メニーコア時代のアプリ性能検討 WG

成果報告書

(WG 活動期間:2017 年 1 月-2019 年 6 月)

2019 年 6 月 28 日

サイエンティフィック・システム研究会

メニーコア時代のアプリ性能検討 WG

■商標について

記載されている製品名などの固有名詞は、各社/各機関の商標または登録商標です。

■著作権について

■測定マシンについて

WG では以下の主な環境を利用させていただきました。

・スーパーコンピュータ「京」
・宇宙航空研究開発機構　FX100
・核融合科学研究所　FX100
・九州大学　名大 FX100、FX10
・東京大学 京、SGI
・名古屋大学　FX100、FX10
・原子力研究機構 名大 FX100
・理化学研究所 FX100
・豊田中央研究所 Primergy
・富士通社内機

［謝辞］　当 WG の測定結果は、記載の測定マシンを利用して得られたものです。

メニーコア時代のアプリ性能検討 WG 成果報告書　目次

1 はじめに

1.1 当報告書について

現在執筆中

1.2 活動概要

(1)活動期間

2017年1月～2019年6月

(2)WG メンバー

			氏名	機関(2019 年 6 月 28 日現在)
会員	担当幹事		荻野　正雄	名古屋大学（第9回会合まで）
			松尾　裕一	宇宙航空研究開発機構（第10回会合）
	推進委員	（まとめ役）	井戸村　泰宏	日本原子力研究開発機構
			石田　崇	宇宙航空研究開発機構
			三浦　英昭	核融合科学研究所
			沼波　政倫	核融合科学研究所
			南里　豪志	九州大学
			渡辺　宙志	東京大学　※会員外
			牧野　総一郎	豊田中央研究所
			加藤　由博	豊田中央研究所（第7回会合から）
			梅田　隆行	名古屋大大学
			片桐　孝洋	名古屋大大学
			検崎　博生	理化学研究所
			似鳥　啓吾	理化学研究所
			南　　一生	理化学研究所
賛助会員(富士通)	推進委員	（まとめ役）	軽部　行洋	富士通(株)
			青木　正樹	富士通(株)
			井口　裕次	富士通(株)
			山中　栄次	富士通(株)
			三吉　郁夫	富士通(株)
			志田　直之	富士通(株)
			千葉　修一	富士通(株)
			内藤　俊也	富士通(株)
			小田和　友仁	富士通(株)
			樽水　康平	富士通(株)（第6回会合から）
			渡邊　健太	富士通(株)
			河野　匡伸	富士通(株)（第6回会合から）

(3)活動実績

■第1回会合 ： 2017年1月23日(月)
- WG活動の方向性/活動スケジュールの検討
- 各委員が研究するアプリの紹介(概要)

■第2回会合 ： 2017年4月19日(水)
- 各委員が研究するアプリの紹介(概要、続き)
- 各委員が研究するアプリの紹介(詳細)
- **富士通からの技術紹介**

■第3回会合 ： 2017年7月25日(火)
- 各委員が研究するアプリの紹介
  - 石田（JAXA）/ 南里（九大）/ 渡辺（東大）/ 井戸村（JAEA）
- **富士通からの技術紹介**
  - **SVE（Scalable Vector Extension）の概要**

■第4回会合 ： 2017年10月20日(金)
- 各委員が研究するアプリの紹介
  - 牧野（豊田中研）/ 片桐（名大）/ 檢崎（理研）
- **富士通からの技術紹介**
  - **ARM 評価に向けて**

■第5回会合 ： 2018年1月24日(水)
- 各委員が研究するアプリの紹介
  - 三浦（核融合研）/ 渡辺（東大）/ 伊藤（核融合研)*ゲスト参加
- **富士通からの技術紹介**
  - **GKV評価報告、GT5D評価報告、POST-FX100 NDAベースでの情報開示**

■第6回会合 ： 2018年4月16日(月)
- 各委員が研究するアプリの紹介
  - 井戸村（JAEA）/ 石田（JAXA）/ 梅田（名大）/ 似鳥（理研）
- 富士通からの技術紹介
  - **ソフトウェアパイプライニングの概要、Post-FX100推定**

■第7回会合 ： 2018年7月20日(金)
- 各委員が研究するアプリの紹介
  - 南里（九大）/ 加藤（豊田中研）/ 荻野（名大）
- 富士通からの技術紹介
  - コンパイラによるループ分割機能について
  - Seism3Dのアプリの評価報告
- 期間延長へ向けての議論等

■第8回会合 ： 2018年10月19日(金)
- 各委員が研究するアプリの紹介
  - 石田（JAXA）/ 片桐（名大）
- 富士通からの技術紹介
  - コンパイラ LTO最適化について
  - C/C++ プログライミングガイドについて
  - N体 カーネルの評価報告
- WG報告書作成計画に関しての議論等

■第9回会合 ： 2019年2月25日(月)
- 各委員が研究するアプリの紹介
  - 檢崎（理研） / 三浦（核融合研）
- 富士通からの技術報告（多倍長精度について）
- WG報告書レビュー

■第10回会合 ： 2019年6月3日(月)
- AI最終確認（対応中のAI、完了済みのAIの結論等）
- WG報告書の確認（未実施の報告書レビュー等）

2.1 ループ変換およびスレッド数を動的変更する自動チューニング

名古屋大学情報基盤センター　片桐孝洋

2.1.1 はじめに

近年の計算機はマルチコア化が進み、メモリやキャッシュの構造や、Graphics Processing Unit (GPU) を有する／有しないなど、計算機アーキテクチャが多様化している。そのため、数値計算ソフトウェアが高性能を発揮するためには性能チューニングが重要となる。しかし、計算機環境に合わせたコード最適化はハードウェアの専門知識が必要であり、手間と時間を必要とする。さらに、特定の計算機環境に特化したチューニングを行ったプログラムは、他の計算機環境では性能が低下する可能性がある。そのため、計算機環境に応じ、再びチューニングが必要となるのが一般的である。

近年の計算機アーキテクチャのトレンドを考慮すると、ノード内のコア数が増加している。そのため、並列化対象となるループは、計算機ハードウェア上の並列性を満たすループ長が必要となる。もし並列化対象となるループ長が短い場合は、より長いループ長を持つループを並列化対象にする必要がある。しかし一般的にループ長は、問題サイズ、Message Processing Interface (MPI)、および、OpenMP で実行時に指定するスレッド数に影響し、静的に最適な並列化対象のループを見つけられないことがある。

一方、OpenMP 実行を考えると、実行時に指定するスレッド数が、対象となるループごとに最適なスレッド数ではないことがあり、対象となるループごとにスレッド数を動的に変更させると高速化できる可能性がある。

そこで本報告では、(1)OpenMP の対象ループを変更する自動チューニング方式;(2)OpenMP のスレッド数を実行時に変更する自動チューニング方式；の有効性を評価することを目的にした。

2.1.2 自動チューニング記述言語 ppOpen-AT

ppOpen-AT は、次世代の科学技術アプリケーションの開発・実行環境 ppOpen-HPC の自動チューニング機構として開発された自動チューニング言語である。並列数値計算ライブラリの開発効率を向上させることを目的に設計された FIBER 方式による自動チューニング機能の付加を支援する自動チューニング言語（ディレクティブ）である。

ここでは、ppOpen-AT に実装可能な自動チューニング方式の検討と機能の有効性の評価を行う。

2.1.3 ループ変換の自動チューニング

2 重ループ以上の多重ループを OpenMP でスレッド並列化する場合において、並列化の対象ループを変更するループ変換を ppOpen-AT の拡張機能として考える。

ppOpen-AT では、!oat$ region start ～ !oat$ region end で囲むプログラムを対象ループとする。拡張機能では、OpenMP のディレクティブの移動を伴うコード変換を行う。具体的には、

```
!oat$ install Exchange(対象ループ番号, 対象ループ番号,…) region start
    ～
!oat$ install Exchange(対象ループ番号, 対象ループ番号,…) region end
```

で囲むことで、交換するループを指定する。ただし対象ループには、OpenMP の並列化指定が 1 つのみ記述されていると仮定する。対象ループ番号は、最外ループから数えて何番目のループを OpenMP の並列化対象とするかである。対象ループ番号をコンマで区切り指定することで、複数のチューニング候補を生成可能である.

評価対象のコードは、プラズマ乱流解析コード GKV (p2) のサブルーチン exb_realspcal の 4 重ループ（p3）である。このループを ppOpen-AT の collapse 構文により変換したループが p5 である。

名古屋大学情報基盤センター設置の Fujitsu PRIMEHPC FX100 での結果を p8 に示す。p8 から、z 方向のループ長に応じて、本機能により生成されるループがオリジナルループより高速化されることがわかる。nz=1 のとき最大でオリジナルループの実行時間に対して 6.49 倍高速化される。ただし当該アプリケーションは nz=16 での実行が想定されているのでこの速度向上は参考値となるが、さらなる高速化のために nz 方向に MPI 並列数を増やさないといけない状況では、nz のサイズは上限で nz=1 まで小さくなることがありうる。また、並列数はそのままで、問題サイズを小さくする状況も考えられる。これらの状況においては、本手法は効果を奏すると期待される。

2.1.4 実行時スレッド数変更の自動チューニング

ppOpen-AT による自動チューニングでは、性能パラメタを設定し、プログラムをチューニングする際に性能パラメタを変化させて性能を測定することで最適パラメタを本実行前に探索して決定する。その後、ユーザによるプログラムの実行時に、チューニング済みのパラメタを参照して設定し、最適な候補を実行時に指定して計算を行う。ここで検討する新機能は、性能パラメタとして OpenMP のスレッド数を指定し自動チューニングを行う機能である(p12)。OpenMP のスレッド数変更は、omp_set_num_threads を使用した。

使用するプログラムは、有限差分法による地震波シミュレーションコード Seism3D を原型とする ppOpen-APPL/FDM である(p10、p11)。ppOpen-APPL/FDM のソースコード は ppOpen-HPC のホームページから提供されている。OpenMP と MPI のハイブリッド実行ができる。ppOpen-APPL/FDM は ppOpen-AT によりコード選択の自動チューニング機能を実装しているが、ここでは、デフォルトの実装で性能評価を行う。

スレッド数の動的変更により、変更しない場合比較して FX100 の 8 プロセスで約 1.0014 倍、32 プロセスで約 1.012 倍の高速化が達成できた(p17)。

速度向上の理由を解析するため、当該ループを精密 PA 可視化により、メモリ・キャッシュビジー率(p19)、および、キャッシュミス率(p20)の詳細性能プロファイルを行った。その結果、スレッド変更を行う場合は、メモリ・キャッシュビジー率の向上(L1 ビジー率、32%、

+5 ポイント)、キャッシュミス率（L1D ミス率、3.01%、-0.24 ポイント）と性能改善の理由が確認できた。

2.1.5 まとめ

本報告では、近年の計算機アーキテクチャのトレンドを考慮し、ノード内のコア数が増加していく場合の性能チューニング手法の検討を行った。並列化対象となるループは、計算機ハードウェア上の並列性を満たすループ長が必要とされる。もし並列化対象となるループ長が短い場合は、より長いループ長を持つループを並列化対象にする必要がある。一方、OpenMP での実行を考えると、実行時に指定するスレッド数が対象となるループごとに最適なスレッド数ではなく、対象となるループごとにスレッド数を動的に変更させると高速化できる可能性がある。

そこで本報告では、(1)OpenMP 対象のループを変更する自動チューニング方式;(2)OpenMP のスレッド数を対象ループごとに実行時に変更する自動チューニング方式；の有効性の検討を行った。その結果、双方の機能ともに速度向上に寄与することを確認した。

ポスト京コンピュータでは、計算ノードが 48 コア+4 アシスタントコア構成となり、ノード内のコア数が従来の計算機より増加する。そのため、本報告で示した機能は、現在より多くの局面において、より効果を奏すると期待される。

本機能を自動チューニング言語 ppOpen-AT へ実装することは今後の課題である。また本機能を、コンパイラによるコード最適化へ組み込むこと（ディレクティブの提供を含む）は重要な課題であると著者は考えている。

ループ交換の
自動チューニング

性能評価に使用するプログラム

- GKV code (GyroKinetic Vlasov code) [1)]

  ジャイロ運動論的シミュレーションコード

  核融合炉開発などに利用される、プラズマ乱流のシミュレーションを行うコード

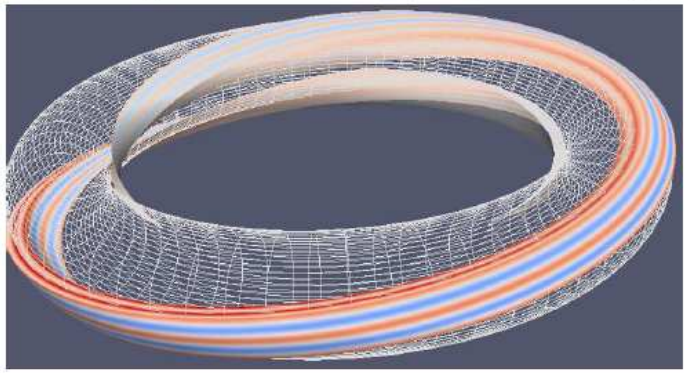
Delta-f Gyrokinetic eqs.

$$\frac{\partial \tilde{f}_s}{\partial t} + \left(v_\parallel \frac{\boldsymbol{B} + \tilde{\boldsymbol{B}}_\perp}{B} + \boldsymbol{v}_{sG} + \boldsymbol{v}_{sC} + \tilde{\boldsymbol{v}}_E \right) \cdot \nabla \tilde{f}_s + \frac{dv_\parallel}{dt} \frac{\partial \tilde{f}_s}{\partial v_\parallel} = S_s + C_s,$$

$$\nabla_\perp^2 \tilde{\phi} = -\frac{1}{\varepsilon_0} \sum_s e_s (\tilde{n}_s + \tilde{n}_{s,\mathrm{pol}}),$$

$$\nabla_\perp^2 \tilde{A}_\parallel = -\mu_0 \sum_s e_s \tilde{u}_{\parallel s},$$

Example of flux-tube geometry



1) Watanabe, T-H., and H. Sugama., “Velocity-space structures of distribution function in toroidal ion temperature gradient turbulence.,” Nuclear Fusion 46.1, (2005):24.

評価対象とするループ

- **サブルーチン**exb_realspcal　(**オリジナルループ**)

```
do iv = 1, 2*nv
 !$OMP parallel do private(mx,my)
 do iz = -nz, nz-1
  do mx = ist_xw, iend_xw
   do my = 0, nyw
       wkdf1_xw(my,mx,iz,iv) = cmplx(real(wkdf1_xw(my,mx,iz,iv),kind=DP) &
          * real(wkeyw_xw(my,mx,iz)-cs1 *vl(iv) &
          * wkbyw_xw(my,mx,iz),kind=DP )-real(wkdf2_xw(my,mx,iz,iv),kind=DP) &
          * real(wkexw_xw(my,mx,iz)-cs1*vl(iv)*wkbxw_xw(my,mx,iz),kind=DP ) &
         , aimag(wkdf1_xw(my,mx,iz,iv))*aimag(wkeyw_xw(my,mx,iz)-cs1*vl(iv) &
          * wkbyw_xw(my,mx,iz))-aimag(wkdf2_xw(my,mx,iz,iv)) &
          * aimag(wkexw_xw(my,mx,iz)-cs1*vl(iv)*wkbxw_xw(my,mx,iz)) &
         , kind=DP ) * cef
   end do
  end do
 end do
 !$OMP end parallel do
end do
```

評価方法

- それぞれのループの実行時間を計測して比較を行った
  - オリジナルループ (ループ1)
  - ppOpen-ATによるループcollapseで自動生成したループ (ループ2、3)
  - 提案手法によりppOpen-ATで自動生成されると想定されるコードを、手動で作成したループ (ループ4、5、6)

- 実験の条件
  - 行列サイズnv=8 、ist_xw=0、iend_xw=127、nyw=64は固定する
  - 行列サイズnzを1から8に変化させる
  - OpenMP並列数を1から最大まで変える
  - ループを1000回繰り返し実行する

性能評価を行うループ(ループcollapse)

```
do iv = 1, 2*nv
!oat$ install LoopFusion region start
!$OMP parallel do private(mx, my)
 do iz = (-nz), nz-1
  do mx = ist_xw, iend_xw
   do my = 0, nyw
    ! 省略
   end do
  end do
 end do
!$OMP end parallel do
!oat$ install LoopFusion region end
end do
```

- ppOpen-ATの機能によりループcollapseを行い、2つのループを生成
- 右のループはループの上半分のみ

ループ2 (ppOpen-AT の ループcollapse)

```
do iv = 1, 2*nv
!$OMP parallel do private(mx,my,mx_my)
 do iz = (-nz), nz-1
  do mx_my = 1 , (iend_xw-ist_xw+1)*(nyw-0+1)
   mx = mod((mx_my-1)/(nyw-0+1),(iend_xw-ist_xw+1)) + ist_xw
   my = mod((mx_my-1),(nyw-0+1)) + 0
```

ループ3 (ppOpen-AT の ループcollapse)

```
do iv = 1, 2*nv
!$OMP parallel do private(mx,my,iz)
 do iz_mx_my = 1 , (nz-1-(-nz)+1)*(iend_xw-ist_xw+1)*(nyw-0+1)
  iz = (iz_mx_my-1)/((iend_xw-ist_xw+1)*(nyw-0+1)) + (-nz)
  mx = mod((iz_mx_my-1)/(nyw-0+1),(iend_xw-ist_xw+1)) + ist_xw
  my = mod((iz_mx_my-1),(nyw-0+1)) + 0
```

性能評価を行うループ

```
!oat$ install Exchange(1, 3, 4)region start
do iv = 1, 2*nv
!$OMP parallel do private(mx, my)
  do iz = (-nz), nz-1
    do mx = ist_xw, iend_xw
      do my = 0, nyw
        ! 省略
      end do
    end do
  end do
!$OMP end parallel do
end do
!oat$ install Exchange(1, 3, 4)region end
```

提案するディレクティブの記述

ppOpen-ATにより自動生成されると想定されるコード

ループ4 (提案手法)

```
!$OMP parallel do private(mx,my,iz)
do iv = 1, 2*nv
  do iz = (-nz), nz-1
    do mx = ist_xw, iend_xw
      do my = 0, nyw
```

ループ5 (提案手法)

```
do iv = 1, 2*nv
  do iz = (-nz), nz-1
!$OMP parallel do private(my)
    do mx = ist_xw, iend_xw
      do my = 0, nyw
```

ループ6 (提案手法)

```
do iv = 1, 2*nv
  do iz = (-nz), nz-1
    do mx = ist_xw, iend_xw
!$OMP parallel do
      do my = 0, nyw
```

実験環境

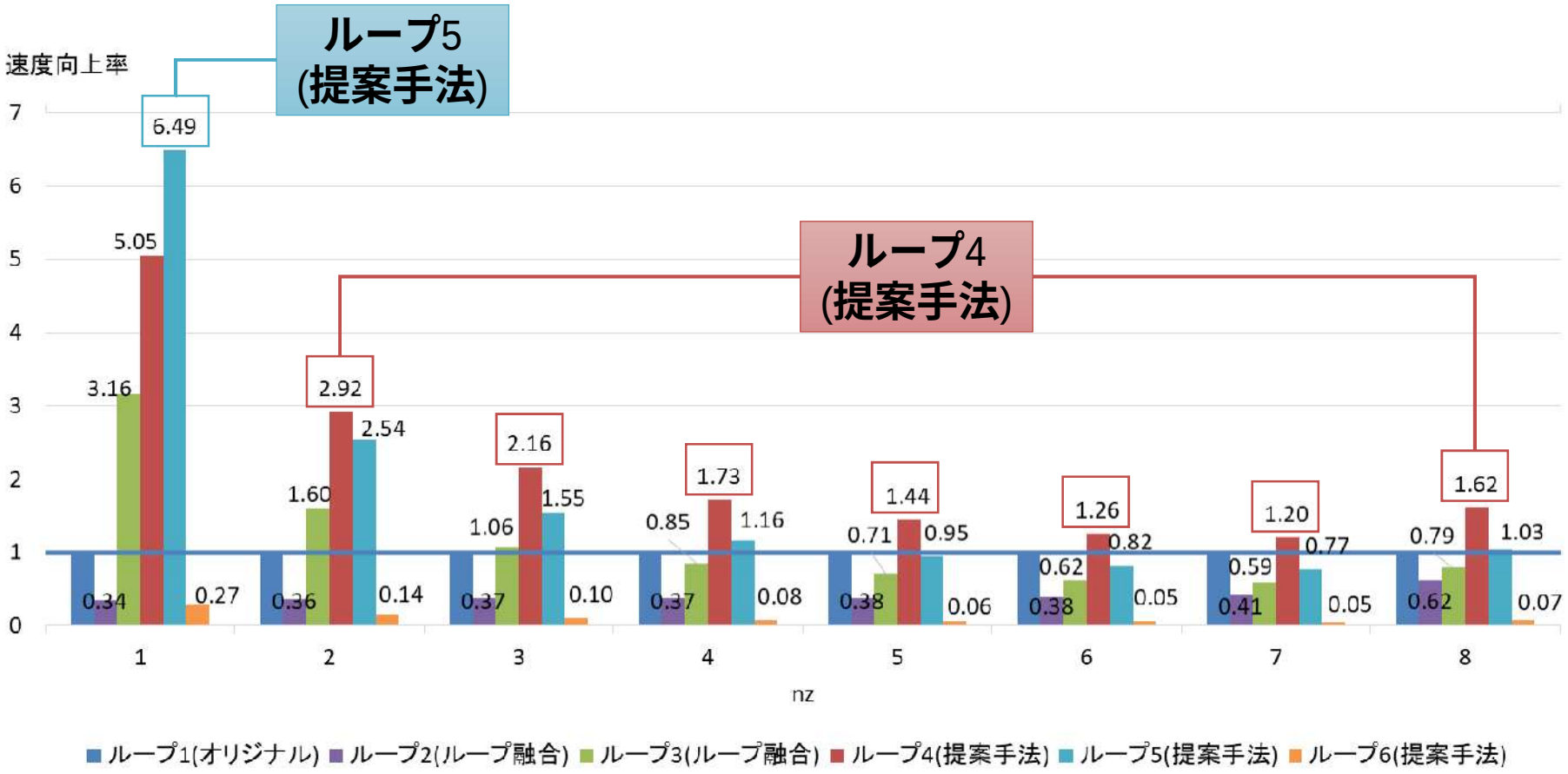
(プロセッサ、コア、スレッド数は1ノードあたり)

	Fujitsu PRIMEHPC FX100 [1)] (以降, FX100)
計算ノード	Fujitsu PRIMEHPC FX100
プロセッサ名	Fujitsu SPARC64 XIfx
プロセッサ数 (コア数)	1 (32)
最大スレッド数	32
コンパイラ	frtpx: Fujitsu Fortran Driver Version 2.0.0
コンパイルオプション	-Kfast -Qt -Cpp -X9 -fs -fw -Kopenmp

1) 名古屋大学情報基盤センター設置

実験結果 (FX100、1ノード32コア)

最速となるスレッド数での実行



実行時のスレッド数変更の自動チューニング

使用するソフトウェア

ppOpen-APPL/FDM

- **並列プログラム作成のためのライブラリ群**
  ppOpen-APPL**のうち、差分法を対象にしたライブラリ**

- http://ppopenhpc.cc.u-tokyo.ac.jp/ppopenhpc/downloads/ **から**ppOpen-FDM var 0.3.1**と**
  ppOpen-APPL/FDM-AT ver.1.0.0**がダウンロード可能**

- **地震動のシミュレーションを行う** Seism3D **のプログラムが原型**

Seism3D[1)]

- 地震動シミュレーションコード
- 差分法計算により陽的に地震波動の伝搬を計算するソフトウェア
- 均質でない構造の地下を伝わる地震波を計算し、地面の揺れを求める
- 地震と津波を連動して解くことができる

www.nsc.riken.jp/target-application/Seism3D-jp.pdf

1) F. Mori, M. Matsumoto, T. Furumura,
"Performance of FDM Simulation of Seismic Wave Propagation using the ppOpen-APPL / FDM Library on the Intel Xeon Phi Coprocessor", Springer LNCS (2014).

本実験でのスレッド数変更方法

oat_max_threads = omp_get_max_threads()
で実行時のスレッドの値を保存しておく

ppOpen-ATにより自動生成されると想定されるコード

変更前の処理

外部の情報からNを決定する
omp_set_num_threads (N)

対象の処理

omp_set_num_threads (oat_max_threads)

変更後の処理

予備実験

- ppOpen-APPL/FDMのカーネル update_stress に対してOpenMPスレッド数の変更を行い、効果を調査
- 計算環境は名古屋大学情報基盤センター設置のPRIMEHPC FX100 (各ノード32コア)
- 実験には8ノードを使用
- プロセス数とスレッド数の組み合わせを変える
- 2000ステップ計算する

実験環境

(プロセッサ、コア、スレッド数は1ノードあたり)

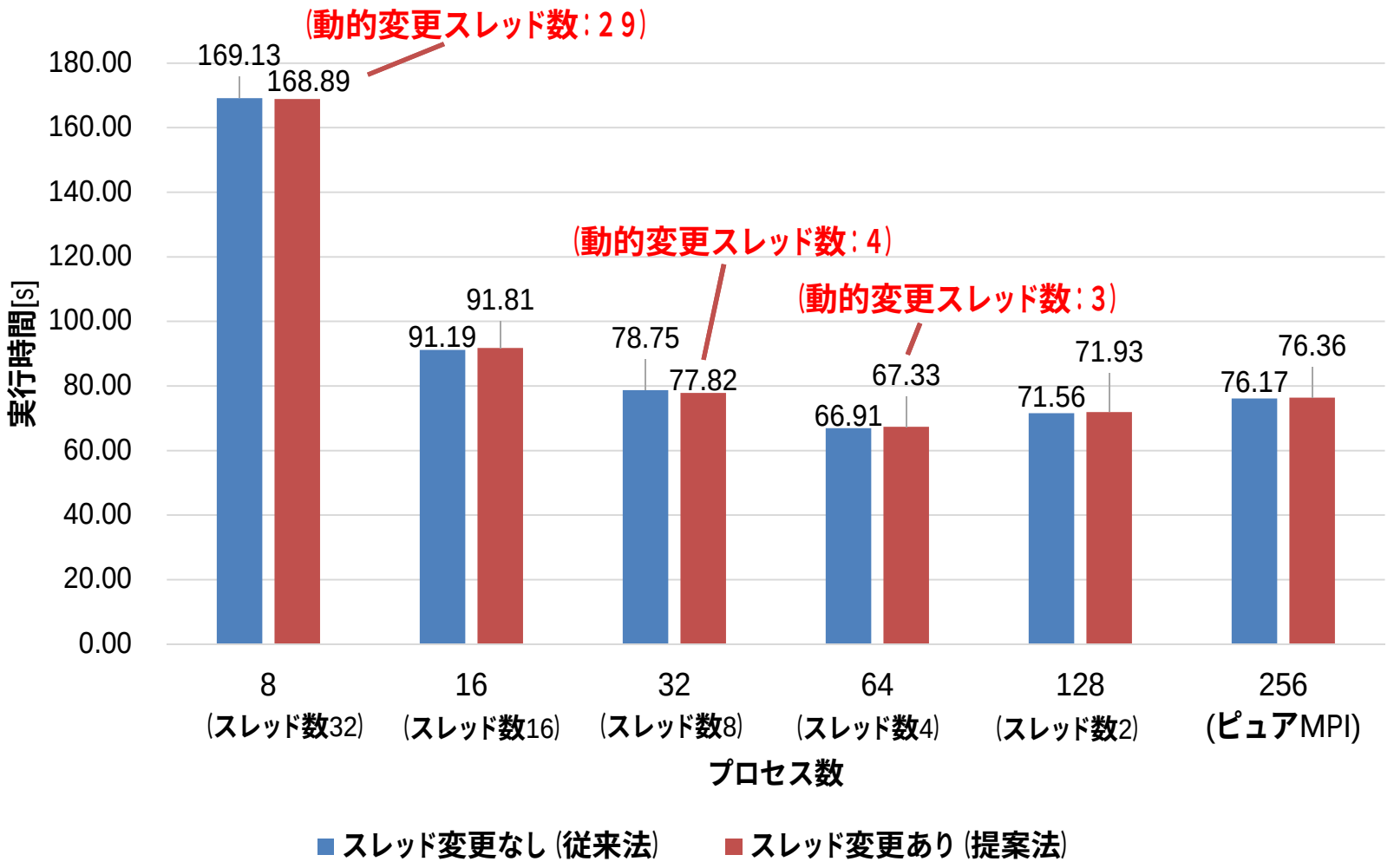
	Fujitsu PRIMEHPC FX100 [1)]
計算ノード	Fujitsu PRIMEHPC FX100
プロセッサ名	Fujitsu SPARC64 XIfx
プロセッサ数 (コア数)	1 (32)
最大スレッド数	32
コンパイラ	mpifrtpx : Fujitsu Fortran Driver Version 2.0.0
コンパイルオプション	-Kfast,openmp

1) 名古屋大学情報基盤センター設置

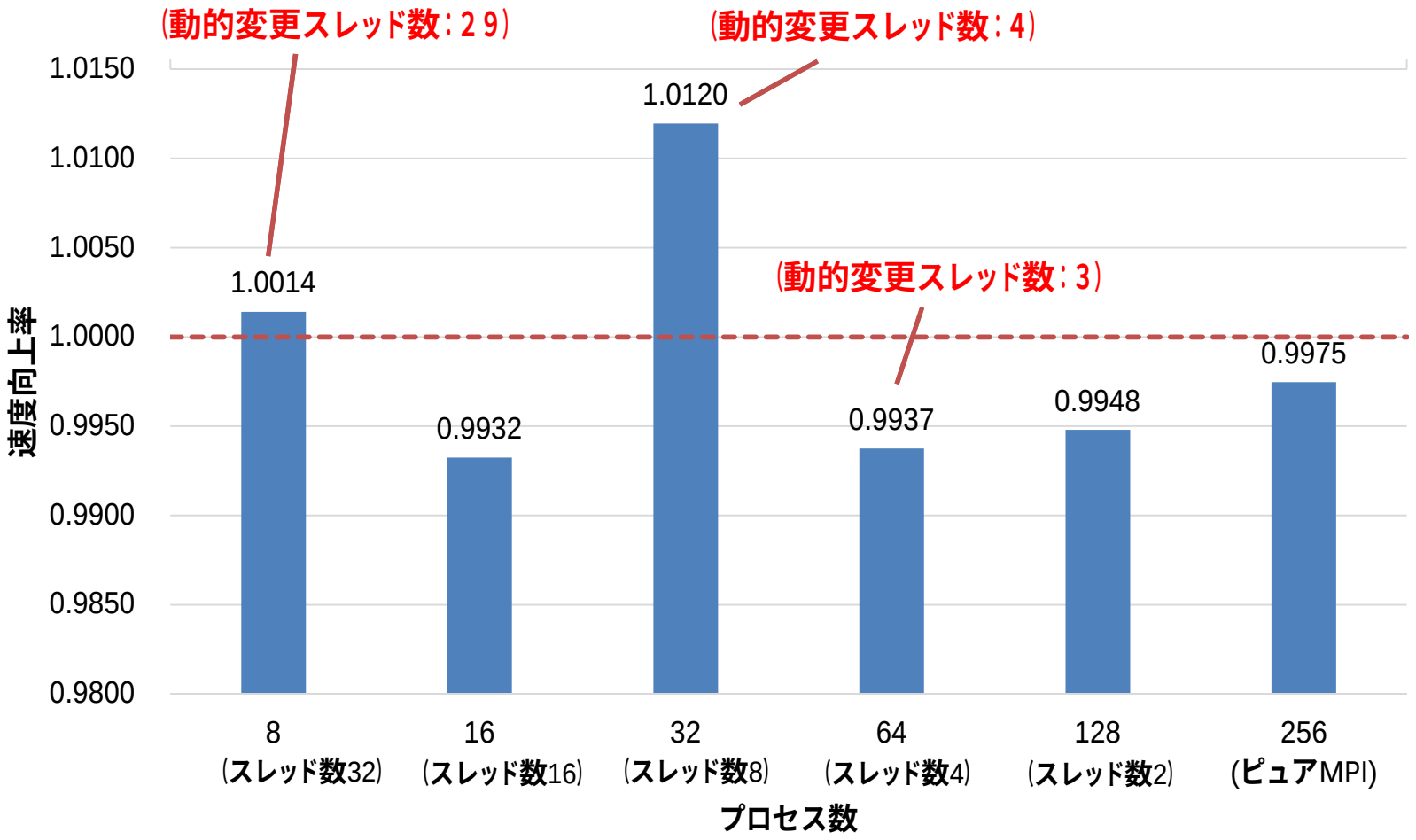
評価方法

- スレッド数の変更をしない場合とする場合で全体実行時間を計測して比較を行った
  - スレッド変更なしは各プロセス数で最大のスレッド数を使用
  - スレッド変更ありは各プロセス数で最大のスレッド数から、そのプロセス数で使用できるすべてのスレッドに変更し，それぞれの時間を計測
- スレッド数変更の場合の環境変数
  - export OAT_EXEC=0
  - export FLIB_FASTOMP=FALSE
  - export FLIB_CTRL_BARRIER_ERR=FALSE

実験結果（FX100、8ノード）



実験結果（FX100、8ノード）



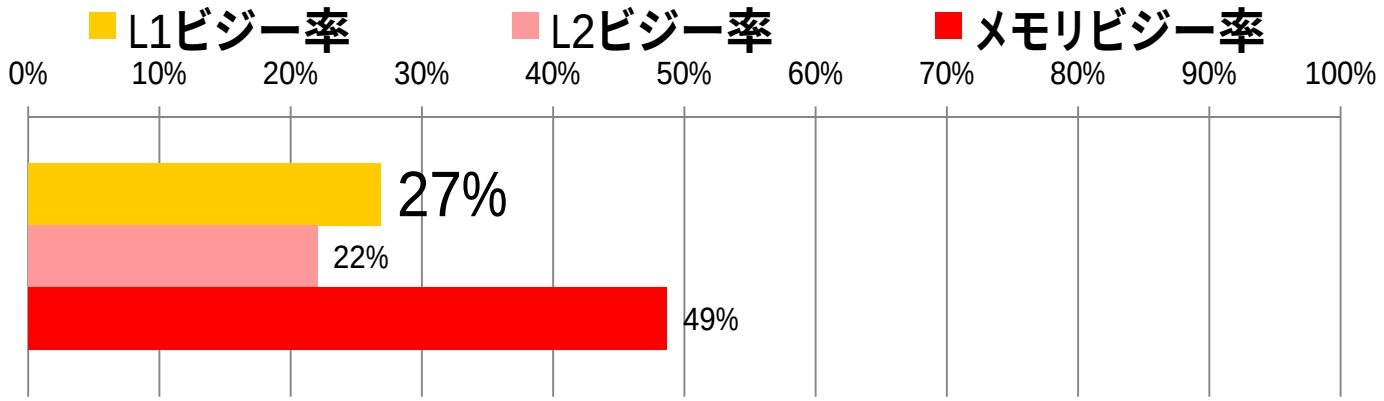
詳細プロファイラ (精密PA)

- FX100の精密PA可視化機能を使用

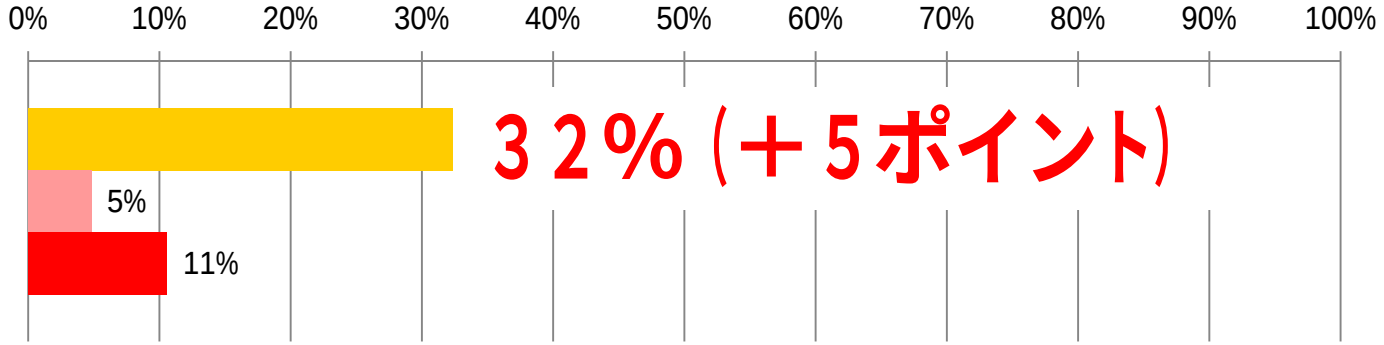
- 8プロセスの場合の結果を比較
- update_stressがプロファイル対象
- スレッド変更ありの結果は、対象カーネルを全てのスレッド数に変えて試した結果，最速となった27スレッドで実行したケースを使用

メモリ・キャッシュビジー率

- スレッド変更なし（従来法）：３２スレッド固定

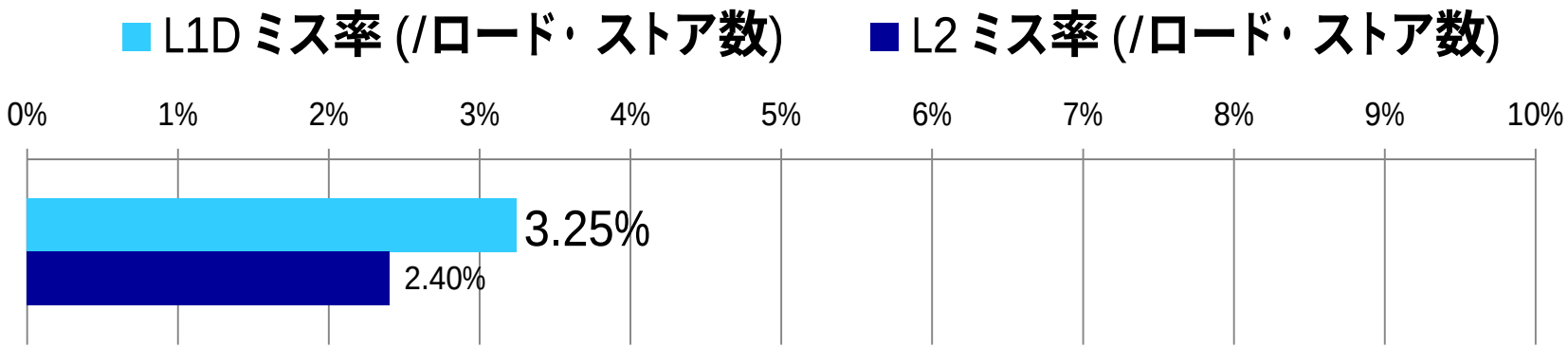


- スレッド変更あり（提案法）
  ：当該カーネルのみ２７スレッド実行

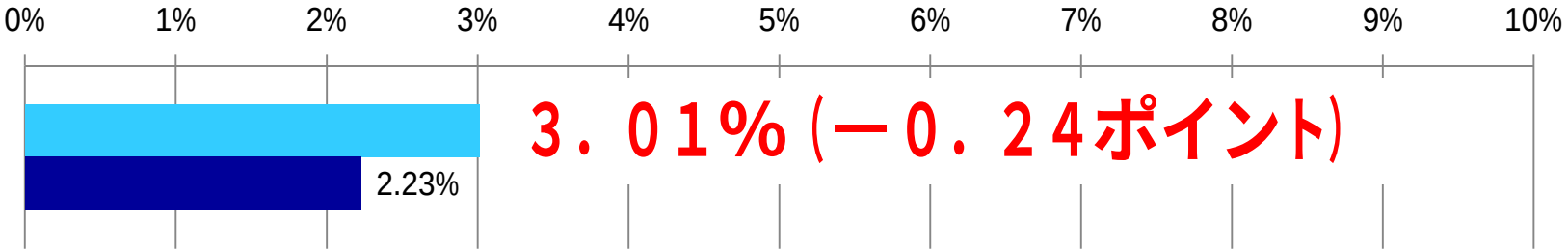


キャッシュミス率

- スレッド変更なし (従来法) ：３２スレッド固定



- スレッド変更あり (提案法)
  ：当該カーネルのみ２７スレッド実行



2.2 非ブロッキング集団通信の通信隠蔽効果に関する調査

九州大学　南里　豪志

2.2.1 はじめに

プロセス並列プログラムにおける通信の標準規格である Message Passing Interface (MPI) で定義されている非ブロッキング通信は、あるプロセス間の通信と、それらのプロセス上での計算を並行して進行させることにより、見かけ上、通信に要する時間の一部を計算時間で隠蔽することを可能とする。このうち、MPI-3.0 規格において採用された非ブロッキング集団通信は、多くの並列プログラムで頻出する、複数のプロセス間でのデータ複製や集約のような定型通信である集団通信について、通信時間の隠ぺいを図るものである。集団通信は、計算機の大規模化に伴って所要時間が増大するため、並列プログラムのスケーラビリティ向上の手段として、この非ブロッキング集団通信が期待されている。

