

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

第5回 プログラム高速化の応用

名古屋大学情報基盤センター 片桐孝洋



1

2019年度 計算科学技術特論A



講義日程と内容について

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

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！
 - ▶ 1ループ中で書いてある<式>がとても多い場合
 - ▶ 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。

自動並列化コンパイラ

！ = 逐次コードを自動で並列コードに書き直すコンパイラ



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間に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(16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) CX400の富士通コンパイラ と CX400のインテルコンパイラ
- ▶ 同一ベンダのコンパイラでも新規ハードでは、同一コードでも同じ品質で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分野でホットな研究テーマ
 - ▶ 「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）

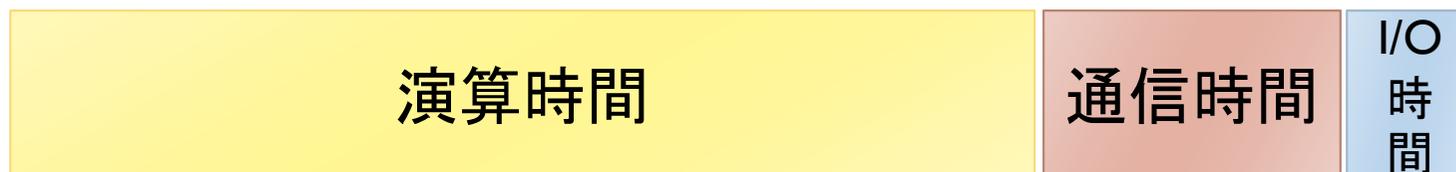
- ▶ **自分のコードのホットスポット(重い部分)を認識せよ**
- ▶ 自分のコードのうち、どの部分が重いのか、実測により確認せよ
 1. **演算時間ボトルネック**(演算時間が多い)
 2. **通信時間ボトルネック**(通信時間が多い)
 3. **I/Oボトルネック**(I/O時間が多い)

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・パイプライン強化オプション、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
 - ▶ 高速ファイルシステム(SSD、バーストバッファなど)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) 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ー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（名大FX100）

- ▶ 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指定部分において、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: `fippix`、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

(名大FX100)

- ▶ 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`(スクリプトファイル) 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

```
$ fipp -A -d Prof
```

基本プロファイラ出力例 (名大FX100) (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 (名大FX100) (2/2)

Procedures profile

```
*****
Application - procedures
*****
```

Cost	%	Mpi	%	Start	End
475	100.0000	312	65.6842	--	-- Application
312	65.6842	312	100.0000	I	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

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

Process 11 - procedures

```
*****
```

Cost	%	Mpi	%	Start	End
5	100.0000	4	80.0000	--	-- Process 11
4	80.0000	4	100.0000	I	45 MAIN__
1	20.0000	0	0.0000	--	-- __GI__sched_yield

```
*****
```

詳細プロファイラ利用例 (名大FX100)

- ▶ 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 (名大FX100)

- ▶ /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 l -lhwm -Hevent=Cache mpirun ./wa2
```

- ▶ 実行する

