

内容に関する質問は
katagiri@cc.nagoya-u.ac.jp
まで

第4回 Hybrid並列化技法 (MPIとOpenMPの応用)

名古屋大学情報基盤センター 片桐孝洋



講義日程と内容について

- ▶ 2019年度 計算科学技術特論A(1学期:木曜3限)
 - ▶ 第1回:プログラム高速化の基礎、2019年4月11日
 - ▶ イントロダクション、ループアンローリング、キャッシュブロック化、数値計算ライブラリの利用、その他
 - ▶ 第2回:MPIの基礎、2019年4月18日
 - ▶ 並列処理の基礎、MPIインターフェース、MPI通信の種類、その他
 - ▶ 第3回:OpenMPの基礎、2019年4月25日
 - ▶ OpenMPの基礎、利用方法、その他
 - ▶ 第4回:Hybrid並列化技法(MPIとOpenMPの応用)、2019年5月9日
 - ▶ 背景、Hybrid並列化の適用事例、利用上の注意、その他
 - ▶ 第5回:プログラム高速化の応用、2019年5月16日
 - ▶ プログラムの性能ボトルネックに関する考えかた(I/O、単体性能(演算機ネック、メモリネック)、並列性能(バランス))、性能プロファイル、その他

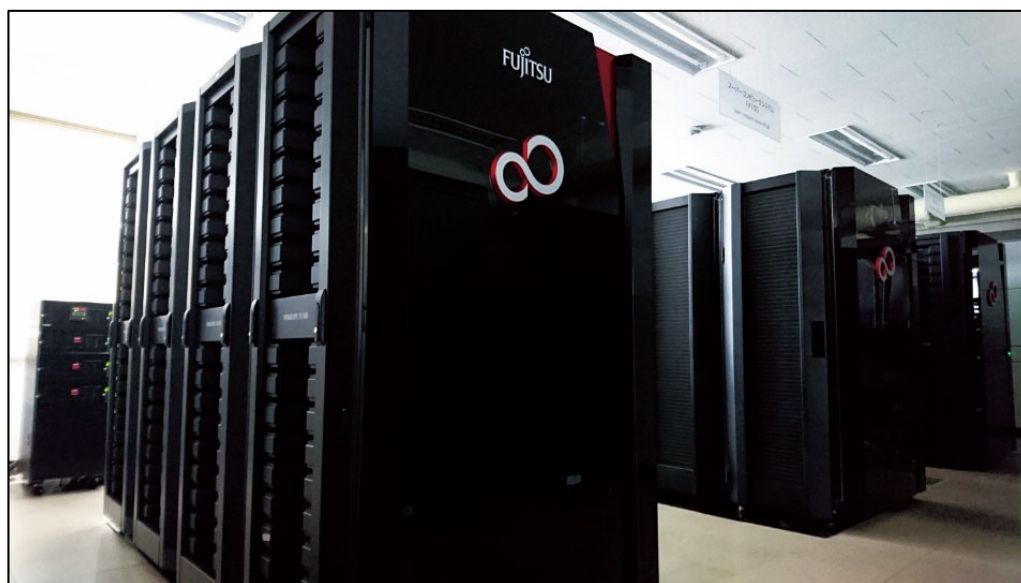
実際の並列計算機構成例



名古屋大学情報基盤センタースパコン

FX100スーパーコンピュータシステム

計算ノード - プロセッサ, メモリ容量	Fujitsu SPARC64 XIfx (2.2 GHz) 32コア, 32 GiB
総ノード数 (総コア数)	2,880 ノード (92,160 コア)
総演算性能	3.2 PFlop/s
総メモリ容量	90 TiB



製品名 :
Fujitsu PRIMEHPC
FX100

2015年9月運用開始



FX100計算ノードの構成

2ソケット、NUMA
(Non Uniform Memory Access)

TOFU2
Network

HMC
16GB

Memory

L2 (17コアで共有、12MB)

速い

遅い

ソケット0 (CMG(Core Memory Group))

ICC

Core
#16

Core
#17

Core
#18

Core
#19

: L1データ
キャッシュ
64KB

Assist.
Core

Core
#28

Core
#29

Core
#30

Core
#31

L2 (17コアで共有、12MB)

ソケット1 (CMC)

Memory

HMC
16GB

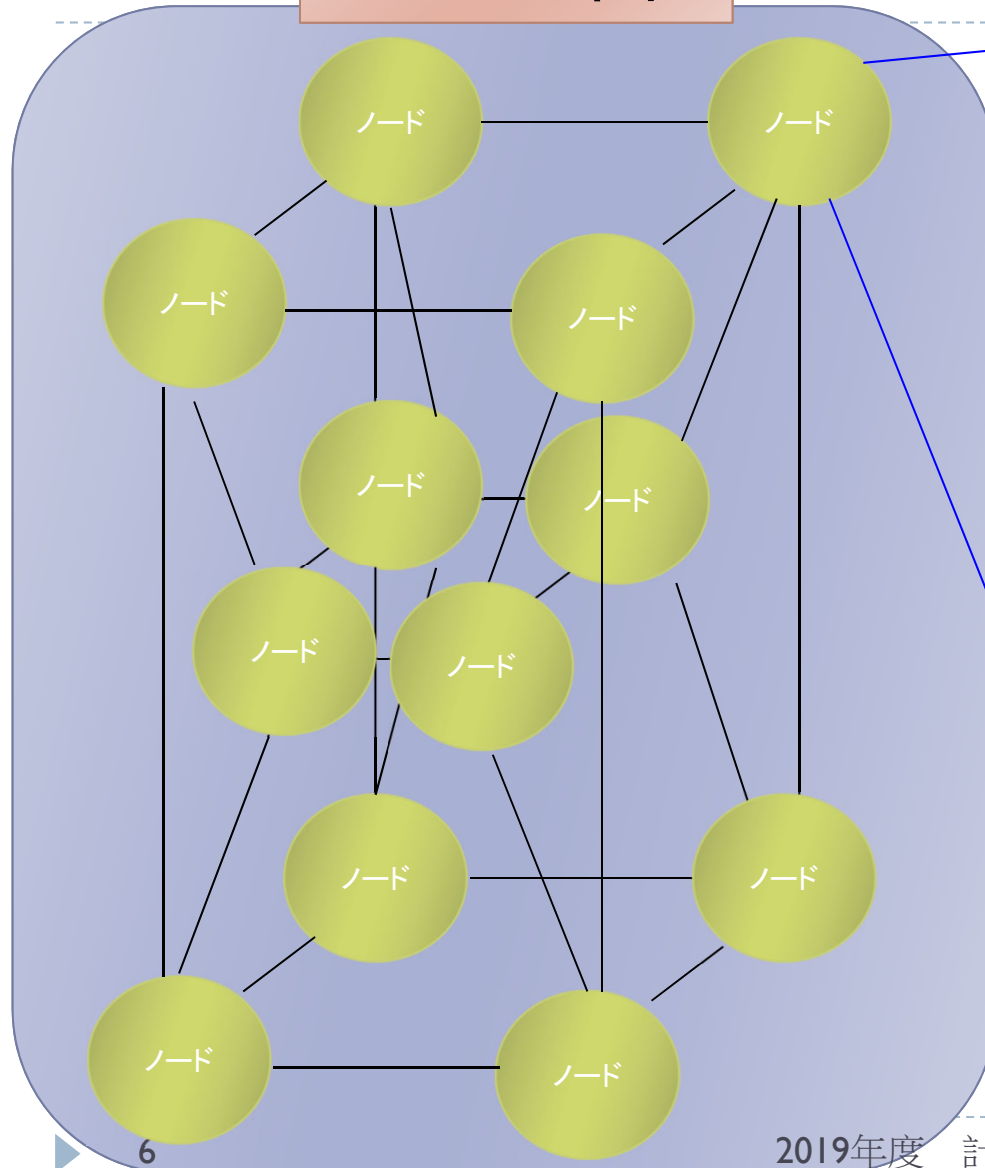
ノード内合計メモリ量: 32GB

読み込み: 240GB/秒

書き込み: 240GB/秒=合計: 480GB/秒

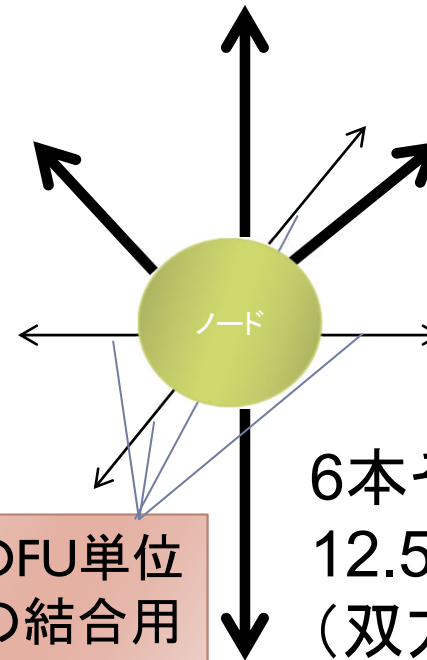
FX100の通信網（1 TOFU単位）

1 TOFU単位



計算ノード内

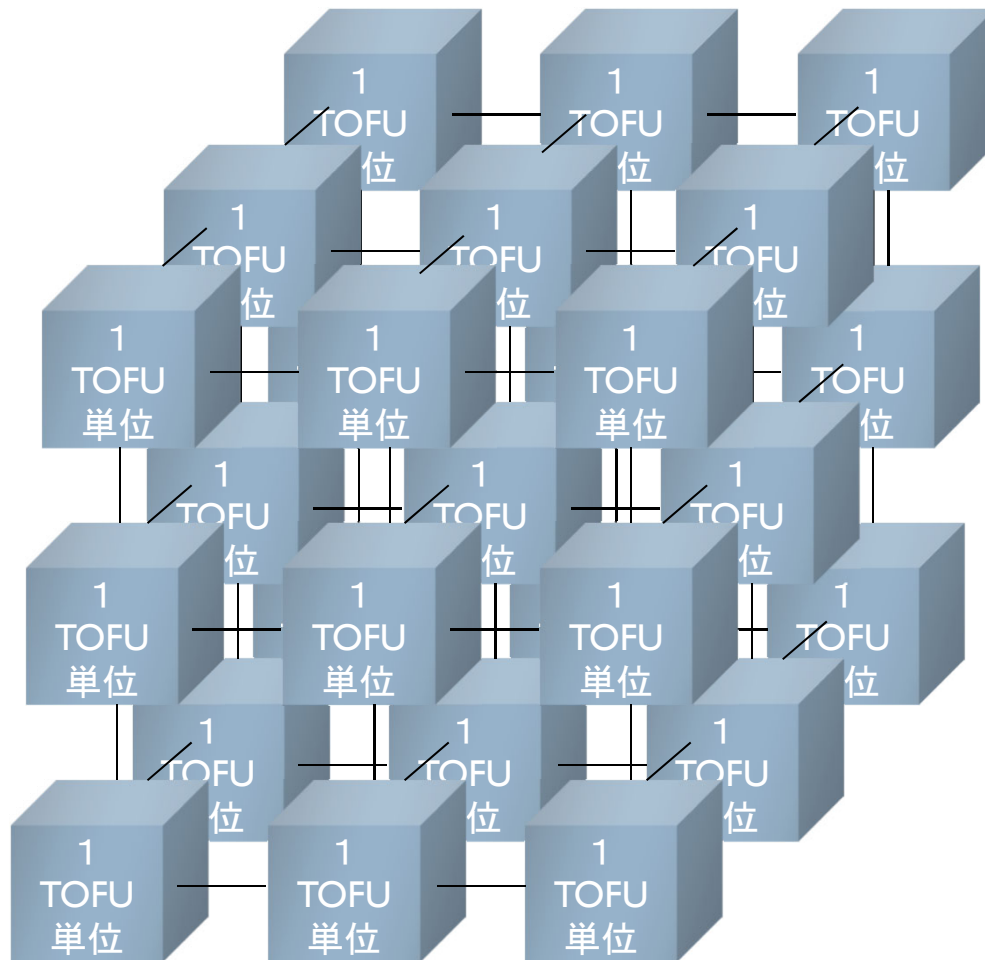
ITOFU単位
間の結合用



6本それぞれ
12.5GB/秒
(双方向)

FX100の通信網（1 TOFU単位間の結合）

3次元接続



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

Oakforest-PACS

(JCAHPC、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)

- 全体構成 (Source: <http://www.cc.u-tokyo.ac.jp/image/Oakforest-PACS.jpg>)

内容	仕様
Total Theoretical Peak	25.004 PFLOPS
Total Number of Nodes	8208
Total Memory Amount	897 TByte
Total Storage Amount	26 PB
Total Storage Bandwidth	500 GB/sec
Total Fast Cache Amount	940 TB
Total Fast Cache Bandwidth	1,560 GB/sec

