第94回　お試しアカウント付き並列プログラミング講習会「KNL実践」

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 2月14日（水）　10：00 - 18：00
- 場所：東京大学情報基盤センター　4階　413遠隔会議室
- 講師：塙　敏博
- 講習会プログラム：
  - 9：30 - 10：00　受付
  - 10：00 - 12：00　Oakforest-PACSログイン、Oakforest-PACSシステム紹介、KNL概要
  - (12：00 - 13：30　昼休憩@小会議室)
  - 13：30 - 15：00　KNLにおける最適化（講義＋演習）
  - (15：00 - 15：15　休憩)
  - 15：15 - 18：00　Oakforest-PACSでのハイブリッド並列化と最適化 (講義+演習)

Oakforest-PACS利用の準備

配布資料「Oakforest-PACS利用の手引き」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： t00441～

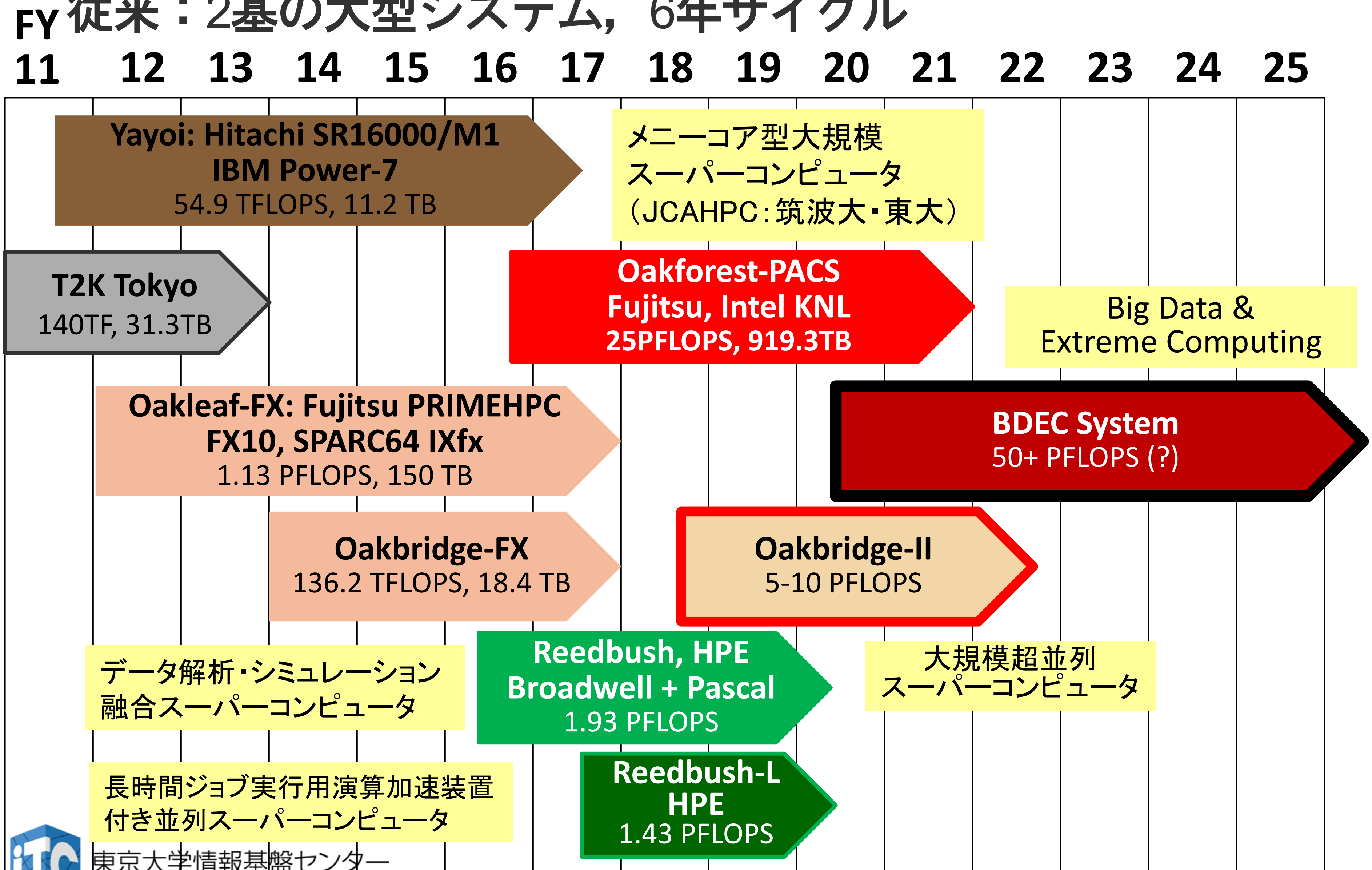
利用グループ： gt00

3/14 17:00まで有効

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

従来：2基の大型システム，6年サイクル



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

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 (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
  - 3.3 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)

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30日

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
+ 1272 TFLOPS
Total number of nodes : 120
Total memory : 30 TB + 8 TB
Peak performance per node : 1209.6 GFLOPS
+ 10.6 TFLOPS
Main memory per node : 256 GB
+ 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3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）

Reedbush-L （SGI Rackable クラスタシステム ）

長時間ジョブ用
制限時間：最大168時間
（１週間）

Total Peak performance	: 1.43 PFLOPS (77.4 TFLOPS + 1356 TFLOPS)
Total number of nodes	: 64
Total memory	: 16TB + 4 TB
Peak performance per node	: 1209.6 GFLOPS + 21.2 TFLOPS
Main memory per node	: 256 GB + 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

2017年11月1日正式運用開始

Reedbush-U+H+Lの合計性能
= 3.36 PFLOPS

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

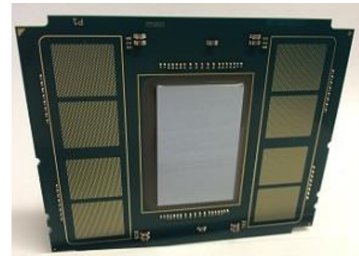
2017年4月3日正式運用開始

Oakforest-PACS 計算ノード

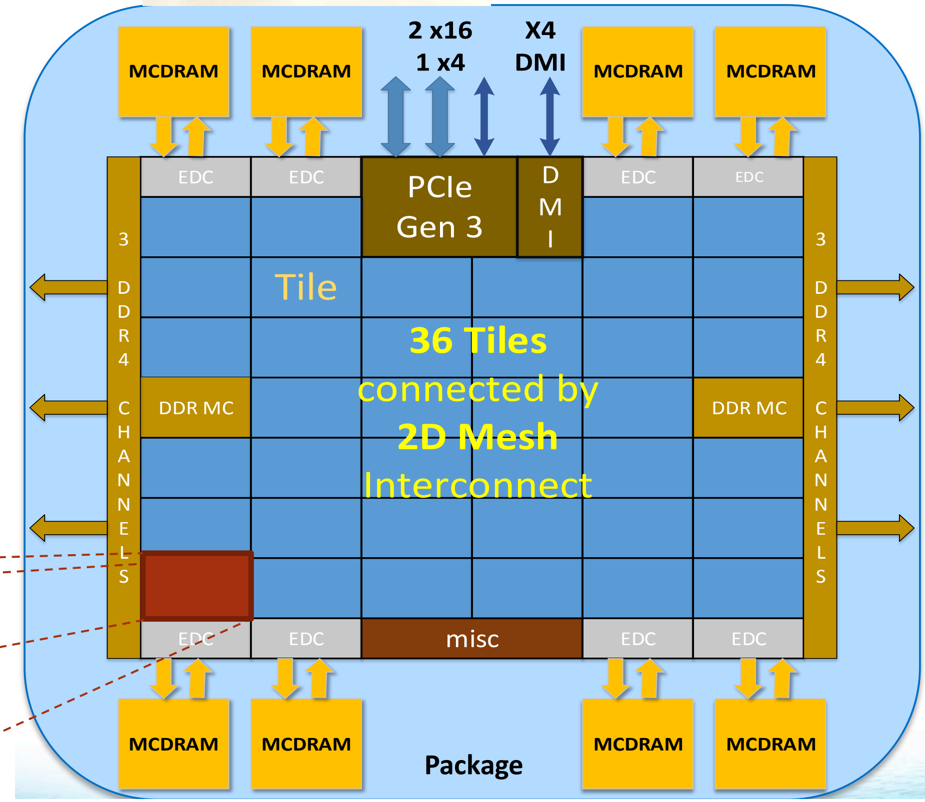
- Intel Xeon Phi (Knights Landing)
  - 1ノード1ソケット
- MCDRAM: オンパッケージの高バンド幅メモリ16GB + DDR4メモリ

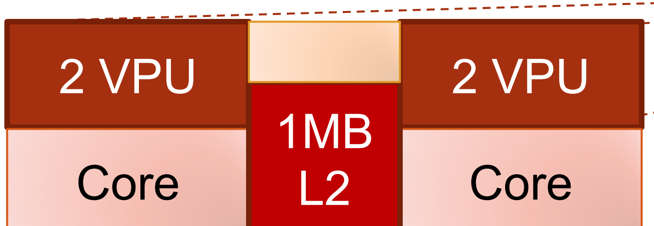
ソケット当たりメモリ量:
16GB×6＝96GB

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

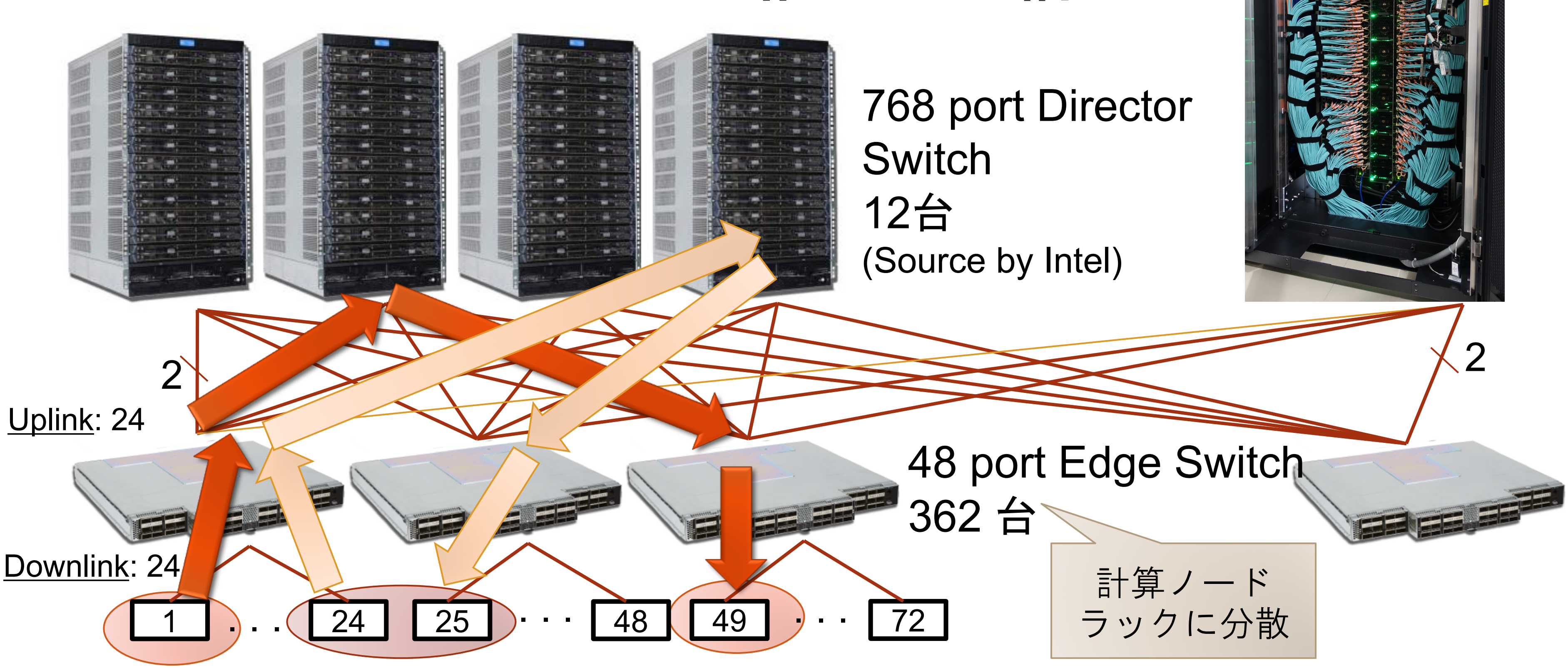


HotChips27
KNLスライドより





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（年間）
  - 100,000円 ： 1口8ノード(基準)、最大512ノードまで実行可
  - 3口まで
- 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
    - パーソナルコースは2ノード相当
  - 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との相互トークン移行も可能

JPY (=Watt)/GFLOPS Rate

Smaller is better (efficient)

System	JPY/GFLOPS
Oakleaf/Oakbridge-FX (Fujitsu) (Fujitsu PRIMEHPC FX10)	125
Reedbush-U (SGI) (Intel BDW)	62.0
Reedbush-H (SGI) (Intel BDW+NVIDIA P100)	17.1
Oakforest-PACS (Fujitsu) (Intel Xeon Phi/Knights Landing)	16.5

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/

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

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

並列プログラムの種類

- マルチプロセス
  - MPI （Message Passing Interface）
    - 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
  - 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
- メモリを共有できるかどうか
  - 異なる「プロセス」はメモリも別
  - 異なる「スレッド」でも同一のメモリを操作できる

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－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

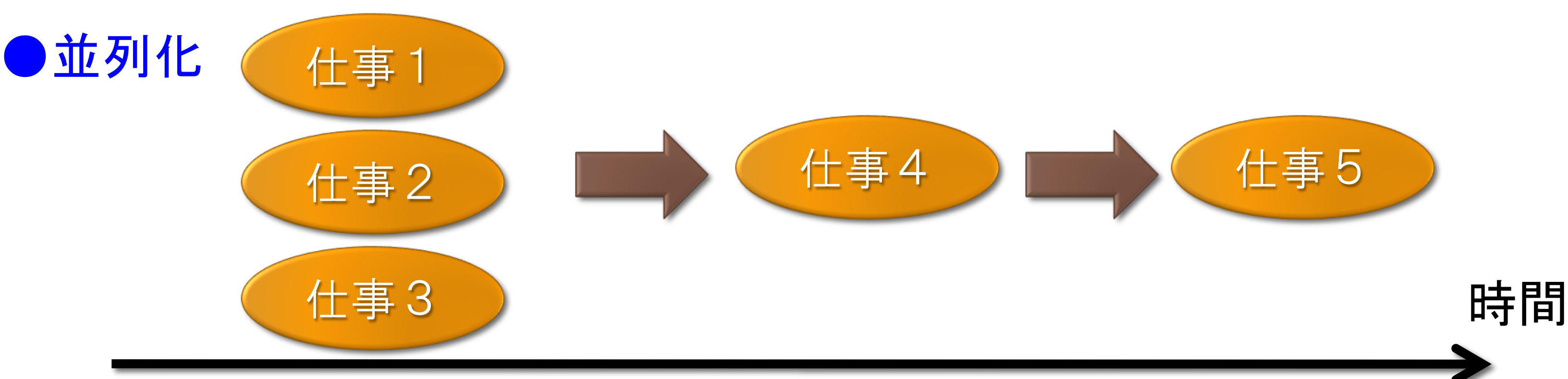
全CPUで共有

$$\begin{array}{l} \text{CPU0} \\ \text{CPU1} \\ \text{CPU2} \end{array} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1*9+2*6+3*3 & 1*8+2*5+3*2 & 1*7+2*4+3*1 \\ 4*9+5*6+6*3 & 4*8+5*5+6*2 & 4*7+5*4+6*1 \\ 7*9+8*6+9*3 & 7*8+8*5+9*2 & 7*7+8*4+9*1 \end{pmatrix}$$

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

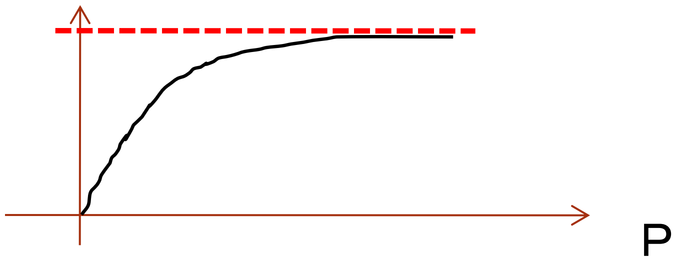


性能評価指標

並列化の尺度

性能評価指標－台数効果

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

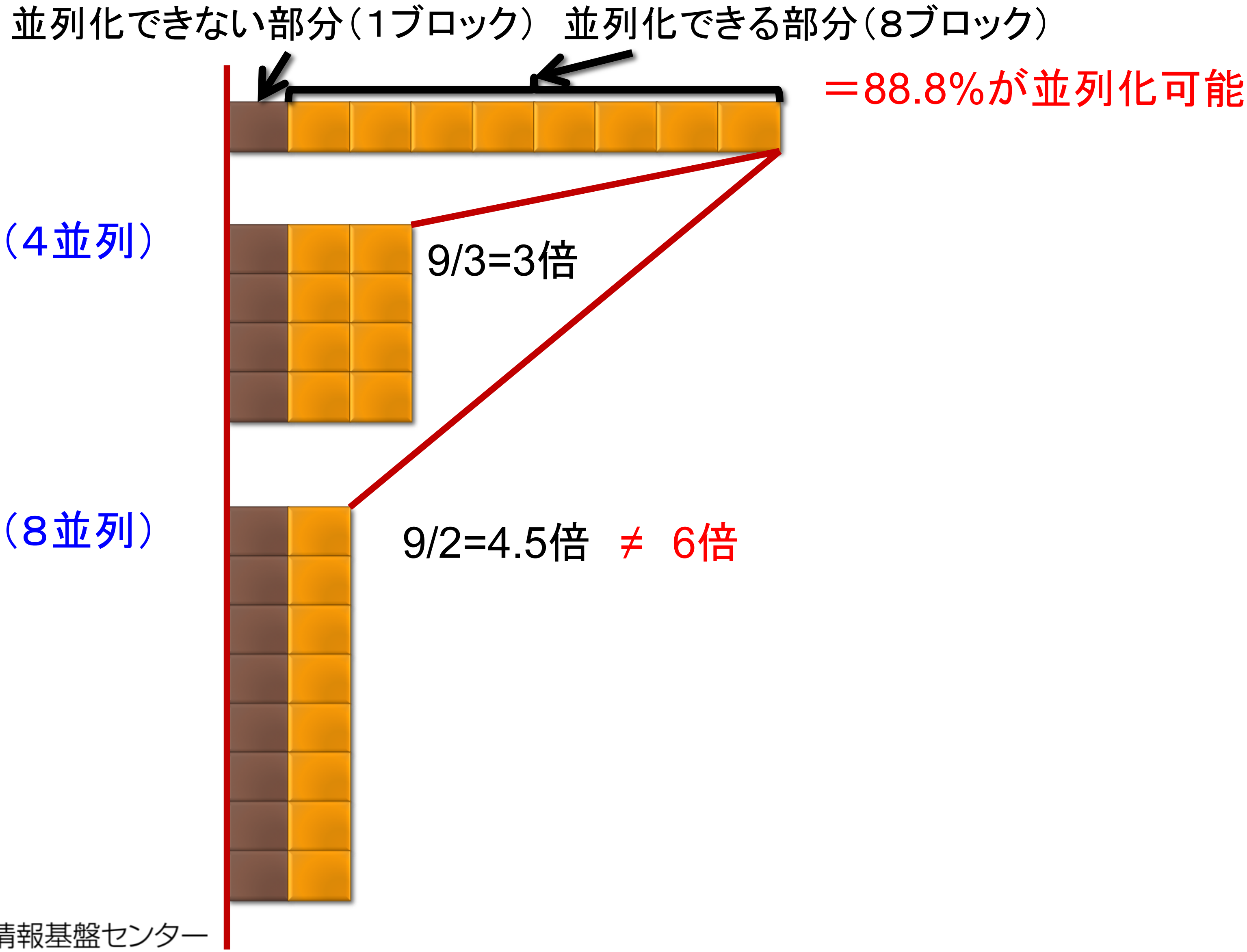
  →高性能を達成するためには、少しでも並列化効率を上げる実装をすることがとても重要である

アムダールの法則の直観例

●逐次実行

●並列実行（4並列）

●並列実行（8並列）



本講習会の目的

並列アルゴリズムはひとまず置いといて…

KNLの多数のコアの扱い方、Oakforest-PACSの多数の計算ノードを組み合わせて、効率良く扱う手法を学ぼう！

- その他の講習会 (予定) http://www.cc.u-tokyo.ac.jp/support/kosyu/
  - 並列アルゴリズムを学びたい方
    - 10/31,11/1 OpenMP/OpenACCによるマルチコア・メニィコア並列プログラミング入門
  - その他
    - 11/7, 8「並列有限要素法とハイブリッド並列プログラミング」
    - 3/1, 2　MPI 基礎
    - 未定　MPI 上級編
    - 未定　GPUクラスタ講習会(仮称)

「こんな講習会が欲しい！」という要望があればぜひアンケートへ！

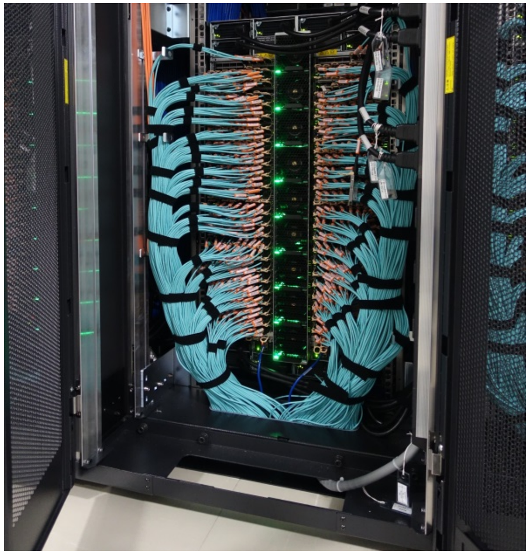
Oakforest-PACSの紹介

Oakforest-PACS

- 最先端共同HPC 基盤施設(JCAHPC: Joint Center for Advanced High Performance Computing)
  - 東京大学情報基盤センター
  - 筑波大学計算科学研究センター
    - 両センターが共同で、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
    - 東京大学柏キャンパスの東京大学情報基盤センター内
- 2016年12月1日稼働開始
- 8,208 Intel Xeon/Phi (KNL)
- ピーク性能25PFLOPS
- **TOP 500　7位（国内1位），HPCG 5位（国内2位）　（2017年6月）**

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
  - 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

<table>
<tr><td colspan="3">総ピーク演算性能</td><td>25 PFLOPS</td></tr>
<tr><td colspan="3">ノード数</td><td>8,208</td></tr>
<tr><td rowspan="4">計算ノード</td><td colspan="2">Product</td><td>富士通 PRIMERGY CX600 M1 (2U) + CX1640 M1 x 8node</td></tr>
<tr><td colspan="2">プロセッサ</td><td>Intel® Xeon Phi™ 7250
（開発コード: Knights Landing）
68 コア、1.4 GHz</td></tr>
<tr><td rowspan="2">メモリ</td><td>高バンド幅</td><td>16 GB, MCDRAM, 実効 490 GB/sec</td></tr>
<tr><td>低バンド幅</td><td>96 GB, DDR4-2400, ピーク 115.2 GB/sec</td></tr>
<tr><td rowspan="3">相互結合網</td><td colspan="2">Product</td><td>Intel® Omni-Path Architecture</td></tr>
<tr><td colspan="2">リンク速度</td><td>100 Gbps</td></tr>
<tr><td colspan="2">トポロジ</td><td>フルバイセクションバンド幅Fat-tree網</td></tr>
</table>

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 のソフトウェア

- OS: Red Hat Enterprise Linux (ログインノード)、CentOS および McKernel (計算ノード、切替可能)
  - McKernel: 理研AICSで開発中のメニーコア向けOS
    - Linuxに比べ軽量、ユーザプログラムに与える影響なし
    - ポスト京コンピュータにも搭載される予定。
- コンパイラ：GCC, Intel Compiler, XcalableMP
  - XcalableMP: 理研AICSと筑波大で共同開発中の並列プログラミング言語
    - CやFortranで記述されたコードに指示文を加えることで、性能の高い並列アプリケーションを簡易に開発することができる。
- ライブラリ・アプリケーション： オープンソースソフトウェア
  - ppOpen-HPC, OpenFOAM, ABINIT-MP, PHASE system, FrontFlow/blue , LAPACK, ScaLAPACK, PETSc, METIS, SuperLU etc.

