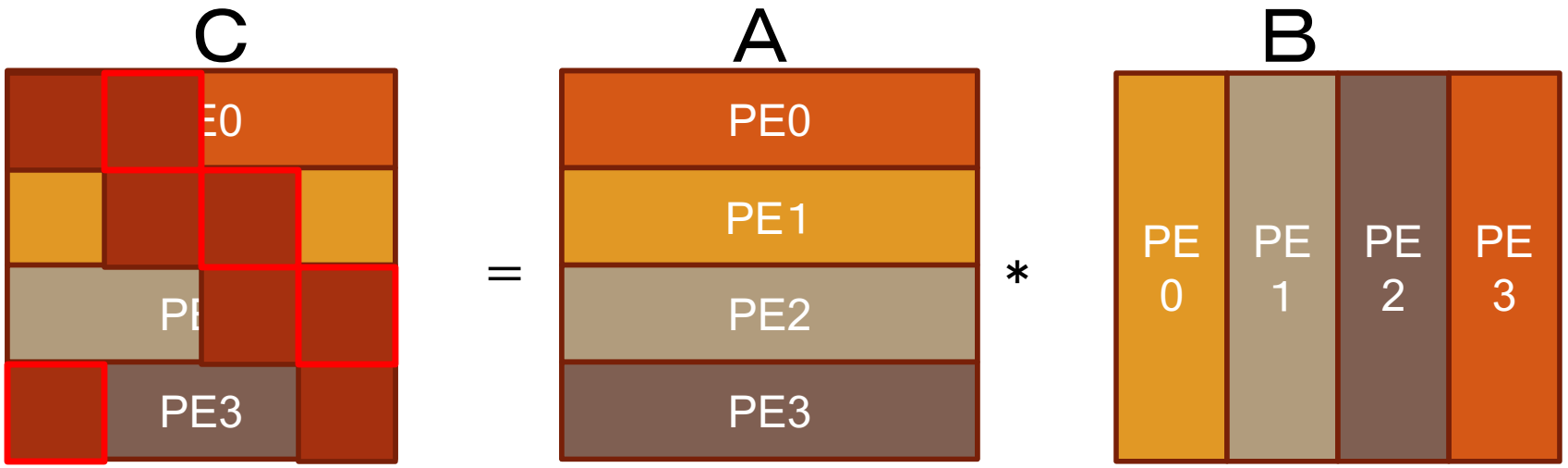
ローカルなデータを使って得られた行列-行列積結果

並列化のヒントB

- ステップ2



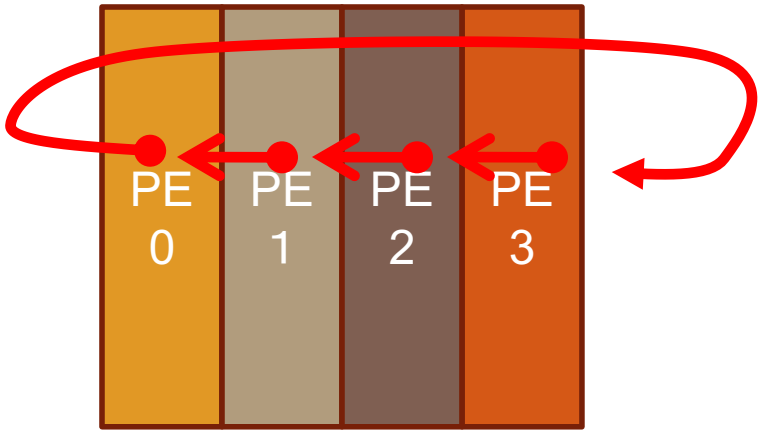
自分の持っているデータを
ひとつ左隣りに転送する
（PE0は、PE3に送る）
【循環左シフト転送】