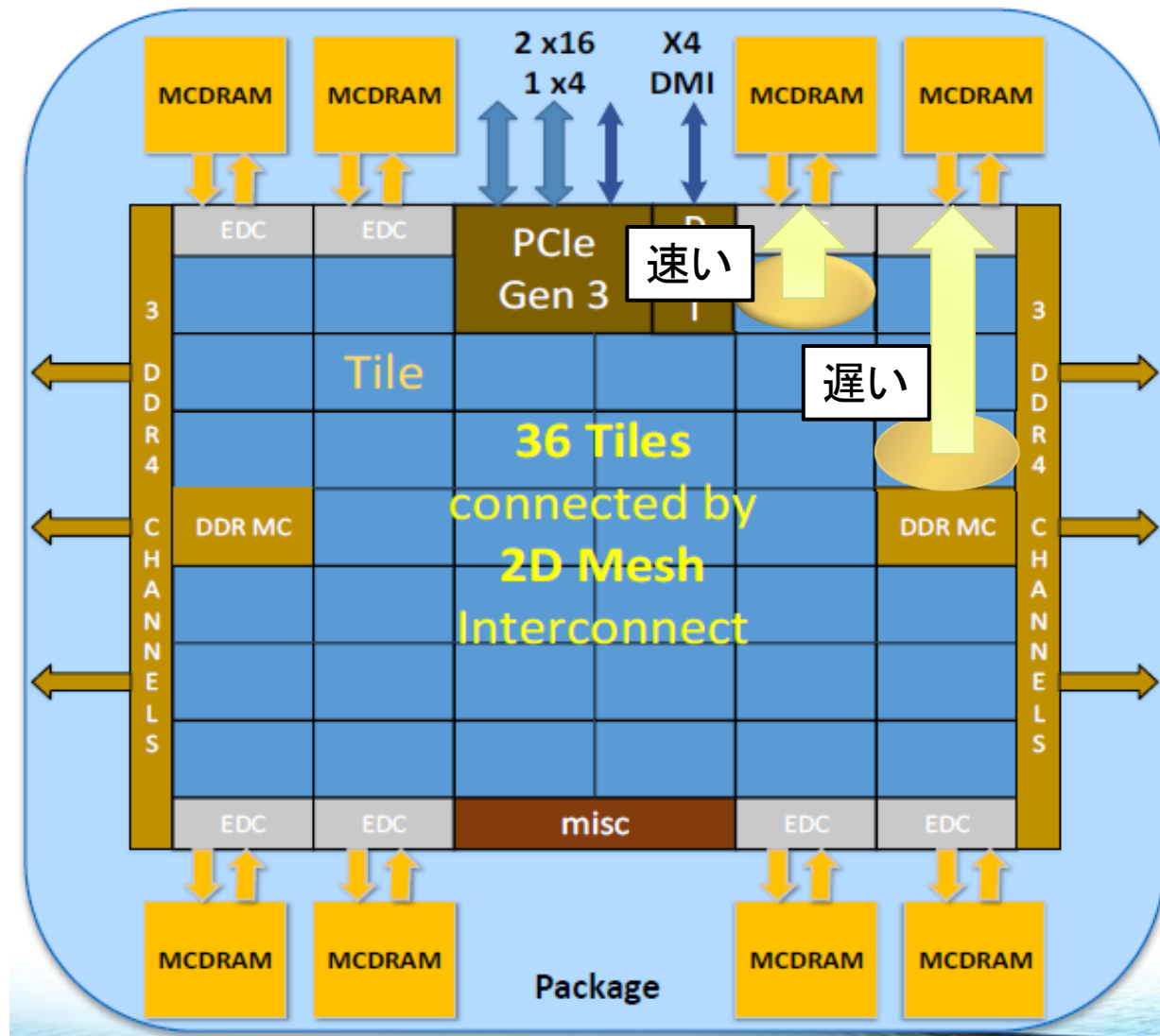


8,208ノード (558,144 コア)

- ノード構成

CPU	Intel® Xeon Phi™ 7250 (Knights Landing, KNL)
#Processors	1 (68 Physical Cores)
Frequency	1.4 GHz
Theoretical Peak for each Node	3.0464 TFLOPS (double) / 6.0928 TFLOPS (single)
Memory	96 GB(DDR4)+ 16 GB(MCDRAM)
Inter ⁸ Connect	Intel® Omni-Path Network (100 Gbps)

KNLのアーキテクチャ



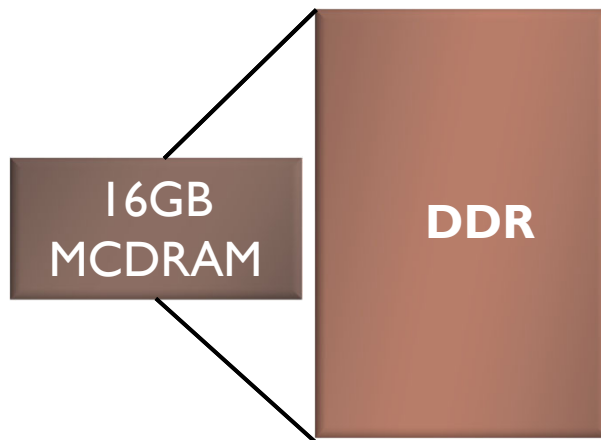
Intel Xeon Phi (Knights Landing)

- Many Cores
 - 各CPUあたり
 - 68物理コア
 - 最大272スレッド
- Non Uniform Memory Access (NUMA)

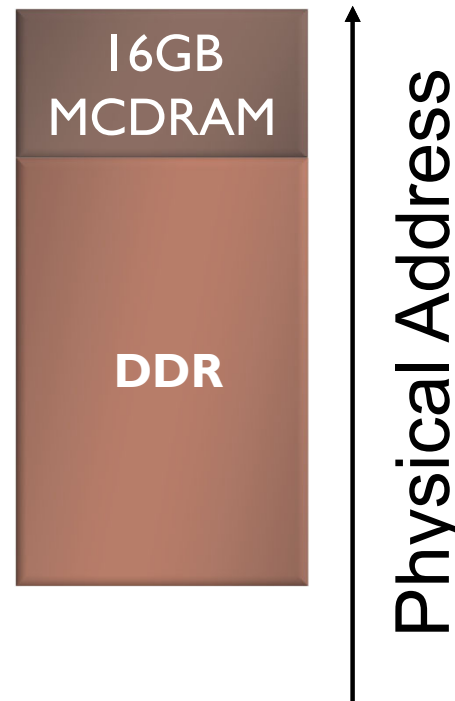
Source: Knights Landing (KNL): 2nd Generation Intel® Xeon Phi™ Processor, Avinash Sodani KNL Chief Architect Senior Principal Engineer, Intel Corp.

KNLの実行モード

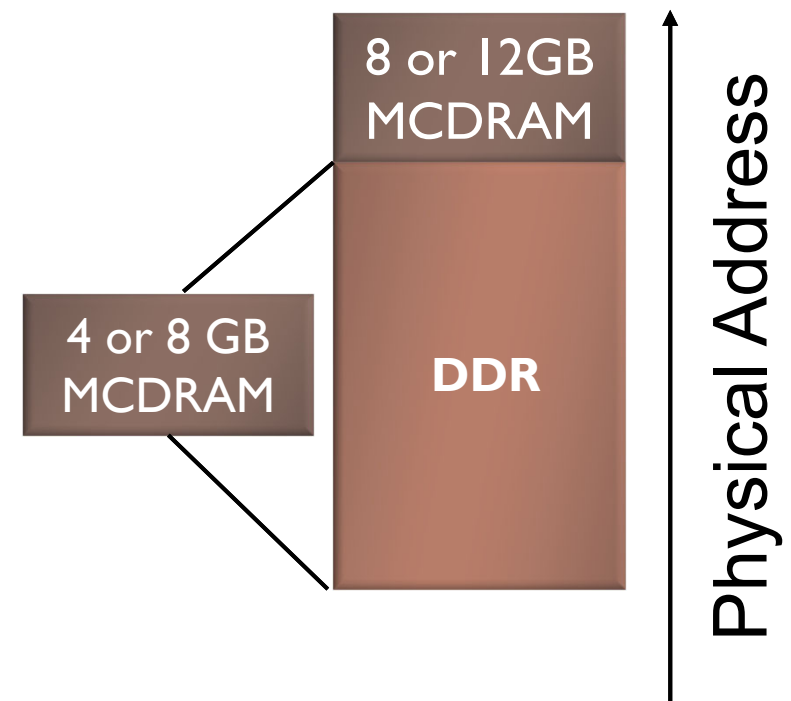
キャッシュ
モード



フラット
モード



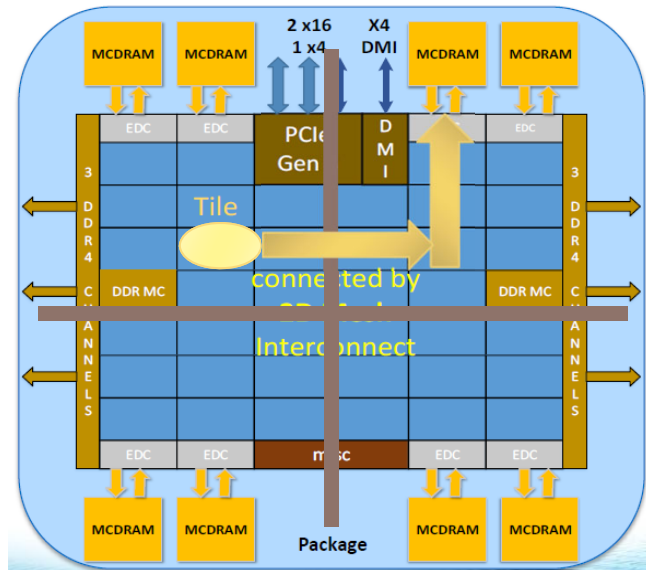
ハイブリッド
モード



Source: Knights Landing (KNL): 2nd Generation Intel® Xeon Phi™ Processor, Avinash Sodani KNL Chief Architect Senior Principal Engineer, Intel Corp.

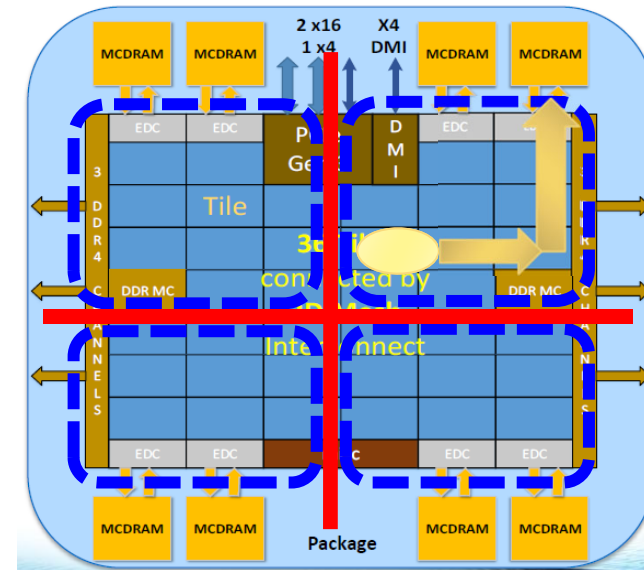
KNLのメッシュモード

Quadrant



- チップ4つの仮想 **Quadrant** に分ける
- 同一のQuadrantとしてメモリ上のディレクトリにアドレスは割り当てられる
- ディレクトリとメモリに対するアフィニティ

Sub-NUMA Clustering (SNC)



- それぞれの**分割したNUMAドメイン**として構成されて、OSに認識される
- タイル、ディレクトリ、メモリで、4ソケットのXeonアフィニティと同じ概念
- 局所通信。各モードで最も少ないレイテンシ。
- 効果を得るためには、**NUMA最適化が必要**。

Source: Knights Landing (KNL): 2nd Generation Intel® Xeon Phi™ Processor, Avinash Sodani KNL Chief Architect Senior Principal Engineer, Intel Corp.

