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

第５回
プログラム高速化の応用

東京大学情報基盤センター　片桐孝洋

東京大学情報基盤センター
Information Technology Center,The University of Tokyo

講義日程と内容について

- 2015年度　CMSI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、2015年4月9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、2015年4月16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、2015年4月23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)、2015年5月7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、2015年5月14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

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）

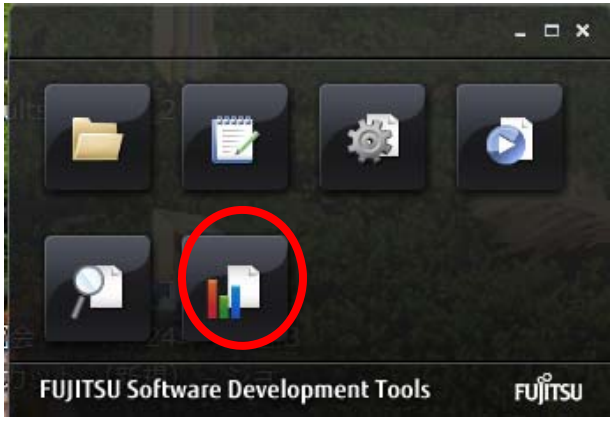
- プログラミング支援ツール(FUJITSU Software Development Tools Version 1.2.1 for Windows) をインストール
  - 以下をアクセス

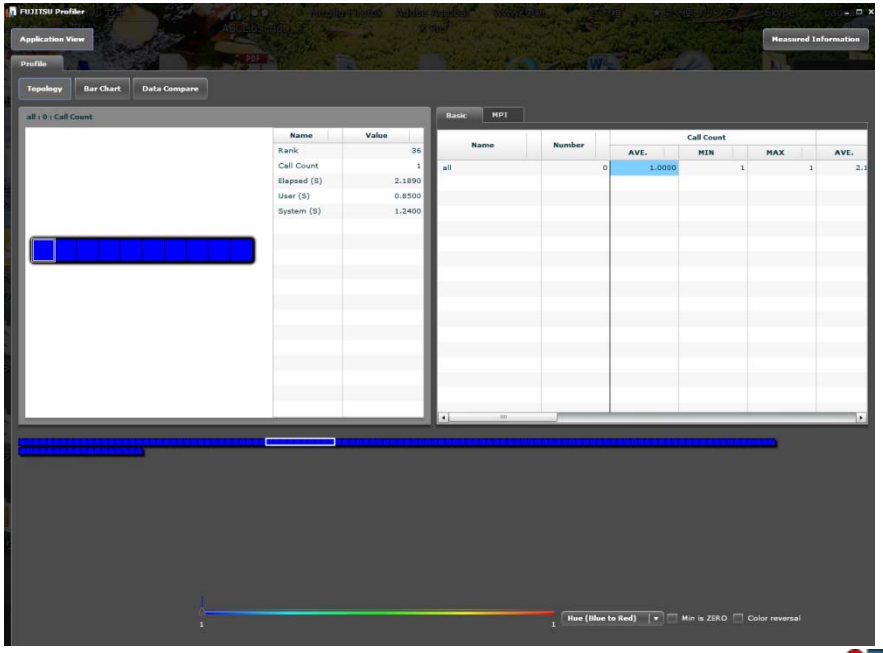
https://oakleaf-fx-1.cc.u-tokyo.ac.jp/fsdtfx10tx/install/index.html