```
$ pjsub wa2-pure.bash
```

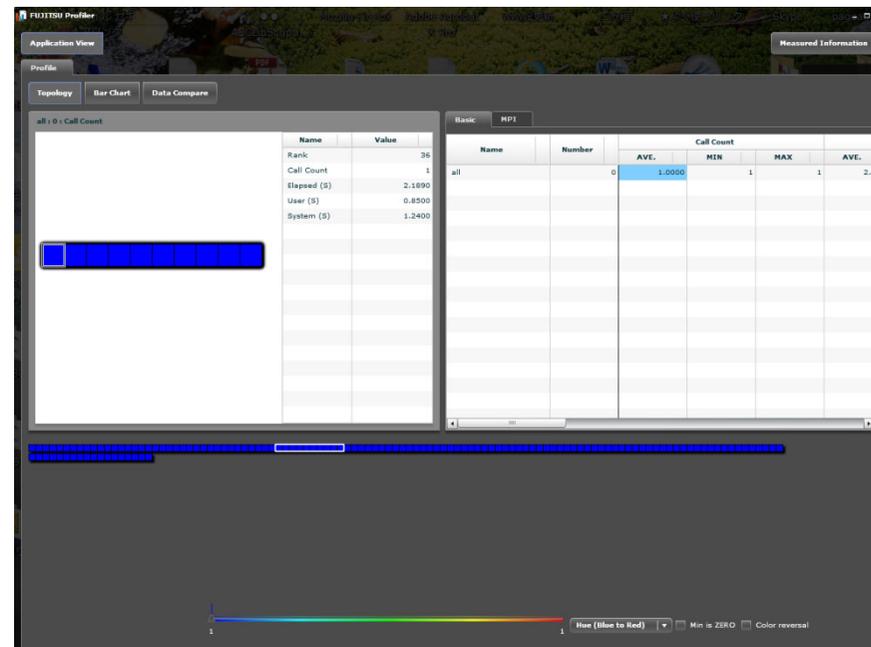
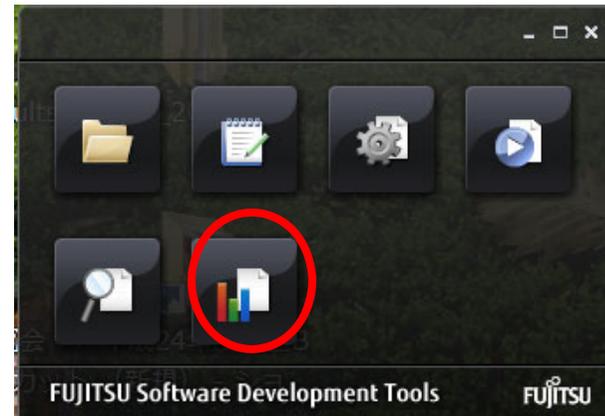
詳細プロファイラGUIによる表示例 (名大FX100)

- ▶ 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



詳細プロファイラGUIによる表示例 (名大FX100)

- ▶ 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 (名大FX100)

- ▶ 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

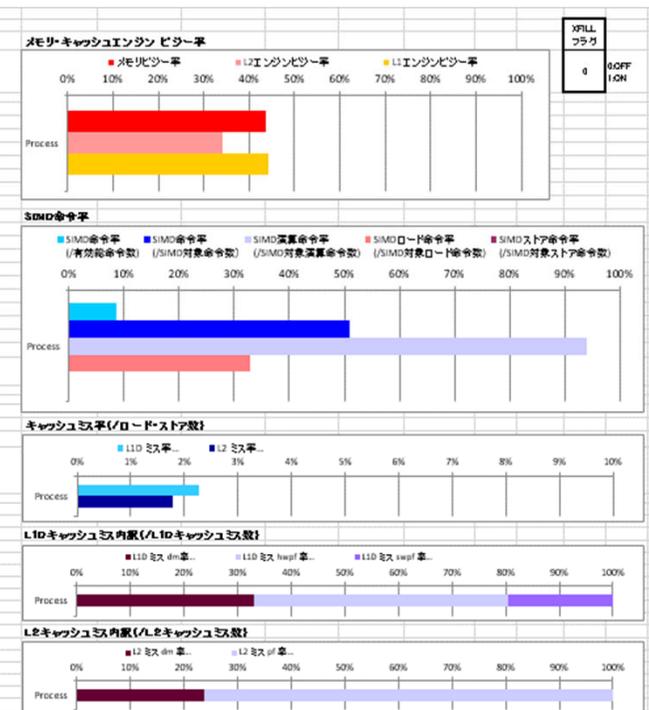
詳細プロファイラ (Excel形式)

▶ 「京」コンピュータ、FX100で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

FX10 PA information					Process No == 0					Interval Name == reazim						
Performance					Memory-Cache					SAMD						
Thread	実行時間 (sec)	演算小数の演算ピーク比	MFLOPS	MIPS	演算小数の演算数	メモリスループット (GB/sec)	L2エンジン スループット (GB/sec)	L2 スループット (GB/sec)	メモリビジー率	L2エンジンビジー率	L1エンジンビジー率	SAMD命令数 (/有効命令数)	SAMD演算命令数 (/SAMD対象命令数)	SAMD演算命令数 (/SAMD対象命令数)	SAMDロード命令数 (/SAMD対象ロード命令数)	SAMDストア命令数 (/SAMD対象ストア命令数)
Thread 0	0.02	2.67%	894	2200	8.45E+06	2.25	6.00	2.70								
Thread 1	0.02	2.67%	895	2274	8.45E+06	2.25	6.07	2.60								
Thread 2	0.02	2.70%	899	2277	8.54E+06	2.25	6.02	2.60								
Thread 3	0.02	2.62%	895	2255	8.49E+06	2.25	6.02	2.60								
Thread 4	0.02	2.62%	896	2271	8.47E+06	2.25	6.07	2.70								
Thread 5	0.02	2.67%	895	2267	8.47E+06	2.21	6.02	2.70								
Thread 6	0.02	2.70%	899	2264	8.54E+06	2.21	6.05	2.70								
Thread 7	0.02	2.70%	892	2296	8.54E+06	2.24	6.11	2.61								
Thread 8	0.02	2.62%	896	2271	8.50E+06	2.21	6.04	2.70	44%	84%						
Thread 9	0.02	2.62%	892	2255	8.45E+06	2.22	6.02	2.77								
Thread 10	0.02	2.62%	891	2275	8.40E+06	2.25	6.02	2.60								
Thread 11	0.02	2.67%	894	2292	8.49E+06	2.24	6.10	2.60								
Thread 12	0.02	2.67%	895	2264	8.49E+06	2.25	6.02	2.70								
Thread 13	0.02	2.62%	896	2261	8.40E+06	2.20	6.00	2.77								
Thread 14	0.02	2.62%	894	2270	8.47E+06	2.22	6.06	2.60								
Thread 15	0.02	2.71%	898	2255	8.55E+06	2.25	6.17	2.70								
Process	0.02	2.67%	6007	85271	1.24E+08	85.52	86.75	44.52								

Cache	L1I ミス率 (/有効命令数)	L1D ミス率 (/ロードストア数)	ロードストア数	L1D ミス数	L1D ミス率 (/L1D ミス数)	L1D miss hit 率 (/L1D ミス数)	L1D miss wup 率 (/L1D ミス数)	L2 ミス率 (/ロードストア数)	L2 ミス数	L2 ミス率 (/L2 ミス数)	L2 miss hit 率 (/L2 ミス数)	μDTLB ミス率 (/ロードストア数)	mDTLB ミス率 (/ロードストア数)
Thread 0	0.02%	2.27%	2.06E+07	4.67E+05	85.19%	47.04%	19.77%	1.89%	3.21E+05	24.86%	75.94%	0.00545%	0.00022%
Thread 1	0.02%	2.26%	2.06E+07	4.67E+05	85.27%	47.20%	19.43%	1.73%	3.54E+05	21.21%	75.29%	0.00535%	0.00022%
Thread 2	0.02%	2.23%	2.06E+07	4.69E+05	85.05%	47.50%	19.44%	1.70%	3.65E+05	25.82%	74.11%	0.00720%	0.00025%
Thread 3	0.02%	2.27%	2.07E+07	4.70E+05	82.95%	47.50%	19.55%	1.73%	3.67E+05	25.45%	74.45%	0.00672%	0.00025%
Thread 4	0.02%	2.27%	2.06E+07	4.67E+05	85.22%	47.17%	19.61%	1.70%	3.65E+05	23.72%	76.27%	0.00872%	0.00027%
Thread 5	0.02%	2.25%	2.07E+07	4.65E+05	85.28%	47.04%	19.65%	1.76%	3.63E+05	25.63%	74.12%	0.00730%	0.00044%
Thread 6	0.02%	2.27%	2.05E+07	4.65E+05	85.27%	47.04%	19.50%	1.73%	3.65E+05	25.84%	74.06%	0.00744%	0.00029%
Thread 7	0.02%	2.29%	2.06E+07	4.70E+05	82.28%	47.26%	19.51%	1.79%	3.65E+05	25.44%	74.55%	0.00767%	0.00040%
Thread 8	0.02%	2.26%	2.07E+07	4.67E+05	85.01%	47.44%	19.55%	1.76%	3.64E+05	24.69%	75.95%	0.00705%	0.00022%
Thread 9	0.02%	2.25%	2.06E+07	4.65E+05	85.33%	47.04%	19.70%	1.76%	3.64E+05	25.82%	74.02%	0.00730%	0.00020%
Thread 10	0.02%	2.29%	2.06E+07	4.70E+05	82.28%	47.50%	19.40%	1.79%	3.67E+05	25.69%	74.27%	0.00644%	0.00021%
Thread 11	0.02%	2.23%	2.07E+07	4.71E+05	85.04%	47.54%	19.41%	1.79%	3.70E+05	25.46%	74.54%	0.00671%	0.00025%
Thread 12	0.02%	2.29%	2.05E+07	4.67E+05	85.13%	47.23%	19.65%	1.79%	3.67E+05	25.70%	74.50%	0.00720%	0.00025%
Thread 13	0.02%	2.24%	2.07E+07	4.62E+05	85.59%	47.12%	19.49%	1.74%	3.69E+05	24.24%	75.74%	0.00672%	0.00024%
Thread 14	0.02%	2.27%	2.07E+07	4.70E+05	85.11%	47.04%	19.50%	1.73%	3.65E+05	25.72%	74.25%	0.00715%	0.00025%
Thread 15	0.02%	2.29%	2.07E+07	4.74E+05	82.28%	47.18%	19.54%	1.79%	3.70E+05	25.45%	74.54%	0.00672%	0.00020%
Process	0.02%	2.27%	2.05E+08	7.49E+06	85.16%	47.21%	19.55%	1.79%	3.54E+05	23.79%	76.21%	0.00727%	0.00051%



そのほかの最適化技法

ループ分割、ループ消滅とスレッド並列化、等



ループ分割とループ消滅の実例（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
- ▶ 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）

- ▶ 作業領域が多い実装のため
 - ▶ 最大問題サイズ: $NX=256, NY=256, NZ=128$ (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
 - ▶ 時間ステップ数が大きい場合、総合時間を考慮すると1ステップ当たりの実行時間を減らすしかない ⇒問題サイズを小さくする
- ▶ 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 E5-2600 v3 (Haswell-EP)、LLC: 45MB [L3/socket]
 - ▶ FX100、Sparc64 XI-fx、LLC: 24 MB [L2/node]
 - ▶ 次世代デバイス (Intel 3D Xpoint、ストレージクラスメモリ) などではTB級 (予想)
 - ⇒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

FX100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.pphofdm_passing_velocity_
517	10.5424	51.7083	128	173	ppohfdm_stress.pphofdm_update_stress_
476	9.7064	47.6076	213	353	ppohfdm_stress.pphofdm_passing_stress_
388	7.9119	38.8062	195	225	ppohfdm_velocity.pphofdm_update_vel_
370	7.5449	37.0059	176	210	ppohfdm_stress.pphofdm_update_stress_sponge_
274	5.5873	27.4044	199	226	ppohfdm_pfd3d.pphofdm_pdiffz3_p4_
274	5.5873	27.4044	169	196	ppohfdm_pfd3d.pphofdm_pdiffy3_m4_
247	5.0367	24.7039	139	166	ppohfdm_pfd3d.pphofdm_pdiffy3_p4_
236	4.8124	23.6038	229	256	ppohfdm_pfd3d.pphofdm_pdiffz3_m4_
218	4.4454	21.8035	108	136	ppohfdm_pfd3d.pphofdm_pdiffx3_m4_

FX100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 = $50.3[\text{sec.}]/87.4[\text{sec.}]$ (演算プロファイルから得た値) * 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 = $8.5[\text{sec.}]/47.6[\text{sec.}]$ (演算プロファイルから得た値) * 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

FX100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
      RLI = LAM (I,J,K)
      RMI = RIG (I,J,K)
      RM2 = RMI + RMI
      RLRM2 = RLI+RM2
      DXVXI = DXVX(I,J,K)
      DYVYI = DYVY(I,J,K)
      DZVZI = DZVZ(I,J,K)
      D3V3 = DXVXI + DYVYI + DZVZI
      DXVYDYVXI = DXVY(I,J,K)+DYVX(I,J,K)
      DXVZDZVXI = DXVZ(I,J,K)+DZVX(I,J,K)
      DYVZDZVYI = DYVZ(I,J,K)+DZVY(I,J,K)
      SXX (I,J,K) = SXX (I,J,K) + (RLRM2*(D3V3)-RM2*(DZVZI+DYVYI) ) * DT
      SYY (I,J,K) = SYY (I,J,K) + (RLRM2*(D3V3)-RM2*(DXVXI+DZVZI) ) * DT
      SZZ (I,J,K) = SZZ (I,J,K) + (RLRM2*(D3V3)-RM2*(DXVXI+DYVYI) ) * DT
      SXY (I,J,K) = SXY (I,J,K) + RMI * DXVYDYVXI * DT
      SXZ (I,J,K) = SXZ (I,J,K) + RMI * DXVZDZVXI * DT
      SYZ (I,J,K) = SYZ (I,J,K) + RMI * DYVZDZVYI * 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
```

```
R4I = C4I/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))*R4I
```

```
    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
  SYX (I,J,K) = ( SYX (I,J,K) + (RLTHETA + RM2*DXVY(I,J,K))*DT ) *QG
  SYY (I,J,K) = ( SYY (I,J,K) + (RLTHETA + RM2*DYVY(I,J,K))*DT ) *QG
  SYZ (I,J,K) = ( SYZ (I,J,K) + (RLTHETA + RM2*DYVZ(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）

▶ ループ消滅 (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))
```

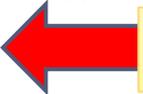
```
  RMAYZ(I) = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I,J+1,K) + 1.0/RIG(I,J,K+1) + 1.0/RIG(I,J+1,K+1))
```

```
  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
```

 **ループ分割点**

```
  SXX (I,J,K) = ( SXX (I,J,K) + (RLTHETA(I) + RM2(I)*DXVX(I,J,K))*DT ) * QG(I)
```

```
  SYX (I,J,K) = ( SYX (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
```

ループ消滅 - 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
```

ループ消滅 - 2重ループ化

▶ 例)

```
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
```

```
  SYX (I,J,K) = ( SYX (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
```

利点: ループ長が増える
NZ → NZ*NY

このループは連続:
コンパイラによるプリフェッチコード生成が可能

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
  SYX (I,J,K) = ( SYX (I,J,K)+ (RLTHETA + RM2*DYVY(I,J,K))*DT )*QG
  SYY (I,J,K) = ( SYY (I,J,K)+ (RLTHETA + RM2*DYVY(I,J,K))*DT )*QG
  SYZ (I,J,K) = ( SYZ (I,J,K)+ (RLTHETA + RM2*DZVZ(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
  SYX (I,J,K) = ( SYX (I,J,K) + (RLTHETA + RM2*DYVY(I,J,K))*DT ) *QG
  SYY (I,J,K) = ( SYY (I,J,K) + (RLTHETA + RM2*DYVY(I,J,K))*DT ) *QG
  SYZ (I,J,K) = ( SYZ (I,J,K) + (RLTHETA + RM2*DZVZ(I,J,K))*DT ) *QG
  SZZ (I,J,K) = ( SZZ (I,J,K) + (RLTHETA + RM2*DZVZ(I,J,K))*DT ) *QG
ENDDO; ENDDO; ENDDO
```

```
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;
```

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

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)

