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

第４講　Hybrid並列化技法 (MPIとOpenMPの応用)

東京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

- 2015年9月12日（土） 第1回並列プログラミング講習会
  座学「並列プログラミング入門」in 金沢
  - 第1講：プログラム高速化の基礎、10:30-12:00
    - 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講：並列処理とMPIの基礎、13:00-14:30
    - 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講：OpenMPの基礎、14:45-16:15
    - OpenMPの基礎、利用方法、その他
  - 第4講：Hybrid並列化技法(MPIとOpenMPの応用)、16：30-18:00
    - 背景、Hybrid並列化の適用事例、利用上の注意、その他
    - 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

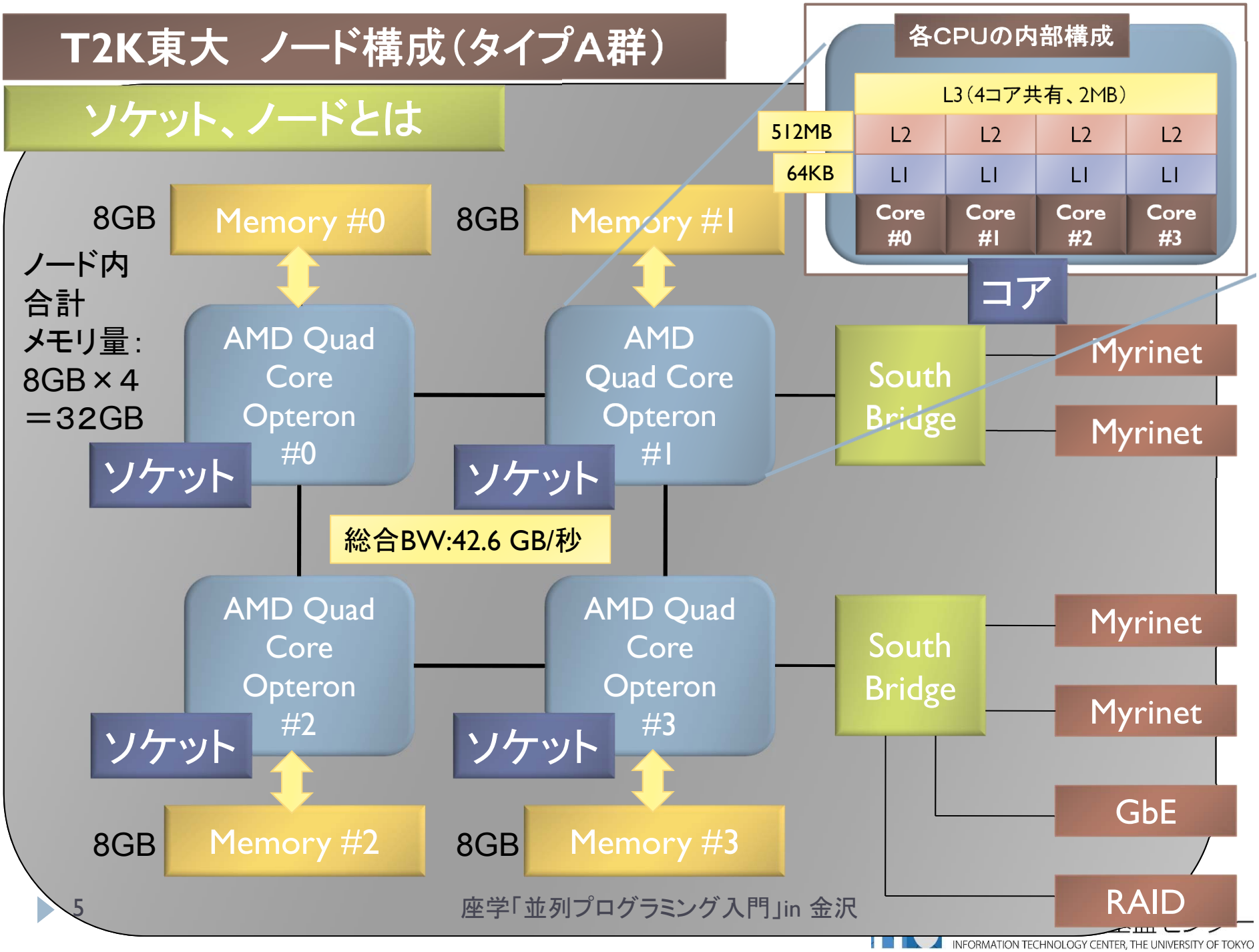
東京大学情報基盤センタースパコン

T2Kオープンスパコン（東大版）（HA8000クラスタシステム）

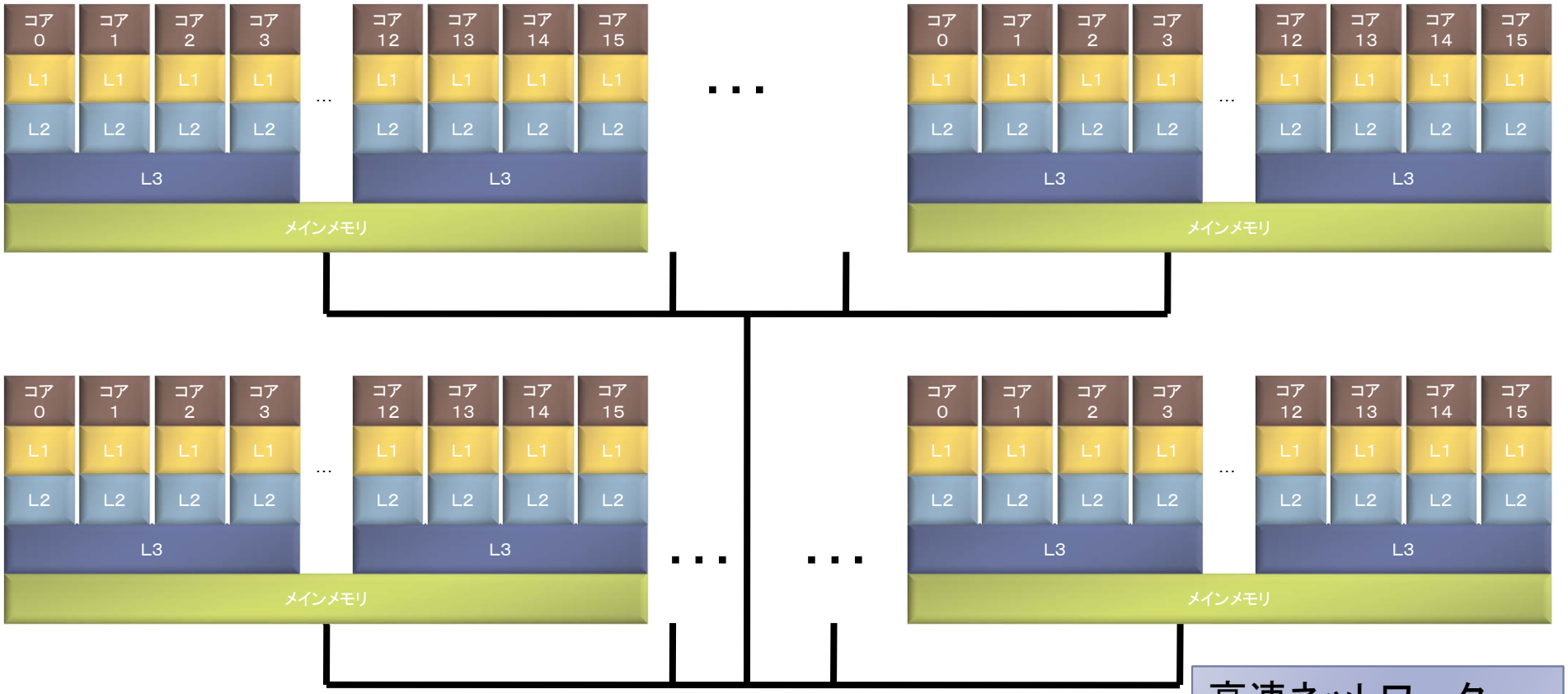
Total Peak performance	: 140 TFLOPS
Total number of nodes	: 952
Total memory	: 32000 GB
Peak performance per node	: 147.2 GFLOPS
Main memory per node	: 32 GB, 128 GB
Disk capacity	: 1 PB
AMD Quad Core Opteron（2.3GHz）	

