「情報システム学特別講義３」
プログラム高速化の応用

東京大学情報基盤センター准教授　片桐孝洋

2014年6月3日（火）14:40－16:10

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

講義日程
（情報システム学特別講義３）

レポートおよびコンテスト課題
（締切：
2014年8月11日（月）24時　厳守

1. ~~４月８日：ガイダンス~~
2. ~~４月15日~~
   - ~~プログラム高速化の基礎（その１）~~
3. ~~４月22日~~
   - ~~プログラム高速化の基礎（その２）~~
4. ~~５月13日~~
   - MPI~~の基礎~~
5. ~~５月20日~~
   - OpenMP~~の基礎~~
6. ~~５月27日~~
   - Hybrid~~並列化技法~~ ~~（~~MPIとOpenMP~~の応用編）~~
7. ６月３日
   - プログラム高速化の応用
8. ６月10日
   - 行列ーベクトル積の並列化
9. ６月17日
   - べき乗法の並列化
10. ６月24日
    - 行列−行列積の並列化
11. ７月８日
    - LU分解の並列化
12. ７月15日
    - 非同期通信
    - 疎行列反復解法の並列化
13. ７月22日
    - ソフトウェア自動チューニング
14. ８月５日（補講日）
    - エクサフロップスコンピューティングに向けて

性能チューニングの応用

コンパイラの最適化とは（1/2）

- コンパイラが自動で行う高速化
- 主に、レジスタへのデータ割り当て、効率の良い命令発行順、および、コードのループ変換を行うことで、実行時間を高速化する
  - ループアンローリング
  - キャッシュブロック化（タイリング）
  - ソフトウェアパイプライニングの強化
  - 積極的にSIMD化されたマシン語を生成する
    ・・・・など
- データ構造の変換は、一般的に行えない
  - 配列がとびとびアクセスになるので、配列の中身を転置するようにデータを置き換える、など
- 演算精度が変わる最適化はユーザの指示がない限り行わない
  - 演算精度を変えてよいかはユーザが指示

コンパイラの最適化とは (2/2)

- コンパイラ最適化の指示の方法
  1. コード全体への指示：
     コンパイラのオプションで指定
     - 例） gcc −O3 foo.c
     - 例） frt −Kfast, simd=2 foo.f90
  2. 個別ループへの指示：
     プログラム中に指示を記載する（ディレクティブ）
     - 例） !OCL UNROLL(8)

```
do i=1, n
  a(i) = a(i) + b(i)
enddo
```


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

コンパイラ最適化が効きやすいコードとは？(1/2)

- いろいろな場合があり一般化できないが・・・
  以下は一例
- ループの開始値、終了値が明確であること
  - for (i=0; i<100; i++) { … }　ならよいが
  - for (i=istart; i<iend; i++) { … }　だと困る
  - 理由は、ループの長さから、(たとえば)適切なソフトウェア／ハードウェアのパイプライニングの実装方法を選べるため

コンパイラ最適化が効きやすいコードとは？(2/2)

- ループ中に書かれている式が多くないこと
  - 式が多いと、組合せ爆発を起こし、データのレジスタへの配置の最適化が困難になっていく
  - ただし、少なくするため、ループ分割しすぎると、かえって速度が低下する
    - レジスタ利用効率が下がるため
  - アーキテクチャを考慮し、ループ中に塩梅の良い多さの式にすること自体が職人芸
- マシン語の対応と1対1になっていること
  - 例）明らかにSIMD命令が発行できるように書く方がいい
    - x = a*a + b*b + c*c　　よりかは
    - x = (a*a + (b*b + (c*c + d)))　　が良い

自動並列化コンパイラとは（1/3）

- ループ中に記載される式に対し、データ依存関係を解析し、自動的にスレッド並列化を行うコンパイラ
- コンパイラオプションに自動並列化の指定をすることで行う
  - 例）ifort −O3 −parallel　foo.f90