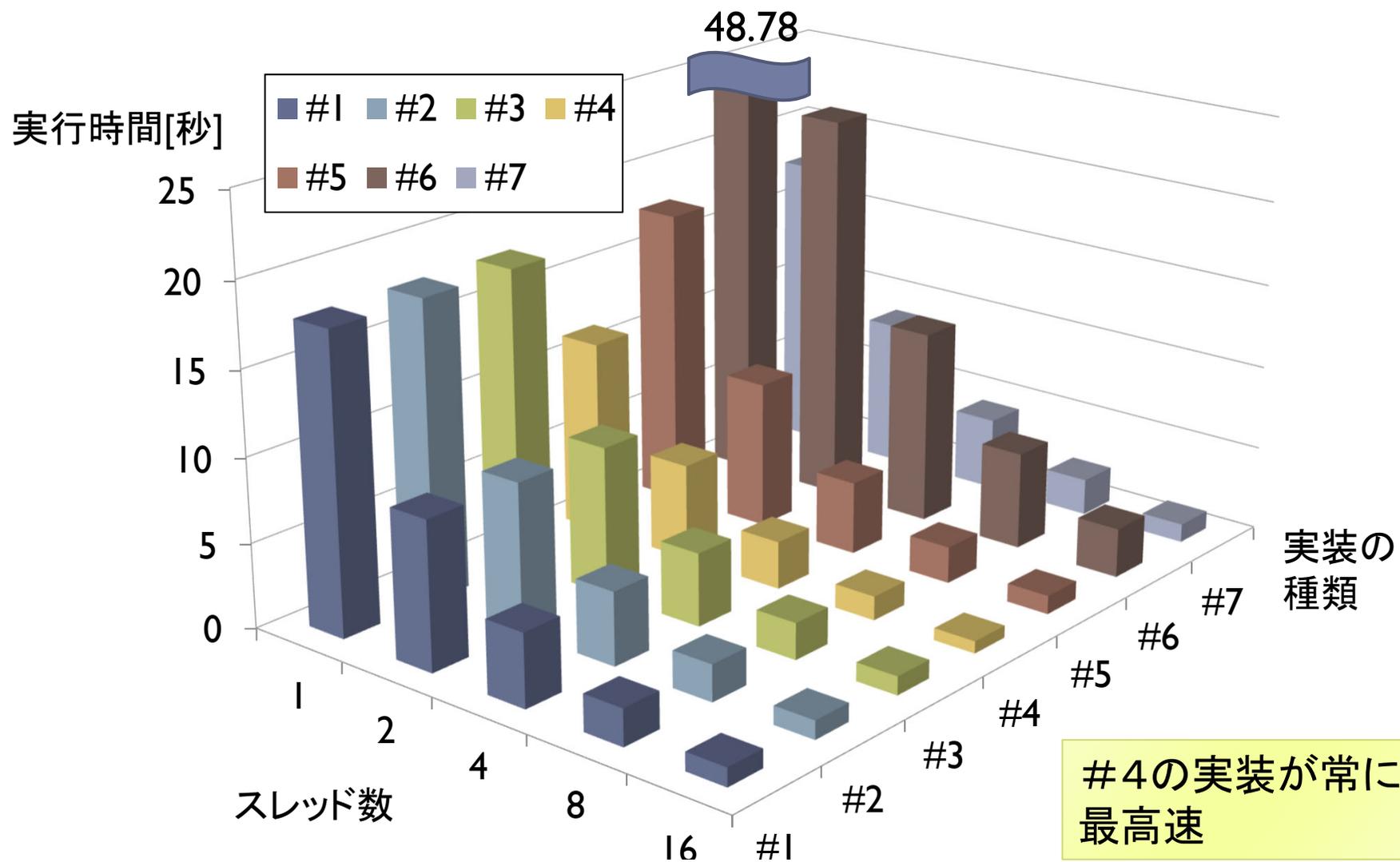
- ▶ #1 : 基の3重ループコード(ベースライン)
- ▶ #2: I-ループ分割のみ
- ▶ #3: J-ループ分割のみ
- ▶ #4: K-ループ分割のみ
- ▶ #5: #2ループに対するループ消滅 (2重ループ化)
- ▶ #6 : #1ループに対するループ消滅 (1重ループ化)
- ▶ #7 : #1ループに対するループ消滅 (2重ループ化)



ループ分割・ループ消滅の効果

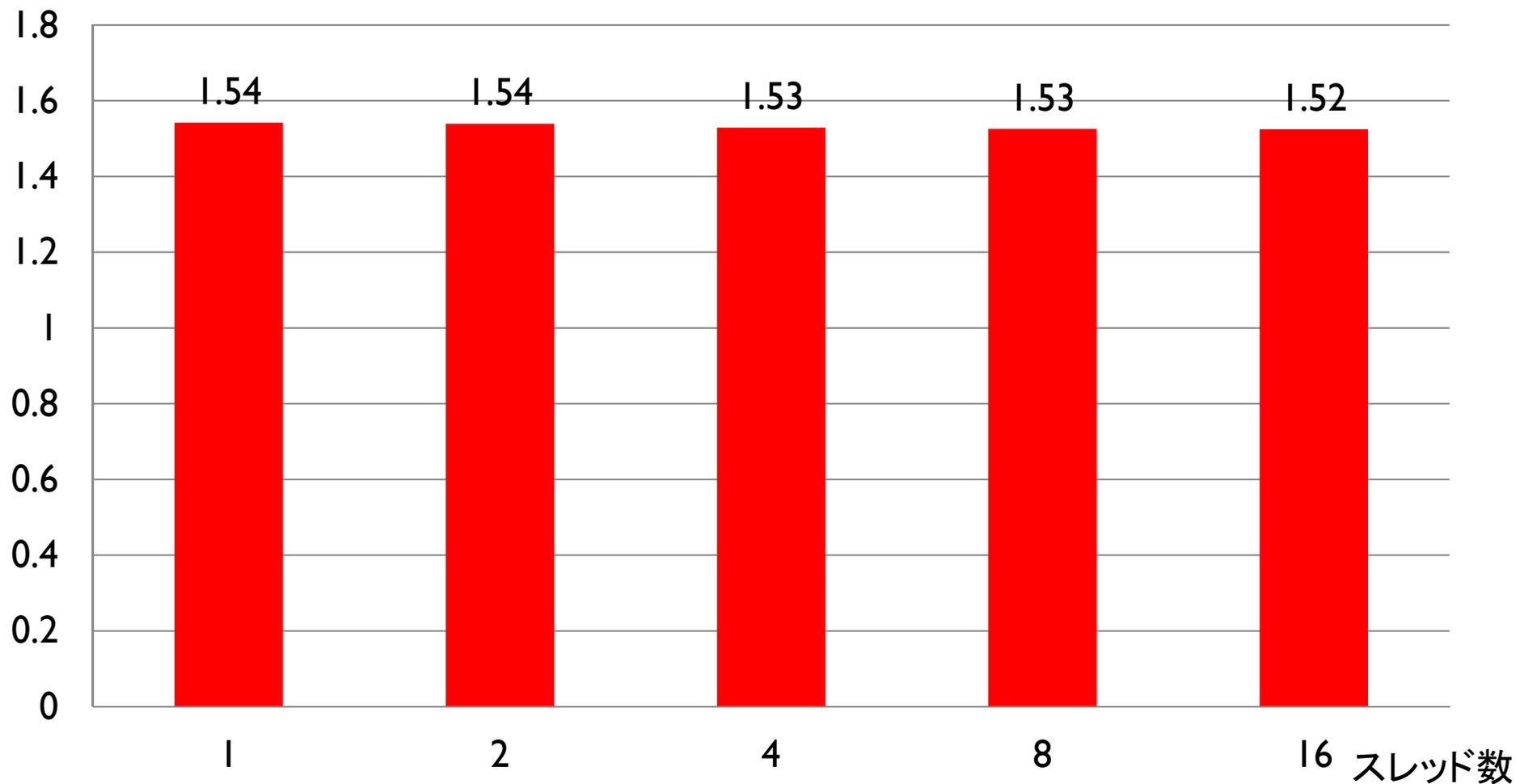
- ▶ 東京大学情報基盤センターFX10を利用
 - ▶ 1ノード、16スレッド
 - ▶ Sparc64 IV-fx (1.848 GHz)
- ▶ 最外ループに対して、OpenMPが適用可能
 - ▶ parallel do構文で並列化可能
- ▶ スレッド数は、1～16まで変更可能