製品名：HITACHI HA8000-tc/RS425

T2K東大　ノード構成（タイプA群）



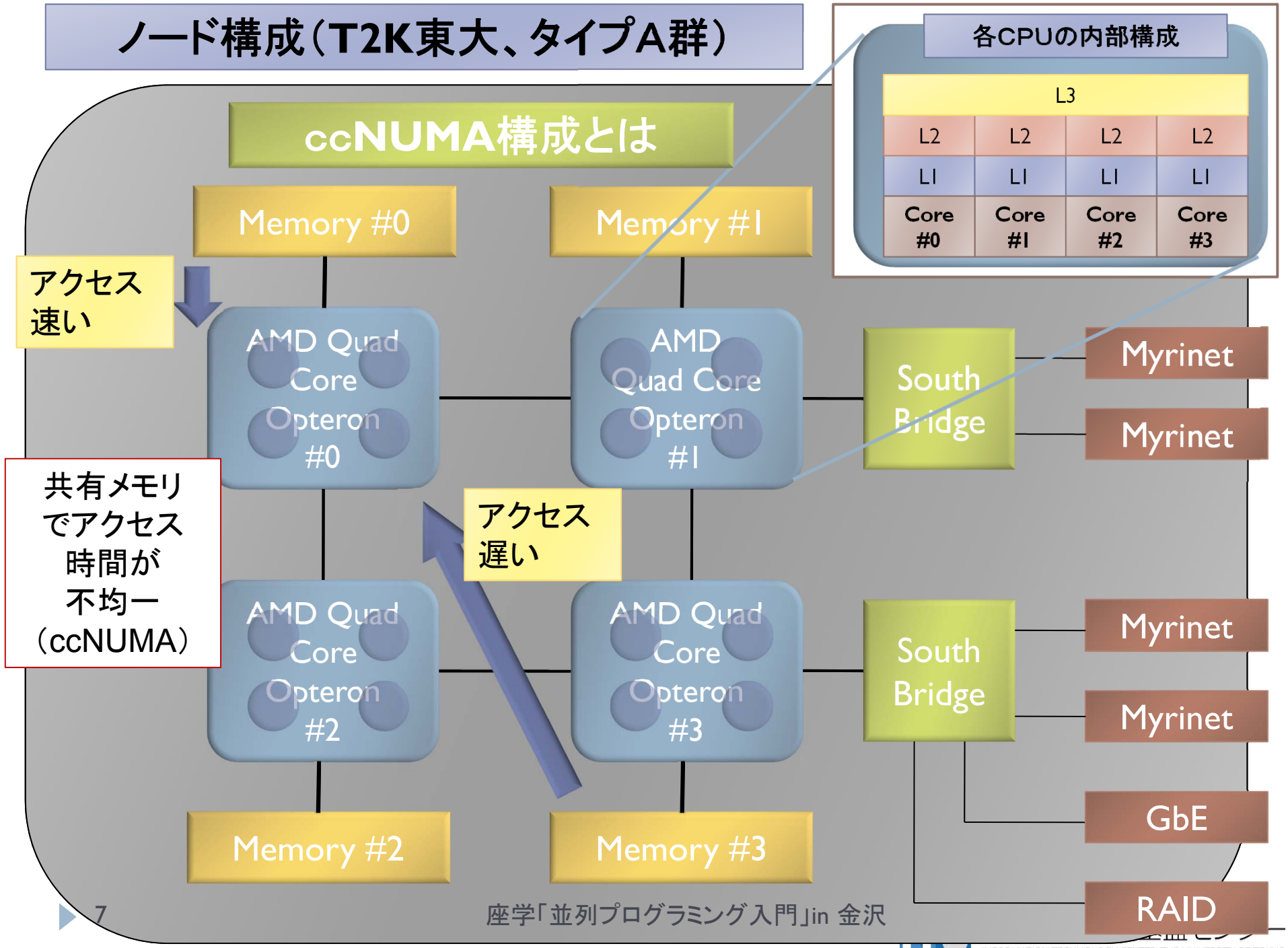
T2K（東大）での全体メモリ構成図



メモリが多段に階層化
（L1、L2、L3、分散メモリ）

高速ネットワーク
（5Gバイト／秒
×双方向）
（タイプA群）

ノード構成（T2K東大、タイプA群）



東京大学情報基盤センタースパコン

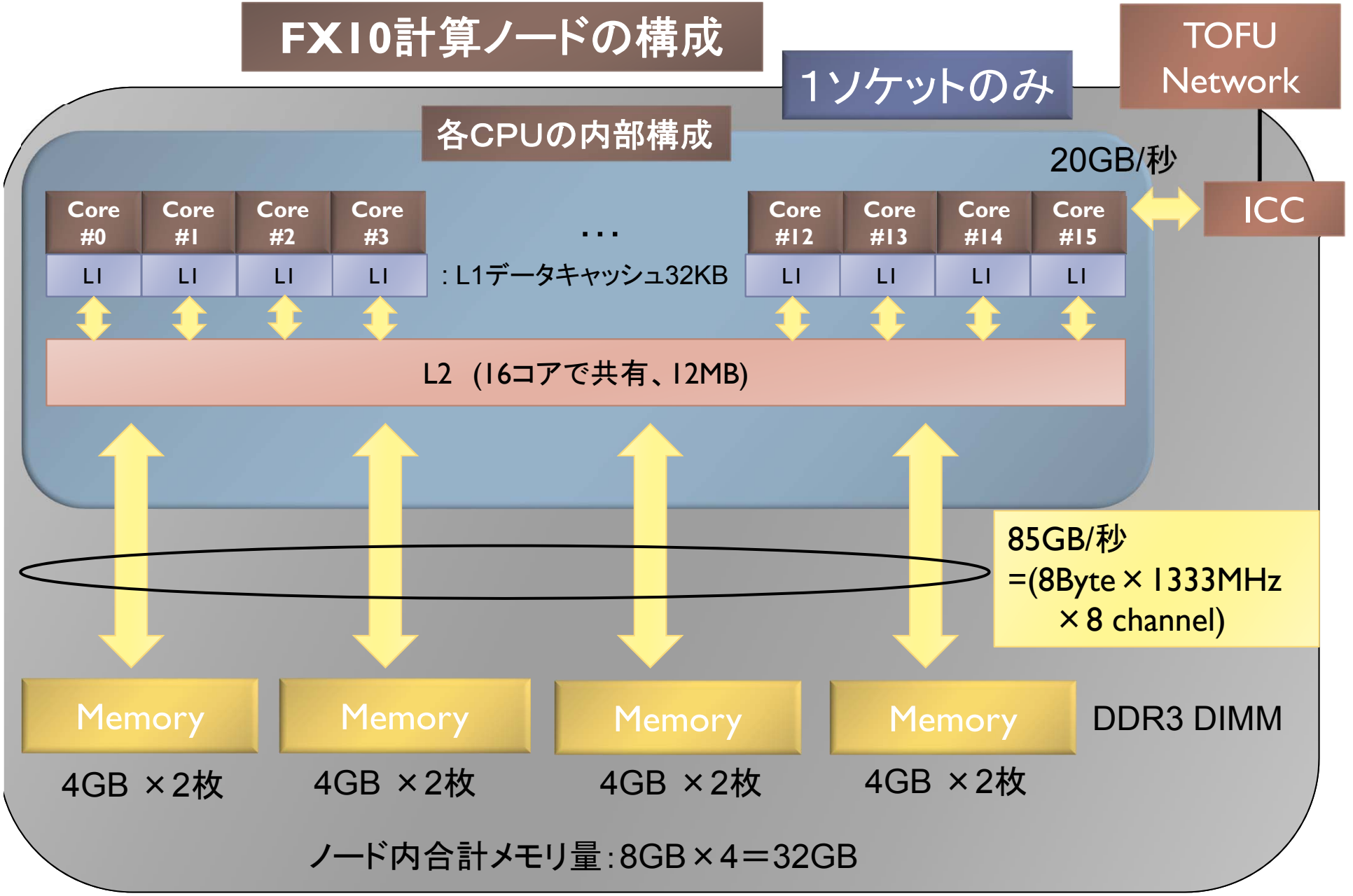
FX10スーパーコンピュータシステム

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）	

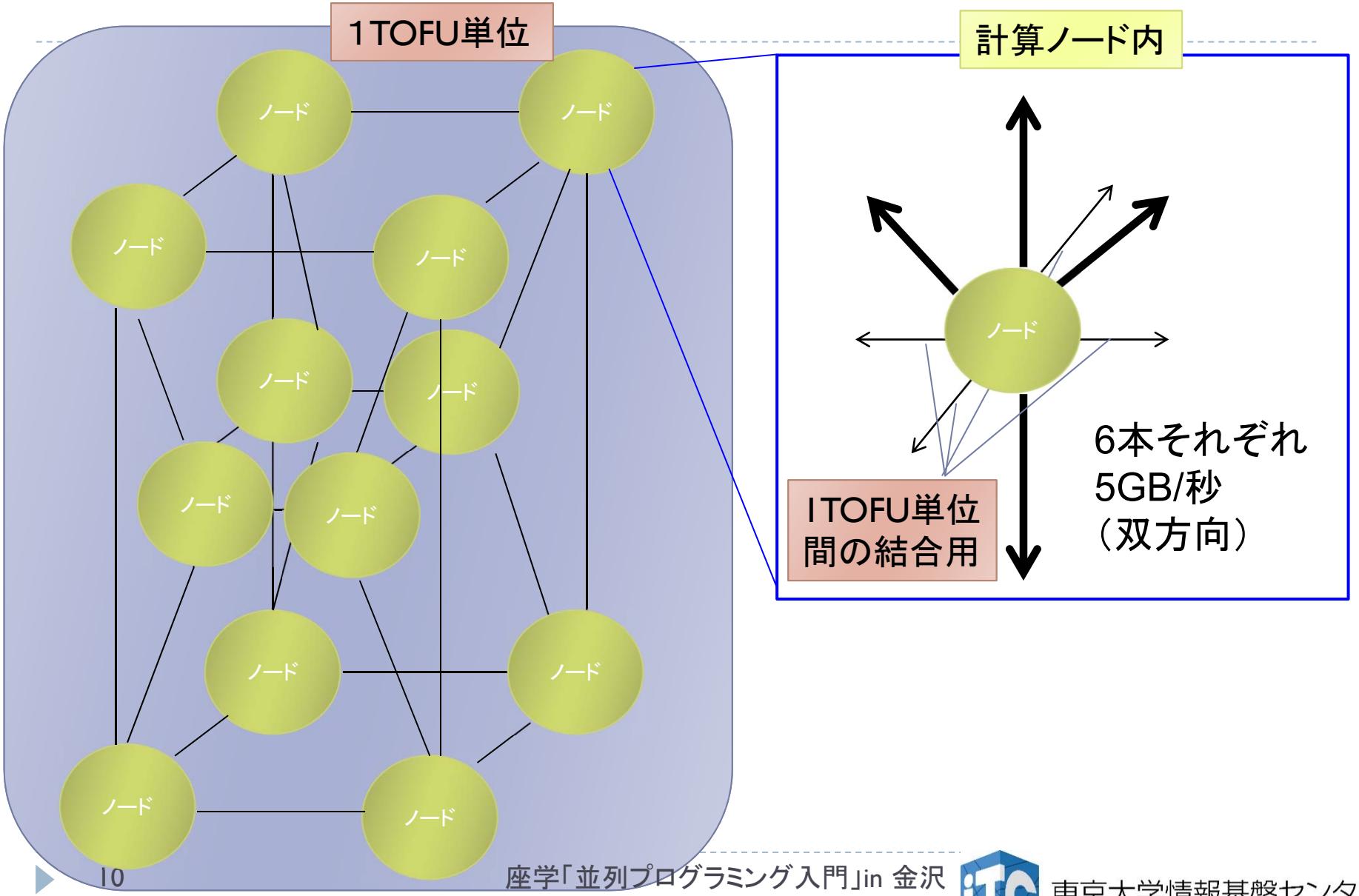
製品名 : Fujitsu PRIMEHPC FX10

2012年4月運用開始

FX10計算ノードの構成

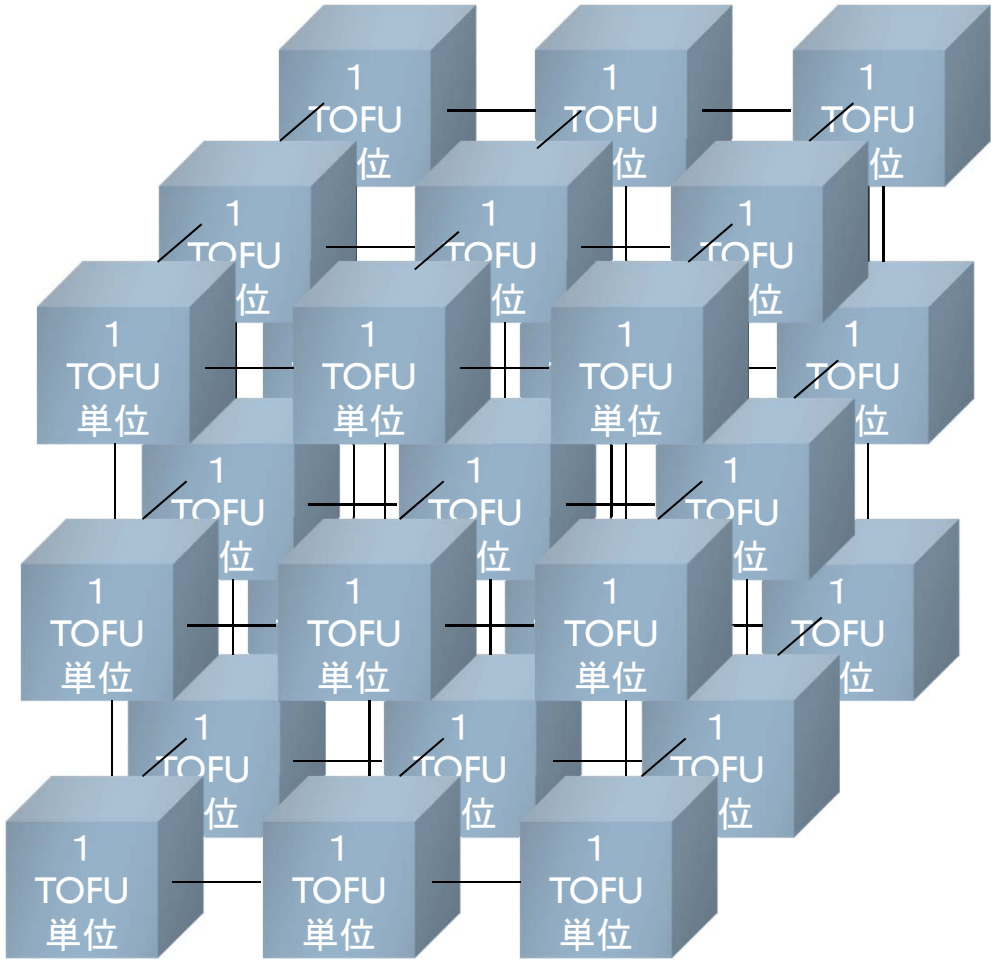


FX10の通信網（１TOFU単位）



FX10の通信網（１TOFU単位間の結合）

３次元接続



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

になっています

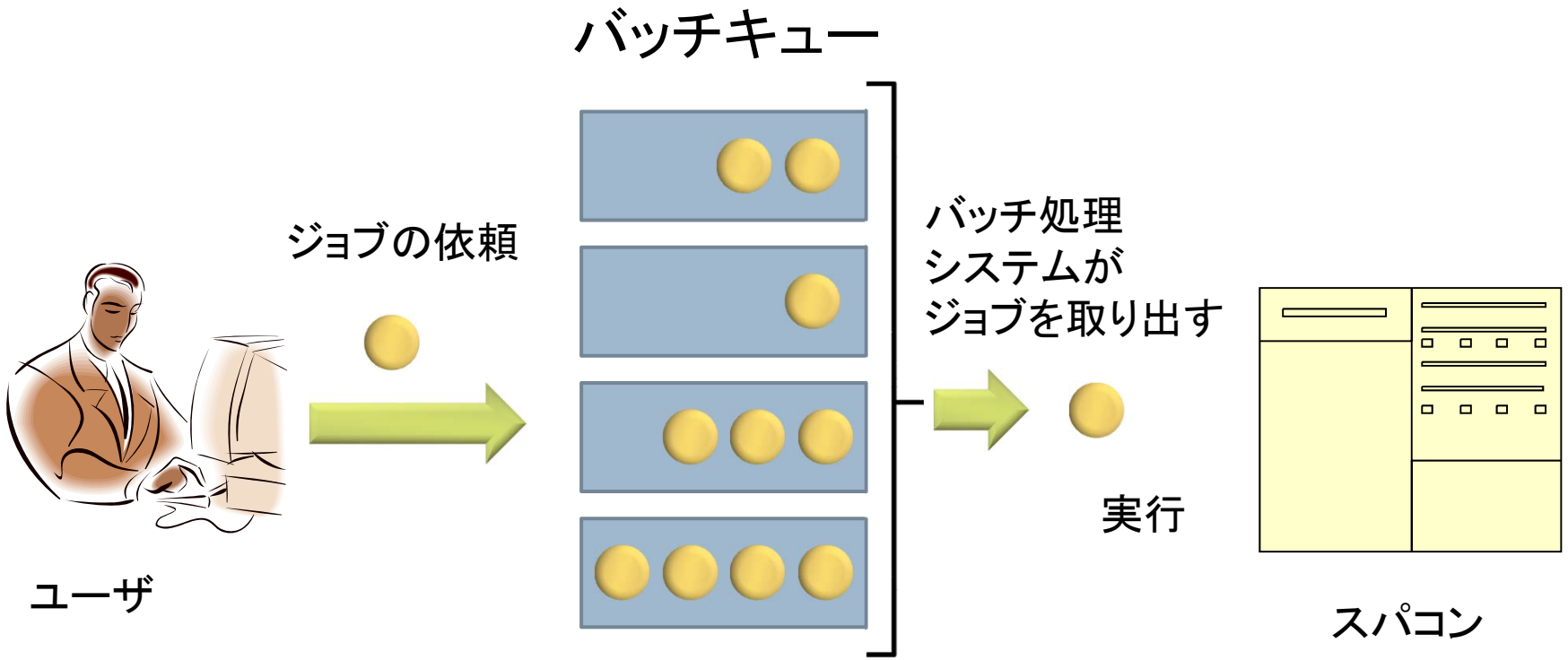
バッチ処理とMPIジョブの投入

FX10スーパコンピュータシステムで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
  - FX10では、以下に限定（東大基盤センターの運用方針）
    - 1ノード（16コア）（2時間まで）
    - 8ノード（128コア）（10分まで）
- バッチジョブ実行
  - バッチジョブシステムに処理を依頼して実行する方法
  - スパコン環境で一般的
  - 大規模実行用
  - FX10 (Oakleaf-FX)では、最大1440ノード（23,040コア）（24時間）
  - FX10 (Oakbridge-FX)では、最大576ノード（9,216コア）（168時間、7日）

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）

- 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（FX10の例）

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

インタラクティブ実行のやり方の例（FX10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

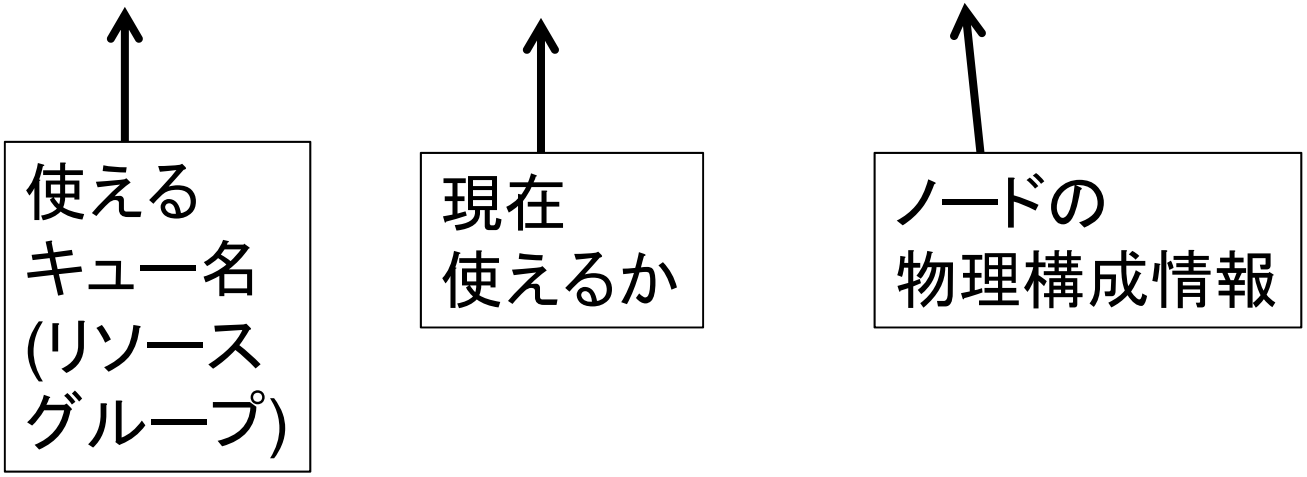
  - 8ノード実行用

    $ pjsub --interact -L “node=8”

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

pjstat --rsc の実行画面例

```
$ pjstat --rsc

RSCGRP          STATUS              NODE:COORD
lecture         [ENABLE,START]      72:2x3x12
lecture8        [DISABLE,STOP]      72:2x3x12
```

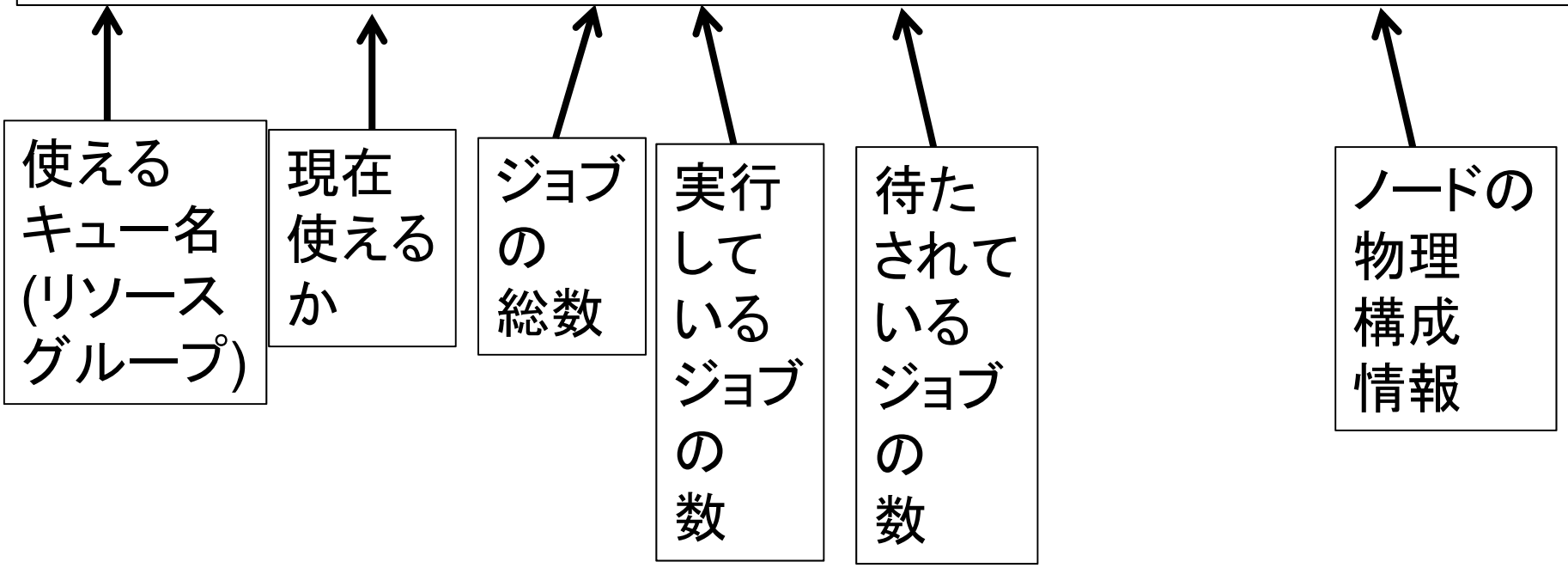


pjstat --rsc -x の実行画面例

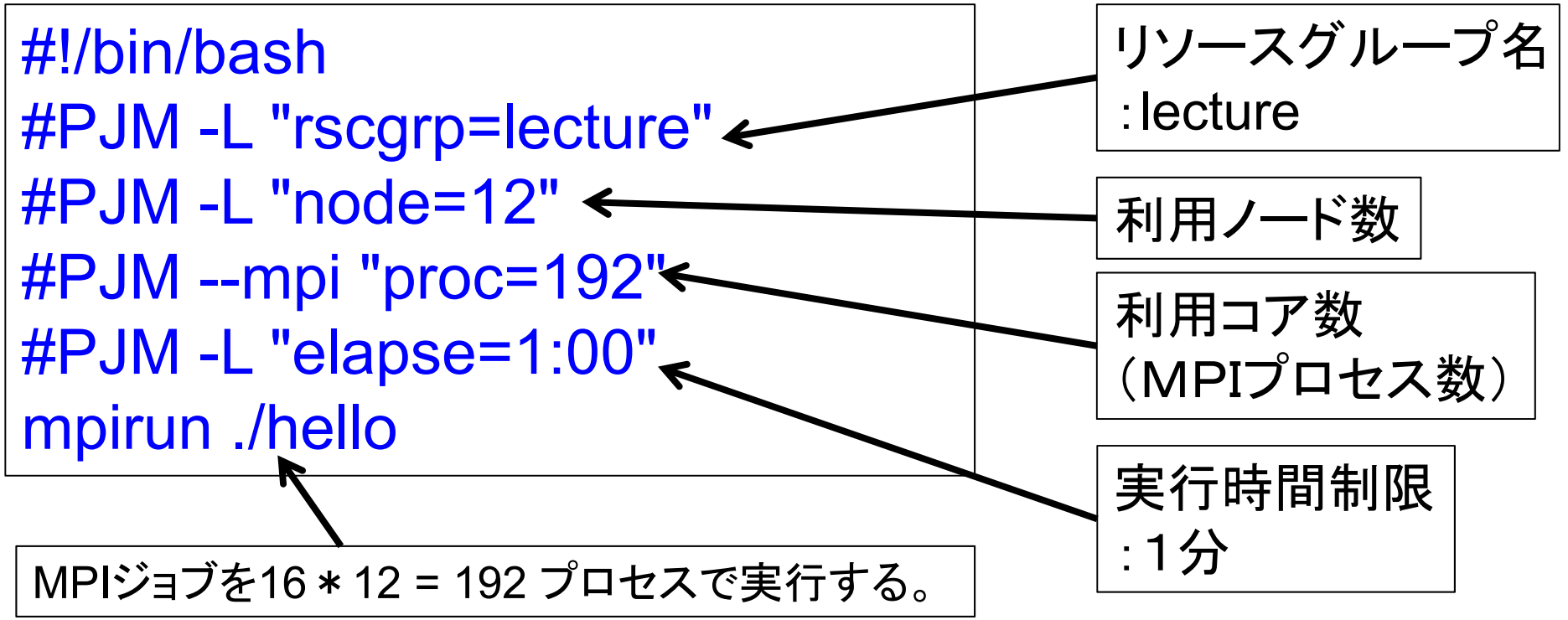
```
$ pjstat --rsc -x

RSCGRP      STATUS           MIN_NODE  MAX_NODE  ELAPSE    MEM(GB)  PROJECT
lecture     [ENABLE,START]          1        12  00:15:00       28  gt58
lecture8    [DISABLE,STOP]          1        12  00:15:00       28  gt58
```

pjstat --rsc -b の実行画面例

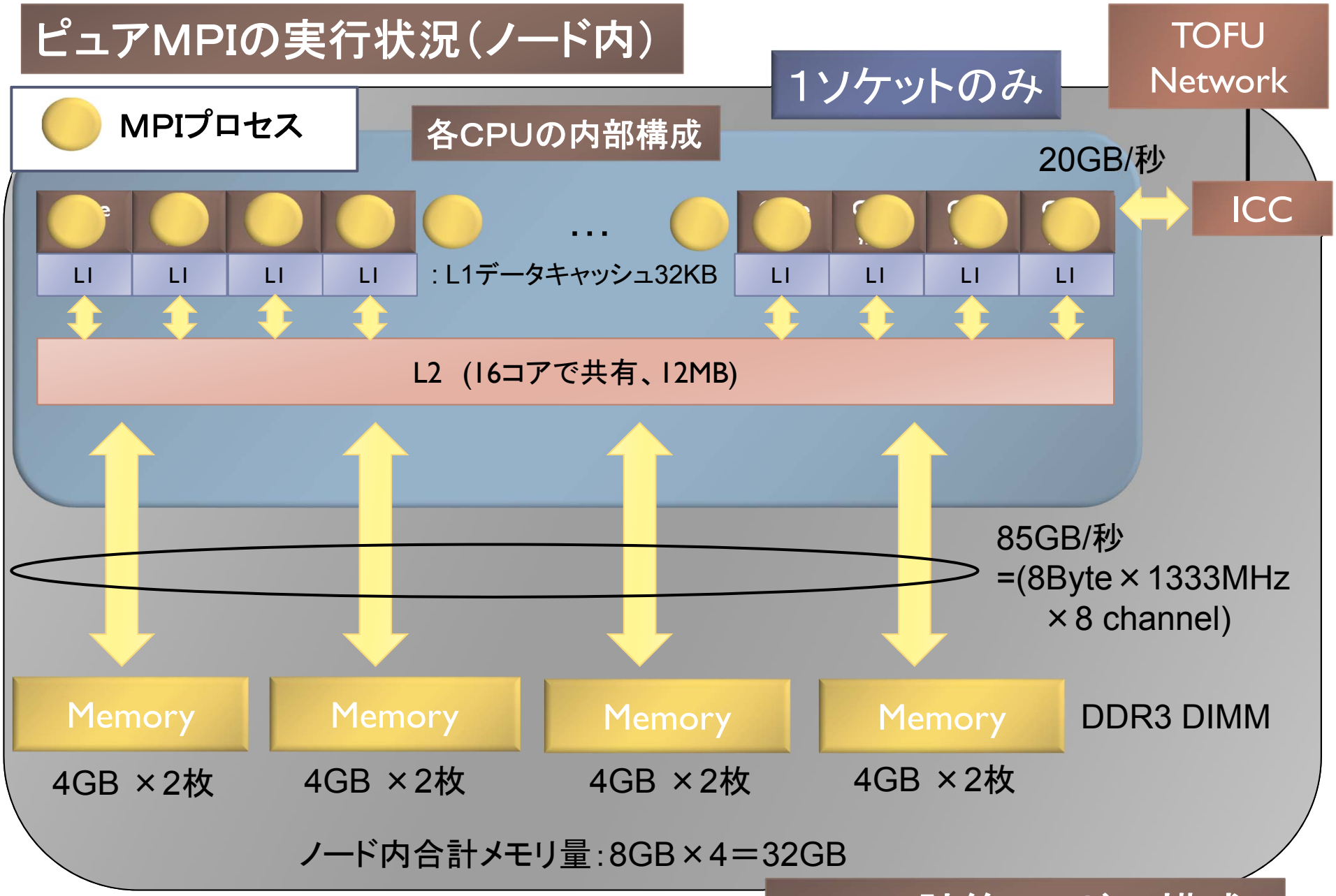
```
$ pjstat --rsc -b

RSCGRP    STATUS           TOTAL  RUNNING  QUEUED  HOLD  OTHER  NODE:COORD
lecture   [ENABLE,START]   0      0        0       0     0      72:2x3x12
lecture8  [DISABLE,STOP]   0      0        0       0     0      72:2x3x12
```



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



ピュアMPIの実行状況（ノード内）



FX10計算ノードの構成

並列版Helloプログラムを実行しよう（ピュアMPI）

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

バッチジョブ実行による標準出力、標準エラー出力

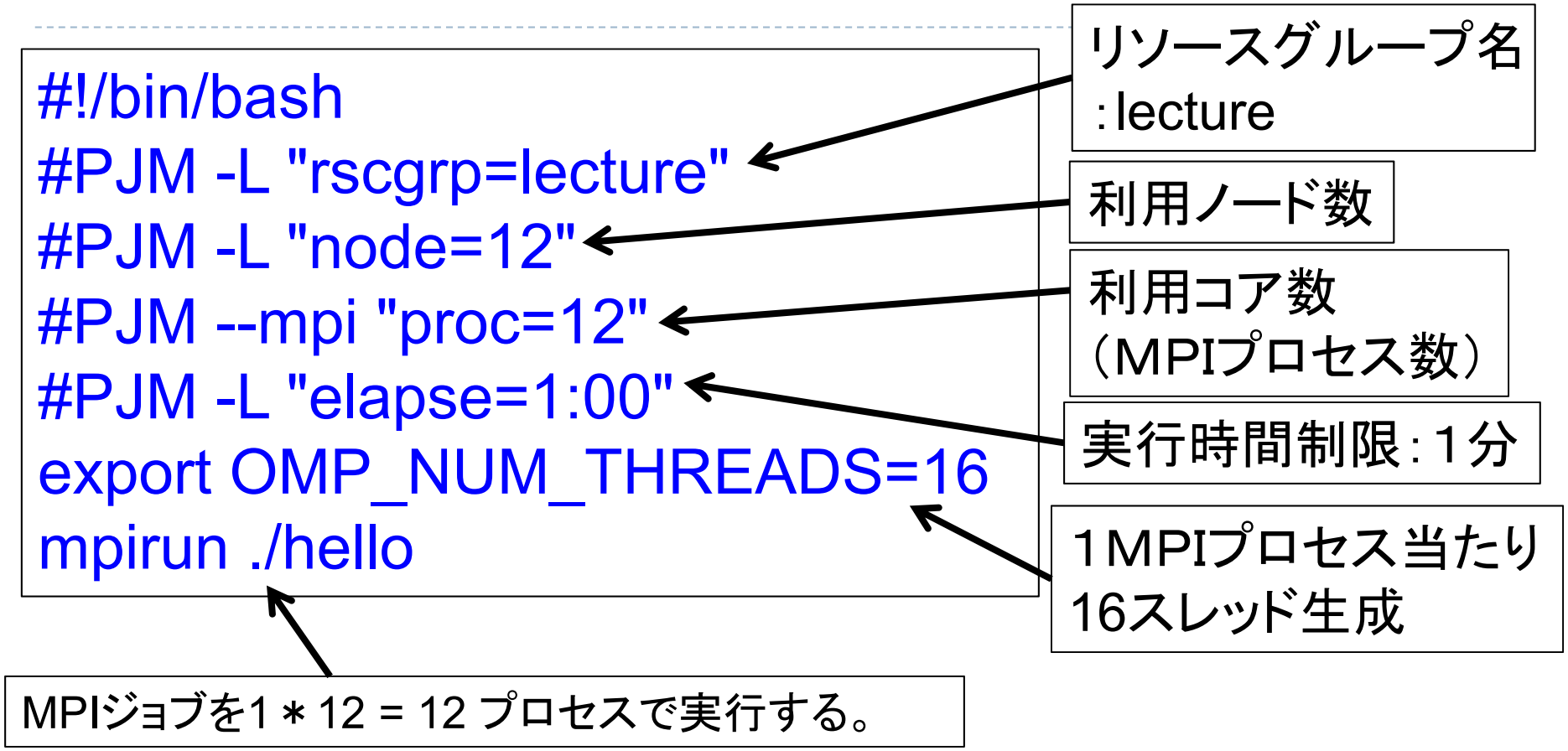
- バッチジョブの実行が終了すると、標準出力ファイルと標準エラー出力ファイルが、ジョブ投入時のディレクトリに作成されます。
- 標準出力ファイルにはジョブ実行中の標準出力、標準エラー出力ファイルにはジョブ実行中のエラーメッセージが出力されます。