非ブロッキング集団通信は、その集団通信に参加する各プロセスが計算を行っている間に、集団通信を完了させるための通信アルゴリズムを推進することで、通信時間を隠蔽するため、この、集団通信アルゴリズムの推進手法が、隠蔽効果に大きく影響する。現在、多くの MPI ライブラリで採用されている集団通信アルゴリズム推進手法のうち、特にプログレススレッドを用いた推進手法は、プログラム中に MPI_Test 関数のようなアルゴリズム推進のための MPI 関数を挿入しなくても高い通信隠蔽率が期待できる上、インターコネクトネットワークに特殊なハードウェアを必要としないという特徴があるため、特に注目されている。一方、この手法は、プロセス毎に集団通信アルゴリズム推進専用のスレッドに CPU コアを割り当てる必要があるため、その分、計算性能が低下する。特に、スレッド並列とプロセス並列によるハイブリッド並列プログラムを利用する場合や、一つのノードの中で複数のプロセスを動作させる場合、この推進手法による非ブロッキング集団通信の効果は、計算性能低下による計算時間の増加と、通信隠蔽による通信時間の減少のトレードオフとなる。しかし、Intel MPI Benchmarks や OSU Micro-Benchmarks のような、一般に用いられているベンチマークプログラムにおける非ブロッキング集団通信の評価では、通信隠蔽率のみを計測するため、この推進手法の実用性を十分に検証することが出来ない。

そこで本報告では、ハイブリッド並列計算と集団通信を並行して実行する簡単なベンチマークプログラムを作成し、それを用いて、プログレススレッドによる非ブロッキング集団通信の実用性を評価する。評価対象の MPI ライブラリとしては、PC クラスタ上でプログレススレッドによる通信隠蔽を行う MVAPICH2-2.2 と Intel MPI 2017、さらに、Fujitsu PRIMEHPC FX100 でプログレススレッド専用のアシスタントコアを使用する富士通社製 MPI を用いる。また、Mellanox 社の SHARP と呼ばれるオフロード機能についても、予備評価を行う。

2.2.2 非ブロッキング集団通信インタフェース

MPI-3.0 規格において採用された非ブロッキング集団通信インタフェースは、従来の一対一通信における非ブロッキング通信と同様、通信の開始と完了待ちをそれぞれ別の関数と

することで、通信完了を待つ間に、その通信と並行して処理可能な計算や通信の実行を可能とするものである。通信開始関数としては、MPI_Iallreduce 関数や、MPI_Ialltoall 関数等、従来のブロッキング集団通信関数名に I を追加したものが用意されている。これらの関数は、完了を待つための情報を、MPI_Request 型のリクエスト変数に格納する。この変数は、一対一の非ブロッキング通信と同じ MPI_Wait 関数や MPI_Waitall 関数等に指定することで、完了を待つことが出来る。これらの関数を用いて通信の開始と完了待ちを指示し、さらにその通信と関係のない処理を開始と完了待ちの間に挿入することで、その通信の時間の一部を他の処理で隠蔽することが期待できる。
現在、MPICH、MVAPICH2、Open MPI といった主要なオープンソースの MPI ライブラリの他、Intel MPI Library や富士通社製 MPI 等の非オープンソースの MPI ライブラリの多くで、この非ブロッキング集団通信インタフェースが提供されている。

2.2.3 非ブロッキング集団通信の実装技術

非ブロッキング集団通信による通信隠蔽の効果は、MPI ライブラリでの実装手段に大きく影響を受ける。特に、集団通信アルゴリズムを他の処理と並行して進行させるためのアルゴリズム推進の実装手段は、通信隠蔽効果への影響が大きい。この、アルゴリズム推進手法は、集団通信アルゴリズムを構成する通信関数やメモリコピー、計算等の処理を、集団通信アルゴリズムで定義された依存関係に従って順に発行する。
現在、MPI ライブラリで採用されているアルゴリズム推進手法としては、主に以下の三通りが挙げられる。

- MPI 関数呼び出し毎の推進
- インターコネクトのオフロード機能による推進
- プログレススレッドによる推進

このうち、MPI 関数呼び出し毎の推進とは、全ての MPI 関数内で、非ブロッキング集団通信を推進させるための推進ルーチンを実行するものである。この推進ルーチンは、その時点で進行中の全ての非ブロッキング集団通信について、アルゴリズムの依存関係に基づき、実行可能な命令を発行する。この推進手法を利用する場合、プログラマは、非ブロッキング集団通信の開始関数と完了待ち関数の間に、例えば MPI_Test 関数のように、プログラムの意味を変えない MPI 関数をプログラム中に挿入することで、完了待ちの前にアルゴリズムを進行させておくことが出来る。この推進手法による性能向上の効果は、通信隠蔽による通信時間の削減と、推進ルーチンのためだけに呼び出す MPI 関数のオーバヘッドのトレードオフとなり、十分な効果が得られるか否かは、プログラムの構造やプログラマの能力に依存する。
一方、インターコネクトのオフロード機能による推進は、インターコネクトの Network Interface Card (NIC) やスイッチ等に集団通信のアルゴリズムを進行させるオフロード機能が用意されている場合に、非ブロッキング集団通信関数の内部でその機能を呼び出すものである。例えば Mellanox 社のインターコネクト技術である InfiniBand は、集団通信アルゴリズムを NIC で処理する CORE-Direct 機能や、スイッチで処理する SHARP 機能を備え

ている。また、これらの機能を非ブロッキング集団通信の内部で呼び出すよう実装されている MPI ライブラリとしては、MVAPICH2 や Mellanox 社の HPC-X 、等がある。基本的に、オフロード機能を有するインターコネクトが利用できる場合、効果的に通信を隠蔽することが出来る。ただし、この機能は、NIC やスイッチのハードウェア上の制約により、使用できる集団通信関数やメッセージサイズ等が制限されている場合が多い。
これらに対して プログレススレッドによる推進は、MPI プログラムを進行させるスレッドとは別に、非ブロッキング集団通信アルゴリズムを進行させるためだけのスレッドを生成する。この推進手法は、使用するインターコネクトがオフロード機能を有しない場合や、プログラム中に適切に MPI_Test 関数等を挿入することが困難な場合でも、集団通信と他の処理を同時に進行させることが出来るため、容易に通信を隠蔽する手段として注目されている。そのため、代表的なオープンソースの MPI ライブラリである Open MPI、MPICH、MVAPICH2 のいずれも、プログレススレッドによる非ブロッキング集団通信の推進を選択可能となっている。このうち Open MPI では、Hoefler らが開発した非ブロッキング集団通信ライブラリ LibNBC をコンポーネントとして追加することにより、この推進手法を実装している。この推進手法は、メインスレッドが、ハンドルキューと呼ぶ待ち行列を介して、集団通信アルゴリズムを構成する処理をタスク単位でプログレススレッドに渡すことにより、アルゴリズムを推進させている。一方、MPICH および MVAPICH2 は、同様の待ち行列を持つ推進手法を独自に開発し、実装している。

2.2.4 アシスタントコア機能

非ブロッキング集団通信の使用にあたって利用者が選択すべき事項のうち、性能に与える影響が大きいものとしては、前節までで紹介したアルゴリズム推進手法の他に、プログレススレッドへの CPU コアの割り当て方法が挙げられる。Fujitsu PRIMEHPC FX100 では、アシスタントコアと呼ばれる CPU コアをプログレススレッドに割り当てることにより、計算用の CPU コアを全て計算に使用することが出来る。この機能は、FX100 向けの富士通 MPI ライブラリで、mpiexec コマンドに以下のオプションを追加することにより、利用できる。

```
--mca opal_progress_thread_mode モード番号
```

ここでモード番号とは、プログレススレッドの動作モードを指定するもので、以下から選択する。

- mode 1： 指定した区間のみプログレススレッドが動作する。区間内では MPI 関数を呼び出すことが出来ない。
- mode 2： 指定した区間のみプログレススレッドが動作する。区間内で MPI 関数を呼び出すことが出来る。
- mode 3： 非ブロッキング集団通信が呼ばれると自動的にプログレススレッドが動作する。

mode 1 および mode 2 の場合、MPI_Ialltoall と MPI_Wait に挟まれた計算部分を、以下のようにプログレススレッドの動作区間として指定することで、アシスタントコアを利用した通信隠蔽を図る。

```
MPI_Ialltoall(M);
FJMPI_Progress_start();
do_comp();
FJMPI_Progress_stop();
MPI_Wait();
```

一方、アシスタントコアを持たない一般の PC クラスタなどの環境でプログレススレッドによるアルゴリズム推進を行う場合、Spare Core もしくは Fully Subscribed と呼ばれる方法で、CPU コアにプログレススレッドを割り当てる。

このうち Spare Core とは、プログレススレッドに一つの CPU コアを占有させるもので、高い通信隠蔽効果を期待できる。しかし、計算に割り当てる CPU コアが 1 プロセス当たり 1 コアずつ削減されるため、特に MPI のみによる並列プログラムや、ハイブリッド並列でプロセスあたりのスレッド数が少ない場合、計算性能の低下による影響が無視できなくなると予想される。

一方 Fully Subscribed は、計算用のスレッドに全ての CPU コアを割り当てておき、プログレススレッドにはどれかの計算用スレッドと CPU コアを共有させるものである。これは、計算スレッドの空き時間に集団通信アルゴリズムを進行させることを期待するもので、Spare Core と比べると、計算性能の低下による影響は少ないと期待できる。しかし、スレッド切り替えのオーバヘッドや、進行状況を確認する頻度の低下により、通信隠蔽効果は低下すると予想される。

なお、Intel MPI Library には、プログレススレッドに割り当てる CPU コアの番号を固定する機能が用意されている。これにより、ノード内のプロセス数が増えても、プログレススレッドに割り当てられる CPU コア数が増加せず、計算性能の低下を軽減する効果が期待できる。

2.2.5 非ブロッキング集団通信性能評価用ベンチマークプログラム

非ブロッキング集団通信を対象としたベンチマークプログラムとして一般的に用いられているのは、オハイオ州立大学で開発されている OSU Micro Benchmarks や、Intel 社で開発されている Intel MPI Benchmarks 等に含まれるプログラムである。これらのプログラムは、通信を隠蔽できた比率や、隠蔽できなかった通信オーバヘッドの計測を主な目的としている。そのため、通信と並行して実行する計算プログラムでは、予め計測していた通信時間を用いて、通信が行われている間に計算が終了しないよう、計算量を制御している。この手法は、通信隠蔽率の評価には適しているものの、以下の点から、この手法だけで実プログラムにおける非ブロッキング集団通信の効果を見積もることは困難である。

- 通信時間の実測値に応じて計測毎に計算量を自動調節するため、プログレススレッド

による計算性能低下の影響を評価できない。

- 計算部分がスレッド並列化されていないため、プログレススレッドの CPU コアへの割り当て方法による性能の変化を評価できない。
- 非ブロッキング集団通信を使わなかった場合の性能が計測されていないため、使用の有無によるプログラムの性能の変動を比較できない。

そこで本稿では、以下に示す簡単なプログラムにより、ハイブリッド並列プログラムにおける非ブロッキング集団通信の効果を計測する。

```
  nn = N / procs;
  MPI_Ialltoall(M);
#pragma omp parallel for private(j,k)
  for (i = 0; i < nn; i++)
    for (k = 0; k < N; k++)
      for (j = 0; j < N; j++)
        c[i*N+j] += a[i*N+k] * b[k*N+j];
  MPI_Wait();
```

このプログラムでは、通信量 M と計算量 N を実行時に指定する。これにより、非ブロッキング集団通信使用の有無による並列プログラムの性能への影響の見積もりが可能となる。

2.2.6 非ブロッキング集団通信による通信隠蔽効果の計測結果

2.2.6.1 アシスタントコアによる通信隠蔽効果

Fujitsu PRIMEHPC FX100 のアシスタントコアを用いた通信隠蔽効果は、添付のスライド１に示すとおりである。今回試した MPI_Iallreduce、MPI_Ialltoall とも、ほとんどの場合で、通信隠蔽の効果を確認できた。例外は、ノード内プロセス数を増やした場合の MPI_Iallreduce、およびメッセージサイズが小さい場合の MPI_Ialltoall であった。前者は、集約の計算量、後者は、メッセージ毎に発生するオーバヘッドが相対的に増えたことにより、アシスタントコアでの所要時間が計算時間を上回り、通信を隠蔽できなくなった、と考えられる。

2.2.6.2 スイッチへのオフロード機能による通信隠蔽効果（参考）

参考のため、Mellanox 社におけるスイッチへのオフロード機能 SHARP による MPI_Iallreduce の通信隠蔽効果を計測した。結果は添付のスライド２に示すとおりである。今回試した実験のうち、本報告で作成した計測プログラムでは、メッセージサイズが小さく、ノード内プロセス数が少ない場合に、通信隠蔽効果が確認できた。一方、PETSc の PIPECG アルゴリズムでは、ノード内プロセス数が多い場合に、SHARP による高速化が見られる、という、矛盾した結果となっている。そのため、このオフロード機能による通信隠蔽については、より詳細な解析が必要である。

2.2.7 むすび

本報告では、非ブロッキング集団通信による通信隠蔽効果を調査した。従来の非ブロッキング集団通信ベンチマークプログラムでは、通信隠蔽に伴う計算時間の変動を考慮した評価が行えなかったため、計算量を固定したハイブリッド並列ループによる通信隠蔽ベンチマークプログラムを作成した。

計測の結果、非ブロッキング集団通信をプログレススレッドで推進する場合、Fujitsu PRIMEHPC FX100 のアシスタントコア機能が有効に働き、高い通信隠蔽効果が得られることを確認した。一方、非ブロッキング集団通信の推進をスイッチにオフロードする Mellanox 社の SHARP では、通信隠蔽の効果が正しく得られているかどうか、現時点では判断できず、より詳細な解析が必要であることが分かった。

非ブロッキング集団通信の通信隠蔽効果に関する調査

SS研 メニーコアWG
南里 豪志 (九州大学)

研究の目的

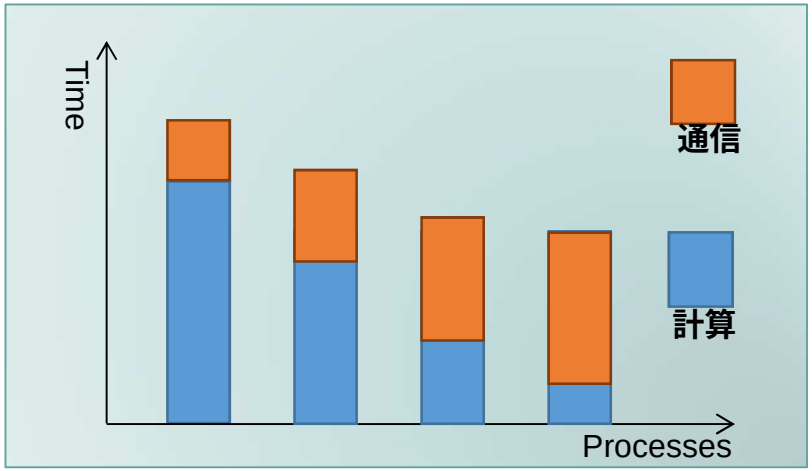
- **非ブロッキング集団通信による通信隠蔽について、以下の技術による効果を検証**

- **検証1）** Fujitsu PRIMEHPC FX100**のアシスタントコア**

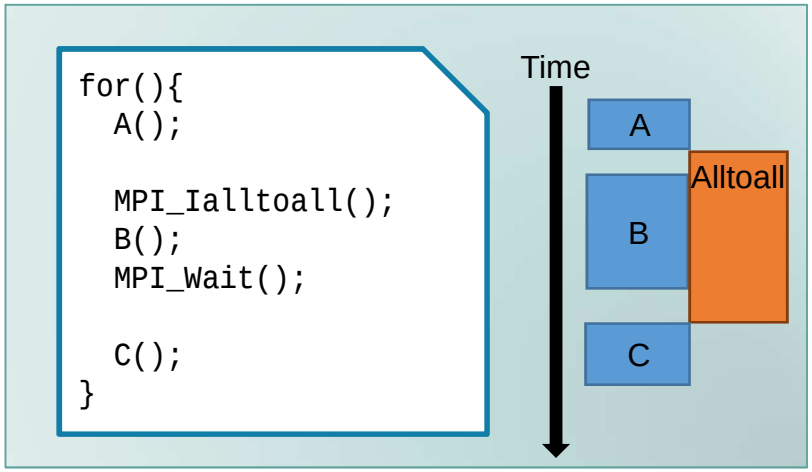
- **検証2）** Mellanox**社** SHARP**によるオフローディング**

背景： 非ブロッキング集団通信

- 集団通信の所要時間
  - プロセス数に応じて増
  - プログラム全体の並列化効果低減
- 非ブロッキング集団通信への期待：
  通信隠蔽による並列化効果向上
- でも、多くのプログラムではアルゴリズムの変更が必要
  - 通信と並行して行える計算の抽出
- 非ブロッキング集団通信が使い物になるか否か、予め知りたい

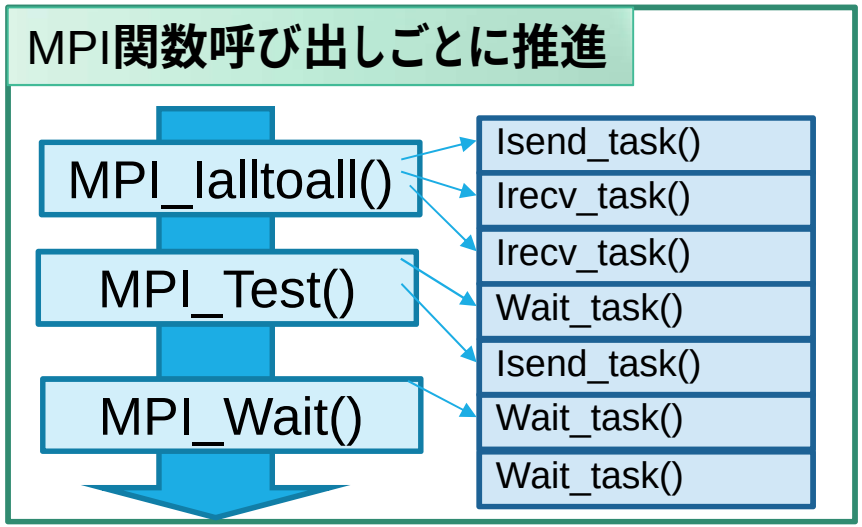


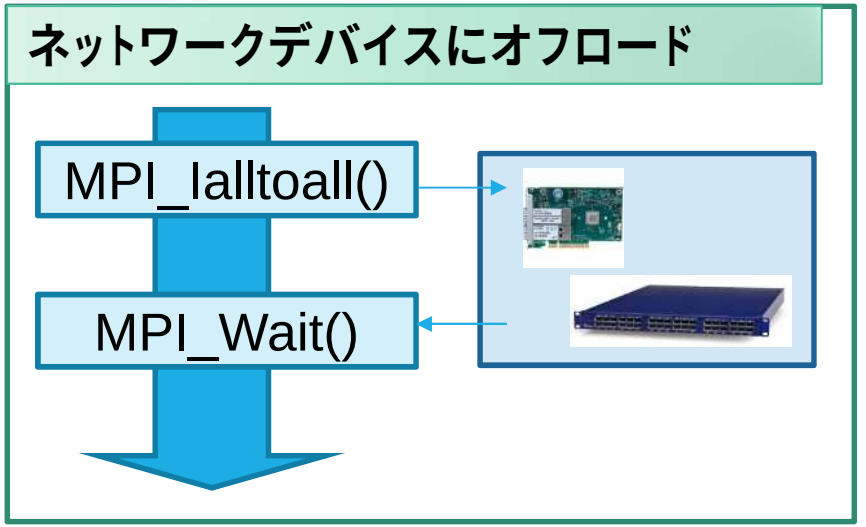
プロセス数の増加に伴う並列プログラムの実行時間の変化

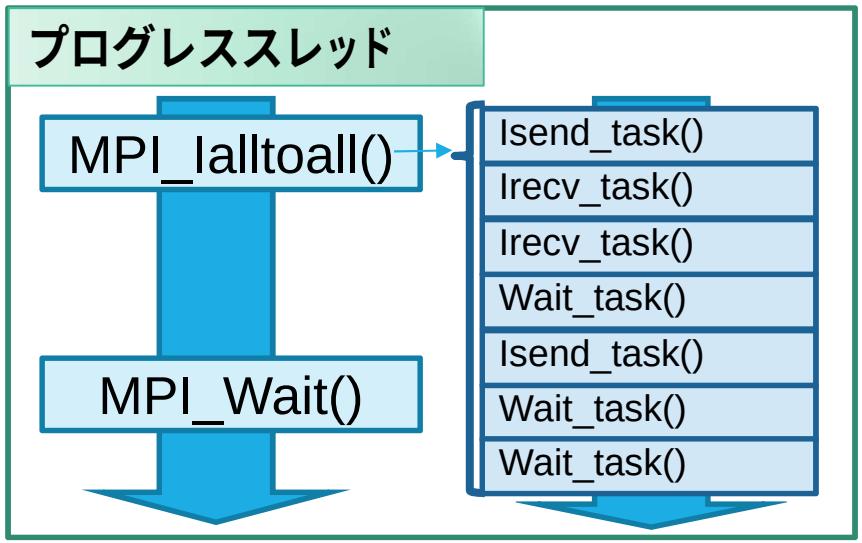


非ブロッキング集団通信による通信隠蔽

非ブロッキング集団通信のアルゴリズムをどうやって計算と並行して進めるか？







アルゴリズム推進手段の比較

	通信隠蔽効果	使いやすさ	環境依存性	全体性能
MPI関数毎に推進	Medium	Bad	Good	Medium
オフロード	High	Good	Bad	???
プログレススレッド	High	Good	Good	???

検証1） アシスタントコアによるプログレススレッドの効果

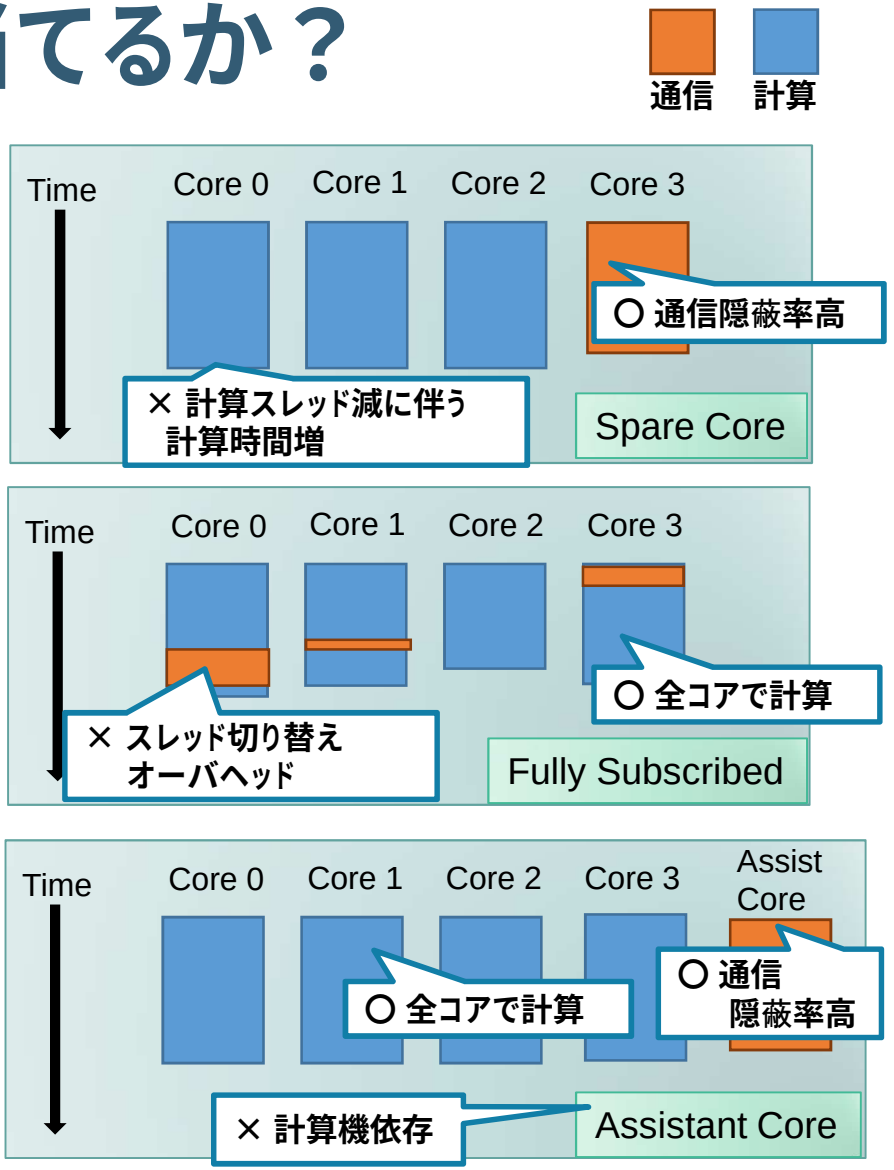
検証2） スイッチへのオフロードの効果

プログレススレッドの利用手段：コアにどのように割り当てるか？

- Spare Core
  - **プログレススレッド用に1コアを留保**
- Fully Subscribed
  - **計算スレッドとコア共有**

検証1）

- Assistant Core
  - **計算機が提供する専用コアを使用**



検証1） 実験

- **計算環境**
  - FX100
    - **名古屋大学情報基盤センター**
    - Fujitsu SPARC64 XIfx 32**コア**, Tofu2
  - CX400
    - **九州大学情報基盤研究開発センター**
    - Intel Xeon (Sandy Bridge) 16**コア**, InfiniBand FDR
- **実施時期**
  - 2017**年** 7**月**
- **評価対象**
  - **集団通信**：Alltoall, Allreduce
  - **一対一通信**：Ring (**片方向のリング状隣接通信**)

FX100のアシスタントコア

- **実行時オプション**
  - **モード番号：**
    - 1: **指定区間** (MPI**呼び出し無し**)
    - 2：**指定区間** (MPI**呼び出し有り**)
    - 3：**自動区間**

```
--mca opal_progress_thread_mode モード番号
```

- **区間指定**
  - **モード番号** 1, 2**のみ**

```
ts = MPI_Wtime();
MPI_Ialltoall(M);
FJMPI_Progress_start();
do_comp(N);
FJMPI_Progress_stop();
MPI_Wait();
Tovlp = MPI_Wtime() - ts;
```

CX400のプログレススレッド

- Intel MPI
  - **環境変数**

```
I_MPI_ASYNC_PROGRESS=1
```

    - **追加オプション：プログレススレッドを割り当てる CPUを固定**

```
I_MPI_ASYNC_PROGRESS_PIN=CPU番号
```

- MVAPICH2
  - **環境変数**

```
MV2_SMP_USE_CMA=0
MV2_ENABLE_AFFINITY=0
MV2_ASYNC_PROGRESS=1
```

非ブロッキング集団通信の効果計測：OSU Benchmark

- オーバラップ率重視
  - 通信時間に応じて計算量を調整

- 実験毎に計算量が変動
  ⇒ 実装手段の相互評価が困難
- 計算まで含めた全体的な効果の評価が困難

```
ts = MPI_Wtime();
MPI_Ialltoall();
MPI_Wait();
Tcomm = MPI_Wtime() - ts;

ts = MPI_Wtime();
MPI_Ialltoall();
tcs = MPI_Wtime();
while (MPI_Wtime() - tcs < Tcomm)
    dummy_comp();
Tcomp = MPI_Wtime() - tcs;
MPI_Wait();
Tall = MPI_Wtime() - ts;

overlap = (Tall - Tcomp) / Tcomm;
```

通信時間

通信時間を超えるまで計算

計算時間

全体時間

今回使用したプログラム

Overlap Test

- 通信量、計算量はパラメータ指定

- 同じパラメータによる全体時間で実装手段の優劣を評価
  - ブロッキング集団通信の時間も計測

- 実装手法相互の比較
- 全体的な効果の評価

```
get_param(&M, &N);

ts = MPI_Wtime();
MPI_Alltoall(M);
Tcomm = MPI_Wtime() - ts;

ts = MPI_Wtime();
MPI_Ialltoall(M); MPI_Wait();
Tnbcomm = MPI_Wtime() - ts;

ts = MPI_Wtime();
do_comp(N);
Tcomp = MPI_Wtime() - ts;

ts = MPI_Wtime();
MPI_Alltoall(M);
do_comp(N);
Tblock = MPI_Wtime() - ts;

ts = MPI_Wtime();
MPI_Ialltoall(M);
do_comp(N);
MPI_Wait();
Tovlp = MPI_Wtime() - ts;
```

ブロッキング集団通信時間

非ブロッキング集団通信時間

計算時間

ブロッキング全体時間

オーバラップ全体時間

計算コード

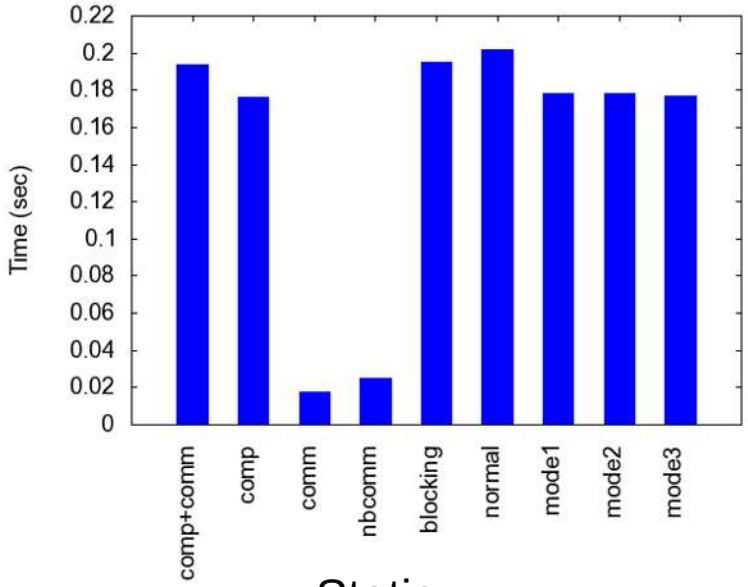
- NxN**密行列積**
  - **プロセス並列**：
    - **行列** A, C**をプロセス分割**
  - **スレッド並列**：
    - **最外ループを並列化**
    - **スケジュール**：static or dynamic

```
do_comp(N)

  nn = N / procs

  #pragma omp parallel for private(j, k) schedule (...)
    for (i = 0; i < nn; i++)
          for (k = 0; k < N; k++)
              for (j = 0; j < N; j++)
                  c[i*N+j] += a[i*N+k] * b[k*N+j];
```

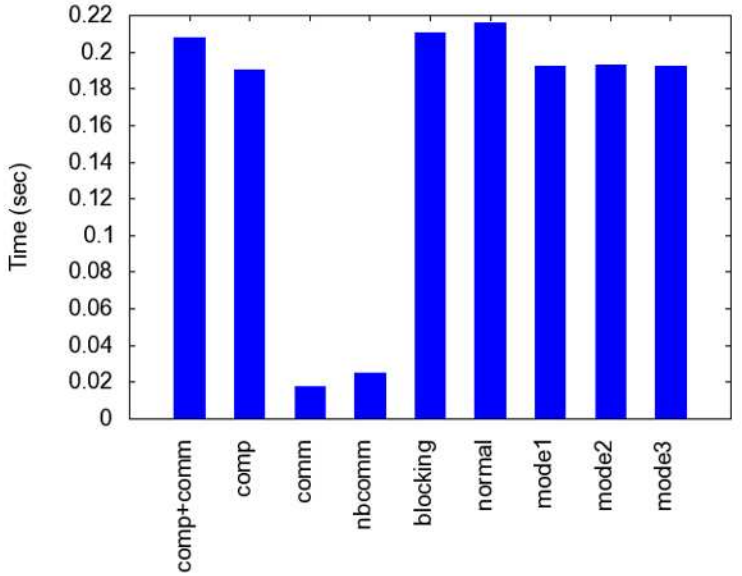
FX100: Alltoall

48nodes, 48procs, 1MB, 1536x1536

comp+comm: **通信時間と計算時間の和**
comp: **計算時間**
comm: **ブロッキング集団通信時間**
nbcomm: **非ブロッキング集団通信時間**
blocking: **ブロッキング集団通信+計算**
normal: **非ブロッキング集団通信+計算**
(プログレススレッド無し)
mode1〜mode3: **非ブロッキング集団通信+計算**
(プログレススレッドあり)



Static

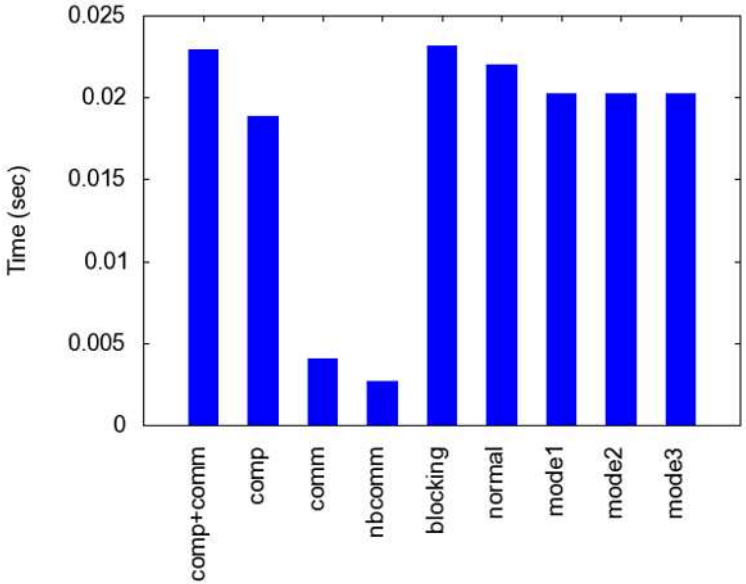


Dynamic

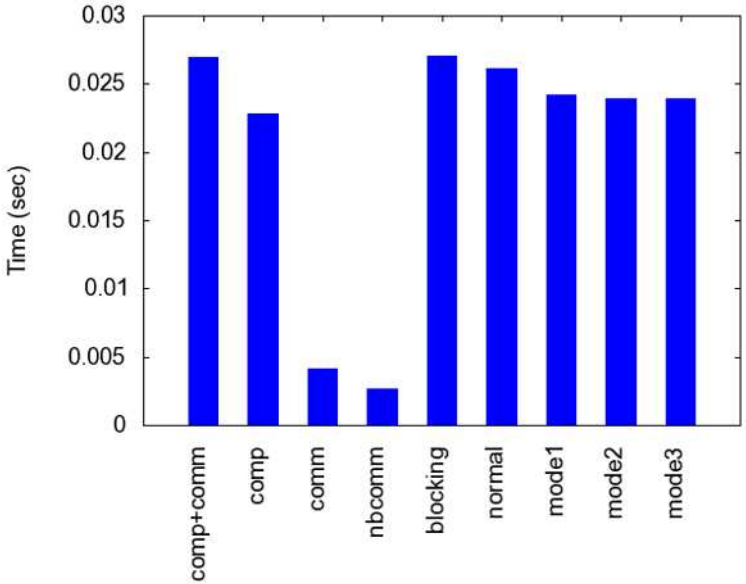
- **通信をほぼ全部隠蔽**
- Dynamic**スケジューリングの方が若干遅い**

FX100: Allreduce

48nodes, 48procs, 1MB, 768x768



static sched

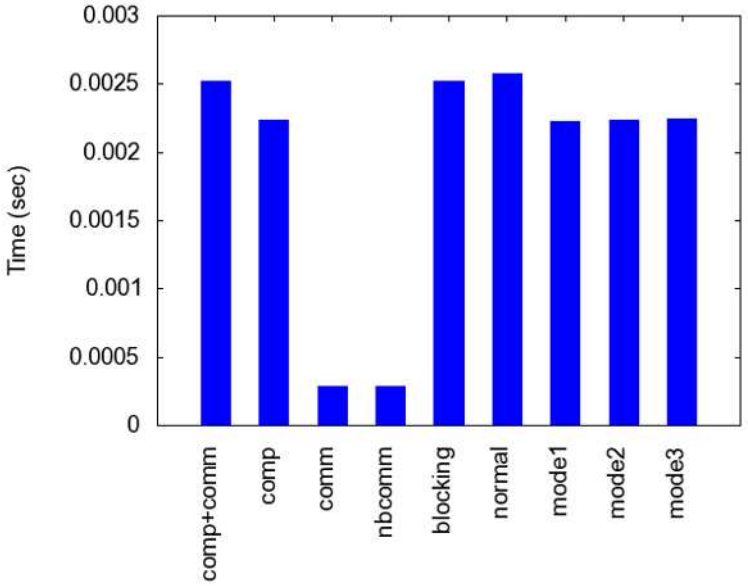


dynamic sched

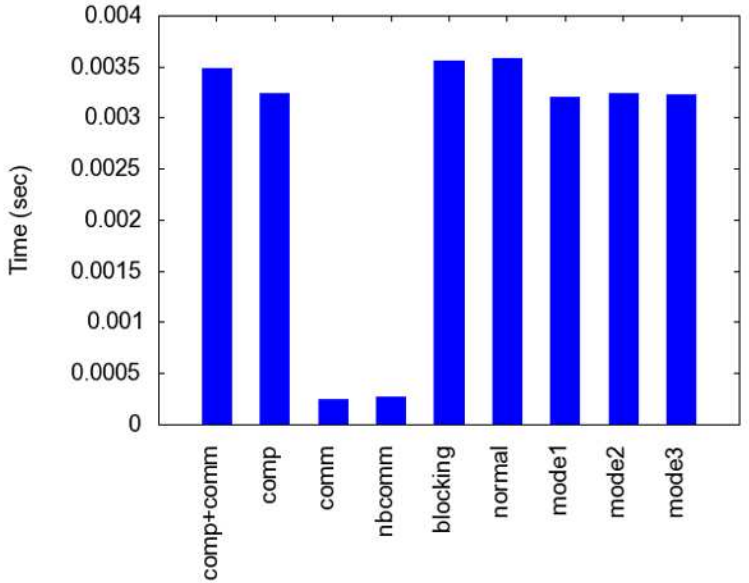
- **通信をある程度隠蔽**
- Dynamic**スケジューリングの方が若干遅い**