チューニングの結果



1 (ベースライン) に対する速度向上

SpeedUP



#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  
        SYX (I,J,K) = ( SYX (I,J,K) + (RLTHETA + RM2*DYVY(I,J,K))*DT ) *QG  
        SYY (I,J,K) = ( SYY (I,J,K) + (RLTHETA + RM2*DYVY(I,J,K))*DT ) *QG  
        SYZ (I,J,K) = ( SYZ (I,J,K) + (RLTHETA + RM2*DZVZ(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  
      ENDDO; ENDDO; ENDDO  
    !$omp end parallel do  
  ENDDO; ENDDO; ENDDO  
!$omp end parallel do
```

ループ消滅への期待

- ▶ 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オーバーヘッドを考えると、ノードあたりのNZはスレッド数の2~3倍必要
 - 例) 68スレッドなら、NZは140~210以上
 - HTで272スレッド実行なら、NZは540~810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消滅が必須

計算機環境

(Xeon Phi (KNC) の8ノード)

● Intel Xeon Phi (KNC)

- **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 -mcmmodel=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)
 - ▶ $NX * NY * NZ = 1536 \times 768 \times 240 / 8$ 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)



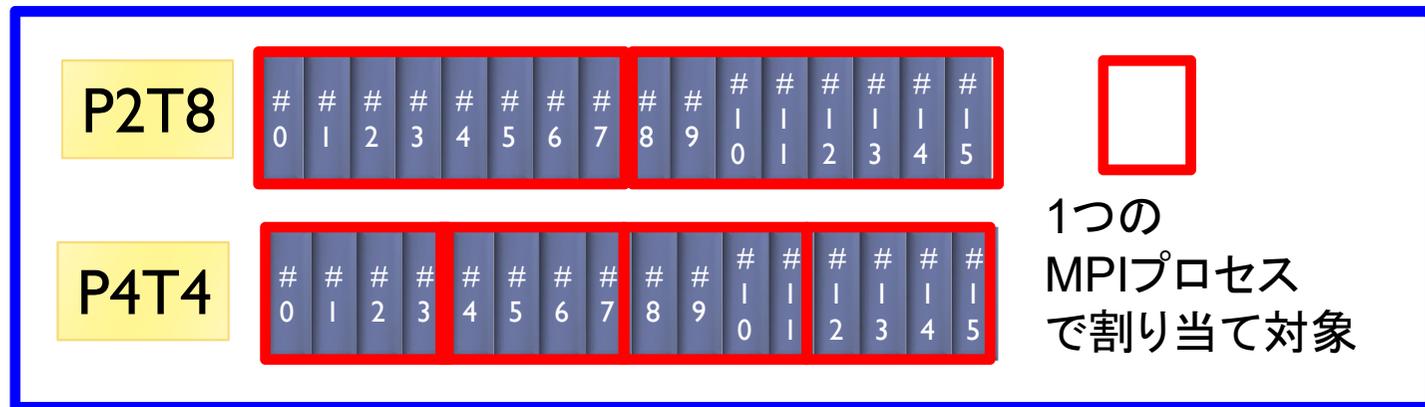
ハイブリッドMPI/OpenMP実行の詳細

- ▶ 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のため)

- ▶ P16T120
- ▶ P32T60
- ▶ P64T30
- ▶ P128T15
- ▶ P240T8
- ▶ P480T4

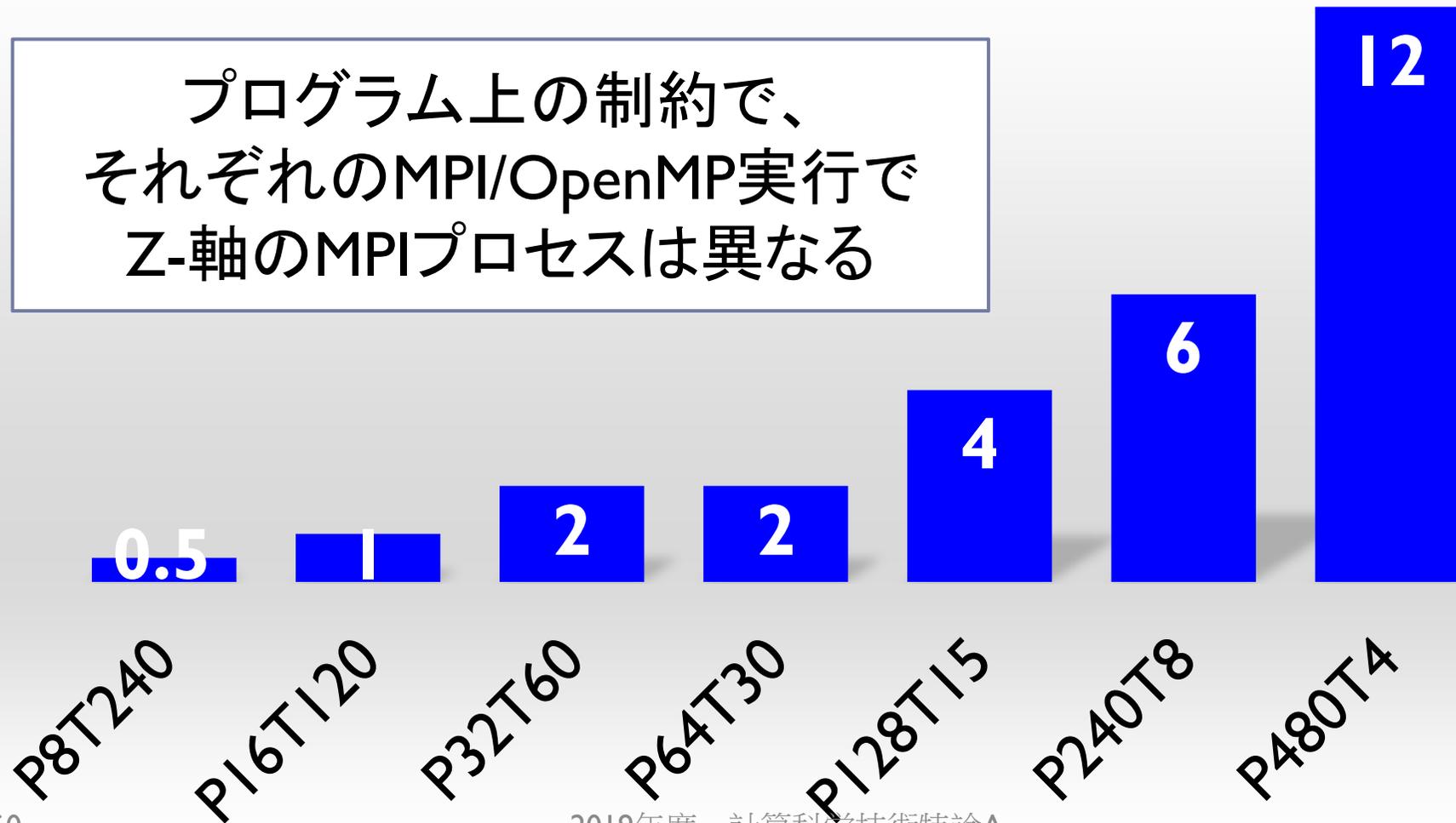


- ▶ **P960T2以下は、この環境ではMPIエラーで実行できなかったため除外**

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

■ Loop length per thread

プログラム上の制約で、
それぞれのMPI/OpenMP実行で
Z-軸のMPIプロセスは異なる



ループ消滅等による 最大の速度向上(Xeon Phi、8ノード)

■ Speedup [%]

558

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

$NX * NY * NZ = 1536 * 768 * 240 / 8ノード$

200

171

30

20

51

update_stress

update_vel

update_stress_sponge

diff_*

passing_*

whole

演算カーネルの種類

最速のコード (update_stress)

●Xeon Phi(KNC) (P240T8)

```
!$omp parallel do private
```

```
(k,j,i,RLI,RMI,RM2,RLRM2,DXVXI,DYVYI,DZVZI,D3V3,DXVYDYVXI,DXVZDZVXI,DYVZDZVI)
```

```
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
```

```
    RLI = LAM (I,J,K); RMI = RIG (I,J,K)
```

```
    RM2 = RMI + RMI; RLRM2 = RLI+RM2
```

```
    DXVXI = DXVX(I,J,K); DYVYI = DYVY(I,J,K); DZVZI = DZVZ(I,J,K);
```

```
    D3V3 = DXVXI + DYVYI + DZVZI;
```

```
    SXX (I,J,K) = SXX (I,J,K) + (RLRM2*(D3V3)-RM2*(DZVZI+DYVYI) ) * DT
```

```
    SYX (I,J,K) = SYX (I,J,K) + (RLRM2*(D3V3)-RM2*(DXVXI+DZVZI) ) * DT
```

```
    SZZ (I,J,K) = SZZ (I,J,K) + (RLRM2*(D3V3)-RM2*(DXVXI+DYVYI) ) * DT
```

```
    DXVYDYVXI = DXVY(I,J,K)+DYVX(I,J,K);
```

```
    DXVZDZVXI = DXVZ(I,J,K)+DZVX(I,J,K);
```

```
    DYVZDZVYI = DYVZ(I,J,K)+DZVY(I,J,K)
```

```
    SXY (I,J,K) = SXY (I,J,K) + RMI * DXVYDYVXI * DT
```

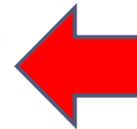
```
    SXZ (I,J,K) = SXZ (I,J,K) + RMI * DXVZDZVXI * DT
```

```
    SYZ (I,J,K) = SYZ (I,J,K) + RMI * DYVZDZVYI * 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