- 「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



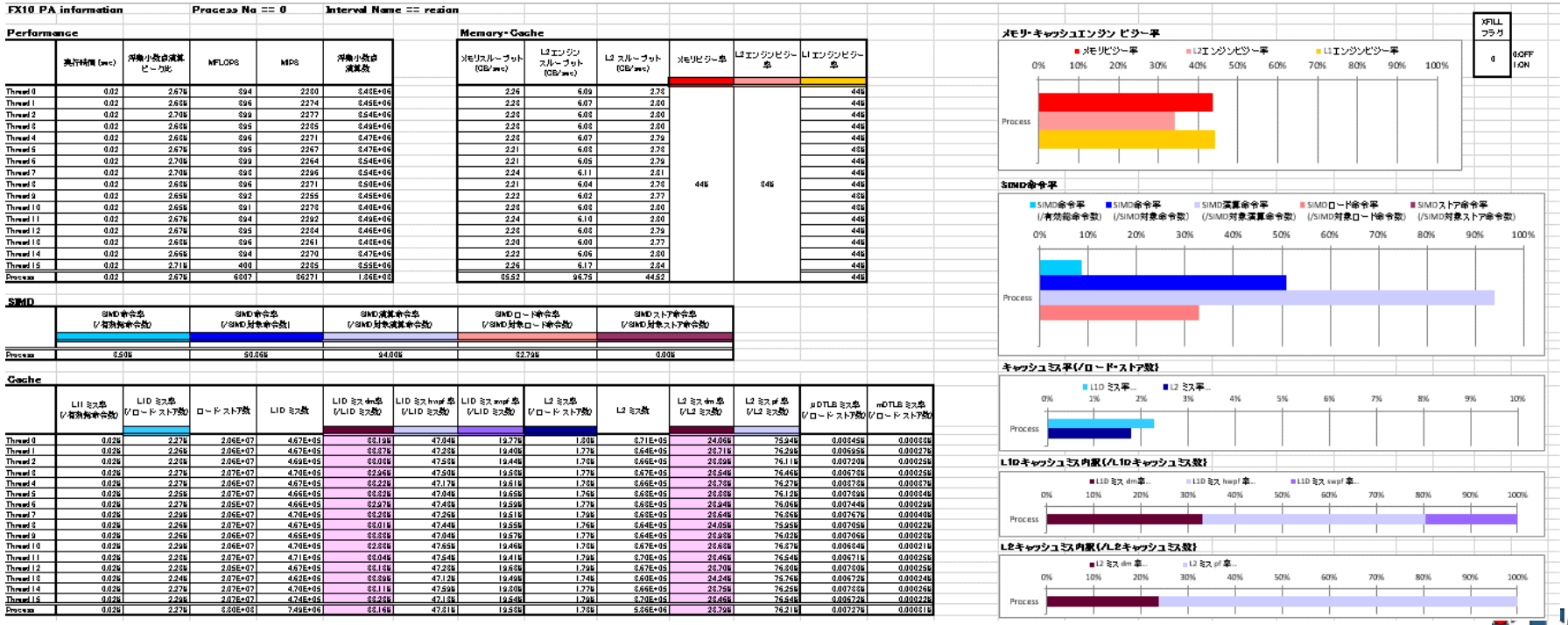


詳細プロファイラで取れるデータ（東大FX10）

- 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）

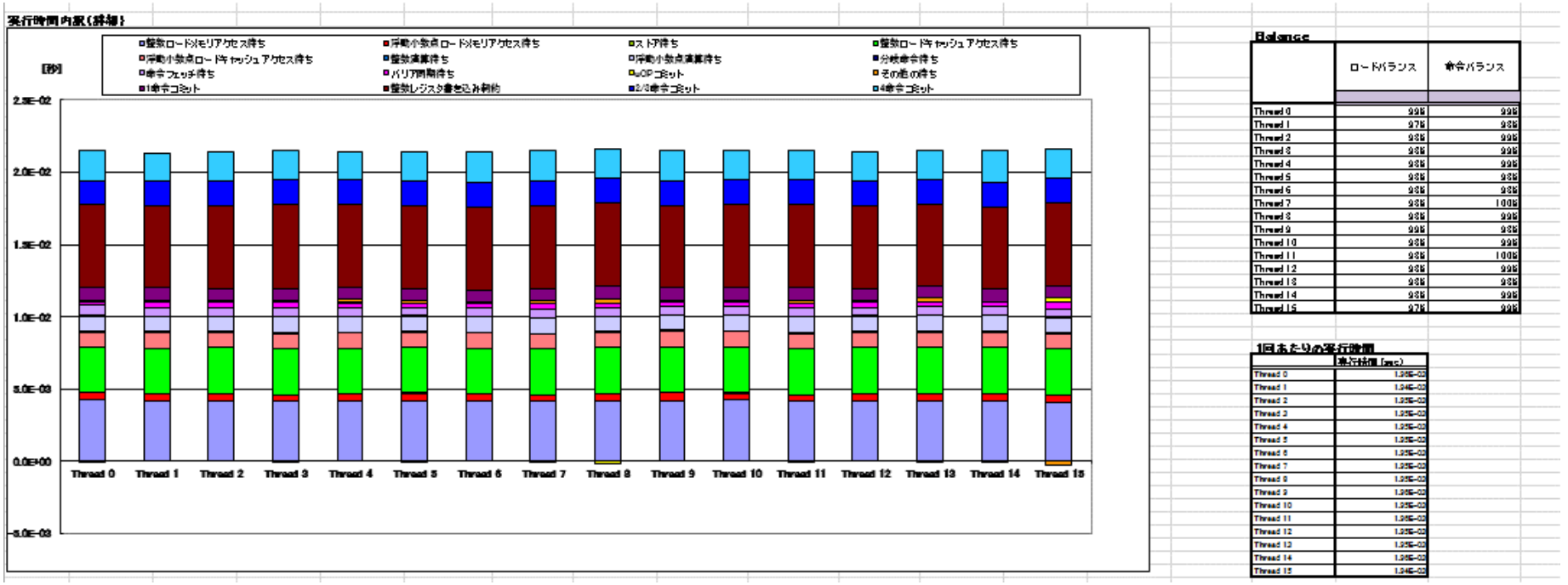
- 富士通FX10には、性能プロファイラによる結果をExcel形式で可視化できるツールがある
  - 演算ピーク比率、MFLOPS、MIPS、メモリスループット、L1キャッシュミス率、TLBミスヒット率、などのデータなどが、まとめて出力される
  - メモリビジー率、などが可視化されて出力される



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)