各種ベンチマーク

- TOP 500（Linpack, HPL）
  - 連立一次方程式ソルバー（直接法），計算速度（FLOPS値）
  - 規則的な密行列：連続メモリアクセス
  - 計算性能
- HPCG
  - 連立一次方程式ソルバー（反復法），計算速度（FLOPS値）
  - 有限要素法から得られる疎行列（ゼロが多い）
    - 不連続メモリアクセス
    - 実アプリケーションに近い
  - メモリアクセス性能，通信性能
- Green 500
  - HPL（TOP500）実行時のFLOPS/W値

48th TOP500 List (November, 2016)

	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	Oak Ridge National Laboratory, USA	**Titan** Cray XK7/NVIDIA K20x, 2012 Cray	560,640	17,590	27,113	8,209
4	Lawrence Livermore National Laboratory, USA	**Sequoia** BlueGene/Q, 2011 IBM	1,572,864	17,173	20,133	7,890
5	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
6	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
7	RIKEN AICS, Japan	**K computer**, SPARC64 VIIIfx , 2011 Fujitsu	705,024	10,510	11,280	12,660
8	Swiss Natl. Supercomputer Center, Switzerland	**Piz Daint** Cray XC30/NVIDIA P100, 2013 Cray	206,720	9,779	15,988	1,312
9	Argonne National Laboratory, USA	**Mira** BlueGene/Q, 2012 IBM	786,432	8,587	10,066	3,945
10	DOE/NNSA/LANL/SNL, USA	**Trinity**, Cray XC40, Xeon E5-2698v3 16C 2.3GHz, 2016 Cray	301,056	8,101	11,079	4,233

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

http://www.top500.org/

49th TOP500 List (June, 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	Oak Ridge National Laboratory, USA	**Titan** Cray XK7/NVIDIA K20x, 2012 Cray	560,640	17,590	27,113	8,209
5	Lawrence Livermore National Laboratory, USA	**Sequoia** BlueGene/Q, 2011 IBM	1,572,864	17,173	20,133	7,890
6	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
7	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
8	RIKEN AICS, Japan	**K computer**, SPARC64 VIIIfx , 2011 Fujitsu	705,024	10,510	11,280	12,660
9	Argonne National Laboratory, USA	**Mira** BlueGene/Q, 2012 IBM	786,432	8,587	10,066	3,945
10	DOE/NNSA/LANL/SNL, USA	**Trinity**, Cray XC40, Xeon E5-2698v3 16C 2.3GHz, 2016 Cray	301,056	8,101	11,079	4,233

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

http://www.top500.org/

50^{th} 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 (SC16, November, 2016)

	Site	Computer	Cores	HPL Rmax (Pflop/s)	TOP500 Rank	HPCG (Pflop/s)	HPCG/ HPL (%)
1	RIKEN AICS, Japan	K computer	705,024	10.510	7	0.6027	5.73
2	NSCC / Guangzhou, China	Tianhe-2	3,120,000	33.863	2	0.5800	1.71
3	JCAHPC, Japan	Oakforest-PACS	557,056	13.555	6	0.3855	2.84
4	National Supercomputing Center in Wuxi, China	Sunway TaihuLight	10,649,600	93.015	1	0.3712	.399
5	DOE/SC/LBNL/NERSC USA	Cori	632,400	13.832	5	0.3554	2.57
6	DOE/NNSA/LLNL, USA	Sequoia	1,572,864	17.173	4	0.3304	1.92
7	DOE/SC/ Oak Ridge National Laboratory, USA	Titan	560,640	17.590	3	0.3223	1.83
8	DOE/NNSA/ LANL/SNL, USA	Trinity	301,056	8.101	10	0.1826	2.25
9	NASA / Mountain View, USA	Pleiades: SGI ICE X	243,008	5.952	13	0.1752	2.94
10	DOE/SC/ Argonne National Laboratory, USA	Mira: IBM BlueGene/Q,	786,432	8.587	9	0.1670	1.94

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

HPCG Ranking (June, 2017)

	Computer	Cores	HPL Rmax (Pflop/s)	TOP500 Rank	HPCG (Pflop/s)	Peak
1	**K computer**	705,024	10.510	8	0.6027	5.3%
2	**Tianhe-2 (MilkyWay-2)**	3,120,000	33.863	2	0.5801	1.1%
3	**Sunway TaihuLight**	10,649,600	93.015	1	0.4808	0.4%
4	**Piz Daint**	361,760	19.590	3	0.4767	1.9%
5	**Oakforest-PACS**	557,056	13.555	7	0.3855	1.5%
6	**Cori**	632,400	13.832	6	0.3554	1.3%
7	**Sequoia**	1,572,864	17.173	5	0.3304	1.6%
8	**Titan**	560,640	17.590	4	0.3223	1.2%
9	**Trinity**	301,056	8.101	10	0.1826	1.6%
10	**Pleiades** – NASA/SGI	243,008	5.952	15	0.1752	2.5%

http://www.hpcg-benchmark.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/

Green 500 Ranking (Nov., 2017)

	Site	Computer	CPU	HPL Rmax (Pflop/s)	TOP500 Rank	Power (kW)	GFLOPS/W
1	RIKEN, Japan	Shoubu system B	ZettaScaler-2.2 HPC system, Xeon D-1571, PEZY-SC2 , ExaScalar	842.0	259	50	17.009
2	KEK, Japan	Suiren2	ZettaScaler-2.2 HPC system, Xeon D-1571, PEZY-SC2 , ExaScalar	788.2	307	47	16.759
3	PEZY, Japan	Sakura	ZettaScaler-2.2 HPC system, Xeon E5-2618Lv3, PEZY-SC2 , ExaScalar	824.7	276	50	16.657
4	**NVIDIA, USA**	**DGX Saturn V Volta**	**Xeon E5-2698v4, NVIDIA Tesla V100 , Nvidia**	**1,070.0**	**149**	**97**	**15.113**
5	JAMSTEC, Japan	Gyoukou	ZettaScaler-2.2 HPC system, Xeon D-1571, PEZY-SC2 , ExaScalar	19,135.8	4	1,350	14.173
6	**Tokyo Tech.**	**TSUBAME3.0**	**SGI ICE XA, IP139-SXM2, Xeon E5-2680v4, NVIDIA Tesla P100 SXM2, HPE**	**8,125.0**	**13**	**792**	**13.704**
7	**AIST, Japan**	**AIST AI Cloud**	**NEC 4U-8GPU Server, Xeon E5-2630Lv4, NVIDIA Tesla P100 SXM2 , NEC**	**961.0**	**148**	**76**	**12.681**
8	**CAIP, RIKEN, JAPAN**	**RAIDEN GPU subsystem -**	**NVIDIA DGX-1, Xeon E5-2698v4, NVIDIA Tesla P100 , Fujitsu**	**635.1**	**305**	**60**	**10.603**
9	**Univ. Cambridge, UK**	**Wilkes-2 -**	**Dell C4130, Xeon E5-2650v4, NVIDIA Tesla P100 , Dell**	**1,193.0**	**100**	**114**	**10.428**
10	**Swiss Natl. SC. Center (CSCS)**	**Piz Daint**	**Cray XC50, Xeon E5-2690v3, NVIDIA Tesla P100 , Cray Inc.**	**19,590.0**	**3**	**2,272**	**10.398**
11	**ITC, U.Tokyo, Japan**	**Reedbush-L**	**SGI Rackable C1102-GP8, Xeon E5-2695v4, NVIDIA Tesla P100 SXM2 , HPE**	**805.6**	**291**	**79**	**10.167**

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/

運用

- 2017年度3月末までは無料（但し停止期間等あり）
- 計算資源は全系を共用（パーティション分けはしない）
  - 全8,208ノード（25PF）を常に全系で運用できるようにしておき，国内最大の計算資源を有効に活用する

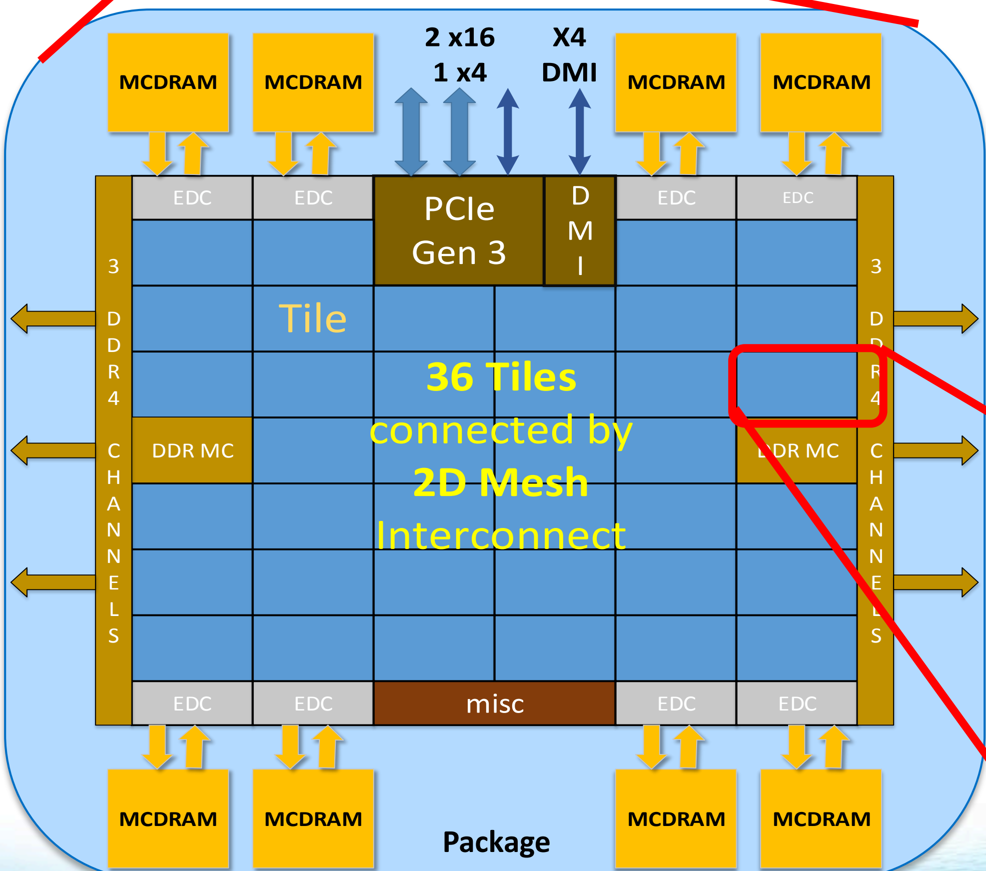
○利用形態

- 各大学独自の利用コース
- HPCI
  - 全資源の20%を「JCAHPC」として拠出，企業利用可能
- JHPCN（学際大規模情報基盤共同利用共同研究拠点）
  - 東大分全資源の5%程度：企業共同研究，国際共同研究も含む（東大のみ）
- 教育（講義，講習会）
- 大規模HPCチャレンジ：全ノード占有，1回24時間程度/月

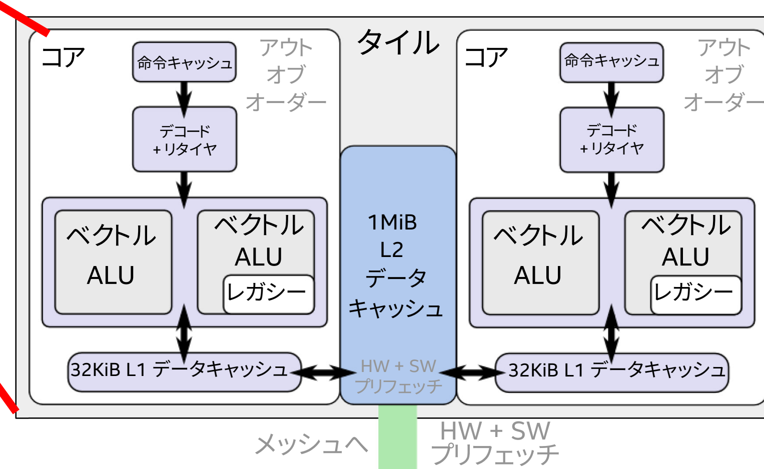
（理論）ピーク性能

- OFPに搭載されているXeon Phi 7250のピーク性能を考える。
  - コア数： 68コア
  - コア当たりのAVX-512 ユニット： 2
  - AVX-512ユニット当たりの同時演算数（倍精度）： 8
  - 積和演算 (Fused Multiply Add: FMA) : 2 に換算
  - クロック周波数: 1.40 GHz
- ノード当たりピーク性能：
  68 * 2 * 8 * 2 * 1.40 = 3046.4 GFLOPS
- しかし、AVX-512ユニットは実は1.40GHzでは動作しない（より低い周波数）
  - ピークに近い性能が得られるはずのもの（OFPでの実測値）
    - DGEMM (倍精度の行列積）： 2200 GFLOPS (ピーク比：72%)
    - HPL: 2000 GFLOPS (ピーク比： 66%)
  - Top500におけるOFPの登録値はピークの54.4%
- 「ピーク性能」の定義を正しく把握しておくことが重要！！
  - CPUメーカーによって定義の仕方も異なる

Intel Xeon Phi (KNL: Knights Landing)



- Atom (Silvermont) コア + AVX512 x2
- 2コア 1タイル
  - 各コアは 4スレッド (HyperThreading)
- 64, 68コア（32, 34タイル）
- MCDRAM: オンパッケージの高バンド幅メモリ16GB+ DDR4メモリ



出典：左：HotChips 27 KNLスライド, 右：Colfax (翻訳：エクセルソフト)

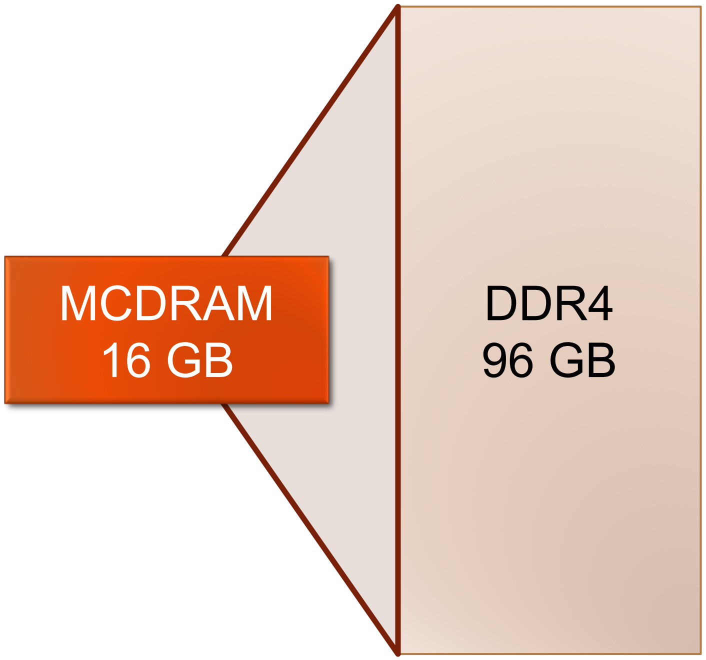
KNLの動作モード

- メモリモード：3種類
  - Flat: MCDRAMとDDR4 が独立したアドレス
  - Cache: MCDRAMはDDR4メモリのキャッシュとして動作
  - Hybrid
- クラスタリングモード: 5種類
  - (All-to-all: アドレス情報が全体に分散… 非推奨)
  - Quadrant, Hemisphere: 内部でアドレス情報が4(または2)に分割（ユーザからは見えない）
  - SNC-4, SNC-2: NUMAドメインが明示的に4 (or 2)に分割

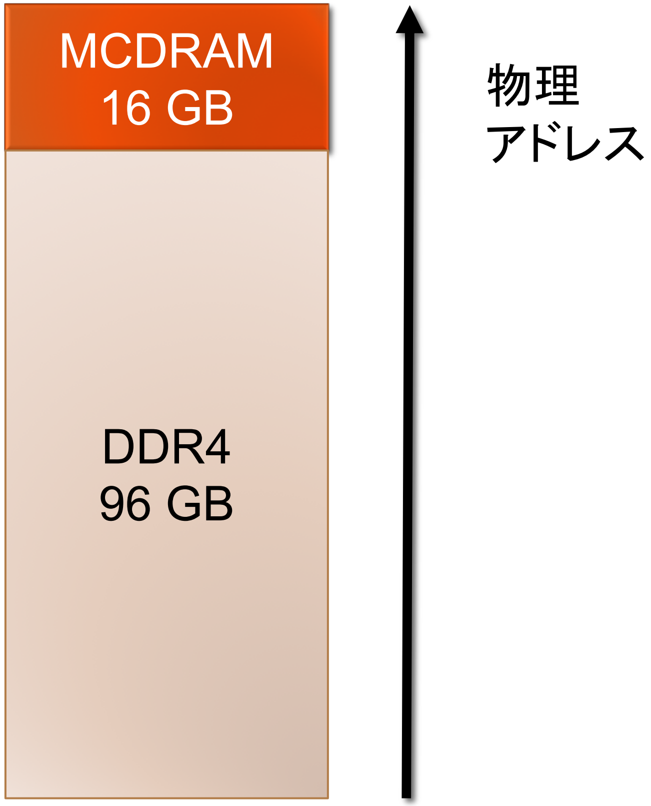
モードの変更には再起動が必要
=> 現時点では、各モード(Flat, Cache)のジョブキューを用意

メモリモード

- Cacheモード
  - MCDRAMはL3キャッシュとして動作

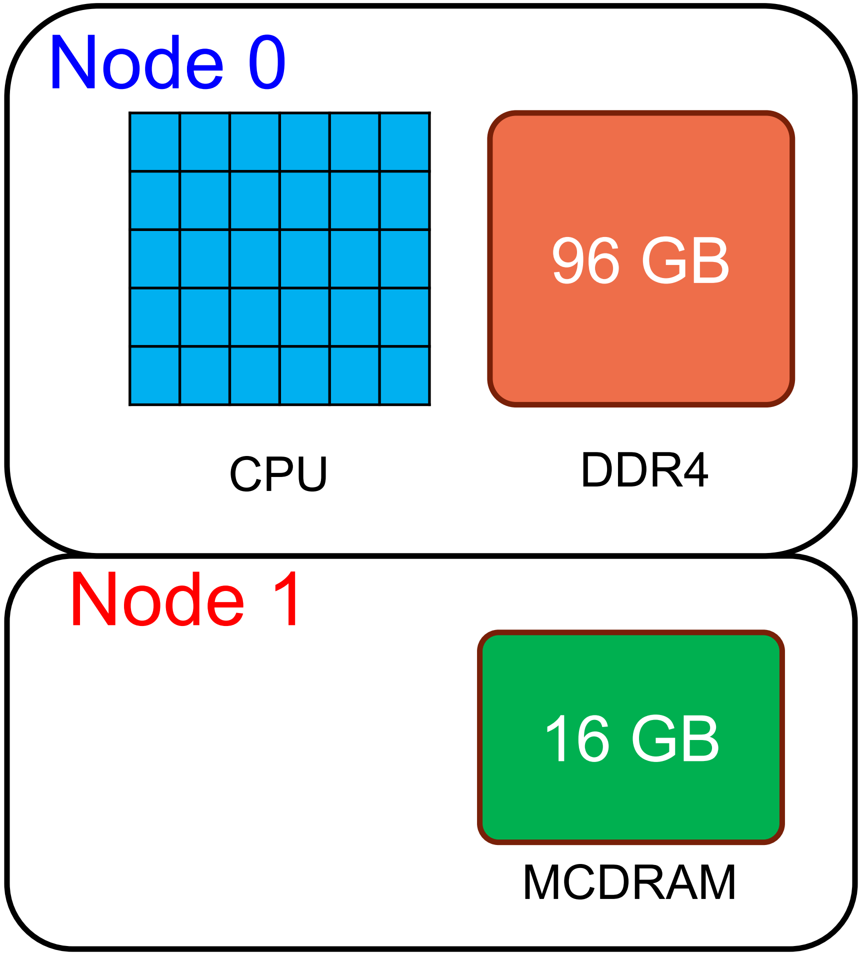


- Flatモード
  - MCDRAMを明示的に使い分け



FlatモードにおけるNUMAドメイン

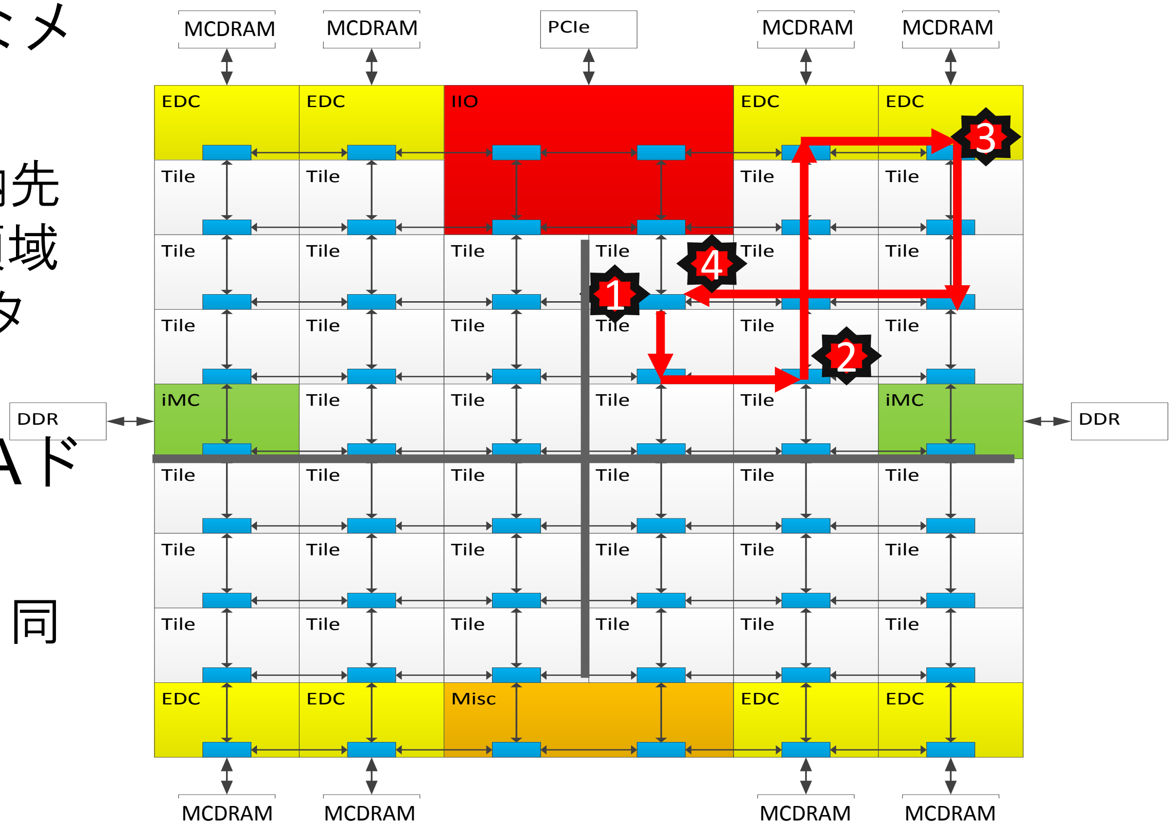
- $ numactl -H
  available: 2 nodes (0-1)
  node 0 cpus: 0 1 2 3 4 5 6 … 271
  node 0 size: 98164 MB
  node 0 free: 92027 MB
  node 1 cpus: （何もない）
  node 1 size: 16384 MB
  node 1 free: 15832 MB
  node distances:
  node 0 1
  0: 10 31
  1: 31 10



クラスタリングモード

現在OFPではQuadrantのみ

- Quadrant: ソフトウェアからはフラットなメモリ空間に見える
  - 内部では4分割、格納先のMCDRAMと同じ領域にキャッシュ情報（タグ）を配置
- SNC-4: 4つのNUMAドメインに見える
  - 4ソケットある場合と同じ

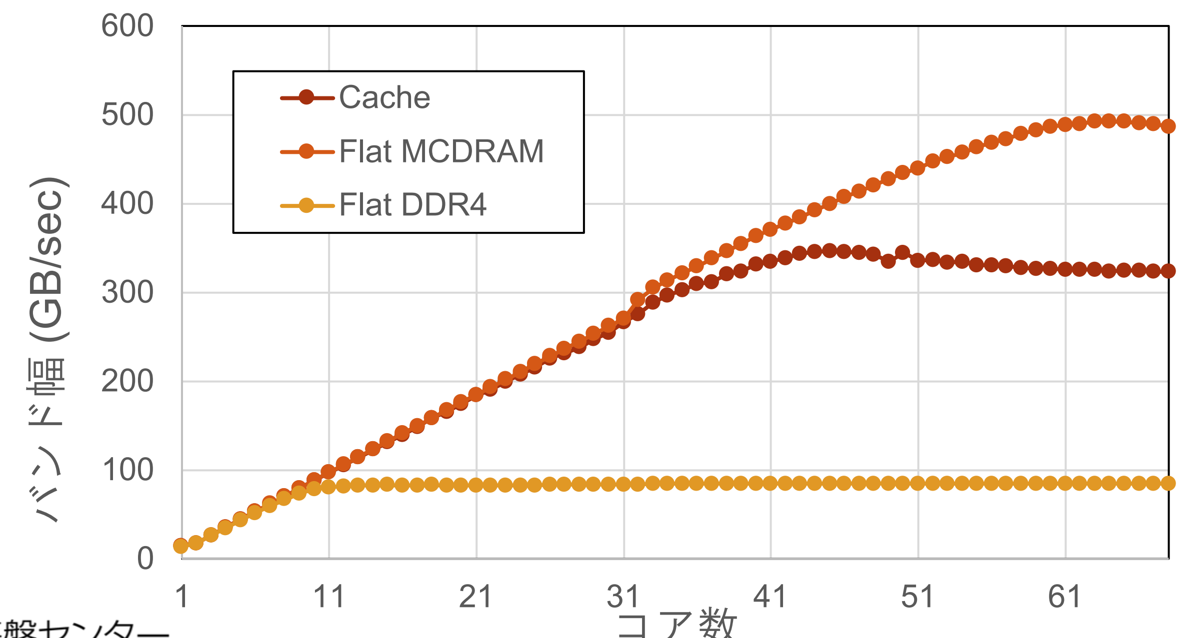


HotChips27
KNLスライドより

Stream Triad性能の比較

- DDR4: micprun -D -k stream
- MCDRAM: micprun -k stream
  - unset KMP_AFFINITY
  - export KMP_HW_SUBSET=66c@2,1t
    タイル0を避ける設定（後述）