2019年度 計算科学技術特論A

バッチ処理とMPIジョブの投入



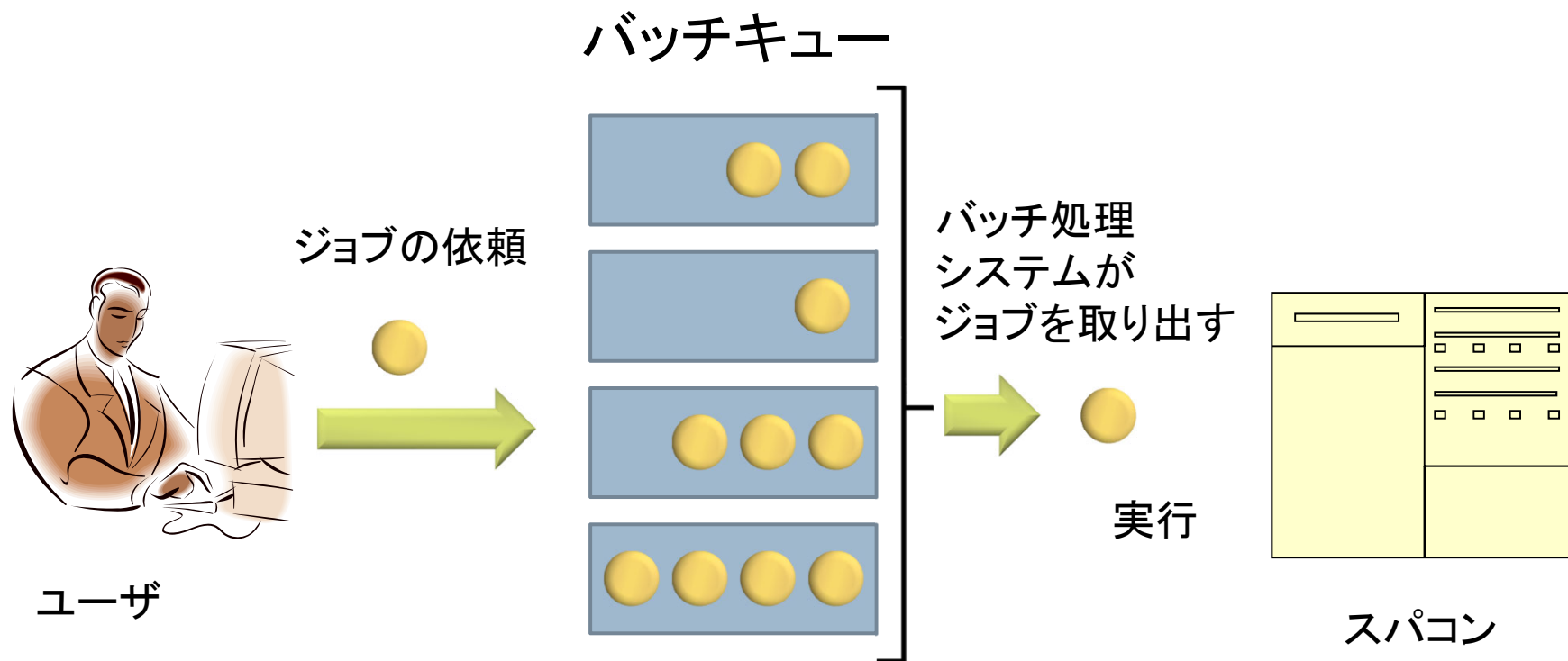
FX100のジョブ実行形態の例

- ▶ 以下の2通りがあります
- ▶ **インタラクティブジョブ実行**
 - ▶ PCでの実行のように、コマンドを入力して実行する方法
 - ▶ スパコン環境では、あまり一般的でない
 - ▶ デバック用、大規模実行はできない
 - ▶ 名大FX100では、以下に限定(名大基盤センターの運用方針)
 - ▶ **最大4ノード(128コア)(標準1時間まで、最大で24時間)**
- ▶ **バッチジョブ実行**
 - ▶ バッチジョブシステムに処理を依頼して実行する方法
 - ▶ スパコン環境で一般的
 - ▶ 大規模実行用
 - ▶ 名大FX100では:
 - ▶ **通常サービス: 最大864ノード(27,648コア)(24時間まで)**
 - ▶ **申込み制: 2,592ノード(82,944コア)(実行時間制限無し)**



バッチ処理とは

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



コンパイラの種類とインタラクティブ実行 およびバッチ実行の例 (FX10、FX100)

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

バッチキューの設定のしかた (FX100の例)

- ▶ バッチ処理は、富士通社のバッチシステム(PJM)で管理されています。
- ▶ 以下、主要コマンドを説明します。
 - ▶ ジョブの投入：
`pjsub <ジョブスクリプトファイル名>`
 - ▶ 自分が投入したジョブの状況確認: `pjstat`
 - ▶ 投入ジョブの削除: `pjdel <ジョブID>`
 - ▶ バッチキューの状態を見る: `pjstat --rsc`
 - ▶ 過去の投入履歴を見る: `pjstat --history`
 - ▶ システム制限値を見る: `pjstat --limit`



インタラクティブ実行のやり方の例 (FX100スーパーコンピュータシステム)

- ▶ コマンドラインで以下を入力

- ▶ 1ノード実行用

- ```
$ pjsub --interact
```

- ▶ 4ノード実行用

- ```
$ pjsub --interact -L "node=4"
```

pjstat --rsc の実行画面例

```
$ pjstat --rsc
```

RSCUNIT	RSCUNIT_SIZE	RSCGRP	RSCGRP_SIZE
fx[ENABLE,START]	4x7x9	fx-interactive[ENABLE,START]	8x3x2
fx[ENABLE,START]	4x7x9	fx-debug[ENABLE,START]	8x3x2
fx[ENABLE,START]	4x7x9	fx-small[ENABLE,START]	8x3x16
fx[ENABLE,START]	4x7x9	fx-middle[ENABLE,START]	2592
fx[ENABLE,START]	4x7x9	fx-large[ENABLE,START]	2592
fx[ENABLE,START]	4x7x9	fx-xlarge[ENABLE,START]	2592
fx[ENABLE,START]	4x7x9	fx-special[ENABLE,START]	2592

使える
キュー名
(リソース
グループ)

現在使えるか
ENABLE: キューに
ジョブ投入可能
START: ジョブが流れて
いる

ノードの
物理構成情報

pjstat の実行画面例

\$ pjstat

```

ACCEPT QUEUED STGIN  READY RUNING RUNOUT STGOUT  HOLD  ERROR  TOTAL
   0         0       0     0       0       2       0       0       0       0
s  0         0       0     0       0       2       0       0       0       0
JOB_ID JOB_NAME  MD  ST   USER  START_DATE  ELAPSE_LIM  NODE_REQUIRE CORE V_MEM
696    run_4.sh  NM  RUN  user01  07/20 15:26:35  0000:10:00      4      -      -
697    run_4.sh  NM  RUN  user01  07/20 15:26:36  0000:10:00      4      -      -

```

表 3-14 FX ジョブ情報の表示項目

項目	説明
JOB_ID	ジョブ ID
JOB_NAME	ジョブ名
MD	ジョブモード (normal, step)
ST	ジョブの現在の状態
USER	ユーザー名
RSCGRP	リソースグループ名 (-v --pattern=1 指定時のみ)
START_DATE	ジョブが実行前の場合は開始予測時刻 ("()" で表示)、実行中および実行後の場合は実際に実行を開始した時刻。 実行開始時刻を指定して投入したジョブが実行を開始するまでの間、時刻の後ろに「@」が出力される。 バックフィルが適用されたジョブは、時刻の後ろに「<」が出力される。
ELAPSE_LIM	ジョブの経過時間 (実行中でないジョブは "--:--:--")
NODE_REQUIRE	ジョブのノード数とノード形状 (nnnn:XXxYxZ)

JOBスクリプトサンプルの説明（**ピュアMP I**）

（hello-pure.bash, C言語、Fortran言語共通）

```
#!/bin/bash
```

```
#PJM -L "rscgrp=fx-debug"
```

```
#PJM -L "node=12"
```

```
#PJM --mpi "proc=384"
```

```
#PJM -L "elapse=1:00"
```

```
mpirun ./hello
```

リソースグループ名
:fx-debug

利用ノード数

利用コア数
(MPIプロセス数)

実行時間制限
:1分

MPIジョブを $32 * 12 = 384$ プロセス
で実行する。

FX100計算ノードの構成

2ソケット、NUMA
(Non Uniform Memory Access)

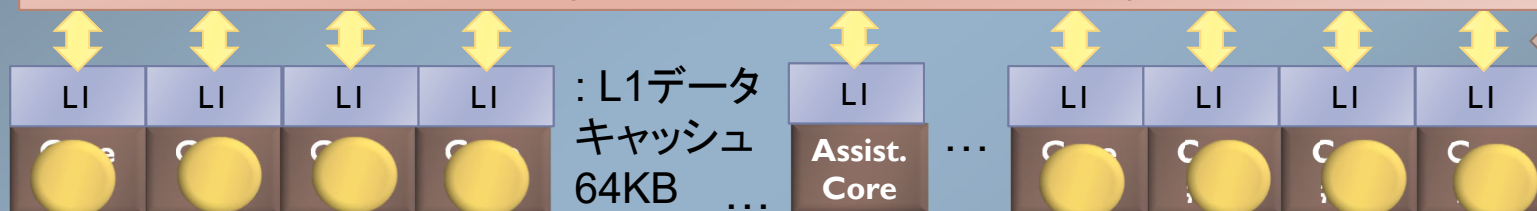
TOFU2
Network

HMC
16GB

MPIプロセス

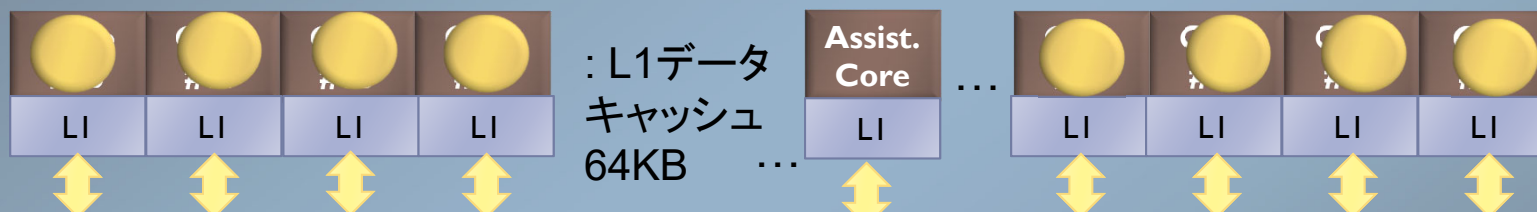
Memory

L2 (17コアで共有、12MB)



ソケット0 (CMG(Core Memory Group))

ICC



L2 (17コアで共有、12MB)

ソケット1 (CMC)

Memory

HMC
16GB

ノード内合計メモリ量: 32GB

読み込み: 240GB/秒
書き込み: 240GB/秒=合計: 480GB/秒

JOBスクリプトサンプルの説明 (ハイブリッドMP I)

(hello-hy16.bash, C言語、Fortran言語共通)

```
#!/bin/bash
#PJM -L "rscgrp=fx-debug"
#PJM -L "node=12"
#PJM --mpi "proc=12"
#PJM -L "elapse=1:00"
export OMP_NUM_THREADS=32
mpirun ./hello
```

リソースグループ名
:fx-debug

利用ノード数

利用コア数
(MPIプロセス数)

実行時間制限: 1分

1 MPIプロセス当たり
32スレッド生成

MPIジョブを $1 * 12 = 12$ プロセスで
実行する。

MPIプロセスのノード割り当て

- ▶ 名大FX100システムでは、何もしないと(デフォルトでは)、確保ノードが物理的に連続に確保されない
⇒通信性能が劣化する場合がある
- ▶ 物理的に連続したノード割り当てをしたい場合は、ジョブスクリプトにその形状を記載する
 - ▶ ただしノード割り当て形状を指定すると、待ち時間が増加する
- ▶ 記載法: **#PJM -L “node= <形状>:<機能>“**
 - ▶ **<形状> := { 1次元 | 2次元 | 3次元 }**
 - ▶ 1次元 := { a }, 2次元 := { a x b }, 3次元 := { a x b x c }
 - ▶ **<機能> := { 離散 | メッシュ | トーラス }**
 - ▶ 離散 := { noncont }, メッシュ := { mesh }, トーラス := { torus } : 12ノード以上
- ▶ 例: 24ノード、3次元(2x4x3)、トーラス
 - ▶ **#PJM -L “node= 2x4x3 : torus“**

NUMA指定について

- ▶ NUMA計算機では、MPIプロセスのソケットへの割り当てが性能面で重要となる
(**NUMA affinity**とよぶ)
- ▶ MPIプロセスのソケット(富士通用語でCMC)の割り当ては、FX100では富士通社のNUMA affinityで設定する
- ▶ 環境変数で設定する

NUMAメモリポリシー指定

▶ 環境変数名 : `plm_ple_memory_allocation_policy`

▶ 代入する値

- ▶ `localalloc`: プロセスが動作中のCPU(コア)の属するNUMAノードからメモリを割り当てる。
- ▶ `interleave_local`: プロセスの「ローカルノード集合」内の各NUMAノードから交互にメモリ割り当てる。
- ▶ `interleave_nonlocal`: プロセスの「非ローカルノード集合」内の各NUMAノードから交互にメモリ割り当てる。
- ▶ `interleave_all`: プロセスの「全ノード集合」内の各NUMAノードから交互にメモリを取得する。
- ▶ `bind_local`: プロセスの「ローカルノード集合」に属する各NUMAノードで、ノードIDの若い順にメモリ割り当てを行う。
- ▶ `bind_nonlocal`: プロセスの「非ローカルノード集合」に属する各NUMAノードで、ノードIDの若い順にメモリ割り当てを行う。
- ▶ `bind_all`: プロセスの「全ノード集合」のNUMAノードにバインドする。
- ▶ `prefer_local`: プロセスの「ローカルノード集合」のうち、NUMAノードIDが最も若いものを「優先ノード」とし、「優先ノード」からメモリ割り当てを行う。
- ▶ `prefer_nonlocal`: プロセスの「非ローカルノード集合」のうち、NUMAノードIDが最も若いものを「優先ノード」とし、「優先ノード」からメモリ割り当てを行う。

▶ 通常は、`localalloc`でよい。

▶ `export plm_ple_memory_allocation_policy=localalloc`

CPU(コア)割り当てポリシー指定

- ▶ 環境変数名 : `plm_ple_numanode_assign_policy`
- ▶ 代入する値
 - ▶ `simplex`: NUMAノードを占有するように割り当てる。
 - ▶ `share_cyclic`: NUMAノードを他のプロセスと共有するように割り当てる。異なるNUMAノードに順番にプロセスを割り当てる。
 - ▶ `share_band`: NUMAノードを他のプロセスと共有するように割り当てる。同一NUMAノードに連続してプロセスを割り当てる。
- ▶ 例) `export plm_ple_numanode_assign_policy=simplex`
- ▶ 各ソケットを各MPIプロセスで独占したいときは`simplex`を指定
 - ▶ 各ノードへ割り当てるMPIプロセス数が2個で、それぞれのMPIプロセスは16個のスレッド実行するとき
- ▶ MPIプロセスをプロセス順に各ソケットに詰め込みたいときは、`share_band`を指定
 - ▶ ノード当たり32個のMPIプロセスを、ランク番号が近い順に割り当てたい場合