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

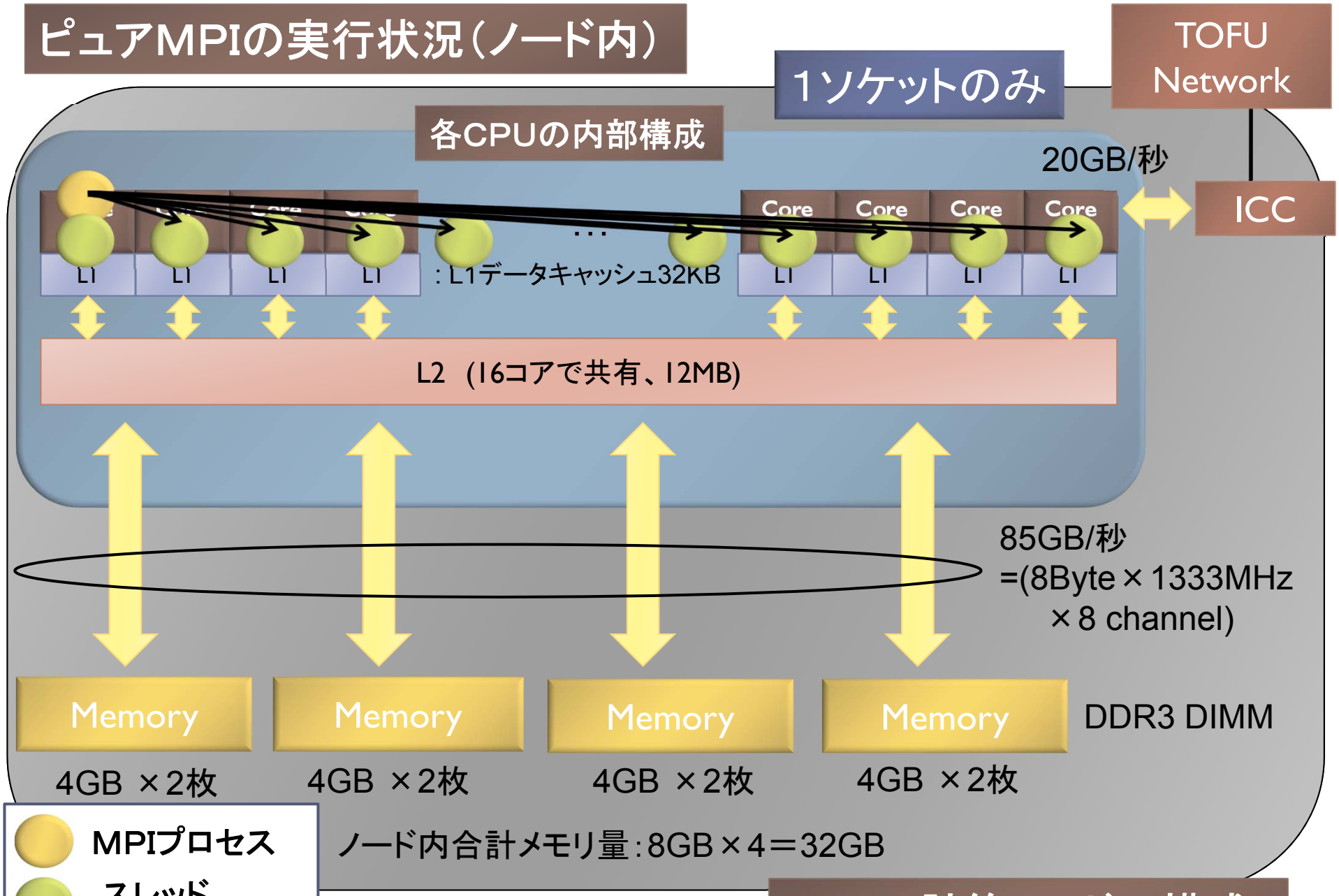
並列版Helloプログラムを実行しよう（ハイブリッドMPI）

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

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



ピュアMPIの実行状況（ノード内）



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
    - スレッド数（論理スレッド数）が上記の利用コア総数
    - 以上を超えても実行できるはずだが、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スレッド（経験値、問題、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

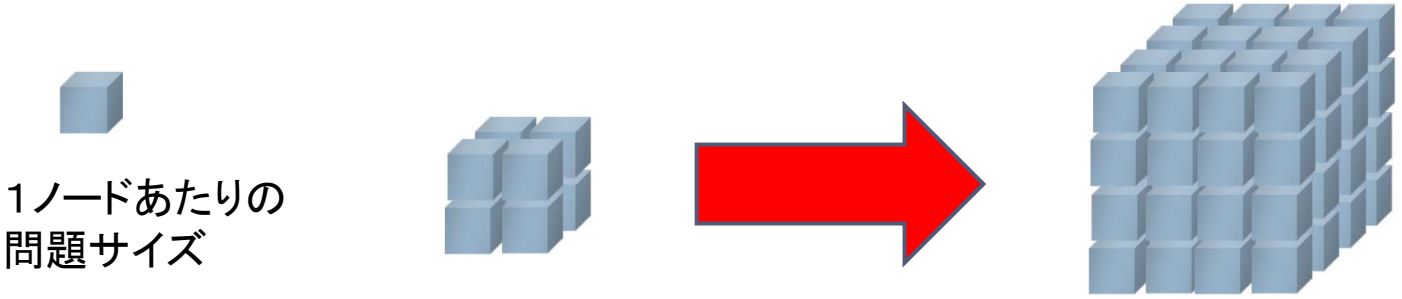
- 一般に、スーパーコンピュータでは、
  MPI実行時の入出力のリダイレクトができません
  - ×例）mpirun ./a.out < in.txt > out.txt

- 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

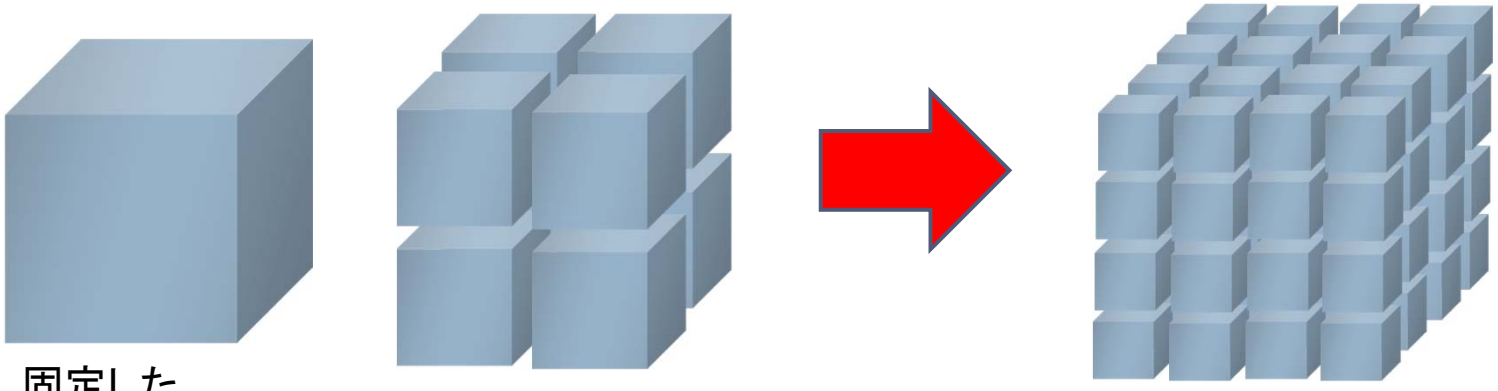
弱スケーリング (Weak Scaling)

- 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（主にシステム的な要因により）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$O(N^3)$



固定した問題サイズ　　8ノード実行での問題サイズ　　64ノード実行での問題サイズ

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
  - 2015年11月のTOP500の、コア当たりの問題サイズ
  - （1位）Tianhe-2、
    N=9,960,000、#cores=3,120,000、N/#cores=3.19
  - （4位）K computer、
    N=11,870,208、#cores=705,024、N/#cores=16.8
  - （6位）Piz Daint、
    N=4,128,768、#cores=115,984、N/#cores=35.5
- 上位のマシンほど、コア当たりの問題サイズが小さい
  ←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

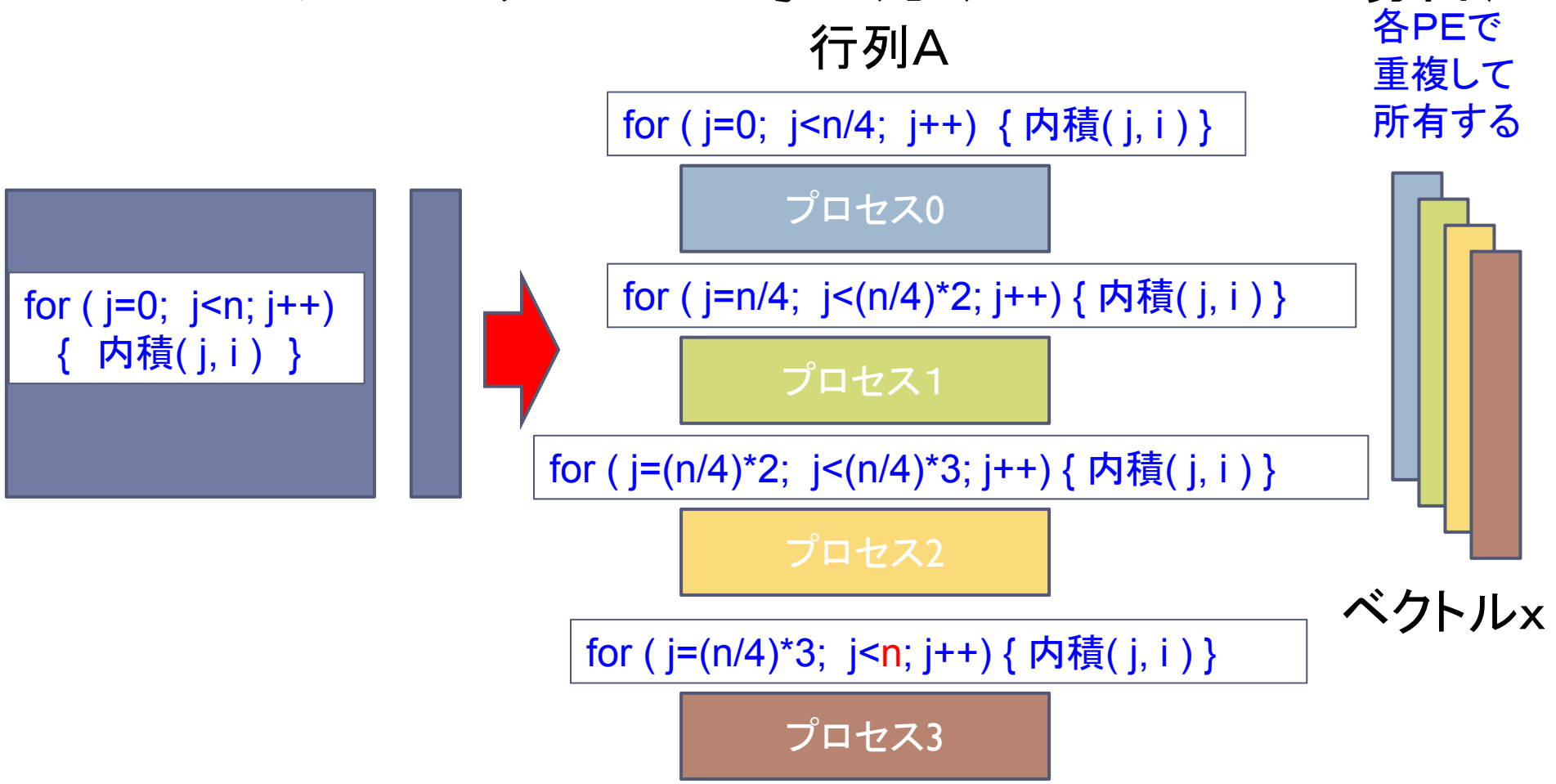
MPI並列化の大前提（再確認）

- 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）

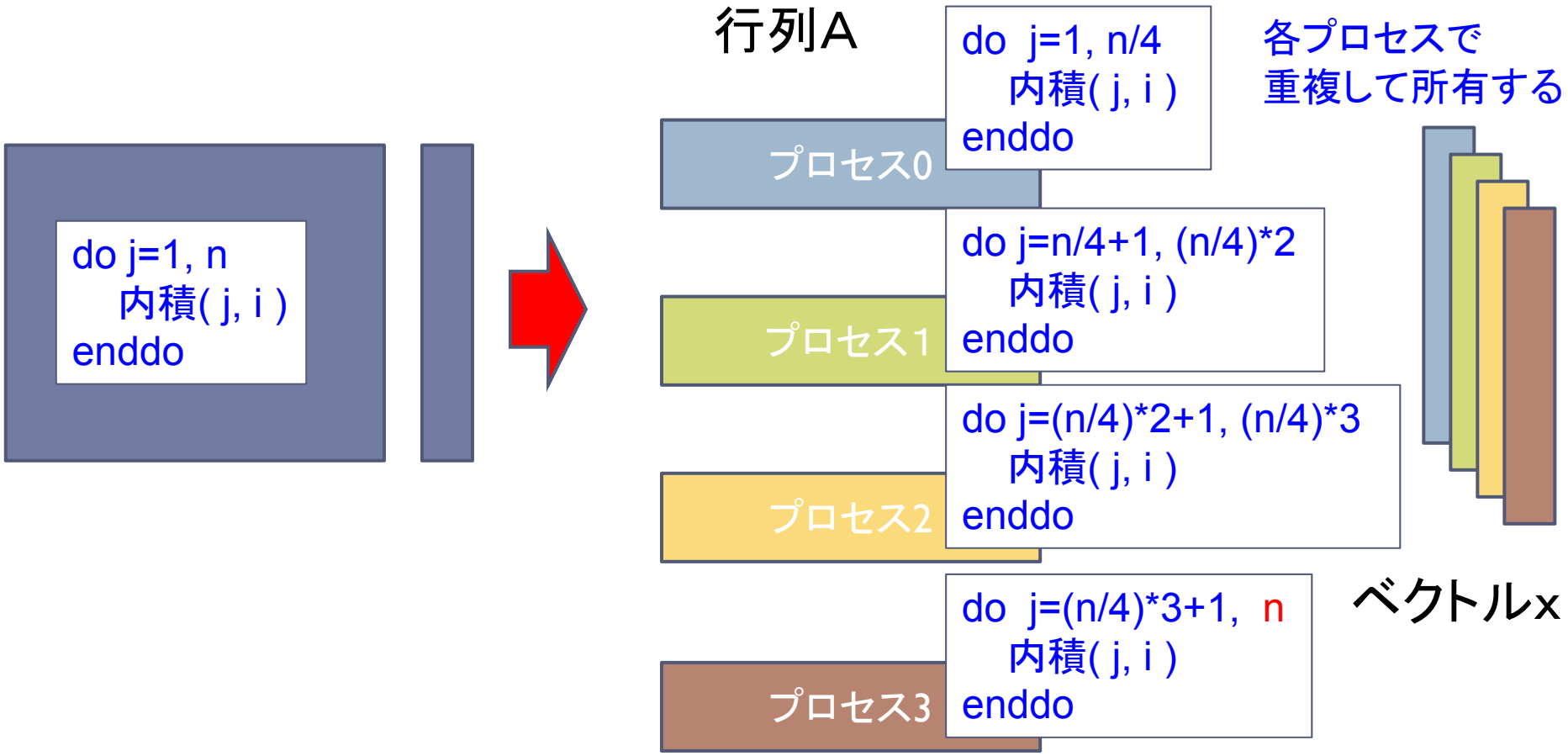
並列化の考え方（C言語）

- SIMDアルゴリズムの考え方（4プロセスの場合）



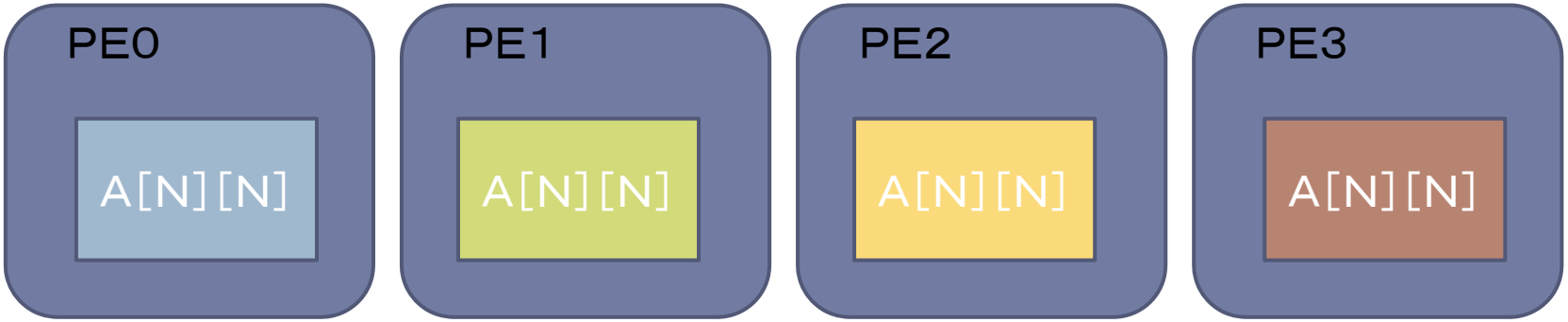
並列化の考え方（Fortran言語）

- SIMDアルゴリズムの考え方（4プロセスの場合）

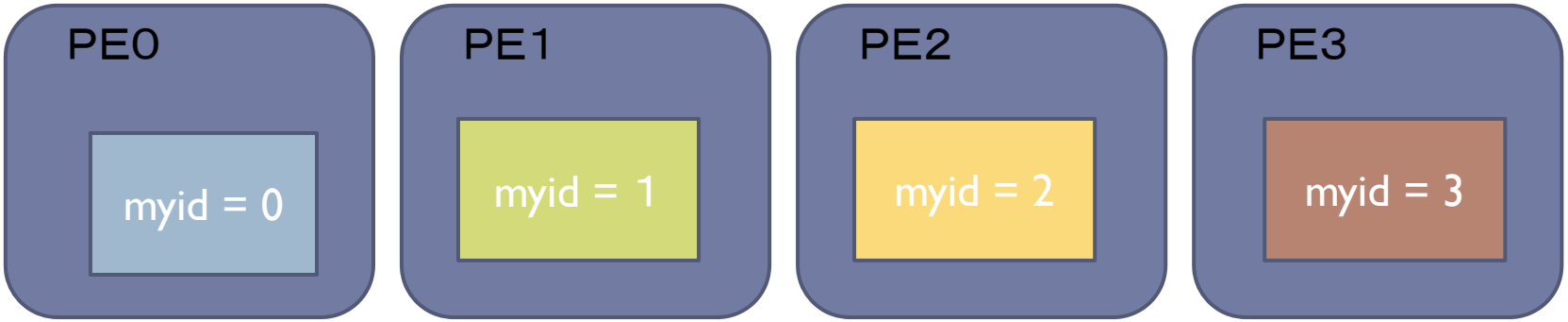


初心者が注意すること

- 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×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×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。