- 1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

- 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。

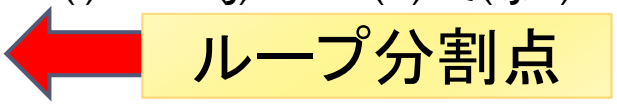
ここでのコード最適化の方針（その２）

- 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;
```

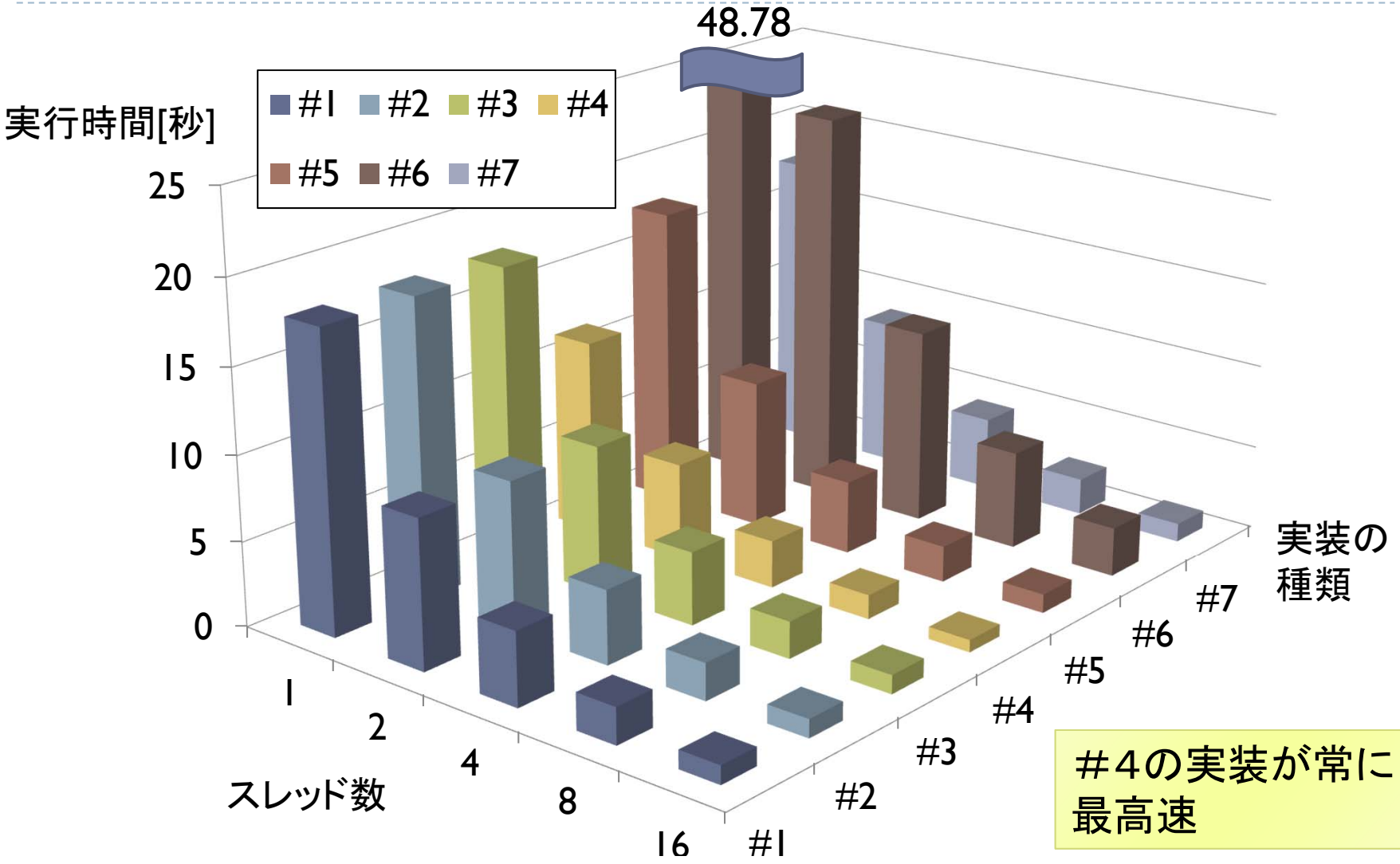
チューニングの可能性のあるコード例（経験的に決めた数例について）

- #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

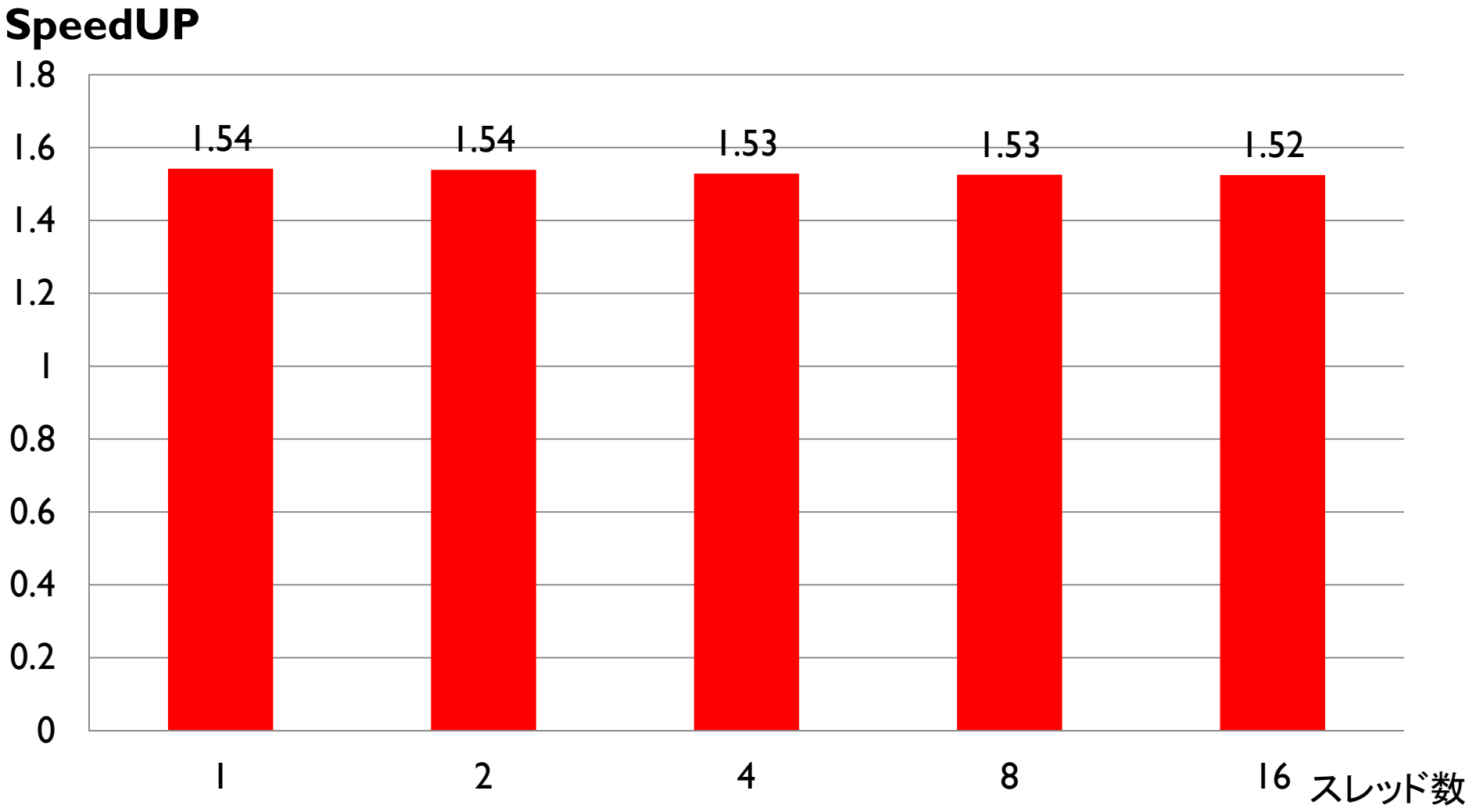
- 東京大学情報基盤センターFX10を利用
  - 1ノード、16スレッド