	DDR4	MCDRAM
Cache	346.5 GB/sec	
Flat	84.8 GB/sec	494.8 GB/sec



FlatモードでのMCDRAMの使い方(1)

(1) numactlによる方法

- MCDRAMはNUMAノード1として認識されている
  - ```
    $ numactl -H
    available: 2 nodes (0-1)
    node 0 cpus: 0 1 2 3 4 … (省略) … 268 269 270 271
    node 0 size: 98164 MB
    node 0 free: 85707 MB
    node 1 cpus:
    node 1 size: 16384 MB
    node 1 free: 15496 MB
    node distances:
    node   0   1
      0:  10  31
      1:  31  10
    ```

- コード書き換えを避けたい場合には便利

× 配列毎の細かな使い分けはできない

- プログラム実行時に以下を指定
  - `$ numactl --membind 1 ./a.out`
    (MCDRAM 16GBに入りきらないとエラー)
  - `$ numactl --preferred 1 ./a.out`
    (MCDRAMに収まり切らない場合、DDR4も使用する)
  - `$ numactl --interleave 1,0 ./a.out`
    (MCDRAMとDDR4を交互に使用する)
    ```

FlatモードでのMCDRAMの使い方(2)

(2) 環境変数による方法：Intel MPIを使う場合

- I_MPI_HBW_POLICY環境変数
  - export I_MPI_HBW_POLICY=hbw_bind
  - export I_MPI_HBW_POLICY=hbw_preferred
  - export I_MPI_HBW_POLICY=hbw_interleave

- 事実上numactlを呼ぶのと一緒

- コード書き換えを避けたい場合には便利

× 配列毎の細かな使い分けはできない
× Intel MPIのときしか有効にならない

FlatモードでのMCDRAMの使い方(3)

(3) memkindライブラリによる方法

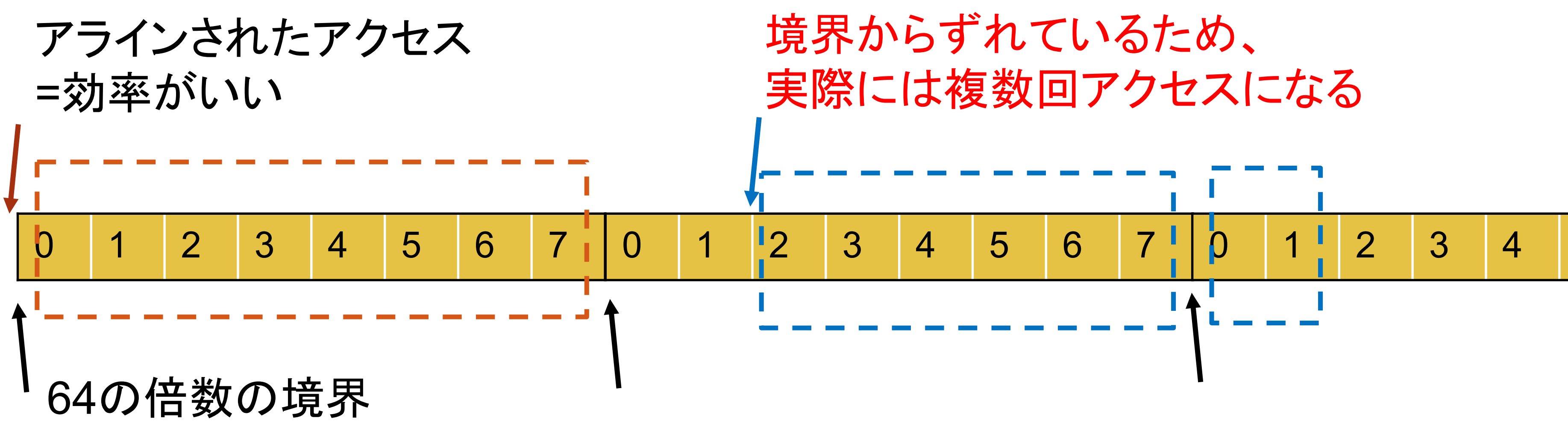
- C
  - #include <hbwmalloc.h> を指定
  - 以下の通り、メモリ操作関連の関数名に”hbw_”をつける
    - malloc() => hbw_malloc()
    - posix_memalign() => hbw_posix_memalign()
    - free() => hbw_free()
  - リンク時に -lmemkind を指定
- Fortran
  - Fastmemの属性を指定
  - !dir$ attributes fastmem :: array
  - リンク時に -lmemkind を指定
- hbw_set_policy()で切り替え
  - HBW_POLICY_BIND
  - HBW_POLICY_PREFERRED （デフォルト）
  - HBW_POLICY_INTERLEAVE

- 配列毎の細かな使い分けが可能

× 動的配列(allocatable)にしか適用できない

メモリのアラインメント

- AVX-512命令で512ビットデータを有効に使うためには、データがメモリ境界に揃っていることが必要
    - 512ビット = 64バイトの倍数のアドレスにアクセスするように
- 境界以外では複数回のアクセスになる
    - 例：倍精度実数(double)を連続8個(=512ビット)読み出す場合



メモリアラインメントの指定

- C言語
  - __attribute__((aligned(64))) を指定 (64 byte = 512 bit)
    - 例： double A[1024] __attribute__((aligned(64)))
  - 動的確保の際は posix_memalign() を使用
    - 上と同様の例：

```
#include <stdlib.h>
double *A;
posix_memalign(&A, 64, 8192);
```

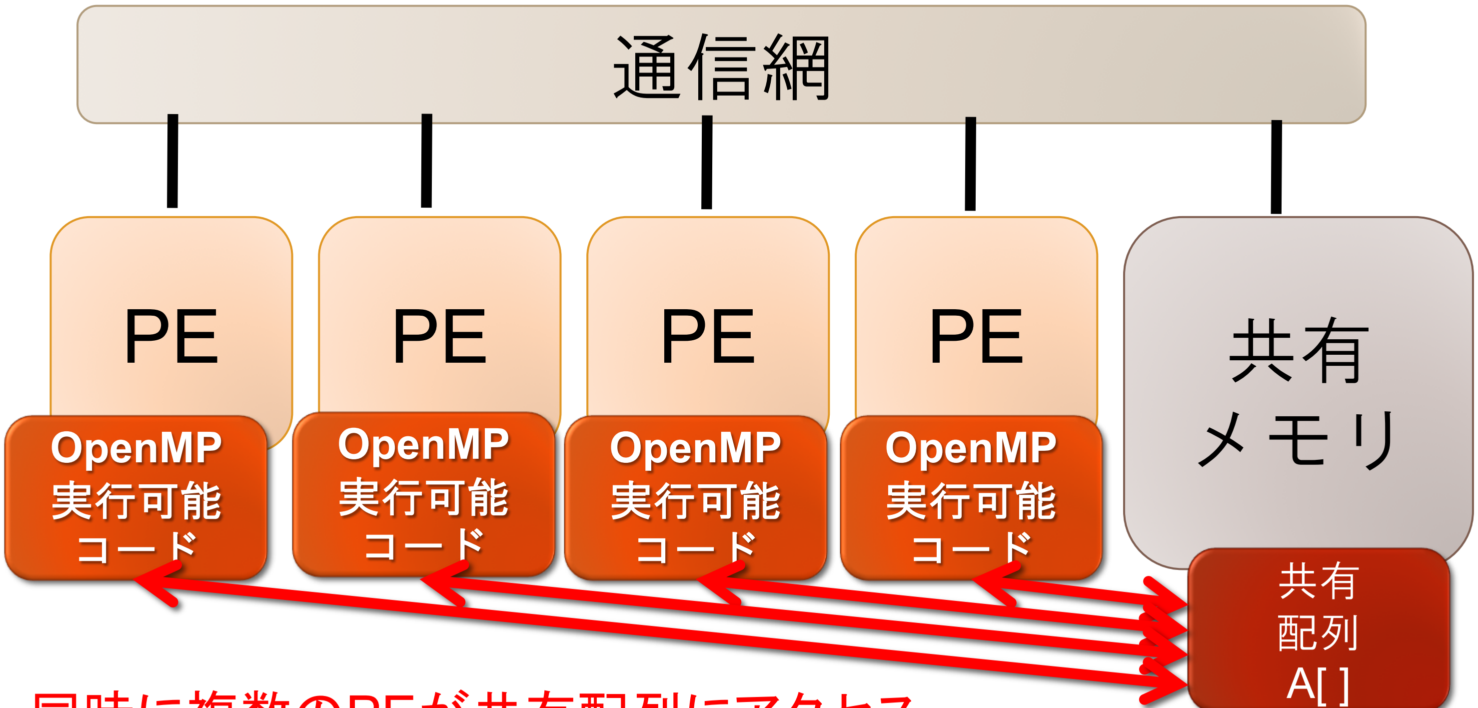
- Fortran
  - コンパイル時に -align array64byte オプションを指定
  - 動的な場合はattribute指定が必要かもしれない(経験上)

```
real*8, dimension(:), allocatable :: A
!dir$ attributes align:64 :: A
```

OpenMPの概要

OpenMPの対象計算機

- OpenMPは共有メモリ計算機のためのプログラム言語



同時に複数のPEが共有配列にアクセス

⇒並列処理で適切に制御をしないと、逐次計算の結果と一致しない

OpenMPとは

- OpenMP (OpenMP Application Program Interface)とは、共有メモリ型並列計算機用にプログラムを並列化するための：
  1. 指示文
  2. ライブラリ
  3. 環境変数

  を規格化したものです。
- ユーザが、並列プログラムの実行させるための指示を与えるものです。コンパイラによる自動並列化ではありません。
- 分散メモリ型並列化（MPIなど）に比べて、データ分散の処理の手間が無い分、実装が簡単です。

OpenMPとマルチコア計算機（その１）

- スレッド並列化を行うプログラミングモデル
- マルチコア、メニーコア計算機に適合
  - 多数のスレッド実行で高い並列化効率を確保するには、プログラミングの工夫が必要
    1. メインメモリ-キャッシュ間のデータ転送能力が演算性能に比べ低い
    2. OpenMPで並列性を抽出できないプログラムになっている（後述）
- ノード間の並列化はOpenMPではできない
  - ノード間の並列化はMPIを用いる

OpenMPとマルチコア計算機（その２）

- 典型的なスレッド数
  - 16スレッド／ノード
    - T2Kオープンスパコン（AMD Quad Core Opteron(Barcelona)、4ソケット）、FX10スーパコンピュータシステム（Sparc64 IXfx）
  - 32～128スレッド／ノード
    - HITACHI SR16000 (IBM Power7)
      - 32物理コア、64～128論理コア（SMT利用時）
    - Reedbush (Intel Xeon E5-2695 v4, Broadwell-EP)
      - 36コア
  - 60～272スレッド／ノード
    - Intel Xeon Phi (Intel MIC(Many Integrated Core)、Knights Conner)
      - 60物理コア、120～240論理コア（HT利用時）
    - Oakforest-PACS (Intel MIC, Knights Landing)
      - 68物理コア、272論理コア利用可能
- 100スレッドを超えたOpenMPによる実行形態が可能に
  - 相当のプログラム上の工夫が必要

OpenMPコードの書き方の原則

- C言語の場合
  - #pragma omp で始まるコメント行
- Fortran言語の場合
  - !$omp で始まるコメント行

OpenMPのコンパイルの仕方

- 逐次コンパイラのコンパイルオプションに、OpenMP用のオプションを付ける
  - 例）Intel Fotran90コンパイラ
    ifort -O3 -qopenmp foo.f
  - 例）Intel Cコンパイラ
    icc -O3 -qopenmp foo.c
- 注意
  - OpenMPの指示がないループは逐次実行
  - コンパイラにより、自動並列化によるスレッド並列化との併用ができる場合があるが、できない場合もある
    - OpenMPの指示行がある行はOpenMPによるスレッド並列化、指示がないところはコンパイラによる自動並列化
    - 例）Intel Fortran90コンパイラ
      ifort -O3 -qparallel -qopenmp foo.f

OpenMPの実行可能ファイルの実行

- OpenMPのプログラムをコンパイルして生成した実行可能ファイルの実行は、そのファイルを指定することで行う
- スレッド数を、環境変数OMP_NUM_THREADSで指定
- 例）OpenMPによる実行可能ファイルがa.outの場合

  $ export OMP_NUM_THREADS=16

  $ ./a.out

または

  $ env OMP_NUM_THREADS=16 ./a.out

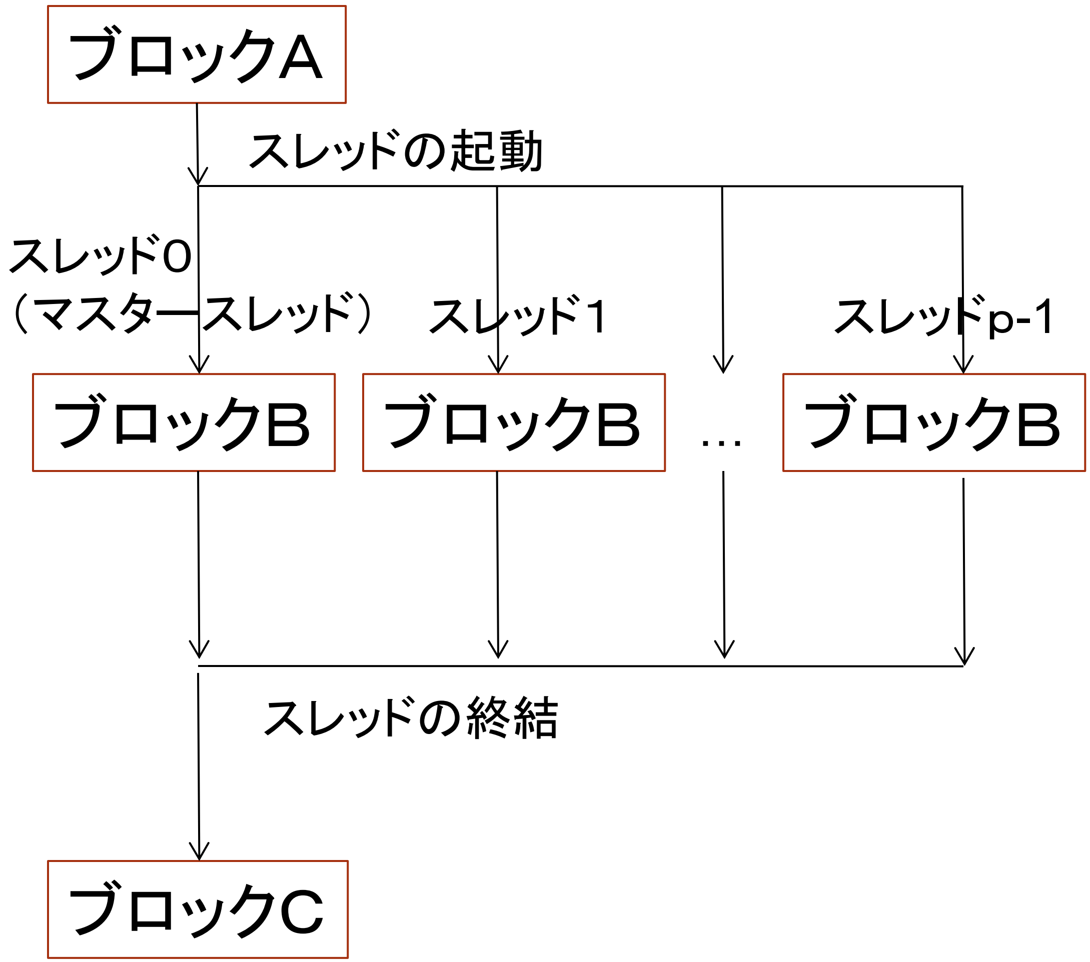
- 注意
  - 逐次コンパイルのプログラムと、OpenMPによるプログラムの実行速度が、OMP_NUM_THREADS=1にしても、異なることがある（後述）
    - この原因は、OpenMP化による処理の増加（オーバーヘッド）
    - 高スレッド実行で、このオーバーヘッドによる速度低下が顕著化
    - プログラミングの工夫で改善可能

OpenMPの実行モデル

OpenMPの実行モデル（C言語）

OpenMP指示文

```
    ブロックA
#pragma omp parallel
{
    ブロックB
}
    ブロックC
```

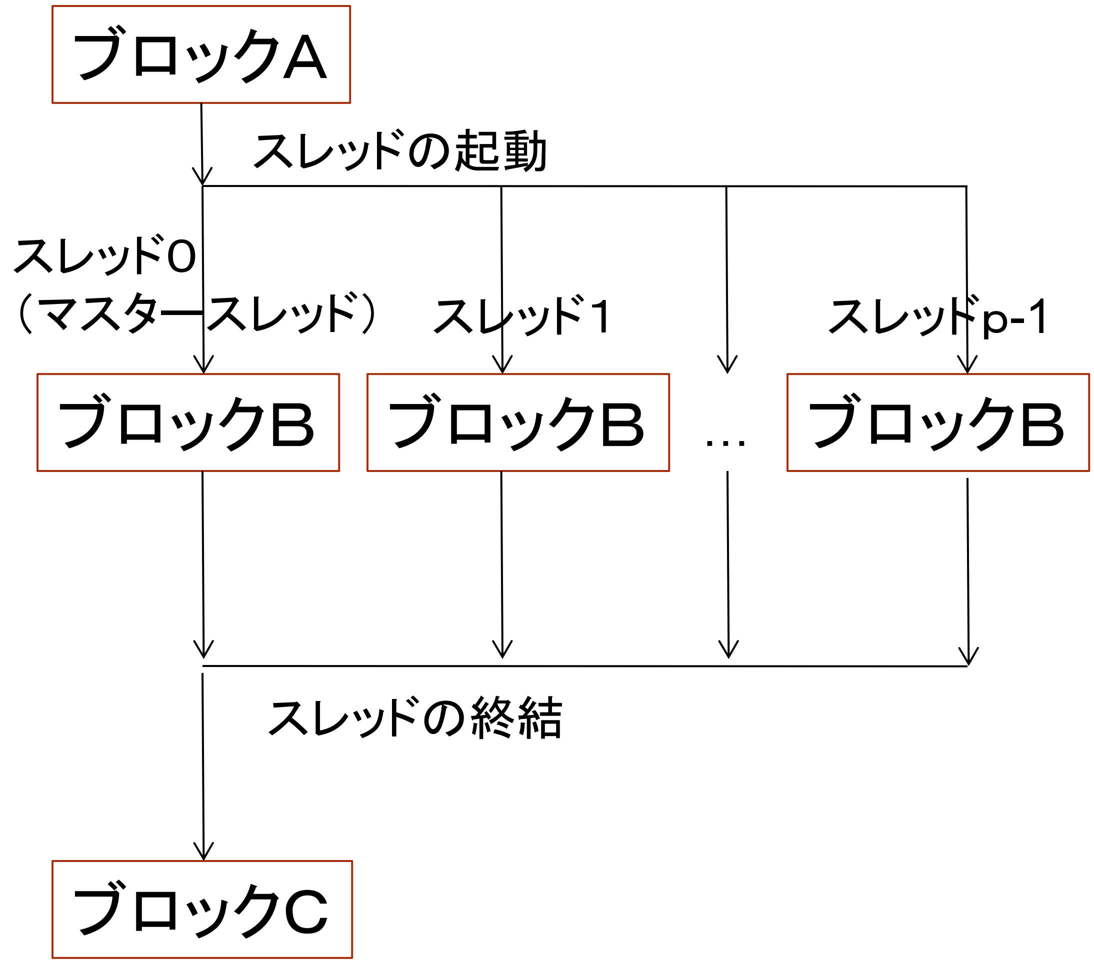


※スレッド数pは、
環境変数
OMP_NUM_THREADS
で指定する。

OpenMPの実行モデル（Fortran言語）

OpenMP指示文

```
    ブロックA
!$omp  parallel
    ブロックB
!$omp  end parallel
    ブロックC
```



※スレッド数pは、
環境変数
OMP_NUM_THREADS
で指定する。

Work sharing構文

- parallel指示文のように、複数のスレッドで実行する場合において、OpenMPで並列を記載する処理（ブロックB）の部分を並列領域(parallel region)と呼ぶ。
- 並列領域を指定して、スレッド間で並列実行する処理を記述するOpenMPの構文をWork sharing構文と呼ぶ。
- Work sharing構文は、大きく分けて以下の２種がある。
  1. 並列領域内で記載するもの
     - for構文（do構文）
     - sections構文
     - single構文 (master構文)、など
  2. parallel指示文と組み合わせるもの
     - parallel for 構文 (parallel do構文)
     - parallel sections構文、など

代表的な指示文

For構文（do構文）

```
#pragma omp parallel for
for (i=0; i<100; i++){
  a[i] = a[i] * b[i];
}
```

※Fortran言語の場合は

```
!$omp  parallel  do
～
!$omp end parallel do
```

上位の処理

スレッドの起動

スレッド0

```
for (i=0; i<25; i++){
  a[i] = a[i] * b[i];
}
```

スレッド1

```
for (i=25; i<50; i++){
  a[i] = a[i] * b[i];
}
```

スレッド2

```
for (i=50; i<75; i++){
  a[i] = a[i] * b[i];
}
```

スレッド3

```
for (i=75; i<100; i++){
  a[i] = a[i] * b[i];
}
```

スレッドの終結

下位の処理

※並列化をしても
正しい結果になることを
ユーザが保障する。

For構文の指定ができない例

```
for (i=0; i<100; i++) {
  a[i] = a[i] +1;
  b[i] = a[i-1]+a[i+1];
}
```

- ループ並列化指示すると、逐次と結果が異なる
  （a[i-1]が更新されていない場合がある）

```
for (i=0; i<100; i++) {
  a[i] = a[ ind[i] ];
}
```

- ind[i]の内容により、ループ並列化できるかどうか決まる
- a[ind[i]]が参照時点で更新される可能性がない場合にループ並列化できる

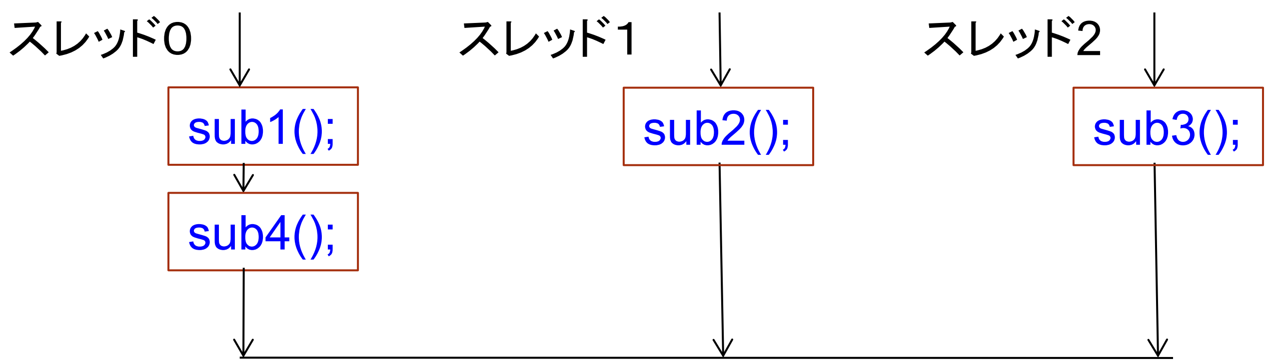
Sections構文

```
#pragma omp sections
{
#pragma omp section
   sub1();
#pragma omp section
   sub2();
#pragma omp section
   sub3();
#pragma omp section
   sub4();
}
```

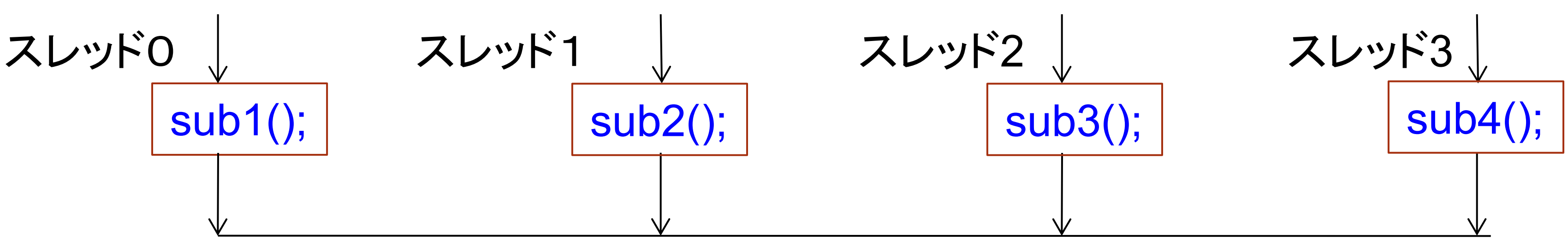
※Fortran言語の場合は