その他の注意事項（その１）

- ▶ **MPI用のコンパイラを使うこと**
 - ▶ MPI用のコンパイラを使わないと、MPI関数が未定義というエラーが出て、コンパイルできなくなる
 - ▶ 例えば、以下のコマンド
 - ▶ Fortran90言語: **mpif90**
 - ▶ C言語: **mpicc**
 - ▶ C++言語: **mpixx, mpic++**
 - ▶ コンパイラオプションは、逐次コンパイラと同じ

その他の注意事項（その2）

▶ ハイブリッドMPIの実行形態

(MPIプロセス数) × (MPIプロセス当たりのOpenMPスレッド数)
≤ 利用コア総数

- ▶ HT(Intel)やSMT(IBM)などの、物理コア数の定数倍のスレッドが実行できるハードの場合（例えば、KNL）
 - ▶ スレッド数（論理スレッド数、例えばHTの数で、物理コアの4倍まで等）が、上記の利用コア総数に相当で換算
 - ▶ 以上を超えても実行できるが、性能が落ちる
- ▶ 必ずしも、1ノード内に1MPIプロセス実行が高速とはならない
 - ▶ 一般に、OpenMPによる台数効果が8～16スレッド（経験値、問題やハードウェア依存）を超えると悪くなるため。
 - 効率の良いハイブリッドMPI実行には、
効率の良いOpenMP実装が必須

MPI実行時のリダイレクトについて

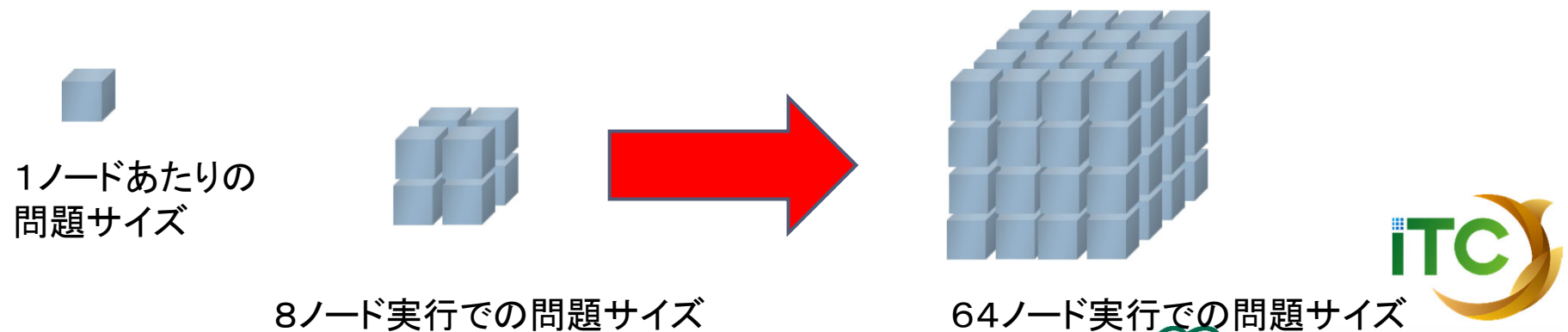
- ▶ 一般に、スーパーコンピュータでは、
MPI実行時の入出力のリダイレクトができません
 - ▶ × 例) `mpirun ./a.out < in.txt > out.txt`
- ▶ 専用のリダイレクト命令が用意されています。
- ▶ FX10 (FX100)でリダイレクトを行う場合、以下のオプションを指定します。
 - ▶ ○例) `mpirun --stdin ./in.txt --ofout out.txt ./a.out`

並列処理の評価指標： 弱スケーリングと強スケーリング



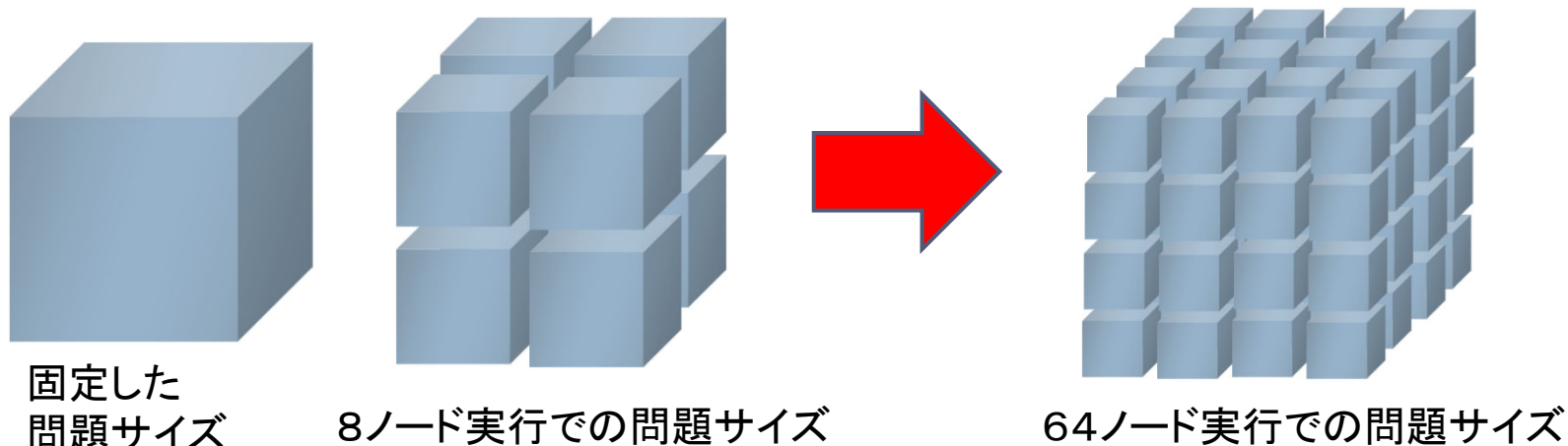
弱スケーリング (Weak Scaling)

- ▶ ノードあたりの問題サイズを固定し、並列処理時の全体の問題サイズを増加することで、性能評価をする方法
- ▶ 問題サイズ N ときの計算量が $O(N)$ である場合、並列処理のノード数が増加しても、**理想的な実行時間は変わらないと期待できる**
 - ▶ 一般的にノード数が増加すると(主にシステムの要因により)通信時間が増大するため、そうはならない
 - ▶ 該当する処理は
 - ▶ 陽解法のシミュレーション全般
 - ▶ 陰解法で、かつ連立一次方程式の解法に反復解法を用いているシミュレーション



強スケーリング (Strong Scaling)

- ▶ 全体の問題サイズを固定し、ノード数を増加することで性能評価をする方法
- ▶ 理想的な実行時間は、ノード数に反比例して減少する。
 - ▶ 一般的にノード数が増加すると1ノードあたりの問題サイズが減少し、通信時間の占める割合が増大するため、理想的に実行時間は減少しない
 - ▶ 該当する処理は
 - ▶ 計算量が膨大なアプリケーション
 - ▶ 例えば、連立一次方程式の解法。データ量 $O(N^2)$ に対して、計算量は $O(N^3)$



弱スケーリングと強スケーリング 適用アプリの特徴

- ▶ 弱スケーリングが適用できるアプリケーションは、
原理的に通信が少ないアプリケーション
 - ▶ 領域分割法などにより、並列化できるアプリケーション
 - ▶ 主な通信は、隣接するプロセス間のみ
 - ▶ ノード数を増すことで、実行時間の面で容易に問題サイズを大規模化
 - ▶ 通信時間の占める割合が超並列実行でも少ないアプリケーション
- ▶ 強スケーリングを適用しないといけないアプリケーションは、
計算量が膨大になるアプリケーション
 - ▶ 全体の問題サイズは、実行時間の制約から大規模化できない
 - ▶ そのため、1ノードあたりの問題サイズは、ノード数が多い状況で小さくなる
 - ▶ その結果、通信処理の占める時間がほとんどになる
 - ▶ 超並列実行時で通信処理の最適化が重要になるアプリケーション



強スケールアプリケーションの問題

▶ TOP500で採用されているLINPACK

- ▶ 密行列に対する連立一次方程式の解法のアプリケーション
- ▶ 2018年11月のTOP500の、コア当たりの問題サイズ
- ▶ (1位) Summit、GPUマシン
 $N = 16,693,248$ (1669万次元)、 $\#cores = 2,397,824$ (239万コア)、 $N/\#cores = 6.98$
- ▶ (3位) Sunway TaihuLight、CPUマシン
 $N = 12,288,000$ (1228万次元)、 $\#cores = 10,649,600$ (1064万コア)、 $N/\#cores = 1.15$
- ▶ (4位) Tianhe-2A、CPUマシン
 $N = 9,773,000$ 、 $\#cores = 4,981,760$ 、 $N/\#cores = 1.96$
- ▶ (14位) Oakforest-PACKS、CPUマシン(メニーコア)
 $N = 9,938,880$ 、 $\#cores = 556,104$ 、 $N/\#cores = 17.8$
- ▶ 上位のマシン、および、CPUマシンほど、コア当たりの問題サイズが小さい傾向がある ← 通信時間の占める割合が大きくなりやすい
- ▶ 今後コア数が増加すると、通信時間の削減が問題になる



ピュアMPIプログラム開発 の基礎



MPI並列化の大前提（再確認）

▶ SPMD

- ▶ 対象のメインプログラムは、
 - ▶ **すべてのコア上で、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