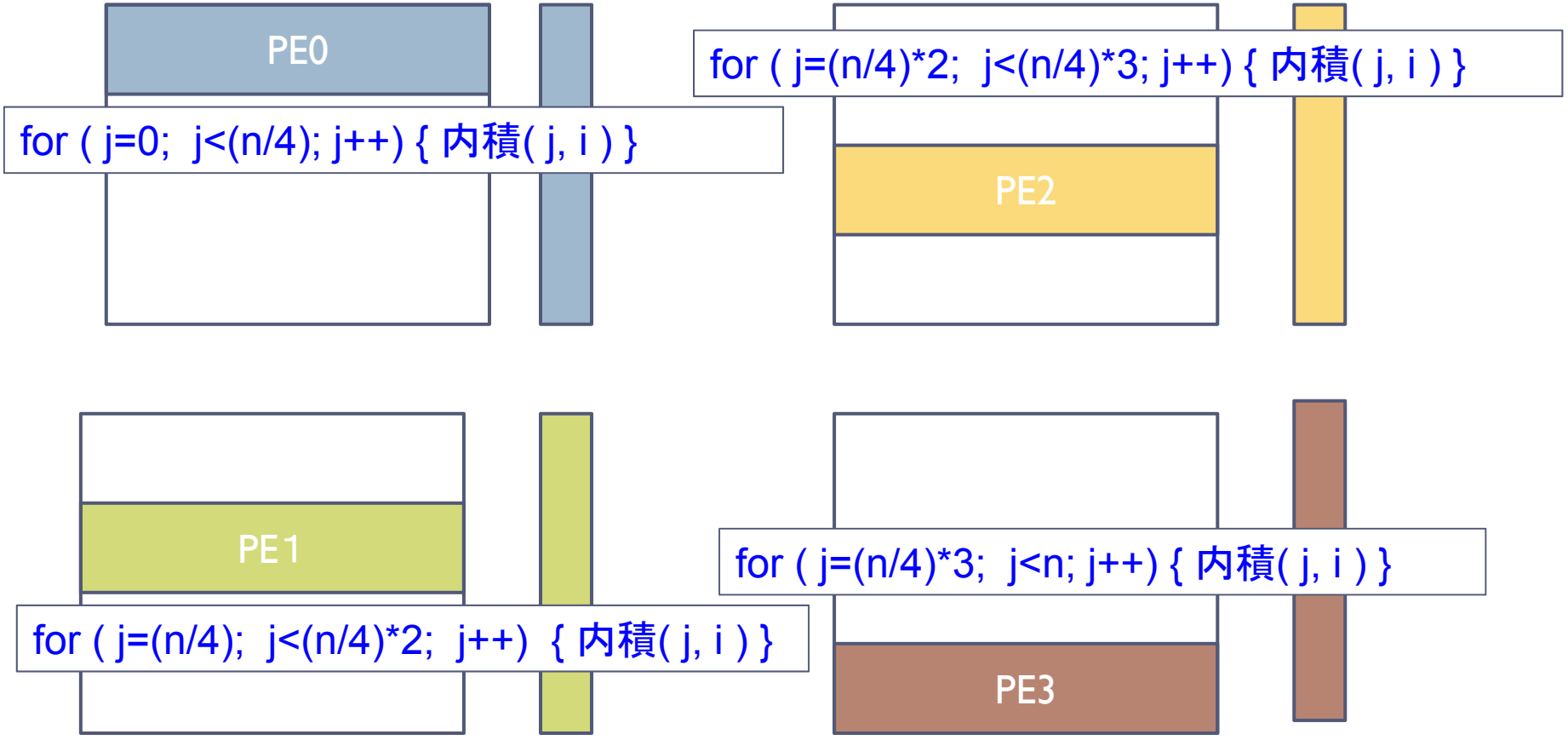
```
do  j=1, ib    …    enddo
```

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）

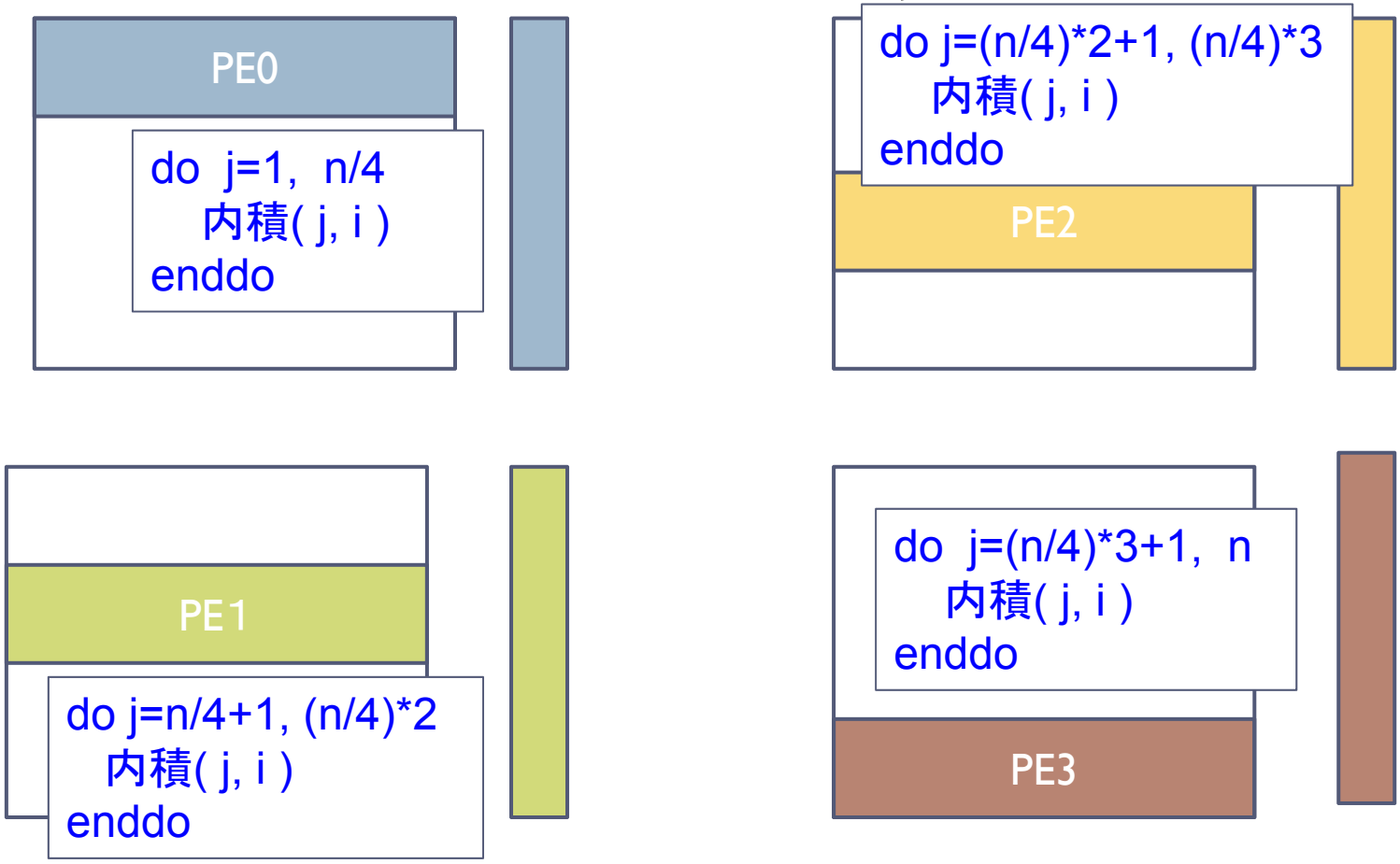
- 全PEでN×N行列を持つ場合



※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）

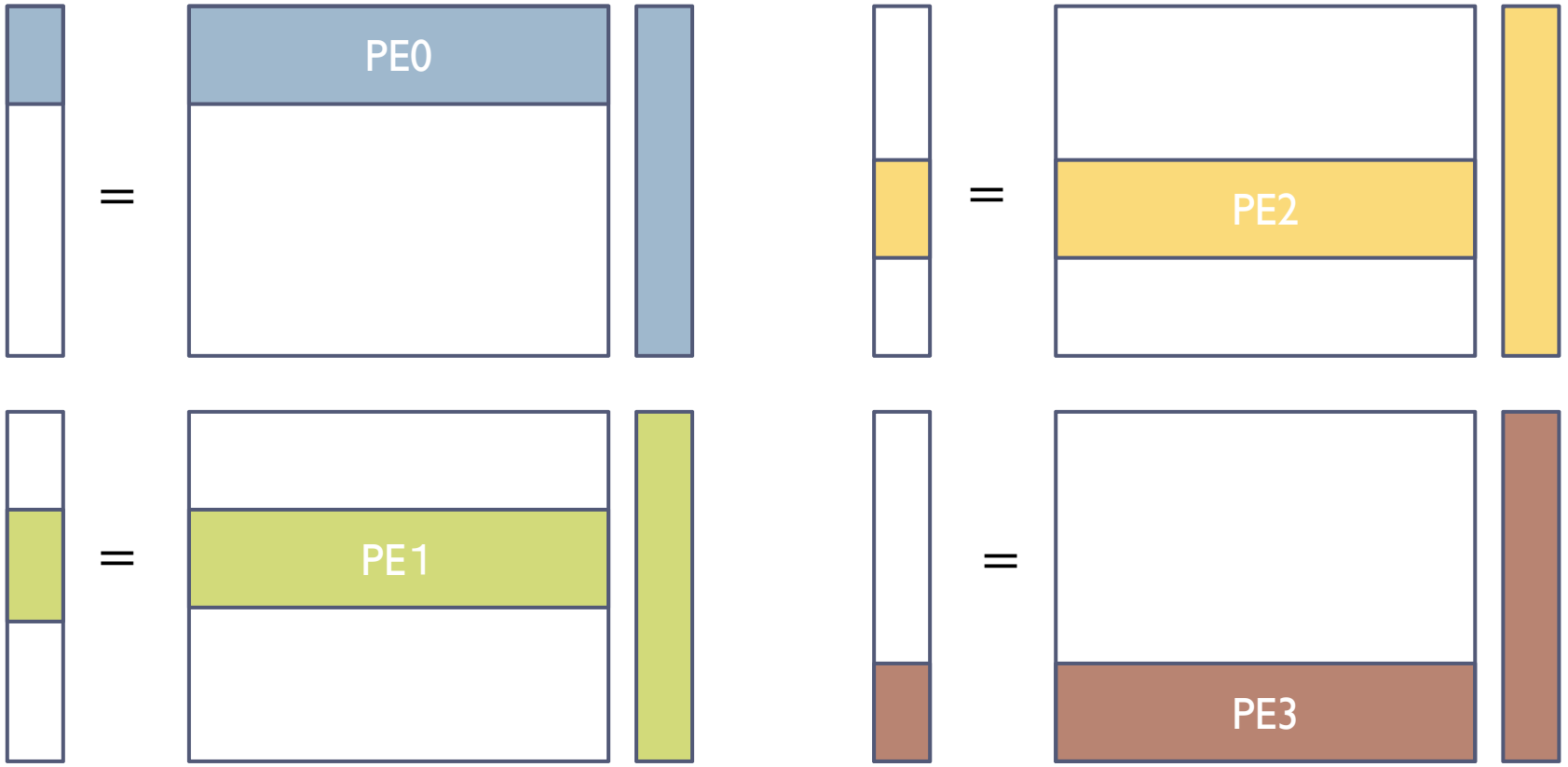
- 全PEでN×N行列を持つ場合



※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4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4）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;

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

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

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

ブロック分散を仮定した担当ループ範囲の定義

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/numprocs + mod(N, 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

- mod(N, numprocs)が大きいと、負荷バランスが悪化
  - 例：N=10、numprocs=6
    - 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_start(0)=1, i_end(0)=2, 2個
    - プロセス1： i_start(1)=3, i_end(1)=4, 2個
    - プロセス2： i_start(2)=5, i_end(2)=6, 2個
    - プロセス3： i_start(3)=7, i_end(3)=8, 2個
    - プロセス4： i_start(4)=9, i_end(4)=9, 1個
    - プロセス5： i_start(5)=10, i_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

ハイブリット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

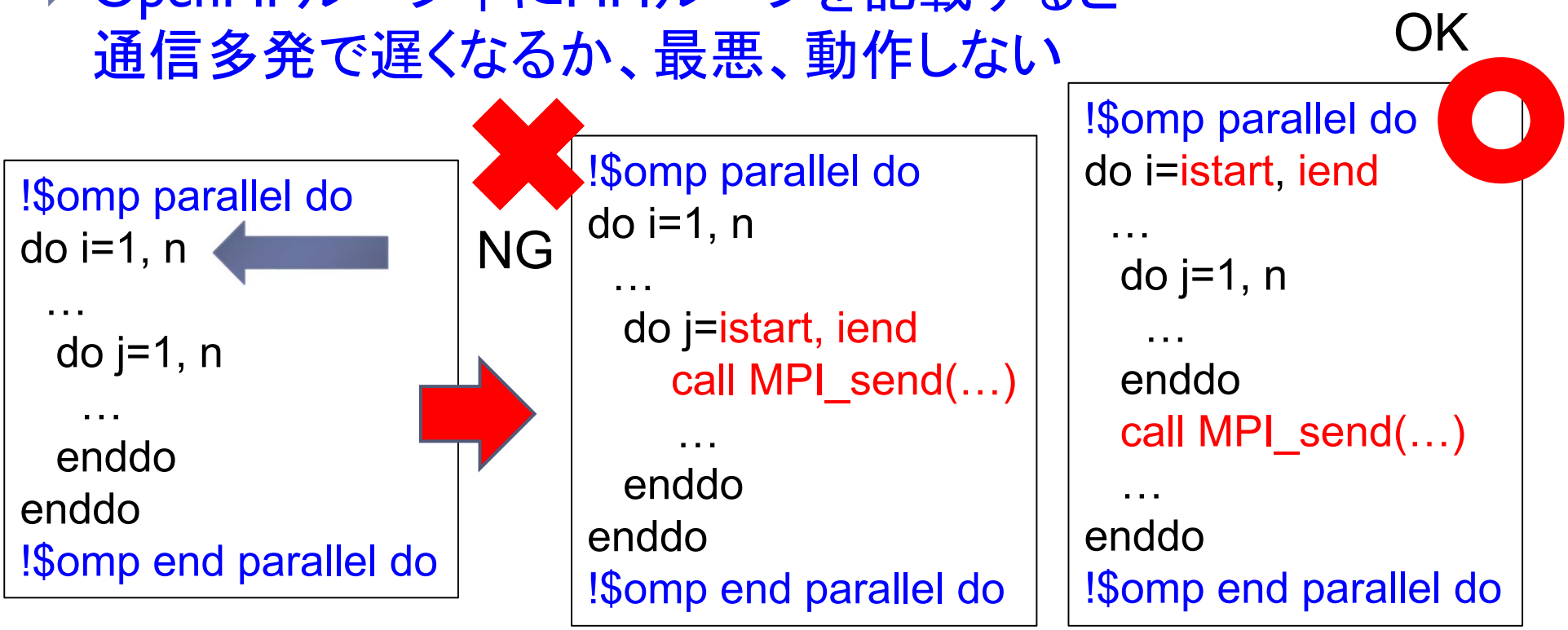
  ことが主な目的
- 例）東京大学のFX10
  - 全系は4,800ノード、76,800コア
  - ピュアMPI実行：76,800プロセス実行
  - ハイブリッドMPI実行（1ノード16スレッド実行）：4,800プロセス
  - MPIプロセス数の比は16倍！

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

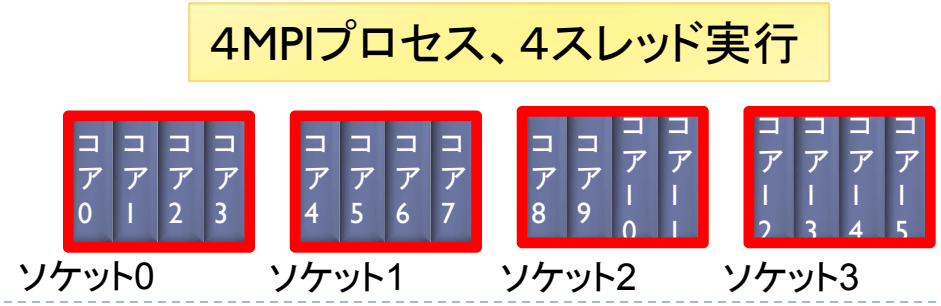
この文を追加するだけ！

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

  - 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）T2K（AMD Quad Core Opteron）、4ソケット、16コア



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る方法がある。（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ト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
```

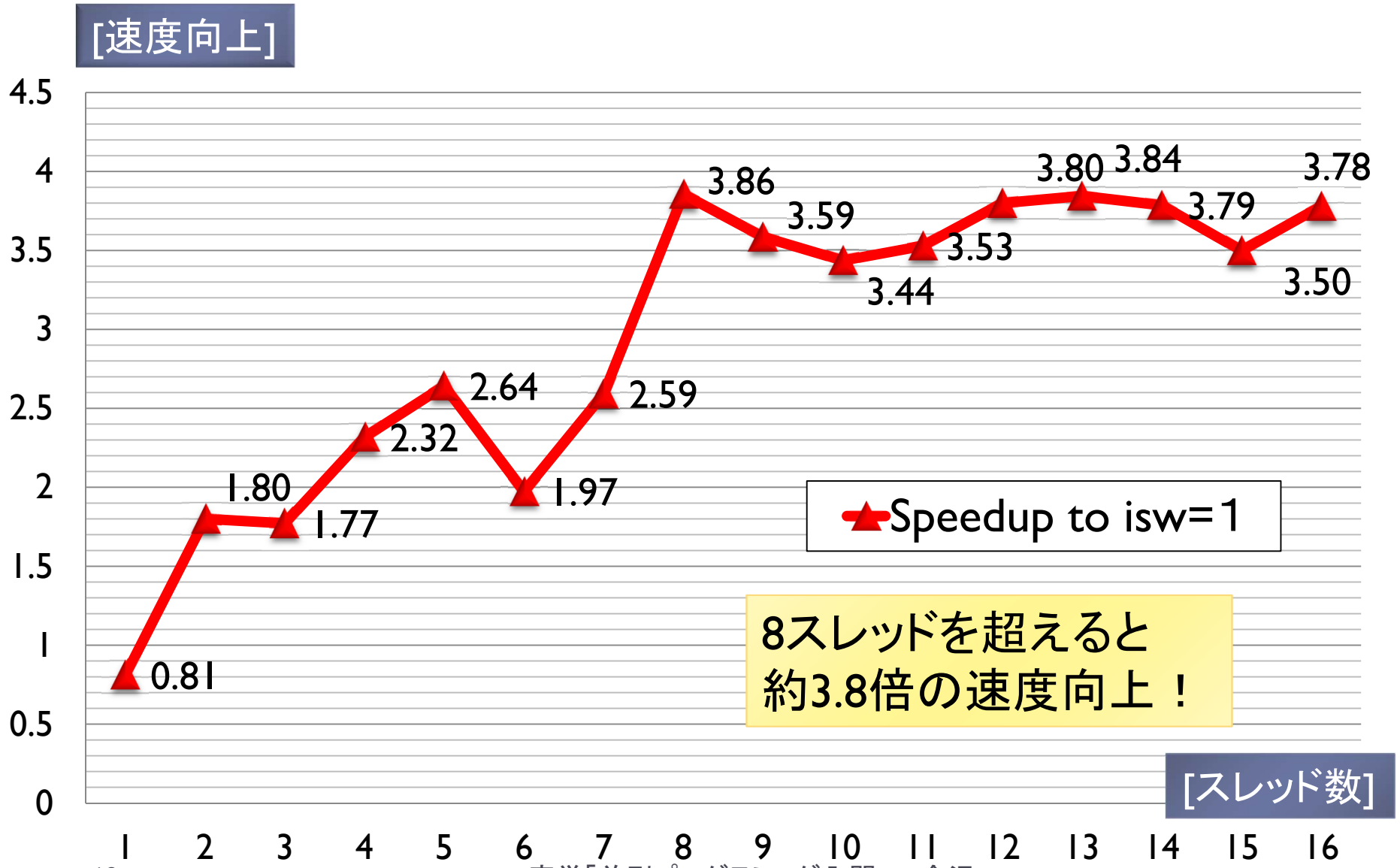
- 上位のループでOpenMP並列化したBLASの呼び出し

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

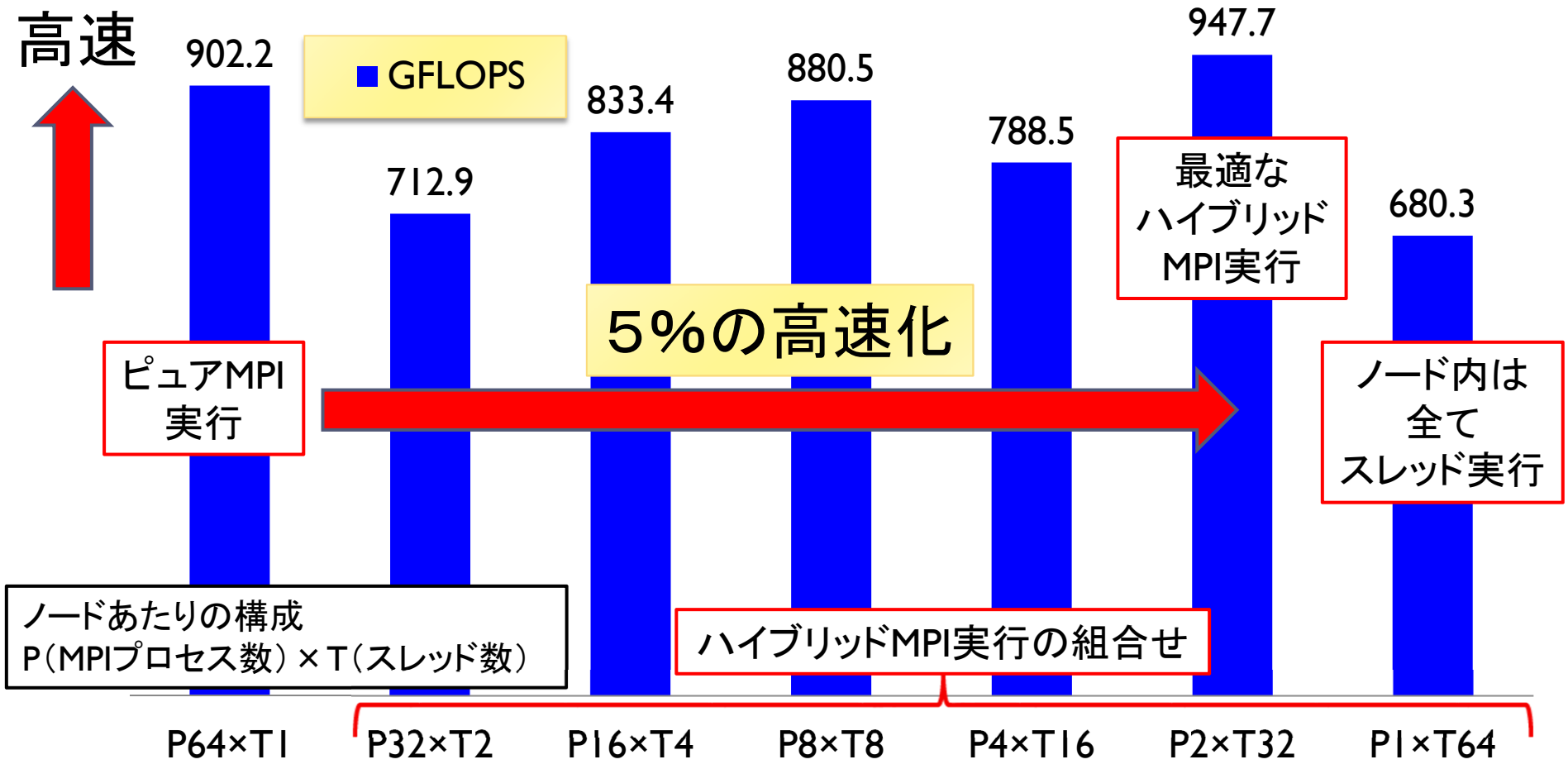


ScaLAPACKにおける ハイブリッドMPI実行の効果の例

- 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が期待される
- 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）

1. [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、192コア）を駆使して、性能評価せよ。
   - 1ノードあたり、12MPI実行、1MPI+16スレッド実行、2MPI+8スレッド実行、4MPI+4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。

性能チューニングの応用

性能チューニングに関する総論（その１）

- コンパイラを過信しない
  - 書き方が悪いと、自動並列化だけでなく、逐次最適化もできない！
    - ベクトル計算機向きに書かれたコードは、１ループ中で書いてある＜式＞がとても多い。
    - スカラ計算機ではレジスタが足りなくなって、メモリにデータを吐き出すコードを生成するので、性能低下する。
      ⇒後述の、手による「ループ分割」が必要になる

性能チューニングに関する総論（その２）

- コンパイラを過信しない（つづき）
  - 自動並列化は＜特に＞過信しない
    - ループ並列性がない逐次コードは並列化できない
    - 書き方が悪いと、原理的に並列化できるループも、自動並列化できない
      - ループの構造（開始値、終了値が明確か、など）
      - 言語的な特徴から生じる問題もある
        - C言語では、並列化したいループがある関数コール時の引数にデータ依存があると判断されると、並列化できない。

性能チューニングに関する総論（その２）

- コンパイラを過信しない（つづき）
  - 例）foo (A, B, C); ←一般にA、B、Cは同一配列で引渡される可能性があるため、A、B、C間は依存があると仮定
    ※ディレクティブ、コンパイラオプション指定で対応