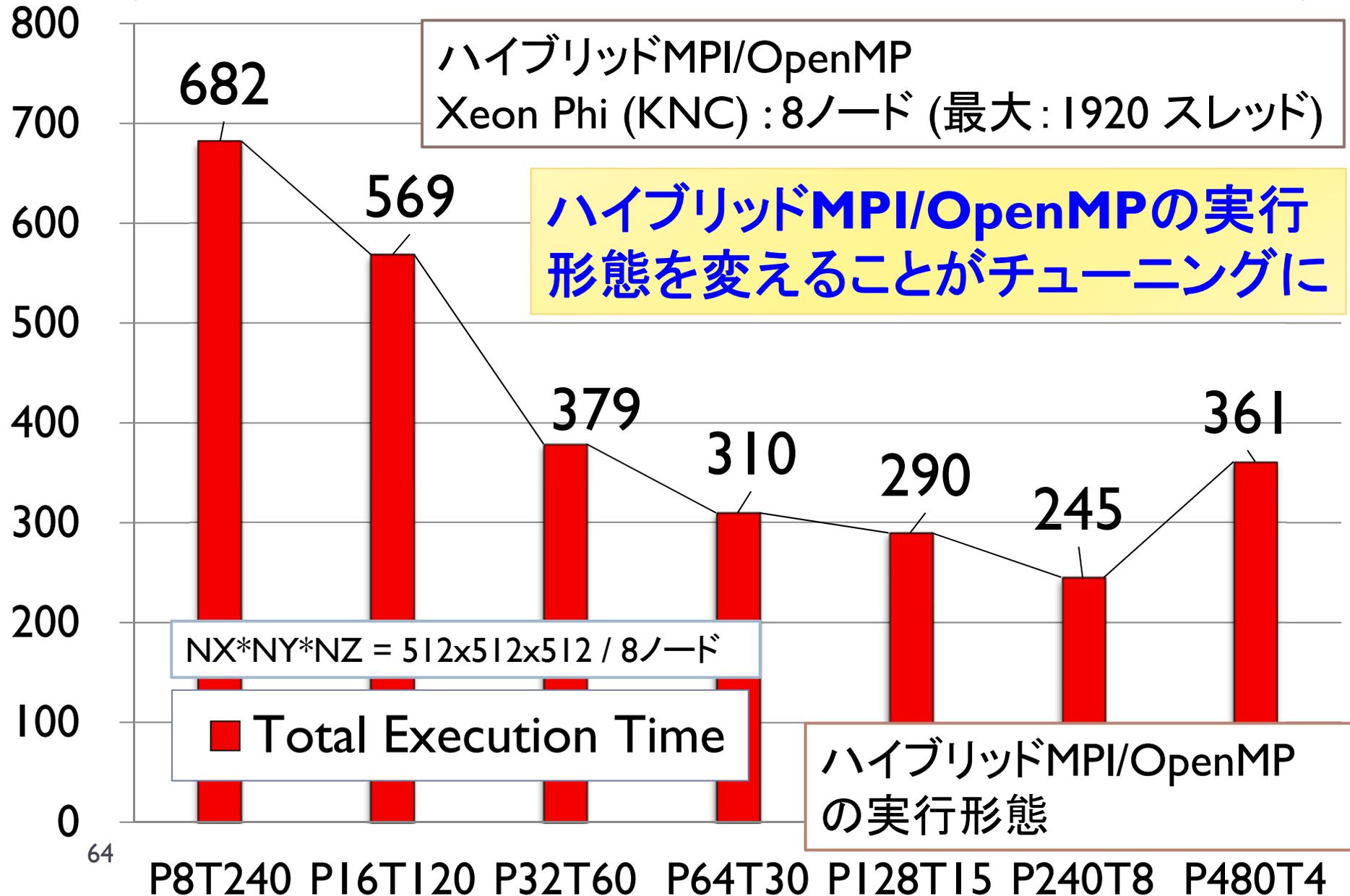
そのほかの最適化技法

ハイブリッドMPI/OpenMPの実行形態



ハイブリッドMPI/OpenMP実行の効果

[秒] (Seism3D、全体時間：2000時間ステップ)



そのほかの最適化技法

B/F値の低いカーネル作成



オリジナル実装 (ベクトル計算機向き)

速度に関する4次の中央差分スキーム (def_stress)

```
call ppohFDM_pdiffx3_m4_OAT(VX,DXVX, NXP,NYP,NZP,NXP0,NXPI,NYP0,...)
call ppohFDM_pdiffy3_p4_OAT(VX,DYVX, NXP,NYP,NZP,NXP0,NXPI,NYP0,...)
call ppohFDM_pdiffz3_p4_OAT(VX,DZVX, NXP,NYP,NZP,NXP0,NXPI,NYP0,...)
call ppohFDM_pdiffy3_m4_OAT(VY,DYVY, NXP,NYP,NZP,NXP0,NXPI,NYP0,...)
call ppohFDM_pdiffx3_p4_OAT(VY,DXVY, NXP,NYP,NZP,NXP0,NXPI,NYP0,...)
call ppohFDM_pdiffz3_p4_OAT(VY,DZVY, NXP,NYP,NZP,NXP0,NXPI,NYP0,...)
call ppohFDM_pdiffx3_p4_OAT(VZ,DXVZ, NXP,NYP,NZP,NXP0,NXPI,NYP0,...)
call ppohFDM_pdiffy3_p4_OAT(VZ,DYVZ, NXP,NYP,NZP,NXP0,NXPI,NYP0,...)
call ppohFDM_pdiffz3_m4_OAT(VZ,DZVZ, NXP,NYP,NZP,NXP0,NXPI,NYP0,...)
```

```
if( is_fs .or. is_nearfs ) then
  call ppohFDM_bc_vel_deriv( KFSZ,NIFS,NJFS,IFSX,IFSX,JFSX,JFSY,JFSZ )
end if
```

モデル境界処理

```
call ppohFDM_update_stress(I, NXP, I, NYP, I, NZP)
```

leap-frogスキームによる陽的時間発展 (update_stress)

オリジナル実装 (ベクトル計算機向き)

```
subroutine OAT_InstallppohFDMupdate_stress(..)
!$omp parallel do private(i,j,k,RLI,RMI,RM2,RLRM2,DXVXI,DYVYI,DZVZI,...)
  do k = NZ00, NZ01
    do j = NY00, NY01
      do i = NX00, NX01
        RLI = LAM (I,J,K); RMI = RIG (I,J,K); RM2 = RMI + RMI; RLRM2 = RLI+RM2
        DXVXI = DXVX(I,J,K); DYVYI = DYVY(I,J,K); DZVZI = DZVZ(I,J,K)
        D3V3 = DXVXI + DYVYI + DZVZI
        SXX (I,J,K) = SXX (I,J,K) + (RLRM2*(D3V3)-RM2*(DZVZI+DYVYI) ) * DT
        SYX (I,J,K) = SYX (I,J,K) + (RLRM2*(D3V3)-RM2*(DXVXI+DZVZI) ) * DT
        SZZ (I,J,K) = SZZ (I,J,K) + (RLRM2*(D3V3)-RM2*(DXVXI+DYVYI) ) * DT
        DXVYDYVXI = DXVY(I,J,K)+DYVX(I,J,K); DXVZDZVXI =
DXVZ(I,J,K)+DZVX(I,J,K)
        DYVZDZVYI = DYVZ(I,J,K)+DZVY(I,J,K)
        SXY (I,J,K) = SXY (I,J,K) + RMI * DXVYDYVXI * DT
        SXZ (I,J,K) = SXZ (I,J,K) + RMI * DXVZDZVXI * DT
        SYZ (I,J,K) = SYZ (I,J,K) + RMI * DYVZDZVYI * DT
      end do
    end do
  end do
return
end 67
```

各手続き呼び出しで
入出力の配列が多い
⇒ B/F値の増加: ~1.7

leap-frogスキームによる陽的時間発展
(update_stress)

別実装（スカラ計算機向き）

- 実装1 (IF文あり)

- ▶ 以下の処理をループ中に記載:

1. 速度に関する4次の中央差分スキーム
2. モデル境界処理
3. leap-frogスキームによる陽的時間発展

- 実装2 (IF文なし。モデル境界処理のIF文ループが別途必要)

- ▶ 実装1からIF文を取り除くため、ループを再構築。

- ▶ 計算順序を変更する。しかし、丸め誤差を除く演算結果は同一。

- ▶ **[主ループ]**

1. 速度に関する4次の中央差分スキーム
2. leap-frogスキームによる陽的時間発展

- ▶ **[モデル境界処理のループ]**

1. 速度に関する4次の中央差分スキーム
2. モデル境界処理

- ▶ 68. leap-frogスキームによる陽的時間発展

実装 1 (スカラ計算機向き)

Stress tensor of Sxx, Syy, Szz

```
!$omp parallel do private
(i,j,k,RLI,RM1,RM2,RLRM2,DXVX, ...)
do k_j=1, (NZ01-NZ00+1)*(NY01-NY00+1)
```

速度に関する
4次の中央差分
スキーム (def_stress)

```
k=(k_j-1)
j=mod((k-1),NY01-NY00+1)+1
do i = NXP, RLI, -1, 1
  RM2 = 1 - 2*RLM2
  ! 4th order central difference
  DXVX0 = (VX(I,J,K) - VX(I-1,J,K))*C40/dx &
    - (VX(I+1,J,K) - VX(I-2,J,K))*C41/dx
  DYVY0 = (VY(I,J,K) - VY(I,J-1,K))*C40/dy &
    - (VY(I,J+1,K) - VY(I,J-2,K))*C41/dy
  DZVZ0 = (VZ(I,J,K) - VZ(I,J,K-1))*C40/dz &
    - (VZ(I,J,K+1) - VZ(I,J,K-2))*C41/dz
```

```
! truncate diff vector
! X dir
if (idx==0) then
  if (i==1) then
    DXVX0 = (VX(I,J,K) - 0.0_PN ) / DX
  end if
  if (i==2) then
    DXVX0 = (VX(2,J,K) - VX(1,J,K)) / DX
  end if
end if
if (idx == IP-1 ) then
  if (i==NXP) then
    DXVX0 = (VX(NXP,J,K) - VX(NXP-1,J,K)) / DX
  end if
```

モデル境界処理

```
! Y dir
if (idy == 0) then ! Shallowmost
  if (j==1) then
    DYVY0 = (VY(I,1,K) - 0.0_PN ) / DY
  end if
  if (j==2) then
    DYVY0 = (VY(2,I,K) - VY(1,I,K)) / DY
  end if
end if
if (idy == JP-1 ) then
  if (j==NYP) then
    DYVY0 = (VY(I,J,NYP) - VY(I,J,NYP-1)) / DY
  end if
end if
! Z dir
if (idz == 0) then ! Shallowmost
  if (k==1) then
    DZVZ0 = (VZ(I,J,1) - 0.0_PN ) / DZ
  end if
  if (k==2) then
    DZVZ0 = (VZ(I,J,2) - VZ(I,J,1)) / DZ
  end if
end if
if (idz == KP-1 ) then
  if (k==NZP) then
    DZVZ0 = (VZ(I,J,NZP) - VZ(I,J,NZP-1)) / DZ
  end if
end if
```

leap-frogスキームによる陽的時間発展 (update_stress)

```
DXVXI = DXVX0; DYVYI = DYVY0
DZVZI = DZVZ0; D3V3 = DXVXI + DYVYI + DZVZI
SXX (I,J,K) = SXX (I,J,K) &
  + (RLRM2*(D3V3)-RM2*(DZVZI+DYVYI)) * DT
SYY (I,J,K) = SYY (I,J,K) &
  + (RLRM2*(D3V3)-RM2*(DXVXI+DZVZI)) * DT
SZZ (I,J,K) = SZZ (I,J,K) &
  + (RLRM2*(D3V3)-RM2*(DXVXI+DYVYI)) * DT
end do
!$omp end parallel do
```