  - Sparc64 IV-fx (1.848 GHz)
- 最外ループに対して、OpenMPが適用可能
  - parallel do構文で並列化可能
- スレッド数は、1～16まで変更可能

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)

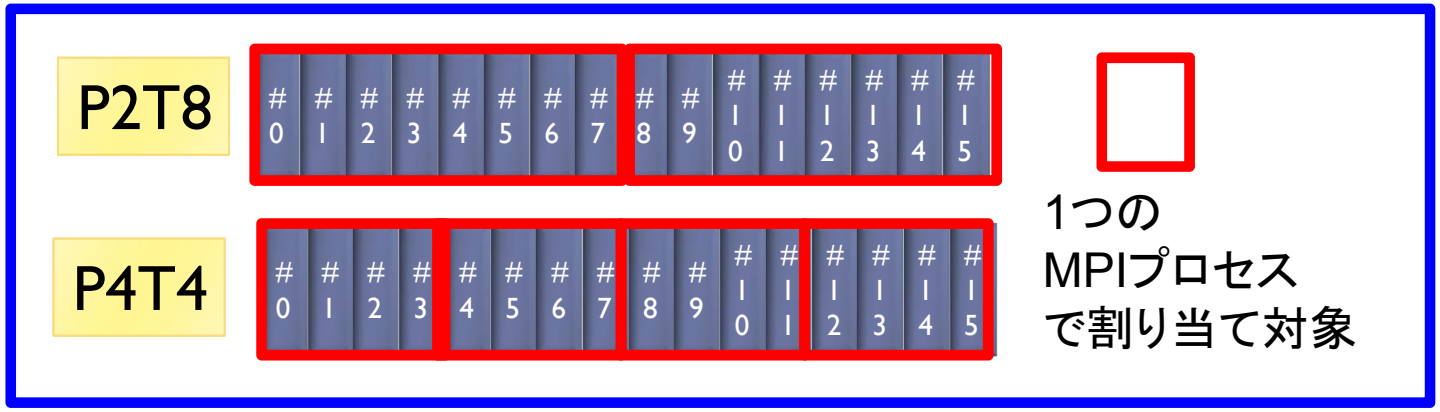
- Intel Xeon Phi
  - Xeon Phi 5110P (1.053 GHz), 60 cores
  - メモリ量：8 GB (GDDR5)
  - 理論ピーク性能：1.01 TFLOPS
  - Xeon Phiのクラスタ（ノード当たり1ボード）
  - InfiniBand FDR x 2 Ports
    - Mellanox Connect-IB
    - PCI-E Gen3 x16
    - 56Gbps x 2
    - 理論バンド幅 13.6GB/s
    - フルバイセクション
  - Intel MPI