```
!$omp sections
～
!$omp end sections
```

●スレッド数が3の場合



●スレッド数が4の場合



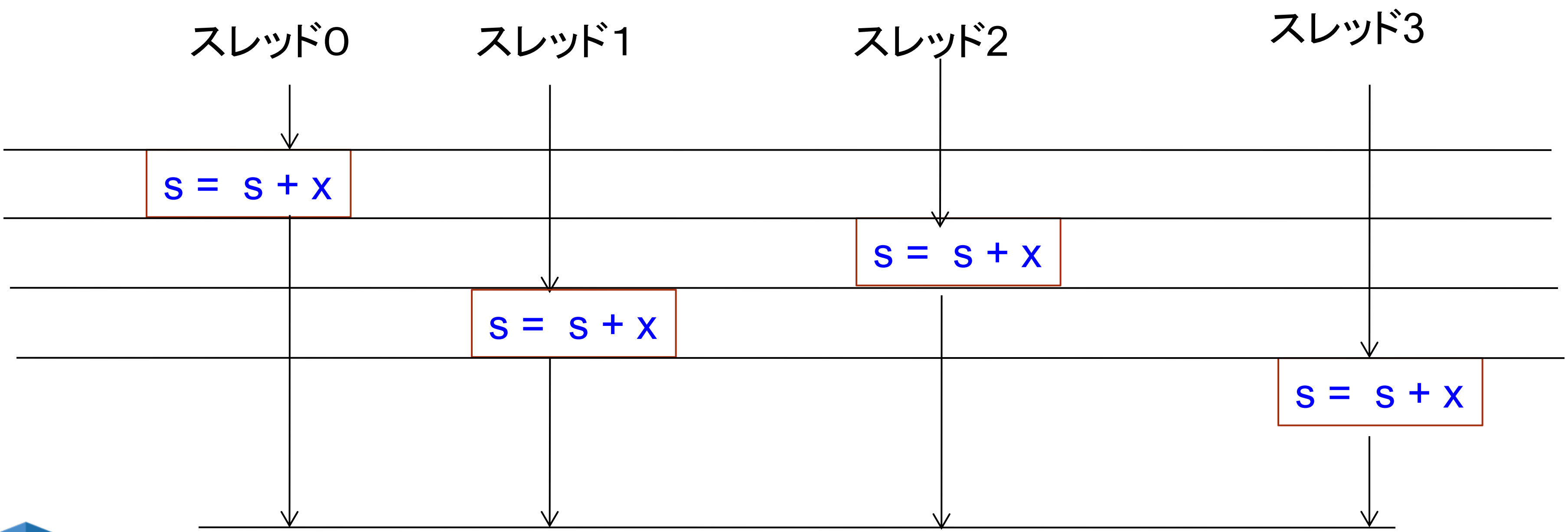
Critical補助指示文

```
#pragma omp critical
{
  s =  s +  x;
}
```

※Fortran言語の場合は

```
!$omp critical
～
!$omp end critical
```

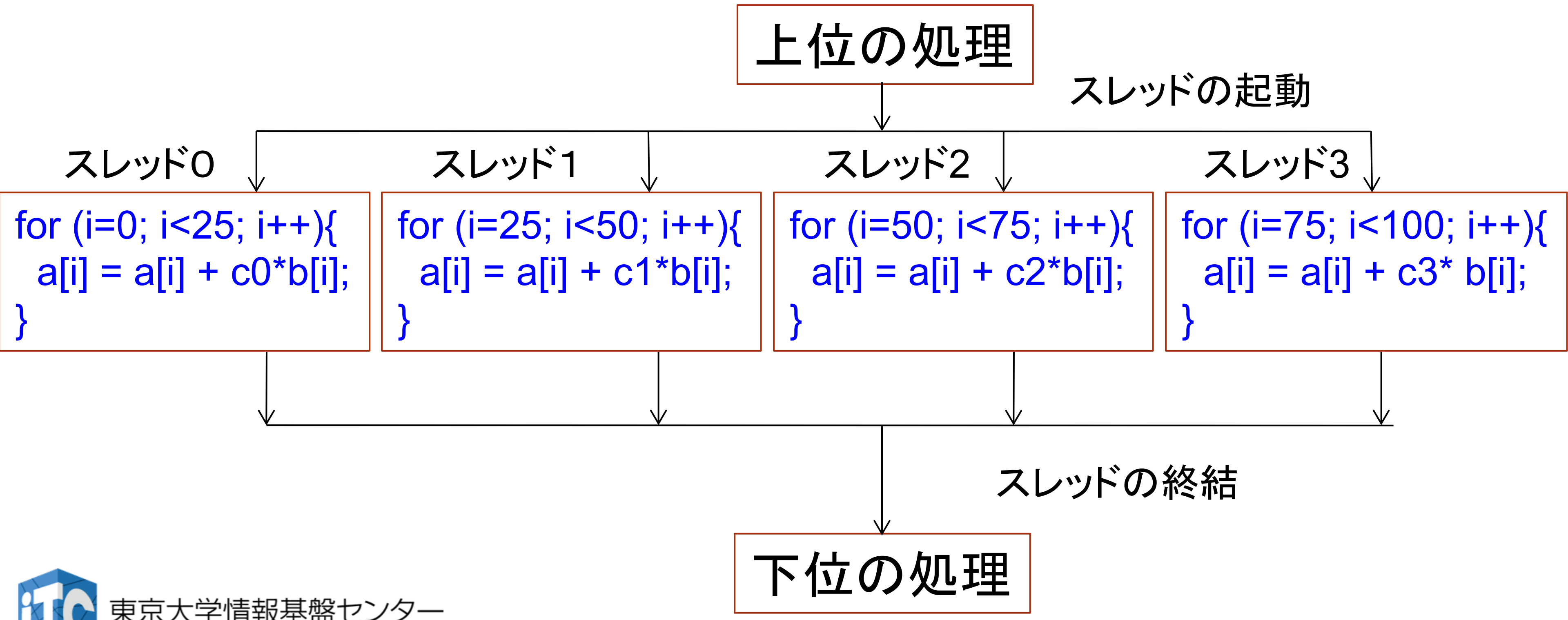
- ある瞬間には1つのスレッドしか実行しないことを保証



Private補助指示文

```
#pragma omp parallel for private(c)
for (i=0; i<100; i++){
  a[i] = a[i] + c * b[i];
}
```

※変数cが各スレッドで
別の変数を確保して実行
→高速化される



Private補助指示文の注意（C言語）

```
#pragma omp parallel for private( j )
for (i=0; i<100; i++) {
  for (j=0;  j<100; j++) {
    a[ i ] = a[ i ] + amat[ i ][ j ]* b[ j ];
}
```

•ループ変数 j が、各スレッドで別の変数を確保して実行される。
•private(j) がない場合、各スレッドで 共有変数 j のカウントを勝手に行ってしまうため、100回のループ実行にならない。
　→演算結果が逐次と異なり、エラーとなる。

Private補助指示文の注意（Fortran言語）

```
!$omp parallel do private( j )
do i=1, 100
   do j=1,  100
       a( i ) = a( i ) + amat( i , j ) * b( j )
   enddo
enddo
!$omp end parallel do
```

- ループ変数 j が、各スレッドで別の変数を確保して実行される。
- private(j) がない場合、各スレッドで 共有変数 j のカウントを勝手に行ってしまうため、100回のループ実行にならない。

  →演算結果が逐次と異なり、エラーとなる。

リダクション補助指示文（C言語）

- 内積値など、スレッド並列の結果を足しこみ、１つの結果を得たい場合に利用する
  - 上記の足しこみはスレッド毎に非同期になされる
  - reduction補助指示文が無いと、ddotは単なる共有変数になるため、並列実行で逐次の結果と合わなくなくなる

```
#pragma omp parallel for reduction(+:ddot )
for (i=1; i<=100; i++) {
    ddot += a[ i ] * b[ i ]
}
```

ddotの場所はスカラ変数のみ記載可能（配列は記載できません）

リダクション補助指示文（Fortran言語）

- 内積値など、スレッド並列の結果を足しこみ、１つの結果を得たい場合に利用する
  - 上記の足しこみはスレッド毎に非同期になされる
  - reduction補助指示文が無いと、ddotは共有変数になるため、並列実行で逐次の結果と合わなくなくなる

```
!$omp parallel do reduction(+:ddot )
do i=1, 100
    ddot = ddot + a(i) * b(i)
enddo
!$omp end parallel do
```

ddotの場所はスカラ変数のみ記載可能（配列は記載できません）

リダクション補助指示文の注意

- reduction補助指示文は、排他的に加算が行われるので、性能が悪い
  - 多くのスレッドを扱う場合、性能劣化が激しい
- 以下のように、ddot用の配列を確保して逐次で加算する方が高速な場合もある（ただし、問題サイズ、ハードウェア依存）

```
!$omp parallel do private ( i )
do j=0, p-1
  do i=istart( j ), iend( j )
     ddot_t( j ) = ddot_t( j ) + a(i) * b(i)
  enddo
enddo
!$omp end parallel do
ddot = 0.0d0
do j=0, p-1
    ddot = ddot + ddot_t( j )
enddo
```

スレッド数分のループを作成：最大ｐスレッド利用

各スレッドでアクセスするインデックス範囲を事前に

各スレッドで用いる、ローカルなddot用の配列ddot_t()を確保し、0に初期化してお

逐次で足しこみ

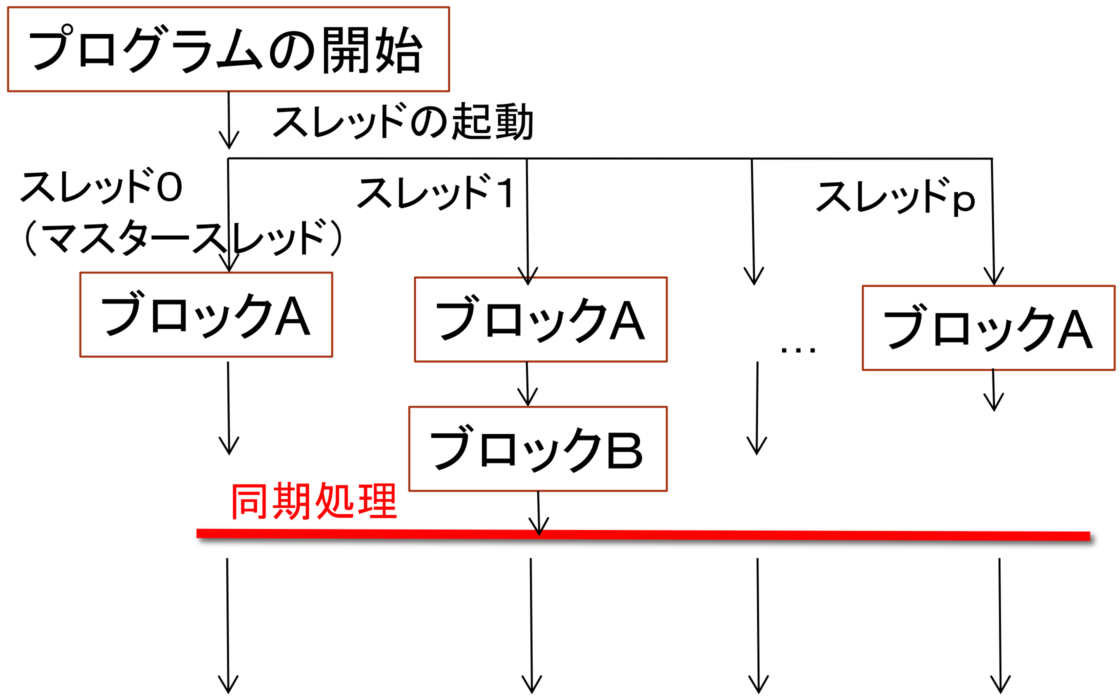
Single構文

※Fortran言語の場合は

```
!$omp single
～
!$omp end single
```

- Single補助指示文で指定されたブロックを、どれか１つのスレッドに割り当てる
- どのスレッドに割り当てられるかは予測できない
- nowait補助指示文を入れない限り、同期が入る

```
#pragma  omp  parallel for
{
   ブロックA
#pragma omp single
{    ブロックB  }
…
}
```



Master構文

- 使い方は、single補助指示文と同じ
- ただし、master補助指示文で指定した処理（先ほどの例の「ブロックB」の処理）は、
  必ずマスタースレッドに割り当てる
- 終了後の同期処理が入らない
  - そのため、場合により高速化される

Flush構文

- 物理メモリとの一貫性を取る
- Flush構文で指定されている変数のみ、その場所で一貫性を取る。それ以外の共有変数の値は、メモリ上の値との一貫性は無い。

（演算結果はレジスタ上に保存されるだけ。メモリに計算結果を書き込んでいない）

  - つまり、flush補助指定文を書かないと、スレッド間で同時に足しこんだ
    結果が、実行ごとに異なる。
  - barrier補助指定文、critical補助指定文の出入口、parallel構文の出口、for、sections、single構文の出口では、暗黙的にflushされている。
- Flushを使うと性能は悪くなる。できるだけ用いない。

#pragma omp flush (対象となる変数名の並び)

省略すると、
全ての変数が対象

Threadprivate構文

- スレッドごとにプライベート変数にするが、スレッド内で大域アクセスできる変数を宣言する。
- スレッドごとに異なる値をもつ大域変数の定義に向く。
  - たとえば、スレッドごとに異なるループの開始値と終了値の設定

```
#include <omp.h>
int myid, nthreds, istart, iend;
#pragma omp threadprivate(istart, iend)
…
void kernel() {
   int i;
   for (i=istart; i<iend; i++) {
      for (j=0; j<n; j++) {
         a[ i ] = a[ i ] + amat[ i ][ j ] * b[ j ];
      }
   }
}
…
```

```
…
void main() {
…
#pragma omp parallel private (myid,
nthreds, istart, iend)   {
  nthreds = omp_num_threds();
  myid = omp_get_thread_num();
  istart = myid * (n/nthreads);
  iend = (myid+1)*(n/nthreads);
  if (myid == (nthreads-1)) {
     nend = n;
  }
  kernel();
}
```

スレッド毎に異なる値を持
大域変数を、parallel構文中
で定義する

よく使うOpenMPの関数

最大スレッド数取得関数

- 最大スレッド数取得には、omp_get_num_threads()関数を利用する
- 型はinteger (Fortran言語)、int (C言語)

● Fortran90言語の例

```
use omp_lib
Integer nthreads

nthreads = omp_get_num_threads()
```

● C言語の例

```
#include <omp.h>
int nthreads;

nthreads = omp_get_num_threads();
```

自スレッド番号取得関数

- 自スレッド番号取得には、omp_get_thread_num()関数を利用する
- 型はinteger (Fortran言語)、int (C言語)

● Fortran90言語の例

```
use omp_lib
Integer myid

myid = omp_get_thread_num()
```

● C言語の例

```
#include <omp.h>
int myid;

myid = omp_get_thread_num();
```

時間計測関数

- 時間計測には、omp_get_wtime()関数を利用する
- 型はdouble precision (Fortran言語)、double (C言語)

● Fortran90言語の例

```
use omp_lib
double precision dts, dte

dts = omp_get_wtime()
  対象の処理
dte = omp_get_wtime()
print *, “Elapse time [sec.] =”,dte-dts
```

● C言語の例

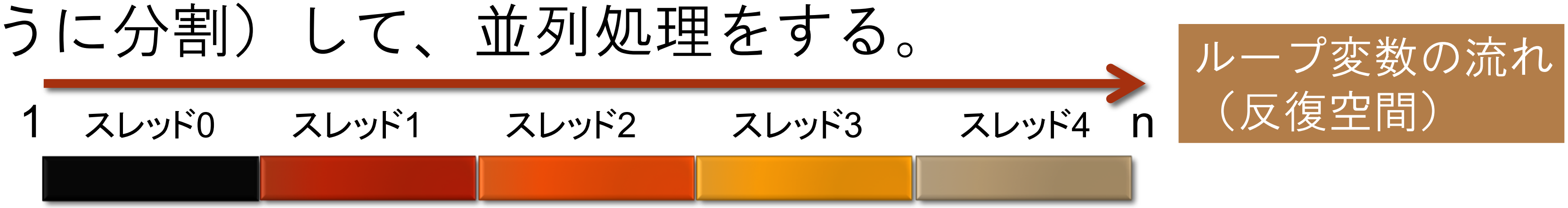
```
#include <omp.h>
double dts, dte;

dts = omp_get_wtime();
  対象の処理
dte = omp_get_wtime();
printf(“Elapse time [sec.] = %lf ¥n”,
  dte-dts);
```

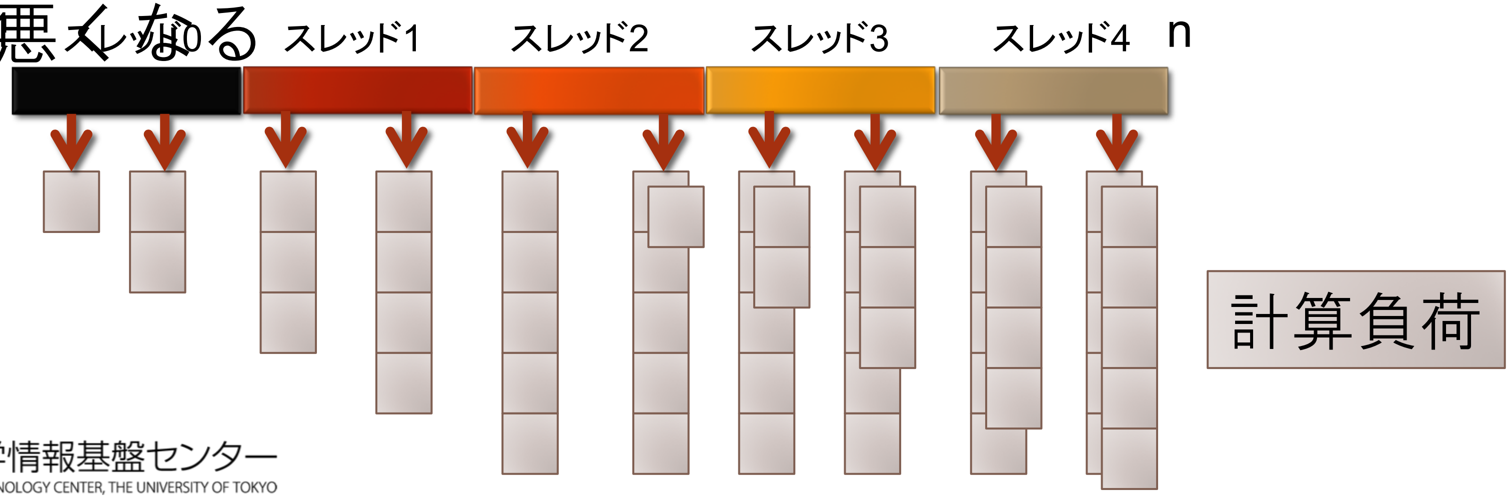
OpenMPにおけるスケジューリング

スケジューリングとは（その１）

- Parallel for/do構文では、対象ループの範囲（例えば１～ｎの長さ）を、単純にスレッド個数分に分割（連続するように分割）して、並列処理をする。

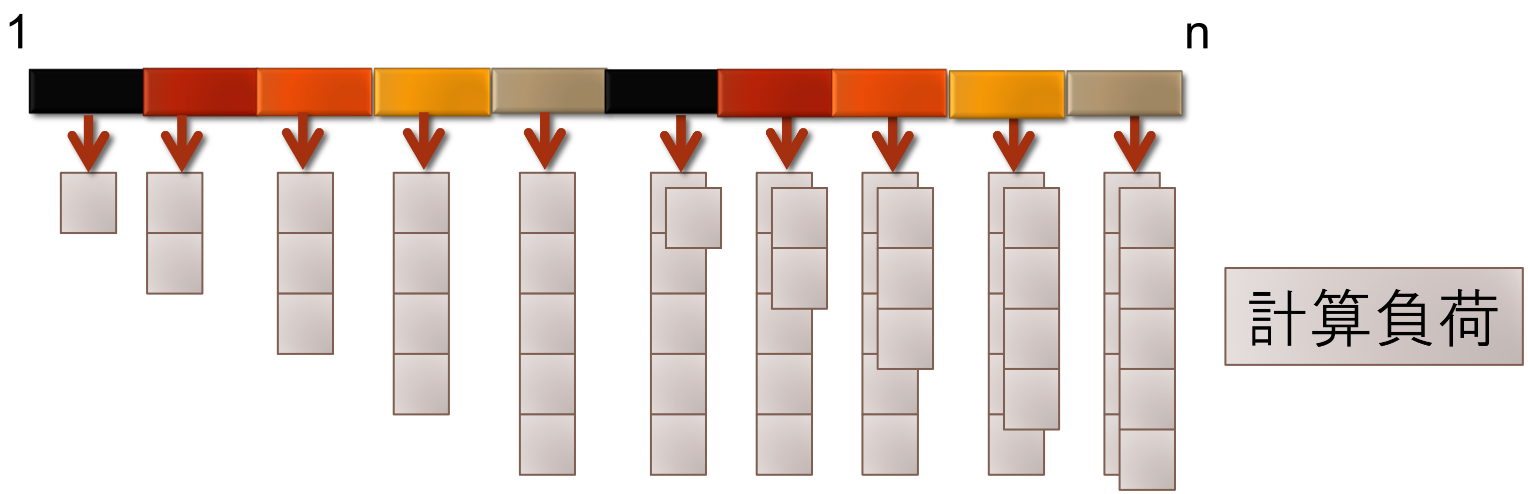


- このとき、各スレッドで担当したループに対する計算負荷が均等でないと、スレッド実行時の台数効果が悪くなる



スケジューリングとは（その２）

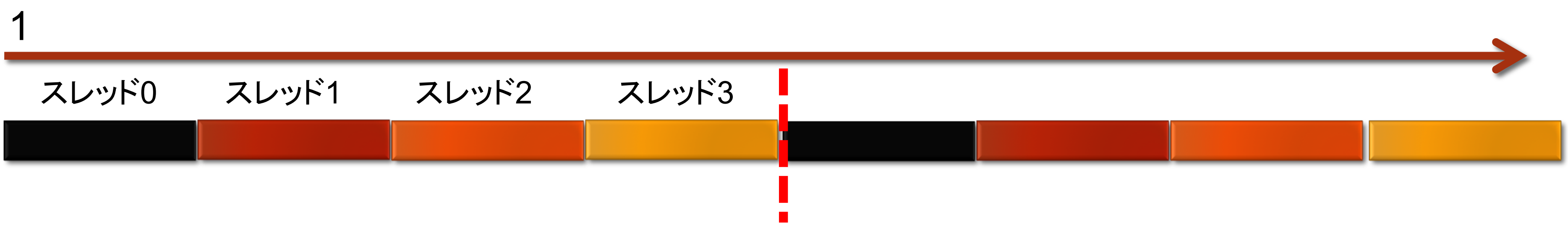
- 負荷分散を改善するには、割り当て間隔を短くし、かつ、循環するように割り当てればよい。



- 最適な、割り当て間隔（チャンクサイズとよぶ）は、計算機ハードウェアと、対象となる処理に依存する。
- 以上の割り当てを行う補助指示文が用意されている

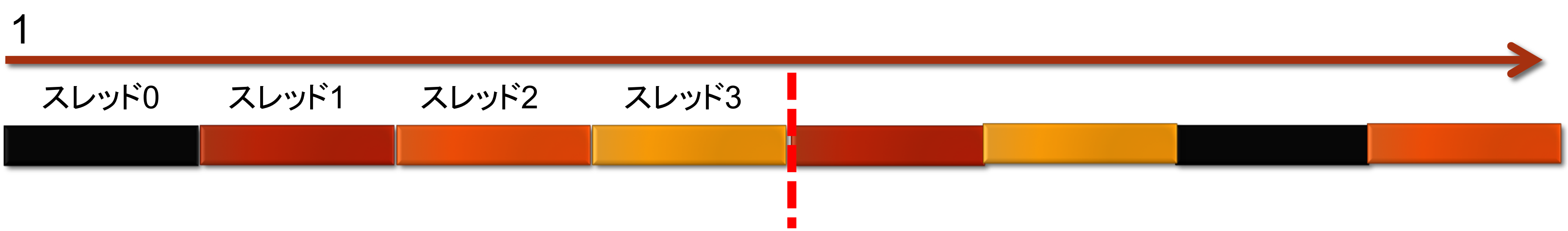
ループスケジューリングの補助指定文（その１）

- schedule (static, n)
  - ループ長をチャンクサイズで分割し、スレッド0番から順番に（スレッド０、スレッド１、・・・というように、ラウンドロビン方式と呼ぶ）、循環するように割り当てる。nにチャンクサイズを指定できる。
  - Schedule補助指定文を記載しないときのデフォルトは、staticで、かつチャンクサイズは、ループ長/スレッド数。



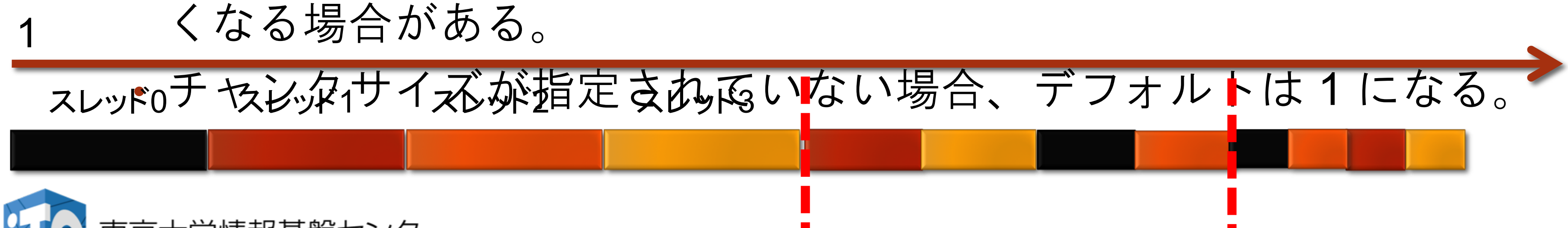
ループスケジューリングの補助指定文（その２）

- schedule(dynamic, n)
  - ループ長をチャンクサイズで分割し、処理が終了したスレッドから早い者勝ちで、処理を割り当てる。nにチャンクサイズを指定できる。



ループスケジューリングの補助指定文（その３）

- schedule(guided, n)
  - ループ長をチャンクサイズで分割し、徐々にチャンクサイズを小さくしながら、処理が終了したスレッドから早い者勝ちで、処理を割り当てる。nにチャンクサイズを指定できる。
  - チャンクサイズの指定が1の場合、残りの反復処理をスレッド数で割ったおおよその値が各チャンクのサイズになる。
  - チャンクサイズは 1 に向かって指数的に小さくなる。
  - チャンクサイズに 1 より大きい k を指定した場合、チャンク サイズは指数的に k まで小さくなるが、最後のチャンクは k より小さくなる場合がある。
  - チャンクサイズが指定されていない場合、デフォルトは 1 になる。



ループスケジューリングの補助指示文の使い方

- Fortran90言語の例

j-ループの反復回数が間接参照により決まるので、i-ループの計算負荷が均等であるか不明。
実行時にしか、計算負荷の状況がわからないため、dynamicスケジューリングを適用