😊B/F値が
0.4 まで減少
☹️IF文が内在
コンパイラによる最適化が
難しい

実装2 (IF文なし)

Stress tensor of Sxx, Syy, Szz

```
!$omp parallel do private(i,j,k,RLI,RMI,...)
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
    RLI = LAM (I,J,K); RMI = RIG (I,J,K);
    RM2 = RMI + RMI; RLRM2 = RLI+RM2
    ! 4th order diff (DXVX,DYVY,DZVZ)
```

速度に関する
4次の中央差分
スキーム (def_stress)

```
DXVX0 = (VX(I,J,K) - VX(I-1,J,K))*C40/dx - (VX(I+1,J,K)-VX(I-2,J,K))*C41/dx
DYVY0 = (VY(I,J,K) - VY(I,J-1,K))*C40/dy - (VY(I,J+1,K)-VY(I,J-2,K))*C41/dy
DZVZ0 = (VZ(I,I,K) - VZ(I,J,K-1))*C40/dz - (VZ(I,J,K+1)-VZ(I,J,K-2))*C41/dz
DXVXI = DXVX0; DYVYI = DYVY0;
DZVZI = DZVZ0;
D3V3 = DXVXI + DYVYI + DZVZI;
SXX (I,J,K) = SXX (I,J,K) + (RLRM2*(D3V3)-RM2* (DZVZI+DYVYI) ) * DT
SYY (I,J,K) = SYY (I,J,K) + (RLRM2*(D3V3)-RM2* (DXVXI+DZVZI) ) * DT
SZZ (I,J,K) = SZZ (I,J,K) + (RLRM2*(D3V3)-RM2* (DXVXI+DYVYI) ) * DT
```

leap-frogスキームによる陽的時間発展
(update_stress)

😊B/F値とコンパイラ
最適化の観点から
良いコード

実装2 (IF文なし)

モデル境界処理のためのループ

```
! 2nd replace
if (is_fs .or. is_nearfs ) then
!$omp parallel do private(i,j,k,RLI,RMI,...)
do i=NX00,NX01
do j=NY00,NY01
do k = KFSZ(i,j)-1, KFSZ(i,j)+1, 2
RLI = LAM (I,J,K); RMI = RIG (I,J,K);
RM2 = RMI + RMI; RLRM2 = RLI+RM2
! 4th order diff
DXVX0 = (VX(I,J,K) -VX(I-1,J,K))*C40/dx &
- (VX(I+1,J,K)-VX(I-2,J,K))*C41/dx
DYVX0 = (VX(I,J+1,K)-VX(I,J,K) ) *C40/dy &
- (VX(I,J+2,K)-VX(I,J-1,K))*C41/dy
DXVY0 = (VY(I+1,J,K)-VY(I ,J,K))*C40/dx &
- (VY(I+2,J,K)-VY(I-1,J,K))*C41/dx
DYVY0 = (VY(I,J,K) -VY(I,J-1,K))*C40/dy &
- (VY(I,J+1,K)-VY(I,J-2,K))*C41/dy
DXVZ0 = (VZ(I+1,J,K)-VZ(I ,J,K))*C40/dx &
- (VZ(I+2,J,K)-VZ(I-1,J,K))*C41/dx
DYVZ0 = (VZ(I,J+1,K)-VZ(I,J,K) ) *C40/dy &
- (VZ(I,J+2,K)-VZ(I,J-1,K))*C41/dy
DZVZ0 = (VZ(I,J,K) -VZ(I,J,K-1))*C40/dz &
- (VZ(I,J,K+1)-VZ(I,J,K-2))*C41/dz
```

速度に関する
4次の中央差分
スキーム (def_stress)

モデル 境界 処理

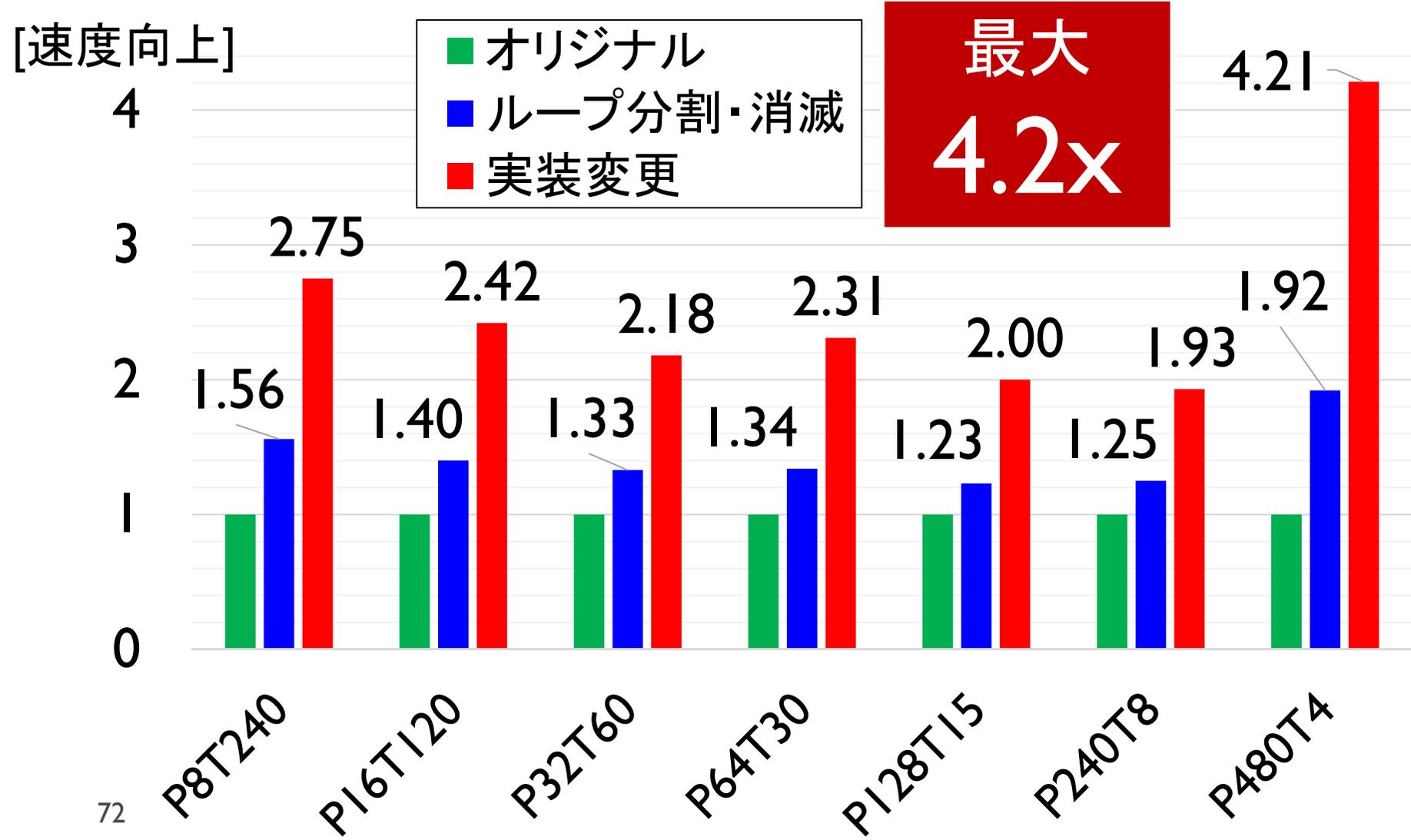
leap-frogスキーム
による
陽的時間発展
(update_stress)

```
!-derive
(K==KFSZ(I,J)+1) then
DZVX0 = (VX(I,J,KFSZ(I,J)+2)-VX(I,J,KFSZ(I,J)+1) )/ DZ
DZVY0 = (VY(I,J,KFSZ(I,J)+2)-VY(I,J,KFSZ(I,J)+1) )/ DZ
else if (K==KFSZ(I,J)-1) then
DZVX0 = (VX(I,J,KFSZ(I,J) )-VX(I,J,KFSZ(I,J)-1) )/ DZ
DZVY0 = (VY(I,J,KFSZ(I,J) )-VY(I,J,KFSZ(I,J)-1) )/ DZ
end if
DXVXI = DXVX0
DYVYI = DYVY0
DZVZI = DZVZ0
D3V3 = DXVXI + DYVYI + DZVZI
DXVYDYVXI = DXVY0+DYVX0
DXVZDZVXI = DXVZ0+DZVX0
DYVZDZVYI = DYVZ0+DZVY0
if (K==KFSZ(I,J)+1)then
KK=2
else
KK=1
end if
SSXX (I,J,K) = SSXX (I,J,KK) &
+ (RLRM2*(D3V3)-RM2*(DZVZI+DYVYI) ) * D
SYY (I,J,K) = SSYY (I,J,KK) &
+ (RLRM2*(D3V3)-RM2*(DXVXI+DZVZI) ) * DT
SZZ (I,J,K) = SSZZ (I,J,KK) &
+ (RLRM2*(D3V3)-RM2*(DXVXI+DYVYI) ) * DT
SXY (I,J,K) = SSXY (I,J,KK) + RMI * DXVYDYVXI * DT
SXZ (I,J,K) = SSXZ (I,J,KK) + RMI * DXVZDZVXI * DT
SYZ (I,J,K) = SSYZ (I,J,KK) + RMI * DYVZDZVYI * DT
```