    - MPICH2、MVAPICH2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

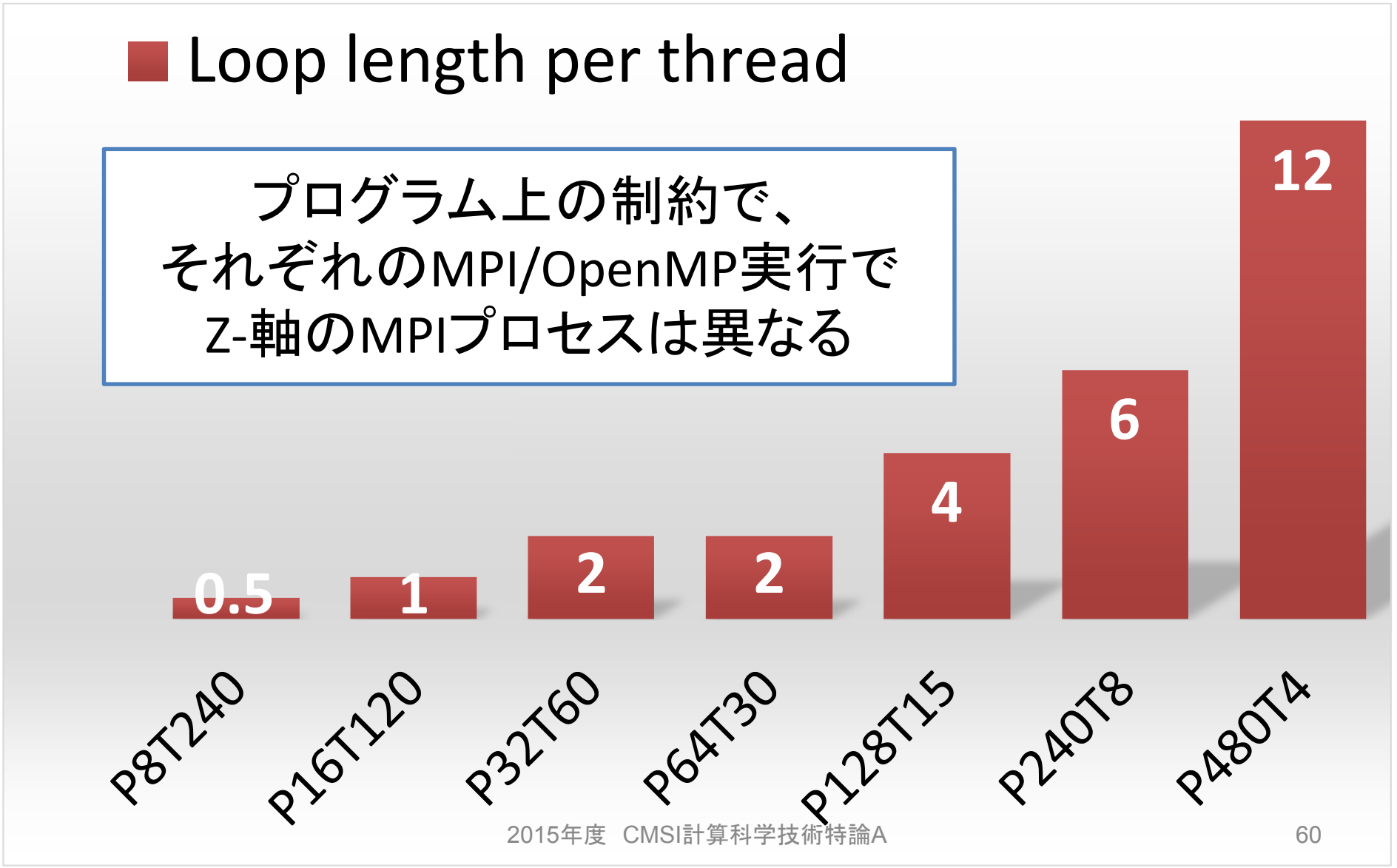
- ppOpen-APPL/FDM ver.0.2
- ppOpen-AT ver.0.2
- 時間ステップ数: 2000 steps
- ノード数: 8 ノード
- Native Mode 実行
- 問題サイズ
  (8 GB/ノードでの最大サイズ)
  - NX * NY * NZ = 1536 x 768 x 240 / 8ノード
  - NX * NY * NZ = 768 * 384 * 120 / ノード
    (MPIプロセス当たりのサイズではない)

ハイブリッドMPI/OpenMP実行の詳細

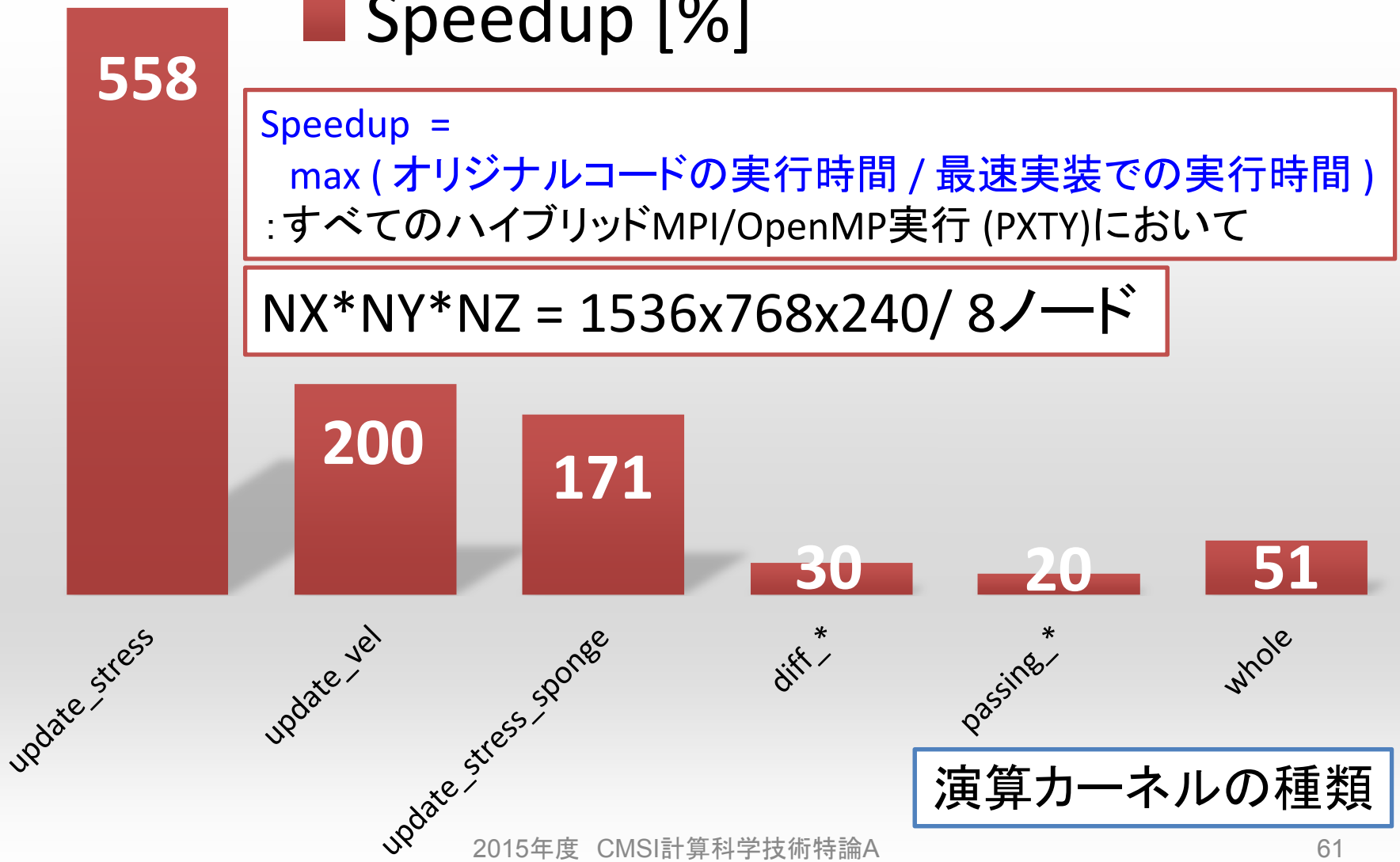
- Xeon PhiにおけるMPIプロセス数とOpenMPスレッド数
  - 4 HT (Hyper Threading)