```
!$omp parallel do private( j, k ) schedule(dynamic,10)
do i=1, n
  do j=indj(i), indj (i+1)-1
     y( i ) = amat( j ) * x( indx( j ) )
  enddo
enddo
!$omp end parallel do
```

- C言語の例

```
#pragma omp parallel for private( j, k )
schedule(dynamic,10)
for (i=0; i<n; i++) {
   for ( j=indj(i); j<indj (i+1); j++) {
      y[ i ] = amat[ j ] * x[ indx[ j ]];
   }
}
```

ループスケジューリングにおける プログラミング上の注意

- dynamic、guidedのチャンクサイズは性能に大きく影響
  - チャンクサイズが小さすぎると負荷バランスは良くなるが反面、処理待ちのオーバヘッドが大きくなる。
  - 一方、チャンクサイズが大きすぎと負荷バランスが悪くなる半面、処理待ちのオーバヘッドが小さくなる。
  - 上記の両者のトレードオフがある。
  - 実行時のチャンクサイズのチューニングが必須で、チューニングコストが増える。
- staticのみで高速実装ができる（場合がある）
  - dynamicなどの実行時スケジューリングは、システムのオーバーヘッドが入るが、staticはオーバーヘッドは（ほとんど）無い。
  - 事前に負荷分散が均衡となるループ範囲を調べた上で、staticスケジューリングを使うと、最も効率が良い可能性がある。
  - ただし、プログラミングのコストは増大する

Staticスケジューリングのみで負荷バランスを均衡化させる実装例

- 疎行列-ベクトル積へ適用した例（詳細は後述）

```
!$omp parallel do private(S,J_PTR,I)
    DO K=1,NUM_SMP
      DO I=KBORDER(K-1)+1,KBORDER(K)
        S=0.0D0
        DO J_PTR=IRP(I),IRP(I+1)-1
          S=S+VAL(J_PTR)*X(ICOL(J_PTR))
        END DO
        Y(I)=S
      END DO
    END DO
!$omp end parallel do
```

スレッド個数文のループ
（スレッドごとのループ担当範囲
を知るために必要）

事前に調べて設定しておいた、
負荷分散が均衡となる
スレッドごとのループ範囲
（各スレッドは、連続しているが、

実行前に、各スレッドが担当するループ範囲について、連続する割り当てで、かつ、それで負荷が均衡する問題に適用できる。
※実行時に負荷が動的に変わっていく場合は適用できない

OpenMPのプログラミング上の注意（全般）

OpenMPによるプログラミング上の注意点

- OpenMP並列化は、

  parallel構文を用いた単純なforループ並列化

  が主になることが多い。
  - 複雑なOpenMP並列化はプログラミングコストがかかるので、OpenMPのプログラミング上の利点が失われる
- parallel構文による並列化は

  private補助指示文の正しい使い方

  を理解しないと、バグが生じる！

Private補助指示文に関する注意（その１）

- OpenMPでは、対象となる直近のループ変数以外は、private変数で指定しない限り、全て共有変数になる。
  - デフォルトの変数は、スレッド間で個別に確保した変数でない
- ループ変数に関する共有変数の例

```
!$omp parallel do
do i=1, 100
  do j=1,  100
    tmp = b(i) + c(i)
    a( i ) = a( i ) + tmp
  enddo
enddo
!$omp end parallel do
```

宣言なしにプライベート変数として確保されるのは、このi-ループ変数のみ

このj-ループ変数は、private宣言なしでは共有変数になる
←スレッド間で早い者勝ちで更新 ←並列実行時にバグ

この変数tmpは、private宣言なしでは共有変数になる
←スレッド間で早い者勝ちで値が代入 ←並列実行時にバグ

Private補助指示文に関する注意（その２）

- Private補助指示文に記載する変数を減らすため、対象部分を関数化し、かつ、その関数の引数を増やすと、関数呼び出し時間が増加し、スレッド並列化の効果を相殺することがある
  - 呼び出し関数の引数が多い例

```
!$omp parallel do
do i=1, 100
   call foo(i,arg1,arg2,arg3,
    arg4,arg5, ….., arg100)
enddo
!$omp end parallel do
```

関数引数は自動的にプライベート変数になるため、private補助指示文に記載する変数を削減できる
←しかし、関数呼び出し時のオーバヘッドが増加する
←スレッド実行時においても、関数呼び出しのオーバーヘッドが無視できなくなり、台数効果が制限される
※解決法：大域変数で引き渡して引数を削減

Private補助指示文に関する注意のまとめ

- OpenMPでは、宣言せずに利用する変数は、すべて共有変数（shared variable）になる
- C言語の大域変数、Fortran90言語のcommon変数、module変数は、そのままでは共有変数になる
  - プライベート変数にしたい場合は、Threadprivate宣言が必要
- parallel構文で関数呼び出ししている場合、その関数内でローカルに宣言している変数も、共有変数になる
  - そのままでは、並列処理で正常動作しない
  - これを防ぐには、以下のコードの変更が必要
    - 上記のローカル変数を引数にした関数呼び出しを作る
    - 上記のローカル変数を大域変数にして、Threadprivate宣言する

Parallel構文の入れ子に関する注意（その１）

- Parallel構文は、do補助指示文で分離して記載できる
- １ループが対象の場合、分離するとdo補助指示文の場所でループごとにforkするコードを生成するコンパイラがあり、速度が低下する場合がある

```
!$omp parallel
!$omp do private(j,tmp)
do i=1, 100
  do j=1, 100
    tmp = b( j ) + c( j )
    a( i ) = a( i ) + tmp
  enddo
enddo
!$omp end do
!$omp end parallel
```

Parallel構文の対象が1ループならparallel doで指定

```
!$omp parallel do private(j,tmp)
do i=1, 100
  do j=1, 100
    tmp = b( j ) + c( j )
    a( i ) = a( i ) + tmp
  enddo
enddo
!$omp end parallel do
```

Parallel構文の入れ子に関する注意（その２）

- Parallel構文は、do補助指示文で分離して記載できる
- 複数ループの内側を並列化したい場合は、分離した方が高速になる
  - ただし、外側ループを並列化できる時はその方が性能が良い
  - 外側ループにデータ依存があり、並列化できない場合

```
do i=1, n
!$omp parallel do
  do j=1, n
    <並列化できる式>
  enddo
!$omp end parallel do
enddo
```

→

```
!$omp parallel
do i=1, n
!$omp do
  do j=1, n
    <並列化できる式>
  enddo
!$omp end do
enddo
!$omp end parallel
```

データ依存関係を壊しバグになる例

- 間接参照があるインデックスに対して加算する例
  - 間接参照のパターン、および、スレッド実行のタイミング次第で、逐次処理と結果が一致し、正常動作だと勘違いする場合がある
    - 理論的には間違っている
  - OpenMPの共有変数は、データ一貫性の保証はしない
    - データ一貫性の保証には、critical補助指定文などの指定が必要

● バグになるプログラム例

```
!$omp parallel do private( j )
do i=1, n
   j = indx( i )
   a( j ) = a( j ) + 1
enddo
!$omp end parallel do
```

→

```
!$omp parallel do private( j )
do i=1, n
   j = indx( i )
!$omp critical
   a( j ) = a( j ) + 1
!$omp end critical
enddo
!$omp end parallel do
```

OpenMPを用いた並列化の欠点（その１）

- OpenMPは単純なループを並列化することに向く
- 実用アプリケーションにおける複雑なループは、そのままではOpenMP化に向いていないことがある。

1. private補助指示文中に書かれる変数名の数が膨大になる
   - 外側ループからOpenMP並列化する場合、内部で使っている変数の数が多いことがある
   - private変数リストに変数を書き忘れても、コンパイラによるエラーは出ない。（並列化の責任はユーザにあるため）
   - 実行すると、タイミングに依存し計算結果が逐次と異なる。どこが間違っているかわからないので、デバックが大変になる。
   - 解決策：コンパイラによっては、最適化情報を出力することができる。その情報から、ちゃんとprivate化されているか確認する。

OpenMPを用いた並列化の欠点（その２）

2. 高スレッド実行時に性能が出ない場合のチューニングが困難
    1. 近年のハードウェアはメモリアクセスの性能が相対的に低い
    2. ループそのものにそれほど並列性がない（ループ長が短い）
- 解決するには、アルゴリズムの変更、実装の変更、が必要になり、OpenMPの利点である容易なプログラミングを損なう

3. 複雑なスレッドプログラミングには向かない
- 単純な数値計算のカーネルループを、parallel for構文で記載する方針で仕様が作られている
- 複雑な処理は、task文、またはPthreadなどのnativeなスレッドAPI、またはTBB, Cilkなどの他の並列化手法で書くほうがやりやすい

OpenMPによるSIMD化

OpenMP+SIMD

- OpenMP 4.0からサポート
- コンパイラが自動ベクトル化もしてくれるが，指示をした方が確実
  - 時としてコンパイラが誤った答えを出すので要確認
- コードの書き換えを必要とする場合も（しばしば）ある
- メモリのアラインメントが重要（前述）

OMP For・Do+SIMD

- C言語：

```
#pragma omp parallel for simd
for (i=0; i < num; i++) {
   sum = sum + a[i];
}
```

- Fortran:

```
!$omp parallel do simd
do i=1, num
   sum = sum + a(i)
!$omp end parallel
```

OMP SIMD単独

- C言語：

```
#pragma omp parallel for
for (i=0; i < num; i++) {
#pragma omp simd
  for (j=0; j < num; j++) {
    sum[i] = sum[i] + a[i][j];
  }
}
```

- Fortran:

```
!$omp parallel do
do i=1, num
!$omp simd
  do j=1, num
    sum(i) = sum(i) + a(j,i)
  enddo
!$omp end simd
enddo
!$omp end parallel
```

Declare SIMD

- 関数をまるごとSIMD化したい場合
  - その関数はOMP SIMDの中で呼ばれる場合にSIMD化される

```
#pragma omp declare simd notinbranch
float min (float a, float b)
{ return a < b ? a : b; }

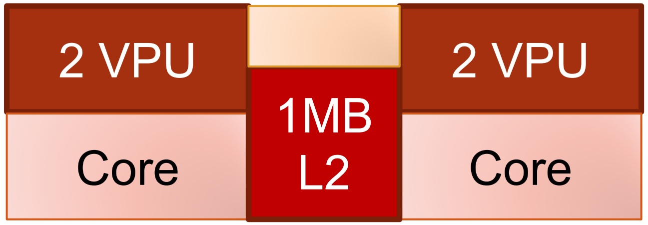
void minner (float *a, float *b, float *c) {
#pragma omp parallel for simd
for (i=0; i<N; i++)
  c[i] = min(a[i], b[i]);
}
```

KNLにおけるOpenMP実行の注意点

KNL(OFP)におけるHyperThreading

- 物理コア: 68コア
- HyperThreading(HT)有効: 1コアあたり 4 HT
  => 合計 272スレッド実行可能

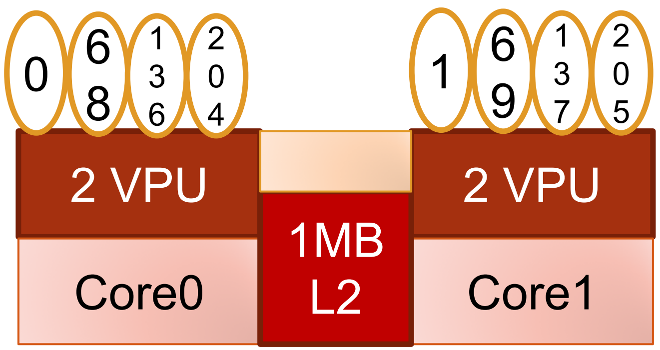
- AVX-512ユニットはあくまでも物理コアあたり2個
  - 2つのAVX-512ユニット(VPU)を使い尽くせれば 1 HTでいいはず
  - AVX-512ユニット1基しか使えていなければ 2 HT使った方がいい
  - メモリアクセスを隠蔽する必要があれば
    3~4 HTがいい場合も



1タイル=2コア

OFPにおけるスレッド割付け

- メニーコア => OS割り込み処理などによるジッタの影響を受けやすい
  - 遅いコアが一人いると全員が影響を受ける
  - 特にタイマ割り込みは 1ms毎にやってくる
- OFPでは、コア0のみタイマ割り込みを受け取るように設定（今後変わる可能性もあり）
  - コア0と、タイル内でL2キャッシュを直接共有しているコア1も影響を受ける
  - HTコアも当然影響を受ける
- コア番号 0,1, 68,69, 136,137, 204, 205 は実行から外した方がいい(場合が多い)
  - 数%以上性能が改善する
  - 具体的な指定方法は後述



1タイル=2コア

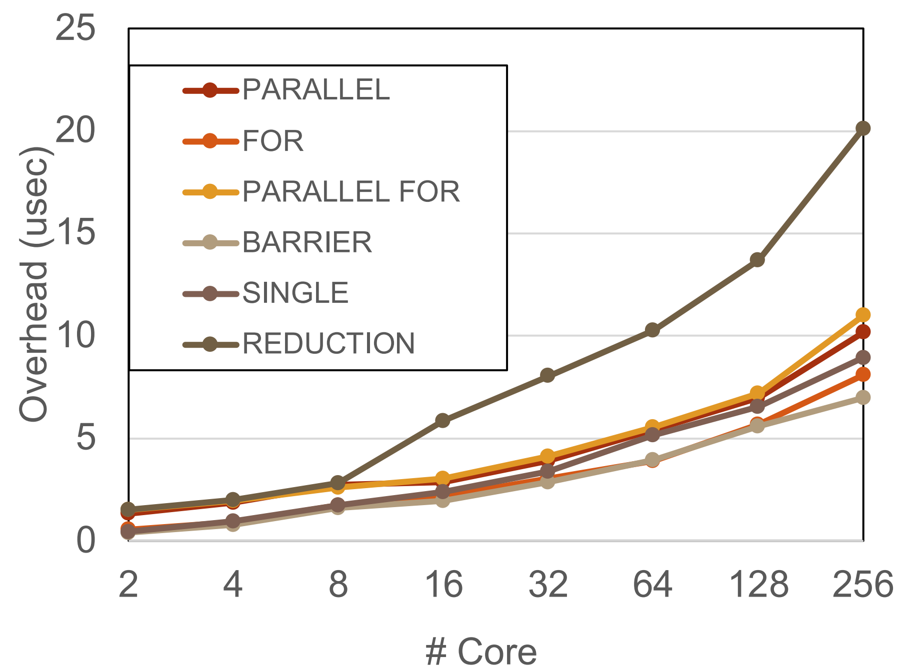
スレッド並列数と性能への影響(1)

Xeon Phiは、Xeonなど通常のマルチコアCPUに比べて

- コアのクロックが遅い
  - ReedbushのBroadwell (2.1GHz)に比べて 1.6倍程度
- メモリアクセスレイテンシも大きい
  - チップ内ネットワークが複雑

⇒OpenMPの指示文に対するオーバヘッドが大きくなる

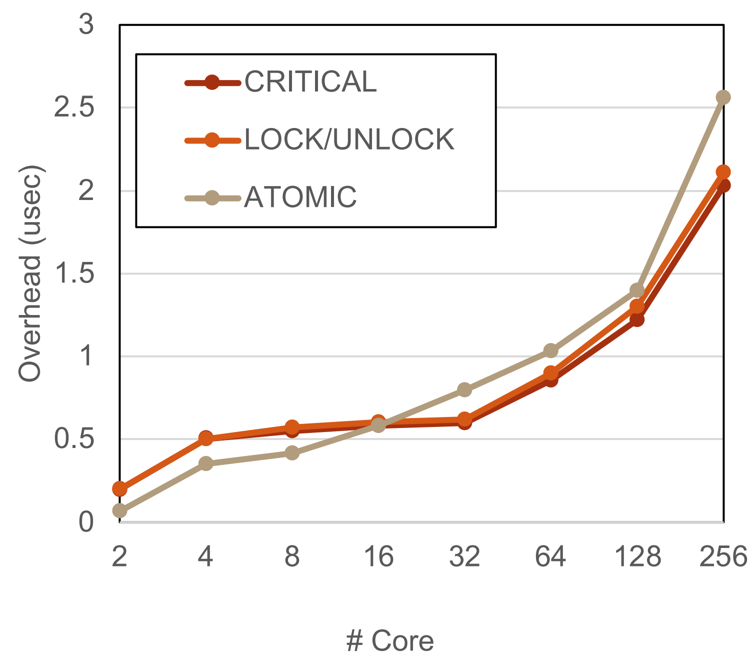
- 右図：各指示文のオーバヘッド (EPCC syncbench)
- 8コアから16コアで大きな影響があるように見える
  - 2コアは同一タイル、
  - 4~64コアは物理コアのみ、
  - 128~256コアはHT使用



スレッド並列数と性能への影響(2)

- 右図：各指示文(Atomic系)のオーバヘッド (EPCC syncbench)
  - Xeon Broadwell 32コアでは Atomic: 0.1us以下
    Critical, Lock/Unlock: 0.4us以下

=> メモリアクセスレイテンシの
影響



対策

- なるべくAtomic処理や
  バリア処理を減らす
  - Parallel for/doなど、様々な指示文にも暗にバリア処理は含まれている
- 最大限大きくParallel regionを取る
  - 毎回Parallel for/doするのではなく、Parallel の中でfor/doだけを繰り返す
- MPI+OpenMPハイブリッド実行により、スレッド並列数を下げる
  - アプリケーションに依存する
  - あとで詳しく

KNL単体(1ノード)における実行

コンパイル

Intel Compiler 2017 update 4 を想定 (OFPのデフォルト)
- Cコンパイラ: icc
- Fortranコンパイラ: ifort

必須オプション
- -xMIC-AVX512
  - KNL向けコード生成
- -O3
  - 最適化レベル
- -qopenmp
  - OpenMP有効
- -align array64byte
  - (Fortranのみ）配列を64byteにアライン（前述）

試した方がいい
- -qopt-streaming-stores= always / never / auto
  - always: キャッシュに載せない命令を使う、キャッシュの生存時間が短い場合に有効
  - neverはその逆
  - デフォルトはauto
  - ループの各配列ごとにも制御可能(以下はalways相当)
    - #pragma vector nontemporal (in C/C++)
    - !DIR$ vector nontemporal (in F)
- -qopt-prefetch={0~5}
  - プリフェッチをどのくらい頑張るか

実行時の指定

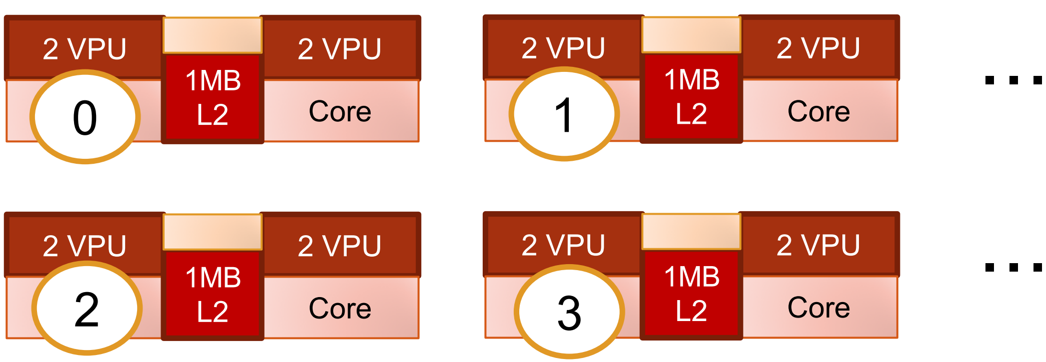
- 68コアのうち、64コアを使うことを想定
  - ジッタを防ぐための設定、コア 0,1, 68,69, 136,137, 204, 205を排除

1. 1タイルあたり1コアのみ、32コア使用
   - キャッシュ利用効率を重視する場合
2. 1コア1HT、64コア使用
3. 1コア2HT、128コア使用
4. 1コア3HT、192コア使用
5. 1コア4HT、256コア使用

32コア使用する場合

- タイルあたり1スレッド
  - キャッシュを十分に使えるが実際バンド幅を出すには足りない
- 明示的にプロセス割付を指定する（しかない）

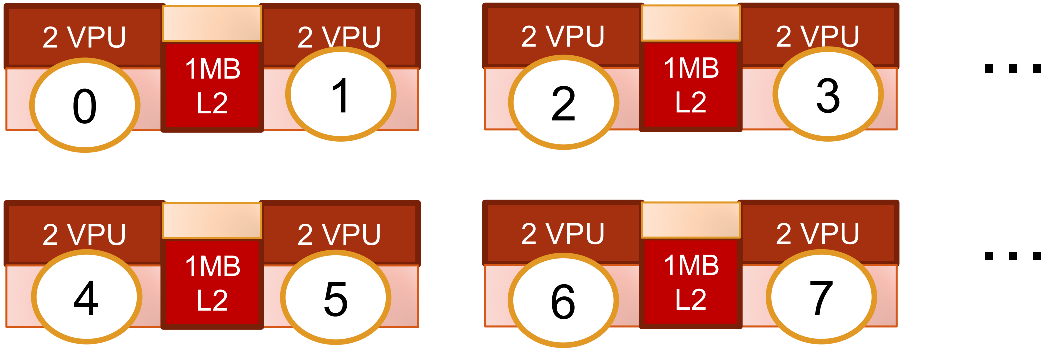
```
export OMP_NUM_THREADS=32
export KMP_AFFINITY=proclist=[2,4,6,8,10,12,14,16,18,20,
       22,24,26,28,30,32,34,36,38,40,42,44,46,48,50,52,54,
       56,58,60,62,64,66],explicit,verbose
```



64コア使用する場合

- コアあたり1スレッド
- KMP_HW_SUBSET環境変数が使える
  - 物理コア数c@オフセット,コアあたりスレッド数t
  - 64c@2,1t: 64コア、コア番号2から使用、コアあたり1スレッド

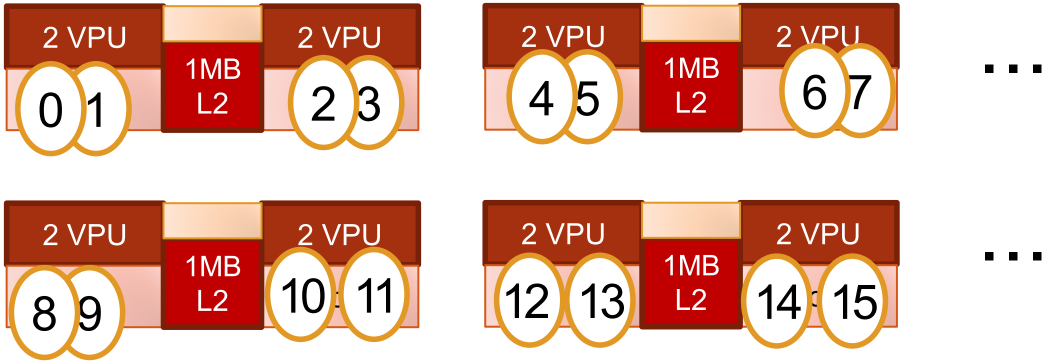
```
export OMP_NUM_THREADS=64
export KMP_HW_SUBSET=64c@2,1t
export KMP_AFFINITY=scatter,verbose
```



128コア使用する場合

- コアあたり2スレッド
- KMP_HW_SUBSET環境変数：64c@2,2t
- KMP_AFFINITYにcompactを指定
  - scatterにすると、同一コアにスレッド0と64、スレッド1と65…のようになり、キャッシュが非効率

```
export OMP_NUM_THREADS=128
export KMP_HW_SUBSET=64c@2,2t
export KMP_AFFINITY=compact,verbose
```



192, 256コア使用する場合

192コア

- コアあたり3スレッド

```
export OMP_NUM_THREADS=192
export KMP_HW_SUBSET=64c@2,
       3t
export KMP_AFFINITY=compact,
       verbose
```

256コア

- コアあたり4スレッド