FX100: Ring

48nodes, 48procs, 1MB, 384x384



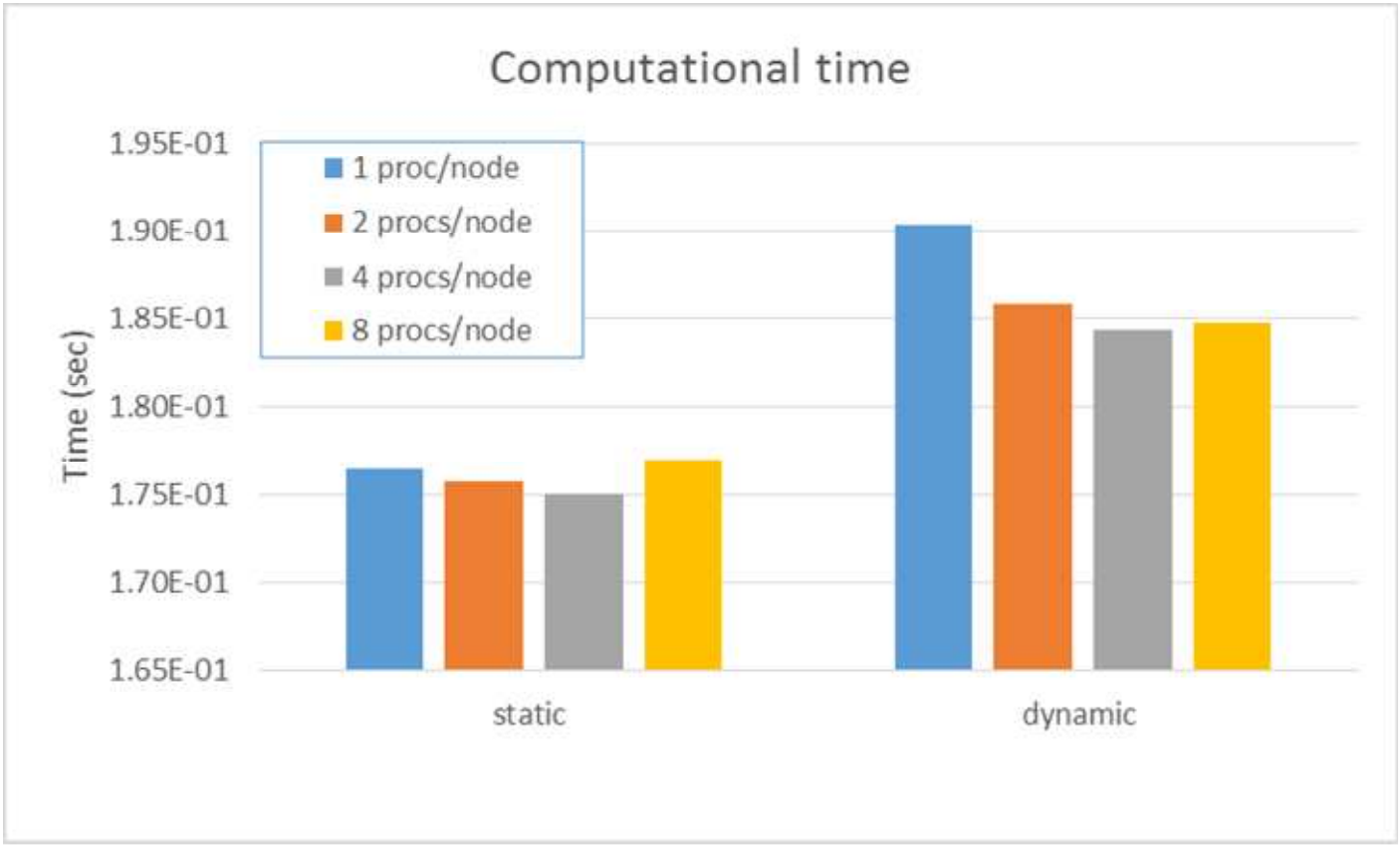
static sched



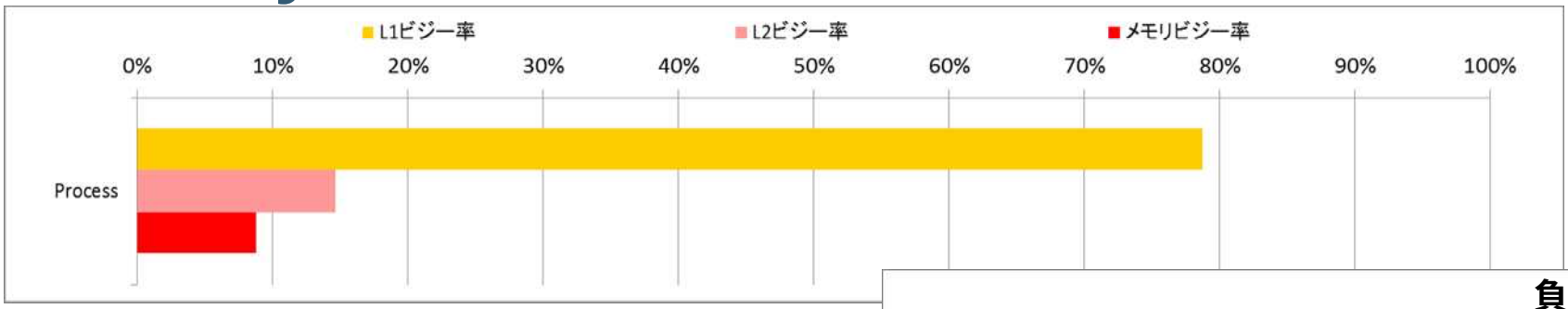
dynamic sched

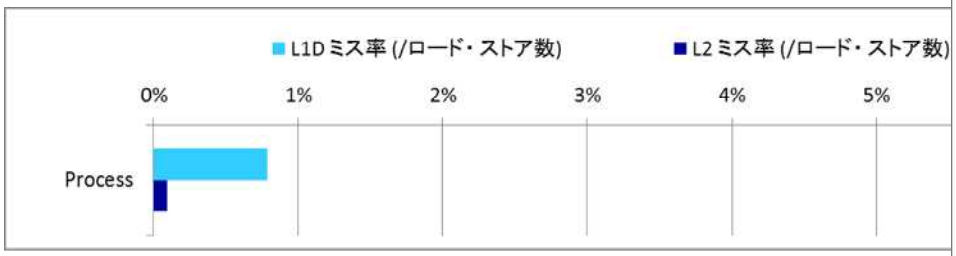
- **通信をほぼ全部隠蔽**
- Dynamic**スケジューリングの方が若干遅い**

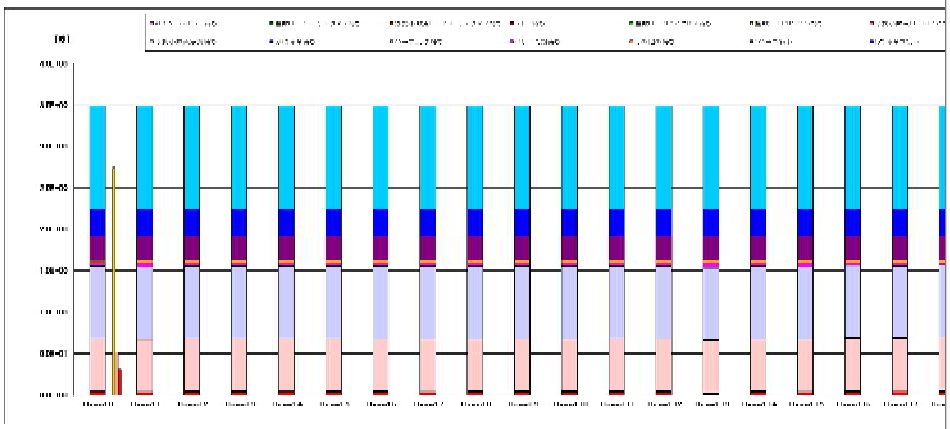
スケジューリングポリシーによる行列積計算時間の違い



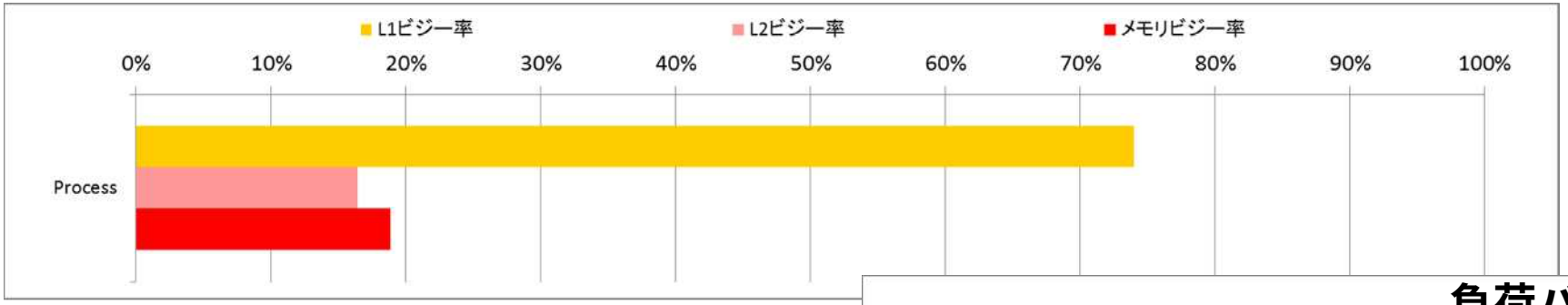
Analysis of Static

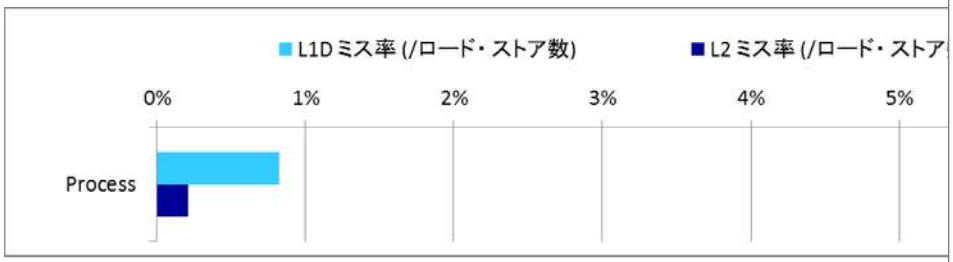


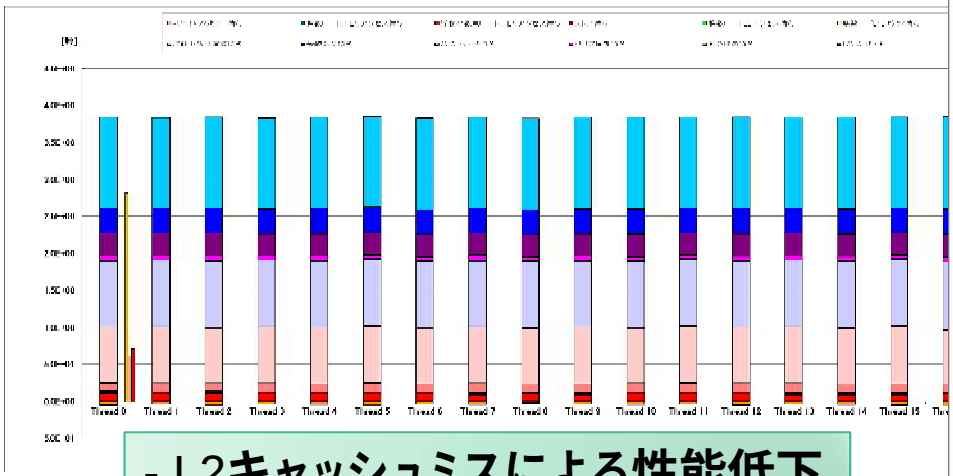




Analysis of Dynamic





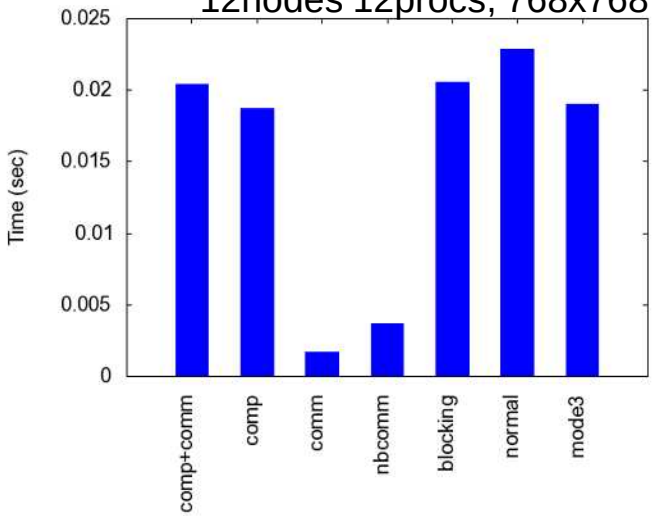


- L2キャッシュミスによる性能低下

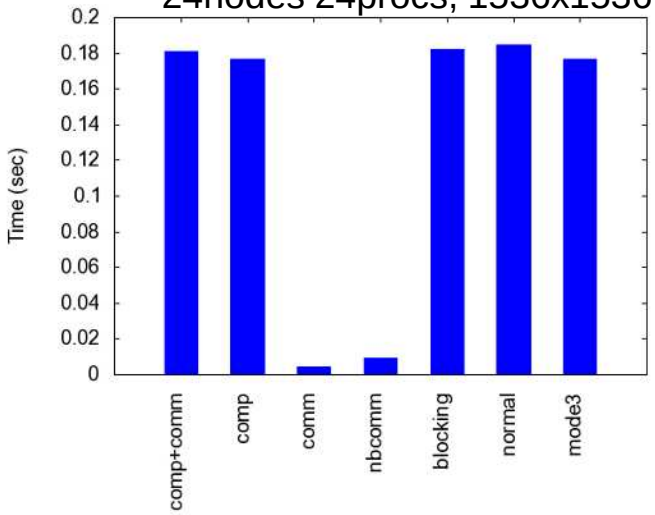
FX100: Alltoall with varying nodes

1MB, static

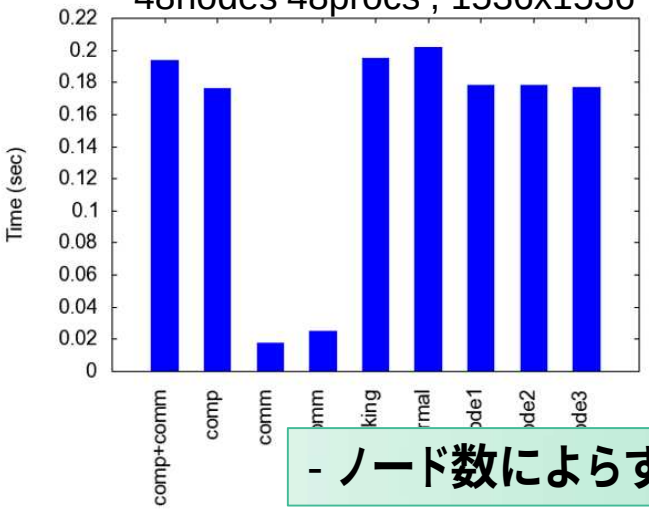
12nodes 12procs, 768x768



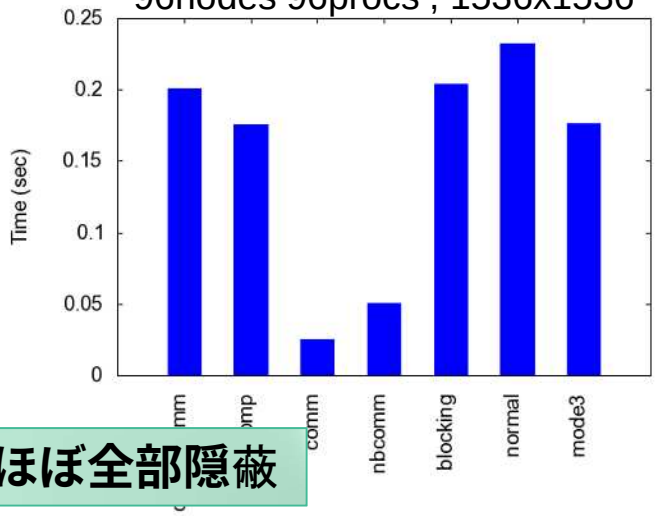
24nodes 24procs, 1536x1536



48nodes 48procs , 1536x1536



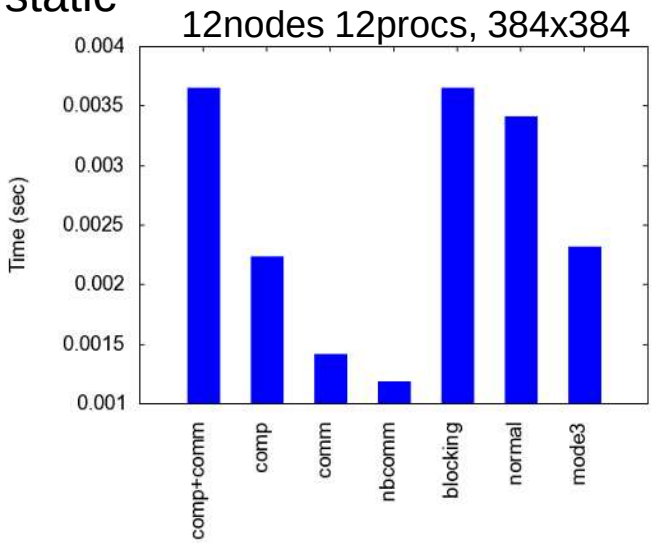
96nodes 96procs , 1536x1536

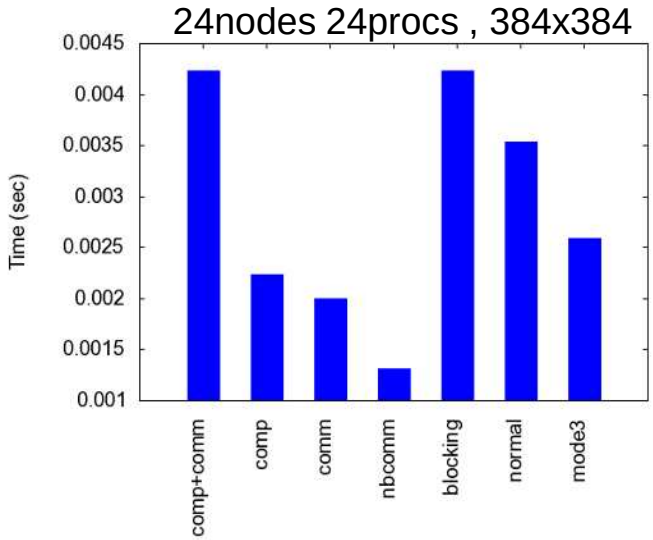


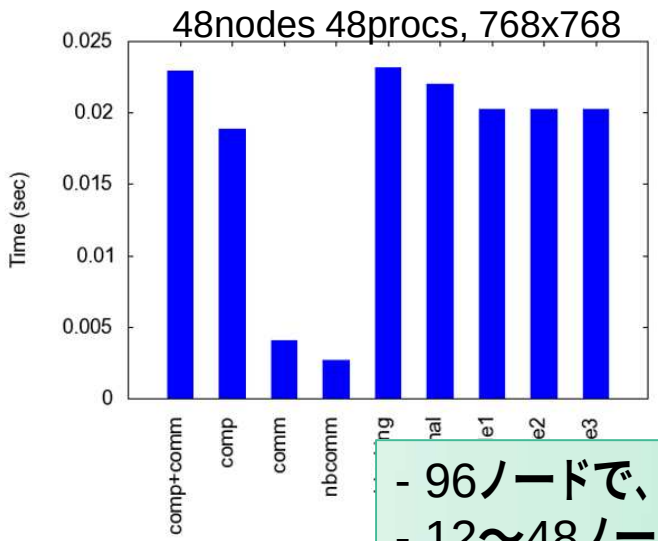
- ノード数によらず、通信をほぼ全部隠蔽

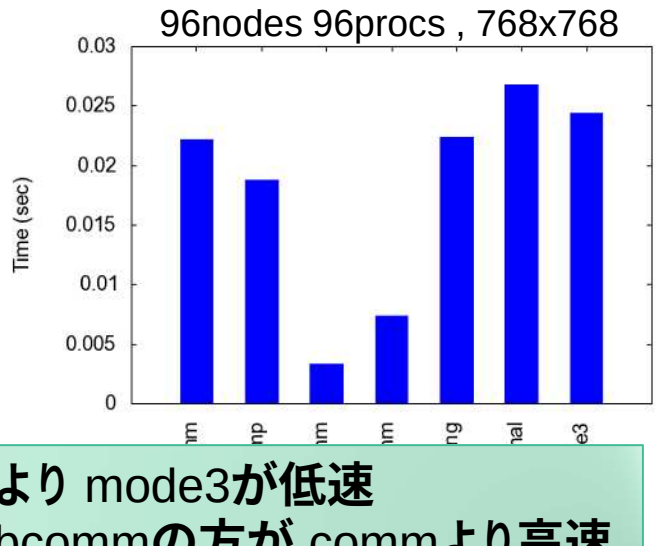
FX100: Allreduce with varying nodes

1MB, static





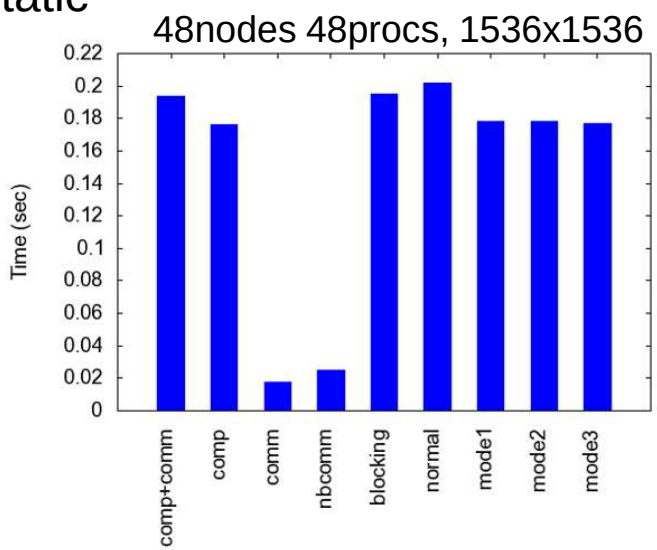


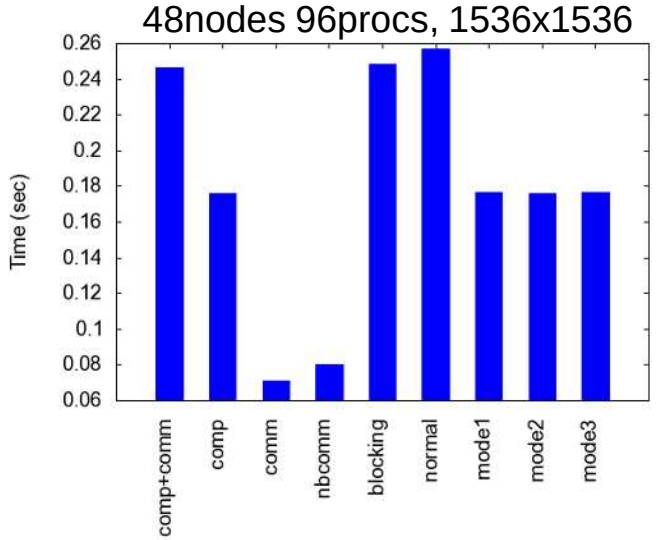


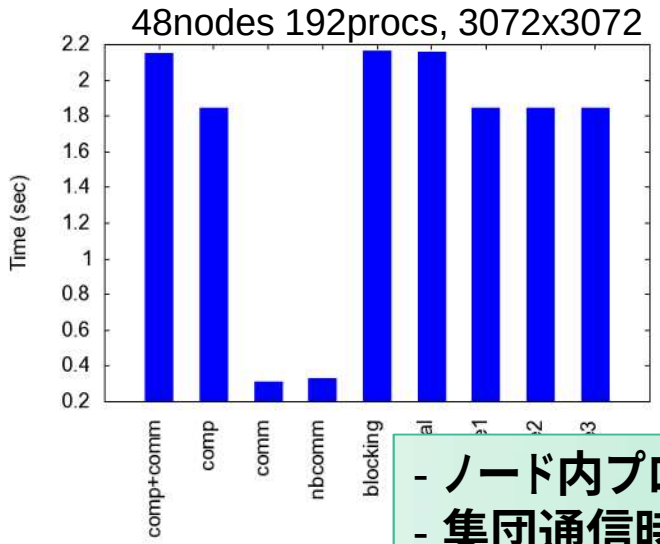
- 96ノードで、blockingより mode3が低速
- 12〜48ノードでは、nbcommの方が commより高速