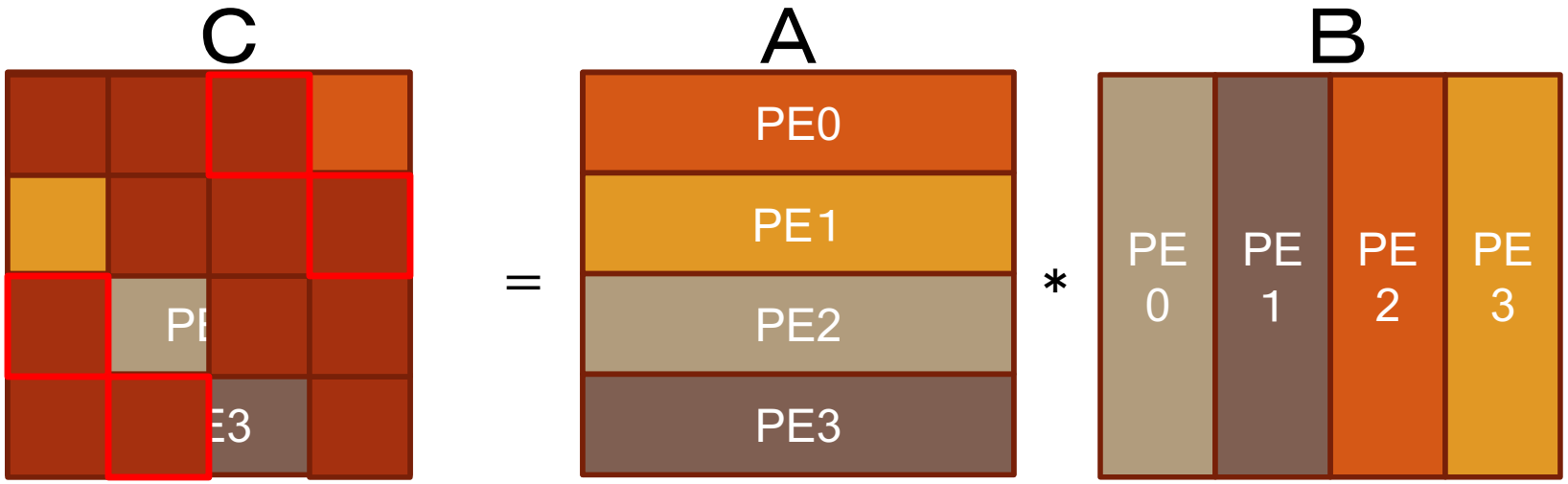
ローカルなデータを使って得られた
行列-行列積結果

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

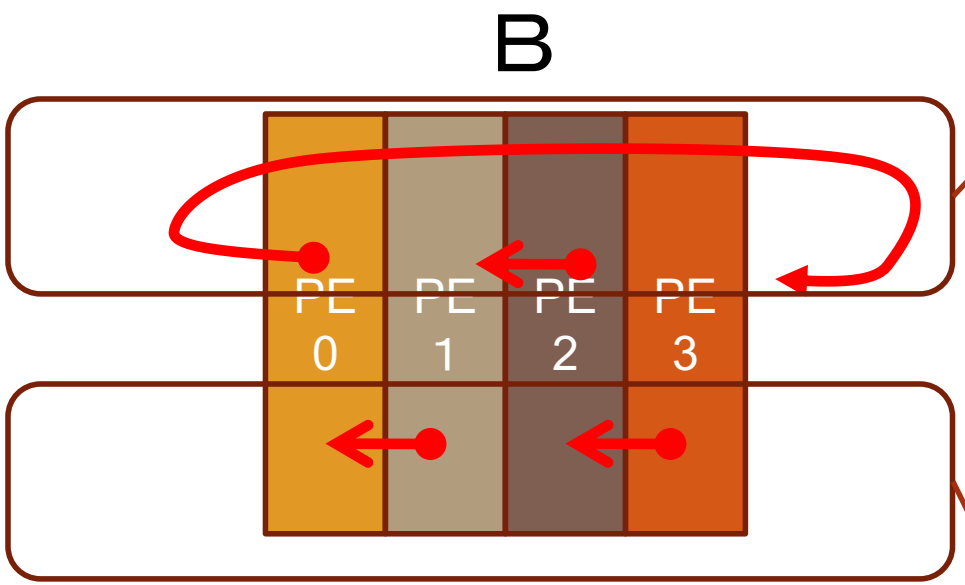
並列化の注意

- 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_REAL8,  isendPE, &
               iloop, MPI_COMM_WORLD,  ierr)
         call MPI_RECV(B_T,  ib*n,  MPI_REAL8,  irecvPE, &
              iloop+numprocs, MPI_COMM_WORLD,  istatus,  ierr)
      else
         call MPI_RECV(B_T,  ib*n,  MPI_REAL8,  irecvPE, &
                        iloop,  MPI_COMM_WORLD,  istatus,  ierr)
         call MPI_SEND(B,  ib*n,  MPI_REAL8,  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