```
export OMP_NUM_THREADS=256
export KMP_HW_SUBSET=64c@2,
       4t
export KMP_AFFINITY=compact,
       verbose
```

コア割り当ての確認

最初は必ず確認すること
（想定通りになっていなくて悲しい思いをする）

- KMP_AFFINITY環境変数にverboseを指定
- 標準エラー出力（e+数字のファイル）にレポート出力
- 見るべきは以下のような出力（最後の方）
  - 32コア実行の例: スレッド0がコア2, スレッド1がコア4, スレッド2がコア6…に割り当て

```
OMP: Info #242: KMP_AFFINITY: pid X tid XX thread 0 bound to OS proc set {2}
OMP: Info #242: KMP_AFFINITY: pid X tid XY thread 1 bound to OS proc set {4}
OMP: Info #242: KMP_AFFINITY: pid X tid XZ thread 2 bound to OS proc set {6}
…
```

ベクトル化、スレッド化、実行効率の確認

- コンパイル時に最適化レポートを確認
  - -qopt-report=5
  - ファイル名.optrpt を確認
- ツールを利用する（後述）
  - Intel Advisor XE
    - ベクトル化、スレッド並列化のアドバイス
  - Intel VTune XE
    - プロファイリング

演習

KNL/OFPにおける MPI+OpenMPハイブリッド

MPI+OpenMPハイブリッド実行の意義

- MPIランク数の増加に伴い、消費メモリもオーバヘッドも増加
- 通常は以下のような組み合わせが良いとされて来た
  - ノード内：OpenMP
  - ノード間：MPI
- またReedbushのように2ソケットでNUMAを構成していれば
  - ソケット内: OpenMP
  - ソケット間+ノード間: MPI
- フラットMPIも可能だがよろしくない

メニーコアクラスタではそう単純でもない

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

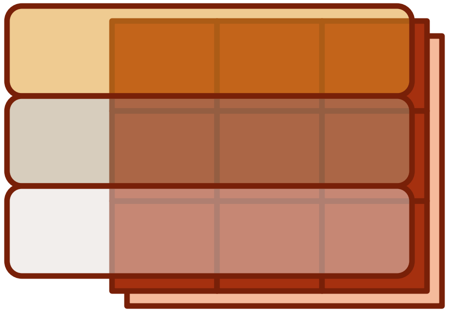
- 郵便物の郵送に同じ
- 郵送に必要な情報：
  1. 自分の住所、送り先の住所
  2. 中に入っているものはどこにあるか
  3. 中に入っているものの分類
  4. 中に入っているものの量
  5. （荷物を複数同時に送る場合の）認識方法（タグ）
- MPIでは：
  1. 自分の認識ID、および、送り先の認識ID
  2. データ格納先のアドレス
  3. データ型
  4. データ量
  5. タグ番号

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_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）

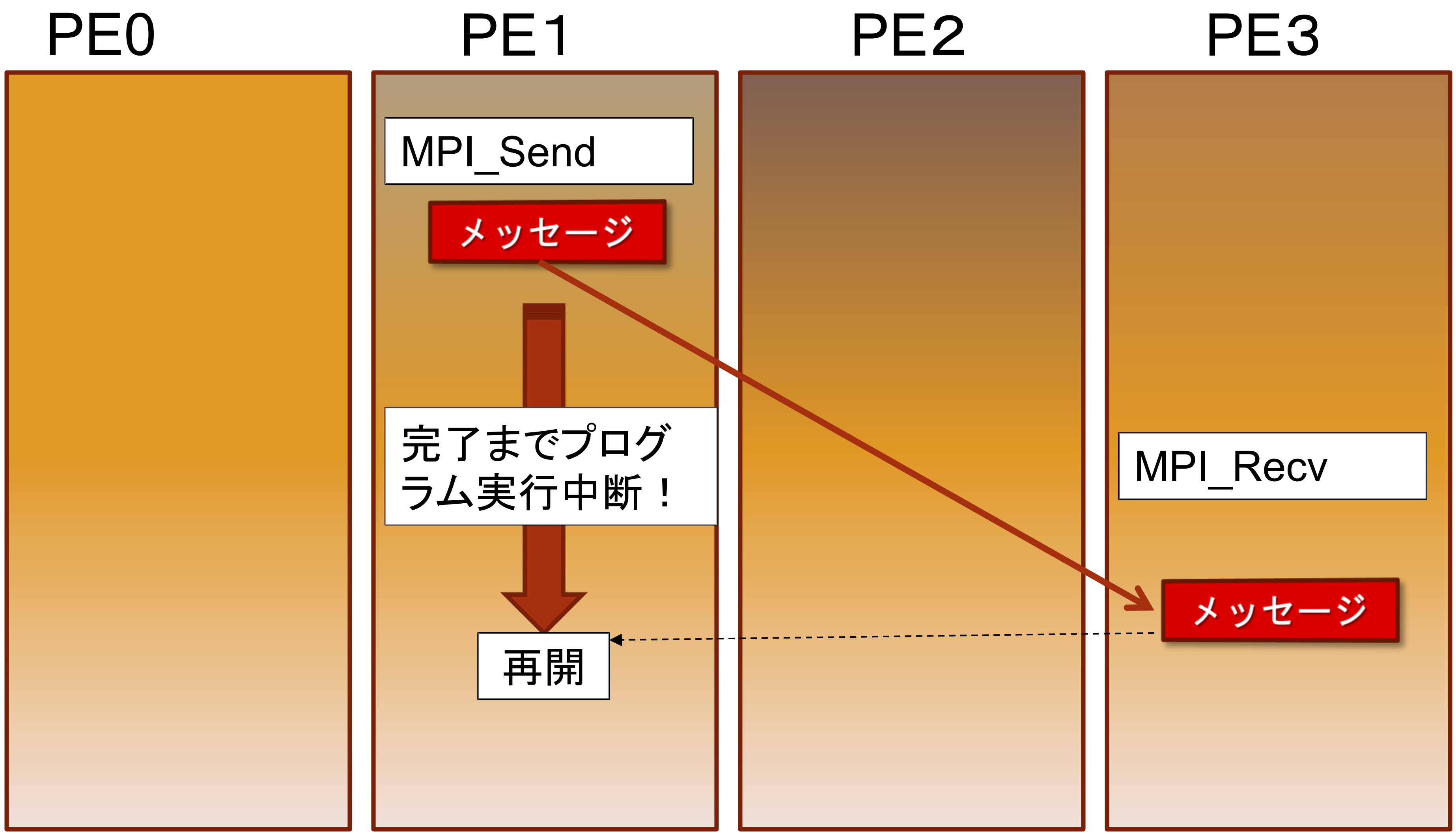
- itag： 整数型。受信したいメッセージに付いているタグの値を指定。
  - 任意のタグ値のメッセージを受信したいときは、MPI_ANY_TAG を指定。
- icomm： 整数型。PE集団を認識する番号であるコミュニケータを指定。
  - 通常ではMPI_COMM_WORLD を指定すればよい。
- istatus： MPI_Status型（整数型の配列）。受信状況に関する情報が入る。かならず専用の型宣言をした配列を確保すること。
  - 要素数がMPI_STATUS_SIZEの整数配列が宣言される。
  - 受信したメッセージの送信元のランクが istatus[MPI_SOURCE]、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

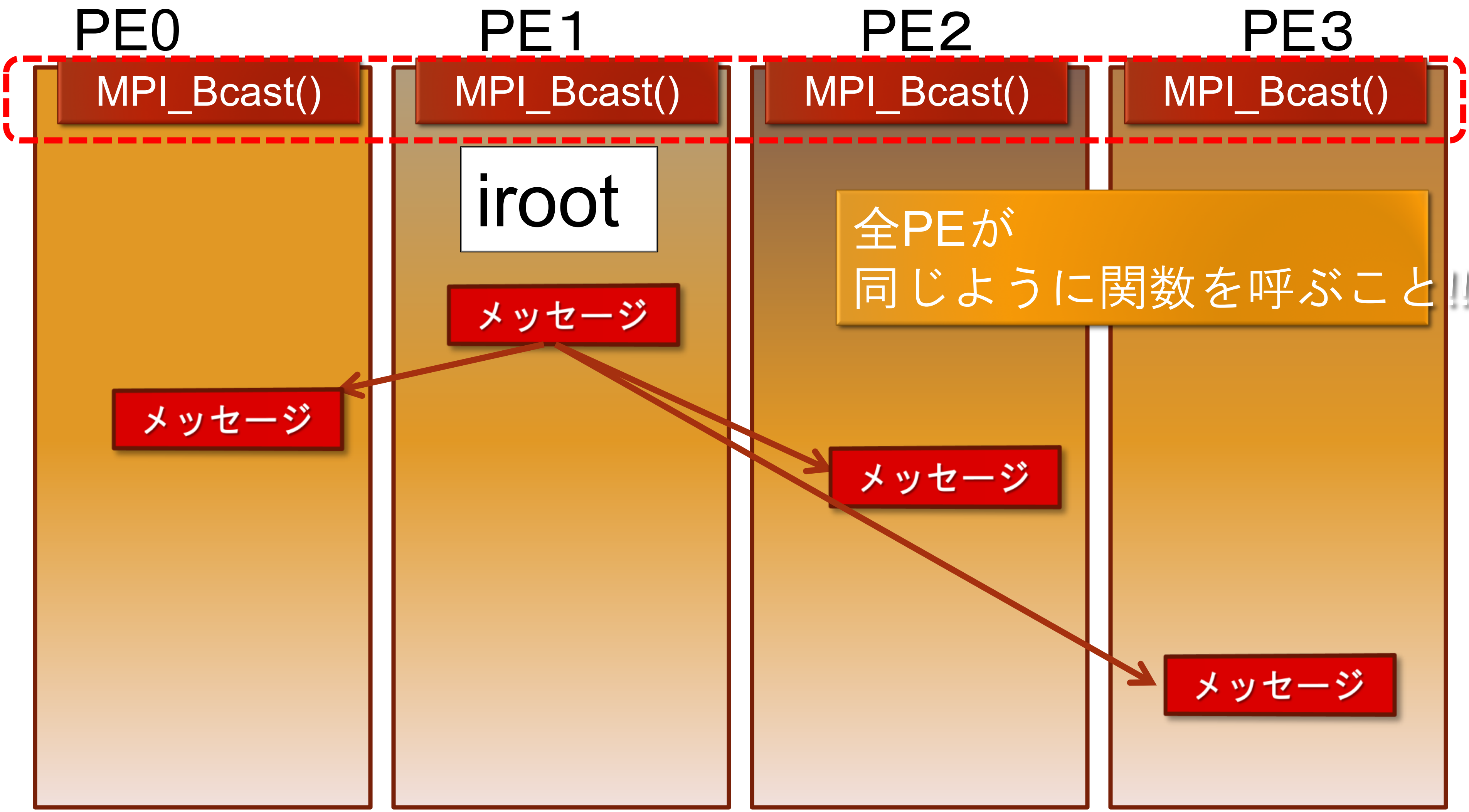


基礎的なMPI関数―MPI_Bcast

- 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）

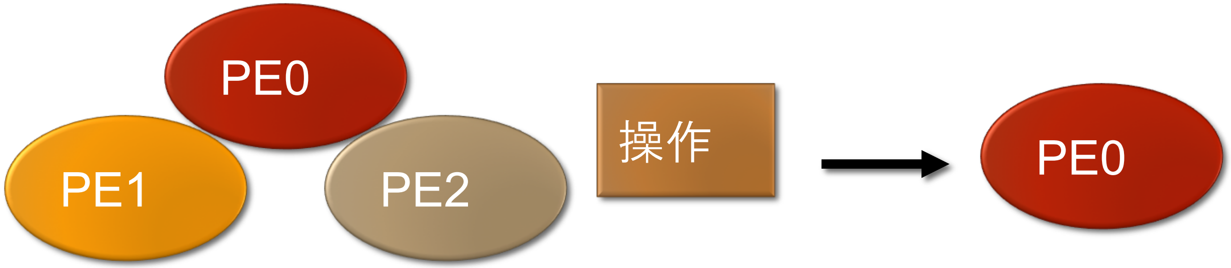


リダクション演算

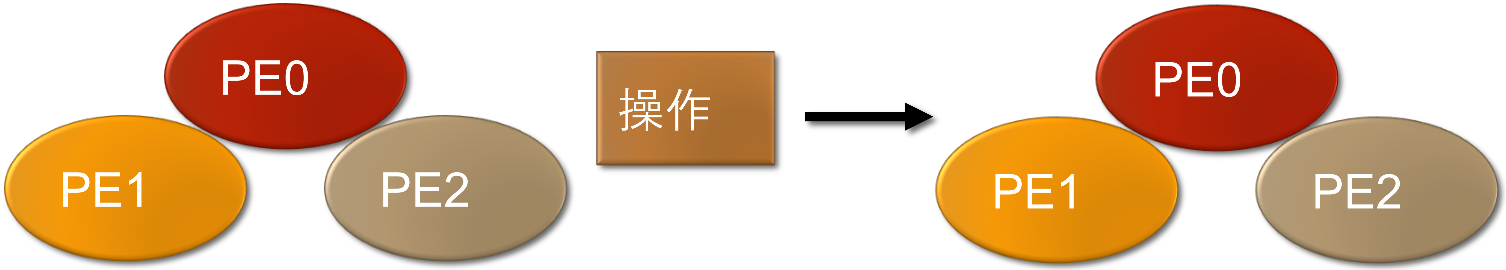
- ＜操作＞によって＜次元＞を減少（リダクション）させる処理
  - 例：　内積演算
    ベクトル（n 次元空間）　→　スカラ（1 次元空間）
- リダクション演算は、通信と計算を必要とする
  - 集団通信演算（collective communication operation）
    と呼ばれる
- 演算結果の持ち方の違いで、２種のインタフェースが存在する

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



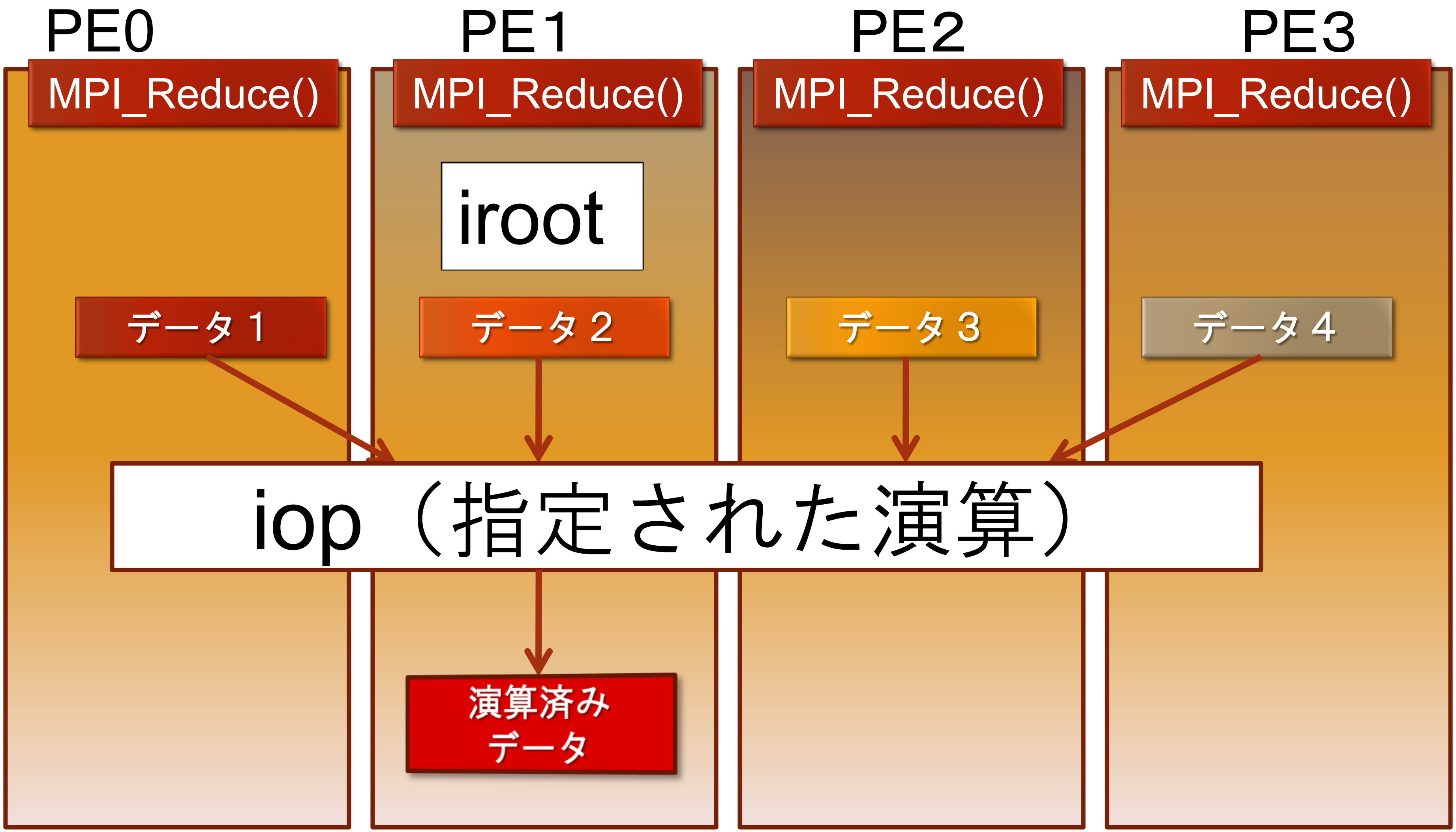
基礎的なMPI関数—MPI_Reduce

- 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_2DOUBLE_PRECISION(倍精度型) 、を指定する。

基礎的なMPI関数—MPI_Reduce

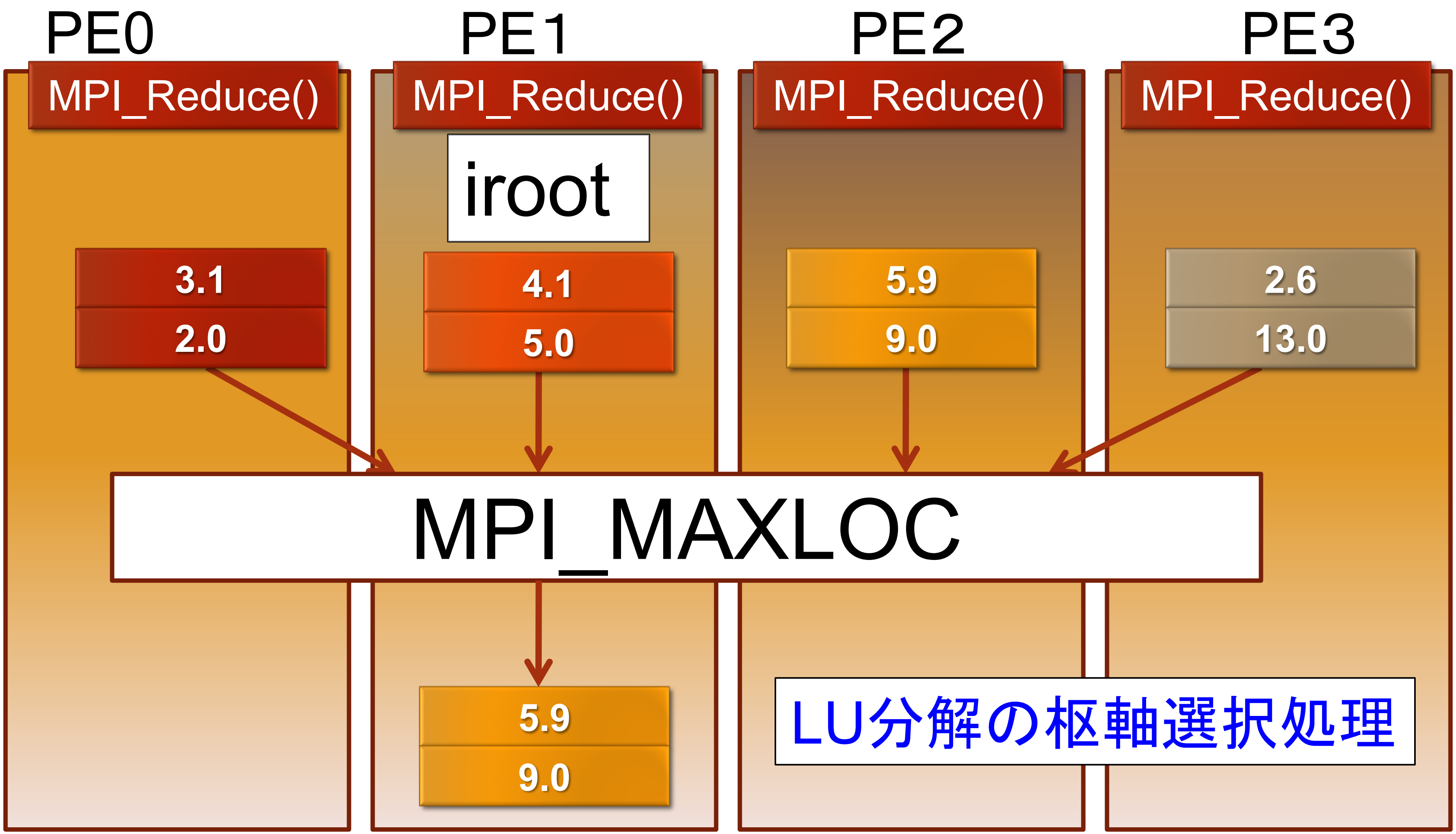
- iop : 整数型。演算の種類を指定する。
  - MPI_SUM (総和)、 MPI_PROD (積)、 MPI_MAX (最大)、MPI_MIN (最小)、 MPI_MAXLOC (最大とその位置)、 MPI_MINLOC (最小とその位置) など。
- iroot : 整数型。結果を受け取るPEのicomm 内でのランクを指定する。全てのicomm 内のPEで同じ値を指定する必要がある。
- icomm : 整数型。PE集団を認識する番号であるコミュニケータを指定する。
- ierr : 整数型。 エラーコードが入る。

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）



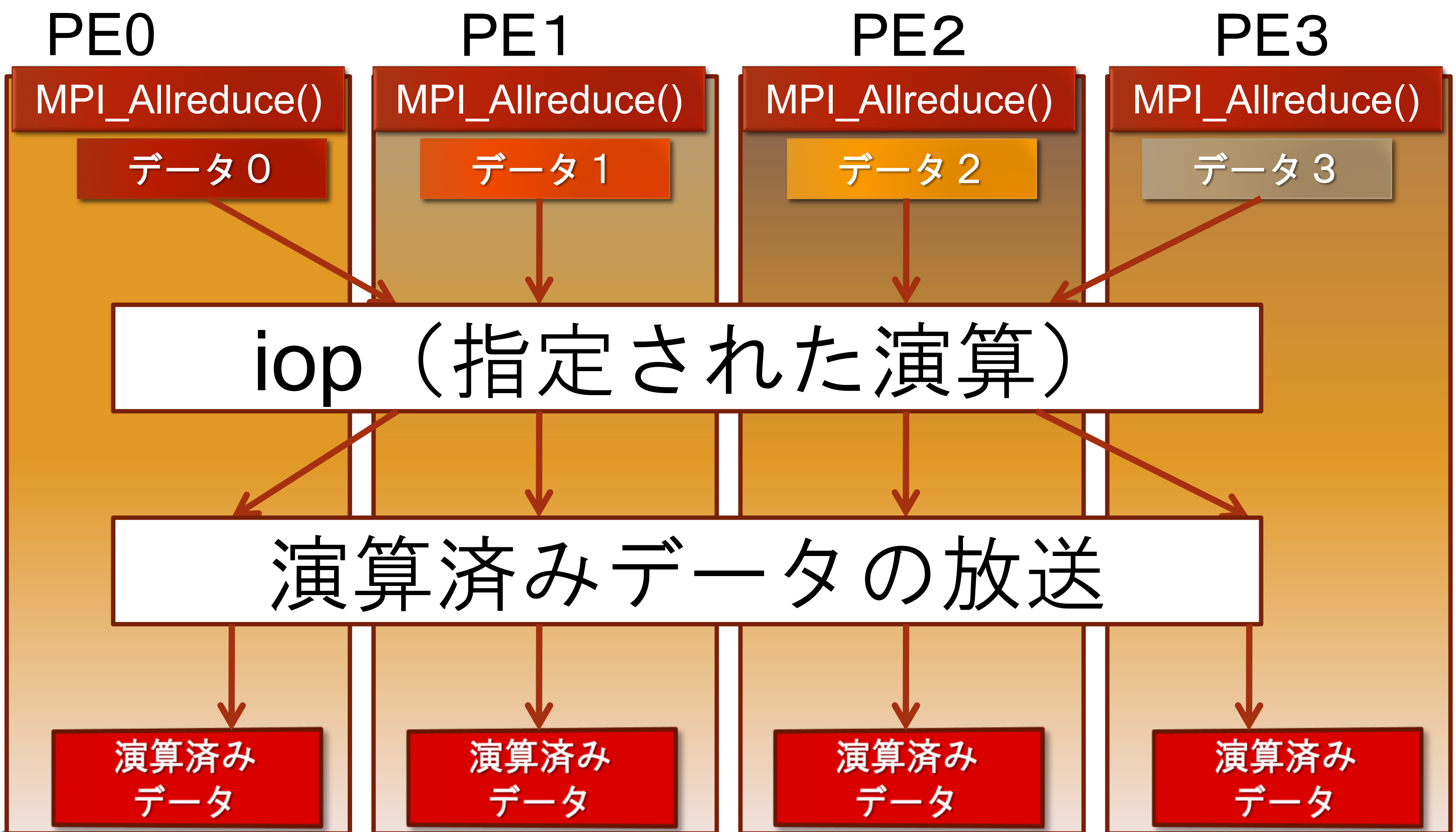
基礎的なMPI関数—MPI_Allreduce

- 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。

基礎的なMPI関数—MPI_Gather

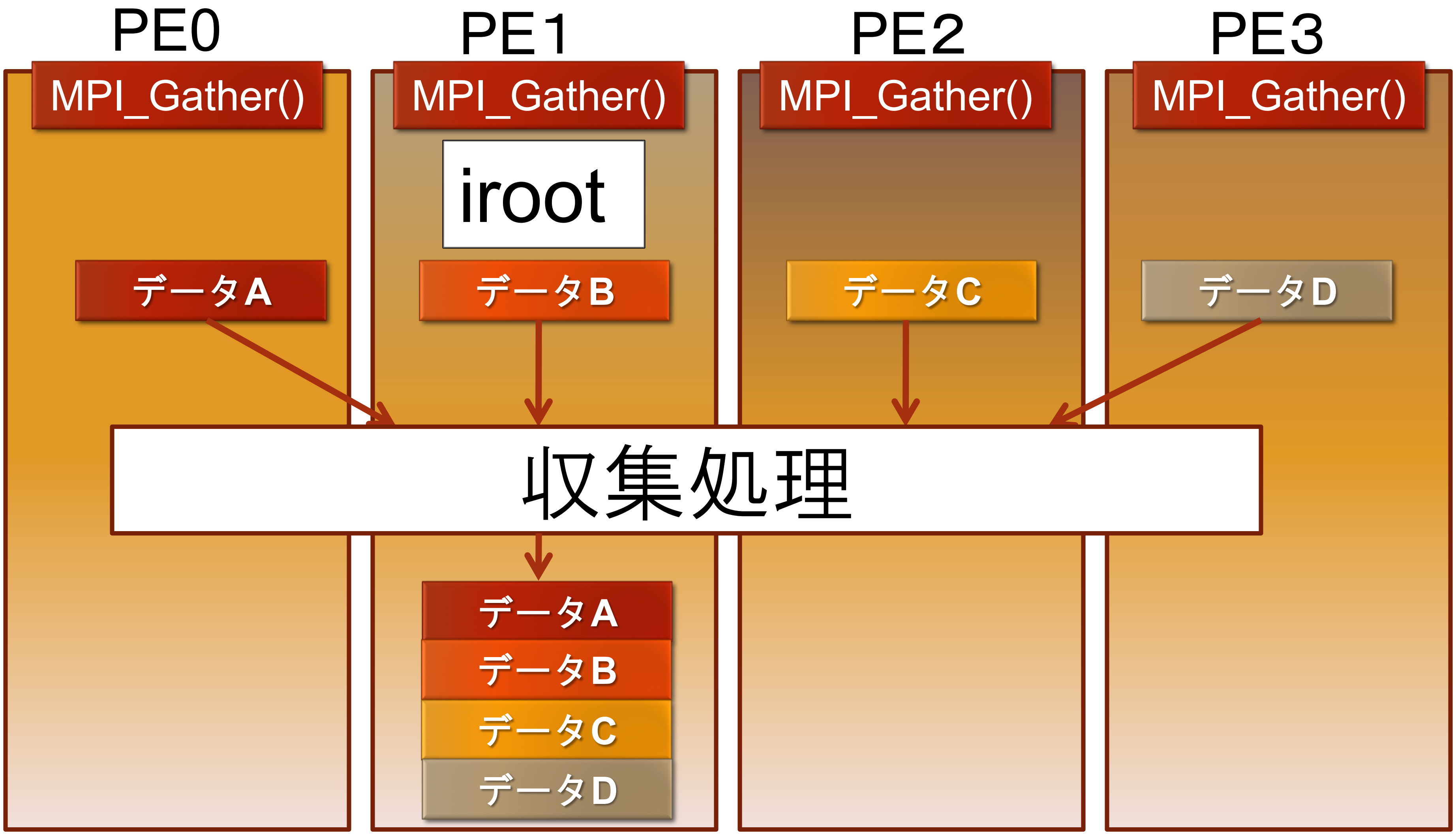
- ierr = MPI_Gather (sendbuf, isendcount, isendtype,
  recvbuf, irecvcount, irecvtype, iroot, icomm);

- 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

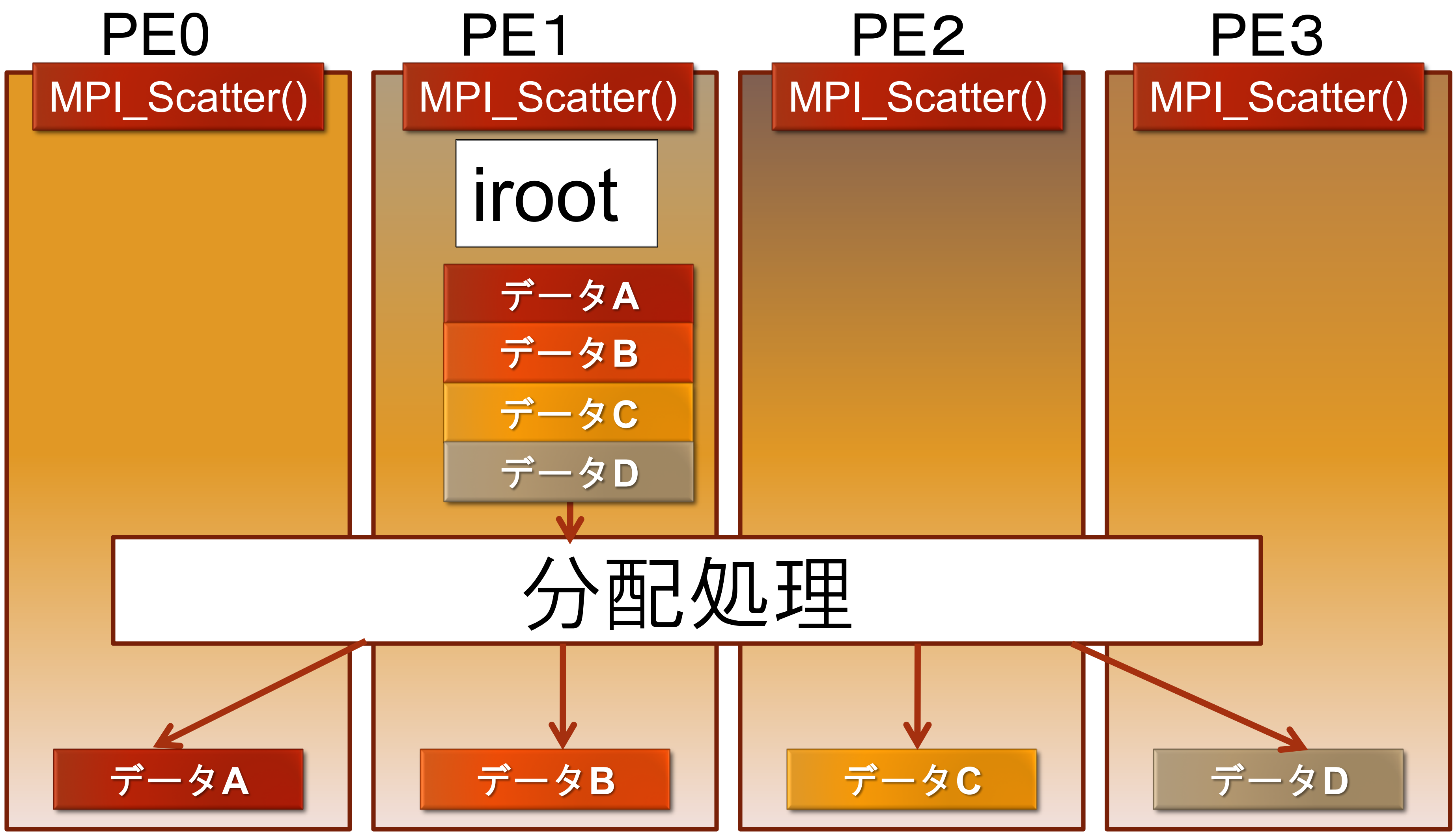
- ierr = MPI_Scatter (sendbuf, isendcount, isendtype, recvbuf, irecvcount, irecvtype, iroot, icomm);

- 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。

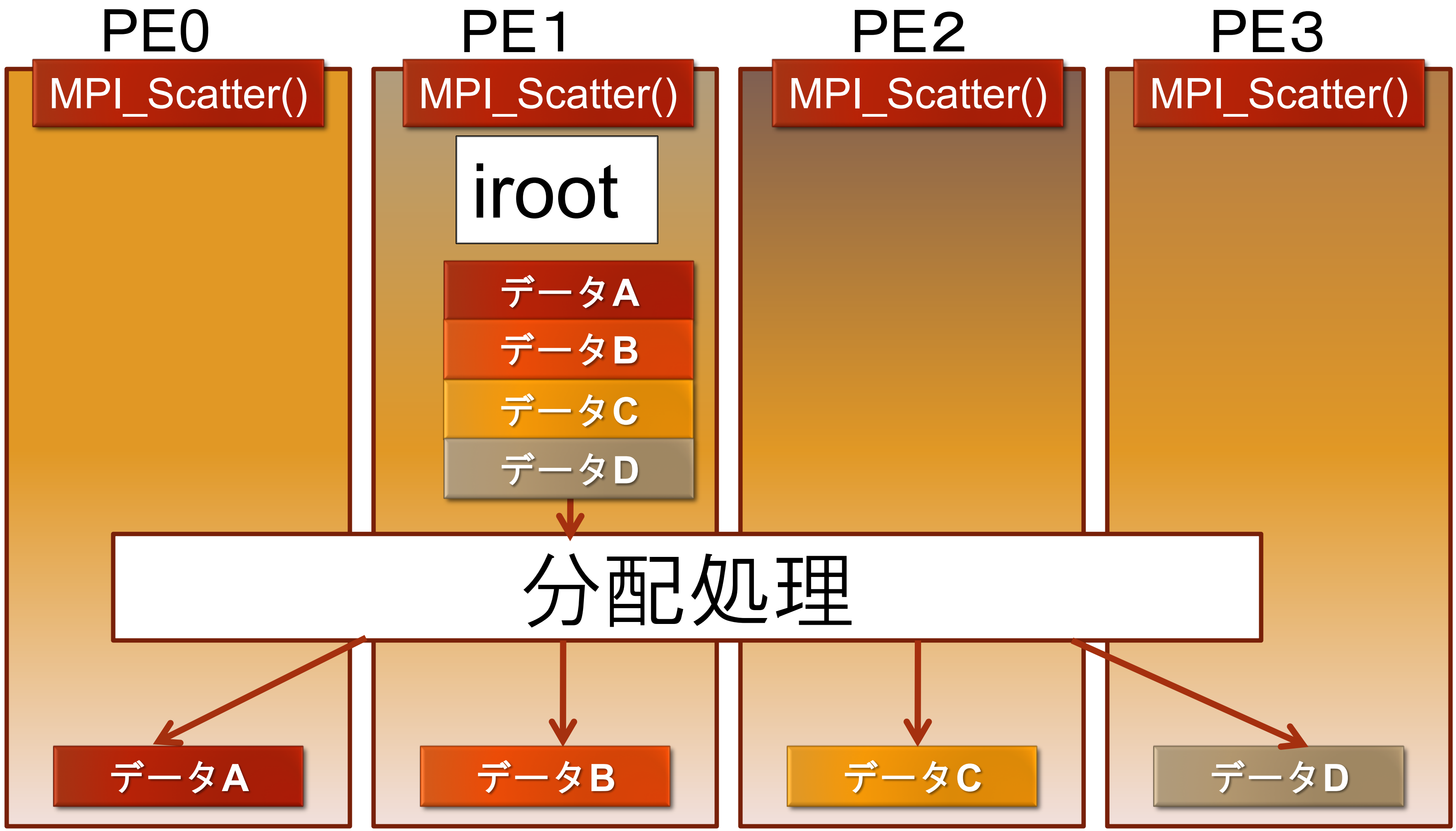
基礎的なMPI関数—MPI_Scatter

- 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）

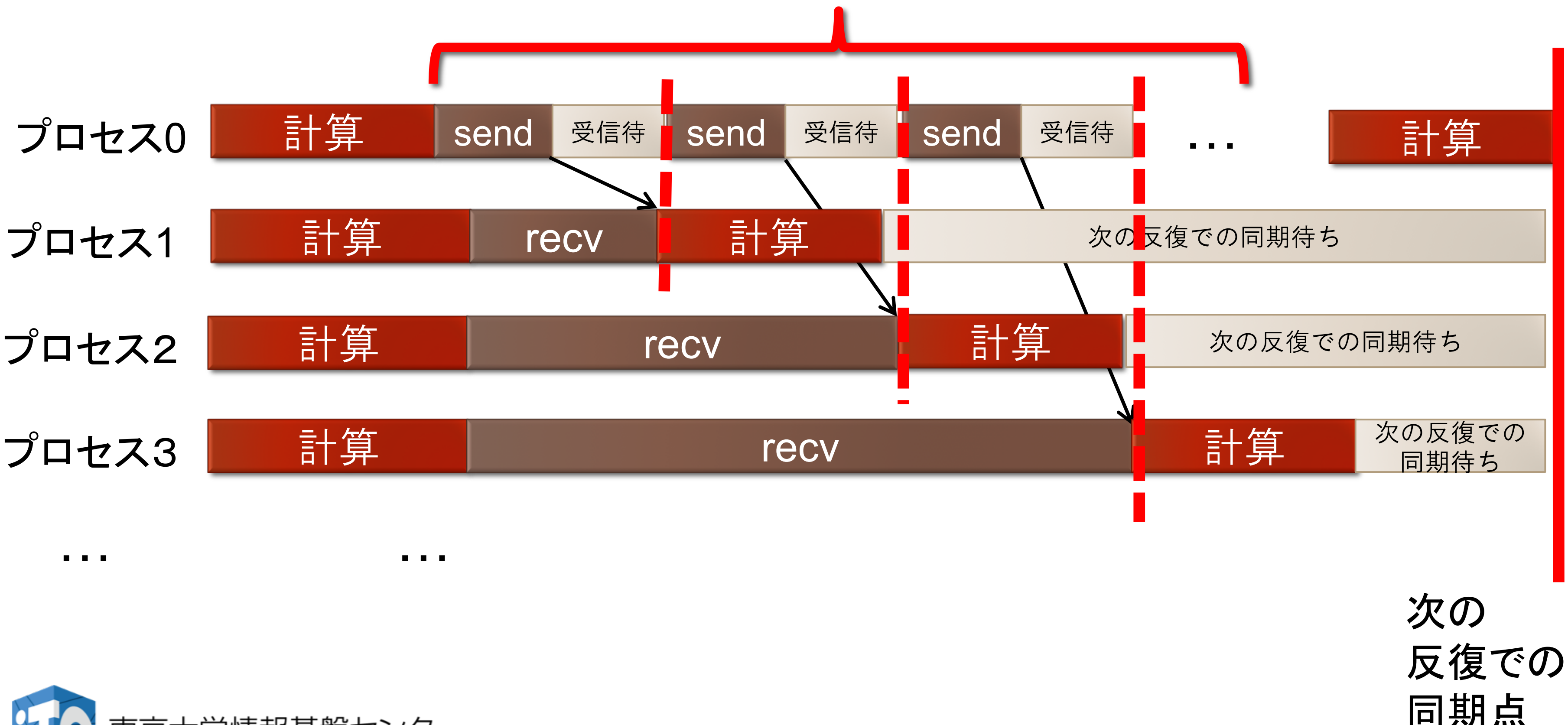


ブロッキング、ノンブロッキング

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。

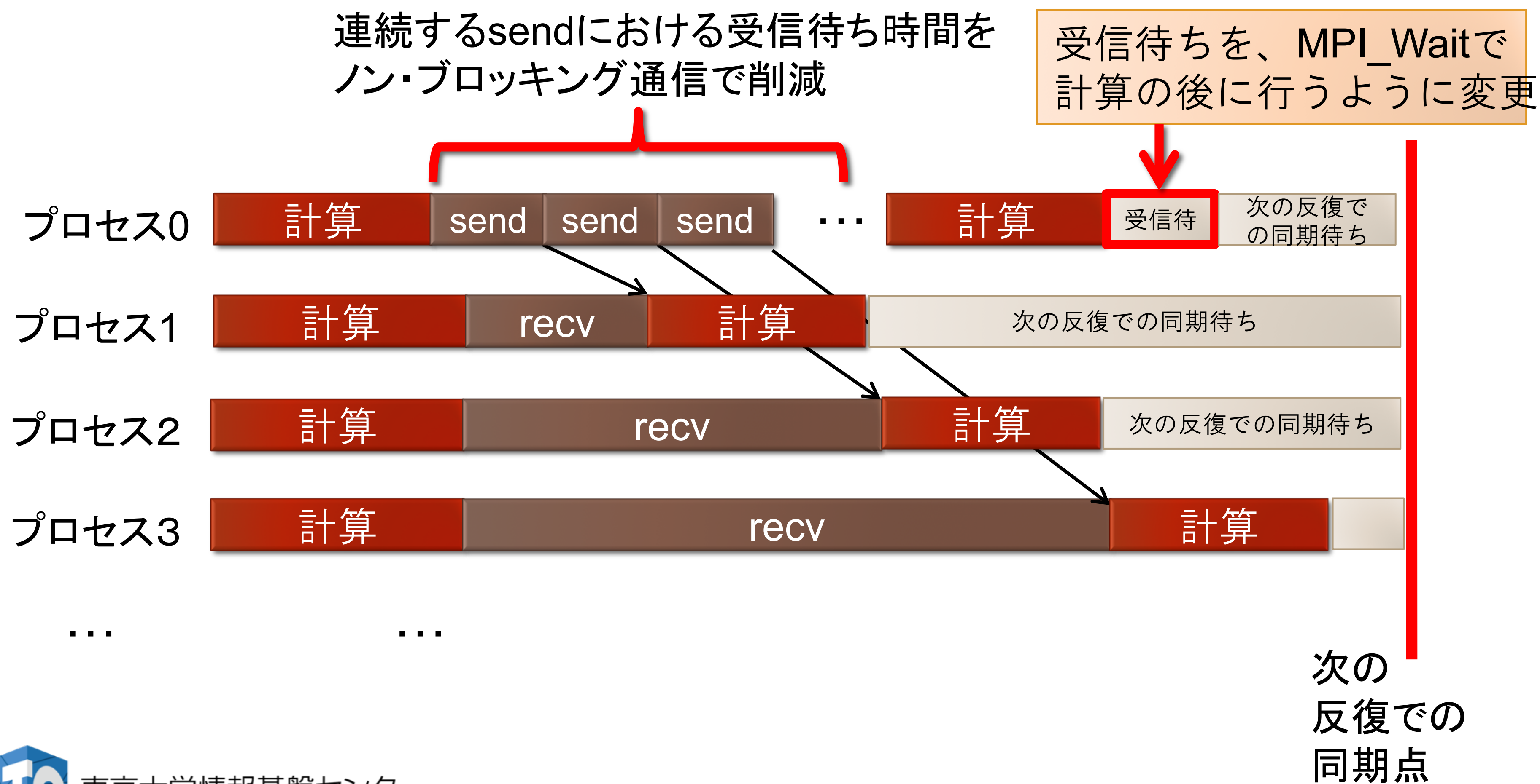
同期待ち関数

- 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に戻る；

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）

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

Omni Pathに関するヒント

プロトコルスタック選択

- I_MPI_FABRICS=ノード内:ノード間
  - shm: 共有メモリ（ノード内のみ選択可）
  - tmi: Omni Pathのおすすめ
  - ofi: 将来の標準
- デフォルトでは shm:tmi だが、経験上集団通信は tmi:tmi にした方が高速

試した方がいい

- HFI_NO_CPUAFFINITY=1
  - 差が出る場合と出ない場合がある？
    - ノード内複数プロセス実行時などは指定した方がいいかも

MPI+OpenMPハイブリッド実行に際して

- MPI関数を、マルチスレッド環境で実行する際は注意が必要
  - 3つのモード: Master only, Funneled, Multiple
    - Multipleでは、集団通信は単純には実行できない、あまりスケールしない
    - Master onlyではParallelリージョンから出なければいけない
  - 特に意識しないのであればFunneledで使うのがよい

Master only