自動並列化コンパイラとは（3/3）

- 自動並列化においても、ループの開始値、終了値が明確であるとよい
  - for (i=0; i<100; i++) { … }
  - 以上においては、コンパイラがスレッド並列化する際、並列化して効果あるかどうか、ループの長さから判断できる。
- 実行時にならないとループ長が決まらない場合
  - for (i=istart; i<iend; i++) { … }
  - コンパイラは、ループ長が短いと予想し並列化しないかもしれないし、逆に長くなると予想し並列化するかもしれない　←コンパイラオプションでユーザが指定

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）

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）
  - 自動並列化は＜特に＞過信しない
    - 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。
    - 例）foo (A, B, C);

      ←一般にA、B、Cは同一配列で引き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

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

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）
  - 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５）

- 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、いまも活発な研究テーマ
    - 「ソフトウェア自動チューニング」の研究分野
    - コストを低く保つ、チューニングの枠組み
      - コードの自動生成技術
      - 自動性能検証技術

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**
  - 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７）

- **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. データを間引き、I/O量を削減する
  2. 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
       - 大規模なデータサイズを1回I/Oをする場合は、ストライプサイズを大きくする
  3. 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. 何らかの性能プロファイルのための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）

既存の性能プロファイラ (1/2)

- フリーコードのもの
  - gprof
    - GNU コンパイラ(gcc)で動作する性能プロファイラ
    - コールグラフを示し、各関数のコストがわかる
  - Jumpshot
    - MPIの実装MPICHと連携した、MPI実行可視化ツール
  - PAPI (Performance Application Programming Interface)、(米国テネシー大)
    - FLOPS値、キャッシュやTLBのミスヒット率など、対応ハードウェアがあるものについては、詳細なデータが取得可能
  - TAU (Tuning and Analysis Utilities) 、(米国オレゴン大)
    - 可視化ツールが充実
  - Scalasca 、(独国Jülich Supercomputing Centre (JSC))
    - 京コンピュータで提供されている

既存の性能プロファイラ（2/2）

- 商用のもの
  - Vampir
    - MPIプログラムのプロファイルでは著名
  - Intel Vtune
    - Intelコンパイラ、Intel統合環境と連携
- スパコン用
  - 各社、通常、スパコンごとにある。
  - 日立　pmpr
  - 富士通　基本プロファイラ、詳細プロファイラ
  - Cray社 PATツール
  - などなど

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

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

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
...
```

各関数の実行間と、全体に占める時間の割合

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 -Ihwm -Hevent=Cache mpirun ./wa2
- 実行する
  $ pjsub wa2-pure.bash

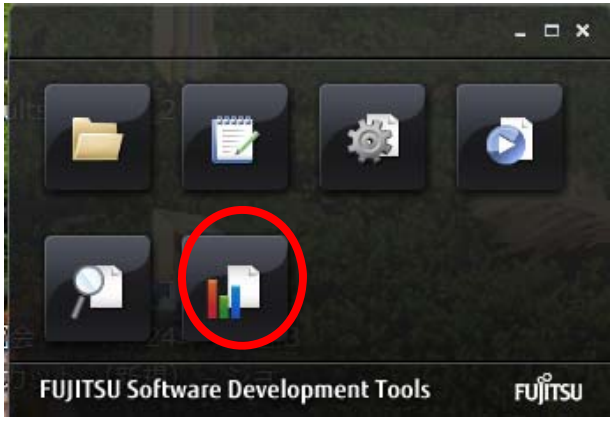
詳細プロファイラGUIによる表示例（東大FX10）

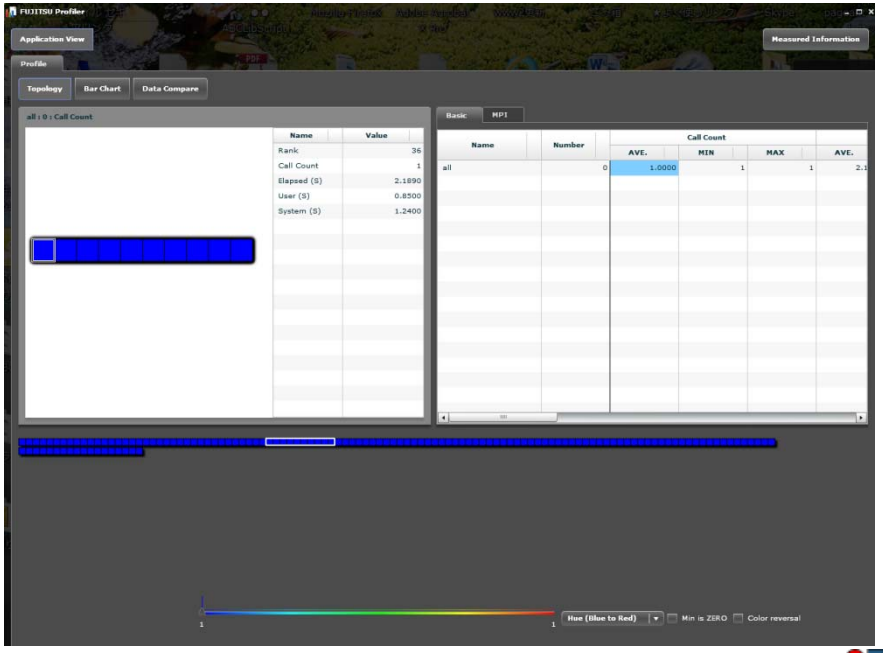
- プログラミング支援ツール(FUJITSU Software Development Tools Version 1.2.1 for Windows) をインストール
  - 以下をアクセス

  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

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）

- Seismic_3D：
  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
```