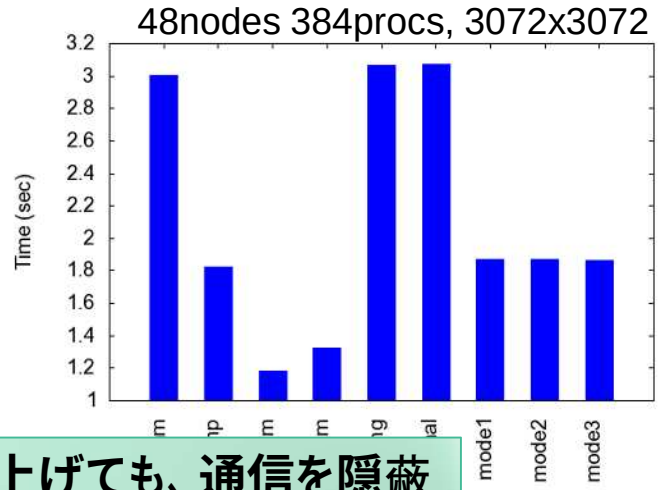
FX100: Alltoall with varying procs/node

1MB, static





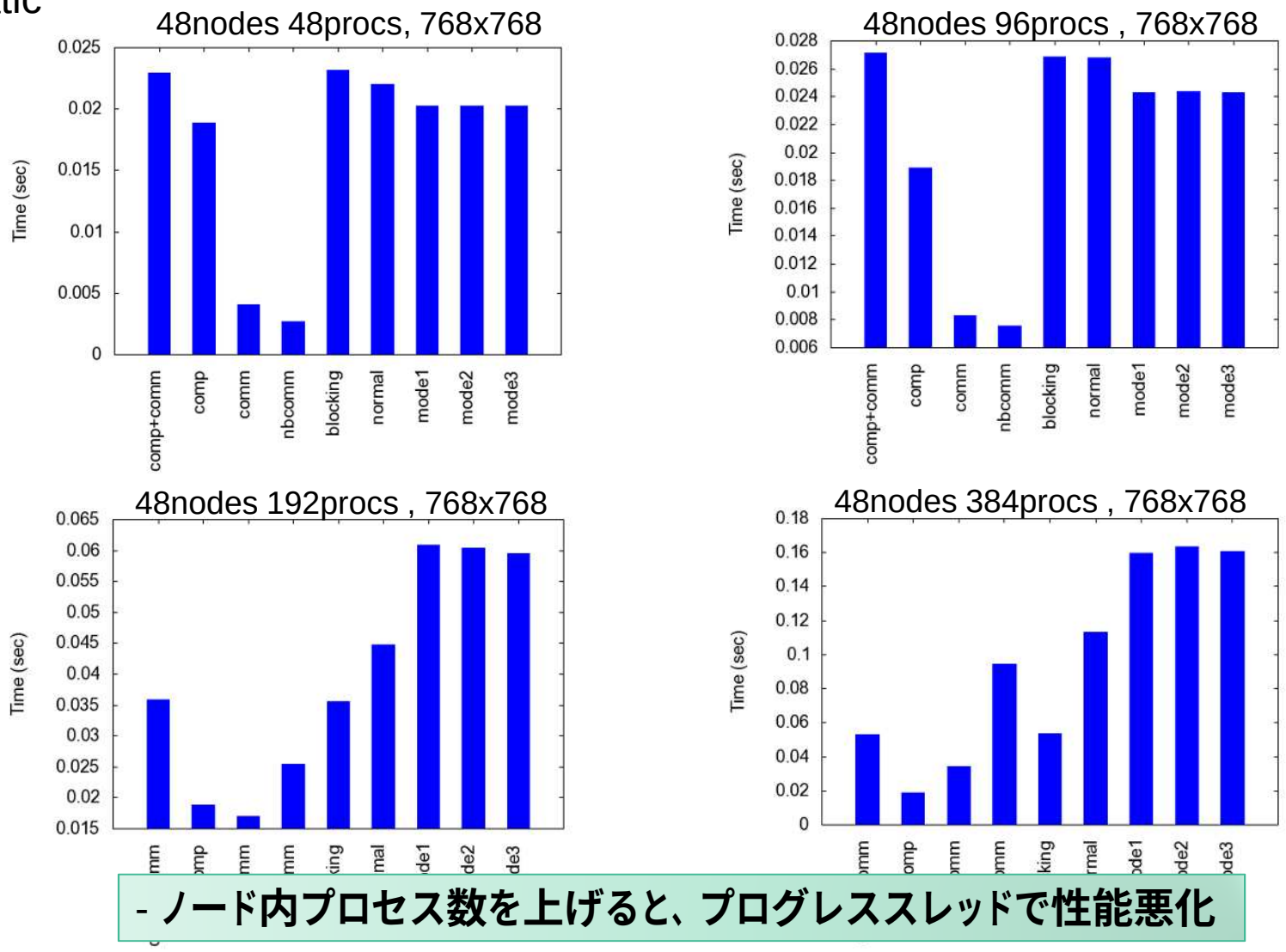




- ノード内プロセス数を上げても、通信を隠蔽
- 集団通信時間はプロセス数に応じて増加

FX100: Allreduce with varying procs/node

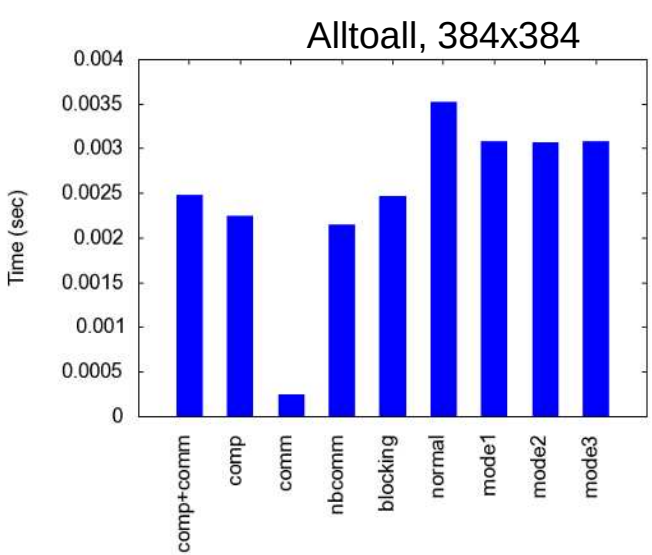
1MB, static

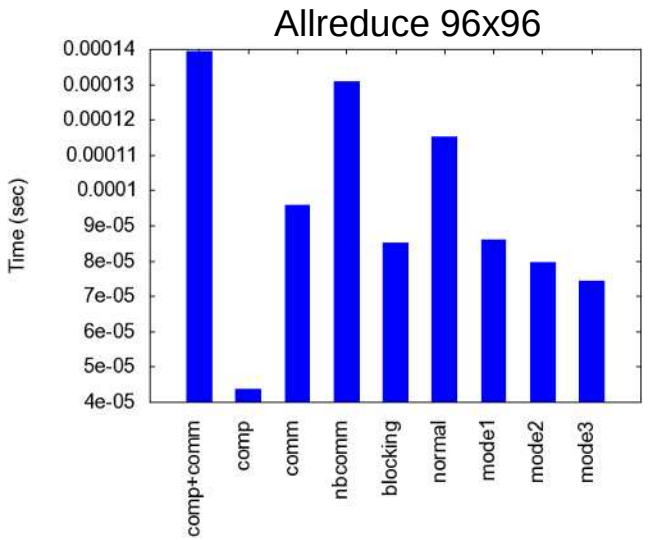


- ノード内プロセス数を上げると、プログレススレッドで性能悪化

FX100: Small message size

48nodes, 48procs, 128bytes, Static

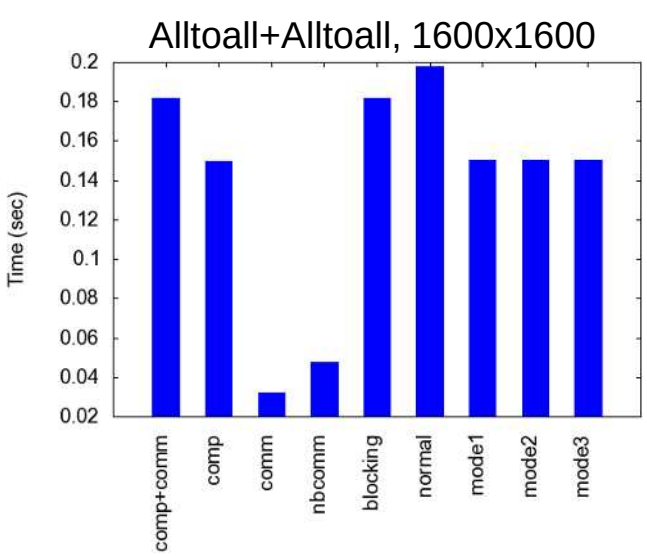


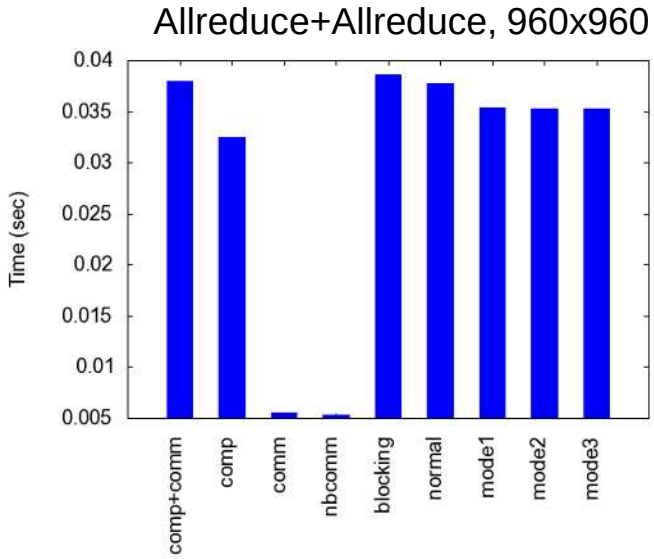


- Alltoallは、性能悪化
- Allreduceは、自動区間モードで性能向上

FX100: Overlapping multiple collectives

48nodes, 48procs, 1MB, Static





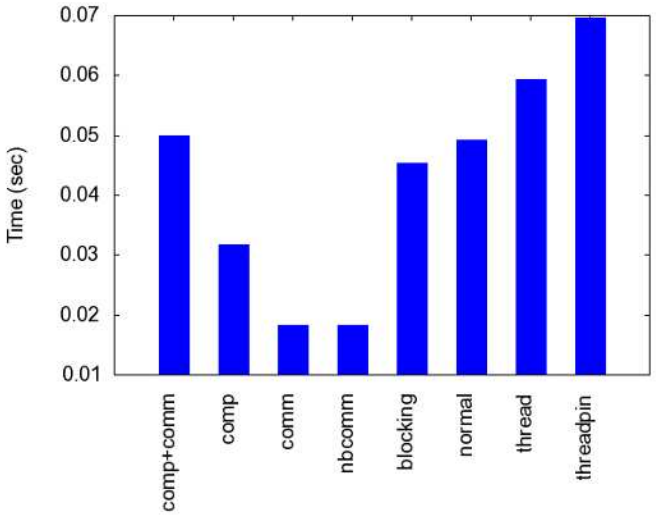
- 複数の集団通信も隠蔽可能

CX400 Intel: Alltoall

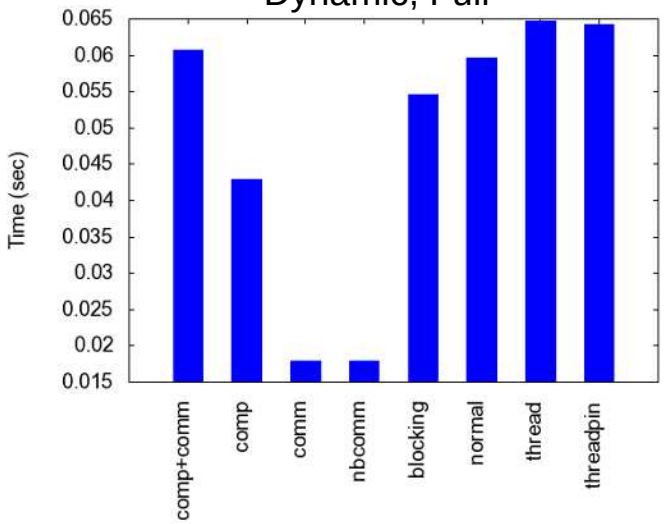
32nodes, 32procs, 1MB, 1024x1024

Static / Dynamic: OpenMPのスケジューリング
Full / Spare: プログレススレッドのコアへの割り当て

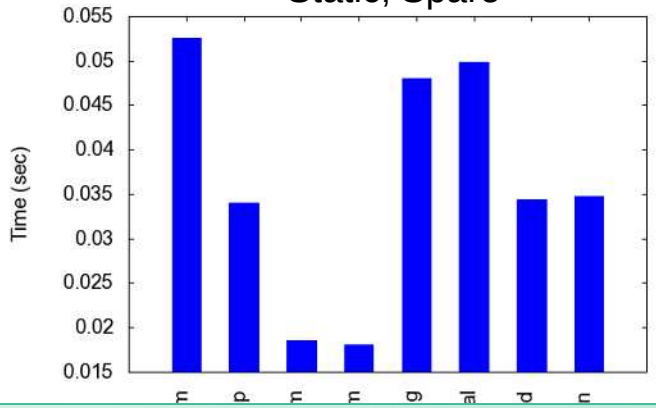
Static, Full



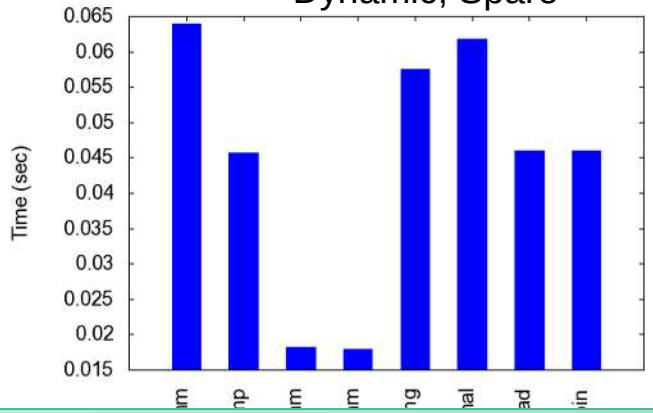
Dynamic, Full



Static, Spare



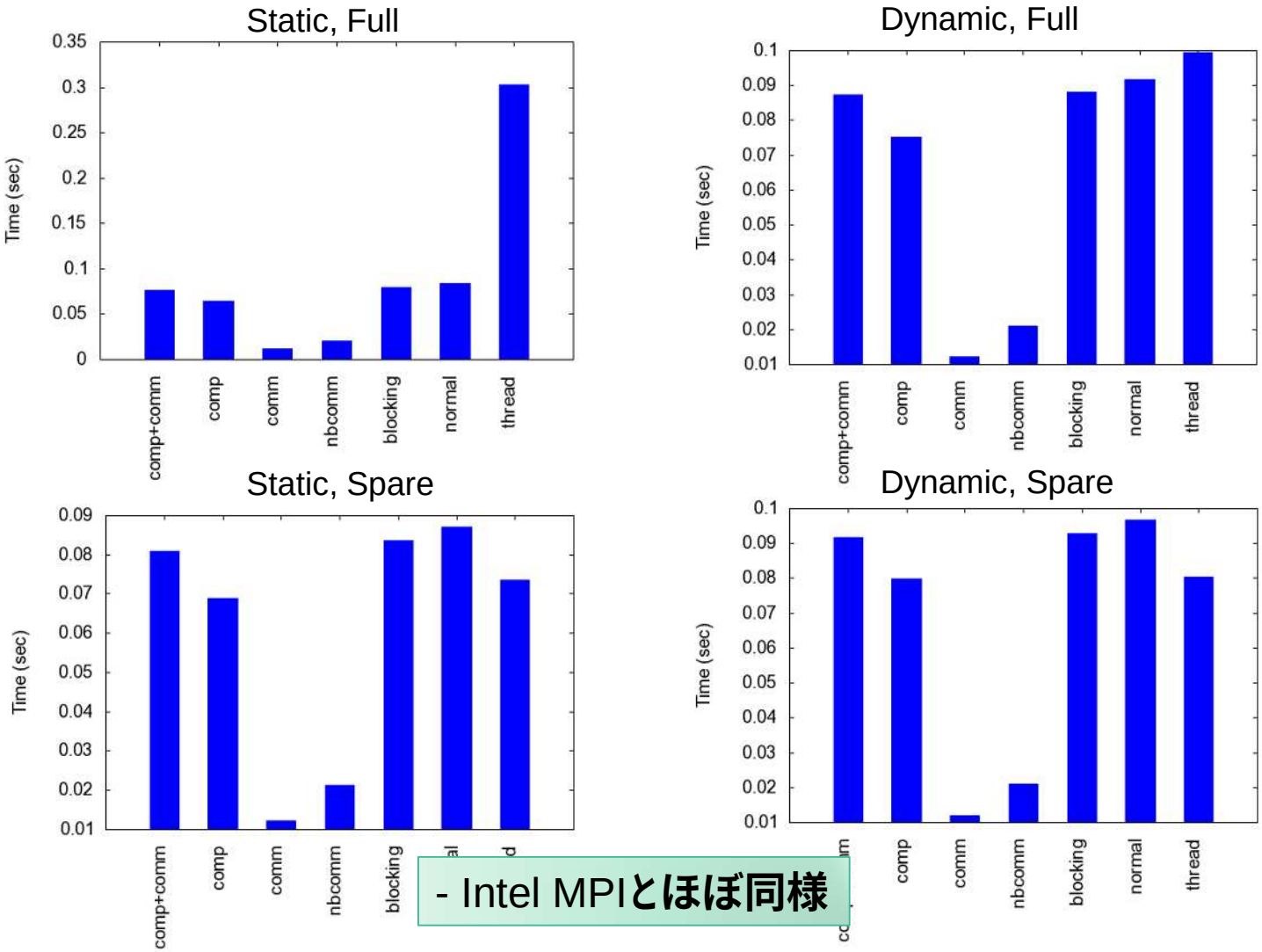
Dynamic, Spare



- Spare コアの利用により計算時間は若干延びるが全体時間は Fullより短い

CX400 MVAPICH2: Alltoall

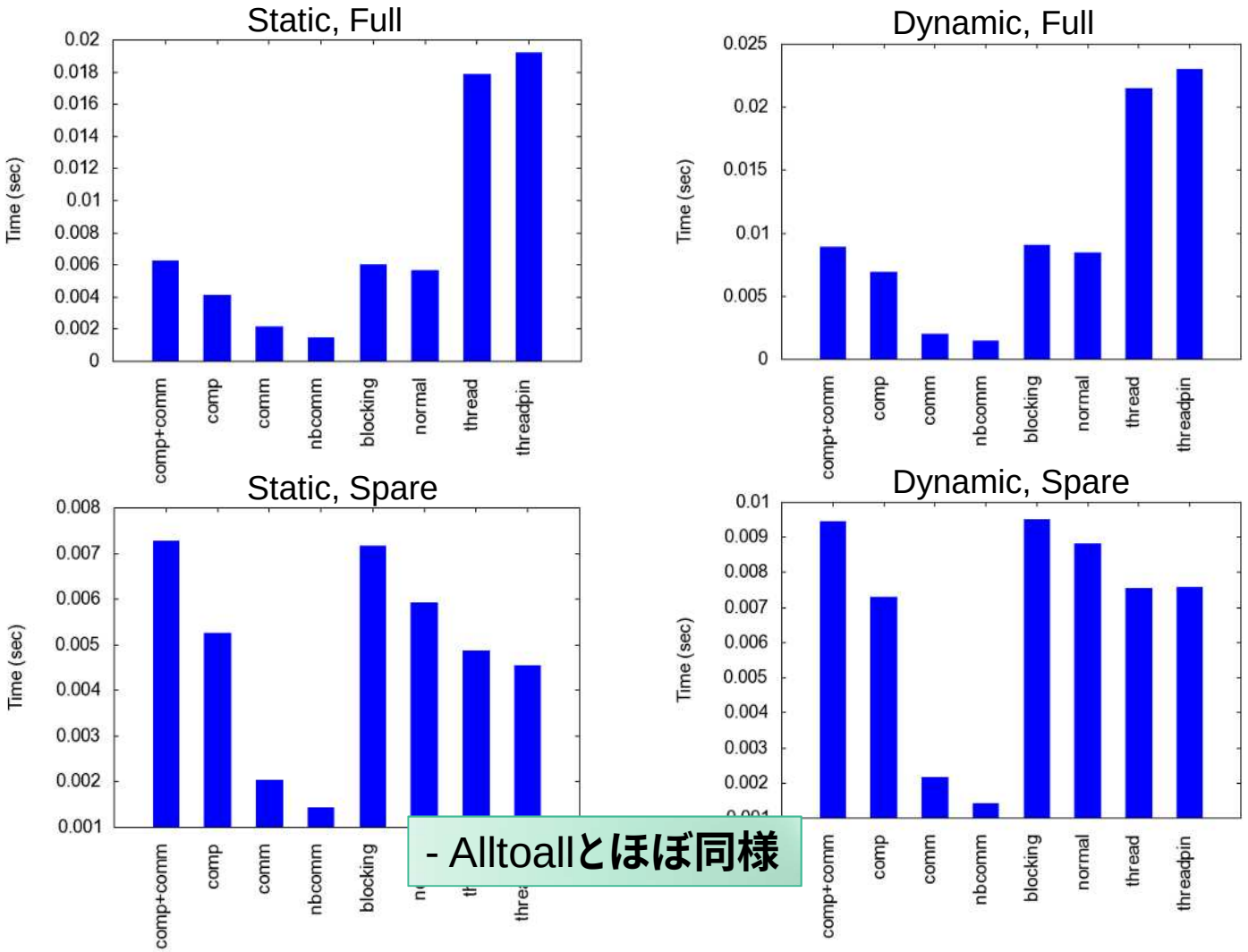
32nodes, 32procs, 1MB, 1024x1024



- Intel MPIとほぼ同様

CX400 Intel: Allreduce

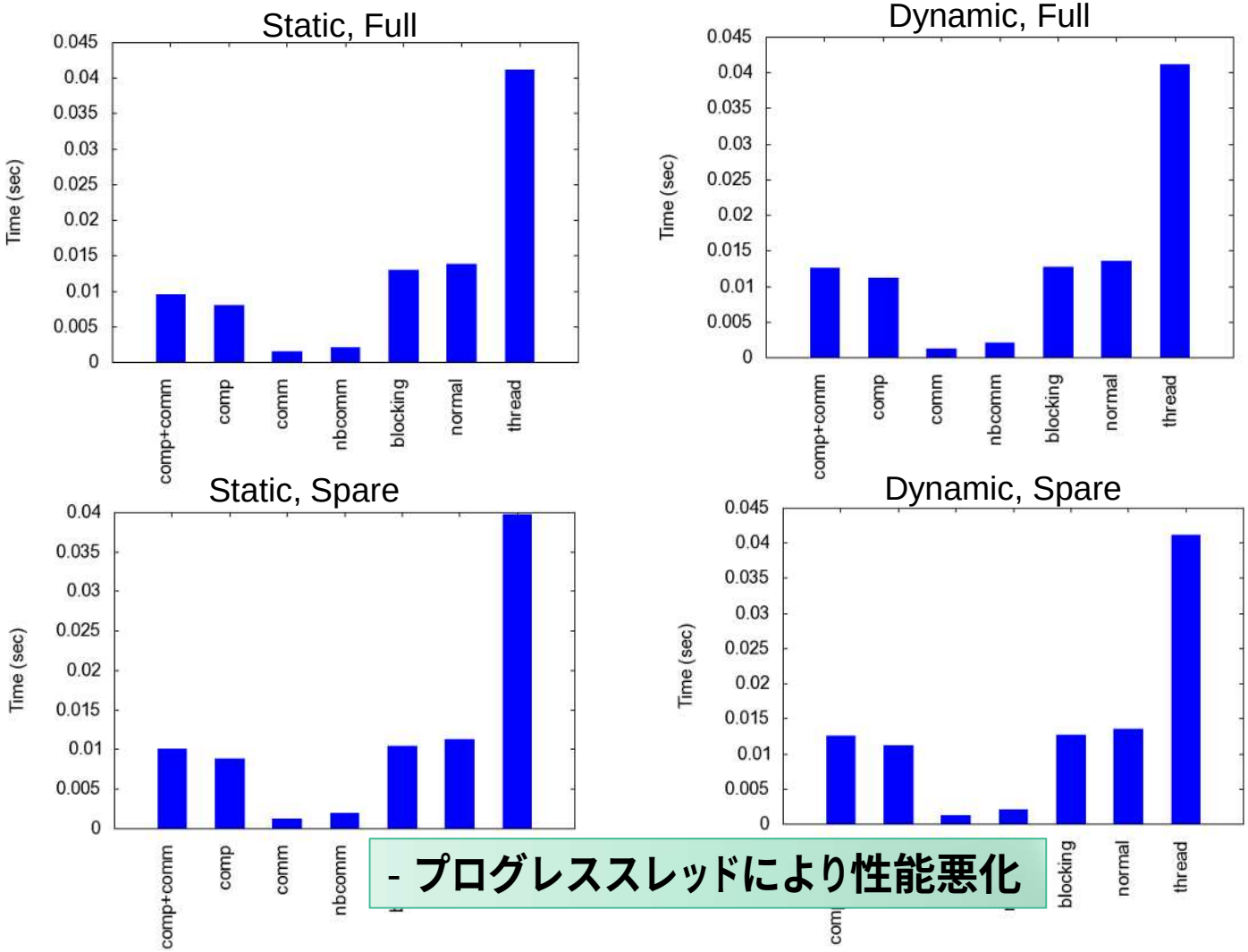
32nodes, 32procs, 1MB, 512x512



- Alltoallとほぼ同様

CX400 MVAPICH2: Allreduce

32nodes, 32procs, 1MB, 512x512

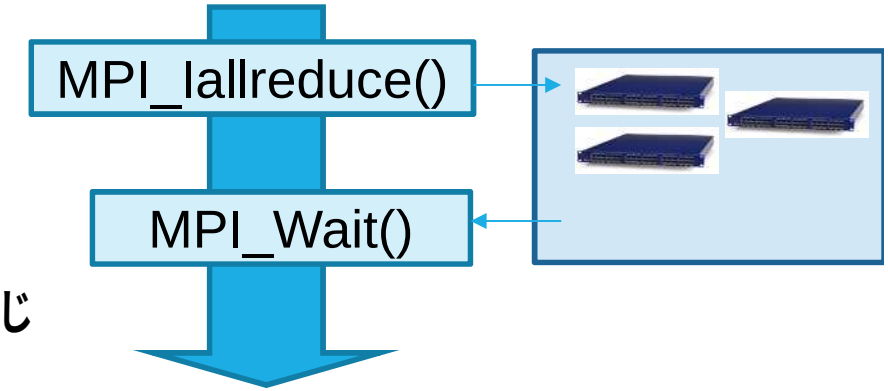


検証1） まとめ

- 非ブロッキング集団通信の効果
  - プログレススレッド専用コアがある場合、通信隠蔽効果が期待できる
    - 効果が得られなかったケース：
      - メッセージサイズが小さい場合
      - ノード数やノード内プロセス数が多い場合の Allreduce
      - MVAPICH2の Allreduce
- OpenMPのスケジューリングポリシー
  - 少なくとも今回の場合、全体性能としては staticの方が良い場合が多かった
    - 計算に依存する

検証2） 集団通信のオフロードの効果

- Mellanox社 SHARP
  - InfiniBandスイッチに「集約型」の集団通信をオフロード
  - 対象：
    Allreduce, Reduce, Bcast, Barrier
    - 入力データと出力データのサイズが同じ
    - 集約操作で実現可能



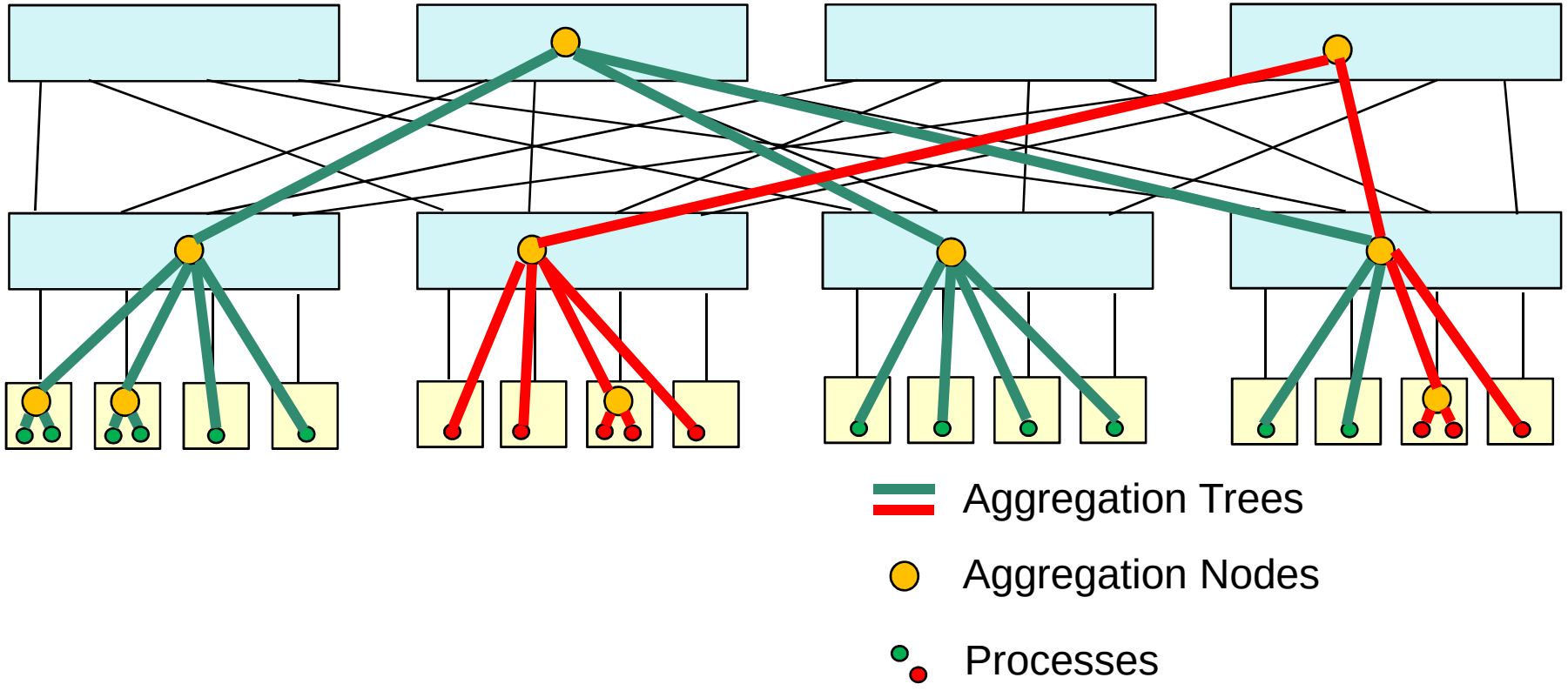
  - 利用法：
    Mellanox社の HPC-X中の MPIライブラリで以下を指定

```
-x HCOLL_ENABLE_SHARP=2
-x HCOLL_ENABLE_NBC=1
-x HCOLL_POLLING_LEVEL=1
```

    - MVAPICH2 2.3 Release Candidate 2 でも利用可能らしい

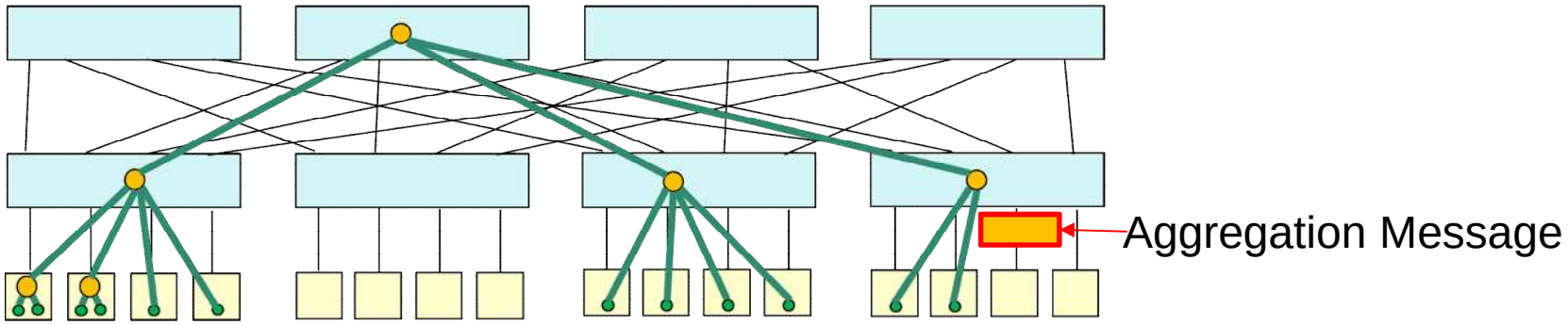
Aggregation Tree

- SHARPが Aggregation Nodeとプロセスで構成する仮想的な木構造
  - Aggregation Node: スイッチや計算ノードに配置する仮想的なノード



SHARPによる Allreduce

- Aggregation Tree上で Aggregation Messageにより上方へ集約し、結果を下方に伝搬
  - Aggregation Message: 最大 256バイトのペイロードを持つ集約用メッセージ
    - 256バイトを超えるサイズ：複数のメッセージに分割
  - 各 Aggregation Nodeで、集約演算の計算順序を保証



SHARPへの期待

- **高い通信隠蔽率**

- Allreduce, Iallreduceの**所要時間短縮**
  - **データ転送経路上で演算するためデータ転送量を削減**

- **OSジッタ等、プロセス間の不均衡な処理による影響軽減**
  - CPUにおける**待ち時間が不要**

通信隠蔽効果の計測手段

- OSU Micro Benchmarks
  - Ohio State University**が開発**
  - **通信の隠蔽率を計測**
- Overlap Test
  - **自作**
  - **隠蔽後の実行時間を計測**
- PETSc
  - Argonne National Laboratory **が中心となって開発**
  - **通信隠蔽アルゴリズムの効果を計測**

検証2） 実験

- **計算環境**
  - **スーパーコンピュータシステム** ITO
    - **九州大学情報基盤研究開発センター**
    - Intel Xeon Gold 6154 (3.0GHz, 18core) x 2 / node
    - Mellanox InfiniBand EDR
    - Red Hat Linux Enterprise 7.3
    - gcc 4.8.5
    - **使用**MPI**ライブラリ**
      - Mellanox HPC-X 1.9.5, MVAPICH2 2.2, Open MPI 2.1.3
- **実施時期**
  - 2018**年**7**月**
- **対象プログラム**
  - OSU Micro Benchmark
  - Overlap Test（**検証1**）**と同じ**）
  - PETSc

PETSc

- 偏微分方程式でモデル化されたアプリケーションの開発ツール
  - 通信を隠蔽した反復法アルゴリズムを提供

CG

```
do {
  b = beta/betaold;
  ierr = VecAYPX(P,b,Z);            /* p <- z + b* p */
  dpiold = dpi;
  ierr = KSP_MatMult(ksp,Amat,P,W); /* w <- Ap       */
  ierr = VecXDot(P,W,&dpi);         /* dpi <- p'w    */
  KSPCheckDot(ksp,dpi);
  betaold = beta;
  a = beta/dpi;                     /* a = beta/p'w  */
  ierr = VecAXPY(X,a,P);            /* x <- x + ap   */
  ierr = VecAXPY(R,-a,W);           /* r <- r - aw   */
  ierr = KSP_PCApply(ksp,R,Z);      /* z <- Br       */
  ierr = VecXDot(Z,R,&beta);        /* beta <- r'*z  */
  KSPCheckDot(ksp,beta);
  dp = PetscSqrtReal(PetscAbsScalar(beta));
  ksp->rnorm = dp;
  ierr = (*ksp->converged)(ksp,i+1,dp,&ksp->reason,ksp->cnvP);
  if (ksp->reason) break;
  ierr = VecXDot(Z,R,&beta);        /* beta <- z'*r  */
  KSPCheckDot(ksp,beta);
  i++;
} while (i<ksp->max_it);
```

PIPE CG

```
do {
  VecNormBegin(R,NORM_2,&dp);
  VecDotBegin(R,U,&gamma);
  VecDotBegin(W,U,&delta);
  PetscCommSplitReductionBegin(R);
  KSP_PCApply(ksp,W,M);             /* m <- Bw */
  KSP_MatMult(ksp,Amat,M,N);        /* n <- Am */
  VecNormEnd(R,NORM_2,&dp);
  VecDotEnd(R,U,&gamma);
  VecDotEnd(W,U,&delta);
  dp = PetscSqrtReal(PetscAbsScalar(gamma));
  ksp->rnorm = dp;
  (*ksp->converged)(ksp,i,dp,&ksp->reason,ksp->cnvP);
  if (ksp->reason) break;
  beta  = gamma / gammaold;
  alpha = gamma / (delta - beta / alpha * gamma);
  VecAYPX(Z,beta,N);   /* z <- n + beta * z */
  VecAYPX(Q,beta,M);   /* q <- m + beta * q */
  VecAYPX(P,beta,U);   /* p <- u + beta * p */
  VecAYPX(S,beta,W);   /* s <- w + beta * s */
  VecAXPY(X, alpha,P); /* x <- x + alpha * p */
  VecAXPY(U,-alpha,Q); /* u <- u - alpha * q */
  VecAXPY(W,-alpha,Z); /* w <- w - alpha * z */
  VecAXPY(R,-alpha,S); /* r <- r - alpha * s */
  gammaold = gamma;
  i++;
} while (i<ksp->max_it);
```

集約操作開始

集団通信開始

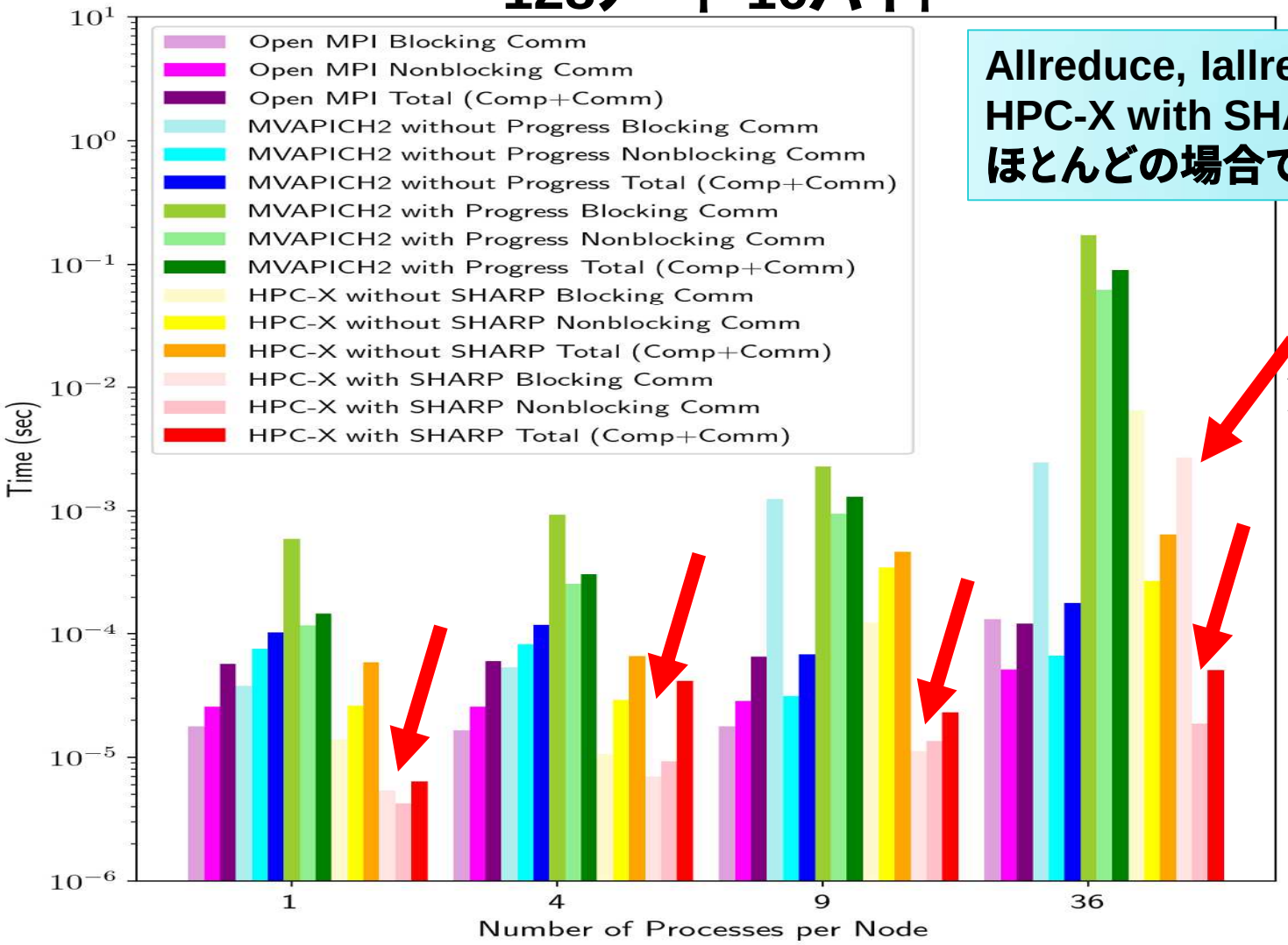
集約操作完了待ち

実験で検証したいこと

- **基本的な** MPI_Allreduce**の性能、通信隠蔽率**
  - OSU Micro Benchmarks **の** MPI_Iallgather**向けコードを** MPI_Iallreduce**向けに改変**
- **通信隠蔽による全体の実行時間への影響**
  - Overlap Test**プログラム**
- PETSc**の通信隠蔽アルゴリズムの効果**
  - tutorial**の** ex3.c **を** CG, PIPE CG**で実行**
    - **前処理**：Jacobi

MPI_Allreduce, Iallreduceの所要時間 (OSU Micro Benchmark)

128ノード 16バイト

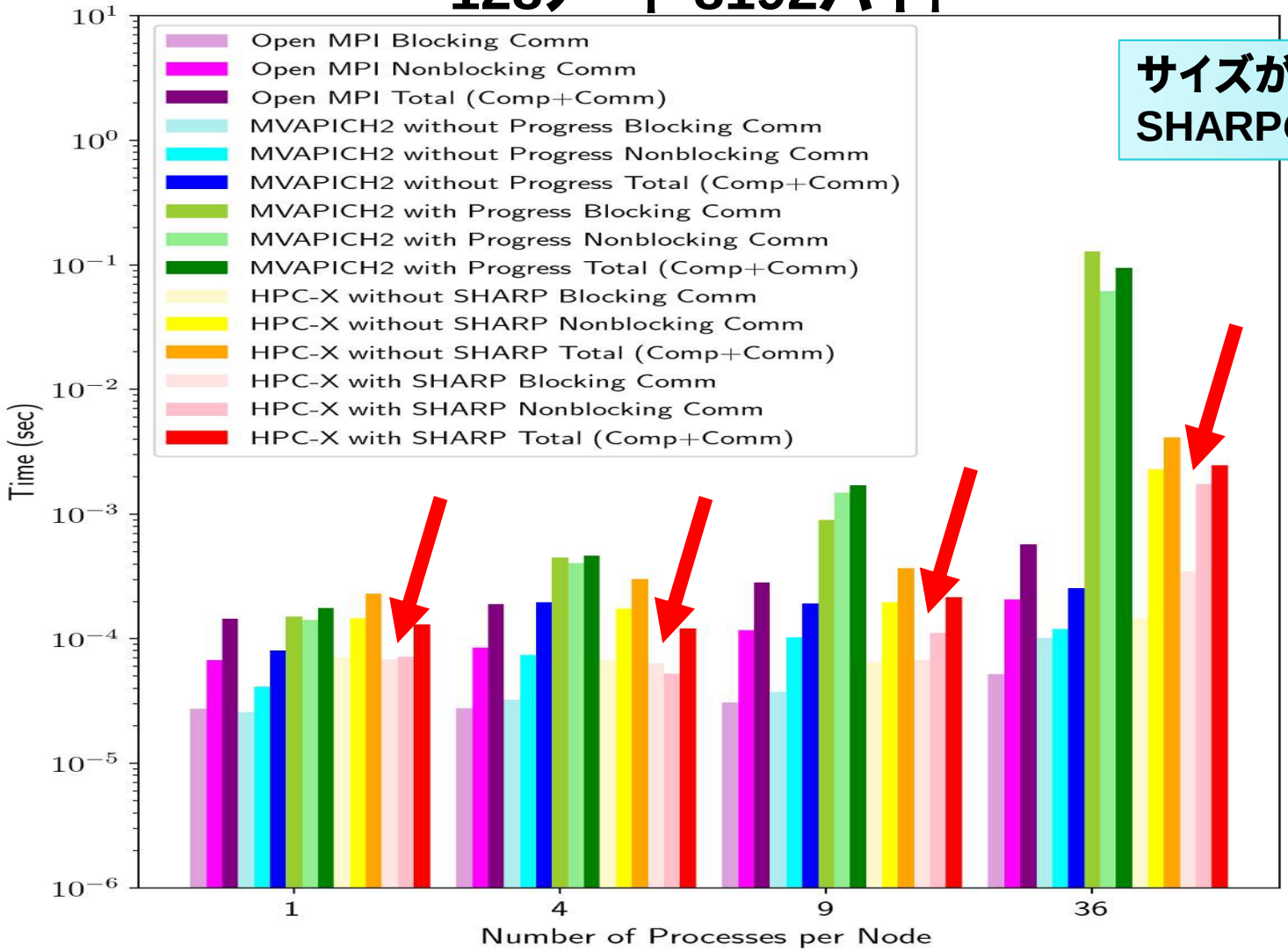


Allreduce, Iallreduceとも HPC-X with SHARPが ほとんどの場合で最速

ノード内プロセス数が多い場合、Allreduceの性能低下

MPI_Allreduce, Iallreduceの所要時間 (OSU Micro Benchmark)

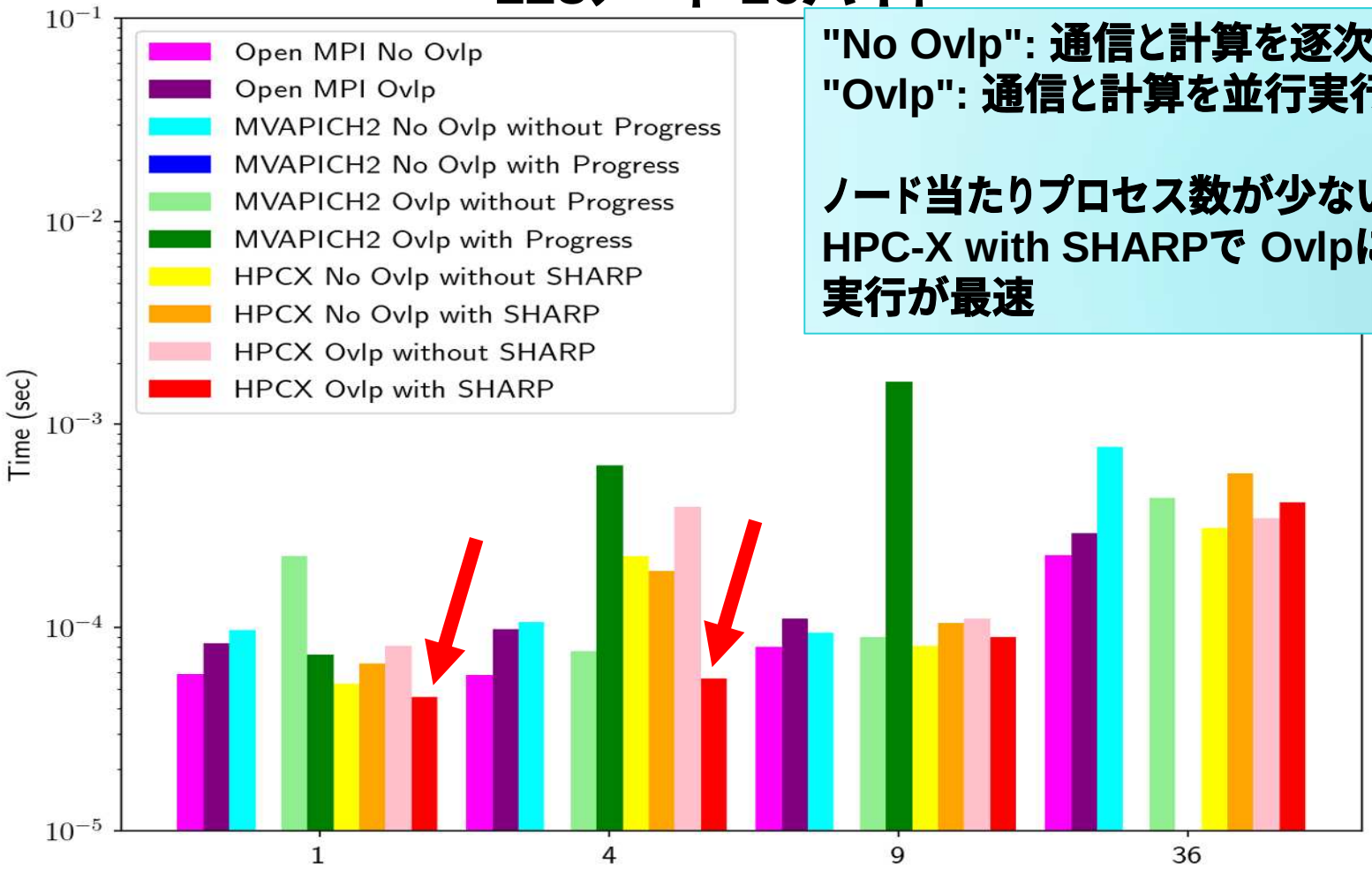
128ノード 8192バイト



サイズが大きくなると、SHARPの効果なし

Overlap Test

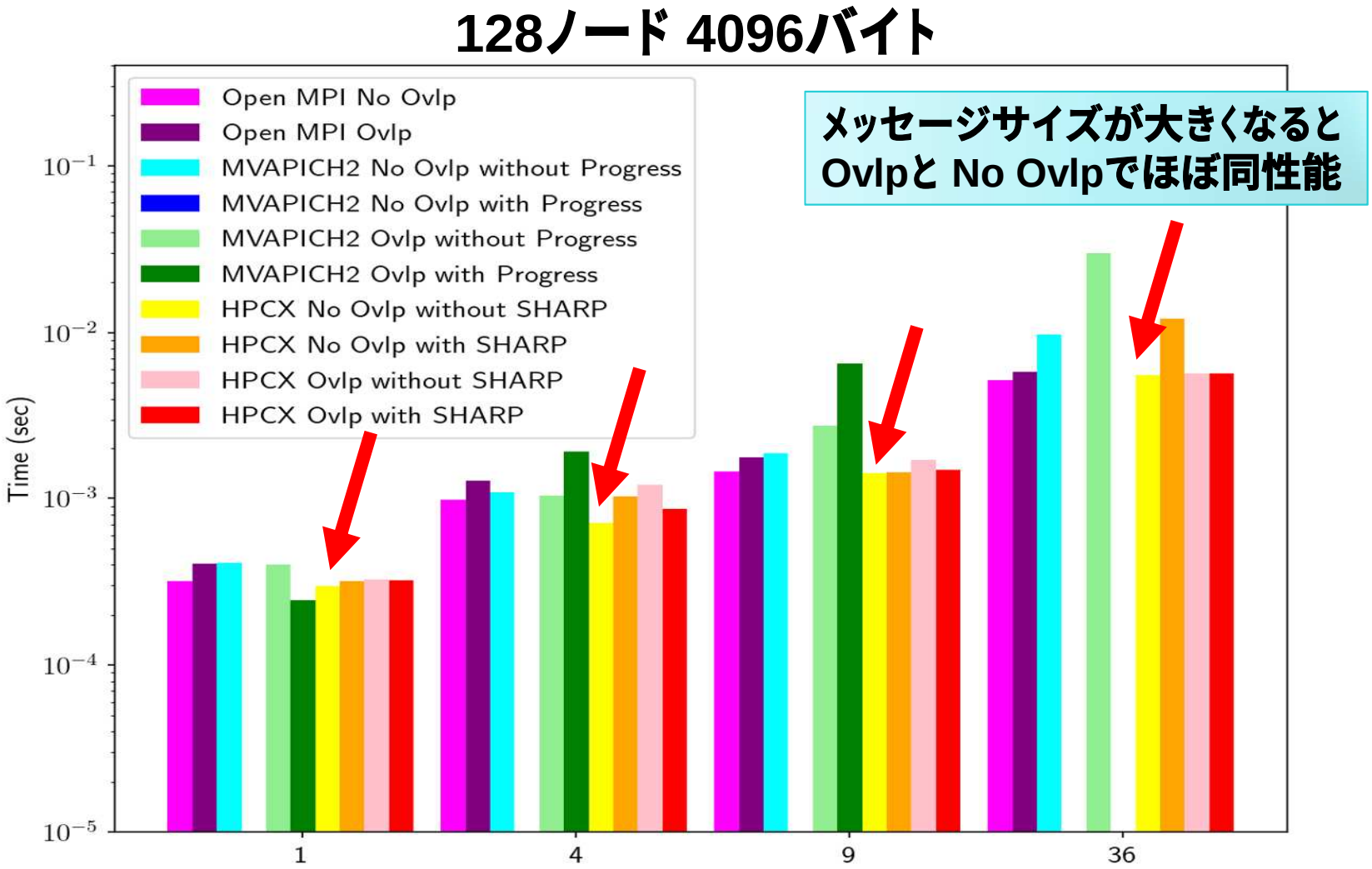
128ノード 16バイト



"No Ovlp": 通信と計算を逐次実行
"Ovlp": 通信と計算を並行実行

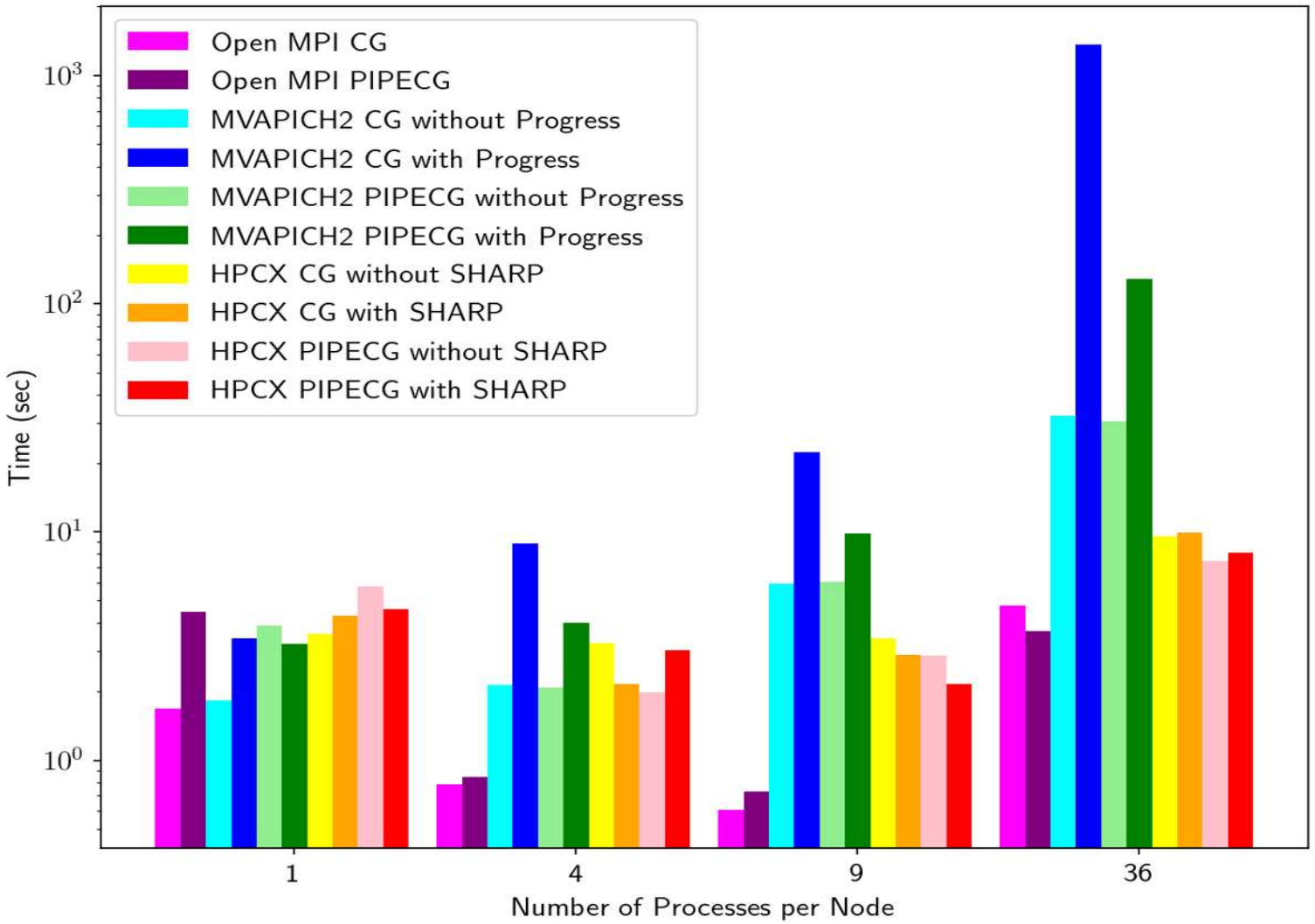
ノード当たりプロセス数が少ない場合
HPC-X with SHARPで Ovlpによる
実行が最速