- ▶ 各プロセスは、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**）

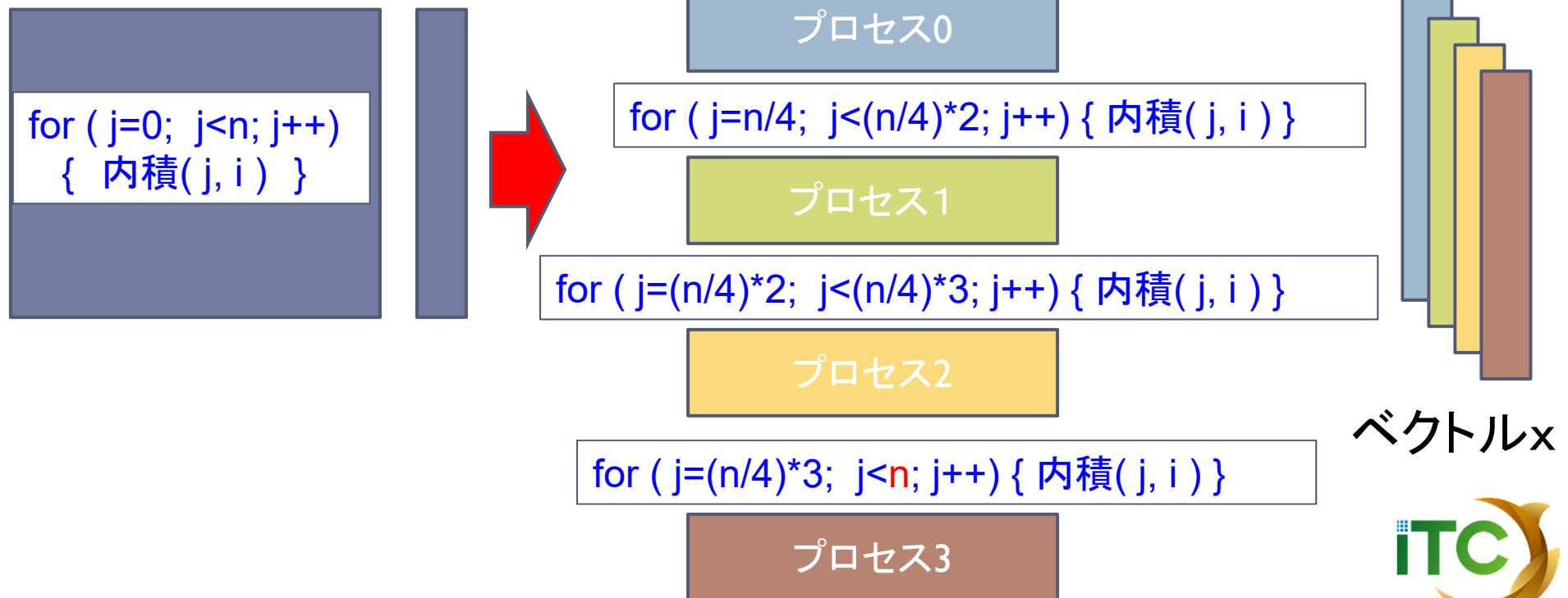


並列化の考え方（C言語）

▶ SIMDアルゴリズムの考え方（4プロセスの場合）

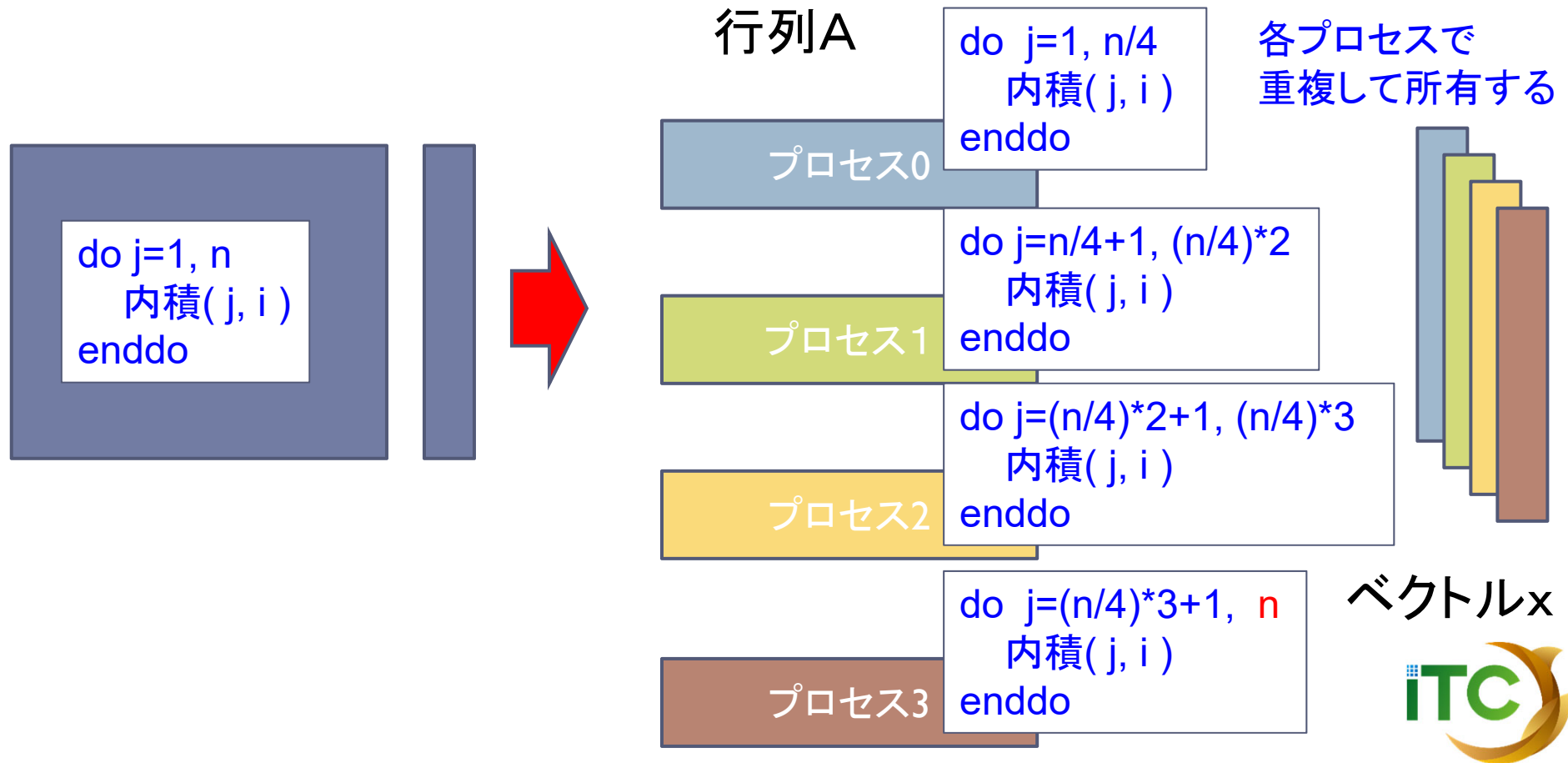
行列A

各PEで
重複して
所有する



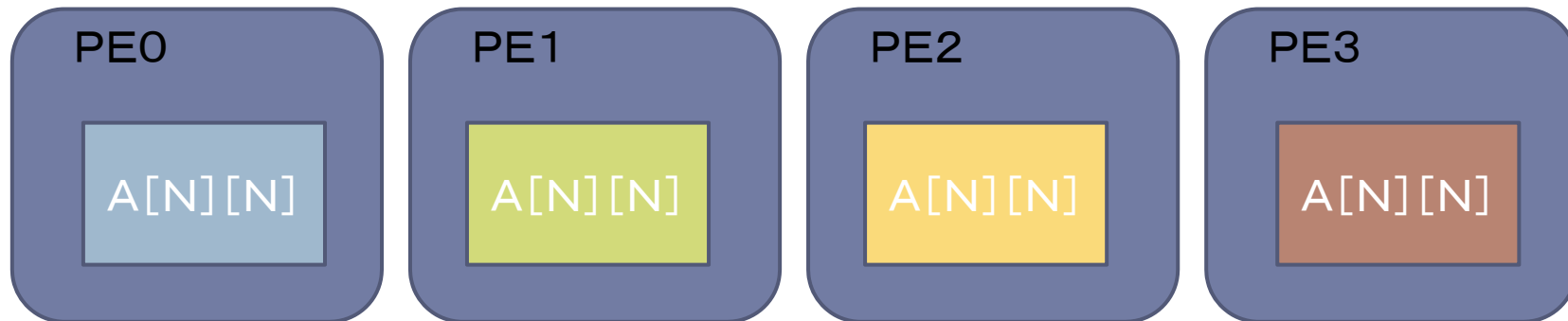
並列化の考え方 (Fortran言語)

▶ SIMDアルゴリズムの考え方(4プロセスの場合)

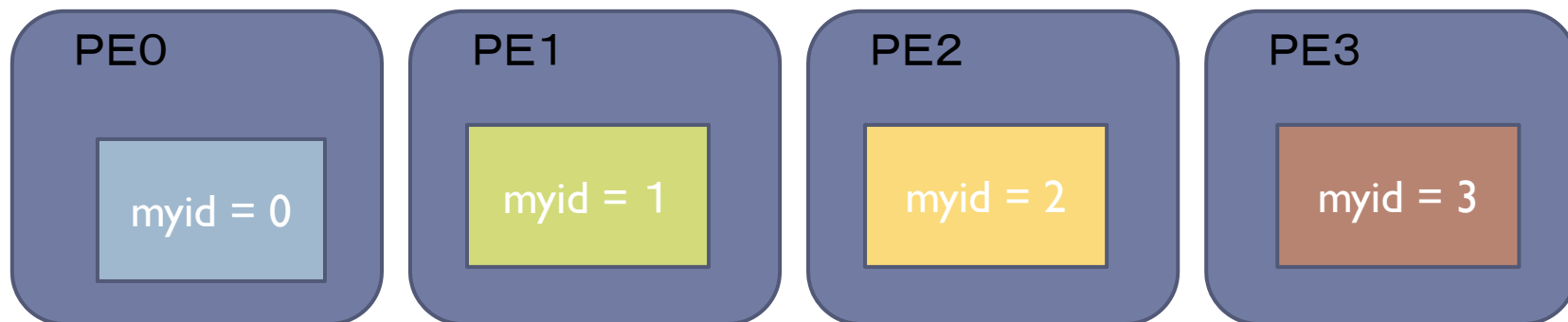


初心者が注意すること

- ▶ 各プロセスでは、**独立した配列が個別に確保**されます。



- ▶ myid変数は、MPI_Init()関数が呼ばれた段階で、**各プロセス固有の値**になっています。



並列プログラム開発の指針

1. 正しく動作する逐次プログラムを作成する
2. 1. のプログラムで、適切なテスト問題を作成する
3. 2. のテスト問題の実行について、適切な処理の単位ごとに、正常動作する計算結果を確認する
4. 1. の逐次プログラムを並列化し、並列プログラミングを行う
5. 2. のテスト問題を実行して動作検証する
6. このとき3. の演算結果と比較し、正常動作をすることを確認する。もし異常であれば、4. に戻りデバックを行う。

数値計算プログラムの特徴を利用して 並列化時のデバックをする

- ▶ 数値計算プログラムの処理単位は、プログラム上の基本ブロック(ループ単位など)ではなく、数値計算上の処理単位(数式レベルで記述できる単位)となる
 - ▶ 離散化(行列作成)部分、行列分解部分(LU分解法部分(LU分解部分、前進代入部分、後退代入部分))、など
- ▶ 演算結果は、なんらかの数値解析上の意味において検証
 - ▶ 理論解(解析解)とどれだけ離れているか、考えられる丸め誤差の範囲内にあるか、など
 - ▶ 計算された物理量(例えば流速など)が物理的に妥当な範囲内にあるか、など
 - ▶ 両者が不明な場合でも、数値的に妥当であると思われる逐次の結果と比べ、並列化した結果の誤差が十分に小さいか、など

並列化の方針

(行列-ベクトル積、 C 言語)

1. 全プロセスで行列Aを $N \times N$ 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. 各プロセスは、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)各プロセス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

(行列-ベクトル積、Fortran言語)

1. 全プロセスで行列Aを $N \times N$ 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. 各プロセスは、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)各プロセス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。

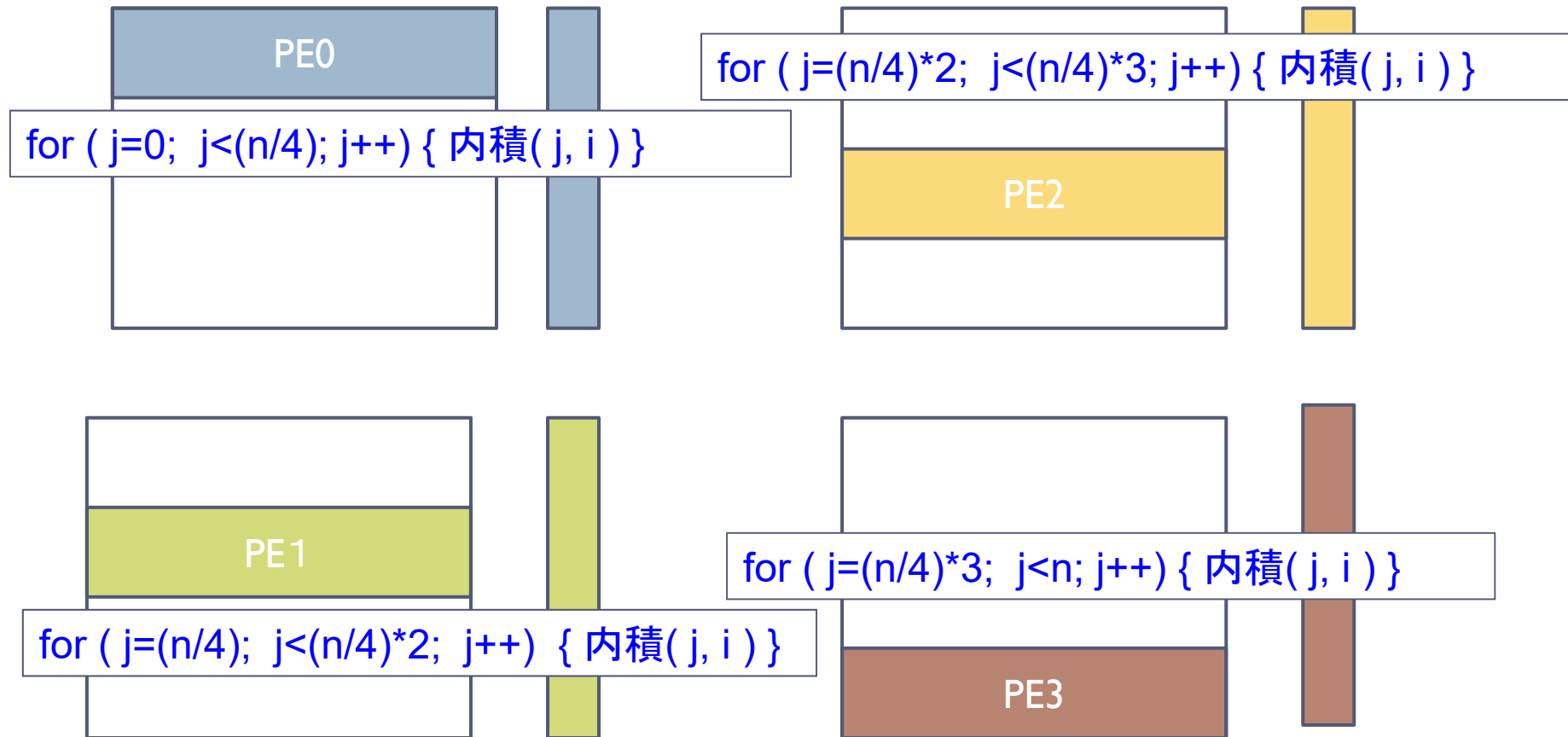
```
do j=1, ib ... enddo
```

データ分散方式に関する注意

- ▶ 負荷分散を考慮し、多様なデータ分散方式を採用可能
- ▶ 数学的に単純なデータ分散方式が良い
 - ▶ ◎: ブロック分散、サイクリック分散 (ブロック幅 = 1)
 - ▶ △ ~ ○: ブロック・サイクリック分散 (ブロック幅 = 任意)
 - ▶ 理由:
 - ▶ 複雑な (一般的な) データ分散は、各 MPI プロセスが所有するデータ分散情報 (インデックスリスト) を必要とするため、メモリ量が余分に必要なる
 - 例: 1 万 並列では、少なくとも 1 万次元の整数配列が必要
 - 数学的に単純なデータ分散の場合は、インデックスリストは不要
 - ローカルインデックス、グローバルインデックスが計算で求まるため