  - PX TY: X MPIプロセス、Y スレッド/プロセス
    - P8T240 : 最少のハイブリッドMPI/OpenMP実行（ppOpen-APPL/FDMでは 8MPIプロセスが最低でも必要なため）
    - P16T120
    - P32T60
    - P64T30
    - P128T15
    - P240T8
    - P480T4
    - P960T2以下は、この環境ではMPIエラーで実行できなかったため除外



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

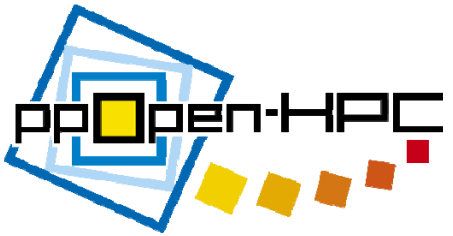


ループ融合等による最大の速度向上(Xeon Phi、8ノード)



最速のコード (update_stress)

●Xeon Phi (P240T8)

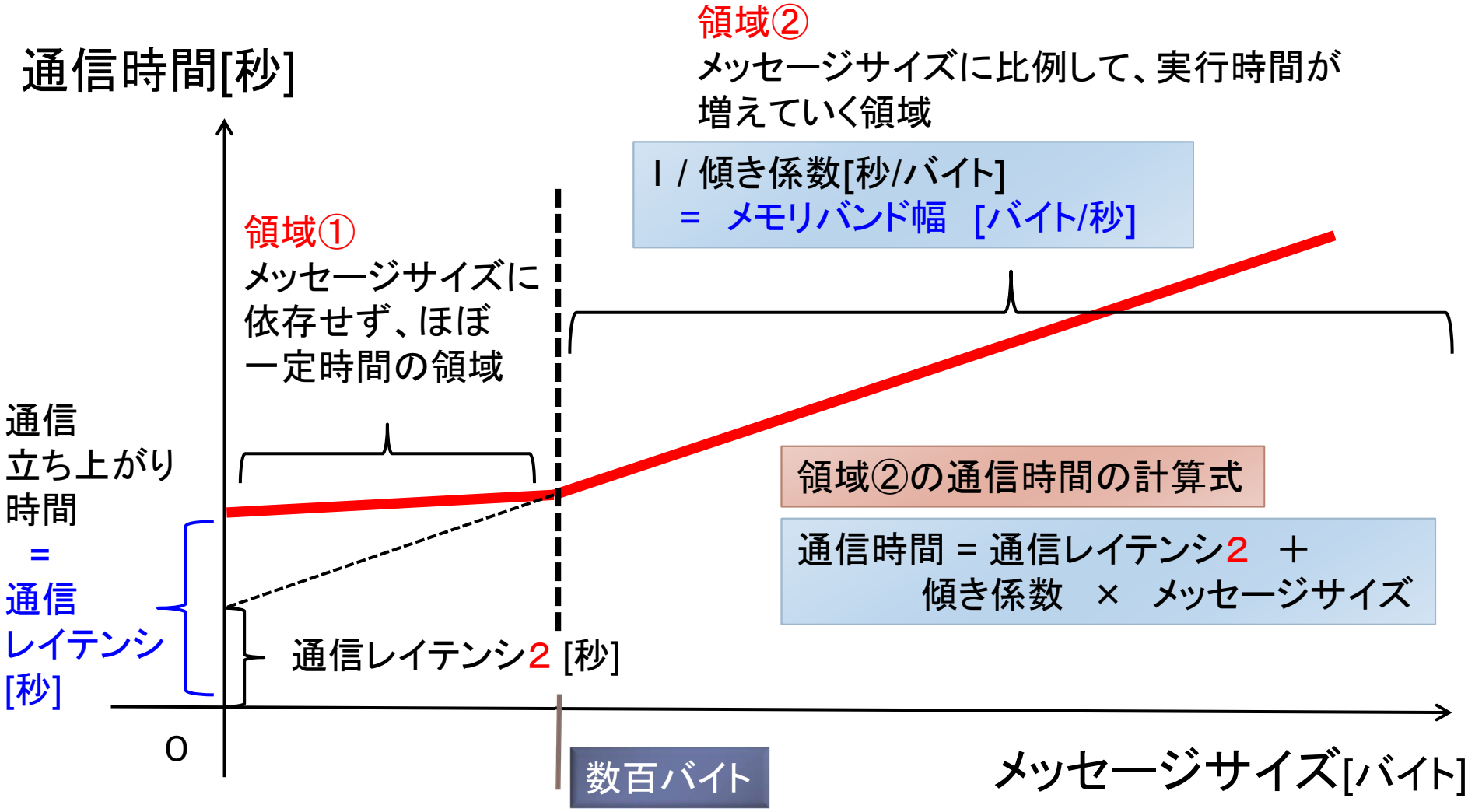