Overlap Test



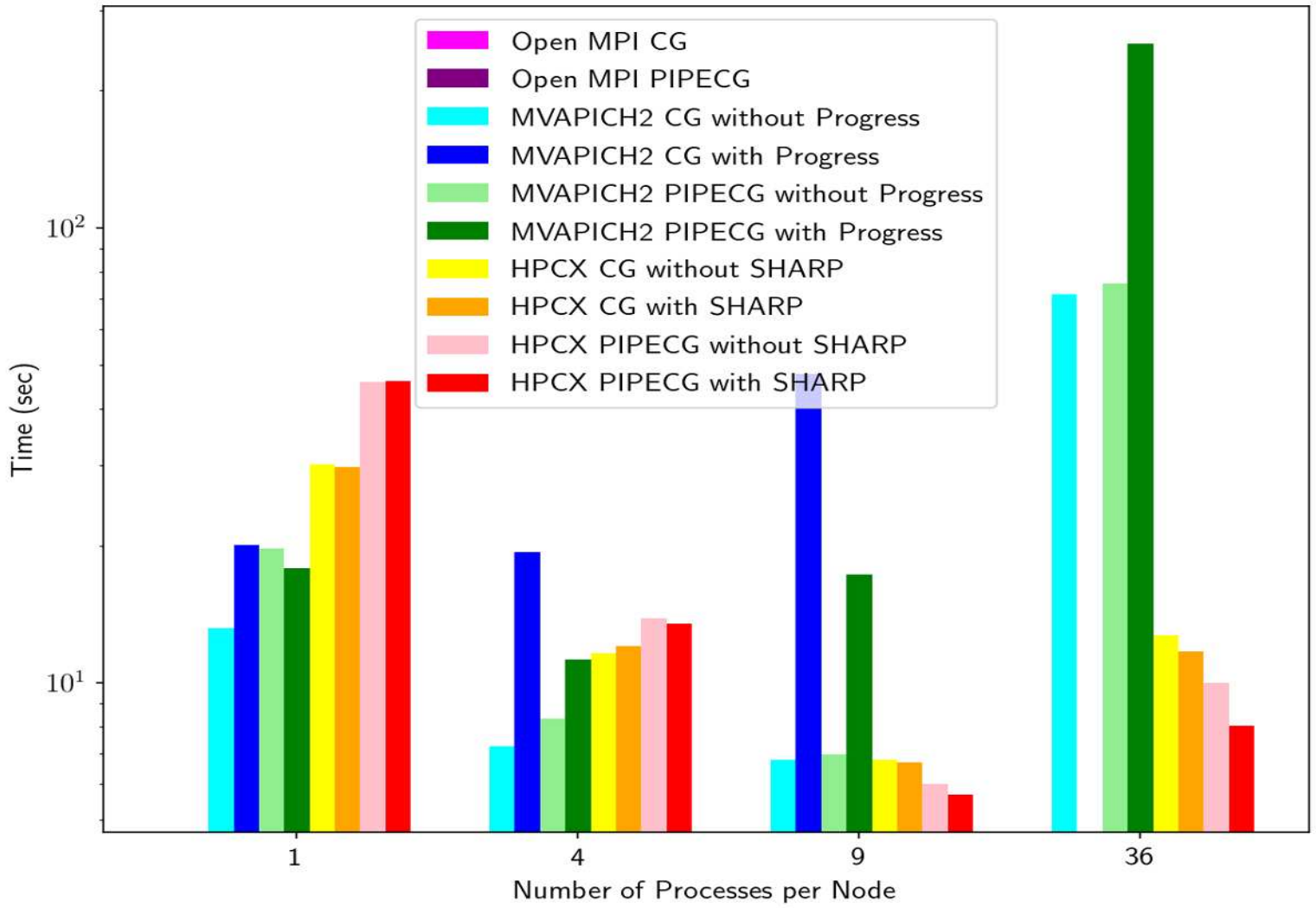
PETSc with Jacobi前処理

128ノード 1536x1536



PETSc with Jacobi前処理

128ノード 3072x3072



検証2） 現時点のまとめ

- Allreduce, Iallreduceの素性能、および Overlap Testは、
  - メッセージサイズが小さく
  - ノード内プロセス数が「**少ない**」

  SHARPで通信隠蔽時が高速

- PETScの CG with Jacobi では、問題サイズが大きく、
  ノード内プロセス数が「**多い**」場合に
  PIPE CG + SHARPが最速

- 結果に矛盾が見られるため、
  通信隠蔽の効果について、より詳細な解析が必要

2.3 アプリケーションの性能予測について

理化学研究所　R-CCS　南　一生

2.3.1　ルーフラインモデル

プロセッサ単体の実行性能を引き出すチューニングの研究に関連し、プログラムの限界性能を予測する手法として、Williams らは Roofline Model(ルーフラインモデル)を提案した。この中で、ソースプログラムの情報から得られるアプリケーションの要求する operational intensity(Flop/Byte)を用い、ハードウェアの理論メモリバンド幅および CPU 理論ピーク性能から、アプリケーションの限界性能が予測可能であることを示した。ルーフラインモデルの概要を図 1 に示す。

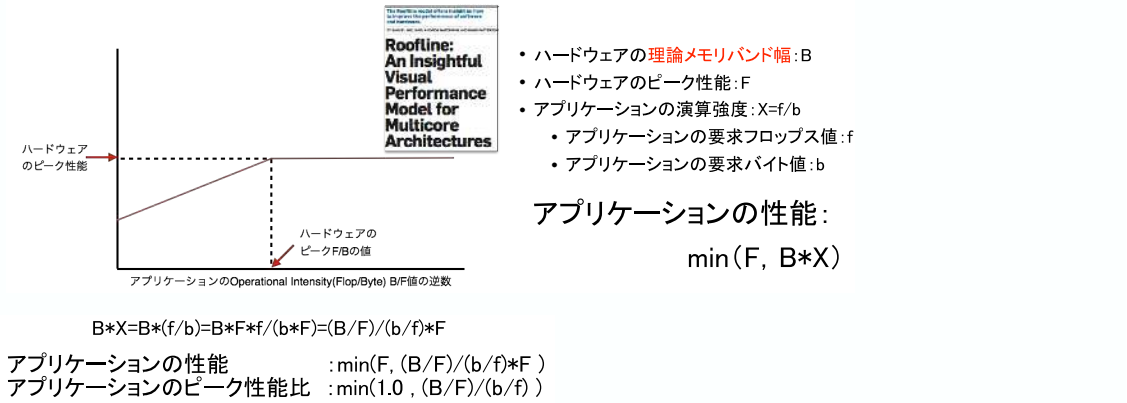


図 1　ルーフラインモデル

我々も、STREAM ベンチマークで測定された実効メモリバンド幅を理論メモリバンド幅の代わりに用いることで、ルーフラインモデルによる性能予測が、実アプリケーションの性能を良く予測できることを確認した。しかしルーフラインモデルは、プログラムの要求 operational intensity(Flop/Byte)の計算において、キャッシュに載っているデータについては考慮しておらず、キャッシュの効果を考慮しないモデルとなっている。実際、以下の図 2 のように性能予測結果と実測性能に乖離が生じることが観測されている。下図において赤枠内のデータは L2 キャッシュアクセスとなっており、ルーフラインモデルによる性能予測値であるピーク性能比 39%に対し実測値は、14.7%となっている。

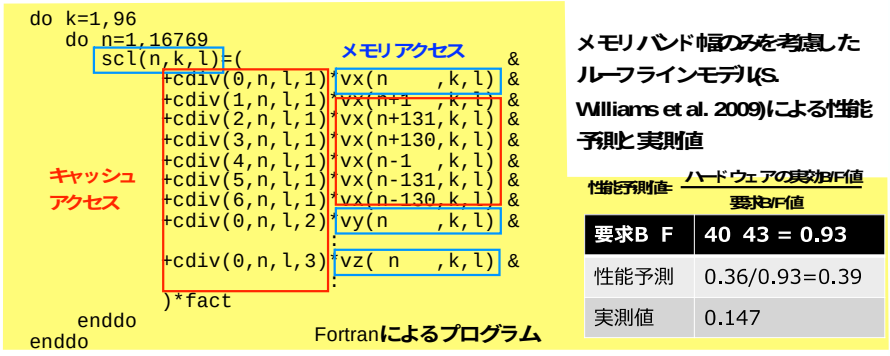


図 2 性能予測結果と実測性能の乖離の例

2.3.2 「京」でのルーフラインモデルの検証

この検証では、配列サイズを変化させることにより、全ての配列がメモリに載った状態(on メモリ)、またはL2 キャッシュに載った状態(onL2)となるテストプログラムを用意した。このテスト要求バイト数を変えないように、演算数を増やし、要求 b/f 値を変化させた。すべての変数が on メモリの場合のメモリ性能テストの結果を図 3 に示す。

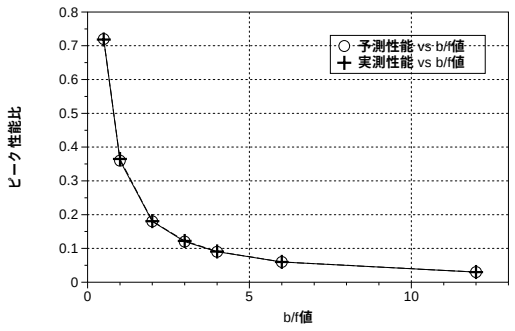


図 3 実効メモリバンド幅を使用したルーフラインモデルの検証(on メモリ)

この結果から、変数がメモリ上にある場合にはルーフラインモデルによる予測値が実測値と良く一致していることが確認された。次に、L2 キャッシュ性能テストの結果を図 4 に示す。

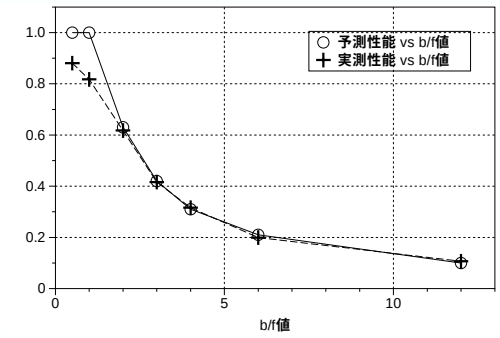


図 4 実効 L2 キャッシュバンド幅を使用したルーフラインモデルの検証(onL2 キャッシュ)

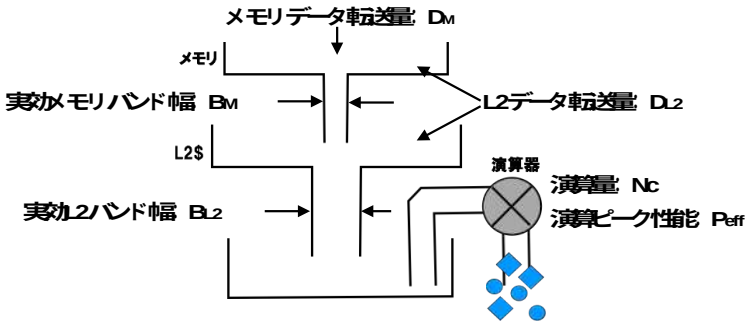
要求 b/f 値が 2.0 以上の L2 キャッシュのバンド幅の平均値から、L2 キャッシュの実効バンド幅を 159.2GB/s とした。実効バンド幅と理論ピーク性能から、L2 キャッシュの理論バンド幅 256GB/s、理論 B/F 値 2.0 を用いれば、L2 キャッシュの実効 B/F 値は 159.2/128=1.24 となる。要求 b/f 値が 1.24 よりも大きい場合に、予測性能と実測性能がほぼ一致していることが分かる。要求 b/f 値が 1.24 より小さい場合は、予測性能がピーク性能になっている。特別にチューニングされた DGEMM のような数学ライプラリでも実効性能としてピーク性能が得られることはなく、ここでも同様に要求 b/f 値が 1.24 よりも小さい場合は、実測性能はピーク性能まで届いていない。このような場合、演算器律速になっており、L2 キャッシュのバンド幅を使い切ることができないため、L2 キャッシュバンド幅の測定値が低くなっている。これらの結果から、L2 キャッシュにおいても L2 キャ

ッシュのデータ供給能力に対し、演算器律速になっていない場合であれば、実効バンド幅を使用することにより、ルーフラインモデルが適用可能であることが明らかになった。

2.3.3 メモリとキャッシュ混合状態での性能限界値予測モデル

2.3.3.1 限界性能予測モデル

実効性能は、プログラムで実行される演算量を実行時間で除した値であるので、実行時間の予測ができれば限界性能の予測が可能となる。メモリ、L2 キャッシュ、演算に対して、メモリデータ転送量、実効メモリバンド幅、L2 キャッシュデータ転送量、実効 L2 キャッシュバンド幅、プログラム演算量、演算実効性能を、それぞれ DM、BM、DL2、BL2、NC、Peff とする。またメモリ律速時の実行時間を tM、L2 キャッシュ律速時の実行時間を tL2、演算器律速時の実行時間を tC とする。メモリ律速時はデータ量を実効メモリバンド幅で割ることにより tM が求められる。L2 キャッシュ律速時も同様である。これらのモデルを図 5 に示す。



メモリデータ移動時間 $t_M=D_M/B_M$

L2データ移動時間 $t_{L2}=D_{L2}/B_{L2}$

最小演算時間 $t_C=N_C/P_{eff}$

実行時間 $t_E = \max\{t_M, t_{L2}, t_C\}$

ピーク性能比 $C_p = N_C/(\max\{t_M, t_{L2}, t_C\} \times P_{eff})$

図 5 メモリとキャッシュ混合状態での性能限界値予測モデル(1)

tE =tM、tE =tL2、tE =tc、のそれぞれの場合の状態の概念図を図 6 に示す。

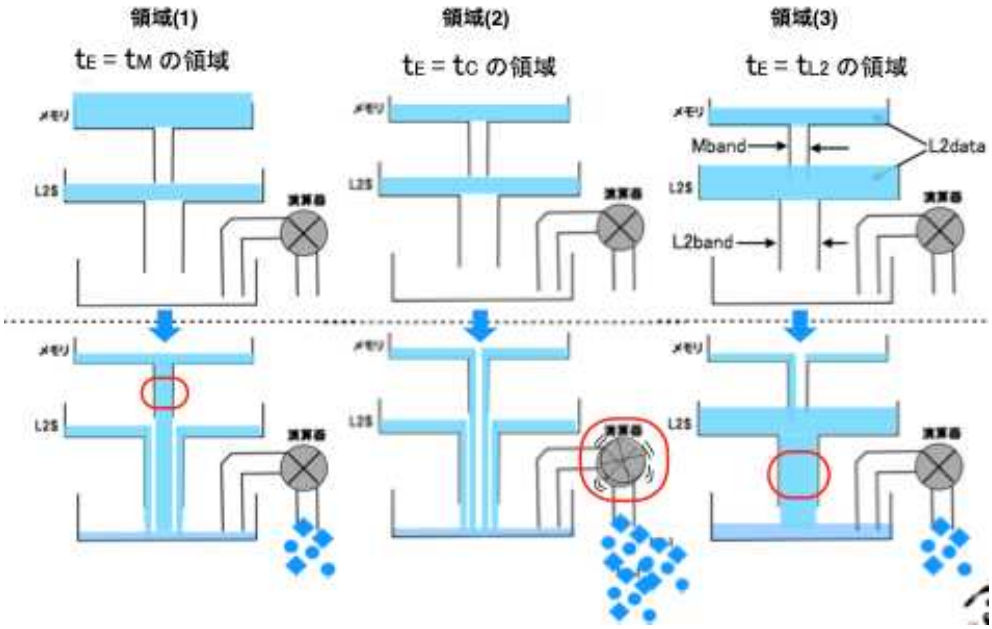


図 6 メモリとキャッシュ混合状態での性能限界値予測モデル(2)

2.3.3.2 ルーフラインモデルとの関係

ルーフラインモデルを 2.3.3.1 に出現した変数を書き換えると図 7 のようになる。この結果と 図 5 を比較すると、従来のルーフラインモデルは、図５のモデルにおいて L2 キャッシュの効果を考慮していないモデルであると言える。

ルーフラインモデル→ 性能$(Flops)=C_p*P_{eff} = min(F, BX)$

F:実効ピーク性能: P_{eff}

B: 実効メモリバンド幅: B_M → $= min(P_{eff}, B_M*N_C/D_M)$

$= min(P_{eff}, N_C/(D_M/B_M))$

X: 演算強度: $f/b=N_C/D_M$

$t_M = D_M / B_M$ → $= min(P_{eff}, N_C/t_M)$

$t_C = N_C / P_{eff}$ → $= min(N_C/t_C, N_C/t_M)$

$= N_C/max(t_C, t_M))$

提案モデルの L2キャッシュを除いた→モデルとなっている { $t_E=max(t_C, t_M)$

図 7 ルーフラインモデルの書き換え

2.3.3.3 「京」での限界性能予測モデルの検証

混合性能テストとして、図 8 に示したプログラムを用いた。このプログラムにおいて、配列:c(i,j-1,k)、c(i,j,k)、 c(i,j+1,k)のうち 1 つはメモリからのロードとなるが、他の 2 つについてはキャッシュに載っている。配列 c の 1 次元目の宣言が、1 から 4000 であるので、配列 c の第 2 添え字に関しては、4000 × 8 = 32kB 離れており、ループ内に a と c の 2 つの配列アクセスが存在することを考えると、 c(i,j+1,k)は L1 キャッシュには載り切らず L2 キャッシュアクセスとなる。メモリ上の a(i,j,k)へのアクセス数を 2 とすると、メモリへの要求バイト数 3×8=24(B)、L2 キャッシュへの要求バイト数 2×8=16(B)、演算数は 2 となるので、メモリへの要求 b/f 値は 12、L2 キャッシュへの要求 b/f 値は 8 となる。3 変数のメモリへのアクセス、2 変数の L2 キャッシュへのアクセス、演算が 2 であるので、カーネルループ名を 3M-2L2-2F と呼ぶこととしこのコードをベースコードとした。このコードに対し、4 行目の代入文の右辺の式に、(右辺)*c(i,j+2,k)+c(i,j-2,k)のように項を追加することにより、L2 キャッシュのアクセスと演算数を増加させた。メモリアクセス数、L2 キャッシュアクセス数、演算数の比を m:n:l とした場合のカーネルループを mM-nL2-lF と呼ぶことにした。

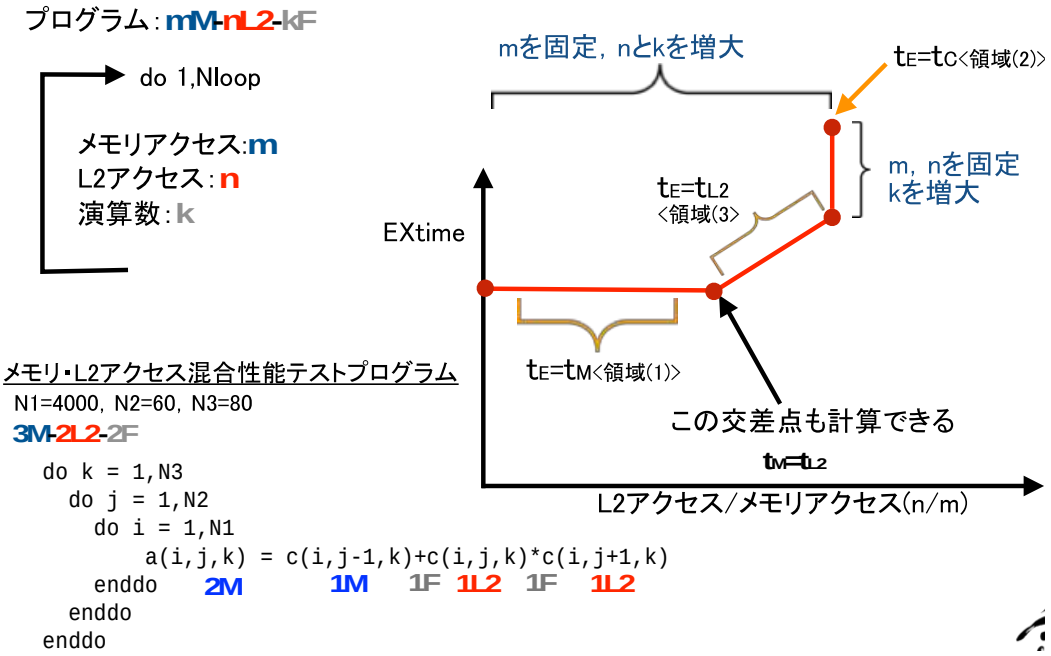
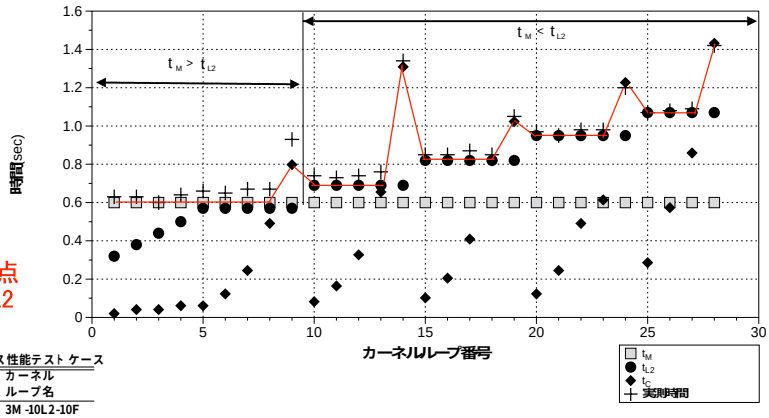


図 8 メモリ・L2 キャッシュ混合テスト

図 8 に示したようにベースコードのメモリへのアクセス数を固定し、L2 キャッシュへのアクセス数、演算数を変化させ測定を行なった。メモリ・L2 キャッシュ混合テストの結果を図 9 に示す。測定に用いたカーネルループ一覧を図 9 に示した。カーネルループ番号の増加と共に、L2 キャッシュアクセス量および演算量が増加している。実効演算性能:Peff は、図 4 の b/f 値 0.5 で得られた実測性能(peak 比)0.88 を用い、Peff = PPEAK× 0.88 で計算した。



計算で求めた交差点
3M-6L2<-->3M-8L2
実測結果と一致

Table 5.3: メモリ・L2キャッシュ混合アクセス性能テスト ケース

カーネル ループ番号	カーネル ループ名	カーネル ループ番号	カーネル ループ名
1	3M-2L2-2F	15	3M-10L2-10F
2	3M-3L2-4F	16	3M-10L2-20F
3	3M-4L2-4F	17	3M-10L2-40F
4	3M-5L2-6F	18	3M-10L2-80F
5	3M-6L2-6F	19	3M-10L2-100F
6	3M-6L2-12F	20	3M-12L2-12F
7	3M-6L2-24F	21	3M-12L2-24F
8	3M-6L2-48F	22	3M-12L2-48F
9	3M-6L2-78F	23	3M-12L2-60F
10	3M-8L2-8F	24	3M-12L2-120F
11	3M-8L2-16F	25	3M-14L2-28F
12	3M-8L2-32F	26	3M-14L2-56F
13	3M-8L2-64F	27	3M-14L2-84F
14	3M-8L2-128F	28	3M-14L2-140F

この結果は「京」の結果であるがインテルのHaswellでも同様の結果が得られている.

11

図 9 メモリ・L2 キャッシュ混合テストの結果<1>

図 9 から、tM>tL2 となっているカーネルループ 1-9 では、実測時間はほぼ tM に一致し、また tM <tL2 となっているカーネルループ 10-28 では、実測時間は、ループ 9、14、19、2428 を除き、ほぼ tL2 に一致していることがわかる。ループ 9、14、19、24、28 は、tL2<tc となっており、実測時間は tc に近い。ただしカーネルループ 9 については実測時間が tc と少しずれている。これは、演算性能については Peff を PPEAK の 0.88 倍としているが、演算性能が、この値まで到達することは保証の限りではなく、多少誤差が生じる傾向にある。ここで、tM>tL2 から tM <tL2 に切り替わる点は、計算することができ、この境目はループ 9 とループ 10 の間となるが、実測値とも一致している。

次に、従来のルーフラインモデルと提案した予測モデルの比較を行う。図 10 は、ピーク性能比のルーフラインモデルによる予測値と実測値をプロットしたものである。図中の括弧つきの数字は、カーネルループ番号である。図 10 左上は、メモリバンド幅律速の場合(tM>tL2)の場合であるが、予測値と実測値がよく一致していることが分かる。一方、図 10 左下は、カーネルループ 10-28 についてのものであるが、tM<tL2 の領域であるため、従来のルーフラインモデルによる予測値と実測値の乖離があることがわかる。図 10 右下は、tM<tL2 の領域に新しい予測モデルを適用した結果であり、予測値と実測値がよく一致していることを示している。

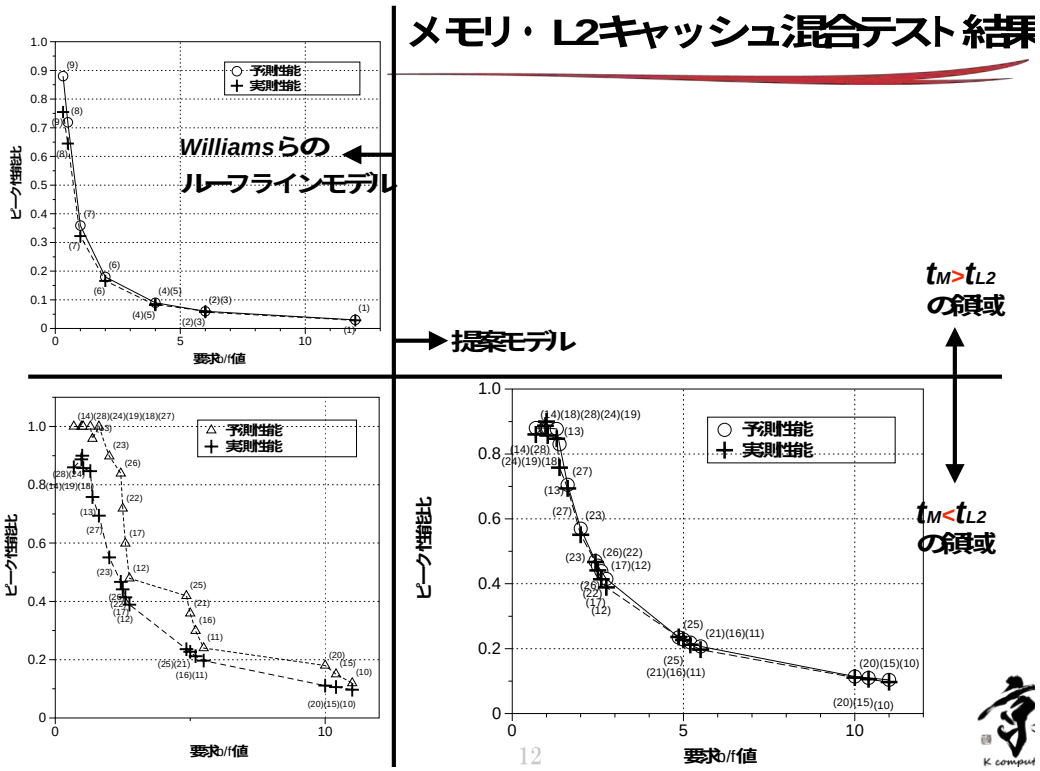


図 10 メモリ・L2 キャッシュ混合テストの結果<2>

2.3.4 実アプリケーション性能予測と評価

実アプリケーションを例として、図 2 のプログラム((A)とする)を含む 4 つのプログラムに対し、本モデルによる性能予測と実測性能を基に、プログラムのチューニングを行った結果について示す。今回の性能予測の目的は、限界性能を予測することであり、実際のアプリケーションで必ずしも限界性能値を得られるわけではない。そこで今までの経験に基づきチューニングによる性能向上の目標は、実測性能が予測性能の 85%程度を達成することとした。性能予測と評価のサマリを図 11 に示す。

						予測性能		tuning 前 (バンド幅:GB/s)			tuning 後 (バンド幅:GB/s)		
	n_M	n_{L2}	n_{L1s}	n_{L1l}	l	従来法 (peak 比)	提案法 (peak 比)	実測性能 (peak 比)	メモリ バンド幅	L2cache バンド幅	実測性能 (peak 比)	メモリ バンド幅	L2cache バンド幅
(A)	5	21	12	6	43	0.387	0.236	0.147	23.0	95.7	0.197	29.7	121.8
(B)	13	2	3	15	60	0.208	0.208	0.129	29.6	55.8	0.193	43.7	70.7
(C)	11	2	0	2	11	0.045	0.045	0.038	45.6	35.8	–	–	–
(D)	3	8	8	0	25	0.375	0.324	0.290	38.4	128.5	–	–	–

図 11 実アプリケーション性能予測と評価

(A)は、L2律速の見積りとなる。

- 従来法の予測より提案法の予測値がL2アクセスを反映し低くなった.
- 最初の実測値は低かったがチューニングにより性能向上。予測値に近づいた。

(B)は、メモリ律速の見積もりとなる。

- 最初の実測値は低かったがチューニングにより性能向上。予測値に近づいた。

(C)は，メモリ律速の見積もりとなる。

- 実測値と予測値はかなり近い値となっている。

(D)は，L2律速の見積もりとなる。

- 従来法の予測より提案法の予測値がL2アクセスを反映し低くなった。
- 実測値と予測値はかなり近い値となっている。

2.3.5 実アプリケーション性能予測と評価

プログラムをチューニングする場合、限界性能を予測することは極めて重要である。ルーフラインモデルを拡張したモデルにより、キャッシュアクセスがある場合でも性能予測ができるようになり、チューニングの際の一つの指標として用いることが出来る。性能予測モデルを実際のプログラムに適用し、提案した性能予測モデルでプログラムの限界性能値を予測し、チューニング実施の判断基準にすることができることを示した。

以上

2.4 富士通 C++コンパイラの最適化機能の改善について

東京大学物性研究所
物質設計評価施設
渡辺宙志

富士通株式会社
次世代テクニカルコンピューティング開発本部
千葉修一

2.4.1 はじめに

HPC 分野においては、Fortran ないし C/C++言語が広く使われている。Fortran、C 言語、C++言語のいずれも活発に仕様が改定され、現在も発展し続けている言語ではあるが、数値計算向けの「普段使いの言語」としては、おそらく Python や Julia などの方が広く使われていると思われる。特に Python は、昨今の機械学習ブームの牽引役として広く使われているようである。今後、Python や Julia といった言語によるスパコン利用も増えていくと思われるし、それにとどまらず Rust のような次世代言語も使われるようになって欲しいと期待もするのだが、スパコンを使うために Fortran/C/C++が必要となる現状は当面続くであろう。

さて、スパコンを使うからには、プログラムは高速に実行されて欲しい。そのためには優秀な最適化機能を備えたコンパイラがあることが望ましい。ところが、Fortran や C 言語はともかく、言語仕様が巨大でプログラムの記述自由度が高い C++言語はコンパイラから見ると最適化が困難な言語である。スパコンユーザは、普段のコード開発を x86 系の PC 上で、GCC/clang もしくはインテルコンパイラを用いて行っていることが多い。x86 上における GCC や clang の最適化機能は高度であり、かなり抽象的なレベルで最適化を行う能力を持っている。また、インテルコンパイラは SIMD ベクトル化や、大域最適化(Link Time Optimization, LTO)に優れている。スパコンユーザはそういった「優秀」なコンパイラを使い慣れているため、富士通 C++コンパイラの挙動に戸惑う場合がある。本 WG では、渡辺が 2017 年 7 月 25 日に富士通の C++コンパイラの問題点と改善の要望を挙げた。また、関連して 2018 年 1 月 24 日にマルチスレッド環境下におけるメモリ管理の問題について報告した。コンパイラについての問題については、富士通の千葉が 2017 年 10 月 20 日と 2018 年 10 月 19 日の二回にわたり、富士通のコンパイラ改善の現状と今後の見通しについて発表した。メモリ管理の問題については、2019 年 2 月 25 日に富士通より報告があったが、こちらの内容については別の報告書にまとめ、本稿では主にコンパイラに対する要望と改善内容について述べ

る。なお、本 WG において渡辺は分子動力学法における力の計算の AVX2、AVX-512 を用いた SIMD ベクトル化手法とその性能向上についても報告を行ったが、本稿において詳細には触れない[1]。ポスト京アーキテクチャは SIMD 幅が 512 ビットになることが発表されており、今後ますます性能におけるベクトル演算の重要性は増していくと思われる。コンパイラによる自動ベクトル化も重要なトピックと思われるが、本稿では C++言語に特有の最適化にフォーカスする。

2.4.2 クラス変数の定義場所に依存する最適化処理

C++言語のようなクラスベースのオブジェクト指向言語においては、いくつかの変数や関数をまとめてクラスとして管理する。
渡辺が開発した分子動力学法コード MDACP[2]では、座標と運動量をまとめて

```cpp
class Variables{
  public:
    double C0;
    double q[N][D],p[N][D];
};
```

という変数クラスを作って管理していた。`N`が原子数、`D`が次元を表し、座標`q`及び運動量`p`をそれぞれ多次元配列として宣言してある。また、`C0`はカットオフに使われる定数である。この`Variable`クラスのインスタンスを受け取り、力の計算をするコードを書いていたのだが、力の計算だけを抜き出したコードは最適化され、満足いく性能が出るにもかかわらず、それとほぼ等価と思われる本番コードは最適化されないことに悩んでいた。本件について調査した結果、コンパイラの最適化機能がクラス内のメンバ変数の定義順序に依存することがわかった。例えば、以下のようなトイコードを考えてみる。本コードでは変数`C0`は使われない。

```cpp
void
calcforce(Variables *vars){
  double (*q)[D] = vars->q;
  double (*p)[D] = vars->p;
  for(int i=0;i<N-1;i++){
    const double qx = q[i][X];
    const double qy = q[i][Y];
    const double qz = q[i][Z];
```

```
    double px = p[i][X];
    double py = p[i][Y];
    double pz = p[i][Z];
    for(int j=i+1;j<N;j++){
      double dx = q[j][X] - qx;
      double dy = q[j][Y] - qy;
      double dz = q[j][Z] - qz;
      double df = (dx*dx + dy*dy + dz*dz);
      px += df*dx;
      py += df*dy;
      pz += df*dz;
      p[j][X] -= df*dx;
      p[j][Y] -= df*dy;
      p[j][Z] -= df*dz;
    }
    p[i][X] = px;
    p[i][Y] = py;
    p[i][Z] = pz;
  }
}
```

これは`Variable`クラスのインスタンスを受け取り、距離から定まる適当な力を計算して運動量に足し込むコードである。
冒頭で座標や運動量のポインタを定義して代入しているが、これは(少なくとも京稼働当初の富士通 C++コンパイラでは)こうしないと最適化がかからなかったためである。
さて、このコードをそのままコンパイラに食わせるとソフトウェアパイプライニング(Software pipelining, SWP)がなされない。
「京」や「FX10」は、豊富なレジスタを活かして力任せにループをソフトウェアパイプライニングすることが前提のアーキテクチャとなっており、ソフトウェアパイプライニングが効かないと性能が出ないことが多い。しかし、ここで`Variable`クラスの宣言順序を入れ替えてみる。

```cpp
class Variables{
  public:
    double q[N][D],p[N][D];
    double C0; // 宣言を後ろにずらした
```

```
};
```

すると先程の力計算のループにソフトウェアパイプライニングが適用され、実行性能が倍以上向上した。
今回のケースをユーザから見ると、「使われないクラスのメンバ変数の定義位置により、プログラムの性能が倍以上変わる」という奇妙な結果となる。
後に、`restrict`修飾されたポインタ間の最適化処理に問題があったのが原因と報告された。
このようにクラスとして保持されたデータのポインタを受け取る、という処理はC++のコンテナでは頻出するイディオムであるため、
同様な理由で性能が出ないユーザコードが他にもあることが想定される。この件については今後改善を検討する旨が報告された。

2.4.3 STLコンテナの最適化について

C++言語においては、生の配列を使わずに`std::vector`といったコンテナを使うのが一般的である。
コンテナに対するループ処理も、ループカウンタではなくイテレータを用いる。コンパイラとして GCC やインテルコンパイラを使っている場合、コンテナを使っても生の配列を使った場合に比べて特に性能は劣化しない。しかし、富士通C++コンパイラはSTLコンテナを使うとループの最適化がうまくできない場合がある。

先程と全く同様なコードを、`std::vector`を使って書き直してみよう。

```cpp
const int N = 20000;
struct pos{
  double x,y,z;
};
void
calcforce(std::vector<pos> &q, std::vector<pos> &p){
  for(int i=0;i<N-1;i++){
    const double qx = q[i].x;
    const double qy = q[i].y;
    const double qz = q[i].z;
    double px = p[i].x;
    double py = p[i].y;
    double pz = p[i].z;
```

```
    for(int j=i+1;j<N;j++){
      double dx = q[j].x - qx;
      double dy = q[j].y - qy;
      double dz = q[j].z - qz;
      double df = (dx*dx + dy*dy + dz*dz);
      px += df*dx;
      py += df*dy;
      pz += df*dz;
      p[j].x -= df*dx;
      p[j].y -= df*dy;
      p[j].z -= df*dz;
    }
    p[i].x = px;
    p[i].y = py;
    p[i].z = pz;
  }
```

このコードはソフトウェアパイプライニングされず、性能が出ない。
可変長配列であることが悪いのかと、固定長配列コンテナである`std::array`を使っても最適化されず、
引数に生配列を渡すと最適化が効き、妥当な性能が出る。なお、GCC、インテルコンパイラともに`std::vector`、`std::array`を使っても生配列を使った場合と遜色ない性能がでることは付記しておきたい。

後に、STL の内部で使われている`volatile`変数が最適化を阻害していることが報告された。こちらについても改善を検討中とのことである。

2.4.4 Ranged-base for についての最適化

`std::vector`その他、要素を多数保持するコンテナクラスについて、その要素全てについてなんらかの処理をする、といったプログラムは頻出する。例えば、`std::vector<int> v`が保持する要素全てを 2 倍したい、という時、従来の for 文を使ったプログラムでは以下のように書けるであろう。

```cpp
for (unsigned int i = 0; i < v.size(); i++) {
  v[i] *= 2;
}
```

```

これは、C++11 から導入された ranged-base for を用いて以下のように書ける。

```cpp
  for (auto &t : v) {
    t *= 2;
  }
```

さて、通常の for を使ったコードと ranged-base for を使ったコードは等価なのであるから、同じように最適化処理がなされて欲しい。

以下のようなコードを書いてみよう。

```cpp
const int N = 10000000;
double a[N];

void
myabs(void){
  for(auto &v: a){
    if(v < 0.0)v = -v;
  }
}
```

単純に、グローバル配列`a`について、その絶対値を取る関数である。このコードをコンパイルさせると、
SIMD 化もソフトウェアパイプライニングも適用されない。通常の for を使ったコードに書き換えてみる。

```cpp
const int N = 10000000;
double a[N];

void
myabs(void){
```
```

```
  for(int i=0;i<N;i++){
    if(a[i] < 0.0)a[i] = -a[i];
  }
}
```

こちらはSIMD化もソフトウェアパイプライニングも適用されており、アセンブリを見ても`fneg,s`や`fcmplted,s`といったSIMD命令が発行されていることが確認できる。

後にこれはループの解釈、とくに終了判定がカウント型(不等式による判定)であるか、終端判定型(等式による判定)であるかの違いが最適化に影響していること、後に改善を検討することが報告された。

2.4.5 構造体に対する最適化

C++では、データのまとまりを構造体として扱うことが多い。
また、演算子オーバーロードを用いて、構造体やクラスをあたかも基本型として演算したりするなど、柔軟な記述ができる。
以下は、二つの整数の組を要素にもつベクトルを表現することを意図した構造体である。

```cpp
struct IntPair{
  int i,j;
};

IntPair add(IntPair x, IntPair y){
  IntPair z = {x.i + y.i, x.j + y.j};
  return z;
}

void
func(int &x, int &y){
  IntPair a = {1,2}, b = {3,4};
  IntPair c = add(a,b);
  x = c.i;
  y = c.j;
}
```

`IntPair`は整数を二つまとめた構造体で、`IntPair add`はその演算子オーバーロードを模した関数である。
関数`func`内では、二つのペアの和を計算し、その要素を参照の形で返却している。
さて、この関数の実行結果は常に`x = 4, y = 6`となるが、最近のコンパイラはそれを認識して、
無駄な構造体を作らず、値を即値で返してしまう。例えば以下は g++ -03 -S の実行結果である。

```asm
func(int&, int&):
        movl    $4, (%rdi)
        movl    $6, (%rsi)
        ret
```