並列化の方針（行列-ベクトル積） （C言語）

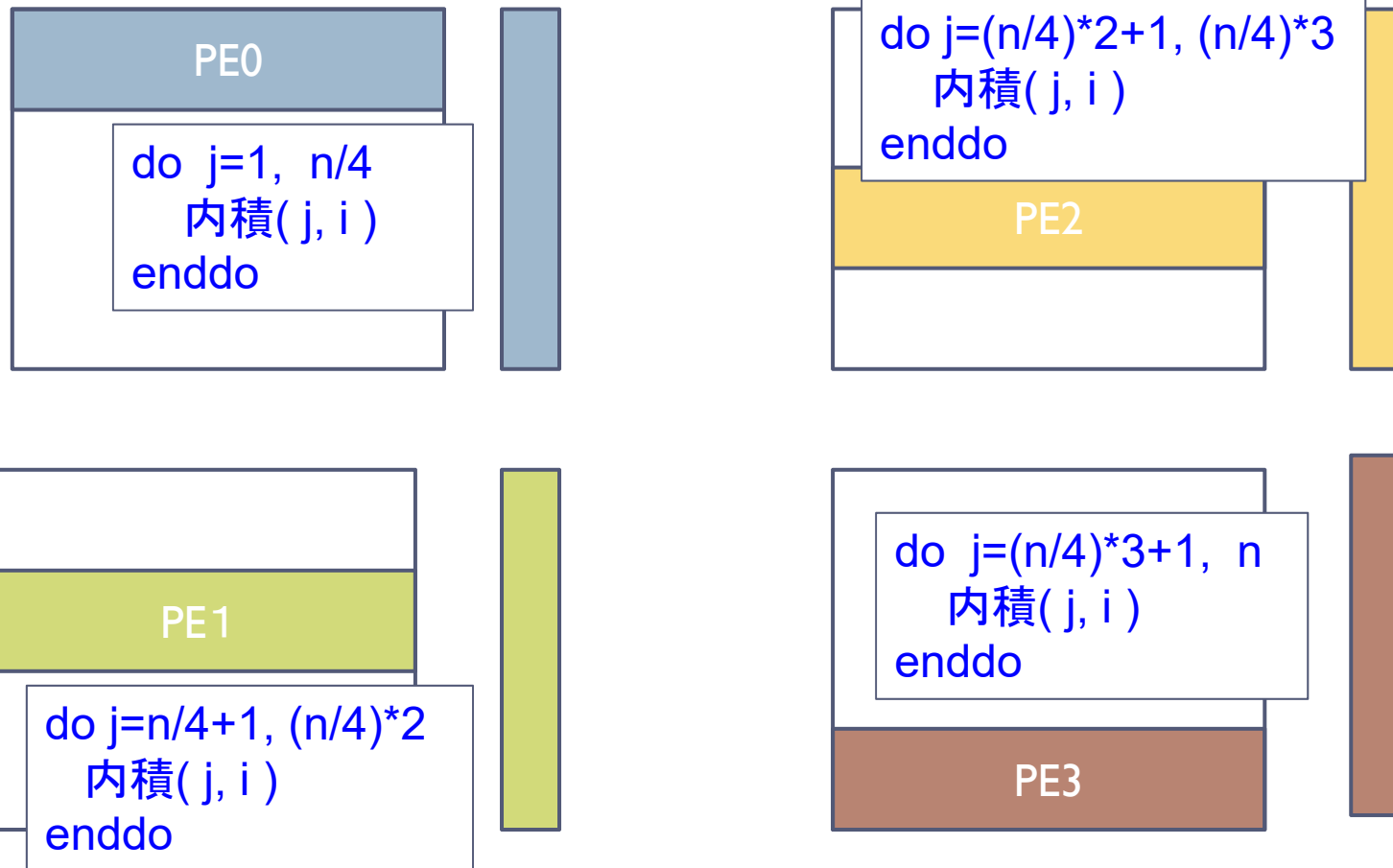
▶ 全PEで $N \times N$ 行列を持つ場合



※各PEで使われない領域が出るが、担当範囲指定がしやすいので実装がしやすい。

並列化の方針（行列-ベクトル積） （Fortran 言語）

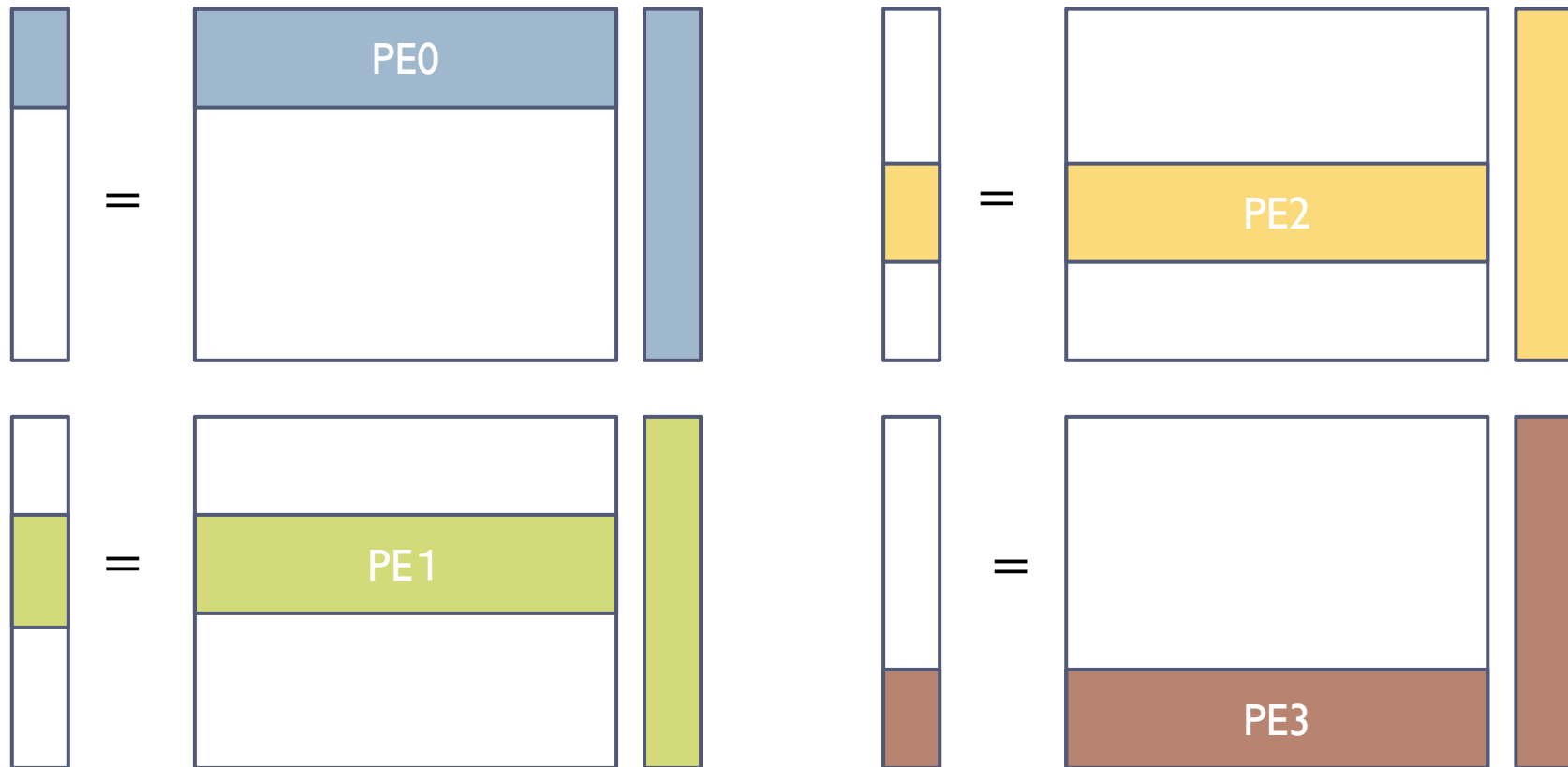
▶ 全PEで $N \times N$ 行列を持つ場合



※各PEで使われない領域が出るが、担当範囲指定がしやすいので実装がしやすい。

並列化の方針（行列-ベクトル積）

- ▶ この方針では、 $y = Ax$ のベクトル y は、以下のように一部分しか計算されないことに注意！



並列化の方針のまとめ

- ▶ 行列全体 ($A[N][N]$) を各プロセスで確保することで、SIMDの考え方を、逐次プログラムに容易に適用できる
 - ▶ ループの開始値、終了値のみ変更すれば、並列化が完成する
 - ▶ この考え方は、MPI、OpenMPに依存せず、適用できる。
 - ▶ 欠点
 - ▶ 最大実行可能な問題サイズが、利用ノード数によらず、1ノードあたりのメモリ量で制限される(メモリに関するスケーラビリティが無い)
- ▶ ステップ3のデバックの困難性を低減できる
 - ▶ 完全な並列化(ステップ3)の際、ステップ2での正しい計算結果を参照できる
 - ▶ 数値計算上の処理単位ごとに、個別に並列化ができる(モジュールごとに、デバックできる)



行列 - ベクトル積のピュアMPI並列化の例 (C言語)

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

```
ib = n/numprocs;  
jstart = myid * ib;  
jend = (myid+1) * ib;  
if ( myid == numprocs-1 ) jend=n;
```

ブロック分散を仮定した
担当ループ範囲の定義

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

MPIプロセスの担当ごとに
縮小したループの構成

行列 - ベクトル積のピュアMPI並列化の例 (Fortran言語)

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

```
...
```

```
ib = n/numprocs
jstart = 1 + myid * ib
jend = (myid+1) * ib
if ( myid .eq. numprocs-1 ) jend = n
```

ブロック分散を仮定した
担当ループ範囲の定義

```
do j = jstart, jend
  y(j) = 0.0d0
  do i=1, n
    y(j) = y(j) + A(j,i) * x(i)
  enddo
enddo
```

MPIプロセスの担当ごとに
縮小したループの構成

nがMPIプロセス数で割切れない時

- ▶ nがプロセス数のnumprocsで割り切れない場合
 - ▶ 配列確保: $A(N/\text{numprocs} + \text{mod}(N, \text{numprocs}), N)$
 - ▶ ループ終了値: numprocs-1のみ終了値がnとなるように実装

```
ib = n / numprocs;  
if ( myid == (numprocs - 1) ) {  
    i_end = n;  
} else {  
    i_end = (myid+1)*ib;  
}  
for ( i=myid*ib; i<i_end; i++) { ... }
```

余りが多い場合

- ▶ $\text{mod}(N, \text{numprocs})$ が大きいと、負荷バランスが悪化
 - ▶ 例: $N=10$ 、 $\text{numprocs}=6$
 - $\text{int}(10/6)=1$ なので、
プロセス0～5は1個のデータ、プロセス6は4個のデータを持つ
 - ▶ 各プロセスごとの開始値、終了値のリストを持てば改善可能
 - プロセス0: $i_{\text{start}}(0)=1$, $i_{\text{end}}(0)=2$, 2個
 - プロセス1: $i_{\text{start}}(1)=3$, $i_{\text{end}}(1)=4$, 2個
 - プロセス2: $i_{\text{start}}(2)=5$, $i_{\text{end}}(2)=6$, 2個
 - プロセス3: $i_{\text{start}}(3)=7$, $i_{\text{end}}(3)=8$, 2個
 - プロセス4: $i_{\text{start}}(4)=9$, $i_{\text{end}}(4)=9$, 1個
 - プロセス5: $i_{\text{start}}(5)=10$, $i_{\text{end}}(5)=10$, 1個
 - ▶ 欠点: プロセス数が多いと、上記リストのメモリ量が増える