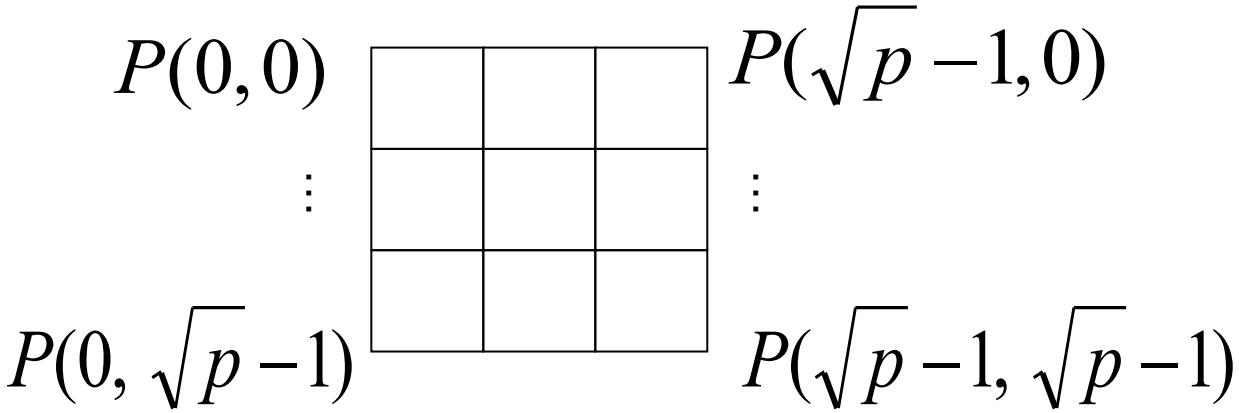
お疲れさまでした

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



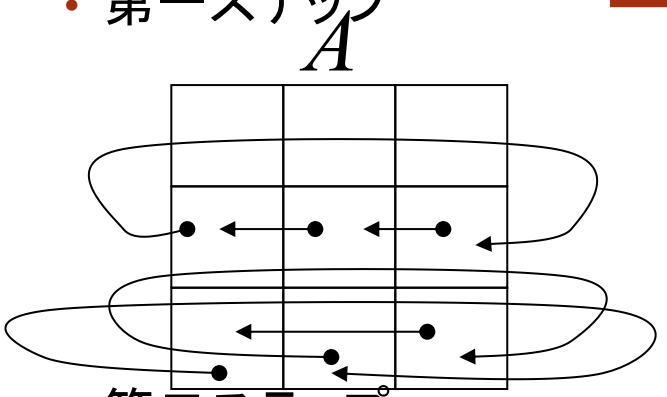
  - PE数が、2のべき乗数しかとれない
  - 各PEは、行列A、B、Cの対応する各小行列を、1つづつ所有
  - 行列A、Bの小行列と同じ大きさの作業領域を所有