```
!$omp parallel do private (k,j,i,RL1,RM1,RM2,RLRM2,DXVX1,DYVY1,DZVZ1,D3V3,DXVYDYVX1,DXVZDZVX1,DYVZDZV1)
 DO k_j = 1 , (NZ01-NZ00+1)*(NY01-NY00+1)
   k = (k_j-1)/(NY01-NY00+1) + NZ00
   j = mod((k_j-1),(NY01-NY00+1)) + NY00
   DO i = NX00, NX01
     RL1 = LAM (I,J,K);  RM1 = RIG (I,J,K)
     RM2 =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

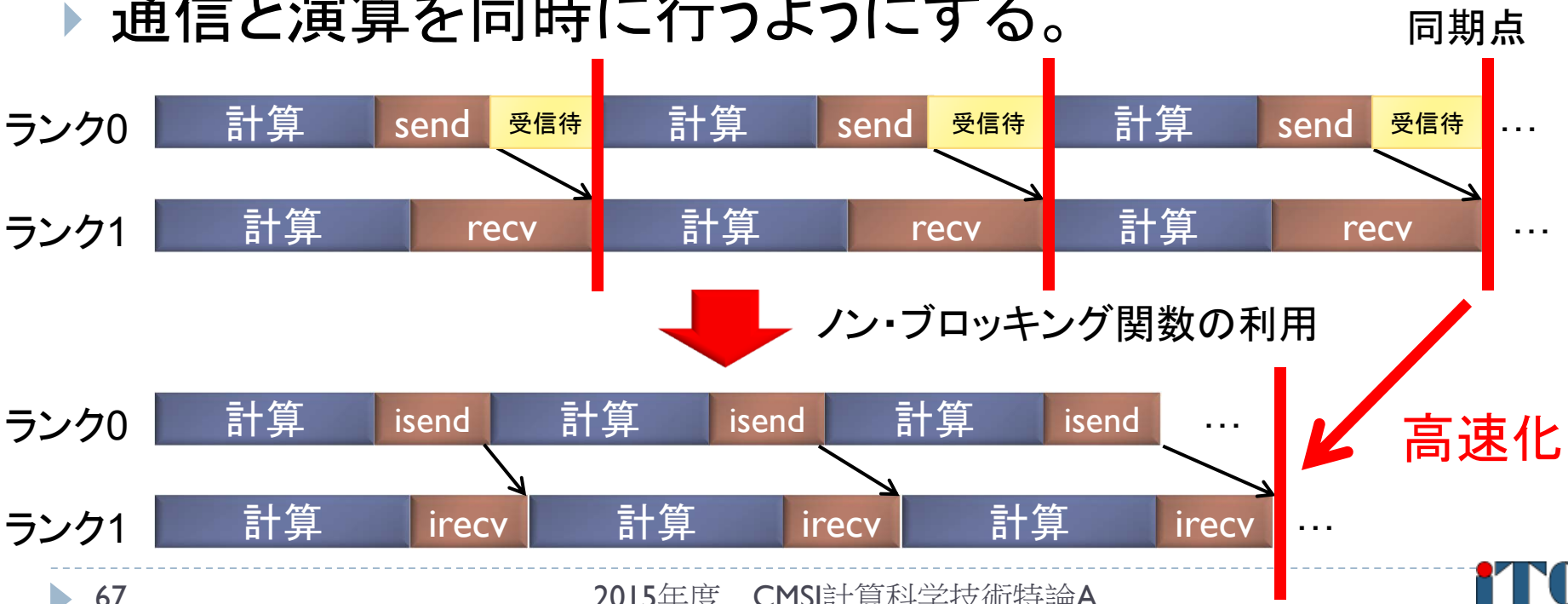
- 自分のアプリケーションの通信パターンについて、以下の観点を知らないと通信の最適化ができない
  - ＜領域①＞＜領域②＞のどちらになるのか
  - 通信の頻度（回数）はどれほどか
- **領域①の場合**
  - 「通信レイテンシ」が実行時間のほとんど
  - 通信回数を削減する
    - 細切れに送っているデータをまとめて1回にする、など
- **領域②の場合**
  - 「メッセージ転送時間」が実行時間のほとんど
  - メッセージサイズを削減する
  - 冗長計算をして計算量を増やしてでもメッセージサイズを削減する、など