4 と 6 を即値で返していることがわかる。しかし、富士通コンパイラは関数`func`について以下のようなアセンブリを吐く。

```asm
func(int&, int&):
        add     %sp,-240,%sp
        mov     1,%xg3
        mov     2,%xg4
        stw     %xg3,[%sp+2255]
        mov     3,%xg5
        mov     4,%xg6
        mov     3,%xg7
        mov     4,%xg8
        mov     1,%g1
        mov     2,%g3
        add     %g1,3,%g2
        add     %g3,4,%g4
        stw     %xg4,[%sp+2259]
        stw     %xg5,[%sp+2263]
        stw     %xg6,[%sp+2267]
        stw     %xg7,[%sp+2247]
        stw     %xg8,[%sp+2251]
        stw     %g1,[%sp+2239]
```

```
        stw     %g3,[%sp+2243]
        stw     %g2,[%o0]
        stw     %g4,[%o1]
        sub     %sp,-240,%sp
        retl
```

なお、わかりやすいように遅延スロットの場所を変えるなどしている。
このアセンブリを読み解くと、関数のインライン展開は行われたが、関数ローカルに(すなわちスタックに)
作られた構造体が、後に不要だとわかっても削除できなかったことが推定される。
ワークアラウンドとしては、`add`関数の引数を参照渡しに変えたり、構造体にコンストラクタを書くことで
最適化が最後まで通り、富士通コンパイラでも即値で返せるようになる。
しかし、それを知らないユーザから見ると、自分の開発環境では全く問題にならなかった場所に余計なコードが大量に生成されており、知らないうちに性能が劣化している、ということになる。この現象をコンパイルメッセージから読み解くのは難しく、性能劣化に気づいた後にアセンブリを詳細に読み解かなければ、何が起きているかを把握できないであろう。

これについては、引数で受けた構造体に関する最適化処理が終盤で行われているため、そのステージでは
ソースコードの情報が欠落しており、最適化が十分に適用されなかった旨が報告された。

2.4.6 リンク時最適化について

　これはC++言語に限らないことだが、大きなプログラムは一般に分割コンパイルが行われる。
分割コンパイルとは、更新のあったソースコードのみオブジェクトファイルを更新し、既にコンパイル済みの他のオブジェクトファイルとリンクをして実行ファイルを作成することで、全体のビルド時間の短縮をはかるものである。
数値計算、特に HPC 分野で使われるコードは大規模化が進んでおり、フルビルドに何時間もかかるものが珍しくない。
したがって、分割コンパイルを行うわけだが、この時に問題となるのがプロシージャ間の最適化である。簡単な例を挙げよう。

整数を受け取って、その絶対値を返す関数`adjust`を`adjust.cpp`に実装する。

```cpp
```

```
int adjust(int a) {
  if (a < 0)return -a;
  else return a;
}
```

他のファイルから使えるように、関数宣言を`adjust.hpp`に書いておく。

```cpp
#pragma once
int adjust(int a);
```

これを`main`関数から以下のように呼んで見よう。ファイルは`main.cpp`とする。

```cpp
#include <cstdio>
#include "adjust.hpp"

int main() {
  printf("%d¥n", adjust(-5));
}
```

さて、このコードをコンパイルする時には`adjust`がどんな関数かわからないので、ただ`call`するしかない。g++でコンパイルした際、対応するアセンブリはこうなる(関数名が見やすいようにc++filtをかけてある)。

```sh
g++ -03 main.cpp adjust.cpp
```

```asm
        movl    $-5, %edi
        call    adjust(int)
        leaq    lC0(%rip), %rdi
        movl    %eax, %esi
        xorl    %eax, %eax
```

```
        call    _printf
```

しかし、このコードを`-flto`オプションをつけてコンパイルすると g++は即値を返せるようになる。

```sh
$ g++ -O3 -flto main.cpp adjust.cpp
$ objdump -S ./a.out
(snip)
0000000000400500 <main>:
  400500:       48 83 ec 08             sub    $0x8,%rsp
  400504:       be 05 00 00 00          mov    $0x5,%esi
  400509:       bf 74 07 40 00          mov    $0x400774,%edi
  40050e:       31 c0                   xor    %eax,%eax
  400510:       e8 cb ff ff ff          callq  4004e0 <printf@plt>
  400515:       31 c0                   xor    %eax,%eax
  400517:       48 83 c4 08             add    $0x8,%rsp
  40051b:       c3                      retq
(snip)
```

`call`による関数呼び出しが消え、`adjust(-5)`の実行結果である 5 がそのまま`printf`に渡されているのがわかる。
これは、リンク時最適化(Link time optimization, LTO)と呼ばれ、オブジェクトファイルを生成する際に中間情報を保存しておき、リンク時にあらためて最適化をかけてから実行バイナリを作るものである。

上記のコードは、たとえば分子動力学法において周期境界条件を補正するためのコードを模したものである。
最近のコンパイラは賢いこともあり、C++プログラマは、「この程度ならインライン展開されるだろう」と期待して、小さな関数をたくさん作ることがある。しかし、コンパイラがLTO に対応していないと、これらは全て関数コールになってしまう。特にループの中で何回も呼び出される関数がインライン展開されないと大きな実行ペナルティとなる。

LTO に対応していないコンパイラを用いる場合の簡単なワークアラウンドとしては、関数の実体をヘッダに書いてしまう、という方法が挙げられる。先程の例であれば`adjust.hpp`の中に関数定義を書いてしまえば富士通コンパイラも即値を返せるようになる。しかし、

なんでもかんでもヘッダファイルに入れてしまうと分割コンパイルの意味がなくなってしまう。
また、いちいち「この関数はたくさん呼ばれるからヘッダファイルに入れよう」などと考えるのも面倒である。

本 WG での発表時点では富士通コンパイラは LTO に事実上未対応であった。実際には C コンパイラは LTO オプションがあったのだが、あまり有効に機能していなかった。渡辺から LTO に対応して欲しいと要望を伝えたところ、千葉より LTO が有効に機能するベンチマークがあること、コードの中小構文木の情報をオブジェクトファイルに埋め込むことで、ポスト京では LTO に対応する予定である旨の報告があった。

2.4.7 マルチスレッド環境下における malloc 性能について

渡辺が開発した分子動力学法コードは、並列化に疑似 Flat-MPI 法という手法を採用している。これは Flat-MPI の計算と MPI/OpenMP ハイブリッド計算が、通信部分を除いて全く同じ内容になるという特徴を持っている。したがって、通信時間が無視できる条件においては、Flat-MPI とハイブリッド計算は同じ計算資源を使うならば同じ結果を与え、かつ計算時間もほとんど変わらないはずであった。事実、別のスパコンサイトでは両者に差は見られなかったが、「京」においてはシングルノードで実行した場合でもすべてをプロセス並列した場合と、スレッド並列した場合で、有意にマルチスレッド実行の方が遅い現象に悩まされていた。その後の調査により、スレッド並列で実行される関数内部で使われている`std::vector`が性能劣化の要因であることがわかった。問題となる関数を簡略化して抜き出すと以下のような処理となる。

```cpp
#pragma omp parallel for
for(int i=0;i < thread_num; i++){
  func(i);
}
```

上記のように、スレッドごとに異なる引数で関数が呼ばれている。この関数の冒頭で、以下のように`std::vector`が宣言され、利用されている。

```cpp
void func(int thread_index){
  std::vector<int> v;
  //...
```

}
```

分子動力学法のホットスポットは力の計算であるが、このケースも 90%以上が力の計算に費やされており、上記の場所は別のサイトであれば全体の計算時間に比べて無視できるほどのコストしかかからないはずである。しかし、「京」でマルチスレッド実行すると上記の処理が極端に遅くなり、全体で 10%程度の性能劣化が見られた。

しかし、この箇所を、

```cpp
void func(int thread_index){
  thread_local std::vector<int> v;
  v.clear();
  //...
}
```

と、スレッドローカル宣言することで性能が改善し、Flat-MPI の場合とほぼ同じ実行時間となった。

C++言語を利用するプログラマのほとんどは、STL (Standard Template Library)を用いる。中でも、C 言語の配列を置き換える`std::vector`の利用頻度は非常に高い。`std::vector`は、これまでプログラマが陽に行ってきたメモリ管理を隠蔽し、メモリの確保や解放を自動で行うことができる。その際、内部では`malloc`や`free`といったメモリ確保、解放の関数を呼び出している。`std::vector`が関数のローカル変数として使われていると、毎回`malloc`/`free`が呼び出されることになる。しかし、`thread_local`宣言をすることで`std::vector`は暗黙に`static`修飾され、今回の実行条件では`free`はプログラムの最後に一度だけ呼ばれるのみとなる。マルチスレッド実行では共有メモリとなることから、Flat-MPI 実行に比べ、`malloc`はより大きなメモリプールについて管理を行う必要がある。これがマルチスレッド実行時の性能劣化原因であろうと推定される。

上記の問題の他、高並列時、およそ数百ノード規模の計算を行った場合、Flat-MPI 実行に比べてハイブリッド並列実行が 20%程度性能劣化すること、関連して、富士通独自の環境変数の役割が分かりづらいことについて報告した。性能劣化については完全な原因究明にはいたっていないが、環境変数の設定変更による性能変化の原因についてや、富士通独自の malloc の環境変数については富士通小田和が報告書をまとめているので参考にされたい。
```

2.4.8 まとめ

本稿では、主に富士通 C++コンパイラへの要望と、それに対する富士通の対応予定について述べた。本 WG で指摘された問題点のほとんどは、ポスト京において改善される見通しである。よく言われることであるが、最適化について「銀の弾丸」は存在しない。一般論としては「環境を変えてもコンパイラがちゃんと最適化してくれるのでコードの可搬性が保たれる」というのは幻想に過ぎず、コードのホットスポットはハードウェアに応じてほぼ書き直す必要があるのが現状である。しかし、だからといってコンパイラの最適化機能はおざなりにしてはならない。むしろ、プログラマがホットスポットのチューニングに注力するためには、コンパイラが「露払い」として、それ以外の部分をきっちり最適化できなければならない。ポスト京を取り巻く環境を概観すると、コアの増加、NUMA、SIMD 幅の増加といったハードウェアの複雑化に伴ってプログラマの負担は増え、プログラミング言語 C++も C++11、14、17 と機能が追加されていき、コンパイラ開発陣の負担も大きくなっている。プログラマも「コンパイラの最適化が甘い」と愚痴るのみならず、出力されたアセンブリを確認して「どんなコードを吐いて欲しいか」を開発に伝え、開発もそのフィードバックを受けてコンパイラを改善することで、ユーザと開発が両輪となり、より良い HPC 環境を構築していくことが望ましい。

参考文献

[1] "SIMD vectorization for the Lennard-Jones potential with AVX2 and AVX-512 instructions", H. Watanabe and K. M. Nakagawa, [Comput. Phys. Commun., Vol. 237, pp. 1 (2019)](https://doi.org/10.1016/j.cpc.2018.10.028)

[2] https://github.com/kaityo256/mdacp

2.5 「京」におけるマルチスレッド malloc の問題について

富士通株式会社 次世代テクニカルコンピューティング開発本部 小田和友仁

2.5.1 はじめに

「京」のメモリ管理ライブラリは、デファクトスタンダードである glibc のメモリ管理をベースに、数十 MiB～数 GiB の大規模連続メモリを使用する HPC アプリケーションに特化した最適化を実施している。さらに、チューニングオプションを用意し、チューニングガイドで挙動や選択指標などを提示してきた。しかし、メモリ利用のパターンは多様でありチューニングの指標も多様であることから、コミュニティでチューニングに関するノウハウが共有され蓄積されることが望ましい。そのためにも、利用者の声をフィードバックしチューニングガイドの成熟を図っていく。

2018 年 1 月 24 日の WG において、東京大学物性研究所の渡辺より「京」のメモリ管理ライブラリの利用に関する問題提起があった（『富士通 C++コンパイラの最適化機能の改善について」の付録『「京」におけるマルチスレッド malloc』スライド）。これに対し、2019 年 2 月 25 日の WG で富士通より見解を報告し（本書付録『「京」におけるマルチスレッド malloc」疑問点について』スライド）、改善について議論した。本報告書では、議論の結果として富士通のチューニングガイドの改善について述べる。

2.5.2 経緯と問題提起

短距離古典分子動力学方コード「MDACP」は、「京」での実行のために MPI(プロセス並列)と OpenMP(スレッド並列)を組み合わせたハイブリッド実行方式を実装に採用している。この実装において、x86 系のクラスタではプロセス並列実行とハイブリッド実行では性能差がほぼないのに対し、「京」や FX10 ではハイブリッド実行が 20%程度遅かった（図 1）。

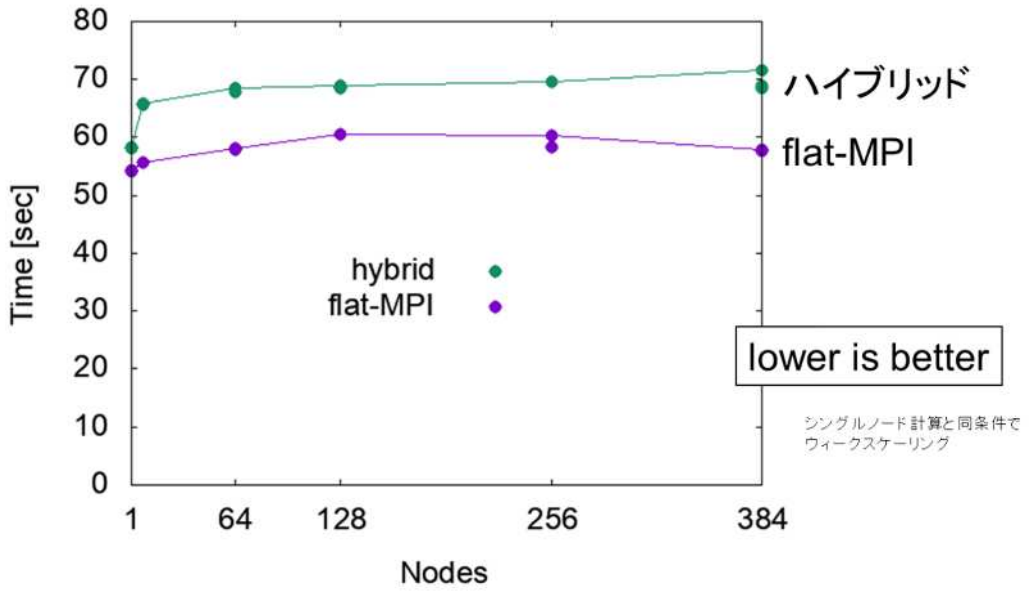


図 1 「京」におけるハイブリッド実行と flat-MPI 実行の実行時間比較

「京」においてプロセス実行に対しハイブリッド実行で遅くなったのは、パラレル処理が終了した後のシリアル処理部であった。このことから、スレッド並列時のメモリ獲得・解放（malloc/free）の繰り返しによりプロセスヒープの空きメモリ管理リストが長くなったことで、シリアル処理部でのメモリ獲得時の空きメモリ管理リストのデフラグ（malloc_consolidate）に時間がかかることが原因と推測した。

改善のため、スレッド毎に独立したヒープ（スレッドヒープ）を利用することで、各ヒープの空きメモリ管理リストが短くなることを期待し、XOS_MMM_L_ARENA_LOCK_TYPE=0（スレッドヒープを作成する）を指定し実行した（表 1）。その結果、デフォルトである XOS_MMM_L_ARENA_LOCK_TYPE=1（スレッドヒープを作成しない）の場合に比べて、さらに処理時間が増加してしまった（図 2）。このことから、スレッド並列においてメモリ獲得性能優先である XOS_MMM_L_ARENA_LOCK_TYPE=1 に変更しても遅くなることがある点が問題提起された【問題 1】。

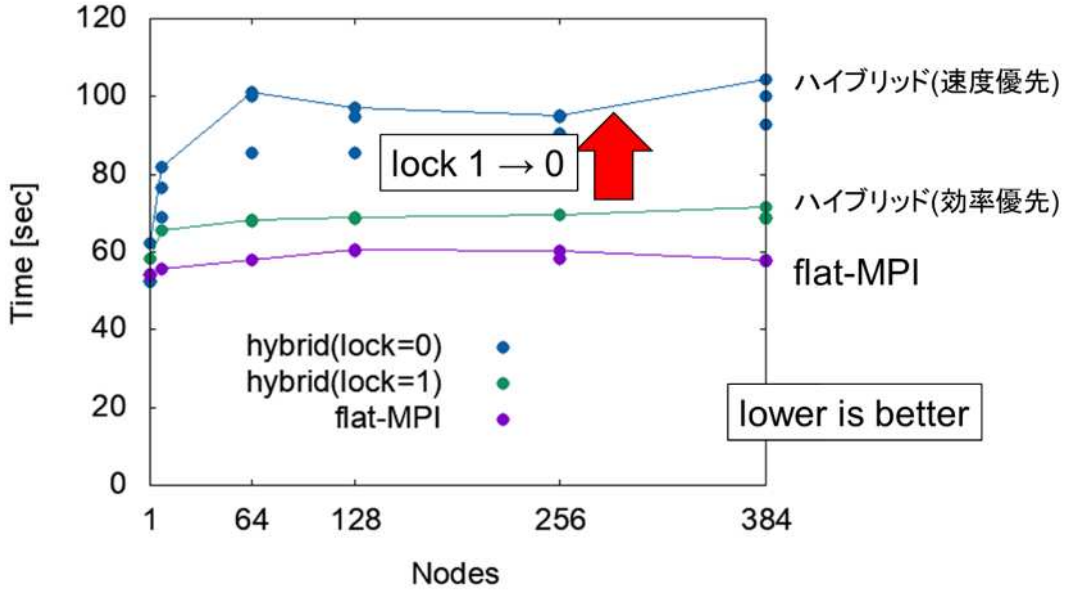


図 2　XOS_MMM_L_ARENA_LOCK_TYPE=0 による効果

表 1　「京」のチューニングガイドより XOS_MMM_L_ARENA_LOCK_TYPE の説明

変数名（デフォルト値）	意味	用途
XOS_MMM_L_ARENA_LOCK_TYPE（1）	0：メモリ獲得性能優先（malloc を並列処理） 1：メモリ使用効率優先（malloc をシリアル処理）	メモリ割り当てポリシーに関する設定です。「0」はメモリ獲得性能優先、「1」はメモリ使用効率優先を意味します。デフォルト値は「1」。「0」「1」以外の値は「1」とみなします。malloc 要求が複数同時に呼び出された場合、「0」は malloc 要求を並列処理するため mmapped アリーナを新規作成します。このため、メモリ使用効率が低下することがあります。「1」は逐次処理するため既存アリーナの空き検索時間が増加することがあります。

XOS_MMM_L_ARENA_LOCK_TYPE=0（スレッドヒープ作成）による性能低下の原因を、各スレッドでのメモリ獲得・解放（malloc/free）の繰り返しにより、各スレッドヒープの作成・解放（mmap/munmap）が繰り返されるためと推測した。

改善のため、作成したスレッドヒープを解放せずに再利用することを期待して、環境変数 XOS_MMM_L_ARENA_FREE=2 も合わせて指定し実行した（表 2）。その結果、スレッドヒープは使用せずにプロセスヒープのみを使用する場合（XOS_MMM_L_ARENA_LOCK_TYPE=1）と同等の処理時間となった（図 3）。このことから、XOS_MMM_L_ARENA_FREE と他の環境変数との関係が不明瞭である点が問題提起された【問題 2】また、XOS_MMM_L_ARENA_FREE という名称が機能を直感的に表していない点が問題提起された【問題 3】

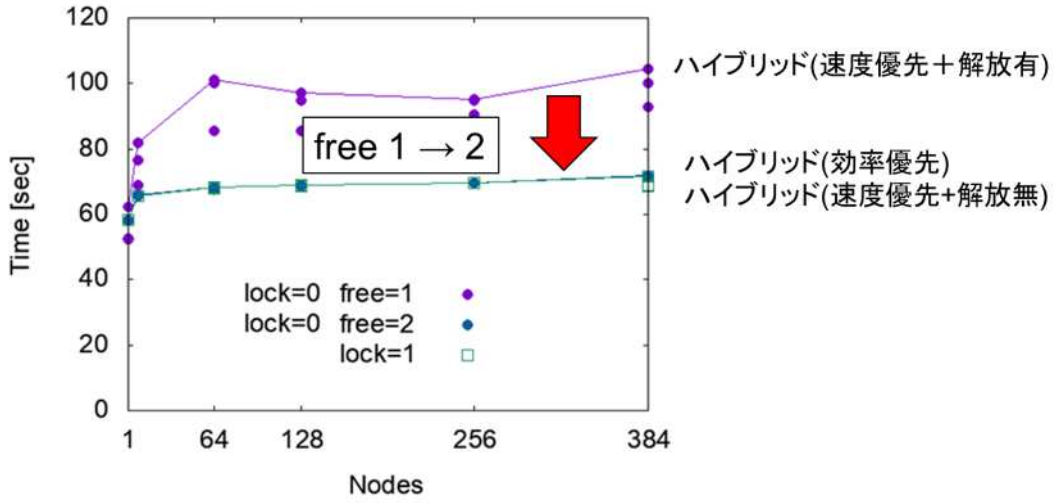


図 3　XOS_MMM_L_ARENA_FREE=2 による効果

表 2　「京」のチューニングガイドより XOS_MMM_L_ARENA_FREE の説明

変数名（デフォルト値）	意味	用途
XOS_MMM_L_ARENA_FREE（1）	1:free 時に解放可能なメモリページを即時解放 2:メモリページを解放しない	free または cfree で解放されたヒープ領域の扱いに関する設定です。「1」はメモリ使用効率を優先、「2」はメモリ獲得性能優先を意味する。デフォルト値は「1」。「1」「2」以外の値は「1」とみなします。 「1」の場合、mmapped チャンクに確保された領域は即時解放される。128MiB 未満のサイズのメモリ要求はヒープ領域から割り当てられ、即時解放の対象となりません。ヒープ領域では保持している空き領域のサイズが 128MiB を超えた場合に解放処理を行います。 「2」は 128MiB 以上のサイズのメモリ要求もヒープ領域から割り当て、解放後も空きメモリ領域として保持します。ヒープ領域の解放処理は行わない。メモリ使用効率低下防止のため単一のヒープ領域で全てのサイズのメモリ割り当てを行う必要があるため、スレッド固有ヒープは使用しません。

以上のように、メモリ管理ライブラリの内部動作まで踏み込んでチューニングを試みたが、富士通から提示されたチューニングガイドのメモリ管理ライブラリの情報記載には不明瞭な点があり、内部動作の解析を必要としたり、仕様誤解を招いたりする原因となった。ポスト「京」に向けては、HPC 向けの機能の明瞭な情報が顧客に流通し、顧客間でもノウハウの共有が活発に行われることが期待される。今後の富士通としての情報流通の方針が問題提起された【問題 4】

2.5.3 問題に対する議論

提起された問題は以下のとおりである。本章ではこれら問題に対する議論の結果を述べる。

【問題 1】スレッド並列でメモリ獲得性能優先である XOS_MMM_L_ARENA_LOCK_TYPE=0 を指定すると遅くなる場合がある

【問題 2】XOS_MMM_L_ARENA_FREE と他の環境変数との関係が不明瞭

【問題 3】XOS_MMM_L_ARENA_FREE という名称が機能を直感的に表していない

【問題 4】今後の富士通としての情報流通の方針

2.5.3.1 【問題 1】スレッド並列でメモリ獲得性能優先である XOS_MMM_L_ARENA_LOCK_TYPE=0 を指定すると遅くなる場合がある

複数スレッドで 128MiB 未満のメモリ獲得・解放（malloc/free）を繰り返すようなアプリケーションでは、XOS_MMM_L_LOCK_TYPE=0 を指定した場合にスレッドヒープ伸縮の繰り返しが発生し、OS カーネルへのメモリ獲得開放（mmap/munmap）が発生することから、性能低下に繋がる可能性がある。

「京」ではヒープの収縮はアプリケーションからのメモリ解放によりヒープ領域の終端に 128MiB 連続の空き領域ができた時に発生する。各スレッドがメモリ獲得開放を繰り返すような状況においては、単一のプロセスヒープを共有する XOS_MMM_L_LOCK_TYPE=1 よりも各スレッドがスレッドヒープを持つ XOS_MMM_L_LOCK_TYPE=0 の方が、ヒープ伸縮回数が増加する可能性が高い。XOS_MMM_L_LOCK_TYPE=1 では複数スレッドで空き領域を取り合うため、ヒープ終端の 128MiB がどのスレッドにも利用されずにすべて空き領域となるタイミングは限られる。これに対し、XOS_MMM_L_LOCK_TYPE=0 では各スレッドが各スレッドヒープを独立利用することから、スレッドヒープに空きメモリがない状況では獲得・解放するたびに伸縮が繰り返されてしまう。

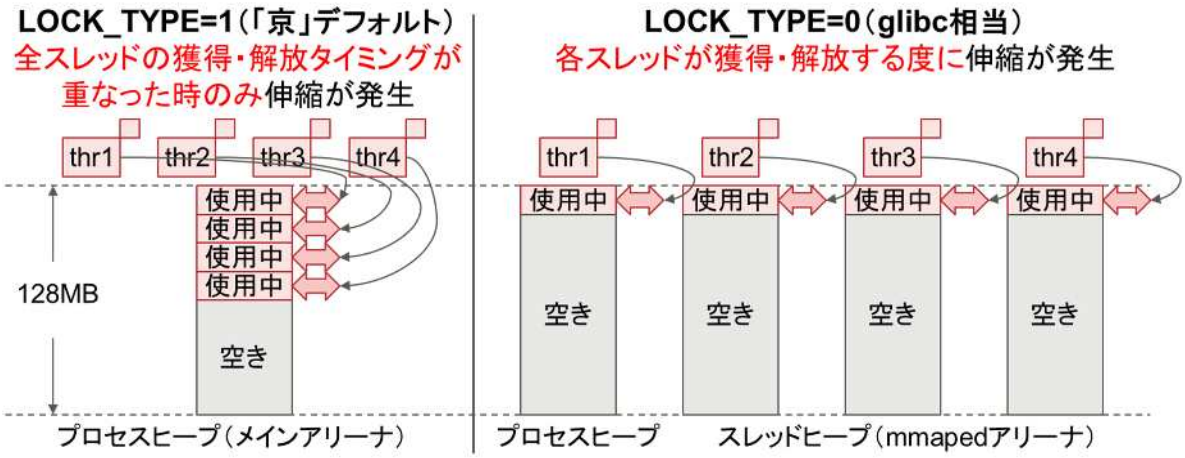


図 4　XOS_MMM_L_ARENA_LOCK_TYPE による終端の伸縮頻度の違いの例

上記のような状態の場合、表 3 の環境変数の組み合わせの指定によりスレッドヒープ伸縮の繰り返しの防止が期待できる。glibc の malloc では MALLOC_TRIM_THRESHOLD_はプロセススレッドの縮小のみに関わるが、「京」の malloc ではスレッドヒープの解放判定にも組み込み、縮小を抑止するようにしている。

表 3　スレッド毎にヒープ利用かつメモリページを開放しないための環境変数設定

環境変数	設定値	目的
XOS_MMM_L_ARENA_LOCK_TYPE	0	スレッド毎にヒープを作成
MALLOC_MMAP_THRESHOLD_	ULONG_MAX	mmap でメモリ獲得する閾値を最大とすることで、mmap を使用せず常にヒープからメモリ割り当て
MALLOC_TRIM_THRESHOLD_	ULONG_MAX	ヒープ縮小の閾値を最大とすることで、ヒープを縮小せずメモリ開放しない

今回のプログラムでもスレッドヒープ伸縮の繰り返しが発生している可能性を疑い、上記の環境変数の適用を試みた。結果として、XOS_MMM_L_ARENA_LOCK_TYPE =0 のみ設定した場合（Hybrid(lock=0)）と、さらに MALLOC_MMAP_THRESHOLD_=ULONG_MAX も指定した場合（Hybrid(lock=0+TH)）、さらに MALLOC_TRIM_THRESHOLD_=ULONG_MAX も指定した場合（Hybrid(lock=0+TH+TRIM)）はほぼ変わらぬ時間となり、効果は得られなかった（図 5)。

以上より、今回の取り組みにおいては、「京」のメモリ管理ライブラリの観点から改善の可能性を探り、環境変数の組み合わせ選択でのチューニングを試みたが、最終的にデフォルト設定を超えるような改善は果たせなかった。これ以上のチューニングには、詳細なソース解析とプロファイラによる動作解析に基づくミクロな視点からの原因調査が必要となる。

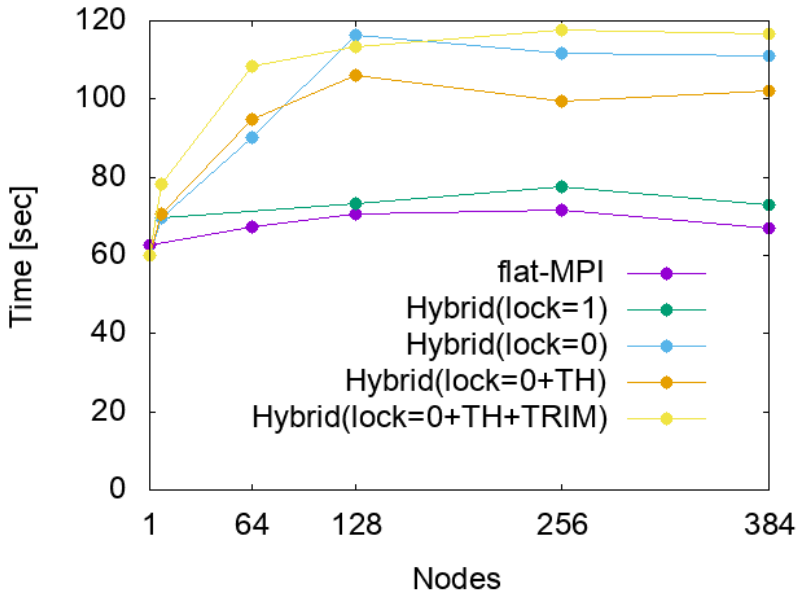


図 5　スレッド毎にヒープ利用かつメモリページを開放しないための環境変数設定の効果

2.5.3.2　【問題 2】XOS_MMM_L_ARENA_FREE と他の環境変数との関係が不明瞭

XOS_MMM_L_ARENA_FREE は、複数スレッドで 128MiB 未満のメモリ獲得・解放(malloc/free)を繰り返すようなアプリケーションのメモリ獲得性能を改善するための環境変数設定の組み合わせを一つの環境変数で指定可能としたものである。

表 4 に XOS_MMM_L_ARENA_FREE=2 指定時に上書きする環境変数と設定値を示す。XOS_MMM_L_ARENA_FREE=1 を設定した場合、各環境変数についてユーザ指定に従い動作する。XOS_MMM_L_ARENA_FREE=2 を設定した場合、各環境変数を上書き設定する。これら環境変数に対し直交しないのはこのためである。

チューニングガイドにこれらの情報が明記されていなかったことが、混乱を招く原因となった。ポスト「京」ではこの環境変数は互換のために残すが、これらの情報を明記し仕様誤解を防ぐ。また、提供する環境変数について、他の環境変数との関係性が明確になるよう、記載を見直す。

表 4　XOS_MMM_L_ARENA_FREE=2 指定時に上書きする環境変数と設定値

環境変数	上書き設定値	目的
XOS_MMM_L_ARENA_LOCK_TYPE	1	プロセスヒープのみ使用
XOS_MMM_L_MAX_ARENA_NUM	1	スレッド毎にヒープ作成しない
MALLOC_MMAP_THRESHOLD_	ULONG_MAX	mmap でメモリ獲得する閾値を最大とすることで、mmap を使用せず常にヒープからメモリ割り当て
MALLOC_TRIM_THRESHOLD_	ULONG_MAX	ヒープ縮小の閾値を最大とすることで、ヒープを縮小せずメモリ開放しない

2.5.3.3 【問題 3】XOS_MMM_L_ARENA_FREE という名称が機能を直感的に表していない

「アリーナ（スレッドヒープ）の解放」に関する設定を想起する環境変数名であるが、実際にはプロセスヒープの空きメモリ解放に関する設定であり、動作仕様に対して直感的ではない。

このような名称となったのは、動作仕様の変更によりアリーナ（スレッドヒープ）との関係が薄れたが、互換のために名称変更しなかったためである。

「京」の運用開始時は mmaped アリーナ(スレッドヒープ)を作成(XOS_MMM_L_LOCK_TYPE=0)し、メインアリーナ（プロセスヒープ）および mmaped アリーナ（スレッドヒープ）の空きメモリを解放しない（MALLOC_TRIM_THRESHOLD_=ULONG_MAX）仕様であった。空きメモリを解放しないことで mmaped アリーナ自体の解放をしないことが主目的だったため、「ARENA_FREE」と名付けた。しかし、各スレッドヒープに空きメモリが分散したことで 128MiB 以上の大規模連続メモリ獲得に失敗する問題が多く発生したことから、mmaped アリーナ（スレッドヒープ）を作成しない動作（XOS_MMM_L_LOCK_TYPE=1）に変更した。これにより、メインアリーナ（プロセスヒープ）の空きメモリを解放しない、という仕様だけが残った。すでに提示済みの環境変数名であったこと、アリーナでのメモリ解放（FREE）に関する設定ともとれることから、互換のために変更をしなかった。

ポスト「京」では、「京」との互換のために環境変数 XOS_MMM_L_ARENA_FREE はそのままの名称で残す。また、環境変数の組み合わせで設定することが望ましいことから、名称を改善した同等の環境変数を設けることはしない。

2.5.3.4 【問題 4】今後の富士通としての情報流通の方針

「京」では、HPC 向けの独自アーキテクチャに対し最適化されたメモリ管理を実現するにあたり、OS カーネルも改造し、独自実装で実現した。その結果、富士通の提示するチューニングガイドが唯一の情報源であり、チューニング情報の不足が問題となった。

ポスト「京」では、富士通の独自技術による価値をいち早くお客様に提供するとともに、基盤となるソフトウェアについては汎用的な使い勝手も重視し、汎用ディストリビューションをそのまま利用できることを目指している。最終的には、富士通が開発した HPC 向けの拡張機能も、OSS として取り込まれ、コミュニティとして情報流通していくことを目指す。そのためにも、ユーザに積極的に HPC 向けの拡張機能を活用していただけるよう、チューニングガイドの改善を図っていく。

メモリ管理ライブラリのチューニングガイドでは以下のような改善を図り、インターネットなどで入手できる glibc のメモリ管理ライブラリに関する情報も合わせて活用可能とする。

- ベースである glibc からの改造ポイントを明記
- 独自環境変数は、目的と選択指針に加え、他の環境変数（特に glibc オリジナル）との関係も明記

また、メモリ管理ライブラリの設計について、OSS 取り込みを意識し以下の観点を徹底す

る。

- 一般コミュニティに受け入れられる直感的な機能名や環境変数名とする
- 新規オプション作成時に既存機能の自然な拡張とし、不必要に直交性を絶たないこと

2.5.4 おわりに

本報告書では、WG で提起された「京」のメモリ管理ライブラリの利用に関する問題提起に対する議論の結果として、不明瞭な環境変数の仕様に対する対応方針と、チューニングガイドの改善方針について報告した。議論の中で、チューニングに苦労した原因として、情報の流通性が根本の問題としてあがった。ポスト京では、汎用ディストリビューションがそのまま利用できることを目指し、富士通の改善機能も OSS に取り込まれ、コミュニティとして情報流通していくことを目指す。

3.1　JAXA LBM コードの性能測定および改善

宇宙航空研究開発機構　石田　崇
富士通株式会社　三吉　郁夫

3.1.1　はじめに

航空宇宙分野，特に航空分野の数値流体力学（Computational Fluid Dynamics ： CFD）においては，巡航状態の航空機翼面上で発生する衝撃波を精度良く捉えるために，Navier-Stokes 方程式をベースとした計算アルゴリズムが発展してきた．非構造格子法，近似リーマン解法，陰的時間積分，乱流モデルといった計算アルゴリズムの進展に加えて，大型計算機の性能向上によって，航空機の巡航状態における空力性能を精度良くかつ実用的な解析時間で予測できるようになってきたため，CFD は風洞試験と並んで航空機設計開発には欠かせないツールとしての地位を確立している．近年では CFD を航空機のオフデザイン（バフェットなどの非定常状態）における空力性能予測に適用するニーズが増えてきている．しかしながら非定常状態に対する Navier-Stokes 方程式をベースとする計算アプローチでは，計算精度・計算コスト・物理モデル依存の観点でまだ課題が多く，効率よく解析することが難しい．このような状況を打破するアプローチとして，近年格子ボルツマン法（Lattice Boltzmann Method ： LBM）が注目されている．LBM では支配方程式がボルツマン輸送方程式であり，以下に示す特徴を持つ．

・支配方程式に非線形項が無い，
・支配方程式は非圧縮 Navier-Stokes 方程式に漸近するが，音の伝搬を直接計算できる，
・衝突（Collision）と並進（Stream）の 2 ステップで時間発展する，
・時間積分は CFL 数が 1 に固定された陽解法である，
・Navier-Stokes 方程式に比べて計算コストが小さい，
・格子点毎に依存関係が無いので容易に並列化が出来る．

JAXA では航空機の離着陸形態における空力騒音問題や低速バフェットの解析に向けた大規模非定常流体解析コードとして LBM コードを開発・整備しており，本稿ではメニーコア WG 内で行った FX100 を用いたコードの性能評価および改善について報告する．

3.1.2 JAXA LBM コードの概要

JAXA LBM コードでは，直交格子法の一つである Building-Cube Method (BCM)をフレームワークとして採用し，LBM と組み合わせて大規模非定常解析コードとして整備している．以降では，LBM および BCM の概要について説明する．

3.1.2.1 LBM のアルゴリズム概要

LBM における支配方程式は，

$$\frac{D}{Dt}f(\mathbf{x},\mathbf{e},t)=\left[\frac{\partial}{\partial t}+\mathbf{e}\cdot\boldsymbol{\nabla}\right]f=\boldsymbol{\Omega} \qquad (1)$$

であり，格子上で離散化すると以下の離散式が得られる．

$$f_i(\mathbf{x}+\mathbf{e}_i\times dt,t+dt)=f_i(\mathbf{x},t)+dt\times\boldsymbol{\Omega}_i,\qquad i=1,\cdots,b \qquad (2)$$

ここでfは状態分布関数，$\mathbf{x}$は位置ベクトル，$\mathbf{e}$は粒子の速度，$\boldsymbol{\Omega}$は衝突項，bは粒子の自由度の数であり，$\boldsymbol{\Omega}$には以下に示す BGK モデルを用いた Single Relaxation Time (SRT)を用いるのが一般的である．

$$\boldsymbol{\Omega}_i^{BGK}=-\frac{1}{\tau}\left(f_i(\mathbf{x},t)-f_i^{eq}(\mathbf{x},t)\right) \qquad (3)$$

ここでf^{eq}は平衡状態分布関数，τは緩和係数である．時間積分は衝突（Collision）と並進（Stream）の 2 ステップからなり，衝突は(4)，並進は(5)のようにそれぞれ記述できる．

$$f_i^*(\mathbf{x},t)=f_i(\mathbf{x},t)+\boldsymbol{\Omega}_i,\qquad i=1,\cdots,b \qquad (4)$$

$$f_i(\mathbf{x}+\mathbf{e}_i\times dt,t+dt)=f_i^*(\mathbf{x},t),\qquad i=1,\cdots,b \qquad (5)$$

SRT の特徴は分散誤差が少ない点であるが，航空機の空力性能予測で対象とするような高 Re 数（低粘性）の流れでは計算が非常に不安定ですぐに破綻してしまう．SRT では全ての状態分布関数を同一の緩和係数で緩和するが，この過程に計算を不安定化させる高次のモードをダンプする仕組みが無いためである．この問題を解決するために様々な衝突項モデルが世界的に研究されており，JAXA LBM コードでは Dr. Mertin Geier によって提案された Cascaded LBM を衝突項に実装している．Cascaded LBM は状態分布関数を(6)に示す central moment に変換し，central moment space で異なる緩和係数で緩和（特に計算を不安定化させる高次のモーメントを dissipative な緩和係数で緩和）させることにより高 Re 数流れでも安定に計算することが出来る．

$$\rho\tilde{\mathbf{M}}_{p,q,r} = \sum_i (e_{ix} - u_x)^p \cdot \left(e_{iy} - u_y\right)^q \cdot (e_{iz} - u_z)^r \cdot f_i \\ = \mathbf{C} \cdot \mathbf{f} \qquad (6)$$