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

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。

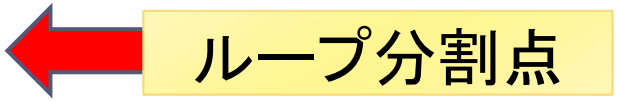
ここでのコード最適化の方針（その２）

- 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;
```

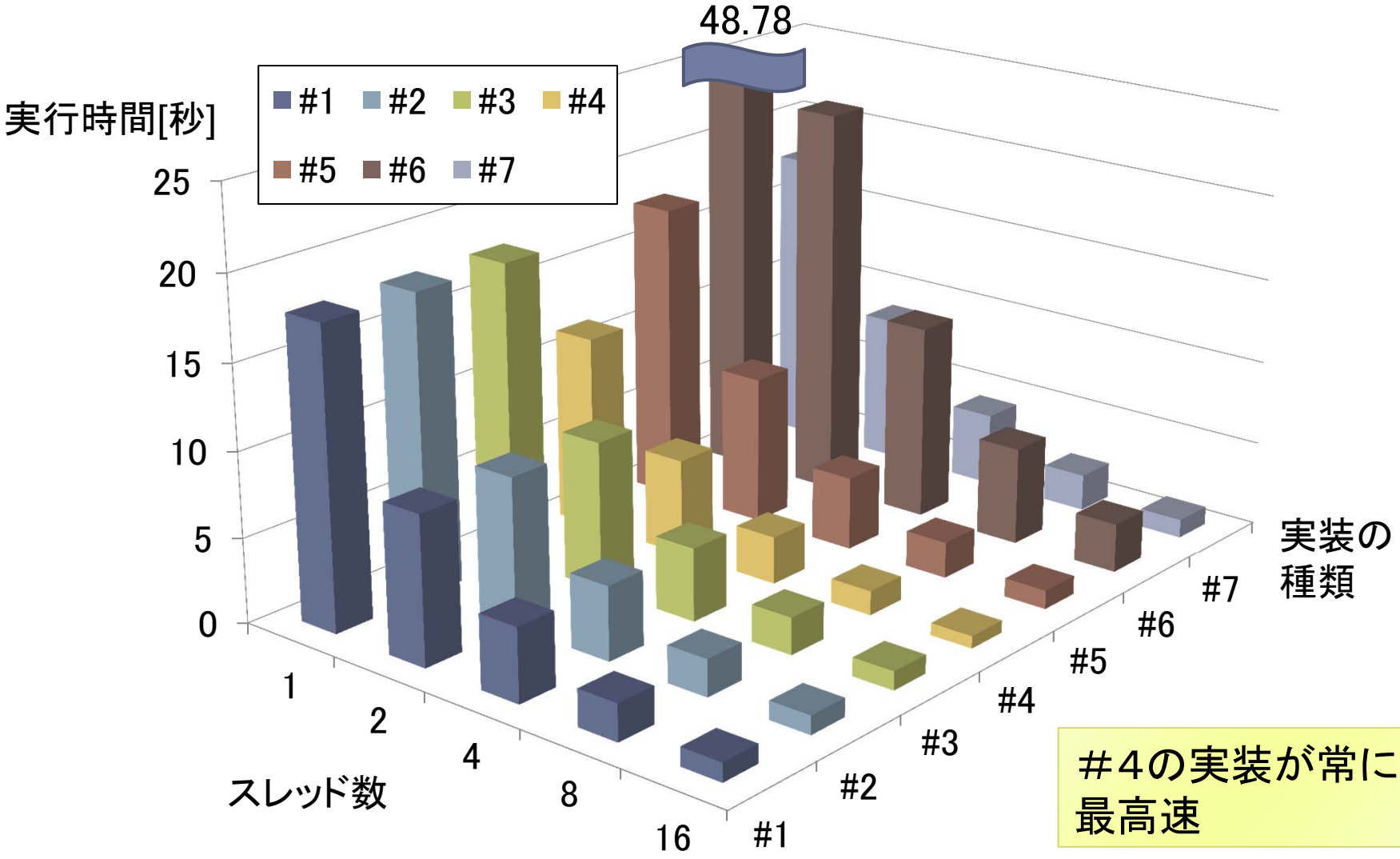
チューニングの可能性のあるコード例（経験的に決めた数例について）

- #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

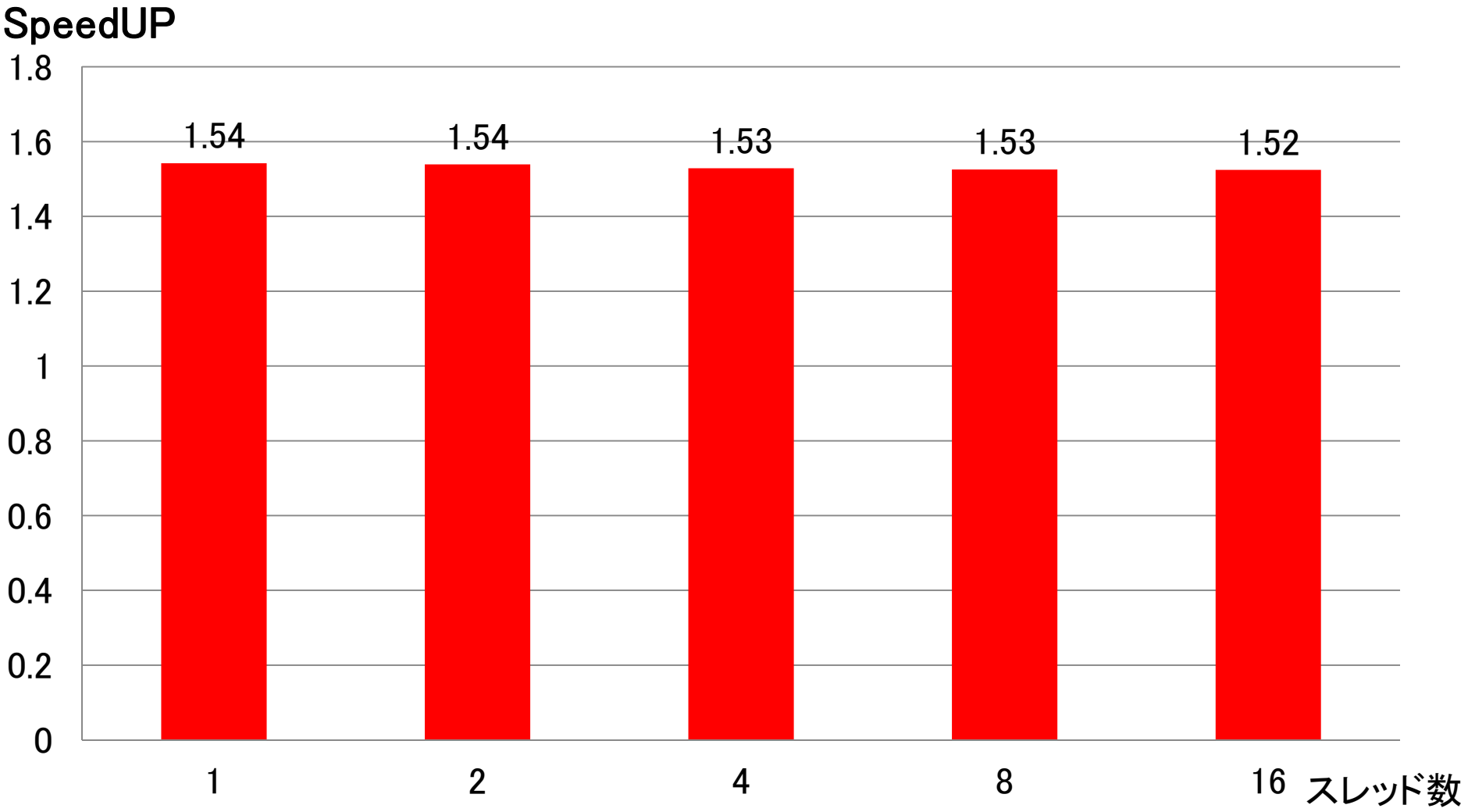
- 東京大学情報基盤センターFX10を利用
  - 1ノード、16スレッド