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



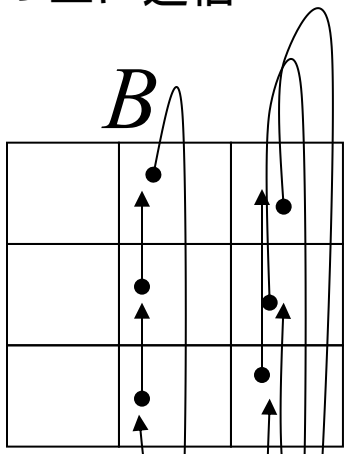
ローカルな
行列-行列積の後

：1つ右に通信

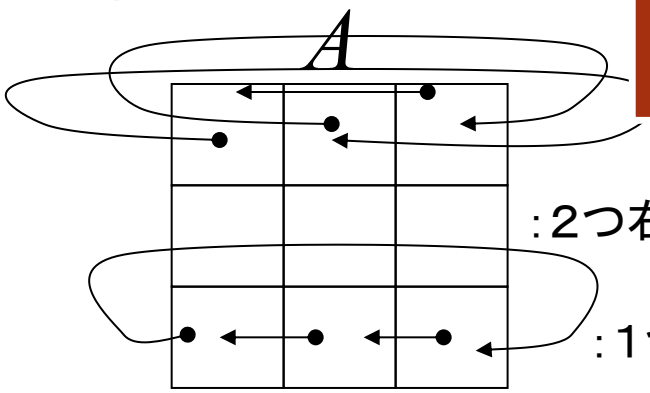
：2つ右に通信

1つ上に通信

2つ上に通信



  - 第二ステップ

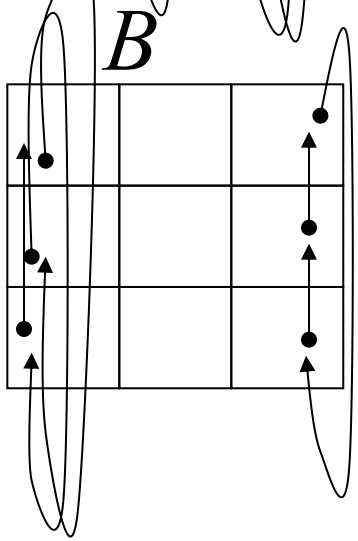


ローカルな
行列-行列積の後

：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】

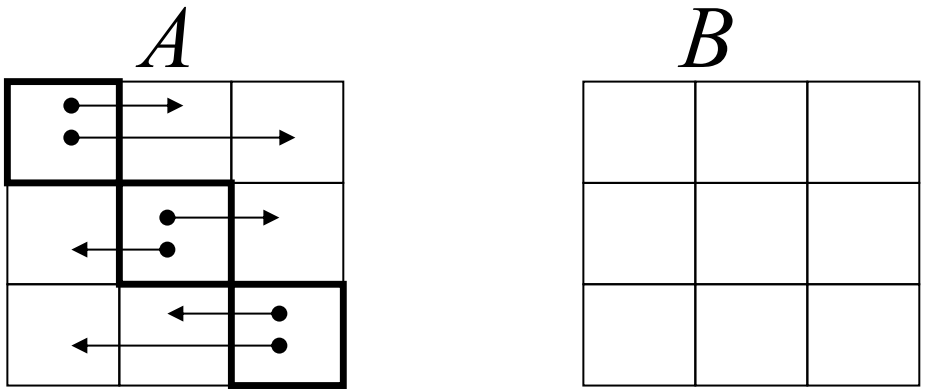
【通信パターンが
1つ右に循環シフト】

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

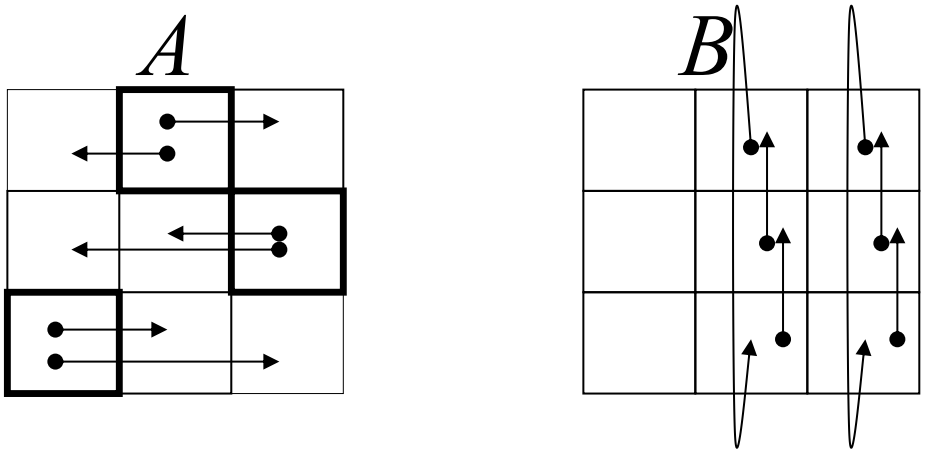
Foxのアルゴリズム