```
#pragma omp parallel
{
....
}
  MPI_Send( ... );
#pragma omp parallel
 {
....
}
```

Funneled

```
#pragma omp parallel
{
  ....
#pragma omp master
  MPI_Send( ... );
  ....
}
```

Master onlyと似ているがparallel節を閉じなくていい

Multiple

```
#pragma omp parallel
{
  ....
  MPI_Send( ... );
  ....
}
```

本当にマルチスレッドだが、1対1通信しかできない

マルチスレッドMPIの初期化

- MPI_Init()の代わりに、MPI_Init_thread()を使用
- C言語:

```
int provided;
MPI_Init_thread(&argc,&argv,MPI_THREAD_FUNNELED,
&provided);
```

- Fortran

```
integer provided, required
required=MPI_THREAD_FUNNELED
call MPI_Init_thread(required, provided, ierr)
```

MPI実行

- ノードあたり1プロセスの最適値
  - HT数が異なる場合は単体OpenMPの設定を参考に
    - (KMP_AFFINITY, KMP_HW_SUBSET)

```
export OMP_NUM_THREADS=64 # 1HTの場合
export I_MPI_DEBUG=5  # コア割り当て確認
export I_MPI_PIN_DOMAIN=256
export I_MPI_PIN_PROCESSOR_EXCLUDE_LIST=0,1,68,69,
             136,137,204,205 # ジッタ対策、コア除け
export KMP_AFFINITY=scatter
export KMP_HW_SUBSET=1t
```

MPI実行（ノードあたり複数プロセス: 2）

- PPN (Process Per Node) = 2 の場合
  - I_MPI_PERHOST=2

  または
  - mpiexec.hydraの引数に "-ppn 2"を追加
- I_MPI_PIN_DOMAINでプロセス毎へのコア割り当てを指定

```
export OMP_NUM_THREADS=32
# 1HTの場合
export I_MPI_DEBUG=5
# コア割り当て確認
export I_MPI_PIN_DOMAIN=128
export I_MPI_PERHOST=2
export
I_MPI_PIN_PROCESSOR_EXCLUDE_LIST=0,1,68,69,136,137,204,205 # ジッタ対策、コア除け
export KMP_AFFINITY=scatter
export KMP_HW_SUBSET=1t
```

MPI実行（ノードあたり複数プロセス: 4）

- PPN (Process Per Node) = 4 の場合
  - I_MPI_PERHOST=4

  または
  - mpiexec.hydraの引数に“-ppn 4”を追加

- I_MPI_PIN_DOMAINでプロセス毎へのコア割り当てを指定

```
export OMP_NUM_THREADS=16
# 1HTの場合
export I_MPI_DEBUG=5
# コア割り当て確認
export I_MPI_PIN_DOMAIN=64
export I_MPI_PERHOST=4
export
I_MPI_PIN_PROCESSOR_EXCLUDE_LIST=0,1,68,69,136,137,204,205 # ジッタ対策、コア除け
export KMP_AFFINITY=scatter
export KMP_HW_SUBSET=1t
```

MPI実行（ノードあたり複数プロセス:8以上、その他）

- 8以上も同様に設定
  - I_MPI_PERHOSTの値と、I_MPI_PIN_DOMAINの値の積が256になっている必要あり、そうしないと、1コアにプロセスが偏ってしまう
- I_MPI_DEBUG=5にすると、プロセスに割り当てられたコア番号が得られるので意図通りか確認すること

```
[40] MPI startup(): shm and tmi data transfer modes
…
[0] MPI startup(): 192     269565   c3850.ofp {2,70,138,206}
```

設定スクリプト

- core-setting.sh
- 使い方：
あらかじめ以下の変数を設定
  - HTX=1 (コア当たりスレッド数)
  - VERBOSE=1 (設定された環境変数を表示)
  - DEBUG=1 (コア割り当ての詳細を表示)

- source core-setting.sh
- これまで説明した最適な設定を生成
- 出力例：

```
Setting:
export I_MPI_PIN_DOMAIN=64
export I_MPI_PERHOST=4
export KMP_HW_SUBSET=1t
export KMP_AFFINITY=verbose,compact
export HFI_NO_CPUAFFINITY=1
export
I_MPI_PIN_PROCESSOR_EXCLUDE_LIST=0,1,68,69,136,137,204,205
[0] MPI startup(): Multi-threaded optimized
library
…
```

時間計測方法（C言語、MPI+OpenMP）

```
double  t0, t1, t2, t_w;
..すでにparallelリージョンにいると仮定
#pragma omp master
{
ierr = MPI_Barrier(MPI_COMM_WORLD);
}
#pragma omp barrier
t1 = omp_get_wtime();

<ここに測定したいプログラムを書く>
#pragma omp reduction(max: t0)
{t2 = omp_get_wtime();
t0 =  t2 - t1;}
#pragma omp master
{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、MPI+OpenMP）

```
  double precision t0, t1, t2, t_w
..すでにparallelリージョンにいると仮定
$!omp master
  call MPI_BARRIER(MPI_COMM_WORLD, ierr)
$!omp end master
$!omp barrier
t1 = omp_get_wtime()

<ここに測定したいプログラムを書く>
$!omp reduction (max:t0)
  t2 = omp_get_wtime()
  t0 =  t2 - t1
$!omp end reduction
$!omp master
  call  MPI_REDUCE(t0, t_w, 1,
&        MPI_DOUBLE_PRECISION,
&        MPI_MAX, 0, MPI_COMM_WORLD, ierr)
$!omp end master
```