  - Sparc64 IV-fx (1.848 GHz)
- 最外ループに対して、OpenMPが適用可能
  - parallel do構文で並列化可能
- スレッド数は、1～16まで変更可能

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以上必要
  - 3次元問題で上記のサイズ確保できるか？

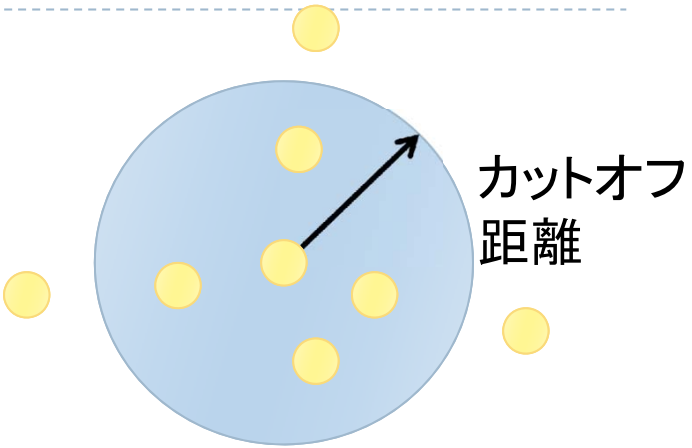
確保できない場合はループ融合が必須

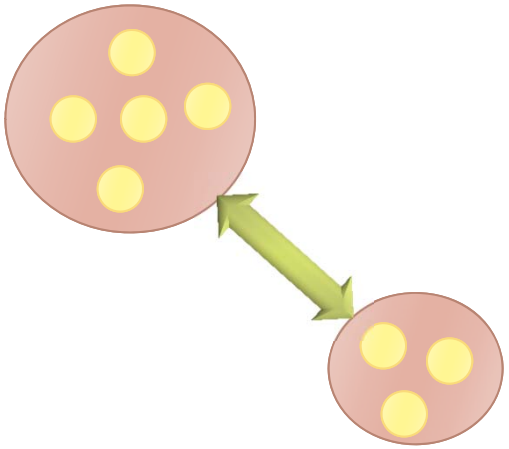
余談　～問題の違いを認識しよう

- 結局のところ、高速化の基本は、扱う問題の特徴の違いをよく理解することにある
- たとえば、「粒子法」という方法があり、以下の2つがキーワードとして良く出てくる
  - MD (Molecular Dynamics): 分子動力学法
  - DEM (Discrete Element Method) : 離散要素法