状態分布関数から central moment への変換行列$\mathbf{C}$は，$\mathbf{C} = \mathbf{C}(\mathbf{u})$とローカルな速度の関数になっているため，格子点毎に計算しなくてはならない．また，central moment space で緩和した後には状態分布関数に逆変換しなければならず，これらの行列演算にかかる計算負荷は大きい．参考に 2 次元 9 速度（D2Q9）モデルにおける変換行列$\mathbf{C}$およびその逆行列である$\mathbf{C}^{-1}$を(7)および(8)にそれぞれ示す．

3.1.2.2 Building-Cube Method (BCM)

Building-Cube Method では，計算領域を立方体領域（Cube）に分割し，各 Cube 内に等間隔直交格子（Cell）を生成して解析するフレームワークである．流体解析のプログラム部分は single block 用に作成し，Cube ループで全体の計算を回す間に適宜 Cube 間の情報交換を実施する．

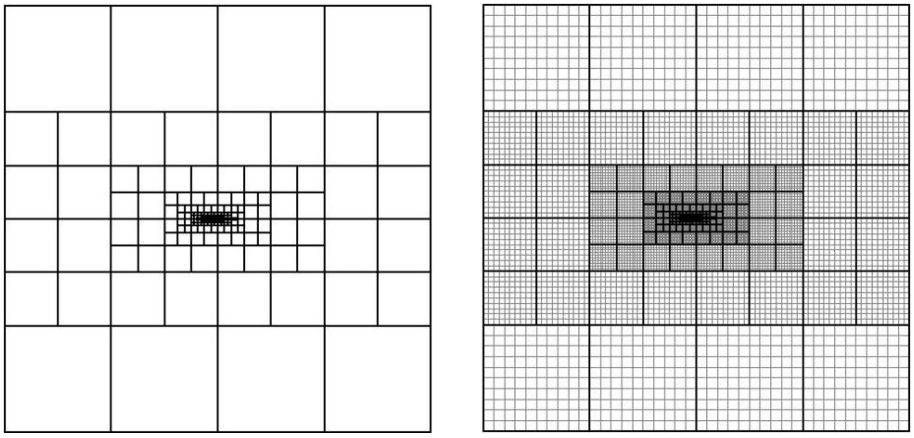


図 6　Cube（左図）と Cell（右図）

$$\begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
-u & 1-u & -u & -1-u & -u & 1-u & -1-u & -1-u & 1-u \\
-v & -v & 1-v & -v & -1-v & 1-v & 1-v & -1-v & -1-v \\
u^2-v^2 & (1-u)^2-v^2 & u^2-(1-v)^2 & (-1-u)^2-v^2 & u^2-(-1-v)^2 & (1-u)^2-(1-v)^2 & (-1-u)^2-(1-v)^2 & (-1-u)^2-(-1-v)^2 & (1-u)^2-(-1-v)^2 \\
u^2+v^2 & (1-u)^2+v^2 & u^2+(1-v)^2 & (-1-u)^2+v^2 & u^2+(-1-v)^2 & (1-u)^2+(1-v)^2 & (-1-u)^2+(1-v)^2 & (-1-u)^2+(-1-v)^2 & (1-u)^2+(-1-v)^2 \\
u\,v & (-1+u)\,v & -u\,(1-v) & (1+u)\,v & -u\,(-1-v) & (1-u)\,(1-v) & (-1-u)\,(1-v) & (-1-u)\,(-1-v) & (1-u)\,(-1-v) \\
-u\,v^2 & (1-u)\,v^2 & -u\,(1-v)^2 & (-1-u)\,v^2 & -u\,(-1-v)^2 & (1-u)\,(1-v)^2 & (-1-u)\,(1-v)^2 & (-1-u)\,(-1-v)^2 & (1-u)\,(-1-v)^2 \\
-u^2\,v & -(1-u)^2\,v & u^2\,(1-v) & -(-1-u)^2\,v & u^2\,(-1-v) & (1-u)^2\,(1-v) & (-1-u)^2\,(1-v) & (-1-u)^2\,(-1-v) & (1-u)^2\,(-1-v) \\
u^2\,v^2 & (1-u)^2\,v^2 & u^2\,(1-v)^2 & (-1-u)^2\,v^2 & u^2\,(-1-v)^2 & (1-u)^2\,(1-v)^2 & (-1-u)^2\,(1-v)^2 & (-1-u)^2\,(-1-v)^2 & (1-u)^2\,(-1-v)^2
\end{pmatrix} \qquad (7)$$

$$\begin{pmatrix}
\left(-1+u^2\right)\left(-1+v^2\right) & 2\,u\left(-1+v^2\right) & 2\left(-1+u^2\right)v & \frac{1}{2}\left(-u^2+v^2\right) & \frac{1}{2}\left(-2+u^2+v^2\right) & 4\,u\,v & 2\,u & 2\,v & 1 \\
-\frac{1}{2}u\,(1+u)\left(-1+v^2\right) & -\frac{1}{2}(1+2\,u)\left(-1+v^2\right) & -u\,(1+u)\,v & \frac{1}{4}\left(1+u+u^2-v^2\right) & \frac{1}{4}\left(1-u\,(1+u)-v^2\right) & -(1+2\,u)\,v & -\frac{1}{2}-u & -v & -\frac{1}{2} \\
-\frac{1}{2}\left(-1+u^2\right)v\,(1+v) & -u\,v\,(1+v) & -\frac{1}{2}\left(-1+u^2\right)(1+2\,v) & \frac{1}{4}\left(-1+u^2-v\,(1+v)\right) & \frac{1}{4}\left(1-u^2-v\,(1+v)\right) & -u\,(1+2\,v) & -u & -\frac{1}{2}-v & -\frac{1}{2} \\
-\frac{1}{2}(-1+u)\,u\left(-1+v^2\right) & -\frac{1}{2}(-1+2\,u)\left(-1+v^2\right) & -(-1+u)\,u\,v & \frac{1}{4}\left(1+(-1+u)\,u-v^2\right) & \frac{1}{4}\left(1+u-u^2-v^2\right) & v-2\,u\,v & \frac{1}{2}-u & -v & -\frac{1}{2} \\
-\frac{1}{2}\left(-1+u^2\right)(-1+v)\,v & -u\,(-1+v)\,v & -\frac{1}{2}\left(-1+u^2\right)(-1+2\,v) & \frac{1}{4}\left(-1+u^2+v-v^2\right) & \frac{1}{4}\left(1-u^2+v-v^2\right) & u-2\,u\,v & -u & \frac{1}{2}-v & -\frac{1}{2} \\
\frac{1}{4}u\,(1+u)\,v\,(1+v) & \frac{1}{4}(1+2\,u)\,v\,(1+v) & \frac{1}{4}u\,(1+u)\,(1+2\,v) & \frac{1}{8}\left(-u\,(1+u)+v+v^2\right) & \frac{1}{8}\left(u+u^2+v+v^2\right) & \frac{1}{4}(1+2\,u)\,(1+2\,v) & \frac{1}{4}(1+2\,u) & \frac{1}{4}(1+2\,v) & \frac{1}{4} \\
\frac{1}{4}(-1+u)\,u\,v\,(1+v) & \frac{1}{4}(-1+2\,u)\,v\,(1+v) & \frac{1}{4}(-1+u)\,u\,(1+2\,v) & \frac{1}{8}\left(u-u^2+v+v^2\right) & \frac{1}{8}\left((-1+u)\,u+v+v^2\right) & \frac{1}{4}(-1+2\,u)\,(1+2\,v) & \frac{1}{4}(-1+2\,u) & \frac{1}{4}(1+2\,v) & \frac{1}{4} \\
\frac{1}{4}(-1+u)\,u\,(-1+v)\,v & \frac{1}{4}(-1+2\,u)\,(-1+v)\,v & \frac{1}{4}(-1+u)\,u\,(-1+2\,v) & -\frac{1}{8}(u-v)\,(-1+u+v) & \frac{1}{8}\left((-1+u)\,u+(-1+v)\,v\right) & \frac{1}{4}(-1+2\,u)\,(-1+2\,v) & \frac{1}{4}(-1+2\,u) & \frac{1}{4}(-1+2\,v) & \frac{1}{4} \\
\frac{1}{4}u\,(1+u)\,(-1+v)\,v & \frac{1}{4}(1+2\,u)\,(-1+v)\,v & \frac{1}{4}u\,(1+u)\,(-1+2\,v) & -\frac{1}{8}(1+u-v)\,(u+v) & \frac{1}{8}\left(u+u^2+(-1+v)\,v\right) & \frac{1}{4}(1+2\,u)\,(-1+2\,v) & \frac{1}{4}(1+2\,u) & \frac{1}{4}(-1+2\,v) & \frac{1}{4}
\end{pmatrix} \qquad (8)$$

3.1.3 コード改善

本章では，FX100（JSS2 MA システム）を用いたコード改善による性能評価について報告する．性能評価にはC++で記述された2次元9速度（D2Q9）モデルのプログラムを用い，Lid Driven Cavity を計算対象として格子点数128×128の single block の直交格子を用いた．時間積分は1000ステップ分実施して性能を評価した．なお，計算は1core で実施し，コンパイルには以下オプションを使用した．

コンパイルオプション：-Kfast -Nlst=t -std=c++11

以降で出てくるサンプルコード内の変数を以下に定義する．

Q　　：自由度の数（2次元：Q=9，3次元：Q=27）
ftmp　：衝突後の状態分布関数
rho　　：密度
CM　　：central moment $\tilde{\mathbf{M}}$
omg　　：モード毎の緩和係数
M　　：変換行列$\mathbf{C}$
Minv　：逆変換行列$\mathbf{C}^{-1}$

3.1.3.1　Collision の SIMD 化

central moment を central moment space でモード毎に緩和させた後に，元の状態分布関数に逆変換する操作において，衝突後の分布関数を格納する配列：ftmp の定義にダブルポインタを用いていたため，コンパイル最適化が十分に効かなかった．そこで SIMD 化を促進するため，以下の変更を実施した．その結果，表 に示す通り 1.74 倍程度高速化することが出来た．

変更前

```
for (int n = 0; n < Q; ++n) {
  ftmp[I][n] = 0;
  for (int m = 0; m < Q; ++m) {
    ftmp[I][n] += rho[I] * (Minv[m][n] * (CM[m] * (1 - omg[m]) + CMeq[m] * omg[m]));
 }
}
```

変更後

```
T ftmp_[Q];
for (int n = 0; n < Q; ++n) {
  ftmp_[n] = 0;
}
for (int m = 0; m < Q; ++m) {
  for (int n = 0; n < Q; ++n) {
    ftmp_[n] += rho[I] * (Minv[m][n] * (CM[m] * (1 - omg[m]) + CMeq[m] * omg[m]));
  }
}
for (int n = 0; n < Q; ++n) {
  ftmp[I][n] = ftmp_[n];
}
```

表 1

	変更前	変更後
Elapsed(s)	29.0696	16.7274
MFLOPS	588.2525	895.9001
MFLOPS/PEAK(%)	1.6655	2.5452

3.1.3.2　central moment $\tilde{M} = C \cdot f$を求める行列演算の書き下し

central moment を求める行列は速度の関数になっており，格子点毎に実施する変換行列の定義およびそれらの演算はコストが大きい．そこで行列演算を書き下して性能が変化するか調査した．その結果，表 に示す通り更に 1.13 倍程度高速化することが出来，2 次元 9 速度モデルでは行列演算の書き下しにある程度効果があることが分かった．

変更前

```
//! Transform matrix from f to Central Moment
T M[Q][Q] = {
      1, 1, 1, 1, 1, 1, 1, 1, 1,
      -u, 1 - u,-u,-1 - u, -u, 1 - u, -1 - u, -1 - u, 1 - u,
      -v, -v,1 - v, -v, -1 - v,1 - v, 1 - v, -1 - v, -1 - v,
      u*v, -v * (1 - u),-u * (1 - v), -v * (-1 - u),-u * (-1 - v),(1 - u)*(1 - v),(-1 - u)*(1 - v),(-1 - u)*(-1 - v),(1 - u)*(-1 - v),
      u2 - v2, ua - v2,u2 - va,ub - v2,u2 - vb,ua - va, ub - va,ub - vb, ua - vb,
      u2 + v2, ua + v2,u2 + va,ub + v2,u2 + vb,ua + va, ub + va,ub + vb, ua + vb,
      -u * v2, v2*(1 - u),-u * va,v2*(-1 - u),-u * vb,(1 - u)*va, (-1 - u)*va,(-1 - u)*vb, (1 - u)*vb,
      -u2 * v, -v * ua,u2*(1 - v),-v * ub,u2*(-1 - v),ua*(1 - v),   ub*(1 - v),ub*(-1 - v), ua*(-1 - v),
      u2*v2, v2*ua,u2*va,v2*ub,u2*vb,ua*va,   ub*va,ub*vb, ua*vb
};
//! Calculation of Central Moment
for (int l = 0; l < Q; ++l) {
      CM[l] = 0;
      for (int n = 0; n < Q; ++n) {
         CM[l] += M[l][n] * f[I][n];
      }
      CM[l] = CM[l] / ro;
}
```

変更後

```
CM[0] = f_[0] + f_[1] + f_[2] + f_[3] + f_[4] + f_[5] + f_[6] + f_[7] + f_[8];
CM[1] = (-u)*f_[0] + (1 - u)*f_[1] + (-u)*f_[2] + (-1 - u)*f_[3] + (-u) * f_[4] +
          (1 - u) * f_[5] + (-1 - u) * f_[6] + (-1 - u) * f_[7] + (1 - u) * f_[8];
CM[2] = (-v)*f_[0] + (-v)*f_[1] + (1 - v)*f_[2] + (-v)*f_[3] + (-1 - v) * f_[4] +
          (1 - v) * f_[5] + (1 - v) * f_[6] + (-1 - v) * f_[7] + (-1 - v) * f_[8];
CM[3] = (u * v)*f_[0] + (-v * (1 - u))*f_[1] + (-u * (1 - v))*f_[2] + (-v * (-1 - u))*f_[3] + (-u * (-1 - v))*f_[4] +
          ((1 - u)*(1 - v))*f_[5] + ((-1 - u)*(1 - v))*f_[6] + ((-1 - u)*(-1 - v))*f_[7] + ((1 - u)*(-1 - v))*f_[8];
CM[4] = (u2 - v2)*f_[0] + (ua - v2)*f_[1] + (u2 - va)*f_[2] + (ub - v2)*f_[3] + (u2 - vb)*f_[4] +
          (ua - va)*f_[5] + (ub - va)*f_[6] + (ub - vb)*f_[7] + (ua - vb)*f_[8];
CM[5] = (u2 + v2)*f_[0] + (ua + v2)*f_[1] + (u2 + va)*f_[2] + (ub + v2)*f_[3] + (u2 + vb)*f_[4] +
          (ua + va)*f_[5] + (ub + va)*f_[6] + (ub + vb)*f_[7] + (ua + vb)*f_[8];
CM[6] = (-u * v2)*f_[0] + (v2*(1 - u))*f_[1] + (-u * va)*f_[2] + (v2*(-1 - u))*f_[3] + (-u * vb)*f_[4] +
          ((1 - u)*va)*f_[5] + ((-1 - u)*va)*f_[6] + ((-1 - u)*vb)*f_[7] + ((1 - u)*vb)*f_[8];
CM[7] = (-u2 * v)*f_[0] + (-v * ua)*f_[1] + (u2*(1 - v))*f_[2] + (-v * ub)*f_[3] + (u2*(-1 - v))*f_[4] +
          (ua*(1 - v))*f_[5] + (ub*(1 - v))*f_[6] + (ub*(-1 - v))*f_[7] + (ua*(-1 - v))*f_[8];
CM[8] = (u2 * v2)*f_[0] + (v2*ua)*f_[1] + (u2*va)*f_[2] + (v2*ub)*f_[3] + (u2*vb)*f_[4] +
           (ua*va)*f_[5] + (ub*va)*f_[6] + (ub*vb)*f_[7] + (ua*vb)*f_[8];
```

表 2

	変更前	変更後
Elapsed(s)	16.7274	14.7741
MFLOPS	895.9001	849.2416
MFLOPS/PEAK(%)	2.5452	2.4126

3.1.3.3　状態分布関数 $\mathbf{f} = \mathbf{C}^{-1} \cdot \tilde{\mathbf{M}}$を求める行列演算の書き下し

2.6.3.2 で行列演算の書き下しによる効果が得られたため，central moment から状態分布関数を求める逆変換の行列演算も書き下して性能が変化するか調査した．その結果，表に示す通り 1.64 倍程度高速化することが出来，2 次元 9 速度モデルにおける逆行列演算に関する書き下しには大きな効果があることが分かった．

変更前

```
T ftmp_[Q];
for (int n = 0; n < Q; ++n) {
    ftmp_[n] = 0;
}
for (int m = 0; m < Q; ++m) {
    for (int n = 0; n < Q; ++n) {
        ftmp_[n] += rho[I] * (MiC2[m][n] * (CM[m] * (1 - omg[m]) + CMeq[m] * omg[m]));
    }
}
for (int n = 0; n < Q; ++n) {
    ftmp[I][n] = ftmp_[n];
}
```

変更後

```
T CMpost[Q];
for (int n = 0; n < Q; ++n) {
    CMpost[n] = ro * (CM[n] * (1 - omg[n]) + CMeq[n] * omg[n]);
}
ftmp[I][0] = (u2 - 1) * (v2 - 1) * CMpost[0] + 0 * CMpost[1] + 0 * CMpost[2] + 4 * u * v * CMpost[3] + 0.5 * (v2 - u2) * CMpost[4] + 0.5 * (u2 + v2 - 2) * CMpost[5] + 2.*u * CMpost[6] + 2.*v * CMpost[7] + 1 * CMpost[8];
ftmp[I][1] = -0.5 * u * (u + 1) * (v2 - 1) * CMpost[0] + 0.5 * CMpost[1] + 0 * CMpost[2] + -(2 * u + 1) * v * CMpost[3] + 0.25 * (u2 + u - v2 + 1) * CMpost[4] + 0.25 * (-u2 - u - v2 + 1) * CMpost[5] + -(u + 0.5) * CMpost[6] + -v * CMpost[7] + -0.5 * CMpost[8];
ftmp[I][2] = -0.5 * v * (v + 1) * (u2 - 1) * CMpost[0] + 0 * CMpost[1] + 0.5 * CMpost[2] + -u * (2 * v + 1) * CMpost[3] + 0.25 * (u2 - v2 - v - 1) * CMpost[4] + 0.25 * (-u2 - v2 - v + 1) * CMpost[5] + -u * CMpost[6] + -(v + 0.5) * CMpost[7] + -0.5 * CMpost[8];
ftmp[I][3] = -0.5 * u * (u - 1) * (v2 - 1) * CMpost[0] + -0.5 * CMpost[1] + 0 * CMpost[2] + v * (1 - 2 * u) * CMpost[3] + 0.25 * (u2 - u - v2 + 1) * CMpost[4] + 0.25 * (-u2 + u - v2 + 1) * CMpost[5] + (0.5 - u) * CMpost[6] + -v * CMpost[7] + -0.5 * CMpost[8];
ftmp[I][4] = -0.5 * v * (v - 1) * (u2 - 1) * CMpost[0] + 0 * CMpost[1] + -0.5 * CMpost[2] + u * (1 - 2 * v) * CMpost[3] + 0.25 * (u2 - v2 + v - 1) * CMpost[4] + 0.25 * (-u2 - v2 + v + 1) * CMpost[5] + -u * CMpost[6] + (0.5 - v) * CMpost[7] + -0.5 * CMpost[8];
ftmp[I][5] = 0.25 * u * v * (u + 1) * (v + 1) * CMpost[0] + 0 * CMpost[1] + 0 * CMpost[2] + 0.25 * (2 * u + 1) * (2 * v + 1) * CMpost[3] + 0.125 * (-u2 - u + v2 + v) * CMpost[4] + 0.125 * (u2 + u + v2 + v) * CMpost[5] + 0.25 * (2 * u + 1)* CMpost[6] + 0.25 * (2 * v + 1) * CMpost[7] + 0.25 * CMpost[8];
ftmp[I][6] = 0.25 * u * v * (u - 1) * (v + 1) * CMpost[0] + 0 * CMpost[1] + 0 * CMpost[2] + 0.25 * (2 * u - 1) * (2 * v + 1) * CMpost[3] + 0.125 * (-u2 + u + v2 + v) * CMpost[4] + 0.125 * (u2 - u + v2 + v) * CMpost[5] + 0.25 * (2 * u - 1) * CMpost[6] + 0.25 * (2 * v + 1) * CMpost[7] + 0.25 * CMpost[8];
ftmp[I][7] = 0.25 * u * v * (u - 1) * (v - 1) * CMpost[0] + 0 * CMpost[1] + 0 * CMpost[2] + 0.25 * (2 * u - 1) * (2 * v - 1) * CMpost[3] + 0.125 * (-u2 + u + (v - 1) * v) * CMpost[4] + 0.125 * (u2 - u + (v - 1) * v) * CMpost[5] + 0.25 * (2 * u - 1) * CMpost[6] + 0.25 * (2 * v - 1) * CMpost[7] + 0.25 * CMpost[8];
ftmp[I][8] = 0.25 * u * v * (u + 1) * (v - 1) * CMpost[0] + 0 * CMpost[1] + 0 * CMpost[2] + 0.25 * (2 * u + 1) * (2 * v - 1) * CMpost[3] + 0.125 * (-u2 - u + (v - 1) * v) * CMpost[4] + 0.125 * (u2 + u + (v - 1) * v) * CMpost[5] + 0.25 * (2 * u + 1) * CMpost[6] + 0.25 * (2 * v - 1) * CMpost[7] + 0.25 * CMpost[8];
```

表 3

	変更前	変更後
Elapsed(s)	14.7741	9.0030
MFLOPS	849.2416	1457.6530
MFLOPS/PEAK(%)	2.4126	4.1411

3.1.3.4　Stream の方針変更

Stream では【自分⇒周囲】の順番で状態分布関数をコピーする操作をしていたが，キャッシュの効率を考慮して【周囲⇒自分】の順番に修正して性能が変化するか調査した．その結果，表 に示す通り更に 1.3 倍程度高速化することが出来た．

変更前

```
for (int I = 0; I < size; ++I) {
      ijk[0] = I % NDIM[0];
      ijk[1] = I / NDIM[0];

      for (int l = 0; l < Q; ++l) {
         indx[0] = ijk[0] - c[l][0];
         indx[1] = ijk[1] - c[l][1];
         indx[0] = (indx[0] >= 0) ? (indx[0] = indx[0]) : (indx[0] = 0);
         indx[1] = (indx[1] >= 0) ? (indx[1] = indx[1]) : (indx[1] = 0);
         indx[0] = (indx[0] < NDIM[0]) ? (indx[0] = indx[0]) : (indx[0] = NDIM[0] - 1);
         indx[1] = (indx[1] < NDIM[1]) ? (indx[1] = indx[1]) : (indx[1] = NDIM[1] - 1);
         J = indx[0] + NDIM[0] * indx[1];
         flag = !mask[I].isInner() * !mask[J].isInner();
         f[J][l] = ftmp[I][l] * (flag) + f[J][l] * (1 - flag);
}
```

変更後

```
for (int I = 0; I < size; ++I) {
      ijk[0] = I % NDIM[0];
      ijk[1] = I / NDIM[0];
      for (int l = 0; l < Q; ++l) {
         indx[0] = ijk[0] - c[l][0];
         indx[1] = ijk[1] - c[l][1];
         indx[0] = (indx[0] >= 0) ? (indx[0] = indx[0]) : (indx[0] = 0);
         indx[1] = (indx[1] >= 0) ? (indx[1] = indx[1]) : (indx[1] = 0);
         indx[0] = (indx[0] < NDIM[0]) ? (indx[0] = indx[0]) : (indx[0] = NDIM[0] - 1);
         indx[1] = (indx[1] < NDIM[1]) ? (indx[1] = indx[1]) : (indx[1] = NDIM[1] - 1);
         J = indx[0] + NDIM[0] * indx[1];
         f_[l] = ftmp[J][l];
      }
      f[I][0] = f_[0];
      f[I][1] = f_[1];
      f[I][2] = f_[2];
      f[I][3] = f_[3];
      f[I][4] = f_[4];
      f[I][5] = f_[5];
      f[I][6] = f_[6];
      f[I][7] = f_[7];
      f[I][8] = f_[8];
}
```

表 4

	変更前	変更後
Elapsed(s)	9.0030	6.9164
MFLOPS	1457.6530	1897.4156
MFLOPS/PEAK(%)	4.1411	5.3904

3.1.3.5　コード改善後の性能評価

3.1.3.2~3.1.3.4 の改善によって，検証ケース（格子点数128×128）では最終的におよそ 4.2 倍高速化することが出来た．そこで格子点数を変化させて SRT とコード改善した Cascaded LBM の解析時間の比較を行い，その結果を図 7 に示す．コード改善によりコストの高い Cascaded LBM が SRT と比較して，元々5.5 倍であった解析コストが高々1.25 倍程度にまで削減出来ていることが分かった．また，解析コスト増分は格子点数によらずほぼ一定であった．コード改善無しの 3 次元コードの場合，Cascaded LBM の解析コストは SRT の約 13 倍程度となっており（図 8），今後は 2 次元におけるコード改善の結果を反映して 3 次元における性能調査を実施する予定である．

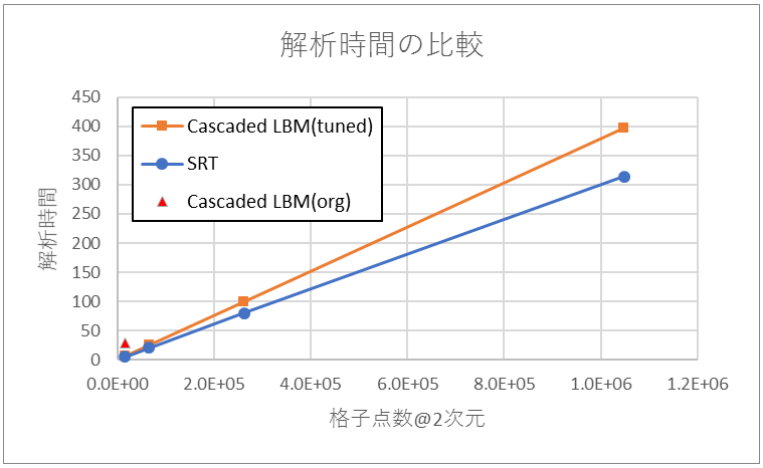


図 7　SRT と Cascaded LBM の解析時間の比較（2 次元）

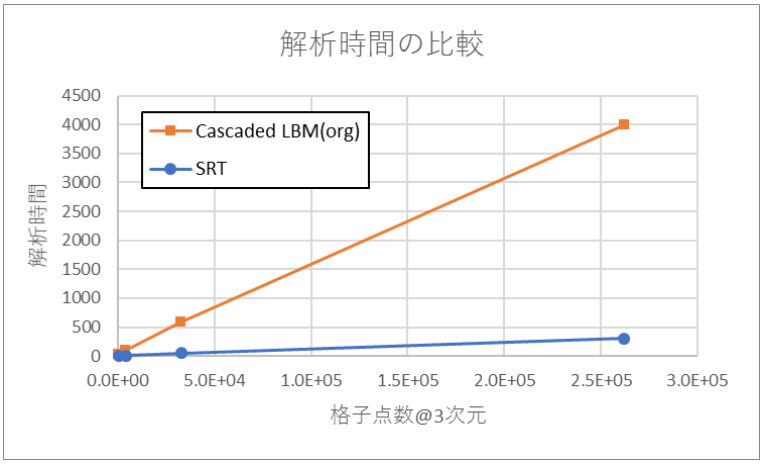


図 8　SRT と Cascaded LBM の解析時間の比較（3 次元）

3.1.4　スレッド並列性能

FX100 1node を用いて行ったスレッド並列性能調査の結果について以下に報告する．性能評価には C++で記述された 3 次元 27 速度（D3Q27）モデルのプログラムを用い，衝突項には SRT を採用した．計算格子には$64block \times 32^3 cell$のマルチブロック直交格子を用いて，スレッド数を 1~32 に変化させて実行時間の変化を調べ，その結果を図 9 に示す．またその際のプロファイラの結果を図 10 に示すが，並列計算時のホットスポットはブロック間で情報通信を実施する interfaceQ という関数であった．LBM は Navier-Stokes 方程式に比べて計算コストが小さいが使用変数が多く，コード改善によって演算に比べて通信がより際立つ結果となっている．今後は通信に関係する時間積分の箇所を分離する等，通信コストをうまく隠蔽できる実装に変更して性能調査を実施する予定である．

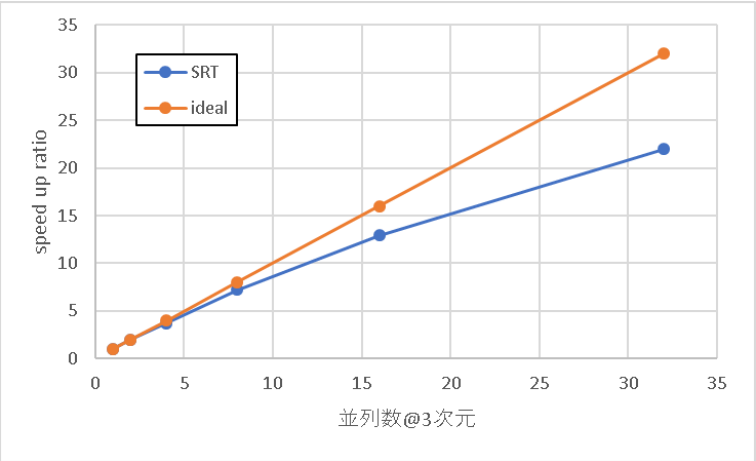


図 9　FX100 1node を用いたスレッド並列性能

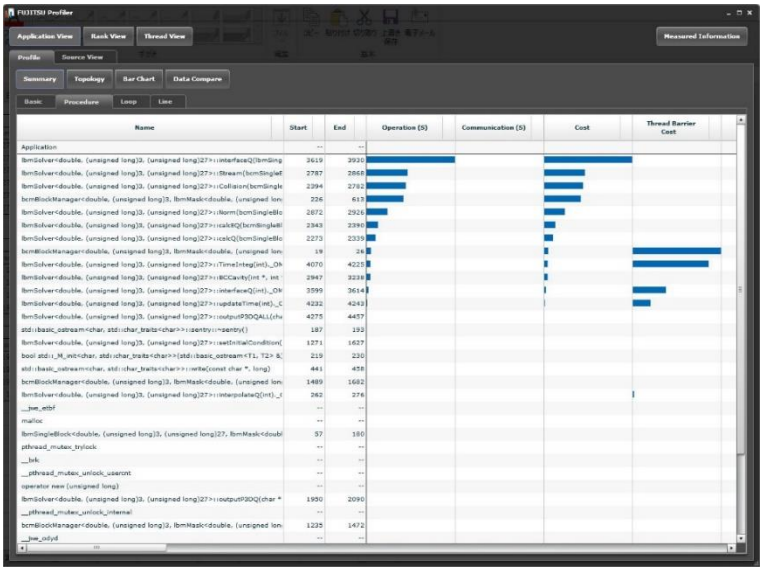


図 10　プロファイラによるホットスポットの抽出

3.1.5 まとめ

メニーコア WG での JAXA LBM コードの性能調査の結果，以下の知見が得られた．

- central moment に関わる行列演算は，2 次元の場合は SIMD 化を促進するより書き下した方が性能向上が期待できる．
- SRT と比較して，コード改善した Cascaded LBM の解析コストは 2 次元で約 1.25 倍程度にまで削減することが出来た．3 次元の場合は27×27の行列を書き下すことになり，演算量や演算の複雑さが増すため，調査する必要がある．
- LBM は Navier-Stokes 方程式ベースのアプローチより格子点当りの演算量は少ないが，格子点当りの変数が 2 次元では 9 変数（NS では 4 変数＋α），3 次元では 27 変数（NS では 5 変数＋α）であり，ノード内・ノード間の情報通信量が相対的に増えるため，ブロック間の情報通信を隠ぺいする等の処置が必要である．
- 現状では変数配列をダブルポインタで管理しているが，アーキテクチャやコンパイラの観点でどのようなデータの持たせ方が良いか（SoA or AoS），検討する必要がある．

3.2 MUTSU コード高速化の検討

核融合科学研究所　三浦英昭

3.2.1 はじめに

MUTSU コードは、核融合・トーラスプラズマの不安定性研究や、これに関わる基礎研究のために使用する汎用流体コードである。MUTSU コードでは、基礎研究用の矩形形状（周期境界、非周期境界条件を含む；直角座標系を使用）や、核融合研究向けのトーラス構造を近似的に表現するための歪んだ矩形形状(この場合には非直交曲線座標系で数式を表現する)、さらには直角座標や非直交曲線座標領域を組み合わせた座標において、拡張 MHD 方程式や Navier-Stokes 方程式を解くことを想定している。対象となる方程式によって使用するモジュールを変える（コード名称も、方程式に応じて MUTSU/cXMHD3D, MUTSH/cNS3D などと名前を変える）が、空間微分や時間発展パートは共通となっている。

MUTSU コードは、入力パラメータで二次精度中心差分、三次精度風上差分(KK スキーム)、四次精度中心差分、六次精度コンパクト差分、八次精度コンパクト、十次精度コンパクト差分を切り替えるように設計されている。数値フィルターも、陽的なフィルターから十次精度コンパクトフィルターまでをサポートしている。また、ほぼ同じ構造のモジュールを使って、3 方向に周期的な場合のフーリエ・擬スペクトル法によるシミュレーションが可能になっている。（こちらは、コード名末尾に”-T3”をつける。）また、矩形格子形状で複雑物体周りの流れを扱うために、volume penalization 用のモジュールも用意されている。

このコードの性能は、方程式の選択や微分・フィルターに関わるスイッチの選択で性能が大きく変わるため、ここではコンパクト差分法のパートと、MUTSU-T3 で使う 3 次元 FFT のパートについて報告する。

3.2.2 MUTSU, MUTSU-T3 コードの開発・コンパイル

このコードの開発は、主に核融合科学研究所の”プラズマシミュレータ”（富士通株式会社製 FX100)で行っている。また、一部の開発は、東京大学の Oakforest-PACS(富士通株式会社製)で行っている。FX100 における主要なコンパイルオプションは、以下の通りである。

```
FC      = mpifrtpx
FFLAGS = -Cpp -Cfpp -Kfast -Kreduction -Kopenmp -Kocl -Kparallel -Koptmsg=2 -Qt
```

また、Oakforest-PACS での主要なコンパイルオプションは以下のとおりである。

```
FC      = mpiifort
FFLAGS = -fpp -O2 -axMIC-AVX512 -qopt-report -qopenmp
           -mcmodel=medium -convert big_endian
```

3.2.3 コンパクト差分パートの高速化

コンパクト差分では､以下のような 3 重もしくは 5 重対角連立方程式を解く必要がある。

$$
\begin{aligned}
&\beta f'_{i+2,j,k} + \alpha f'_{i+1,j,k} + f'_{i,j,k} + \alpha f'_{i-1,j,k} + \beta f'_{i-2,j,k} \\
&\qquad = c\left(f_{i+3,j,k} - f_{i-3,j,k}\right) + b\left(f_{i+2,j,k} - f_{i-2,j,k}\right) + a\left(f_{i+1,j,k} - f_{i-1,j,k}\right) \\
&\beta f'_{i,j+2,k} + \alpha f'_{i,j+1,k} + f'_{i,j,k} + \alpha f'_{i,j-1,k} + \beta f'_{i,j-2,k} \\
&\qquad = c\left(f_{i,j+3,k} - f_{i,j-3,k}\right) + b\left(f_{i,j+2,k} - f_{i,j-2,k}\right) + a\left(f_{i,j+1,k} - f_{i,j-1,k}\right) \\
&\beta f'_{i,j,k+2} + \alpha f'_{i,j,k+1} + f'_{i,j,k} + \alpha f'_{i,j,k-1} + \beta f'_{i,j,k-2} \\
&\qquad = c\left(f_{i,j,k+3} - f_{i,j,k-3}\right) + b\left(f_{i,j,k+2} - f_{i,j,k-2}\right) + a\left(f_{i,j,k+1} - f_{i,j,k-1}\right)
\end{aligned}
$$

MUTSU コードにおいてこれらの微分評価を行うサブルーチンは、上記の方程式 3 セットの右辺の計算と、連立方程式の求解パートで構成される。右辺項の計算は、中心差分法などと同じ構造なので、比較的容易に、そこそこ高速な性能（FX100 でピーク性能比 6-8%前後）を達成可能である。連立方程式の解法としては、ここでは LU 分解法を採用した。（メニーコア WG の会合では、問題規模から考えて、行列反転型の直接解法の方が速いとの指摘があった。）

八次精度のコンパクト差分法を使用した場合（上の式で係数 β=0）について、LU 分解パートのループ融合などの最適化を行った。具体的には、

- ベクトル３成分の勾配（速度勾配テンソル）など、複数のコンパクト差分を一括して計算することによって流量を増加
- LU 分解で使用する作業配列の次元を増やし、ループ融合

  たとえば、入力データ b1, b2, b3 について

```
do k = 1, n2
do I = 1, n1
   dm(1,k,i) = b1(i,k,ijk)
   dm(2,k,i) = b2(i,k,ijk)
   dm(3,k,i) = b3(i,k,ijk)
end do
end do
```

  などの処理について、事前に入力データ b1,b2,b3 を一つの配列 b にまとめ、

```
do k = 1, n2
do i = 1, n1
do j = 1, 3
   dm(j,k,i) = b(i,k,ijk,j)
end do
end do
end do
```

  とする
- 融合配列の次元の入れ替え（最内側⇔再外側）を行う

などの工夫を施した。この結果、ピーク性能比で 4-9%を実現した。なお、配列の第２成分(x, y, z の直角座標では y 成分)に関する微分操作については、ストライドアクセスが発生することによる速度の低下のため、３方向についての微分の中では最も低速な演算(ピーク性比 4%)となった。

3.2.4 3 次元 FFT の通信時間隠蔽による周期境界用コード MUTSU-T3 コードの高速化

3 方向に対して周期境界条件が課される流体シミュレーションでは、中心差分法やコンパクト差分法に代えて、擬スペクトル法を使うことが多い。以下は、非圧縮性 Hall MHD シミュレーション版(MUTSU /iHallMHD3D-T3)の場合についての報告である。

擬スペクトル計算の場合、3 次元 FFT がコードの性能を決定づける。3 次元 FFT では通信・データの転置が計算時間のかなりの部分を占めるため、スーパーコンピュータがその性能を十全に発揮するのは難しい。ここでは、3 次元高速フーリエ変換ライブラリ“FFTE”(http://www.ffte.jp)をベースに、そのパーツを分解し、複数のフーリエ変換を束ねることで計算による通信時間の隠ぺいを図った。なお、この隠蔽によるシミュレーションコードの高速化は JHPCN 平成 30 年度共同研究課題にもなっており、共同研究者・FFTE の開発者である高橋大介博士から FFTE Ver.6.0 および 6.2α の提供を受けて行った。

この研究における通信時間の隠ぺいは、流体コードが往々にして保存形式で記述されることを利用している。たとえば速度勾配テンソルの演算は速度場３成分それぞれの勾配を求めることになり、３つのフーリエ変換を３セット行うと考えることができる。１回のフーリエ変換は、一方向へのフーリエ変換、転置・通信を３回繰り返すことで構成されるため、一つの変数のフーリエ変換と他の変数の通信・転置を重ねることで、通信時間をある程度隠蔽できると考えられる。

3.2.5 3 次元 FFT における計算時間の隠ぺい結果

計算時間の隠ぺいを行う場合と行わない場合の MUTSU /iHallMHD-T3 における 1 ステップあたりの計算時間を、FX100 および Oakforest-PACS で測定した。この隠蔽の際の関数の呼び出しは、もう一つの著名な 3 次元 FFT パッケージである P3DFFT (https://www.p3dfft.net) において、p3dfft_ftran_r2c_many, p3dfft_ftran_c2r_many といった名前の末尾に”_MANY”がつく関数と対応がつくため、P3DFFT と FFTE の比較も行った。