- 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】



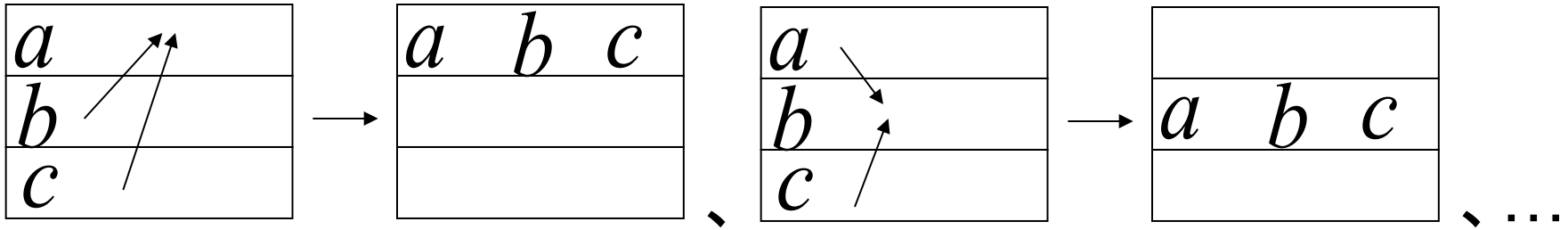
1つ上に通信

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）

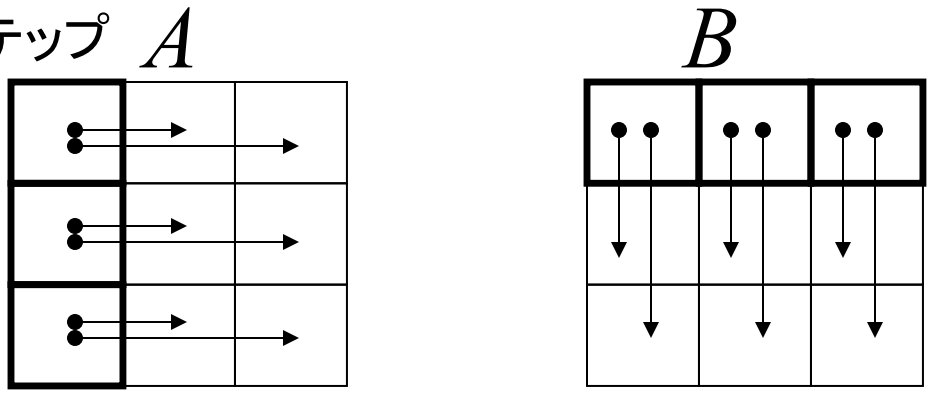
SUMMA、PUMMA

- 近年提案された並列アルゴリズム

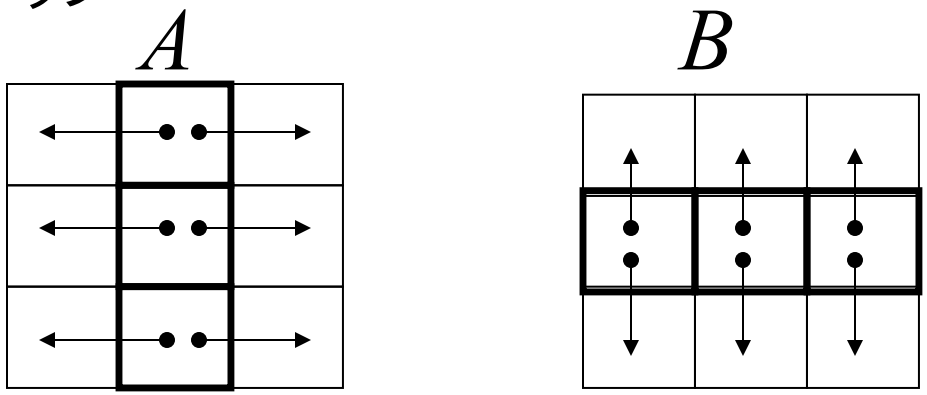
1. SUMMA（Scalable Universal Matrix Multiplication Algorithm）
   - R.Van de Geijinほか、1997年
   - 同時放送（マルチキャスト）のみで実現
2. PUMMA（Parallel Universal Matrix Multiplication Algorithms）
   - Choiほか、1994年
   - 二次元ブロックサイクリック分散方式むきのFoxのアルゴリズム