```
int  foo(double A[N][N], double B[N][N], double C[N][N])  {
   int i, j, k;
   for (i=0; i<N; i++)  {
       for (j=0; j<N; j++)  {
          for (k=0; k<N; k++)  {
             C[i][j] += A[i][k] * B[k][j]; } } }
}
```

性能チューニングに関する総論（その３）

- コンパイラを過信しない（つづき）
  - スレッド数の増加
    - 低スレッド並列（２～４スレッド）向きのコードと、高スレッド並列（８スレッドを超える）向きコードは、まったく異なる
    - コンパイラは、実行前にユーザが使うスレッド数を知ることが出来ない
      - 平均的なスレッド数を仮定、まあまあな性能のコードを生成する
    - 並列数が増加すると、ループ長が短くなることで、ループ並列性が無くなる
      ⇒後述の、手による「ループ融合」が必要になる

性能チューニングに関する総論（その４）

- コンパイラを過信しない（つづき）
  - あるベンダ提供のコンパイラで最適化できたとしても、別のベンダ提供のコンパイラで最適化できる保証はない
    - 例）SR16000の日立コンパイラ　と　FX10の富士通コンパイラ
  - 同一ベンダのコンパイラでも、新規ハードで同一コードを最適化できる保証がない
  - 従来からあるコード（レガシーコード）で、ハードウェア、および、ソフトウェア環境が変わっても、高い性能を保つこと（性能可搬性と呼ぶ）は、HPC分野で活発な研究テーマ
    - 「ソフトウェア自動チューニング」の研究分野
    - ソフトウェア性能工学 (Software Performance Engineering, SPE)
    - ソフトウェア開発コストを低く保つ、チューニングの枠組み
      - コード自動生成技術
      - 性能モデリング、最適パラメタ探索、機械学習、の技術が必要

性能チューニングに関する総論（その５）

- **自分のコードのホットスポット（重い部分）を認識せよ**
  - 自分のコードのうち、どの部分が重いのか、実測により確認せよ
    1. 演算時間ボトルネック（演算時間が多い）
    2. 通信時間ボトルネック（通信時間が多い）
    3. I/Oボトルネック（I/O時間が多い）

性能チューニングに関する総論（その６）

- **自分のコードのホットスポット（重い部分）を認識せよ**
  - 計算量など、机上評価はあてにならない
    - 実性能は計算機環境や実行条件に依存
      - □思わぬところに　ホットスポット（重い部分）
      - □チューニング状況に応じホットスポットは変わる
    - 計算量が多くても、問題サイズが小さく、キャッシュにのる場合は、演算時間が占める割合は少ない
    - 通信量が少なくても、通信＜回数＞　が多いと、通信レイテンシ律速
    - I/O量が少なくても、I/Oハードウェアが貧弱、実行時に偶発的にI/O性能が劣化すると、I/O律速

状況に応じて変化していくホットスポット

- 最初は演算律速

- 演算チューニングをすると、次は通信律速に

ホットスポット判明後の最適化方針の一例

- 演算ボトルネックの場合（順番は検討する優先度）
  1. コンパイラオプションの変更
     - プリフェッチ、ソフトウェア・パイプライニング強化オプション、など
     - アンローリング、タイリング（ブロック化）のディレクティブ追加、など
  2. アルゴリズムを変更し、計算量が少ないものを採用
  3. アルゴリズムを変更し、キャッシュ最適化向きのものを採用
     - 「ブロック化アルゴリズム」の採用
  4. コンパイラが自動で行わないコードチューニングを手で行う
     - アンローリングなど
     - 高速化（連続アクセス）に向くデータ構造を採用

ホットスポット判明後の最適化方針の一例

- 通信ボトルネックの場合
  - 通信レイテンシが主要因（通信回数が多い）
    1. こま切れの通信をまとめて送る
       （通信のベクトル化）
    2. 冗長計算による通信回数の削減
    3. 非同期通信による通信の隠ぺい
  - 通信量が主要因（1回当たり通信データが多い）
    1. 冗長計算による通信量の削減
    2. より高速な通信実装方式の採用
       （Remote Direct Memory Access (RDMA) など）
    3. 非同期通信による通信の隠ぺい

ホットスポット判明後の最適化方針の一例

- I/Oボトルネックの場合
  1. 高速なファイルシステムを使う
     - ファイルステージングの利用
  2. データを間引き、I/O量を削減する
  3. OSシステムパラメタの変更
     - I/Oストライプサイズの変更
       - 大規模データサイズを1回I/Oする場合は、ストライプサイズを大きくする
  4. より高速なI/O方式を採用する
     - ファイル書き出しは、MPIプロセスごとに別名を付け、同時にI/O出力する実装であることが多い
     - 高速なファイルI/O（Parallel I/O、MPI-IOなど）を使う
       - 複数のファイルを1つに見せることができる

ホットスポットをどのようにして知るのか

1. プログラム中にタイマを設定して調べる
2. 性能プロファイラを利用する

- 演算ボトルネック
  - プロファイラの基本機能により調査可能
  - ループごとの詳細プロファイルにより、ハードウェア性能（キャッシュヒット率など）を調査可能
    - 例）日立pmpr、富士通　基本プロファイラ、など
- 通信ボトルネック
  - プロファイラの基本機能により調査可能
    - 例）富士通　基本プロファイラ、など
- I/Oボトルネック
  - 一般にあまり提供されていない
  - スパコンベンダーによっては専用プロファイラを提供している
    - 例）Cray社のプロファイラ（CrayPat Performance Analysis Tool）

その他の注意

- I/Oを行うため、プロセス0にデータを集積し、プロセス0のみがI/Oをするプログラム
  - データ集積のために、MPI_AllgatherV関数などが使われる
  - I/Oのための通信時間が占める割合が大きくなる
    - ノード数が増えるほど、上記のI/O時間の割合は大きくなる

⇒超並列向きではない実装

- I/Oは、プロセスごとに並列に行うほうが良い
  - ただし、プロセスごとに分散されて生成されるファイルの扱いが問題になる
  - できるだけ、MPI-IOや、その他のシステムソフトウェア提供の機能を使い、プロセスごとにファイルを見せない実装がよい

性能プロファイリング

性能プロファイリングの重要性

- プログラムにおいて、どの箇所（手続き（関数））に時間がかかっているか調べないと、チューニングを行っても効果がない
  - 手続きA:100秒、手続きB：10秒、手続きC：1秒、全体：111秒
  - 手続きAは全体時間の90%なので、これをチューニングすべき
- 性能プロファイルを行うには、一般的には、スパコン提供メーカが提供しているプロファイラを使うとよい
  - 多くは、コンパイラと連携している
    1. コンパイラオプションで指定し、実行可能コードを生成
    2. 実行可能コードを実行
    3. 性能プロファイルのためのファイル（ログファイル）が作成される
    4. 専用のコマンドを実行する

性能プロファイラでわかること

- 性能プロファイラツールに大きく依存
- ノード内性能
  - 全体実行時間に占める、各手続き（関数）の割合
  - MFLOPS（GFLOPS）値
  - キャッシュヒット率
  - スレッド並列化の効率（負荷バランス）
  - I/O時間が占める割合
- ノード間性能
  - MPIなどの通信パターン、通信量、通信回数
    （多くは専用のGUIで見る）

性能プロファイラ（富士通FX10）

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

性能プロファイラの種類の詳細

- 基本プロファイラ
  - コマンド例：fipp –C
  - 表示コマンド：fipppx、GUI（WEB経由）
  - ユーザプログラムに対し一定間隔(デフォルト時100 ミリ秒間隔)毎に割り込みをかけ情報を収集する。
  - 収集した情報を基に、コスト情報等の分析結果を表示。
- 詳細プロファイラ
  - コマンド例：fapp –C
  - 表示コマンド：GUI(WEB経由)
  - ユーザプログラムの中に測定範囲を設定し、測定範囲のハードウェアカウンタの値を収集。
  - 収集した情報を基に、MFLOPS、MIPS、各種命令比率、キャッシュミス等の詳細な分析結果を表示。

基本プロファイラ利用例（東大FX10）

- プロファイラデータ用の空のディレクトリがないとダメ
- 調べるべきプログラムのあるディレクトリに
  Profディレクトリを作成
  $ mkdir Prof
- wa2(対象の実行可能ファイル) の wa2-pure.bash 中に以下を記載
  fipp -C -d Prof mpirun ./wa2
- 実行する
  $ pjsub wa2-pure.bash
- テキストプロファイラを起動
  $ fipppx –A -d Prof

基本プロファイラ出力例（東大FX10）（1/2）

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

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

各MPIプロセスの
経過時間、ユーザ時間、システム時間

基本プロファイラ出力例（東大FX10）(2/2)

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

各関数の実行時間が、全体時間に占める割合

具体的な箇所と、ソースコード上の行数の情報

詳細プロファイラ利用例（東大FX10）

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

詳細プロファイラ利用例（東大FX10）

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

詳細プロファイラGUIによる表示例（東大FX10）

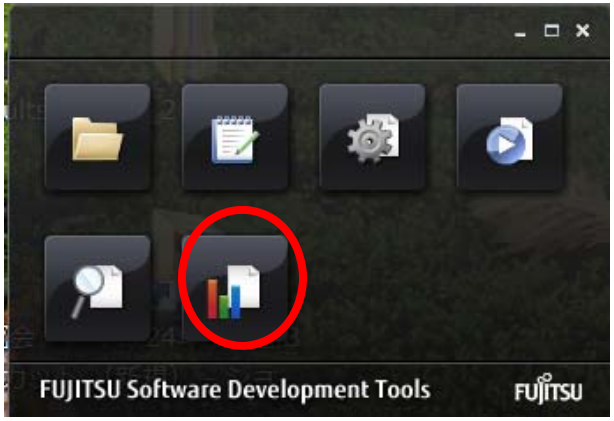
- プログラミング支援ツール(FUJITSU Software Development Tools Version 1.2.1 for Windows) をインストール
  - 以下をアクセス

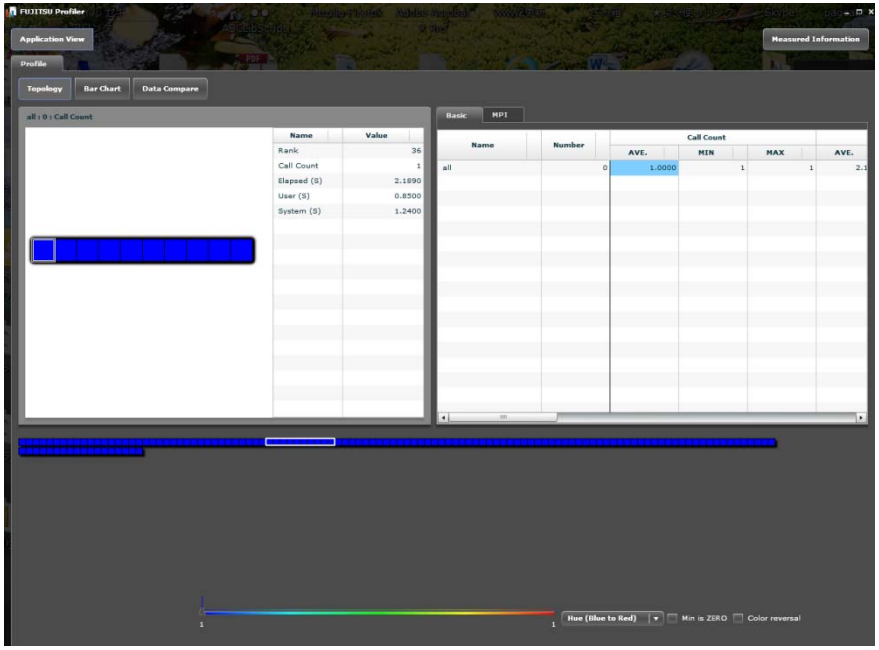
https://oakleaf-fx-1.cc.u-tokyo.ac.jp/fsdtfx10tx/install/index.html

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

詳細プロファイラGUIによる表示例（東大FX10）

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



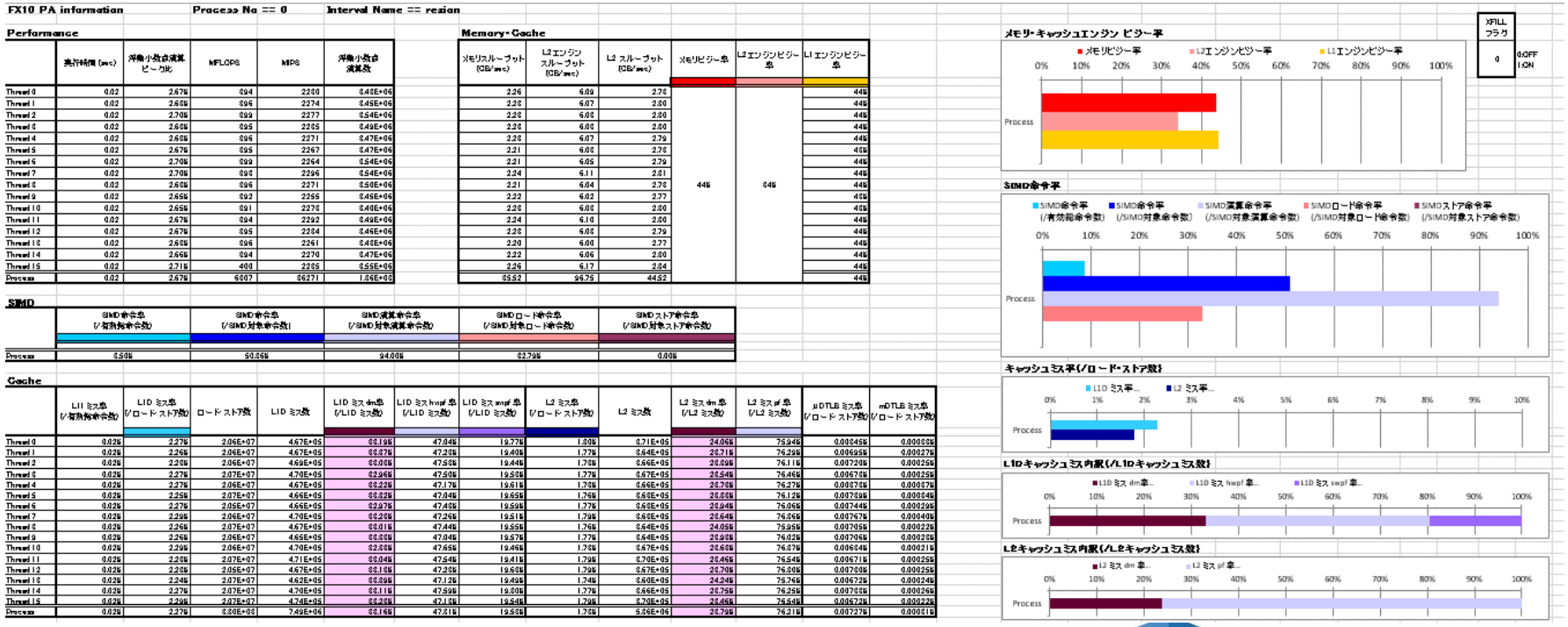


詳細プロファイラで取れるデータ（東大FX10）

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

詳細プロファイラ（Excel形式）

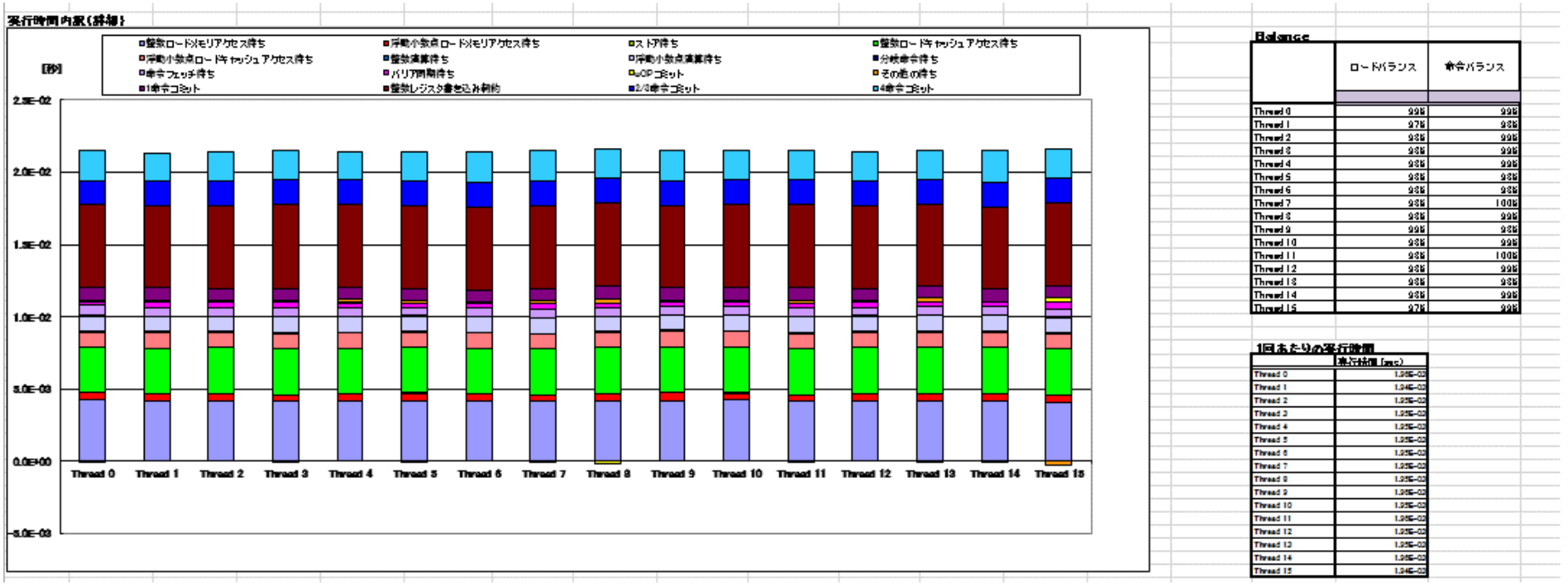
- 富士通FX10には、性能プロファイラによる結果をExcel形式で可視化できるツールがある
  - 演算ピーク比率(%)、MFLOPS、MIPS、メモリスループット(GB/s.)、L1キャッシュミス率(%)、TLBミスヒット率(%)、などのデータなどが、まとめて出力される
  - メモリビジー率、などが可視化されて出力される



詳細プロファイラ（Excel形式）

- 実行時間の内訳について、
  - メモリへのデータロード／ストア待ち時間（浮動小数点、整数）
  - レジスタへの書き込み時間

  などの時間が占める割合が、可視化して表示される。



そのほかの最適化技法
ループ分割、ループ融合とスレッド並列化

ループ分割とループ融合の実例（その１）

- Seism3D：
  東京大学古村教授が開発した地震波のシミュレーション
  プログラム（における、ベンチマークプログラム）
  - 東京大学情報基盤センターで開発中の
    数値計算ミドルウェアppOpen-HPCにおける
    ppOpen-APPL/FDMとして開発中
- 有限差分法（Finite Differential Method (FDM)）
- ３次元シミュレーション
  - ３次元配列が確保される
- データ型: 単精度 (real*4)

ループ分割とループ融合の実例（その２）

- 作業領域が多数必要
  - 最大問題サイズ: NX=257, NY=256, NZ=128
    （32GBメモリ）
    - たった 32.1MB分しか問題空間として確保できない
      - ほとんどのデータは、キャッシュに載ってしまう
- 近年のマルチコア計算機の傾向
  - L3キャッシュ（Last Level Cache, LLC）が
    大きくなってきている
    - Xeon E5-2670, Sandy Bridge
    - LLC: 20MB [L3/socket]

    ⇒問題空間の配列データが小さい時、キャッシュ上にデータが
    のりやすくなってきている

ＦＸ１０での基本プロファイルによる全体時間

- 1ノード8コア実行

```
********************************************************************************
  Application - procedures
********************************************************************************
        Cost        %  Operation (S)    Start      End
------------------------------------------------------------
        4904   100.0000       490.4783       --       --   Application
------------------------------------------------------------
         874    17.8222        87.4140       49      192   ppohfdm_velocity.ppohfdm_passing_velocity_
         517    10.5424        51.7083      128      173   ppohfdm_stress.ppohfdm_update_stress_
         476     9.7064        47.6076      213      353   ppohfdm_stress.ppohfdm_passing_stress_
         388     7.9119        38.8062      195      225   ppohfdm_velocity.ppohfdm_update_vel_
         370     7.5449        37.0059      176      210   ppohfdm_stress.ppohfdm_update_stress_sponge_
         274     5.5873        27.4044      199      226   ppohfdm_pfd3d.ppohfdm_pdiffz3_p4_
         274     5.5873        27.4044      169      196   ppohfdm_pfd3d.ppohfdm_pdiffy3_m4_
         247     5.0367        24.7039      139      166   ppohfdm_pfd3d.ppohfdm_pdiffy3_p4_
         236     4.8124        23.6038      229      256   ppohfdm_pfd3d.ppohfdm_pdiffz3_m4_
         218     4.4454        21.8035      108      136   ppohfdm_pfd3d.ppohfdm_pdiffx3_m4_
```

ＦＸ１０基本プロファイルによる全体時間(通信時間)

- Iノード8コア実行

MPI	%	Communication (S)	Start	End	
603	12.2961	60.3096	--	--	Application
503	57.5515	50.3080	49	192	ppohfdm_velocity.ppohfdm_passing_velocity_
0	0.0000	0.0000	128	173	ppohfdm_stress.ppohfdm_update_stress_
85	17.8571	8.5014	213	353	ppohfdm_stress.ppohfdm_passing_stress_

- 49行～192行　ppohfdm_velocity.ppohfdm_passing_velocity_は、多くの時間が通信時間　＝　50.3[sec.]/87.4[sec.](演算プロファイルから) * 100 = 57.5%（MPI_Isend, MPI_Irecv）、
  あと(42.5%)はメッセージのパッキングと受信データのアンパッキング(コピー時間)
- 213行～353行　ppohfdm_stress.ppohfdm_passing_stress_　の通信時間　＝ 8.5[sec.]/47.6[sec.](演算プロファイルから) * 100 = 17.8%、
  あと(82.2%)はメッセージのパッキングと受信データのアンパッキング(コピー時間)
- 上記(コピー時間の予測)は、対応するソースコードの場所を見ることで判明

ＦＸ１０基本プロファイルによる主要関数（プロセス毎）

- 1ノード8コア実行（プロセス4のログ）

```
        Cost       %  Operation (S)   Start     End
  ------------------------------------------------------
         629  100.0000       62.9100      --      --  Process     4
  ------------------------------------------------------
         160   25.4372       16.0025      49     192  ppohfdm_velocity.ppohfdm_passing_velocity_
          64   10.1749        6.4010     213     353  ppohfdm_stress.ppohfdm_passing_stress_
          62    9.8569        6.2010     128     173  ppohfdm_stress.ppohfdm_update_stress_
          43    6.8362        4.3007     176     210
ppohfdm_stress.ppohfdm_update_stress_sponge_
          39    6.2003        3.9006     195     225  ppohfdm_velocity.ppohfdm_update_vel_
          37    5.8824        3.7006     139     166  ppohfdm_pfd3d.ppohfdm_pdiffy3_p4_
          33    5.2464        3.3005     199     226  ppohfdm_pfd3d.ppohfdm_pdiffz3_p4_
          32    5.0874        3.2005     229     256  ppohfdm_pfd3d.ppohfdm_pdiffz3_m4_
          30    4.7695        3.0005      79     105  ppohfdm_pfd3d.ppohfdm_pdiffx3_p4_
          28    4.4515        2.8004     108     136  ppohfdm_pfd3d.ppohfdm_pdiffx3_m4_
```

通信関連処理

主要カーネル（第1位）：全体の9.8%

subroutine ppohFDM_update_stress (ファイル名：m_stress.f90)

```
do k = NZ00, NZ01
    do j = NY00, NY01
      do i = NX00, NX01
        RL1   = LAM (I,J,K)
        RM1   = RIG (I,J,K)
        RM2   = RM1 + RM1
        RLRM2 = RL1+RM2
        DXVX1 = DXVX(I,J,K)
        DYVY1 = DYVY(I,J,K)
        DZVZ1 = DZVZ(I,J,K)
        D3V3  = DXVX1 + DYVY1 + DZVZ1
        DXVYDYVX1 = DXVY(I,J,K)+DYVX(I,J,K)
        DXVZDZVX1 = DXVZ(I,J,K)+DZVX(I,J,K)
        DYVZDZVY1 = DYVZ(I,J,K)+DZVY(I,J,K)
        SXX (I,J,K) = SXX (I,J,K) + (RLRM2*(D3V3)-RM2*(DZVZ1+DYVY1) ) * DT
        SYY (I,J,K) = SYY (I,J,K)  + (RLRM2*(D3V3)-RM2*(DXVX1+DZVZ1) ) * DT
        SZZ (I,J,K) = SZZ (I,J,K)  + (RLRM2*(D3V3)-RM2*(DXVX1+DYVY1) ) * DT
        SXY (I,J,K) = SXY (I,J,K) + RM1 * DXVYDYVX1 * DT
        SXZ (I,J,K) = SXZ (I,J,K) + RM1 * DXVZDZVX1 * DT
        SYZ (I,J,K) = SYZ (I,J,K) + RM1 * DYVZDZVY1 * DT
      end do
    end do
  end do
```

主要カーネル（第2位）：全体の6.8%

subroutine ppohFDM_update_stress_sponge (ファイル名：m_stress.f90)

```
do k = NZ00, NZ01
    gg_z = gz(k)
    do j = NY00, NY01
      gg_y = gy(j)
      gg_yz = gg_y * gg_z
      do i = NX00, NX01
        gg_x = gx(i)
        gg_xyz = gg_x * gg_yz
        SXX(I,J,K)   = SXX(I,J,K) * gg_xyz
        SYY(I,J,K)   = SYY(I,J,K) * gg_xyz
        SZZ(I,J,K)   = SZZ(I,J,K) * gg_xyz
        SXY(I,J,K)   = SXY(I,J,K) * gg_xyz
        SXZ(I,J,K)   = SXZ(I,J,K) * gg_xyz
        SYZ(I,J,K)   = SYZ(I,J,K) * gg_xyz
      end do
    end do
  end do
```

主要カーネル（第3位）：全体の6.2%

subroutine ppohFDM_update_vel (ファイル名：m_velocity.f90)