ハイブリッドMPIプログラム開発 の基礎



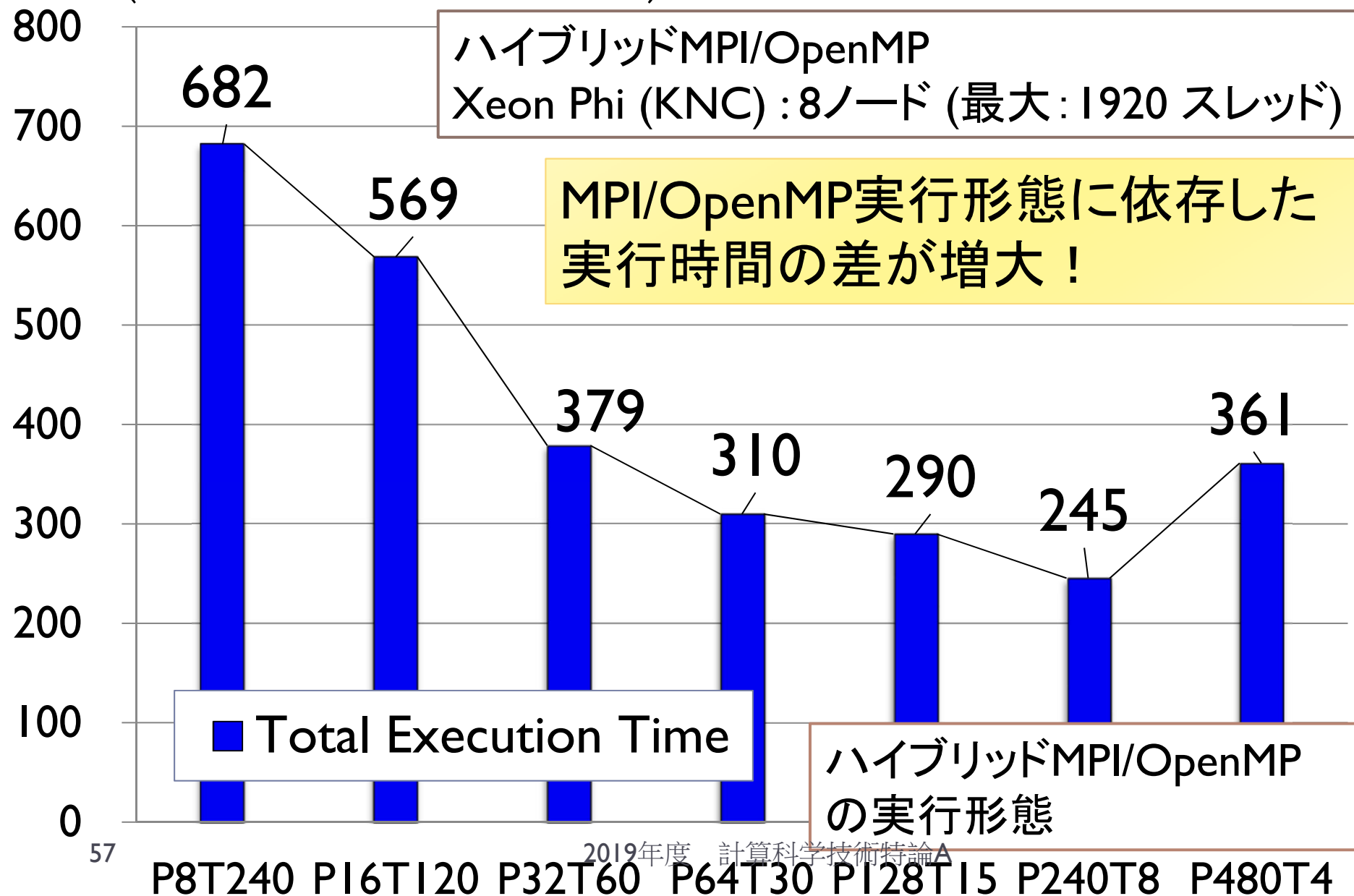
用語の説明

- ▶ **ピュアMPI実行**
 - ▶ 並列プログラムでMPIのみ利用
 - ▶ MPIプロセスのみ
- ▶ **ハイブリッドMPI実行**
 - ▶ 並列プログラムでMPIと何か(X(エックス))を利用
 - ▶ MPIプロセスと何か(X)の混合
 - ▶ 何か(X)は、OpenMPによるスレッド実行、もしくは、GPU実行が主流
- ▶ **「MPI+X」の実行形態**
 - ▶ 上記のハイブリッドMPI実行と同義として使われる
 - ▶ Xは、OpenMPや自動並列化によるスレッド実行、CUDAなどのGPU向き実装、OpenACCなどのGPUやメニーコア向き実行、などの組合せがある。主流となる計算機アーキテクチャで変わる。

ハイブリッドMPI実行の目的

- ▶ 同一の資源量（総コア数）の利用に対し
 - ▶ ピュアMPI実行でのMPIプロセス数に対し、ハイブリッドMPI実行でMPIプロセス数を減らすことで、通信時間を削減することが主な目的
- ▶ 例) JCAHPC（東大・筑波大）の Oakforest-PACS
 - ▶ 全系は8,208ノード、558,144（55万）物理コア
 - ▶ ピュアMPI実行（HT4利用時）
: 2,232,576（223万）プロセス実行
 - ▶ ハイブリッドMPI実行（HT4利用時）：
（1ノード272スレッド実行）: 8,208プロセス
 - ▶ MPIプロセス数の比は272倍！

ハイブリッドMPI/OpenMP実行の実例 [秒] (ある有限差分法のアプリ)



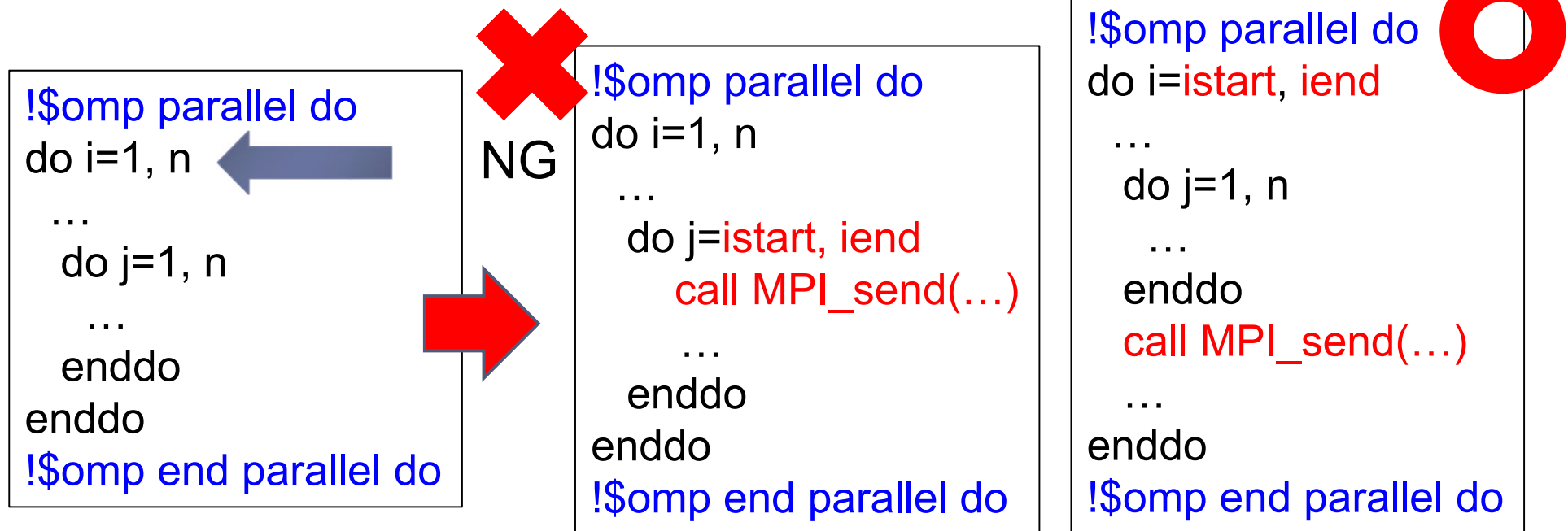
ハイブリッドMPI/OpenMP並列プログラム開発の指針

1. 正しく動作するピュアMPIプログラムを開発する
2. OpenMPを用いて対象カーネルをスレッド並列化する
3. 2. の性能評価をする
4. 3. の評価結果から性能が不十分な場合、対象カーネルについてOpenMPを用いた性能チューニングを行う。
3. へ戻る。
5. 全体性能を検証し、通信時間に問題がある場合、通信処理のチューニングを行う。



ハイブリッドMPI/OpenMP並列化の方針 (OpenMPプログラムがある場合)

- ▶ すでに開発済みのOpenMPプログラムを元にMPI化する場合
- ▶ OpenMPのparallelループをMPI化すること
- ▶ OpenMPループ中にMPIループを記載すると通信多発で遅くなるか、最悪、動作しない



行列 - ベクトル積の ハイブリッドMPI並列化の例（C言語）

```
ierr = MPI_Init(&argc, &argv);
ierr = MPI_Comm_rank(MPI_COMM_WORLD, &myid);
ierr = MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
...
ib = n/numprocs;
jstart = myid * ib;
jend = (myid+1) * ib;
if ( myid == numprocs-1 ) jend=n;
#pragma omp parallel for private(i)
for( j=jstart; j<jend; j++) {
    y[ j ] = 0.0;
    for(i=0; i<n; i++) {
        y[ j ] += A[ j ][ i ] * x[ i ];
    }
}
```

ブロック分散を仮定した
担当ループ範囲の定義

この一文を追加するだけ！

MPIプロセスの担当ごとに
縮小したループの構成

行列 - ベクトル積の ハイブリッドMPI並列化の例 (Fortran言語)

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

...

```
ib = n/numprocs
jstart = 1 + myid * ib
jend = (myid+1) * ib
if ( myid .eq. numprocs-1 ) jend = n
```

```
!$omp parallel do private(i)
```

```
do j = jstart, jend
```

```
  y(j) = 0.0d0
```

```
  do i=1, n
```

```
    y(j) = y(j) + A(j,i) * x(i)
```

```
  enddo
```

```
enddo
```

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

ブロック分散を仮定した
担当ループ範囲の定義

この文を追加するだけ！

MPIプロセスの担当ごとに
縮小したループの構成

ハイブリッドMPI/OpenMP実行の注意点 (その1)

- ▶ ハイブリッドMPI/OpenMP実行では、MPIプロセス数に加えて、スレッド数がチューニングパラメタとなり、複雑化する。

- ▶ 例) 1ノード16コア実行

2MPIプロセス、8スレッド実行



4MPIプロセス、4スレッド実行




1つのMPI
プロセス
の割り当て
対象

- ▶ ccNUMAの計算機では、ソケット数ごとに1MPIプロセス実行が高速となる可能性がある(ハードウェア的に)

- ▶ 例) Fujitsu PRIMEHPC、2ソケット、32コア

2MPIプロセス、16スレッド実行



ハイブリッドMPI/OpenMP実行の注意点 (その2)

- ▶ ハイブリッドMPI/OpenMP実行の実行効率を決める要因
 1. ハイブリッドMPI化による通信時間の削減割合
 2. OpenMP等で実現される演算処理のスレッド実行効率
- ▶ 特に、2は注意が必要。
 - ▶ 単純な実装だと、【経験的に】8~16スレッド並列を超えると、スレッド実行時の台数効果が劇的に悪くなる。
 - ▶ 効率の良いスレッド並列化の実装をすると、ハイブリッドMPI/OpenMP実行時に効果がより顕著になる。
 - ▶ 実装の工夫が必要。たとえば
 1. ファーストタッチ(すでに説明済み)の適用
 2. メモリ量や演算量を増加させても、スレッドレベルの並列性を増加させる
 3. アンローリングなどの逐次高速化手法を、スレッド数に特化させる

ハイブリッドMPI/OpenMP実行の注意点 (その3)

- ▶ 通信処理の時間に含まれる、データのコピー時間が、通信時間よりも大きいことがある
 - ▶ 問題空間の配列から送信用の配列にコピーする処理
(**パッキング**)
 - ▶ 受信用の配列から問題空間の配列へコピーする処理
(**アンパッキング**)
 - ▶ 上記のコピー量が多い場合、コピー操作自体もOpenMP化すると高速化される場合がある。
 - ▶ 特に、強スケーリング時
 - ▶ 問題サイズやハードウェアによっては、OpenMP化すると遅くなる。
このときは、逐次処理にしないといけない。
- ▶ **パッキング、アンパッキングをOpenMP化する／しない、もハイブリッドMPI実行では重要なチューニング項目になる**

ハイブリッドMPI/OpenMPの起動方法

- ▶ スパコンごとに異なるが、以下の方法が主流（すでに説明済み）。
 1. バッチジョブシステムを通して、MPIの数を指定
 2. 実行コマンドで、OMP_NUM_THREADS環境変数でスレッド数を指定
- ▶ ccNUMAの場合、MPIプロセスの割り当てを、期待する物理ソケットに割り当てないと、ハイブリッドMPI実行の効果が無くなる
 - ▶ Linuxでは、numactlコマンドで実行時に指定する
 - ▶ スパコン環境によっては、プロセスを指定する物理コアに割り当て方法がある。
(各スパコンの利用マニュアルを参考)



数値計算ライブラリとハイブリッドMPI実行

- ▶ 数値計算ライブラリのなかには、ハイブリッドMPI実行をサポートしているものがある
 - ▶ 数値計算ライブラリがスレッド並列化されている場合
 - ▶ 特に、密行列用ライブラリのScaLAPACKは、
通常、ハイブリッドMPI実行をサポート
 - ▶ ScaLAPACKは、MPI実行をサポート
 - ▶ ScaLAPACKは、逐次のLAPACKをもとに構築
 - ▶ LAPACKは基本数値計算ライブラリBLASをもとに構築
 - ▶ BLASは、スレッド実行をサポート
- ⇒BLASレベルのスレッド実行と、
ScaLAPACKレベルのMPI実行を基にした
ハイブリッドMPI実行が可能

スレッド並列版BLAS利用の注意

- ▶ BLASライブラリは、OpenMPスレッド並列化がされている
- ▶ 利用方法は、OpenMPを用いた並列化と同じ
 - ▶ OMP_NUM_THREADSで並列度を指定
- ▶ **BLASで利用するスレッド数が利用可能なコア数を超えると動かないか、動いたとしても速度が劇的に低下する**
- ▶ BLASを呼び出す先がスレッド並列化をしている場合、BLAS内でスレッド並列化をすると、総合的なスレッド数が、利用可能なコア数を超えることがある。このため、速度が劇的に低下する。
- ▶ **一般的に、逐次実行の演算効率が、OpenMPスレッド並列の実行効率に比べて、高い**
 - ▶ 上位のループをOpenMPスレッド並列化し、そのループから逐次BLASを呼び出す実装がよい



逐次BLASをスレッド並列化して呼び出す例

▶ 通常のBLASの呼び出し

```
do i=1, Ak  
  call dgemm(...) ←スレッド並列版BLASを呼び出し  
                    (コンパイラオプションで指定)  
enddo
```