SUMMA

- アルゴリズムの概略
  - 第一ステップ

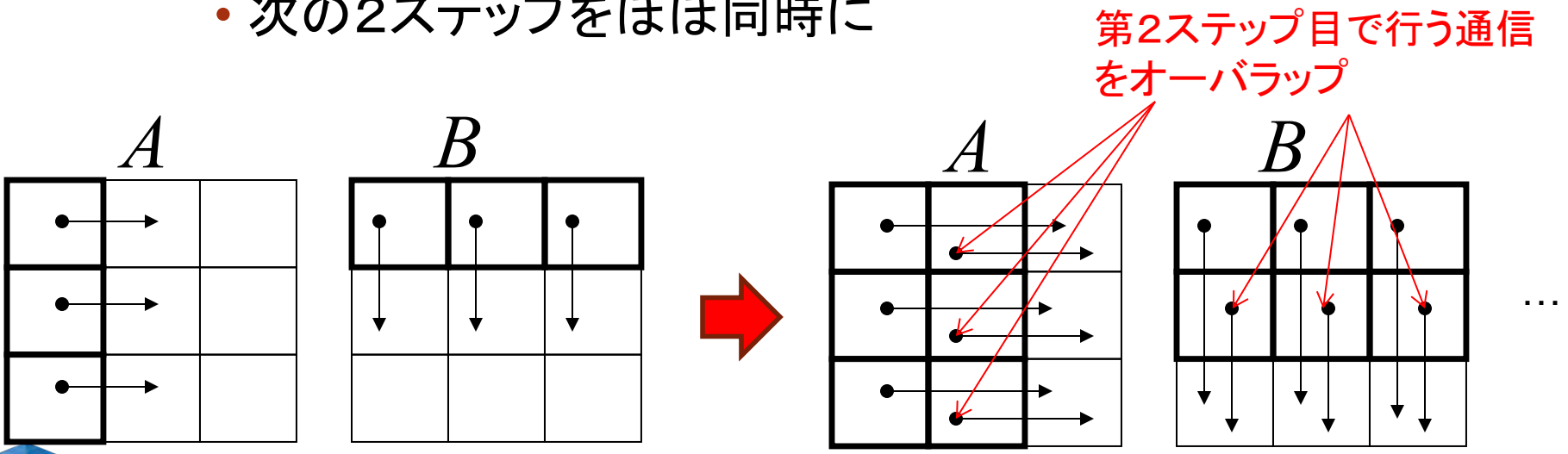


  - 第二ステップ



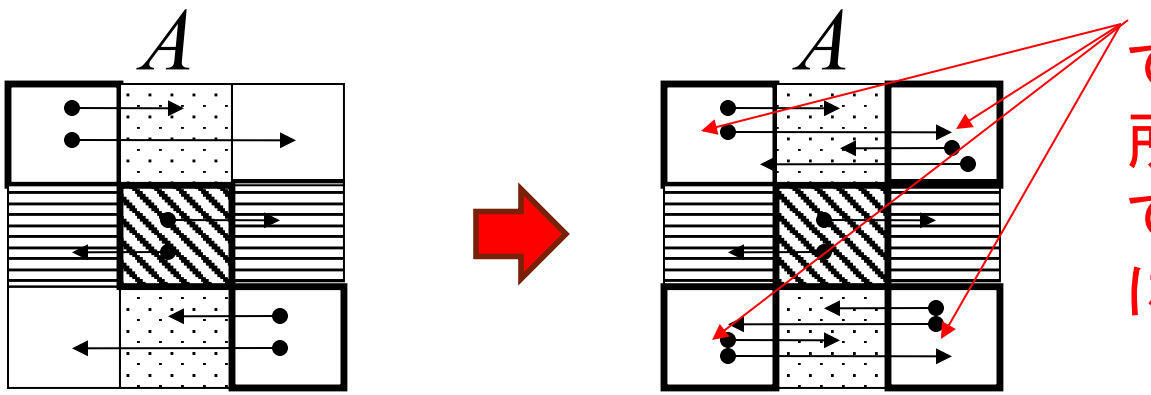
SUMMA

- 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。