- 双方とも、隣接する粒子(分子)との相互作用が並列性を阻害するが、扱う問題が異なるため、問題の性質が(まったく)異なる

MDとDEMの並列化の観点の違い

- MD
  - カットオフ距離がある
  - カットオフ距離内に分子（粒子）が集中して、計算が偏ることが多い
    - →負荷バランスが悪くなり、並列処理で問題になる
  - 分子（粒子）間に働く力がある
    - 分子（粒子）をあつめてモデル化することで、相互間の依存を無くして、並列性を高めることができる
    - Fast Multipole Method (FMM)



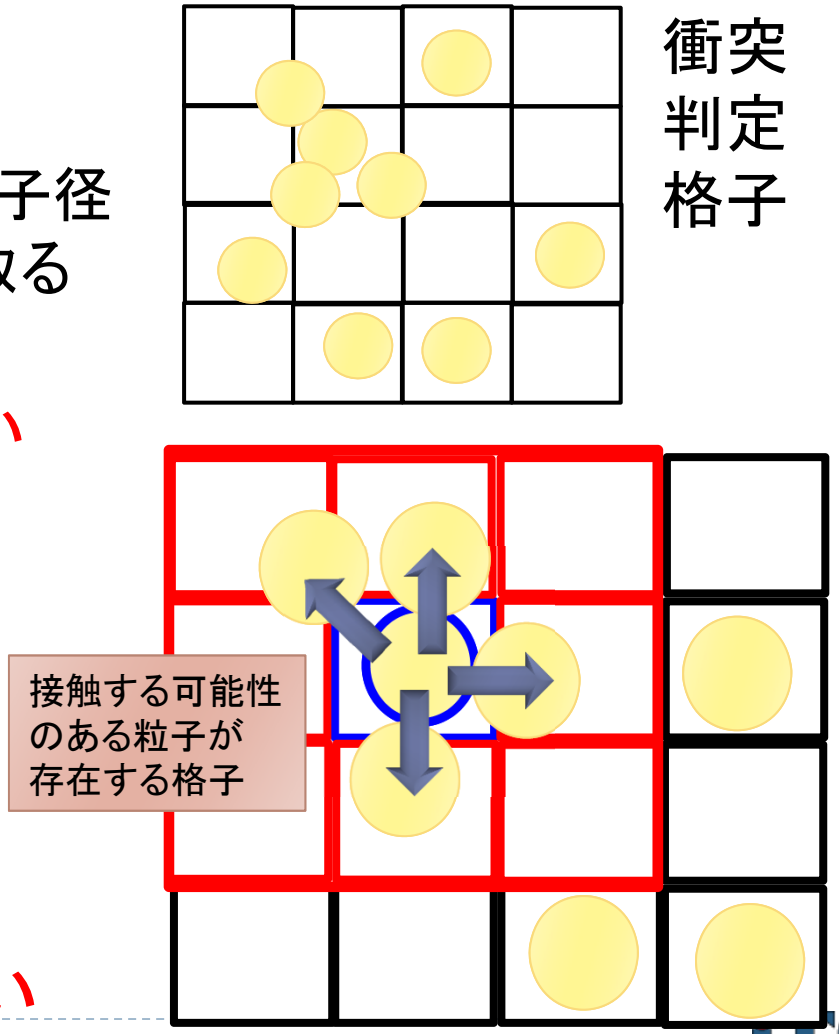


比較的並列化がしやすい

MDとDEMの並列化の観点の違い

- DEM
  - 衝突判定格子がある
  - 衝突判定格子内の粒子数は、粒子径と同じサイズに衝突判定格子を取ることで、たかだか数個にできる
    - →負荷バランスは悪くならない
  - 分子（粒子）間に働く力がないか、無視できるほど小さい
  - 全ての粒子が、全ての粒子と接触する可能性がある
    - 衝突する可能性がある粒子には強い依存性があり並列化できない

➡ MDより並列化がしにくい



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-Mat-noopt-fx.tar

レポート課題（その２）

1. [L5] 身近なコンパイラの最適化オプションについて調査せよ。
2. [L5] 身近なコンパイラの自動並列化機能について調査せよ。
3. [L10] 身近な環境での性能プロファイラの機能を調査せよ。行列-行列積の性能プロファイルと取得せよ。
4. [L10] 性能プロファイラの1つである「PAPI」をインストールし、行列-行列積の性能プロファイルを取得せよ。
5. [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。