バリア同期後
時間を習得し保存

各プロセッサで、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

通信と計算のオーバラップ

- “dynamic/runtime”
- “!$omp master～!$omp end master”

コア同士の動作モードのばらつきも大きいので、動的スケジューリングが効くと考えられる

```
!$omp parallel private (neib,j,k,i,X1,X2,X3,WVAL1,WVAL2,WVAL3)
!$omp&         private (istart,inum,ii,ierr)

!$omp master                  Communication is done by the master thread (#0)
!C
!C– Send & Recv.
(…)
      call MPI_WAITALL (2*NEIBPETOT, req1, sta1, ierr)
!$omp end master

!C                         The master thread can join computing of internal
!C-- Pure Inner Nodes      nodes after the completion of communication

!$omp do schedule (runtime)
      do j= 1, Ninn
         (…)
      enddo
!C
!C-- Boundary Nodes        Computing for boundary nodes are by all threads

!$omp do                      default: !$omp do schedule (static)
      do j= Ninn+1, N
         (…)
      enddo

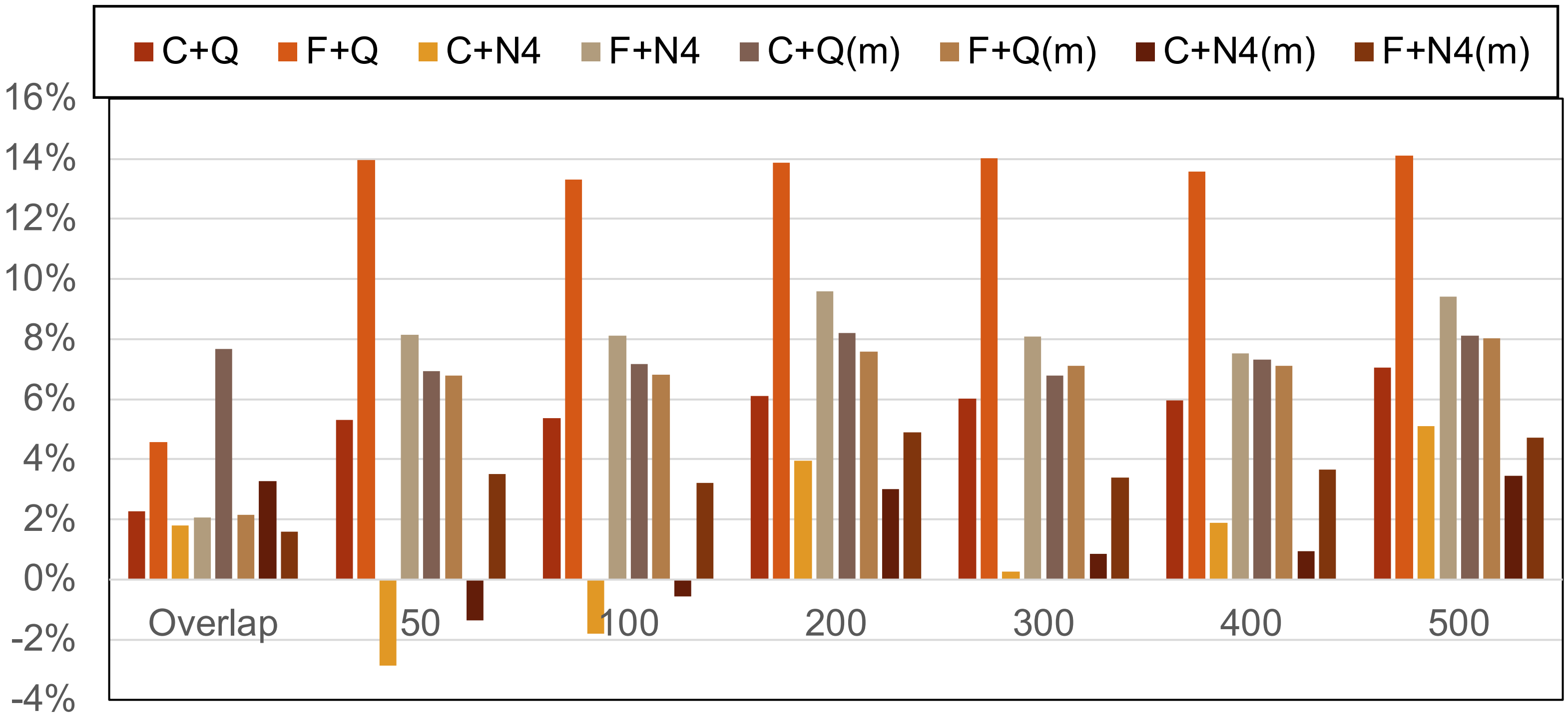
!$omp end parallel
```

export OMP_SCHEDULE="dynamic,[chunksize]"

Idomura, Y. et al., Communication-overlap techniques for improved strong scaling of gyrokinetic Eulerian code beyond 100k cores on the K-computer, Int. J. HPC Appl. 28, 73-86, 2014

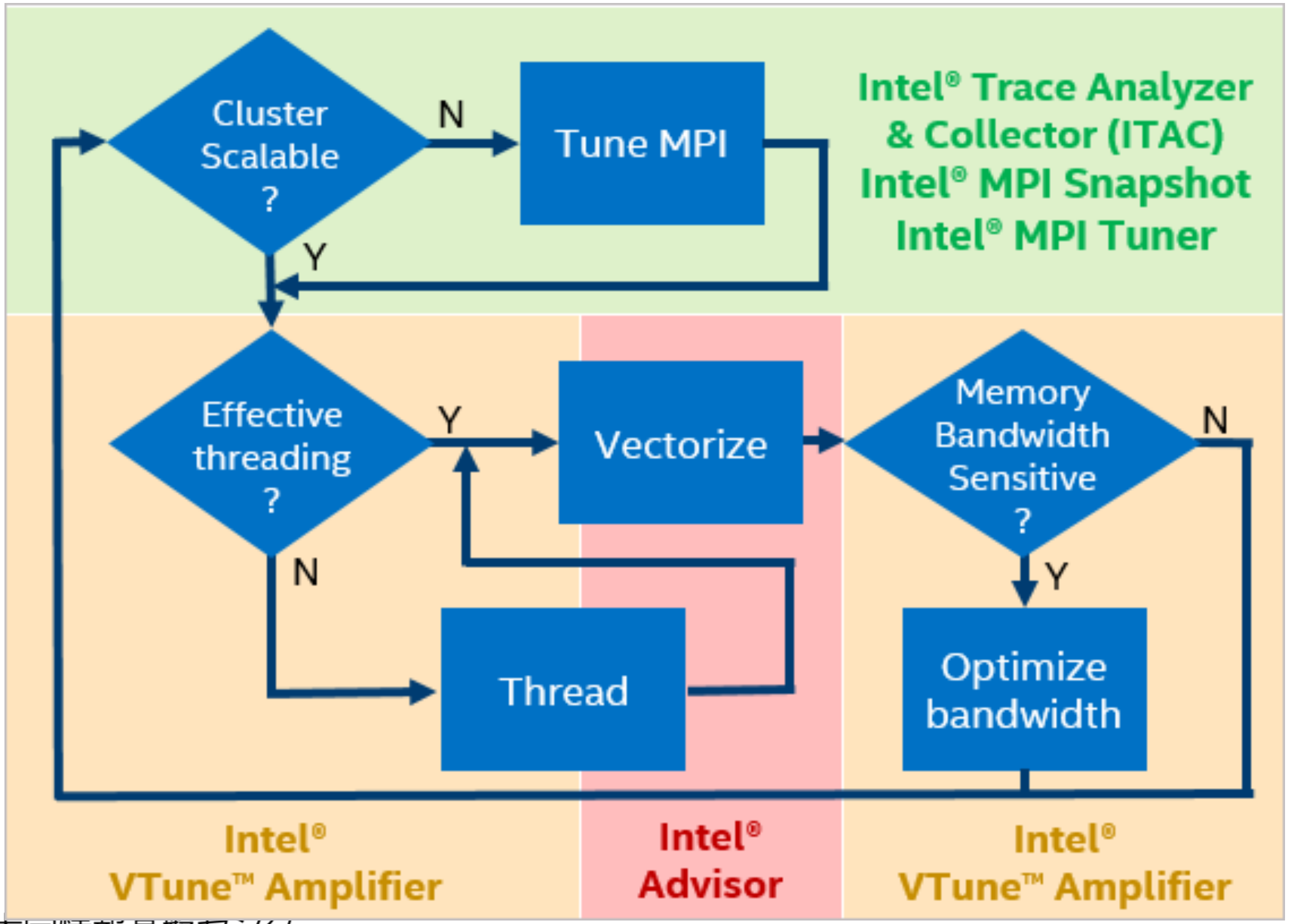
Speedup by communication overlapping (OFP: 32 nodes)

- Memory model: C: Cache, F: Flat
- Sub NUMA: Q: Quadrant, N4: SNC-4
- (m) : core binding to avoid "busy" core



プロファイリング

- 以下，Intel VTune 2017ドキュメントより



通信性能安定化

- MPI通信で触る配列を、2Mバイトにアラインする

Fortranでは、(注：Fortranではバージョン2018から利用可能)

```
!DIR$ ATTRIBUTES ALIGN : 2097152 :: arrayMPI
real(8),allocatable :: arrayMPI(:,:,:,:)
```

Cでは、（コンパイラのバージョン依存なし）

```
arrayMPI = (double *)__mm_malloc(array_size,
2*1024*1024);
```

Intel Advisorによるデータ収集

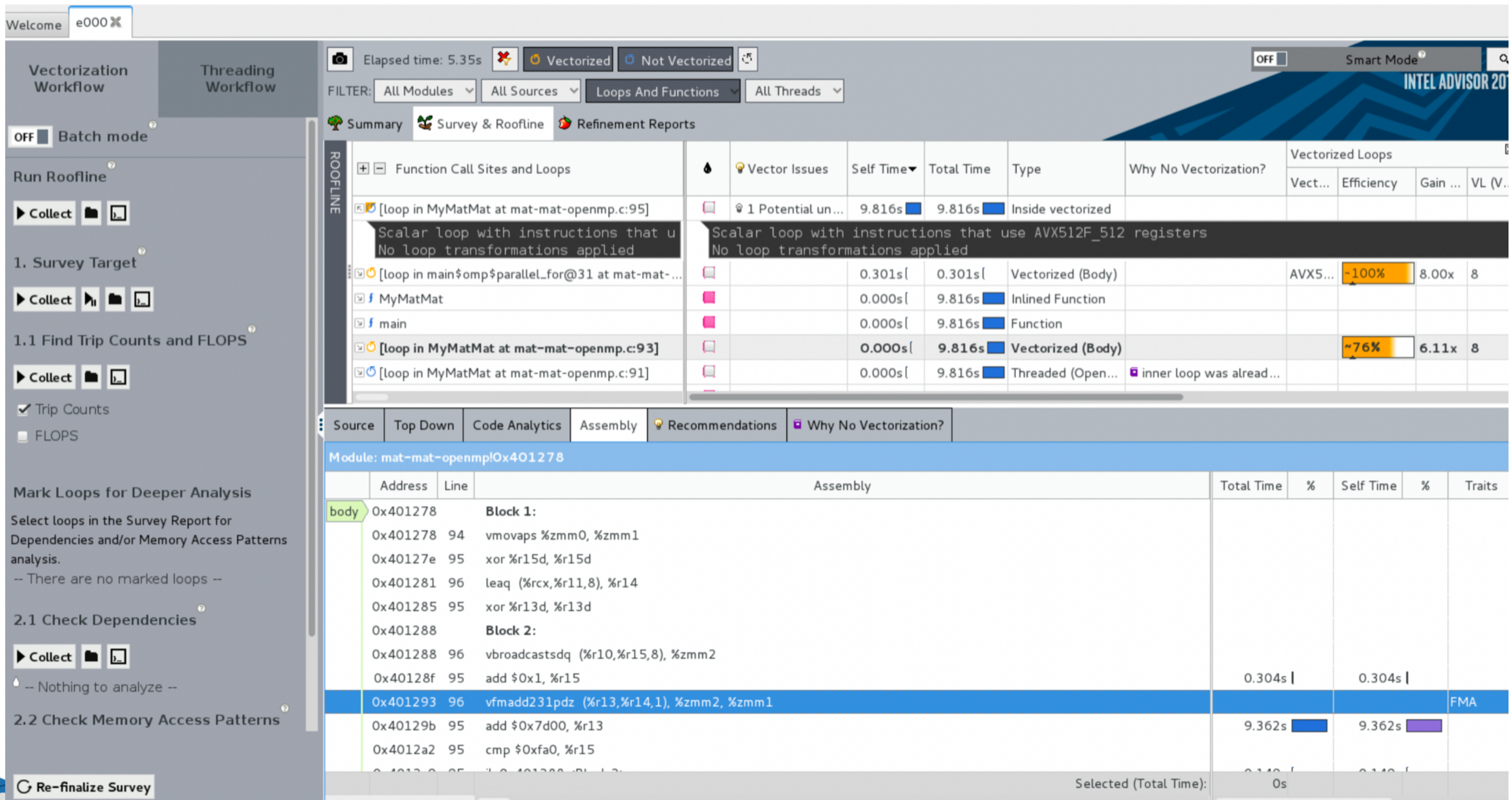
- ベクトル化，スレッド並列化支援
  - module load advisorにて有効
- データの収集：advixe-cl コマンド
  - advixe-cl --collect survey --project-dir ./advi --no-auto-finalize -- ./a.out
  - MCDRAMを使うなら：
    numactl --membind=1 advixe-cl --collect survey --project-dir ./advi --no-auto-finalize -- ./a.out
  - MPIを使うなら：
    mpiexec.hydra -n ${PJM_MPI_PROC} advixe-cl --collect survey --project-dir ./advi --no-auto-finalize -- ./a.out
- Tips:
  - --no-auto-finalize : KNLにfinalizeまでやらせると遅いので，あとでログインノードでやってもらう

Intel Advisorによる結果表示(1)

- VNCサーバ利用準備
  - ofp01$ vncserver :0 -geometry 1280x1024
    （初回はVNC接続用パスワードを聞かれる）
  - ofp01$ export DISPLAY=:0
    （上の :0 に合わせる）
- 手元の端末でVNC用ポートを転送
  - note$ ssh -N -L 55901:localhost:5900 z30105@ofp01.jcahpc.jp &
  - VNCサーバが :0 なら 5900, :1 なら 5901, …
  - VNCクライアントで localhost:55901 を指定

Intel Advisorによる結果表示(2)

- あらかじめsource, binaryのパスを設定しておく



Intel VTuneによるデータ収集

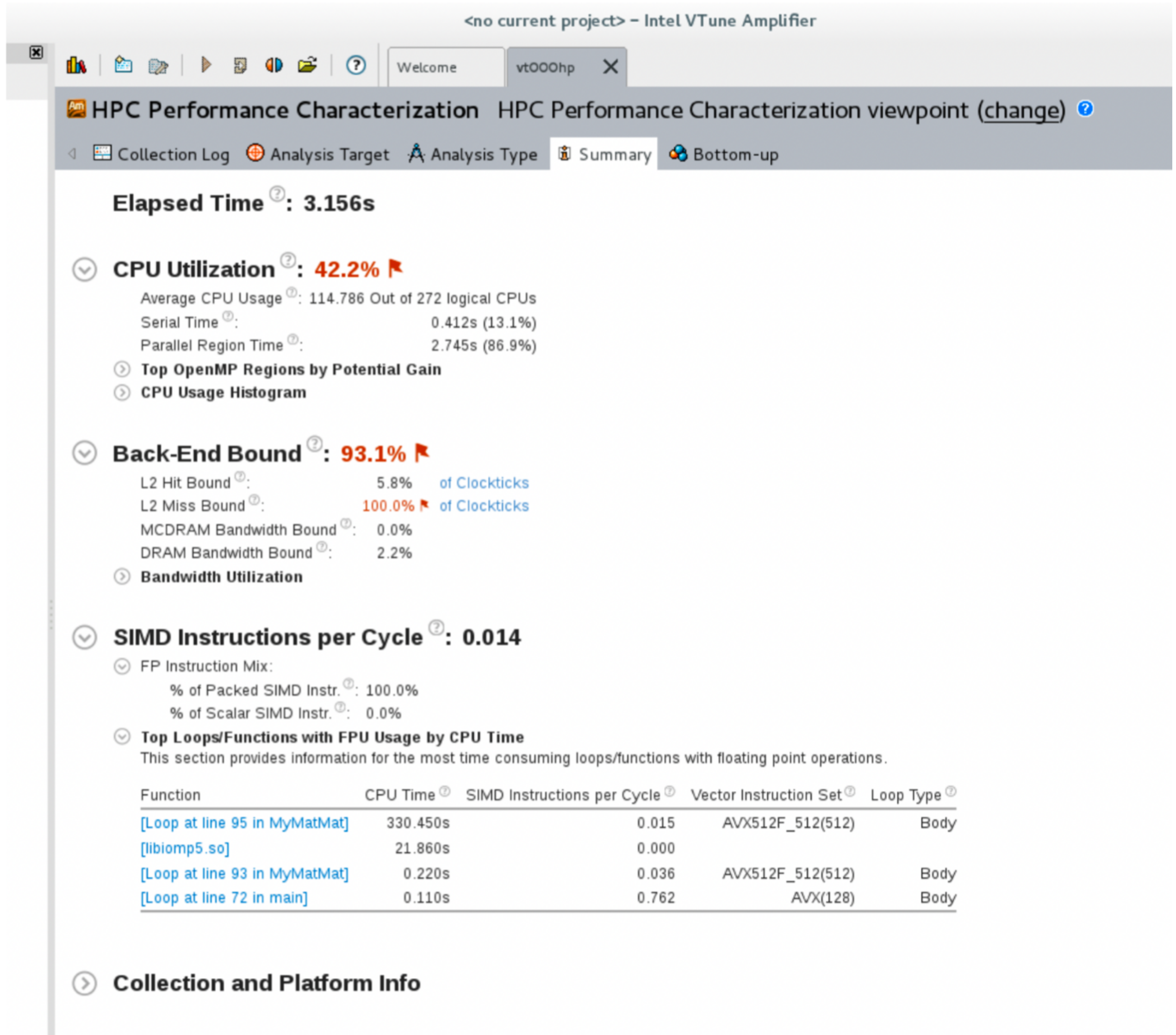
- メモリバンド幅等，リソース使用量を取得
  - module load vtuneにて有効
- データの収集：amplxe-cl コマンド
  - amplxe-cl -collect hpc-performance --no-auto-finalize -r vt000hp -- ./a.out
  - MCDRAMを使うなら：
    numactl -m 1 amplxe-cl -collect hpc-performance --no-auto-finalize -r vt000hp -- ./a.out
  - MPIを使うなら：
    mpiexec.hydra -n ${PJM_MPI_PROC} amplxe-cl -collect hpc-performance --no-auto-finalize -r vt000hp -- ./a.out
  - その他 knob
- Tips:
  - --no-auto-finalize : KNLにfinalizeまでやらせると遅いので，あとでログインノードでやってもらう

Intel VTuneによる結果表示(1)

- VNCサーバ利用準備
  - Advisorと同様

- CUIからfinalize
  - $ amplxe-cl -finalize -search-dir . -r vt000hp

Intel VTuneによる結果表示(2)



バーストバッファ (DDN: Infinite Memory Engine)

- SSD搭載のサーバが25台，計50ノード
  - 容量：960 TB，理論ピーク性能 1.5TB/sec
  - IO500でTopの性能
- クライアントでパリティを計算して書き込む
  - erasure coding
- Lustreファイルシステムのキャッシュとしても動作
  - /cache でキャッシュへのアクセス（計算ノードのみ）
  - ステージングが必要（ステージイン，ステージアウト操作）

    #PJM -x STGIN_LIST=<一覧をリストしたファイル名>

    #PJM -x STGOUT_LIST=<一覧をリストしたファイル名>
  - ステージインを待ちたいとき
    - prestage-wait コマンド

IMEの使い方

1. POSIX:
   /work/gt00/t0044x の代わりに /cache/gt00/t0044x を使う

Intel MPIを使いたい場合にも有効！！

2. MPI IO:
   a. MPIをIntel MPIからMVAPICH2に切り替え

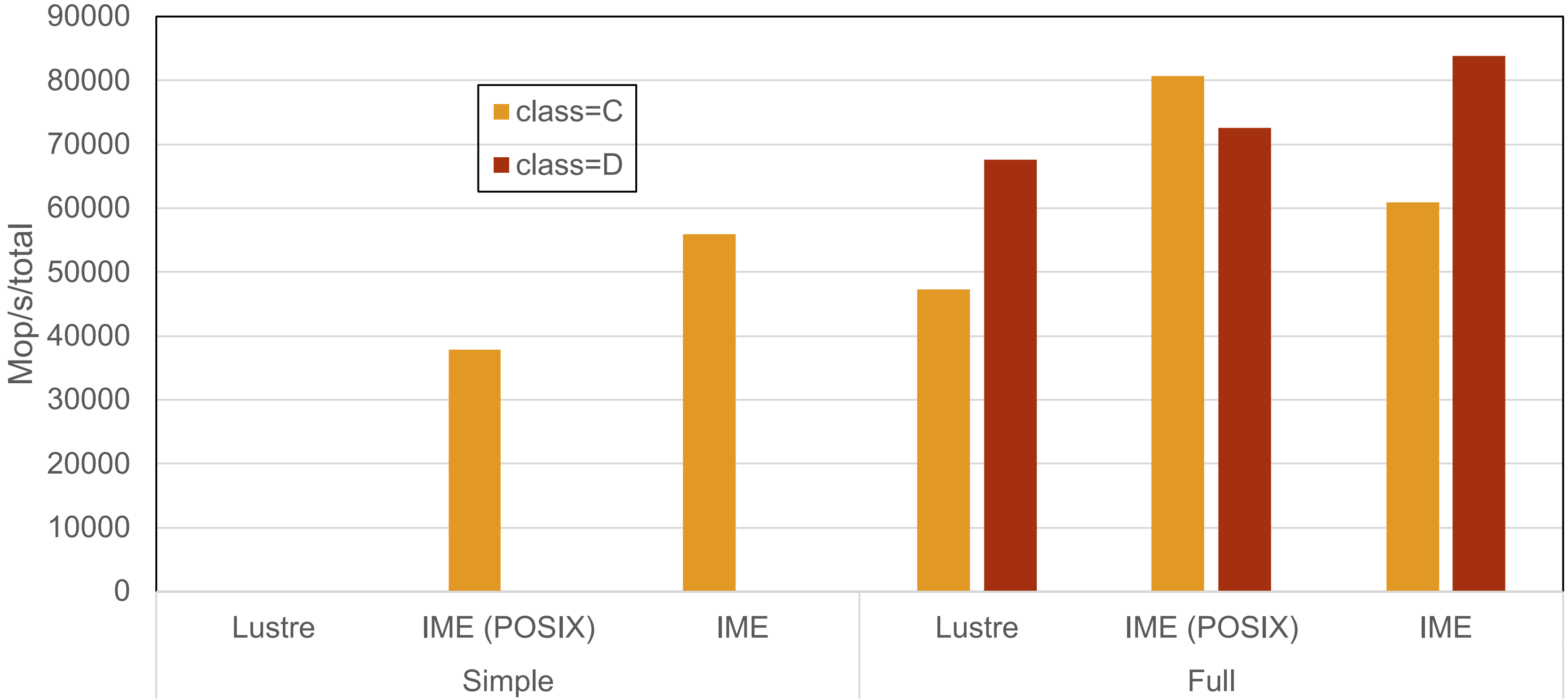
```
module switch impi mvapich2
```

   b. ファイルパスに ime:// を追加

MPI_File_open(MPI_COMM_WORLD,"ime:///work/gt00/t0044x…", …);

IMEの性能

- NAS Parallel benchmark, BT-IOの例
  - class=C, 64プロセス (16ノード x 4)
    - LustreでのSimpleは時間内に終了せず
  - class=D, 256プロセス(16ノードx16)
    - LustreでのSimpleは時間内に終了せず, IMEでのSimpleはエラー



演習：
サンプルプログラム集

- Ｃ言語版・Fortran90版共通ファイル：
  Samples-knl.tar.gz
- tarで展開後、各サンプルとともに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）

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/t0044x

1. /work/gt00/z30105 にある　Samples-knl.tar.gz　を自分のディレクトリにコピーする

   $ cp /work/gt00/z30105/Samples-knl.tar.gz ./

2. Samples-knl.tar.gz を展開する

   $ tar xvfz Samples-knl.tar.gz

3. Samples フォルダに入る

   $ cd Samples-knl

4. アプリ名のフォルダに入る

   $ cd stream

5. C言語　：$ cd C

   Fortran90言語：$ cd F

プログラムをコンパイルしよう(2/2)

6. make する
```
$ make
```
7. 実行ファイル(stream)ができていることを確認する
```
$ ls
```

プログラムを実行しよう

1. streamフォルダ中で以下を実行する
   `$ pjsub  stream.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
   stream.bash.eXXXXXX
   stream.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 stream.bash.oXXXXXX`
5. エラー出力は以下のファイルにあるので念のため確認
   `$ cat stream.bash.eXXXXXX`