```
  do k = NZ00, NZ01
     do j = NY00, NY01
       do i = NX00, NX01
         ! Effective Density
         ROX = 2.0_PN/( DEN(I,J,K) + DEN(I+1,J,K) )
         ROY = 2.0_PN/( DEN(I,J,K) + DEN(I,J+1,K) )
         ROZ = 2.0_PN/( DEN(I,J,K) + DEN(I,J,K+1) )
        VX(I,J,K) =VX(I,J,K)  +
( DXSXX(I,J,K)+DYSXY(I,J,K)+DZSXZ(I,J,K) )*ROX*DT
         VY(I,J,K) =VY(I,J,K)  +
( DXSXY(I,J,K)+DYSYY(I,J,K)+DZSYZ(I,J,K) )*ROY*DT
         VZ(I,J,K) =VZ(I,J,K)  +
( DXSXZ(I,J,K)+DYSYZ(I,J,K)+DZSZZ(I,J,K) )*ROZ*DT
       end do
     end do
   end do
```

主要カーネル（第4位）：全体の5.8%

subroutine ppohFDM_pdiffy3_p4 (ファイル名：m_pfd3d.f90)

```
R40 = C40/DY
 R41 = C41/DY

 do K = 1, NZ
   do I = 1, NX
     do J = 1, NY
       DYV (I,J,K) = (V(I,J+1,K)-V(I,J,K)  )*R40 - (V(I,J+2,K)-V(I,J-1,K))*R41
     end do
   end do
 end do
```

カーネルループの構造

- 以下の3重ループを検討する
  （ppOpen-APPL/FDMの第1位ループと同等）

```
DO K = 1, NZ
DO J = 1, NY
DO I = 1,  NX
    RL  = LAM (I,J,K)
    RM  = RIG (I,J,K)
    RM2 = RM + RM
    RMAXY = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I+1,J,K)  + 1.0/RIG(I,J+1,K) + 1.0/RIG(I+1,J+1,K))
    RMAXZ = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I+1,J,K)  + 1.0/RIG(I,J,K+1) + 1.0/RIG(I+1,J,K+1))
    RMAYZ = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I,J+1,K)  + 1.0/RIG(I,J,K+1) + 1.0/RIG(I,J+1,K+1))
    RLTHETA = (DXVX(I,J,K)+DYVY(I,J,K)+DZVZ(I,J,K))*RL
    QG  = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
    SXX (I,J,K) = ( SXX (I,J,K)  + (RLTHETA + RM2*DXVX(I,J,K))*DT )*QG
    SYY (I,J,K) = ( SYY (I,J,K)  + (RLTHETA + RM2*DYVY(I,J,K))*DT )*QG
    SZZ (I,J,K) = ( SZZ (I,J,K)  + (RLTHETA + RM2*DZVZ(I,J,K))*DT )*QG
    SXY (I,J,K) = ( SXY (I,J,K)  + (RMAXY*(DXVY(I,J,K)+DYVX(I,J,K)))*DT )*QG
    SXZ (I,J,K) = ( SXZ (I,J,K)  + (RMAXZ*(DXVZ(I,J,K)+DZVX(I,J,K)))*DT )*QG
    SYZ (I,J,K) = ( SYZ (I,J,K)  + (RMAYZ*(DYVZ(I,J,K)+DZVY(I,J,K)))*DT )*QG
END DO
END DO
END DO
```

ここでのコード最適化の方針（その１）

- ループ分割 (Loop Splitting)
  - スピルコード（レジスタから追い出されるデータがあるコード）を防ぐ目的で行う。
    - レジスタを最大限に使うプログラムで、メモリからのデータ読み出しを削減し、高速化する。

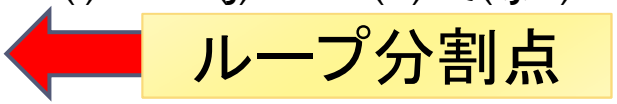
ここでのコード最適化の方針（その２）

- ループ融合 (Loop Fusion) (ループ1重化（Loop Collapse))
  - 対象は3重ループ　→ 以下の2つの方針がある
  - 1次元ループ化
    - スレッド並列実行のため、最外側のループ長を増加させる目的で行う
    - ベクトル計算機用のコンパイラで行われることが多い
    - メニーコア計算機でも状況により効果が見込まれる
  - 2次元ループ化
    - スレッド並列実行のため、最外側のループ長を増加させる目的で行う
    - コンパイラによる最内ループのプリフェッチ処理を増進
    - 近年のメニーコア計算機でもっとも有望と思われる方法

ループ分割の例 – 分割点

- 例:以下の箇所でループ分割する例

```
DO K = 1, NZ
DO J = 1, NY
DO I = 1,  NX
    RL(I)  = LAM (I,J,K)
    RM(I)  = RIG (I,J,K)
    RM2(I) = RM(I) + RM(I)
    RMAXY(I) = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I+1,J,K)  + 1.0/RIG(I,J+1,K) + 1.0/RIG(I+1,J+1,K))
    RMAXZ(I) = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I+1,J,K)  + 1.0/RIG(I,J,K+1) + 1.0/RIG(I+1,J,K+1))
    RMAYZ(I) = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I,J+1,K)  + 1.0/RIG(I,J,K+1) + 1.0/RIG(I,J+1,K+1))
    RLTHETA(I) = (DXVX(I,J,K)+DYVY(I,J,K)+DZVZ(I,J,K))*RL(I)
    QG(I) = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
END DO
DO I = 1, NX
    SXX (I,J,K) = ( SXX (I,J,K)  + (RLTHETA(I) + RM2(I)*DXVX(I,J,K))*DT )*QG(I)
    SYY (I,J,K) = ( SYY (I,J,K)  + (RLTHETA(I) + RM2(I)*DYVY(I,J,K))*DT )*QG(I)
    SZZ (I,J,K) = ( SZZ (I,J,K)  + (RLTHETA(I) + RM2(I)*DZVZ(I,J,K))*DT )*QG(I)
    SXY (I,J,K) = ( SXY (I,J,K)  + (RMAXY(I)*(DXVY(I,J,K)+DYVX(I,J,K)))*DT )*QG(I)
    SXZ (I,J,K) = ( SXZ (I,J,K)  + (RMAXZ(I)*(DXVZ(I,J,K)+DZVX(I,J,K)))*DT )*QG(I)
    SYZ (I,J,K) = ( SYZ (I,J,K)  + (RMAYZ(I)*(DYVZ(I,J,K)+DZVY(I,J,K)))*DT )*QG(I)
END DO
END DO
END DO
```



ループ融合 – １重ループ化

- 例）

利点：ループ長が増える
NZ → NZ*NY*NX

```
DO KK = 1, NZ * NY * NX
    K = (KK-1)/(NY*NX) + 1
    J = mod((KK-1)/NX,NY) + 1
    I = mod(KK-1,NX) + 1
    RL  = LAM (I,J,K)
    RM  = RIG (I,J,K)
    RM2 = RM + RM
    RMAXY = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I+1,J,K)  + 1.0/RIG(I,J+1,K) + 1.0/RIG(I+1,J+1,K))
    RMAXZ = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I+1,J,K)  + 1.0/RIG(I,J,K+1) + 1.0/RIG(I+1,J,K+1))
    RMAYZ = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I,J+1,K)  + 1.0/RIG(I,J,K+1) + 1.0/RIG(I,J+1,K+1))
    RLTHETA = (DXVX(I,J,K)+DYVY(I,J,K)+DZVZ(I,J,K))*RL
    QG  = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
    SXX (I,J,K) = ( SXX (I,J,K)  + (RLTHETA + RM2*DXVX(I,J,K))*DT )*QG
    SYY (I,J,K) = ( SYY (I,J,K)  + (RLTHETA + RM2*DYVY(I,J,K))*DT )*QG
    SZZ (I,J,K) = ( SZZ (I,J,K)  + (RLTHETA + RM2*DZVZ(I,J,K))*DT )*QG
    SXY (I,J,K) = ( SXY (I,J,K)  + (RMAXY*(DXVY(I,J,K)+DYVX(I,J,K)))*DT )*QG
    SXZ (I,J,K) = ( SXZ (I,J,K)  + (RMAXZ*(DXVZ(I,J,K)+DZVX(I,J,K)))*DT )*QG
    SYZ (I,J,K) = ( SYZ (I,J,K)  + (RMAYZ*(DYVZ(I,J,K)+DZVY(I,J,K)))*DT )*QG
END DO
```

ループ融合 – ２重ループ化

- 例）

利点：ループ長が増える
NZ → NZ*NY

このI-ループは連続：
コンパイラによるプリフェッチコード生成が可能

```
DO KK = 1, NZ * NY
  K = (KK-1)/NY +  1
  J = mod(KK-1,NY) + 1
  DO I = 1, NX
    RL  = LAM (I,J,K)
    RM  = RIG (I,J,K)
    RM2 = RM + RM
    RMAXY = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I+1,J,K)  + 1.0/RIG(I,J+1,K) + 1.0/RIG(I+1,J+1,K))
    RMAXZ = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I+1,J,K)  + 1.0/RIG(I,J,K+1) + 1.0/RIG(I+1,J,K+1))
    RMAYZ = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I,J+1,K)  + 1.0/RIG(I,J,K+1) + 1.0/RIG(I,J+1,K+1))
    RLTHETA = (DXVX(I,J,K)+DYVY(I,J,K)+DZVZ(I,J,K))*RL
    QG  = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
    SXX (I,J,K) = ( SXX (I,J,K)  + (RLTHETA + RM2*DXVX(I,J,K))*DT )*QG
    SYY (I,J,K) = ( SYY (I,J,K)  + (RLTHETA + RM2*DYVY(I,J,K))*DT )*QG
    SZZ (I,J,K) = ( SZZ (I,J,K)  + (RLTHETA + RM2*DZVZ(I,J,K))*DT )*QG
    SXY (I,J,K) = ( SXY (I,J,K)  + (RMAXY*(DXVY(I,J,K)+DYVX(I,J,K)))*DT )*QG
    SXZ (I,J,K) = ( SXZ (I,J,K)  + (RMAXZ*(DXVZ(I,J,K)+DZVX(I,J,K)))*DT )*QG
    SYZ (I,J,K) = ( SYZ (I,J,K)  + (RMAYZ*(DYVZ(I,J,K)+DZVY(I,J,K)))*DT )*QG
  ENDDO
END DO
```

さらなる改良：定義－参照距離の変更

```
DO K = 1, NZ
DO J = 1, NY
DO I = 1, NX
    RL  = LAM (I,J,K)
    RM  = RIG (I,J,K)
    RM2 = RM + RM
    RLTHETA = (DXVX(I,J,K)+DYVY(I,J,K)+DZVZ(I,J,K))*RL
    QG  = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
    SXX (I,J,K) = ( SXX (I,J,K)+ (RLTHETA + RM2*DXVX(I,J,K))*DT )*QG
    SYY (I,J,K) = ( SYY (I,J,K)+ (RLTHETA + RM2*DYVY(I,J,K))*DT )*QG
    SZZ (I,J,K) = ( SZZ (I,J,K) + (RLTHETA + RM2*DZVZ(I,J,K))*DT )*QG

    RMAXY = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I+1,J,K) + 1.0/RIG(I,J+1,K) + 1.0/RIG(I+1,J+1,K))
    RMAXZ = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I+1,J,K) + 1.0/RIG(I,J,K+1) + 1.0/RIG(I+1,J,K+1))
    RMAYZ = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I,J+1,K)  + 1.0/RIG(I,J,K+1) + 1.0/RIG(I,J+1,K+1))
    SXY (I,J,K) = ( SXY (I,J,K)   + (RMAXY*(DXVY(I,J,K)+DYVX(I,J,K)))*DT )*QG
    SXZ (I,J,K) = ( SXZ (I,J,K)   + (RMAXZ*(DXVZ(I,J,K)+DZVX(I,J,K)))*DT )*QG
    SYZ (I,J,K) = ( SYZ (I,J,K)   + (RMAYZ*(DYVZ(I,J,K)+DZVY(I,J,K)))*DT )*QG
END DO
END DO
END DO
```

T2K（AMD Opteron）で、約50%の速度向上

修正コード + I-ループ分割の例

```
DO K = 1, NZ
DO J = 1, NY
DO I = 1, NX
    RL  = LAM (I,J,K)
    RM  = RIG (I,J,K)
    RM2 = RM + RM
    RLTHETA  = (DXVX(I,J,K)+DYVY(I,J,K)+DZVZ(I,J,K))*RL
    QG  = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
    SXX (I,J,K) = ( SXX (I,J,K)  + (RLTHETA + RM2*DXVX(I,J,K))*DT )*QG
    SYY (I,J,K) = ( SYY (I,J,K)  + (RLTHETA + RM2*DYVY(I,J,K))*DT )*QG
    SZZ (I,J,K) = ( SZZ (I,J,K)   + (RLTHETA + RM2*DZVZ(I,J,K))*DT )*QG
ENDDO
DO I = 1, NX
    STMP1 = 1.0/RIG(I,J,K)
    STMP2 = 1.0/RIG(I+1,J,K)
    STMP4 = 1.0/RIG(I,J,K+1)
    STMP3 = STMP1 + STMP2
    RMAXY = 4.0/(STMP3  + 1.0/RIG(I,J+1,K) + 1.0/RIG(I+1,J+1,K))
    RMAXZ = 4.0/(STMP3  + STMP4 + 1.0/RIG(I+1,J,K+1))
    RMAYZ = 4.0/(STMP3  + STMP4 + 1.0/RIG(I,J+1,K+1))
    QG  = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
    SXY (I,J,K) = ( SXY (I,J,K) + (RMAXY*(DXVY(I,J,K)+DYVX(I,J,K)))*DT )*QG
    SXZ (I,J,K) = ( SXZ (I,J,K) + (RMAXZ*(DXVZ(I,J,K)+DZVX(I,J,K)))*DT )*QG
    SYZ (I,J,K) = ( SYZ (I,J,K)  + (RMAYZ*(DYVZ(I,J,K)+DZVY(I,J,K)))*DT )*QG
END DO
END DO
END DO
```

ループ分割すると、
QGの再計算が必要になる

通常のコンパイラでは
ユーザの判断が必要
なので、できない

修正コード + K-ループの分割の例

```
DO K = 1, NZ
DO J = 1, NY
DO I = 1, NX
    RL  = LAM (I,J,K)
    RM  = RIG (I,J,K)
    RM2 = RM + RM
    RLTHETA  = (DXVX(I,J,K)+DYVY(I,J,K)+DZVZ(I,J,K))*RL
    QG  = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
    SXX (I,J,K) = ( SXX (I,J,K)  + (RLTHETA + RM2*DXVX(I,J,K))*DT )*QG
    SYY (I,J,K) = ( SYY (I,J,K)  + (RLTHETA + RM2*DYVY(I,J,K))*DT )*QG
    SZZ (I,J,K) = ( SZZ (I,J,K)   + (RLTHETA + RM2*DZVZ(I,J,K))*DT )*QG
ENDDO;  ENDDO; ENDDO
```

完全に別の３重ループに分かれる
←分かれた３重ループに対し、
　コンパイラによるさらなる最適化の可能性

```
DO K = 1, NZ
DO J = 1, NY
DO I = 1, NX
    STMP1 = 1.0/RIG(I,J,K)
    STMP2 = 1.0/RIG(I+1,J,K)
    STMP4 = 1.0/RIG(I,J,K+1)
    STMP3 = STMP1 + STMP2
    RMAXY = 4.0/(STMP3  + 1.0/RIG(I,J+1,K) + 1.0/RIG(I+1,J+1,K))
    RMAXZ = 4.0/(STMP3  + STMP4 + 1.0/RIG(I+1,J,K+1))
    RMAYZ = 4.0/(STMP3  + STMP4 + 1.0/RIG(I,J+1,K+1))
    QG  = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
    SXY (I,J,K) = ( SXY (I,J,K) + (RMAXY*(DXVY(I,J,K)+DYVX(I,J,K)))*DT )*QG
    SXZ (I,J,K) = ( SXZ (I,J,K) + (RMAXZ*(DXVZ(I,J,K)+DZVX(I,J,K)))*DT )*QG
    SYZ (I,J,K) = ( SYZ (I,J,K)  + (RMAYZ*(DYVZ(I,J,K)+DZVY(I,J,K)))*DT )*QG
END DO;  END DO;  END DO;
```

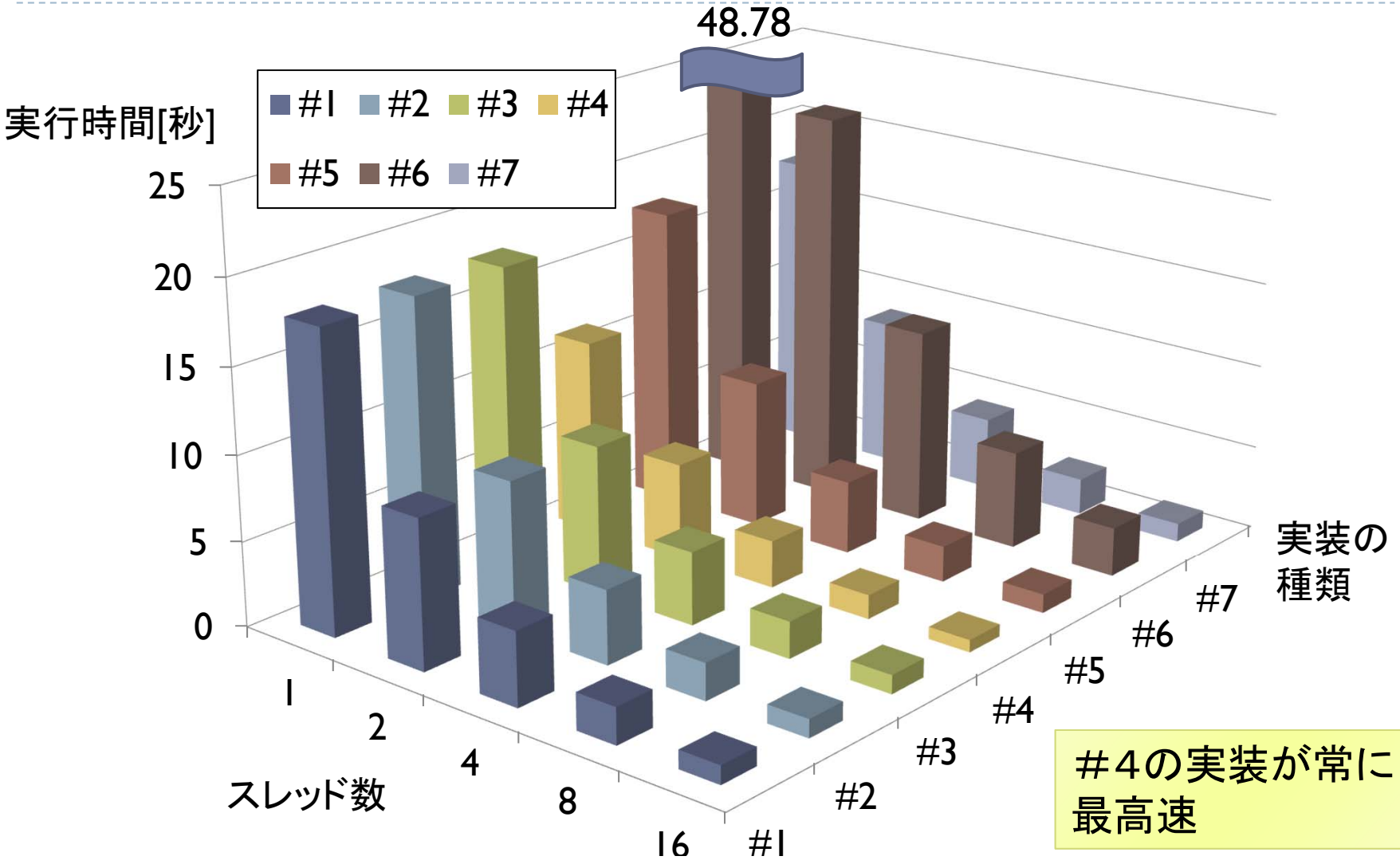
チューニングの可能性のあるコード例（経験的に決めた数例について）

- #1 : 基の３重ループコード（ベースライン）
- #2: I-ループ分割のみ
- #3: J-ループ分割のみ
- #4: K-ループ分割のみ
- #5: #2ループに対するループ融合 (2重ループ化)
- #6 : #1ループに対するループ融合(１重ループ化)
- #7 : #1ループに対するループ融合(2重ループ化)

ループ分割・ループ融合の効果

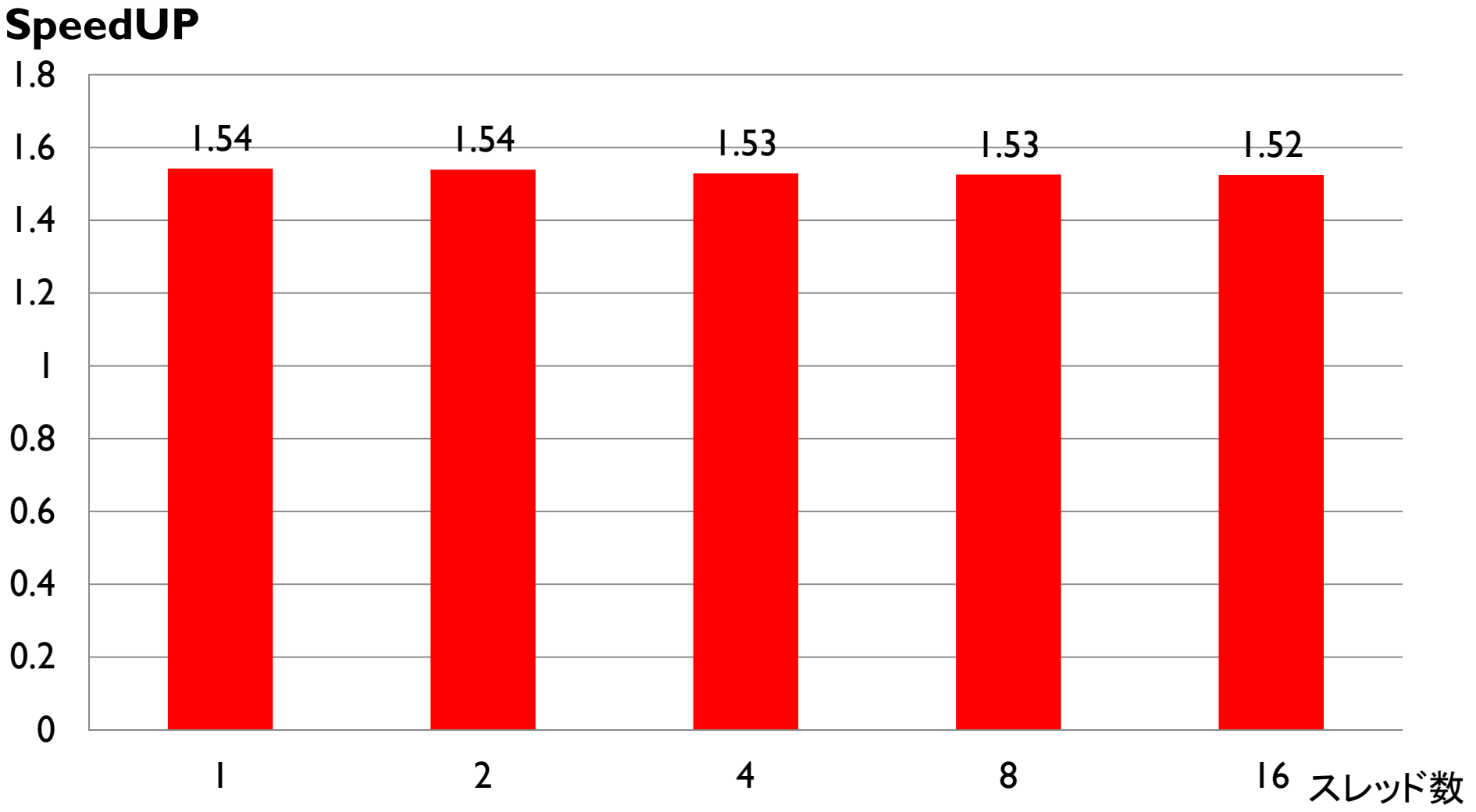
- 東京大学情報基盤センターFX10を利用
  - 1ノード、16スレッド
  - Sparc64 IV-fx (1.848 GHz)
- 最外ループに対して、OpenMPが適用可能
  - parallel do構文で並列化可能
- スレッド数は、1～16まで変更可能

チューニングの結果



#4の実装が常に最高速

＃１（ベースライン）に対する速度向上



#4のK-ループの分割のコード