▶ 上位のループでOpenMP並列化したBLASの呼び出し

```
!$omp parallel do  
do i=1, Ak  
  call dgemm(...) ←逐次BLASを呼び出し  
                    (コンパイラオプションで指定)  
enddo  
!$omp end parallel do
```

<スレッド並列版BLAS>と<逐次BLASを上位の ループでスレッド並列呼び出し>する時の性能例

▶ T2Kオープンスパコン(東大版)

- ▶ AMD Quad Core Opteron
- ▶ 1ノード(16コア)を利用
- ▶ 日立製作所によるCコンパイラ(日立最適化C)
- ▶ OpenMP並列化を行った
 - ▶ 最適化オプション: “-Os -omp”
- ▶ BLAS
 - ▶ GOTO BLAS ver.1.26
(スレッド並列版, および逐次版の双方)

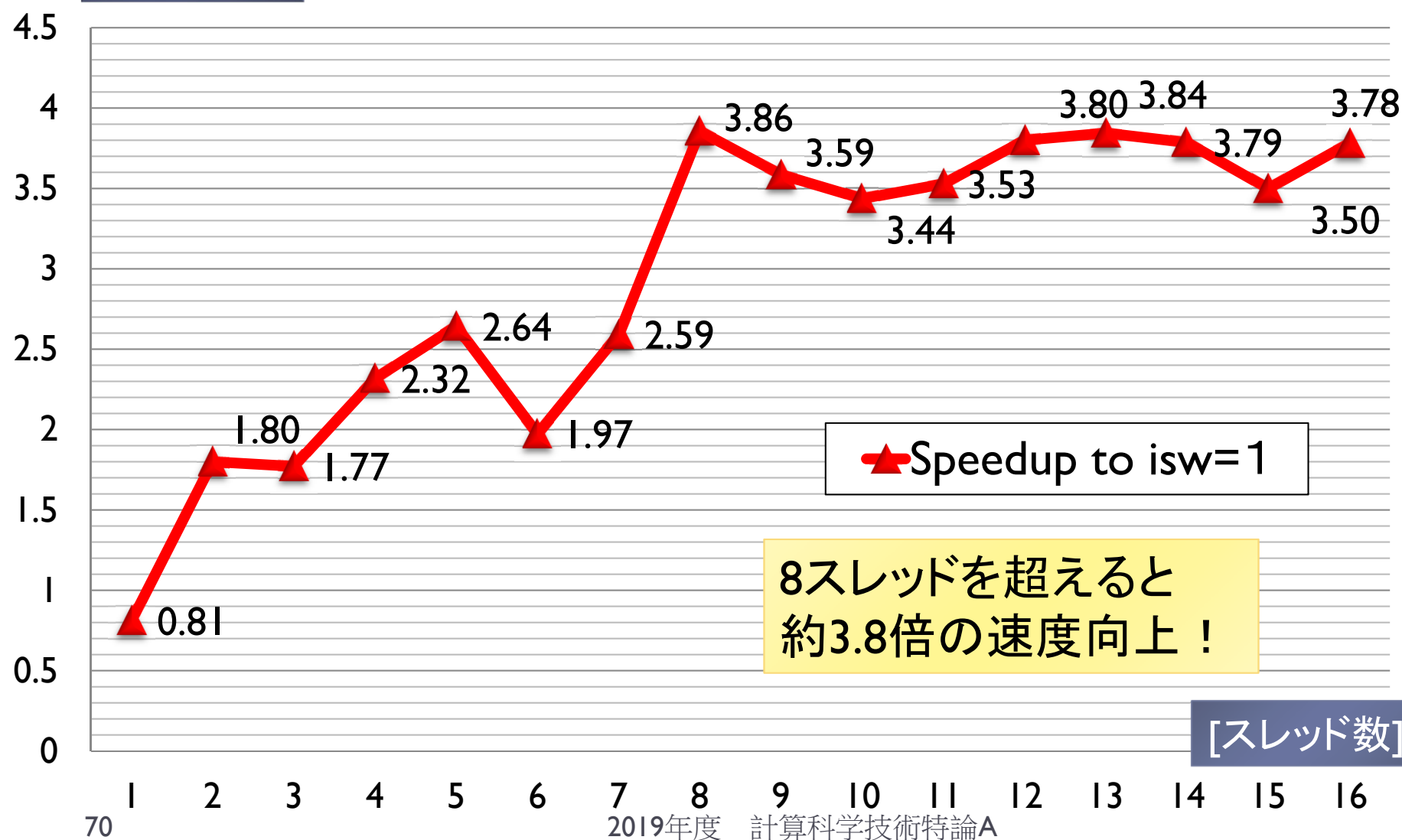
▶ 対象処理

- ▶ 高精度行列 - 行列積の主計算
- ▶ 複数の行列 - 行列積(dgemm呼び出し)を行う部分



n=1000での性能（T2K(1ノード, 16コア）） BLAS内でスレッド並列化する場合に対する速度向上

[速度向上]



[スレッド数]

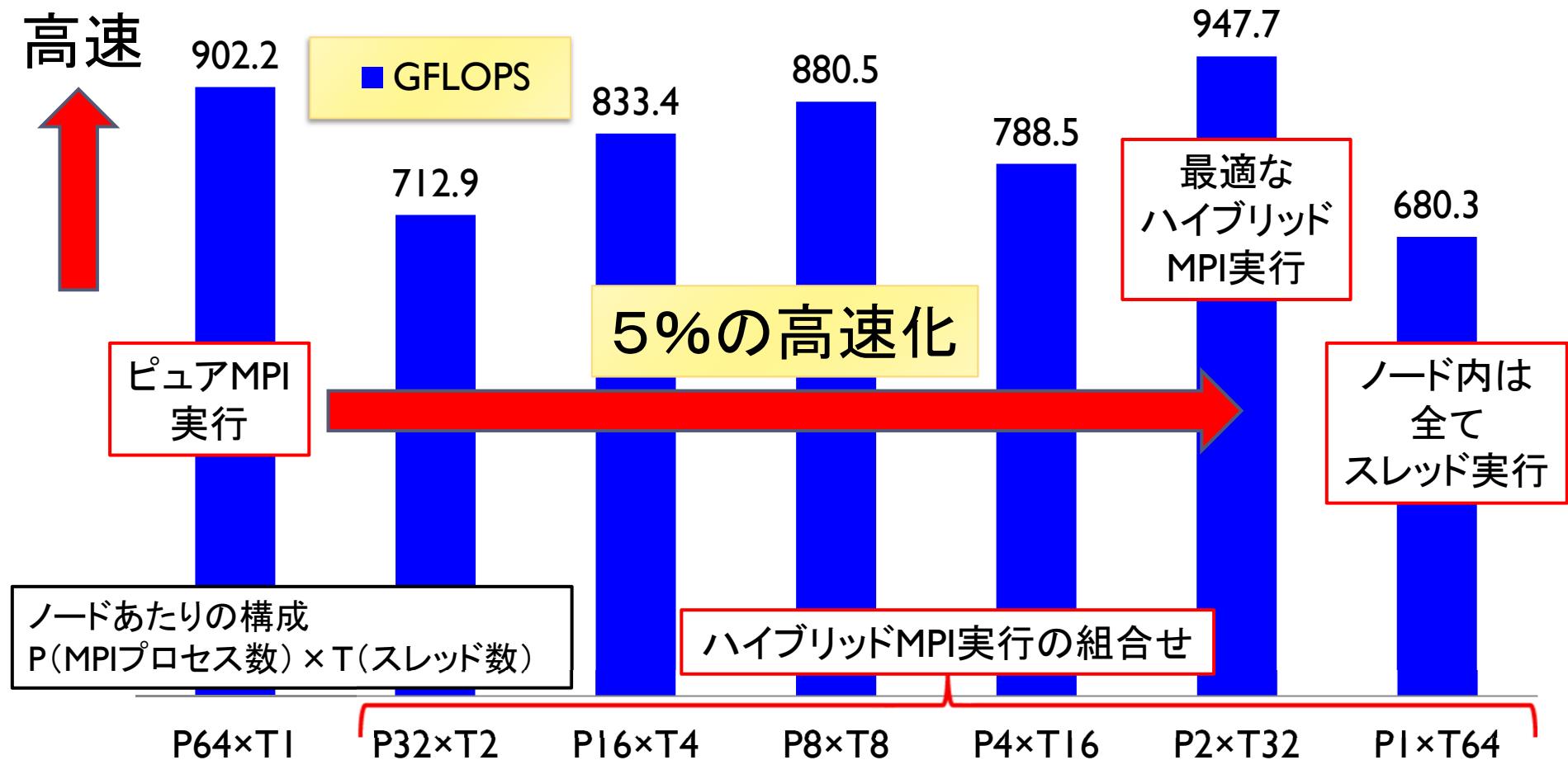
ScaLAPACKにおける ハイブリッドMPI実行の効果の例

- ▶ ScaLAPACKの連立一次方程式解法ルーチン
PDGESV
- ▶ 東京大学情報基盤センターのHITACHI SR16000
 - ▶ IBM Power7 (3.83GHz)
 - ▶ 1ノード4ソケット、1ソケットあたり8コア、合計32コア、
980.48GFLOPS／ノード
 - ▶ SMT利用で、1ノード64論理スレッドまで利用可能
 - ▶ ScaLAPACKは、同環境で提供されているIBM社の
ESSL(Engineering and Scientific Subroutine Library)
ライブラリを利用



ScaLAPACKにおける ハイブリッドMPI実行の効果の例

SR16000の2ノードでの実行 (問題サイズ $N=32,000$)



コンパイラ最適化の影響（その1）

- ▶ MPI化、および、OpenMP化に際して、**ループ構造を逐次から変更**することになる
- ▶ この時、コンパイラに依存し、コード最適化が並列ループに対して、効かない(遅い)コードを生成することがある
- ▶ 上記の場合、逐次実行での効率に対して、並列実行での効率が低下し、**台数効果の向上を制限する**
- ▶ たとえば、**ループ変数に大域変数を記載すると、コンパイラの最適化を阻害することがある**
 - ▶ 特に並列処理制御変数である、**全体のMPIプロセス数を管理する変数、自分のランク番号を管理する変数は、大域変数であることが多いので注意。**



コンパイラ最適化の影響（その2）

- ▶ MPI並列コードで、ループに大域変数を使っている例

C言語の例

```
ib = n/numprocs;  
for( j= myid * ib; j<(myid+1) * ib; j++) {  
    y[ j ] = 0.0;  
    for(i=0; i<n; i++) {  
        y[ j ] += A[ j ][ i ] * x[ i ];  
    }  
}
```

Fortran言語の例

```
ib = n/numprocs  
do j = 1 + myid * ib, (myid+1) * ib  
    y( j ) = 0.0d0  
    do i=1, n  
        y( j ) = y( j ) + A( j, i ) * x( i )  
    enddo  
enddo
```

- ▶ 上記のmyidは大域変数で、自ランク番号を記憶している変数
- ▶ コンパイラがループ特徴を把握できず、最適化を制限
 - ▶ ←逐次コードに対して、演算効率が低下し、台数効果を制限
- ▶ 解決策：局所変数を宣言しmyidを代入。対象を関数化。



ハイブリッドMPIプログラミングのまとめ

- ▶ ノード数が増えるほど、ピュアMPI実行に対する効果が増加
 - ▶ 経験的には、1000MPIプロセスを超える実行で、ハイブリッドMPI実行が有効となる
 - ▶ 現状での効果はアプリケーションに依存するが、経験的には数倍(2~3倍)高速化される
 - ▶ 現在、多くの実例が研究されている
 - ▶ エクサに向けて10万並列を超える実行では、おそらく数十倍の効果期待される
- ▶ ノードあたりの問題サイズが小さいほど、ハイブリッドMPI実行の効果が増大
 - ▶ 弱スケーリングより強スケーリングのほうがハイブリッドMPI実行の効果がある



レポート課題（その1）

▶ 問題レベルを以下に設定

問題のレベルに関する記述:

- L00: きわめて簡単な問題。
- L10: ちょっと考えればわかる問題。
- L20: 標準的な問題。
- L30: 数時間程度必要とする問題。
- L40: 数週間程度必要とする問題。複雑な実装を必要とする。
- L50: 数か月程度必要とする問題。未解決問題を含む。

※L40以上は、論文を出版するに値する問題。

▶ 教科書のサンプルプログラムは以下が利用可能

- ▶ Samples-fx.tar
- ▶ Mat-Vec-fx.tar
- ▶ PowM-fx.tar
- ▶ Mat-Mat-fx.tar
- ▶ Mat-Mat-d-fx.tar
- ▶ LU-fx.tar



レポート課題（その2）

- I. [L20] 使える並列計算機環境で、教科書のサンプルプログラムを並列化したうえで、
ピュアMPI実行、および、ハイブリッドMPI実行で
性能が異なるか、実験環境（たとえば、12ノード、384コア）
を駆使して、性能評価せよ。
 - ▶ 1ノードあたり、12MPI実行、1MPI+32スレッド実行、2MPI+16スレッド実行、4MPI+8スレッド実行など、組み合わせが多くある。

レポート課題（その3）

2. [L10] ハイブリッドMPI実行がピュアMPI実行に対して有効となるアプリケーションを、論文等で調べよ。
3. [L20～] 自分が持っている問題に対し、ハイブリッドMPI実行ができるようにプログラムを作成せよ。また、実験環境を用いて、性能評価を行え。