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。

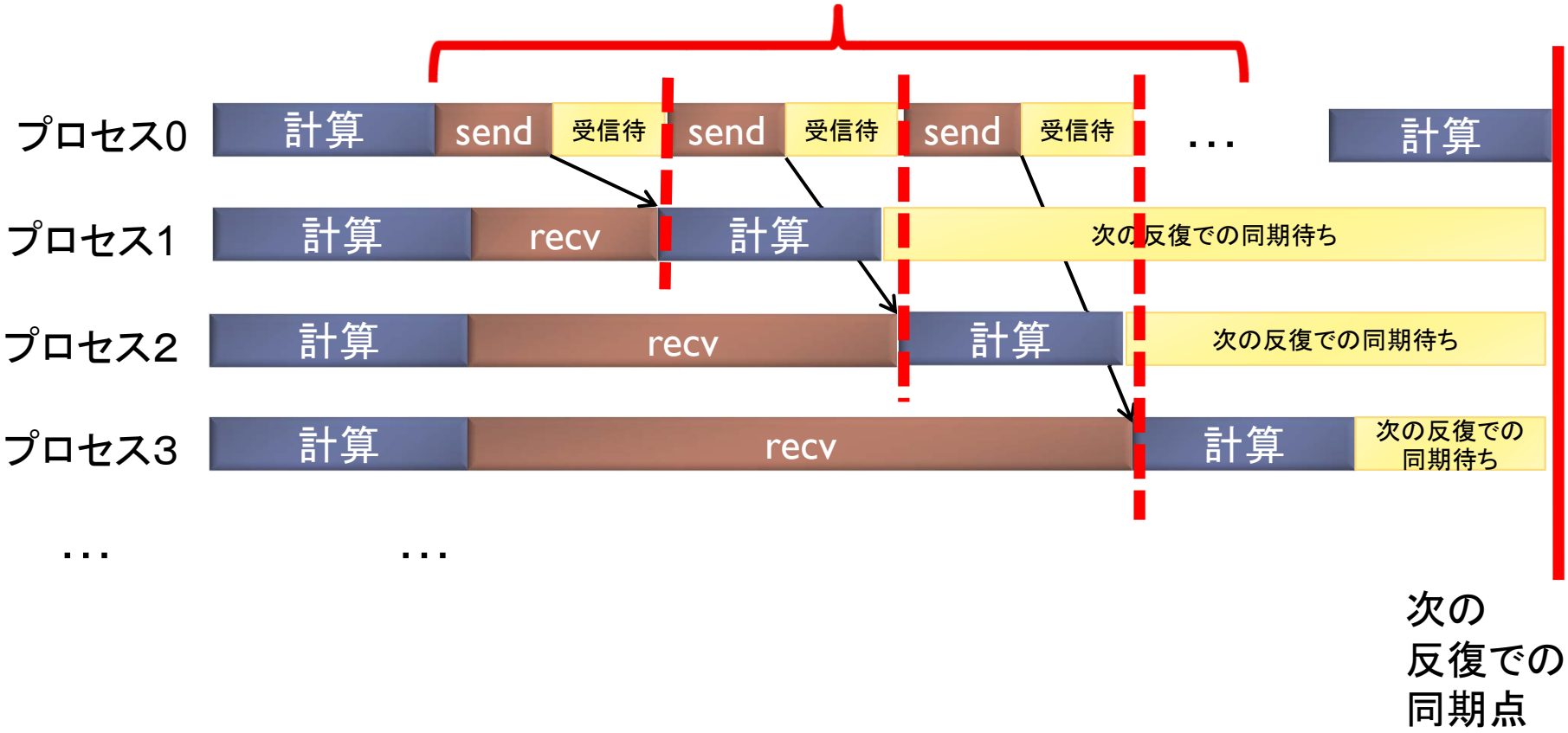


非同期通信：Isend、Irecv、永続的通信関数

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

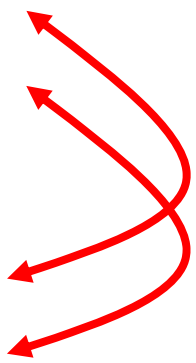
- MPI_Isend関数
  - ノンブロッキング
  - 標準通信モード（ノンローカル）
    - 通信バッファ領域の状態にかかわらず戻る
    - バッファ領域がとれる場合は、メッセージがバッファリングされ、対応する受信が起動する前に、送信処理が完了できる
    - バッファ領域がとれない場合は、対応する受信が発行され、メッセージが受信側に完全にコピーされるまで、送信処理が完了できない
    - MPI_Wait関数が呼ばれた場合の振舞いと理解すべき。

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

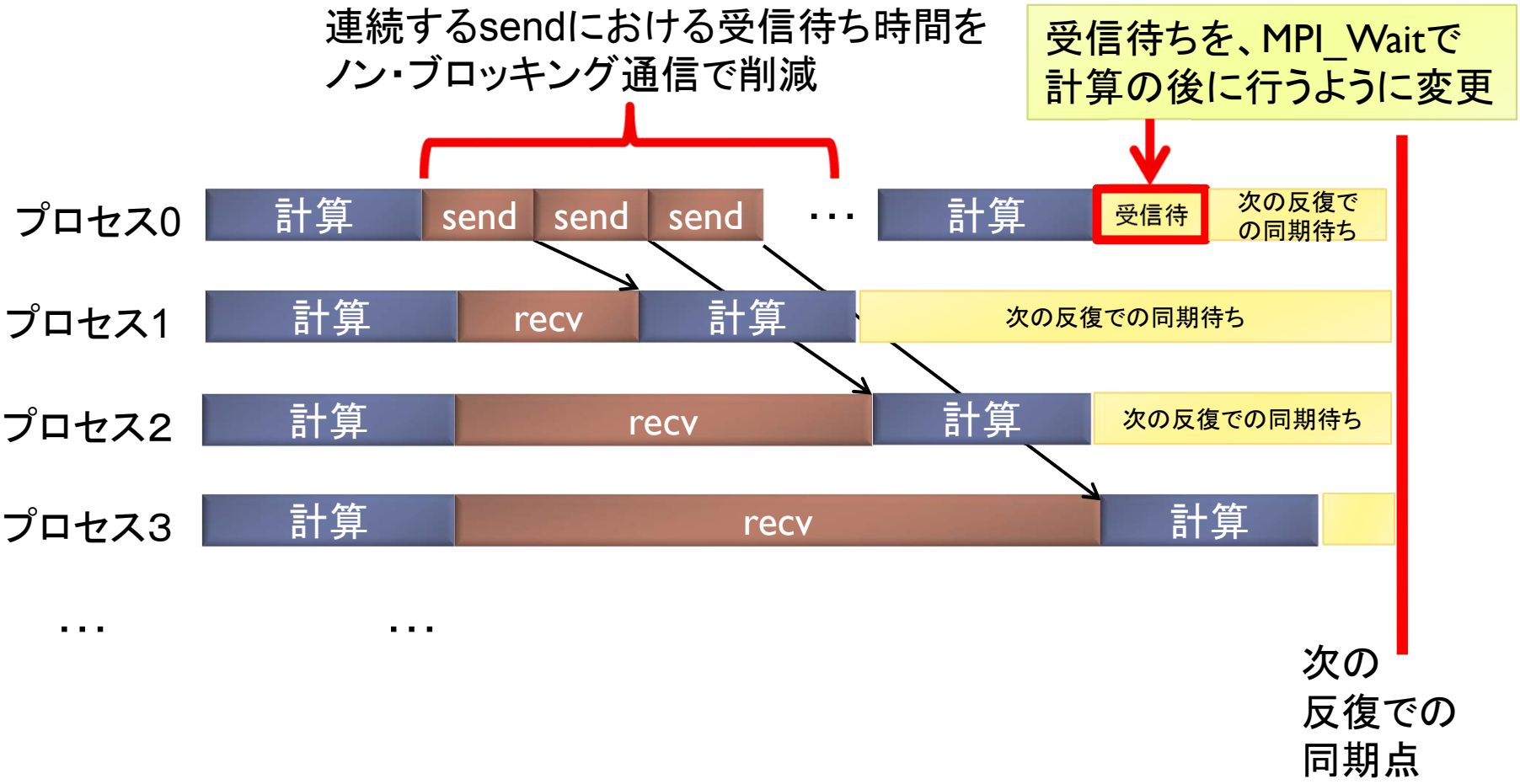
ランク1~numprocs-1までのプロセスは、ランク0からの受信待ち。

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。