```
!$omp parallel do private(k,j,i,STMP1,STMP2,STMP3,STMP4,RL,RM,RM2,
!$omp& RMAXY,RMAXZ,RMAYZ,RLTHETA,QG)
      DO K = 1, NZ
      DO J = 1, NY
      DO I = 1, NX
         RL  = LAM (I,J,K);  RM  = RIG (I,J,K); RM2 = RM + RM;
         RLTHETA  = (DXVX(I,J,K)+DYVY(I,J,K)+DZVZ(I,J,K))*RL
         QG  = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
         SXX (I,J,K) = ( SXX (I,J,K)  + (RLTHETA + RM2*DXVX(I,J,K))*DT )*QG
         SYY (I,J,K) = ( SYY (I,J,K)  + (RLTHETA + RM2*DYVY(I,J,K))*DT )*QG
         SZZ (I,J,K) = ( SZZ (I,J,K)   + (RLTHETA + RM2*DZVZ(I,J,K))*DT )*QG
      ENDDO;  ENDDO; ENDDO
!$omp end parallel do
!$omp parallel do private(k,j,i,STMP1,STMP2,STMP3,STMP4,RL,RM,RM2,
!$omp& RMAXY,RMAXZ,RMAYZ,RLTHETA,QG)
      DO K = 1, NZ
      DO J = 1, NY
      DO I = 1, NX
         STMP1 = 1.0/RIG(I,J,K);  STMP2 = 1.0/RIG(I+1,J,K); STMP4 = 1.0/RIG(I,J,K+1);
         STMP3 = STMP1 + STMP2
         RMAXY = 4.0/(STMP3  + 1.0/RIG(I,J+1,K) + 1.0/RIG(I+1,J+1,K))
         RMAXZ = 4.0/(STMP3  + STMP4 + 1.0/RIG(I+1,J,K+1))
         RMAYZ = 4.0/(STMP3  + STMP4 + 1.0/RIG(I,J+1,K+1))
         QG  = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
         SXY (I,J,K) = ( SXY (I,J,K) + (RMAXY*(DXVY(I,J,K)+DYVX(I,J,K)))*DT )*QG
         SXZ (I,J,K) = ( SXZ (I,J,K) + (RMAXZ*(DXVZ(I,J,K)+DZVX(I,J,K)))*DT )*QG
         SYZ (I,J,K) = ( SYZ (I,J,K)  + (RMAYZ*(DYVZ(I,J,K)+DZVY(I,J,K)))*DT )*QG
      END DO;  END DO;  END DO;
!$omp end parallel do
```

メニーコア環境でのループ融合への期待

- 一般に、3次元陽解法のカーネルは以下の構造
- OpenMPのスレッド並列化は最外側ループに適用
- この時、並列性はK-ループ長のNZで決まる

```
!$omp parallel do private(…)
DO K = 1, NZ
DO J = 1, NY
DO I = 1, NX
  ＜離散化手法に基づく数式＞
ENDDO
ENDDO
ENDDO
!$omp end parallel do
```

（NZ＞スレッド数） が並列性のため必要

- OpenMPオーバヘッドを考えると、ノードあたりのNZはスレッド数の2～3倍必要
  - 例）60スレッドなら、NZは120～180以上
  - HTで240スレッド実行なら、NZは480～720以上
- 3次元問題で上記のサイズ（ノード当たりの問題サイズ）は確保できるか？

確保できない場合はループ融合が必須

計算機環境
(Xeon Phiの８ノード)

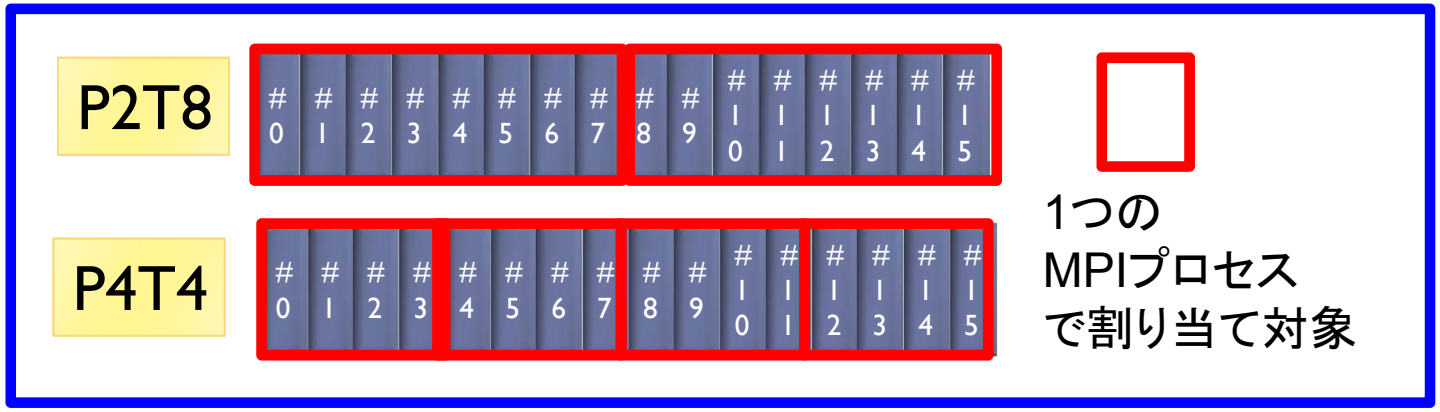
- Intel Xeon Phi
  - Xeon Phi 5110P (1.053 GHz), 60 cores
  - メモリ量：8 GB (GDDR5)
  - 理論ピーク性能：1.01 TFLOPS
  - Xeon Phiのクラスタ（ノード当たり1ボード）
  - InfiniBand FDR x 2 Ports
    - Mellanox Connect-IB
    - PCI-E Gen3 x16
    - 56Gbps x 2
    - 理論バンド幅 13.6GB/s
    - フルバイセクション
  - Intel MPI
    - MPICH2、MVAPICH2ベース
    - 4.1 Update 3 (build 048)
  - コンパイラ：Intel Fortran version 14.0.0.080 Build 20130728
  - コンパイラオプション：
    -ipo20 -O3 -warn all -openmp -mcmodel=medium -shared-intel -mmic -align array64byte
  - KMP_AFFINITY=granularity=fine, balanced (ソケット間へスレッドを均等割り当て)

実行詳細

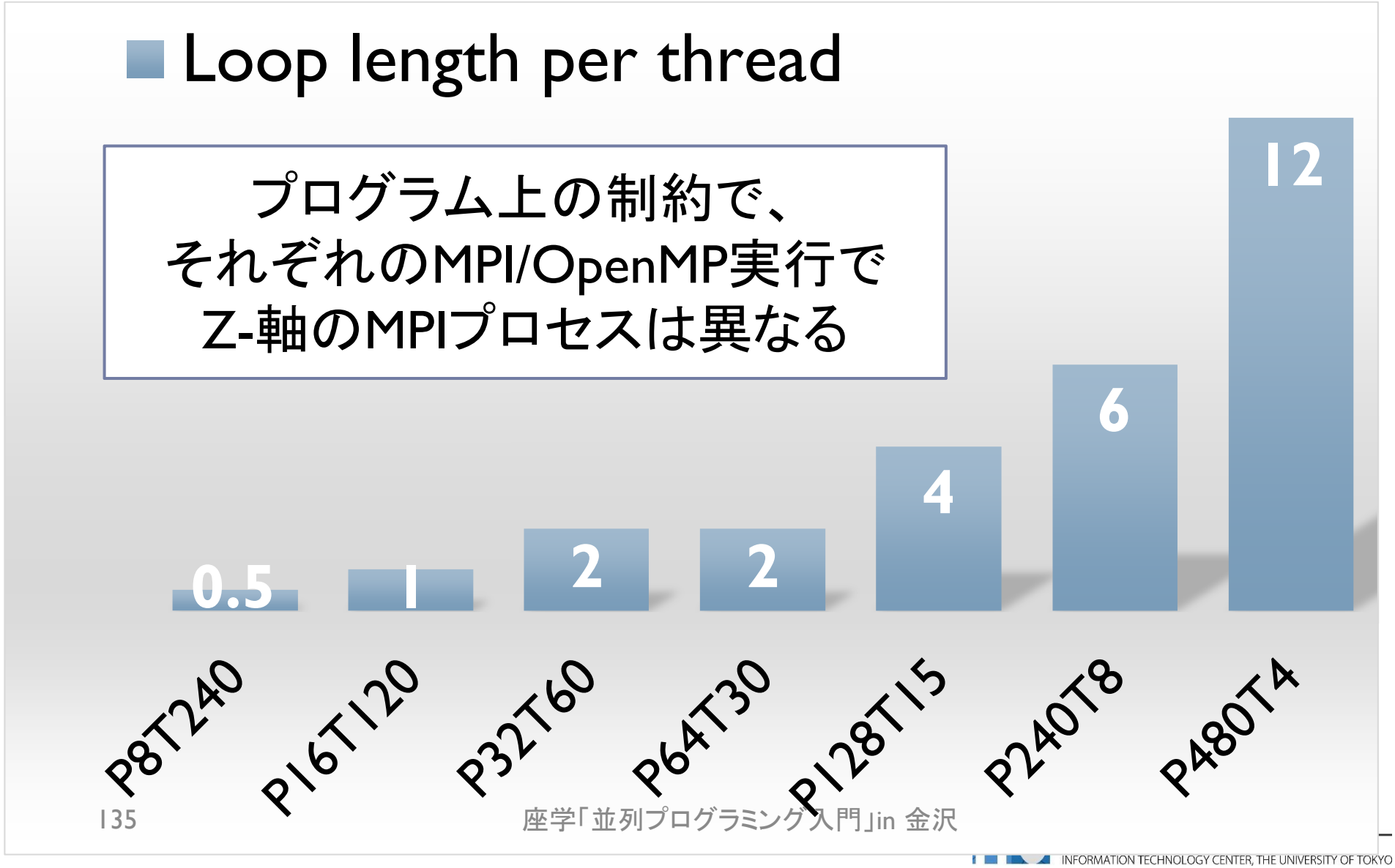
- ppOpen-APPL/FDM ver.0.2
- ppOpen-AT ver.0.2
- 時間ステップ数: 2000 steps
- ノード数: 8 ノード
- Native Mode 実行
- 問題サイズ
  (8 GB/ノードでの最大サイズ)
  - NX * NY * NZ = 1536 x 768 x 240 / 8ノード
  - NX * NY * NZ = 768 * 384 * 120 / ノード
    (MPIプロセス当たりのサイズではない)

ハイブリッドMPI/OpenMP実行の詳細

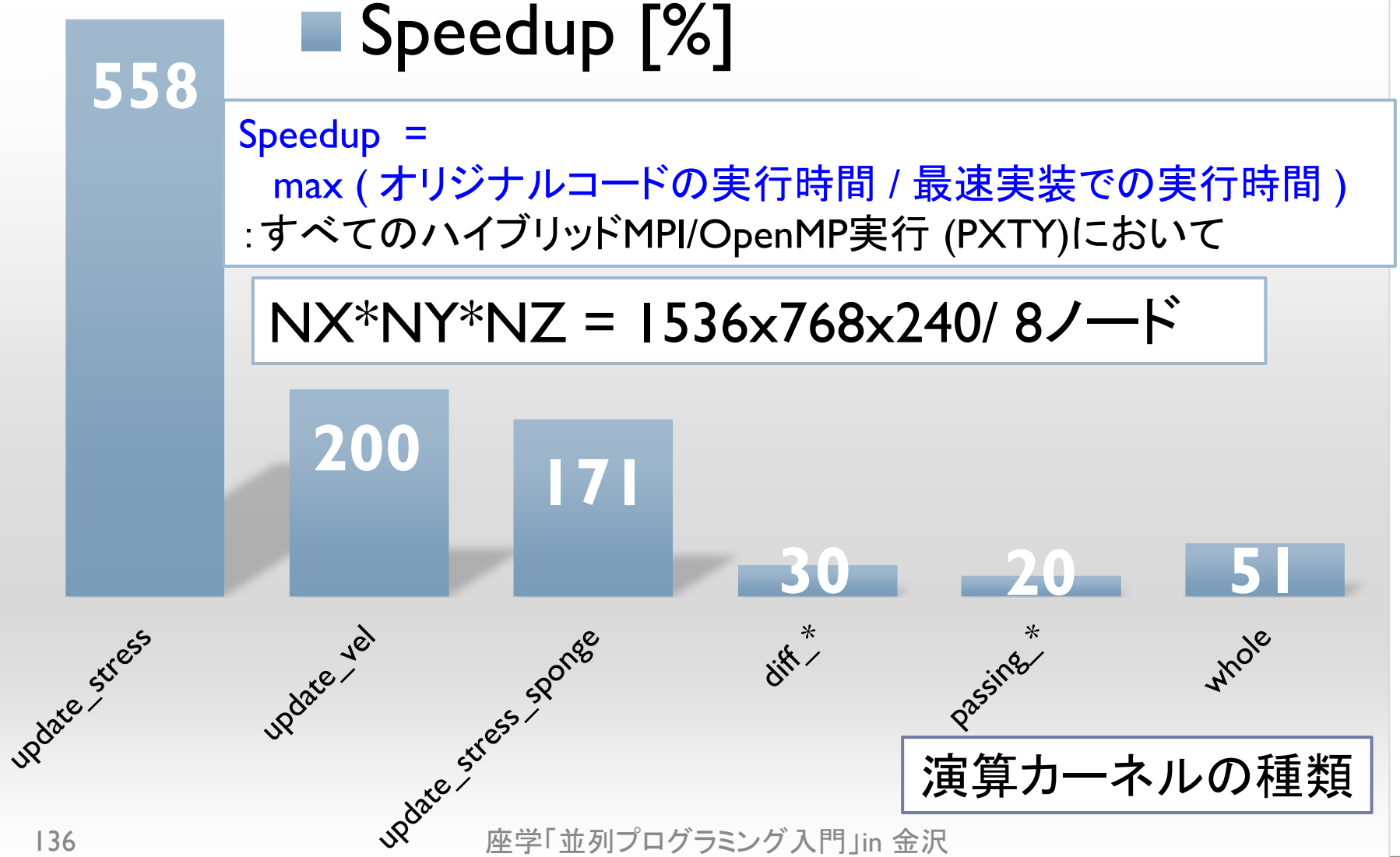
- Xeon PhiにおけるMPIプロセス数とOpenMPスレッド数
  - 4 HT (Hyper Threading)
  - PX TY: X MPIプロセス、Y スレッド/プロセス
    - P8T240 : 最少のハイブリッドMPI/OpenMP実行（ppOpen-APPL/FDMでは 8MPIプロセスが最低でも必要なため）
    - P16T120
    - P32T60
    - P64T30
    - P128T15
    - P240T8
    - P480T4
    - P960T2以下は、この環境ではMPIエラーで実行できなかったため除外



スレッド当たりのループ長 (Z-軸)
(8ノード、1536x768x240 / 8 ノード)



ループ融合等による最大の速度向上(Xeon Phi、８ノード)



Speedup =
max (オリジナルコードの実行時間 / 最速実装での実行時間)
: すべてのハイブリッドMPI/OpenMP実行 (PXTY)において

NX*NY*NZ = 1536x768x240/ 8ノード

最速のコード (update_stress)

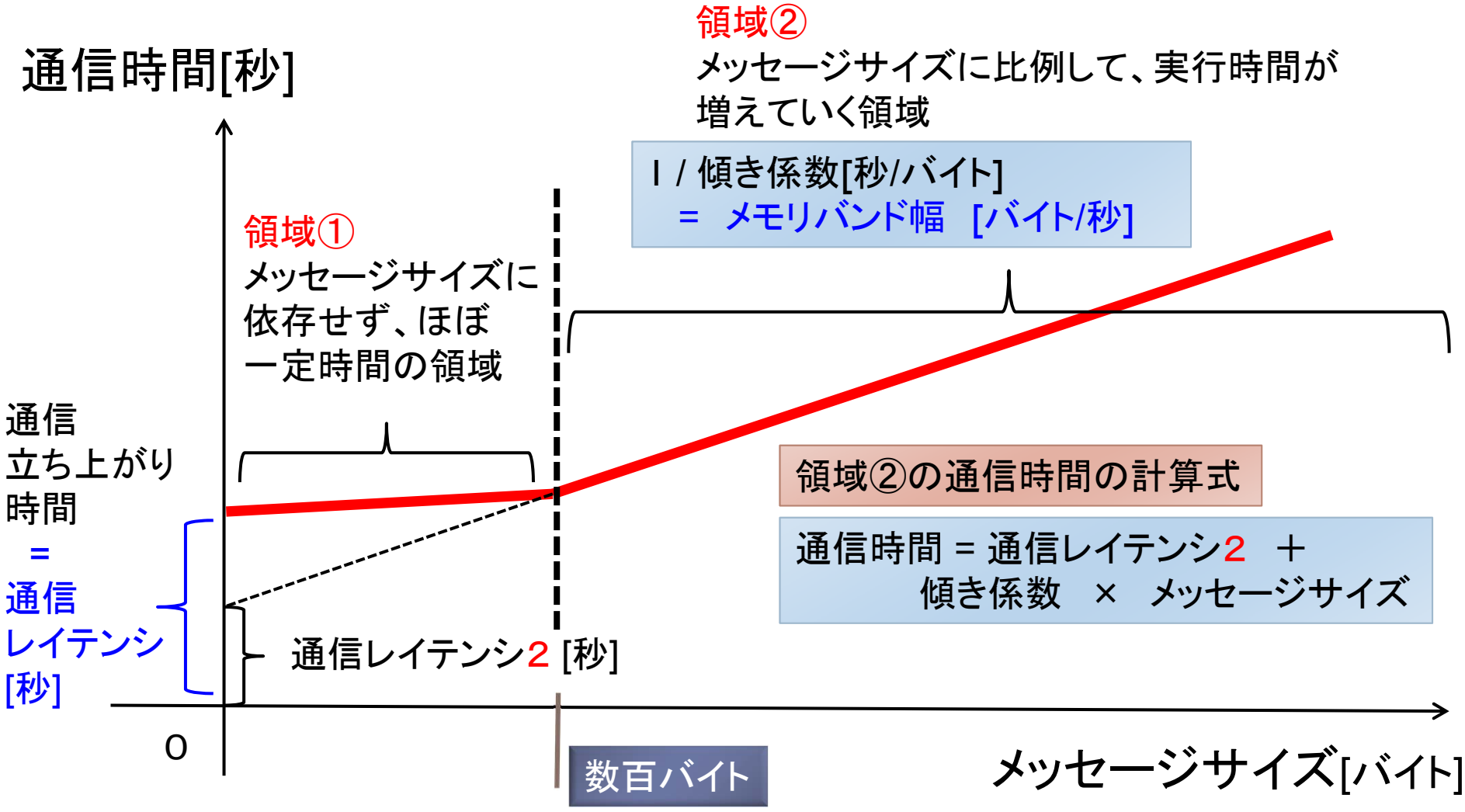
- Xeon Phi (P240T8)