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

速度向上 (update_stress)

(a) Xeon Phi (8 Nodes)



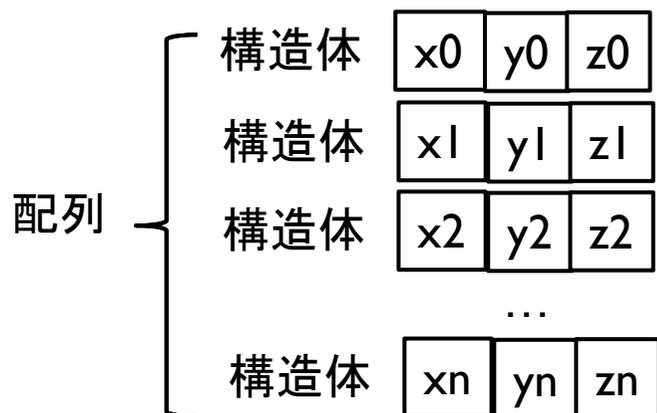
そのほかの最適化技法

データ構造の変換：AoS、SoA



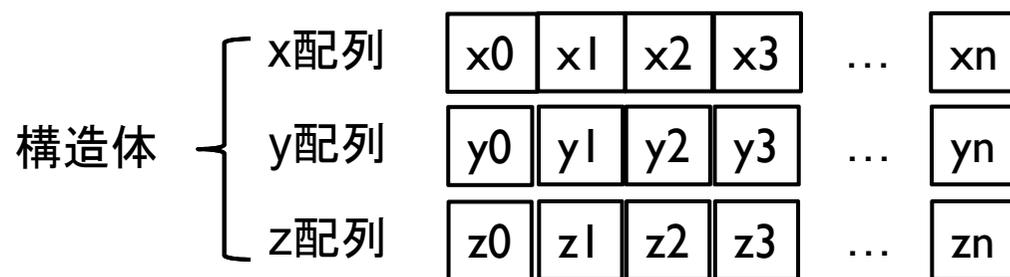
データ構造変換：AoS vs. SoA

▶ 構造体配列：Array of Structures (AoS)



- 利点
 - 自然なデータ構造で扱いやすい
- 欠点
 - $x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow \dots \rightarrow x_n$ とアクセスするとキャッシュミスが生じ、性能劣化する

▶ 配列構造体：Structure of Arrays (SoA)



Modylasにおけるデータ構造変換による高速化の実例

表 配列形状変更によるP2P(code1)演算時間の削減効果. 単位はms.
¹ITO, -xHost -qopt-zmm-usage=high指定. ²Oakforest-PACS, -axMIC-AVX512指定.

	従来 (AoS)	SoA	実行時間 削減率
Skylake-SP ¹	2.140	1.976	8%
KNL ²	8.582	6.048	30%

(source) Modylas:

[1] www.modylas.org

[2] Yoshimichi Ando, et.al., "MODYLAS: A Highly Parallelized General-Purpose Molecular Dynamics Simulation Program for Large-Scale Systems with Long-Range Forces Calculated by Fast Multipole Method (FMM) and Highly Scalable Fine-Grained New Parallel Processing Algorithms", J. Chem. Theo. Comp.,9, 3201-3209 (2013)

[3] Yoshimichi Andoh, et.al., "A thread-level parallelization of pairwise additive potential and force calculations suitable for current many-core architectures", The Journal of Supercomputing, 74 (6), pp. 2449-2469 (2018)



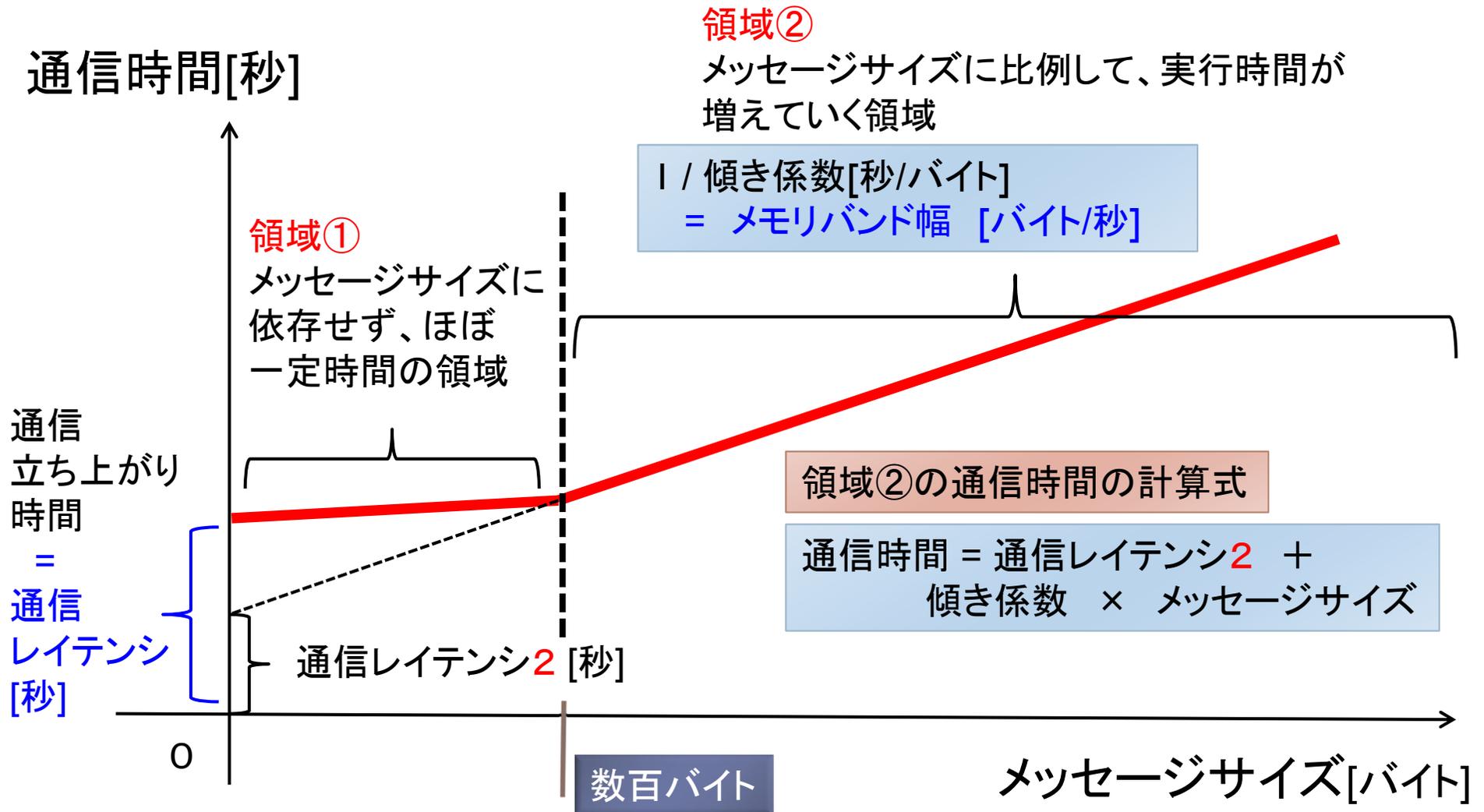
AoS から SoA のデータ変換の注意点

- ▶ 実際のプログラムでは、元のデータ構造が AoS になっていることが多く、プログラム全体を SoA に書き換えることができないことが多い
- ▶ そのため、以下の手順を取ることが多い
 1. 対象の計算に入る前に SoA の配列を確保
 2. AoS のデータを SoA に変換 (コピー)
 3. SoA で計算
 4. SoA のデータを AoS に変換 (コピー)
- ▶ 1 により **2 倍のメモリ量が必要**、2, 4 により、**コピー時間が必要**。このデメリットに対して、**3 の計算時間の高速化で元が取れる** 場合のみ有効

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）

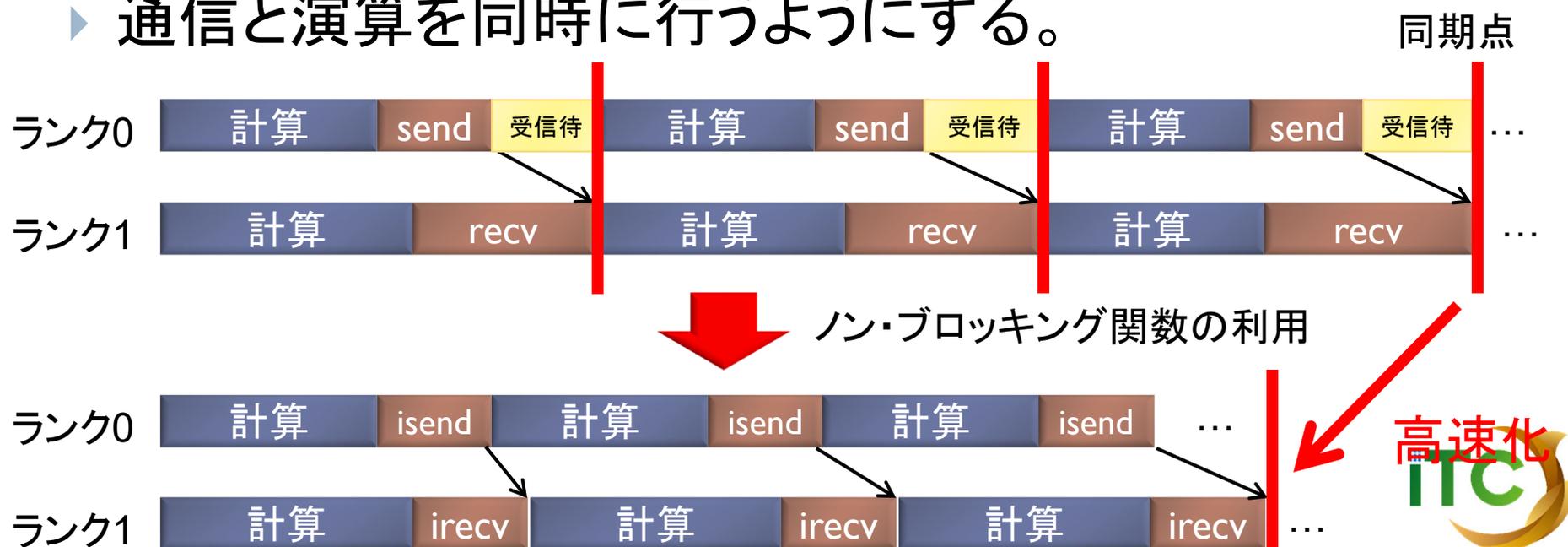
- ▶ 自分のアプリケーションの通信パターンについて、以下の観点を知らないと通信の最適化ができない
 - ▶ <領域①><領域②>のどちらになるのか
 - ▶ 通信の頻度(回数)はどれほどか
- ▶ **領域①の場合**
 - ▶ 「通信レイテンシ」が実行時間のほとんど
 - ▶ 通信回数を削減する
 - ▶ 細切れに送っているデータをまとめて1回にする、など
- ▶ **領域②の場合**
 - ▶ 「メッセージ転送時間」が実行時間のほとんど
 - ▶ メッセージサイズを削減する
 - ▶ 冗長計算をして計算量を増やしてでもメッセージサイズを削減する、など

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。



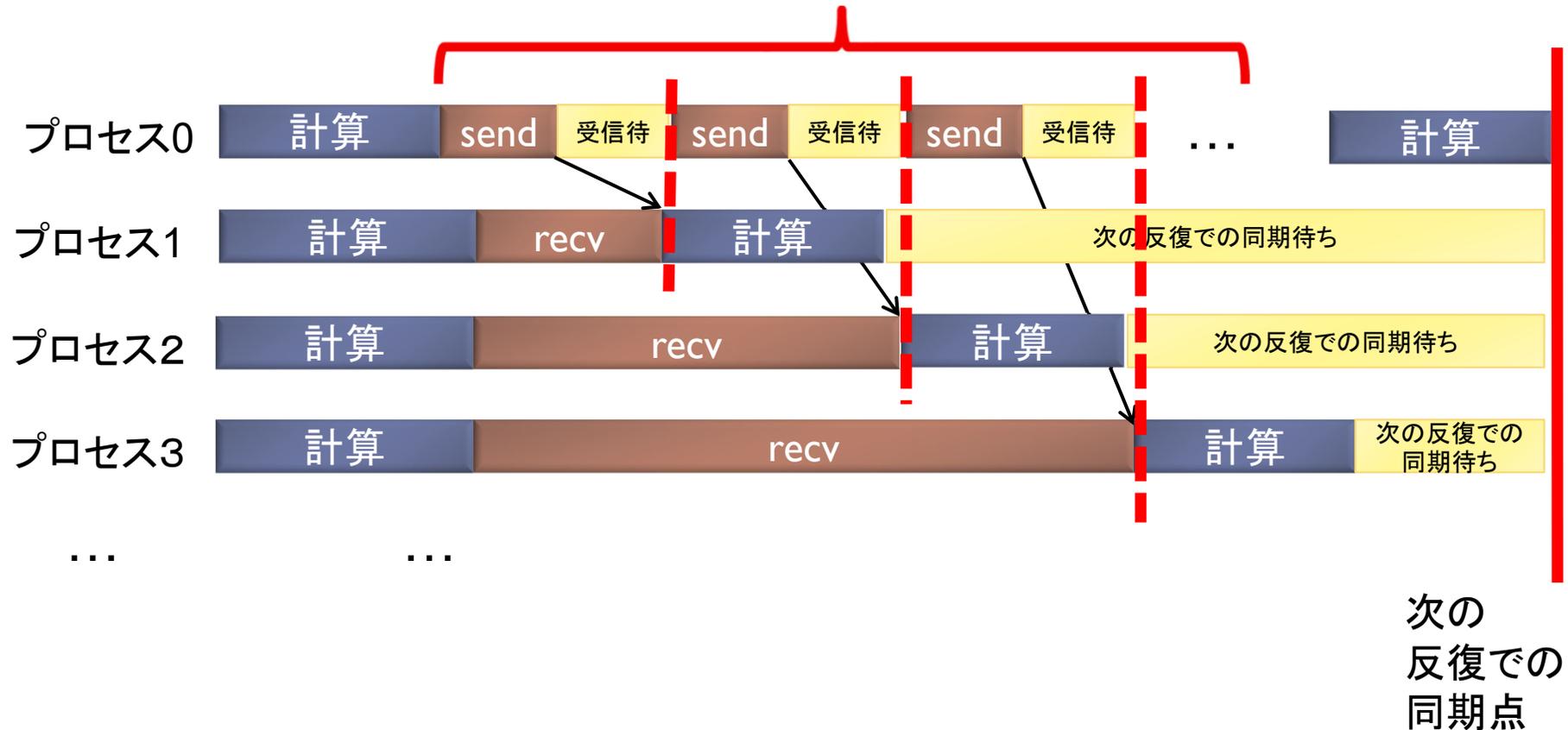
非同期通信： Isend、Irecv、永続的通信関数



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。
 - ▶ ハンドシェイ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

- ▶ **icom** : 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, icsomm,  
                  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
- ▶ **icsomm**: 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つた場合は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
```

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。

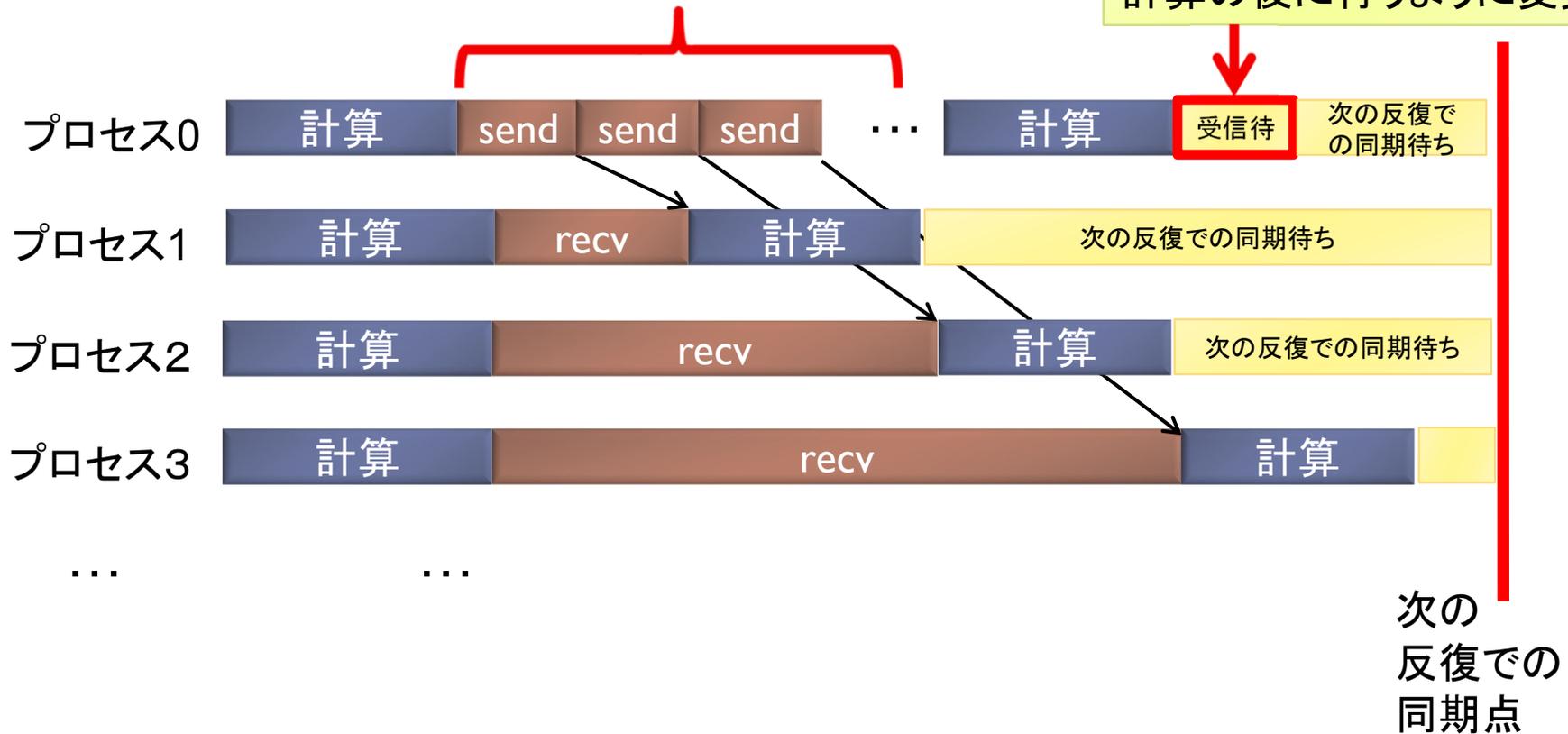
ランク0のプロセスは、
ランク1~numprocs-1までの
プロセスに対するそれぞれの送信
に対し、それぞれが受信完了
するまでビジーウェイト
(スピンウェイト)する。

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

連続するsendにおける受信待ち時間を
ノン・ブロッキング通信で削減

受信待ちを、MPI_Waitで
計算の後に行うように変更



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ー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、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
```

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

/ 以降は、ISENDの例と同じ */*

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] MPIにおけるブロッキングは、必ずしも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。