```
!$omp parallel do private
(k,j,i,RL1,RM1,RM2,RLRM2,DXVX1,DYVY1,DZVZ1,D3V3,DXVYDYVX1,DXVZDZVX1,DYVZDZV1)
 DO k_j = 1, (NZ01-NZ00+1)*(NY01-NY00+1)
   k = (k_j-1)/(NY01-NY00+1) + NZ00
   j = mod((k_j-1),(NY01-NY00+1)) + NY00
    DO i = NX00, NX01
       RL1 = LAM (I,J,K);  RM1 = RIG (I,J,K)
       RM2 = RM1 + RM1; RLRM2 = RL1+RM2
       DXVX1 = DXVX(I,J,K); DYVY1 = DYVY(I,J,K);  DZVZ1 = DZVZ(I,J,K);
       D3V3  = DXVX1 + DYVY1 + DZVZ1;
       SXX (I,J,K) = SXX (I,J,K)  + (RLRM2*(D3V3)-RM2*(DZVZ1+DYVY1) ) * DT
       SYY (I,J,K) = SYY (I,J,K)  + (RLRM2*(D3V3)-RM2*(DXVX1+DZVZ1) ) * DT
       SZZ (I,J,K) = SZZ (I,J,K)  + (RLRM2*(D3V3)-RM2*(DXVX1+DYVY1) ) * DT
       DXVYDYVX1 = DXVY(I,J,K)+DYVX(I,J,K);
      DXVZDZVX1 = DXVZ(I,J,K)+DZVX(I,J,K);
       DYVZDZVY1 = DYVZ(I,J,K)+DZVY(I,J,K)
       SXY (I,J,K) = SXY (I,J,K) + RM1 * DXVYDYVX1 * DT
       SXZ (I,J,K) = SXZ (I,J,K) + RM1 * DXVZDZVX1 * DT
       SYZ (I,J,K) = SYZ (I,J,K) + RM1 * DYVZDZVY1 * DT
    END DO
 END DO
!$omp end parallel do
```

ループ融合により
ループ長
（＝並列性）が増加

通信最適化の方法

メッセージサイズと通信回数



通信最適化時に注意すること（その１）

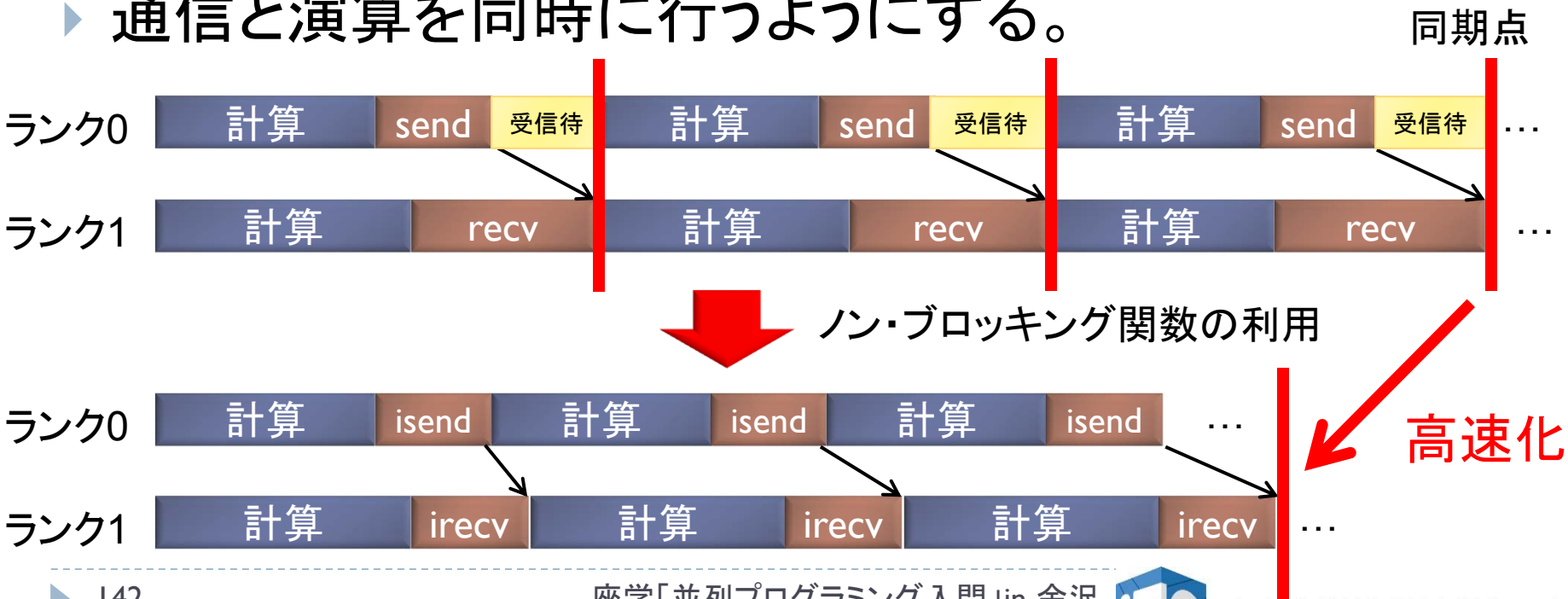
- 自分のアプリケーションの通信パターンについて、以下の観点を知らないと通信の最適化ができない
  - ＜領域①＞＜領域②＞のどちらになるのか
  - 通信の頻度（回数）はどれほどか
- 領域①の場合
  - 「通信レイテンシ」が実行時間のほとんど
  - 通信回数を削減する
    - 細切れに送っているデータをまとめて1回にする、など
- 領域②の場合
  - 「メッセージ転送時間」が実行時間のほとんど
  - メッセージサイズを削減する
  - 冗長計算をして計算量を増やしてでもメッセージサイズを削減する、など

領域①となる通信の例

- 内積演算のためのリダクション(MPI_Allreduce)などの送信データは倍精度１個分（8バイト）
- 8バイトの規模だと、数個分を同時にMPI_Allreduceする時間と、１個分をMPI_Allreduceをする時間は、ほぼ同じ時間となる
  - ⇒複数回分の内積演算を一度に行うと高速化される可能性あり
- 例）連立一次方程式の反復解法CG法中の内積演算
  - 通常の実装だと、１反復に3回の内積演算がある
  - このため、内積部分は通信レイテンシ律速となる
  - k反復を１度に行えば、内積に関する通信回数は1/k回に削減
    - ただし、単純な方法では、丸め誤差の影響で収束しない。
    - 通信回避CG法（Communication Avoiding CG, CACG）として現在活発に研究されている。

通信最適化時に注意すること（その２）

- 「同期点」を減らすことも高速化につながる
  - MPI関数の「ノン・ブロッキング関数」を使う
  - 例：　ブロッキング関数 MPI_SEND()
    → ノン・ブロッキング関数 MPI_ISEND()
  - 通信と演算を同時に行うようにする。

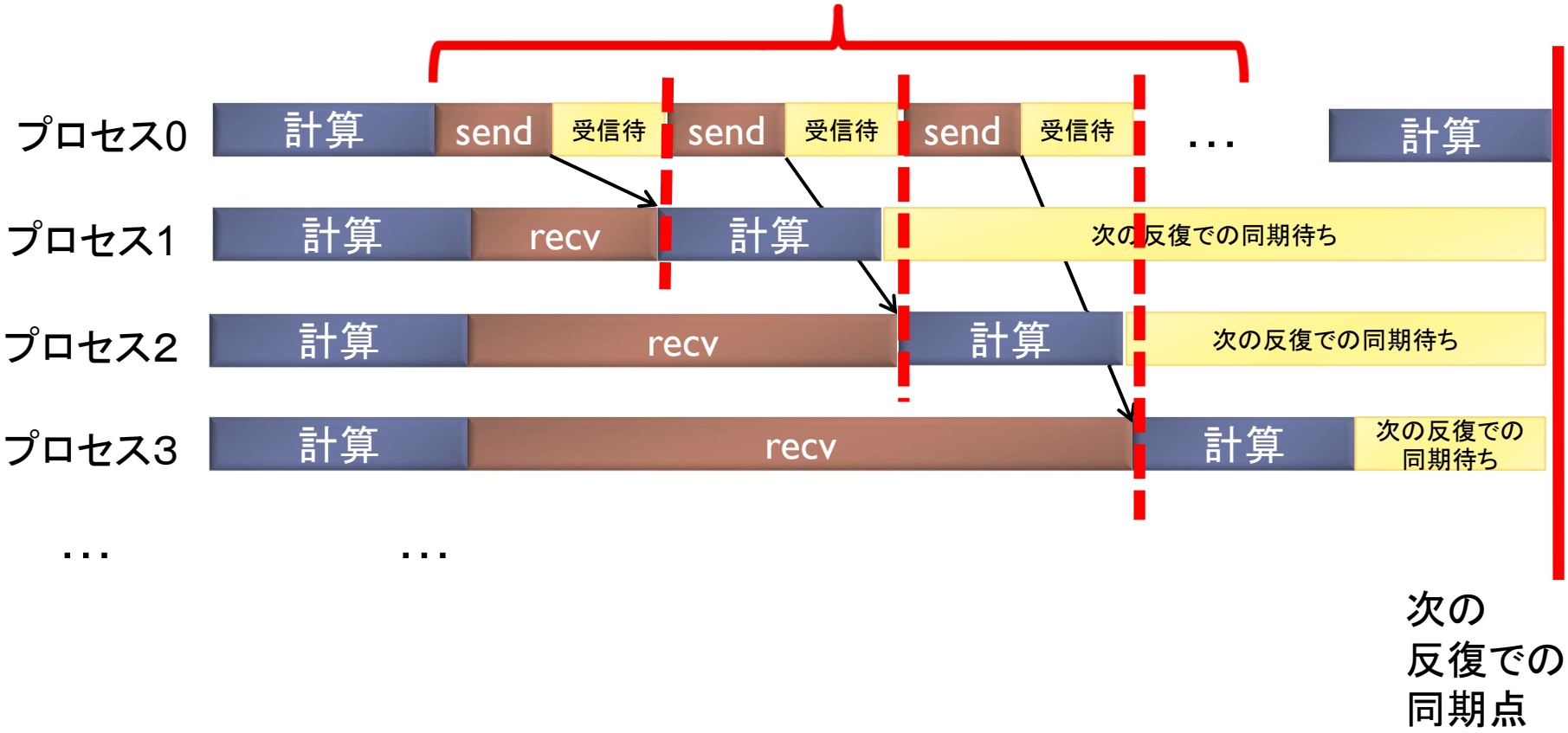


非同期通信：Isend、Irecv、永続的通信関数

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



１対１通信に対するMPI用語

ブロッキング？ノンブロッキング？

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
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

ローカル、ノンローカル

- ローカル
  - 手続きの完了が、それを実行しているプロセスのみに依存する。
  - ほかのユーザプロセスとの通信を必要としない処理。
- ノンローカル
  - 操作を完了するために、別のプロセスでの何らかのMPI手続きの実行が必要かもしれない。
  - 別のユーザプロセスとの通信を必要とするかもしれない処理。

通信モード（送信発行時の場合）

1. 標準通信モード（ノンローカル）：デフォルト
   - 送出メッセージのバッファリングはMPIに任せる。
     - バッファリングされるとき：相手の受信起動前に送信を完了可能；
     - バッファリングされないとき：送信が完全終了するまで待機；
2. バッファ通信モード（ローカル）
   - 必ずバッファリングする。バッファ領域がないときはエラー。
3. 同期通信モード（ノンローカル）
   - バッファ領域が再利用でき、かつ、対応する受信／送信が開始されるまで待つ。
4. レディ通信モード（処理自体はローカル）
   - 対応する受信が既に発行されている場合のみ実行できる。それ以外はエラー。
     - ハンドシェーク処理を無くせるため、高い性能を発揮する。

実例－MPI_Send

- MPI_Send関数
  - ブロッキング
  - 標準通信モード（ノンローカル）
    - バッファ領域が安全な状態になるまで戻らない
    - バッファ領域がとれる場合：
      メッセージがバッファリングされる。対応する受信が起動する前に、送信を完了できる。
    - バッファ領域がとれない場合：
      対応する受信が発行されて、かつ、メッセージが受信側に完全にコピーされるまで、送信処理を完了できない。

非同期通信関数

- 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

非同期通信関数

- 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。

実例－MPI_Isend

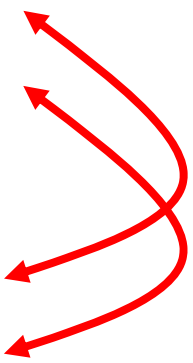
- MPI_Isend関数
  - ノンブロッキング
  - 標準通信モード（ノンローカル）
    - 通信バッファ領域の状態にかかわらず戻る
    - バッファ領域がとれる場合は、メッセージがバッファリングされ、対応する受信が起動する前に、送信処理が完了できる
    - バッファ領域がとれない場合は、対応する受信が発行され、メッセージが受信側に完全にコピーされるまで、送信処理が完了できない
    - MPI_Wait関数が呼ばれた場合の振舞いと理解すべき。

注意点

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

並列化の注意（MPI_Send、MPI_Recv）

- 全員がMPI_Sendを先に発行すると、その場所で処理が止まる。(cf. 標準通信モードを考慮)
  （正確には、動いたり、動かなかったり、する）
  - MPI_Sendの処理中で、場合により、バッファ領域がなくなる。
  - バッファ領域が空くまで待つ（スピンウェイトする）。
  - しかし、送信側バッファ領域不足から、永遠に空かない。
- これを回避するためには、例えば以下の実装を行う。
  - ランク番号が2で割り切れるプロセス：
    - MPI_Send();
    - MPI_Recv();
  - それ以外：
    - MPI_Recv();
    - MPI_Send();



それぞれに対応

非同期通信 TIPS

- メッセージを完全に受け取ることなく、受信したメッセージの種類を確認したい
  - 送信メッセージの種類により、受信方式を変えたい場合
  - MPI_Probe 関数（ブロッキング）
  - MPI_Iprobe 関数（ノンブロッキング）
  - MPI_Cancel 関数（ノンブロッキング、ローカル）

MPI_Probe 関数

- ierr = MPI_Probe(isource, itag, icomm, istatus) ;
  - isource: 整数型。送信元のランク。
    - MPI_ANY_SOURCE (整数型)も指定可能
  - itag: 整数型。タグ値。
    - MPI_ANY_TAG (整数型) も指定可能
  - icomm: 整数型。コミュニケータ。
  - istatus : ステータスオブジェクト。
  - isource, itagに指定されたものがある場合のみ戻る

MPI_Iprobe関数

- ierr = MPI_Iprobe(isource, itag, icomm, iflag, istatus) ;
  - isource: 整数型。送信元のランク。
    - MPI_ANY_SOURCE (整数型) も指定可能。
  - itag: 整数型。タグ値。
    - MPI_ANY_TAG (整数型) も指定可能。
  - icomm: 整数型。コミュニケータ。
  - iflag: 論理型。isource, itagに指定されたものがあった場合はtrueを返す。
  - istatus : ステータスオブジェクト。

MPI_Cancel 関数

- ierr = MPI_Cancel(irequest);
  - irequest: 整数型。通信要求(ハンドル)

- 目的とする通信が実際に取り消される以前に、可能な限りすばやく戻る。
- 取消しを選択するため、MPI_Request_free関数、MPI_Wait関数、又は MPI_Test関数（または任意の対応する操作）の呼出しを利用して完了されている必要がある。

ノン・ブロッキング通信例（C言語）

```
if (myid == 0)  {
      …
    for (i=1;  i<numprocs;  i++)  {
        ierr = MPI_Isend( &a[0],  N,  MPI_DOUBLE,  i,
            i_loop,  MPI_COMM_WORLD,  &irequest[i] );
    }
 }  else  {
    ierr = MPI_Recv( &a[0],  N,  MPI_DOUBLE,  0,  i_loop,
                MPI_COMM_WORLD,  &istatus );
}
          a[ ]を使った計算処理;
if (myid == 0)   {
     for (i=1;  i<numprocs;  i++)  {
         ierr = MPI_Wait(&irequest[i],  &istatus);
     }
}
```

ランク0のプロセスは、ランク1~numprocs-1までのプロセスに対して、ノンブロッキング通信を用いて、長さNのDouble型配列データを送信

ランク1~numprocs-1までのプロセスは、ランク0からの受信待ち。

プロセス0は、recvを待たず計算を開始

ランク0のPEは、ランク1~numprocs-1までのプロセスに対するそれぞれの送信に対し、それぞれが受信完了するまでビジーウェイト（スピンウェイト）する。

ノン・ブロッキング通信の例（Fortran言語）

```
if (myid .eq. 0)  then
      …
    do i=1, numprocs - 1
       call MPI_ISEND( a,  N,  MPI_DOUBLE_PRECISION,
          i,  i_loop,  MPI_COMM_WORLD,  irequest, ierr )
    enddo
else
       call  MPI_RECV( a,  N,  MPI_DOUBLE_PRECISION ,
          0,  i_loop, MPI_COMM_WORLD,  istatus,  ierr )
endif
          a( )を使った計算処理
if (myid .eq.  0) then
     do  i=1,  numprocs - 1
        call  MPI_WAIT(irequest(i),  istatus, ierr )
     enddo
endif
```

ランク0のプロセスは、ランク1~numprocs-1までのプロセスに対して、ノンブロッキング通信を用いて、長さNのDOUBLE PRECISION型配列データを送信

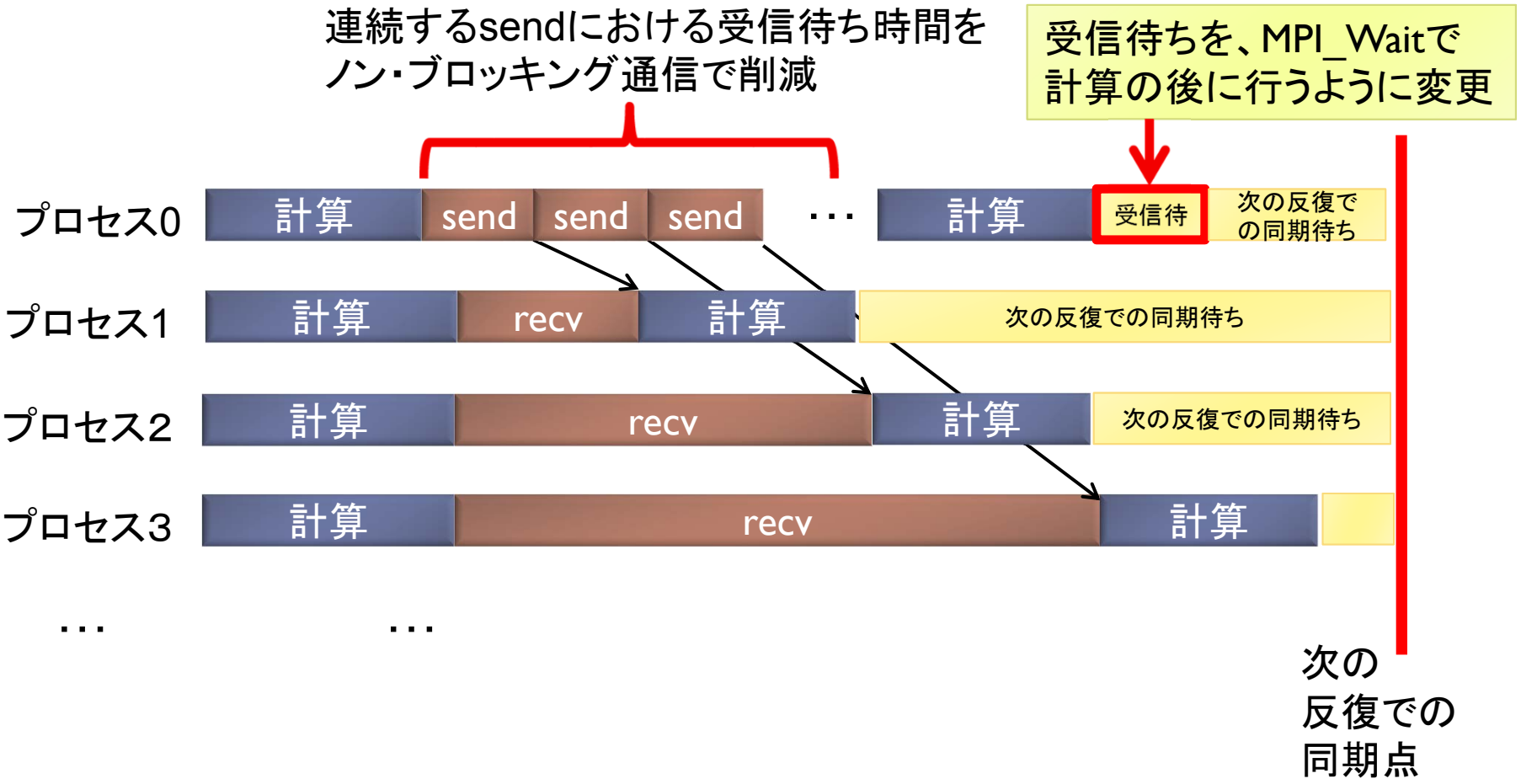
ランク1~numprocs-1までのプロセスは、ランク0からの受信待ち。

プロセス0は、recvを待たず計算を開始

ランク0のプロセスは、ランク1~numprocs-1までのプロセスに対するそれぞれの送信に対し、それぞれが受信完了するまでビジーウェイト（スピンウェイト）する。

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



永続的通信（その１）

- ノン・ブロッキング通信は、MPI_ISENDの実装が、MPI_ISENDを呼ばれた時点で本当に通信を開始する実装になっていないと意味がない。
- ところが、MPIの実装によっては、MPI_WAITが呼ばれるまで、MPI_ISENDの通信を開始しない実装がされていることがある。
  - この場合には、ノン・ブロッキング通信の効果が全くない。
- 永続的通信（Persistent Communication）を利用すると、MPIライブラリの実装に依存し、ノン・ブロッキング通信の効果が期待できる場合がある。
  - 永続的通信は、MPI-１からの仕様（たいていのMPIで使える）
    - しかし、通信と演算がオーバラップできる実装になっているかは別問題

永続的通信（その２）

- 永続的通信の利用法
  1. 通信を利用するループ等に入る前に1度、通信相手先を設定する初期化関数を呼ぶ
  2. その後、SENDをする箇所にMPI_START関数を書く
  3. 真の同期ポイントに使う関数(MPI_WAIT等)は、ISENDと同じものを使う
- MPI_SEND_INIT関数で通信情報を設定しておくと、MPI_START時に通信情報の設定が行われない
  - 同じ通信相手に何度でもデータを送る場合、通常のノン・ブロッキング通信に対し、同等以上の性能が出ると期待
- 適用例
  - 領域分割に基づく陽解法
  - 陰解法のうち反復解法を使っている数値解法

永続的通信の実装例（C言語）

```
MPI_Status istatus;
MPI_Request  irequest;
…
if (myid == 0) {
  for (i=1; i<numprocs; i++) {
      ierr = MPI_Send_init (a,   N,   MPI_DOUBLE_PRECISION,   i,
                    0,   MPI_COMM_WORLD,   irequest );
  }
}
…
if (myid == 0) {
  for (i=1; i<numprocs; i++) {
     ierr = MPI_Start ( irequest );
  }
}

/* 以降は、Isendの例と同じ */
```

メインループに入る前に、送信データの相手先情報を初期化する

ここで、データを送る

永続的通信の実装例（Fortran言語）

```
integer istatus(MPI_STATUS_SIZE)
integer irequest(0:MAX_RANK_SIZE)
…
if (myid .eq. 0) then
  do i=1, numprocs-1
    call  MPI_SEND_INIT  (a,  N,  MPI_DOUBLE_PRECISION,  i,
          0,  MPI_COMM_WORLD,  irequest(i),  ierr)
  enddo
endif
…
if (myid .eq. 0) then
  do i=1, numprocs-1
     call MPI_START (irequest, ierr )
  enddo
endif

/* 以降は、ISENDの例と同じ */
```

メインループに入る前に、送信データの相手先情報を初期化する

ここで、データを送る

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
  - 付属のサンプルプログラム全てが利用可能

レポート課題（その２）

1. [L5] ブロッキングは同期でないことを説明せよ。
2. [L10] MPIにおけるブロッキング、ノンブロッキング、および通信モードによる分類に対応する関数を調べ、一覧表にまとめよ。
3. [L15] 利用できる並列計算機環境で、ノンブロッキング送信（MPI_Isend関数）がブロッキング送信（MPI_Send関数）に対して有効となるメッセージの範囲（N=0～適当な上限）について調べ、結果を考察せよ。
4. [L20] MPI_Allreduce関数 の＜限定機能＞版を、ブロッキング送信、およびノンブロッキング送信を用いて実装せよ。さらに、その性能を比べてみよ。なお、＜限定機能＞は独自に設定してよい。

レポート課題（その３）

5. [L15] MPI_Reduce関数を実現するRecursive Halving
アルゴリズムについて、その性能を調査せよ。この際、
従来手法も調べて、その手法との比較も行うこと。
6. [L35] Recursive Halvingアルゴリズムを、ブロッキング送信／
受信、および、ノンブロッキング送信／受信を用いて実装せよ。また、それらの性能を評価せよ。
7. [L15] 身近の並列計算機環境で、永続的通信関数の性能を
調べよ。
8. [L10～] 自分が持っているプログラムに対し、ループ分割、
ループ融合、その他のチューニングを試みよ。
9. [L10～] 自分が持っているMPIプログラムに対し、ノンブロッキング通信（MPI_Isend, MPI_Irecv）を実装し、性能を評価せよ。
また永続的通信が使えるプログラムの場合は実装して評価せよ。