この計測結果は下表 (表 1, 表 2) のとおりである。表 1 は FX100、表 2 は Oakforest-PACS で計測した結果である。表中で「(通常版)」は、3 次元 FFT を個別にコールする場合を示している。P3DFFT の項目で「(_MANY)」は、p3dfft_ftran_r2c_many, p3dfft_ftran_c2r_many を用いて複数の変数に対する一括フーリエ変換を行う場合を示す。FFTE の「(隠蔽版)」項は、FFTE のコードを使用して複数のフーリエ変換（ここでは３変数）を行いつつ、通信について隠蔽を行った場合を示している。FX100 上での計測においては、アシスタントコアを使用していない。この点において、FX100 に特化した最適化によってさらに改善をもたらす余地は残っている。

この表からわかる通り、多くの場合において P3DFFT よりも FFTE の通常版の方が短時間で演算を終えていることがわかる。特に表 2 では、P3DFFT の_MANY 版の方が通常版よりも

7-30%高速であること、FFTE では通常版よりも隠蔽版が 11%程度高速であるが、最大規模の計算($N^3=4096^3$)では通常版の方が速い。最後の点についてはさらに調査が必要であり、また、P3D は比較的スレッド並列化を苦手とする（フラット MPI に近いほど、性能が顕著に高い）ためにスレッド並列数についての精査が必要であるなど、いくつかの課題は残る。しかし、概ね所期の目標を達成することができたと考えられる。他方、FX100 に比べて、Oakforest-PACS での性能はやや低いものとなっている。この点から、Oakforest-PACS についての最適化をさらに進める必要がある。

表 1. 核融合科学研究所 F100"プラズマシミュレータ"における 3 次元 FFT の通信時間の隠ぺい(MUTSU / iHallMHD3D-T3 の 1 ステップに要する時間；単位[秒])

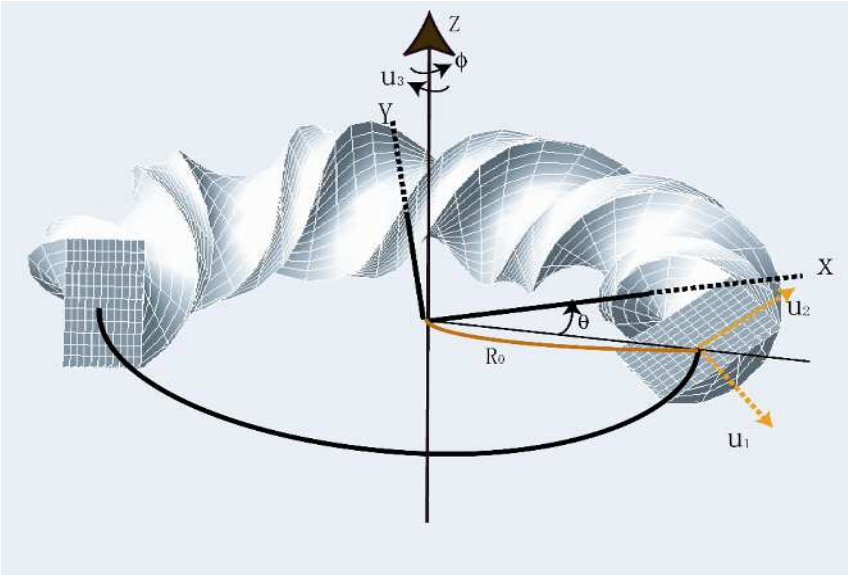
N^3	Nodes (Processes)	P3DFFT (_MANY)	FFTE(通常版)	FFTE(隠蔽あり)
2048^3	512 (1024)	37.939	—	24.990
1024^3	128 (256)	14.784	14.094	12.584
512^3	32 (64)	5.000	4.846	3.742
256^3	8 (16)	2.276	3.096	1.751

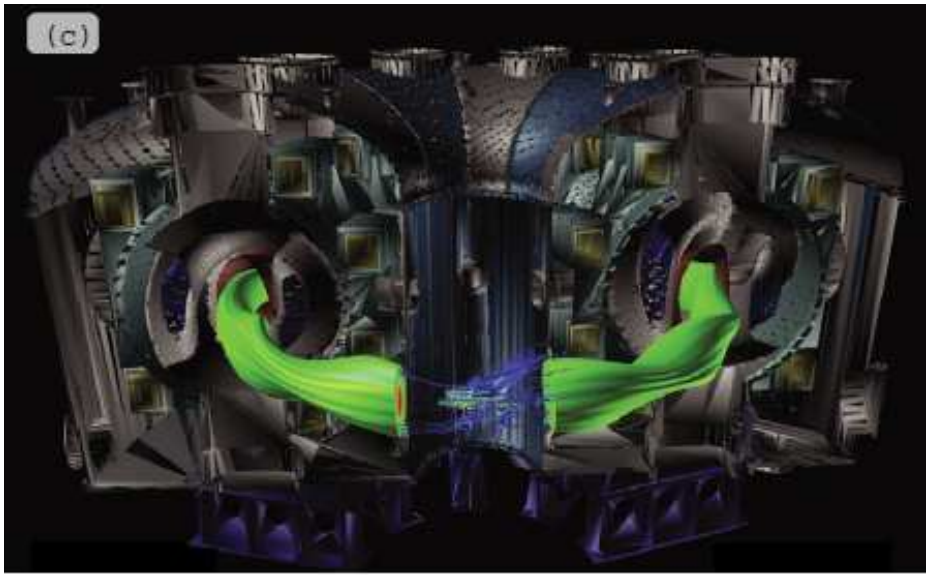
表 2. Oakforest-PACS における 3 次元 FFT の通信時間の隠ぺい(MUTSU / iHallMHD3D-T3 の 1 ステップに要する時間；単位[秒])

格子点数	Nodes (Processes)	P3DFFT (通常版)	P3DFFT (_MANY)	FFTE (通常版)	FFTE (隠蔽版)
$N^3=4096^3$	1024(16384)	327.221	306.214	231.546	247.666
$N^3=2048^3$	512(16384)	42.553	38.879	30.668	27.524
$N^3=1024^3$	128(4096)	18.573	17.844	13.408	12.867
$N^3=512^3$	32(1024)	8.662	8.228	5.398	4.906
$N^3=256^3$	8(256)	4.321	2.700	2.700	2.550

コード開発の目的

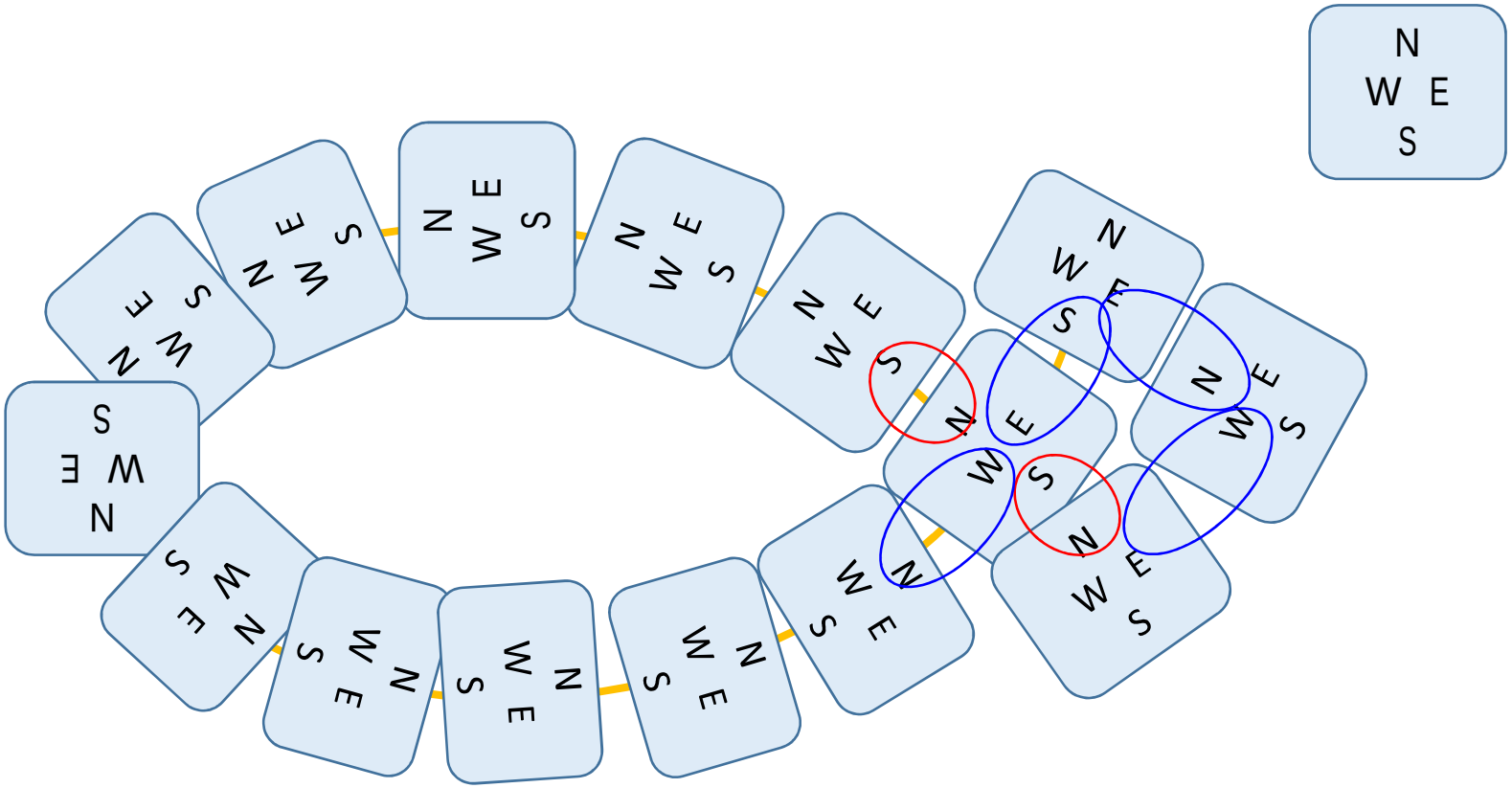
- 流体現象に関わる様々なシミュレーションを、矩形格子と高次差分で実行する
- 直角座標、
  極座標等直交座標、
  非直交曲線座標
- 周期境界・非周期境界





コード開発の経緯と現在の方針

- AMR コードとしてスタート
- AMR 部分はpending
- AMR構造を活かした柔軟な構造による多目的化 (たとえば C-grid, O-gridの変形など)を優先



コンパクト差分の実装

ベクトル３成分それぞれのgradientの計算
(多成分化による流量の増加)

ベクトル３成分それぞれのコンパクト差分右辺計算
(中心差分と似た構造なので、この部分の効率は低くない)

```
      subroutine compact_der3uniXYZ(a1, a2, a3, b1x, b1y, b1z,
     &     b3x, b3y, b3z, dx, dy, dz, iaflag1, iaflag2, iafla

       call compact_der_XYZ(a1, b1x, b1y, b1z, dx, dy, dz, iaflag1, iaflag2, iafla
g3)

       call compact_der_XYZ(a2, b2x, b2y, b2z, dx, dy, dz, iaflag1, iaflag2, iafla
g3)

       call compact_der_XYZ(a3, b3x, b3y, b3z, dx, dy, dz, iaf
g3)

       ifg = 1

       call Banbks3D_3vars(NWx,NWx,NWx,NWy,NWz,lu_m1,lu_m2,lu_dim,lu_dim, &
      &     work_Current%indx1,work_Current%ar1,work_Current%ar1L,     &
      &     b1x,b2x,b3x,ifg)

       ifg = 2

       call Banbks3D_3vars(NWy,NWy,NWx,NWy,NWz,lu_m1,lu_m2,lu_dim,lu_dim, &
      &     work_Current%indx2,work_Current%ar2,work_Current%ar2L,     &
      &     b1y,b2y,b3y,ifg)

       ifg = 3

       call Banbks3D_3vars(NWz,NWz,NWx,NWy,NWz,lu_m1,lu_m2,lu_dim,lu_dim, &
      &     work_Current%indx3,work_Current%ar3,work_Current%ar3L,     &
      &     b1z,b2z,b3z,ifg)

       return
      end subroutine compact_der3uniXYZ
```

3成分のコンパクト公式左辺のx,y,z一括LU分解

X-**方向**(Case1)

Y-**方向**(Case2)

Z-**方向**(Case3)

```
!option mp(p(0)), langlvl(save(0))
    subroutine Banbks_2D_3vars(n,np,n1,n2,n3,m1,m2,mp,al,mpl,indx,ifg,ijk,a,b1,b
2,b3)
       real(dd), dimension(n1,n2,n3)      :: b1, b2, b3

       select case(ifg)

       case (1)
!$omp parallel do default(none) &
!$omp&private(ijk,kk,ii,in2,i,k,l,dm1,dm1a_R,dm1b_R,dm1c_R,aval,ainv) &
!$omp&shared(n1,n2,n3,b1,b2,b3,a,al,indx,mm,m1)
          do ijk=1,n3

          do ii=1,n1,n1/NN
             do kk=1,n2,n2/NN
                do i=ii,min(ii+n1/NN-1,n1)
                   do k=kk,min(kk+n2/NN-1,n2)
                      dm1(1,k,i) = b1(i,k,ijk)
                      dm1(2,k,i) = b2(i,k,ijk)
                      dm1(3,k,i) = b3(i,k,ijk)
                   enddo
                enddo
             enddo
          enddo
```

Case1:
b(i ,j,k)
↑この成分への入出力
(外側 k で並列化可能)

```
      case (2)
!$omp parallel do default(none) &
!$omp& private(dm2a_R,dm2b_R,dm2c_R,l,k,ijk,i,aval,ainv,in1) &
!$omp& shared(n1,n2,n3,b1,b2,b3,m1,mm,a,al,indx)
         do ijk=1,n3

            l = m1
            do k = 1, n2
               i = indx(k)

               if(i.ne.k) then
                  do in1 = 1, n1
                     dm2a_R(in1) = b1(in1,k,ijk)
                     b1(in1,k,ijk) = b1(in1,i,ijk)
                     b1(in1,i,ijk) = dm2a_R(in1)

                     dm2b_R(in1) = b2(in1,k,ijk)
                     b2(in1,k,ijk) = b2(in1,i,ijk)
                     b2(in1,i,ijk) = dm2b_R(in1)

                     dm2c_R(in1) = b3(in1,k,ijk)
                     b3(in1,k,ijk) = b3(in1,i,ijk)
                     b3(in1,i,ijk) = dm2c_R(in1)
                  end do
```

Case2:
b(i, j, k)
↑この成分への入出力
(ストライドアクセスの発生)

```
      case (3)
!$omp parallel do default(none) &
!$omp& private(dm3a_R,dm3b_R,dm3c_R,l,k,i,ijk,aval,ainv,in1) &
!$omp& shared(n1,n2,n3,b1,b2,b3,m1,mm,a,al,indx)
         do ijk=1,n2

            l = m1
            do k = 1, n3
               i = indx(k)

               if(i.ne.k) then
                  do in1 = 1, n1
                     dm3a_R(in1) = b1(in1,ijk,k)
                     b1(in1,ijk,k) = b1(in1,ijk,i)
                     b1(in1,ijk,i) = dm3a_R(in1)

                     dm3b_R(in1) = b2(in1,ijk,k)
                     b2(in1,ijk,k) = b2(in1,ijk,i)
                     b2(in1,ijk,i) = dm3b_R(in1)

                     dm3c_R(in1) = b3(in1,ijk,k)
                     b3(in1,ijk,k) = b3(in1,ijk,i)
                     b3(in1,ijk,i) = dm3c_R(in1)
                  end do
               end if
```

Case3:
b(i, j, k)
↑この成分への入出力
(内側 j で並列化可能)

コンパクト差分パート最適化

- 最適化前の高コスト部の状態

高コストルーチン	SIMD化	SWP化	自動並列化	経過時間 (コード全体200ステップ 898秒)
Banbks_2d_3vars	○	△	△	74.91
Banbks_2d_u2	○	△	△	61.91
Compact_der_xyz	○	○	○	12.80
Compact_der_z1	○	○	△	3.54

- OpenMP化が速い場合とFX100自動並列化の方が速い場合が混在 (ループ長に関する分岐判断のため)
  全ループをOpenMP化すると低速化するので, 個別に切り分け
- キャッシュミス等の分析と配列融合で最適化

配列融合について（1-1)

・banbks_2d_3vars の case 1（修正前）

```
real(dd), dimension(n1,n2,n3)     :: b1, b2, b3
   :
      do k = 1, n2
        do i = 1, n1
          dm(1,k,i) = b1(i,k,ijk)
          dm(2,k,i) = b2(i,k,ijk)
          dm(3,k,i) = b3(i,k,ijk)
        enddo
      enddo
      l = m1
      do k = 1, n1
        i = indx(k)

        if(i.ne.k) then
           do in2 = 1, n2
              adm1 = dm(1,in2,k)
              dm(1,in2,k) = dm(1,in2,i)
              dm(1,in2,i) = adm1
              adm2 = dm(2,in2,k)
              dm(2,in2,k) = dm(2,in2,i)
              dm(2,in2,i) = adm2
              adm3 = dm(3,in2,k)
              dm(3,in2,k) = dm(3,in2,i)
              dm(3,in2,i) = adm3
           end do
        end if
```

(中略)

```
      do k = 1,n2
        do i = 1,n1
          b1(i,k,ijk)   = dm(1,k,i)
          b2(i,k,ijk)   = dm(2,k,i)
          b3(i,k,ijk)   = dm(3,k,i)
        enddo
      enddo
```

LU**分解** banbks_*_3vars ... 3**変数一括変換**

配列融合前
Real(dd),dimension(n1,n2,n3)::b1,b2,b3
配列融合後（内側）
Real(dd),dimension(3,n1,n2,n3)::b
配列融合後（外側）
Real(dd),dimension(n1,n2,n3,3)::b

配列融合について（1-2）
最内で融合

・banbks_2d_3vars の case 1（修正前）

```
real(dd), dimension(n1,n2,n3)      :: b1, b2, b3
   :
      do k = 1, n2
        do i = 1, n1
          dm(1,k,i) = b1(i,k,ijk)
          dm(2,k,i) = b2(i,k,ijk)
          dm(3,k,i) = b3(i,k,ijk)
        enddo
      enddo
      l = m1
      do k = 1, n1
        i = indx(k)

        if(i.ne.k) then
           do in2 = 1, n2
              adm1 = dm(1,in2,k)
              dm(1,in2,k) = dm(1,in2,i)
              dm(1,in2,i) = adm1
              adm2 = dm(2,in2,k)
              dm(2,in2,k) = dm(2,in2,i)
              dm(2,in2,i) = adm2
              adm3 = dm(3,in2,k)
              dm(3,in2,k) = dm(3,in2,i)
              dm(3,in2,i) = adm3
           end do
        end if
```

(中略)

```
      do k = 1,n2
        do i = 1,n1
          b1(i,k,ijk)   = dm(1,k,i)
          b2(i,k,ijk)   = dm(2,k,i)
          b3(i,k,ijk)   = dm(3,k,i)
        enddo
      enddo
```

・banbks_2d_3vars の case 1（修正後-最内で配列融合）

```
real(dd), dimension(3,n1,n2,n3)      :: b
   :
      do k = 1, n2
        do i = 1, n1
          do j=1,3
            dm(j,k,i) = b(j,i,k,ijk)
          enddo
        enddo
      enddo       l = m1
      do k = 1, n1
        i = indx(k)

        if(i.ne.k) then
           do in2 = 1, n2
              adm1 = dm(1,in2,k)
              dm(1,in2,k) = dm(1,in2,i)
              dm(1,in2,i) = adm1
              adm2 = dm(2,in2,k)
              dm(2,in2,k) = dm(2,in2,i)
              dm(2,in2,i) = adm2
              adm3 = dm(3,in2,k)
              dm(3,in2,k) = dm(3,in2,i)
              dm(3,in2,i) = adm3
           end do
        end if
        if(l.lt.n1) l = l + 1
        do i = k+1, l
           aval = al(k,i-k)
           do in2 = 1, n2
              dm(1,in2,i)=dm(1,in2,i)-aval*dm(1,in2,k)
              dm(2,in2,i)=dm(2,in2,i)-aval*dm(2,in2,k)
              dm(3,in2,i)=dm(3,in2,i)-aval*dm(3,in2,k)
           end do
        end do
      end do
```

(中略)

```
      do k = 1,n2
        do i = 1,n1
          do j=1,3
            b(j,i,k,ijk)   = dm(j,k,i)
          enddo
        enddo
      enddo
```

配列融合について（1-3）
最外で融合

・banbks_2d_3vars の case 1（修正前）

```
real(dd), dimension(n1,n2,n3)     :: b1, b2, b3
   :
      do k = 1, n2
        do i = 1, n1
          dm(1,k,i) = b1(i,k,ijk)
          dm(2,k,i) = b2(i,k,ijk)
          dm(3,k,i) = b3(i,k,ijk)
        enddo
      enddo
      l = m1
      do k = 1, n1
        i = indx(k)

        if(i.ne.k) then
           do in2 = 1, n2
              adm1 = dm(1,in2,k)
              dm(1,in2,k) = dm(1,in2,i)
              dm(1,in2,i) = adm1
              adm2 = dm(2,in2,k)
              dm(2,in2,k) = dm(2,in2,i)
              dm(2,in2,i) = adm2
              adm3 = dm(3,in2,k)
              dm(3,in2,k) = dm(3,in2,i)
              dm(3,in2,i) = adm3
           end do
        end if
```

(中略)

```
      do k = 1,n2
        do i = 1,n1
          b1(i,k,ijk)   = dm(1,k,i)
          b2(i,k,ijk)   = dm(2,k,i)
          b3(i,k,ijk)   = dm(3,k,i)
        enddo
      enddo
```

・banbks_2d_3vars の case 1（修正後-最外で配列融合

```
real(dd), dimension(n1,n2,n3,3)     :: b
   :
      do k = 1, n2
        do i = 1, n1
          do i = 1, n1
             dm(j,k,i) = b(i,k,ijk,j)
          enddo          enddo
      enddo        l = m1
      do k = 1, n1
        i = indx(k)

        if(i.ne.k) then
           do in2 = 1, n2
              adm1 = dm(1,in2,k)
              dm(1,in2,k) = dm(1,in2,i)
              dm(1,in2,i) = adm1
              adm2 = dm(2,in2,k)
              dm(2,in2,k) = dm(2,in2,i)
              dm(2,in2,i) = adm2
              adm3 = dm(3,in2,k)
              dm(3,in2,k) = dm(3,in2,i)
              dm(3,in2,i) = adm3
           end do
        end if
```

(中略)

```
      do j=1,3
        do k = 1,n2
          do i = 1,n1
            b(i,k,ijk,j)   = dm(j,k,i)
          enddo
        enddo
      enddo
```

配列融合について（1-4）
結果比較(1ノード4プロセス500回コール時間)

Banbks_2D_3varsの実行時間					
	Case 1	Case 2	Case 3	計(秒)	改善率
修正前	3.24	1.44	1.90	6.59	-
最内	3.17 (4.09%)	1.53 (9.09%)	1.74 (6.52%)	6.43	2.30%
最外	3.63	1.45	1.90	6.99	-6.12%

Banbks_2D_u2 (Banbks_2D_3vars類似;変数長偶数限定) 実行時間					
	Case 1	Case 2	Case 3	計(秒)	改善率
修正前	0.76	0.39	0.66	1.81	-
最内	0.94	0.72	0.97	2.63	-45.01%
最外	0.75	0.39	0.66	1.80	0.59%

Case 1-3の実行回数にも依存するが、最内がやや有利
非保存的な方程式などでは、高速化の保証がない
Case 1 がコンパクト差分最適化（ピーク性能比向上）のための問題点

3次元FFT “FFTE”を用いた計算時間の隠蔽

- 平成30年度JHPCN 共同研究課題
  「電磁流体力学乱流の高精度・高並列LESシミュレーションコード開発研究」
  共同研究者：高橋大介先生（筑波大）他

- 3次元FFT ... FFTE (http://www.ffte.jp) ver 6.0

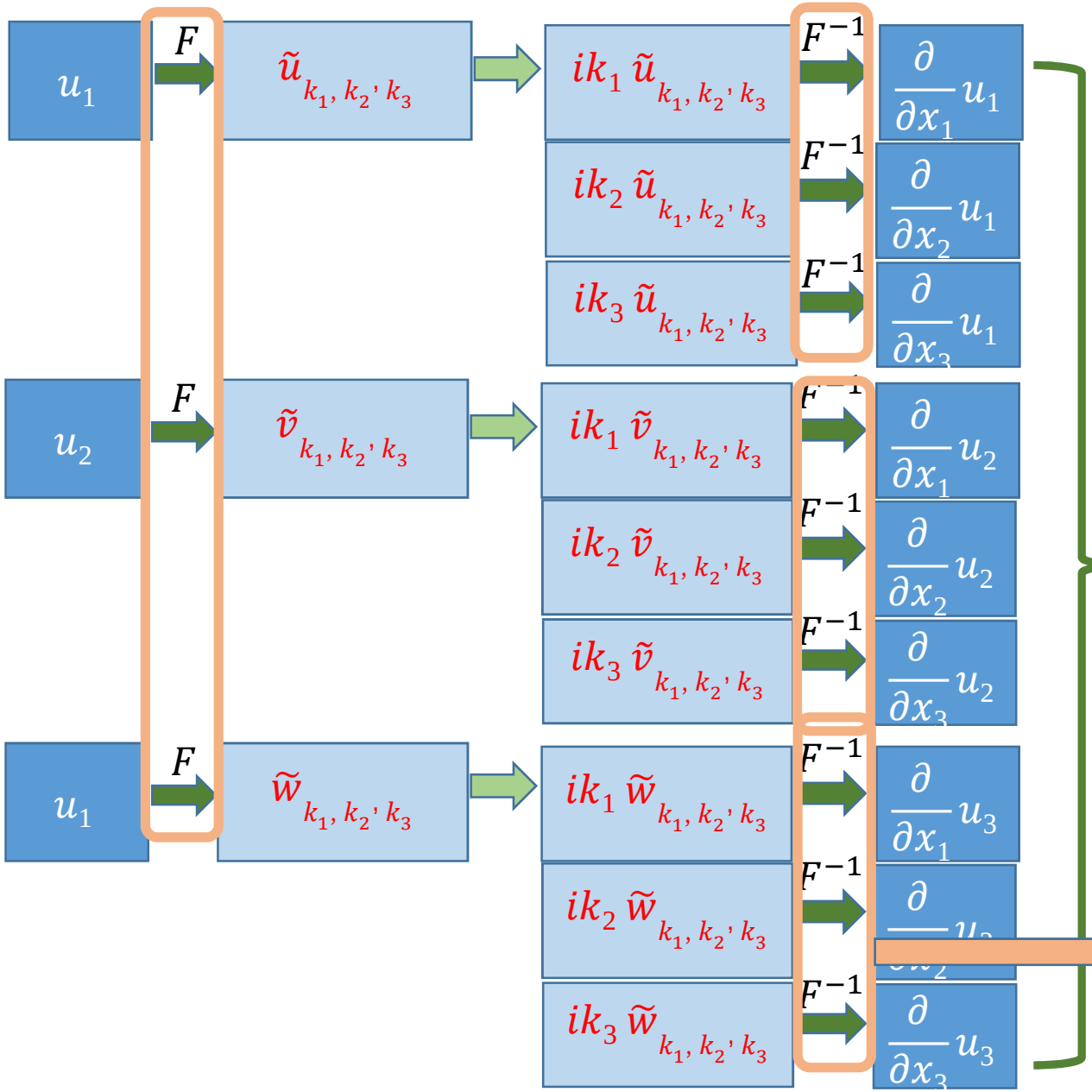
- 基本方針
  3次元FFT を分解
  多重FFT=>転置⇒多重FFT=>転置⇒多重FFT
  通信と計算・転置のオーバーラップ
  * p3dfft の _MANY 機能と同様

FX100性能測定詳細（ループ交換等最適化前後比較）

	Elapsed(secs)			１次キャッシュミス率(%)		
	変更前	変更後	改善率	変更前	変更後	改善率
FFTINVXL1	42.56	34.00	20.1%	29.97%	26.94%	10.1%
FFTINVYL1	31.14	26.74	14.1%	30.32%	26.93%	11.2%
FFTINVZL1	36.86	29.26	20.6%	30.20%	26.96%	10.7%

nproc	変更	timesteploop	正変換	逆変換	a)正変換 (1回)	b)逆変換 (1回)	比(b/a)
256	前	782.4	247.8	412.5	1.2388	1.4029	113.2%
	後	757.2	240.0	394.7	1.2002	1.3423	111.8%
512	前	491.1	161.8	289.8	0.8088	0.9859	121.9%
	後	480.8	158.3	281.9	0.7916	0.9589	121.1%
1024	前	249.6	88.2	145.8	0.4409	0.4958	112.5%
	後	244.7	86.5	142.6	0.4323	0.4851	112.2%
2048	前	105.9	35.3	62.7	0.1766	0.2132	120.7%
	後	105.4	34.6	62.9	0.1732	0.2138	123.5%
4096	前	76.5	26.4	46.4	0.1319	0.1579	119.8%
	後	76.3	26.2	46.3	0.1309	0.1576	120.4%

擬スペクトル計算における”_MANY”



$$u_1\frac{\partial u_1}{\partial x_1}+u_2\frac{\partial u_1}{\partial x_2}+u_3\frac{\partial u_1}{\partial x_3}$$

$$u_1\frac{\partial u_2}{\partial x_1}+u_2\frac{\partial u_2}{\partial x_2}+u_3\frac{\partial u_2}{\partial x_3}$$

$$u_1\frac{\partial u_3}{\partial x_1}+u_2\frac{\partial u_3}{\partial x_2}+u_3\frac{\partial u_3}{\partial x_3}$$

3次元FFT:
正変換*3
逆変換*9

- 一括処理（通信隠蔽のチャンス）
- P3dfft に同様の機能
  ⇒複雑で最適化に難
- FFTEを用いて実装

隠蔽効果の検証 (FX100)

測定データ 格子数：2048x2048x2048
ステップ数：10
スレッド数：16

Time step loop							
	P3dfft 2.7.5		ffte(X**ペンシル**)		ffte(Z**ペンシル**)		
nproc	**通常版**	many**版**	**通常版**	**隠蔽版**	**通常版**	**隠蔽版**	
256	2283.016	2138.858	1577.800	1139.746	1092.112	821.853	
512	1036.223	986.424	1160.650	973.610	762.255	549.588	
1024	482.646	428.720	511.279	349.656	342.041	246.963	
2048	198.744	199.216	201.124	135.184	132.365	93.802	
4096	88.054	89.913	140.027	106.548	86.406	61.901	

- **川島康弘氏（富士通(株)）による計測**
- **注：現在** FFTE 6.0⇒6.2α**移行、隠蔽版は若干速度低下(最適化中)**

隠蔽効果の検証(Oakforest-PACS)

- **ライブラリ**
  fftw … fftw 3.3.6-pl2
  p3dfft … p3dfft3.7.7
  ffte … ffte ver 6.0

- **コンパイル**
  mpiifort –fpp –O2
  -axMIC-AVX512 –qopt-report –qopenmp
  -mcmodel=medium –convert big_endian

隠蔽効果の検証(Oakforest-PACS)

	Time for 1-time-step (secs)				
		P3dfft		FFTE (z-pencil)	
	Node,procs	**通常版**	MANY**版**	**通常版**	MANY**版**
$N^3=4096^3$	1024, 16384	327.221	306.214	231.546	247.666
$N^3=2048^3$	512, 16384	42.553	38.879	30.668	27.524
$N^3=1024^3$	128, 4096	18.573	17.844	13.408	12.867
$N^3=512^3$	32,1024	8.662	8.228	5.398	4.906
$N^3=256^3$	8,256	4.321	2.700	2.700	2.550

FFTE $N^3=4096^3$ **で通常版とMANY版の計算時間の逆転については、要調査。**

3.3　生体分子粗視化シミュレータ CafeMol の FX100 での性能測定とチューニング

理化学研究所 情報システム部
検崎博生
富士通株式会社 TC ソリューション事業本部
渡邉健太

3.3.1　はじめに

CafeMol はタンパク質や核酸のような生体分子の粗視化シミュレータである。CafeMol で用いる粗視化モデルでは、全原子モデルに比べて粒子数が 100 分の 1 程度まで減り、計算量が大幅に減る。このことにより長時間の時間発展の計算が可能になり、生体分子の大規模な構造変化を取り扱うことができるのであるが、同時に並列性能を出しにくいことにもつながっている。具体的には、時間発展の 1 step が 1 ms 以下になることも多く、ノード間通信が問題になってくるので複数ノードでの並列計算を行うことは難しく、1 ノードでの性能向上を図ることが大事になってきている。

本 WG では、CafeMol の性能測定とチューニングについて検崎よる 2 回の発表と富士通渡邉による 1 回の発表があったので、ここではそれらをまとめることとする。最初に検崎による 1 回目の発表では、FX100 と skylake の比較と計算量が一番多い部分について配列の構造を Structure of Array(SoA)から Array of Structure(AoS)への変更を行うチューニングを行った。次に、渡邉の発表では FX100 が Skylake に比べて性能が出ていない部分について性能測定とチューニングが行われた。最後に、検崎による 2 回目の発表では、渡邊により示されたタイマーのオーバーヘッドの確認と追加のチューニングを行った。

3.3.2 プログラム概要

分子動力学シミュレーションでは、分子の動きを時間積分することにより系を時間発展させるわけであるが、分子間に働く力の計算が最も計算時間が掛かる部分となっている。CafeMol では力の計算にネイバリングリスト方式を用いて計算量を減らしており、ネイバリングリストを分割する形で MPI と OpenMP によるハイブリッド並列化を行っている。

粗視化された粒子間に働く相互作用はさまざまなものがあり、相互作用毎に異なるチューニングが必要になってくる。ここではタンパク質/DNA 複合体であるヌクレオソームの 10,000 step の時間発展の性能測定を行う。ヌクレオソームは粗視化粒子数は 2,000 程度の系で、計算量が一番多いのが静電相互作用の計算で粒子数の 2 乗程度の計算量であり、それ以外に粒子数の数倍程度の計算量であるさまざまな相互作用がある。

3.3.3　測定環境

検崎による性能測定には、理化学研究所の HOKUSAI システムを使用した。

- FX100：SPARC64TMXIfx(1.975H)、富士通コンパイラ、-Kfast
- Intel Xeon：Intel Xeon Gold 6148(3.1GHz)、インテルコンパイラ

富士通渡邉による性能測定には、以下のシステムを用いた。

- FX100：SPARC64TMXIfx(1.5H)、富士通コンパイラ
- Intel Xeon：Intel Xeon Gold 6148(3.1GHz)、インテルコンパイラ

3.3.4　検崎による1回目の発表のまとめ

以前の性能測定で、FX100 で 1 ノード 32 コアを用いた時は、MPI による 2 プロセス並列と OpenMP による 16 スレッド並列の組合せが一番速くなることを確認していた。Skylake の 1 ノード 40 コアでは、4 プロセスと 10 スレッド並列の組合せが一番速くなった。FX100 と Skylake を比較すると、FX100 は一番計算量の大きい静電相互作用は SIMD 化により Skylake よりも速くなった。一方、他の相互作用は SIMD 化が困難で Skylake に比べて数倍程度遅くなるものが多く、全体の計算時間も FX100 の方が遅くなった。

次に、skylake において静電相互作用のさらなるチューニングを行った。具体的には静電相互作用につかうネイバリングリストの配列の構造を Array of Structure(AoS)から Structure of Array(SoA)に変更した。結果として、静電相互作用の計算時間が 6.1 s から 4.6 s に高速化された。条件によってどちらの構造を使った方がいいかは変わるようだが、配列の構造を変えることがチューニングの 1 手法として有効であることを確認できた。

3.3.5　富士通渡邉による発表のまとめ

上記の検崎による測定結果で、多くの相互作用で FX100 の性能が Skylake に比べて性能が悪かったので、富士通渡邉により性能測定とチューニングが行われた。状況を簡単にするために、ボンド長、ボンド角、Go ポテンシャルの 3 つの相互作用について調べることとした。

まず、MPI_wtime により時間計測を行っていたが、オーバーヘッドが大きいことが示された。オーバーヘッドが少ないタイマーとして、FX100 では gettod を、Skylake では clockx を使うこととした。

チューニングとしてはボンド長について行われ、手動によるループアンスイッチングと”!ocl simd_redundant_vl()”を配列式の直前に挿入することが行われ、10%程度の性能向上が得られた。また、ループ最後のリダクション演算部にインダイレクトアクセスがあり、データの重なりがないようなレイアウトにすると大きな最適化の効果を得られることが推測された。

3.3.6　検崎による2回目の発表のまとめ

富士通渡邉による指摘を受け時間計測の方法についての確認とボンド長とボンド角の相互作用計算のチューニングを行った。

タイマーについては、MPI_wtime と gettod に加えて clock_time も試してみたところ、MPI_wtime のオーバーヘッドが大きいことが再確認され、gettod と clock_time の差は小さかったので、移植性を考え clock_time を採用することとした。

ボンド長のチューニングについては、OpenMP のスレッド並列のチャンクサイズを 1 にすることによって行った。これは、ボンド長のネイバリングリストでは、リダクション部分がスレッド並列毎に異なる領域を持ち、ネイバリングリストの作成方法から 4 つ以上離れていればリダクション部分が重ならいないようになっていることを利用している。このことから、チャンクサイズを 1 にしてスレッド並列数が 3 以上ならば、リダクション部分に

重なりがないことが保証されるのである。よって、”!ocl norecurrence()”を付けることにより、ボンド長の計算がSIMD化できた。ただし、ボンド長ではチャンクサイズを1にすることによるオーバーヘッドが大きく、SIMD 化による性能向上は小さかったので、結果として高速化にはならなかった。

また、ボンド角についても同様のチューニングが行った。こちらはSIMD化による性能向上の効果が比較的大きく、オーバーヘッドを込みで30%程度高速化した。ただし、Skylakeに比べると性能はまだまだ不十分ではあり、さらなる高速化が求められる。

3.3.7 まとめ

CafeMolをFX100とSkylake上で性能測定を行い、いくつかのチューニングを試みた。最も計算量の多い静電相互作用については、skylakeについてAoSからSoAの変更により性能が向上した。それ以外の計算量が少ない相互作用について FX100 についてチューニングを行ったところ若干の性能向上に成功したが、今回のチューニングにより差が縮まったとはいえ Skylake に比べて十分な性能を出すことが出来なかった。この理由は CPU やコンパイラによる out of order などの命令の充実に差があるためと考えられ、ポスト京では一部改善が見込めると考えらえる。

また、MPI_Wtime については無視できないオーバーヘッドがあり、時間計測を行う際はsystem_clock やシステム依存なルーチンの利用などを行う必要がある場合がある。この指摘に対し、FX100 では経過時間を測定する際、NTP による時間自動調整結果が補正されないように clock_gettime システムコールを用いているためであると富士通より報告があった。なお、ポスト京で MPI_Wtime のオーバーヘッドは FX100 で用意されている gettod 相当の処理に変更され、MPI_Wtime の性能は改善される見込みであることも併せて富士通から報告されている。

2017/10/20
SS研メニーコア時代のアプリ性能WG
第4回会合

生体分子粗視化シミュレータCafeMol Skylakeでのチューニング

検崎博生
理研 情報システム部

概要

- 粗視化モデルとCafeMolについて
  - 生体分子シミュレーションと粗視化
  - CGタンパク質/DNAモデル
  - 計算方法と並列化
- Skylakeでのベンチマーク結果
  - 1ノードでのMPI並列数とスレッド並列数
- Skylakeでのチューニング結果
  - 静電相互作用のSIMD化
  - データ構造変換

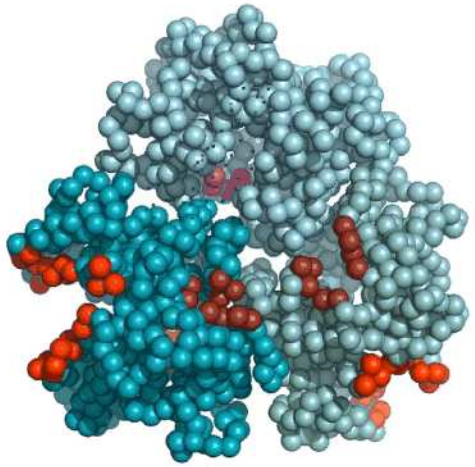
HOKUSAI System

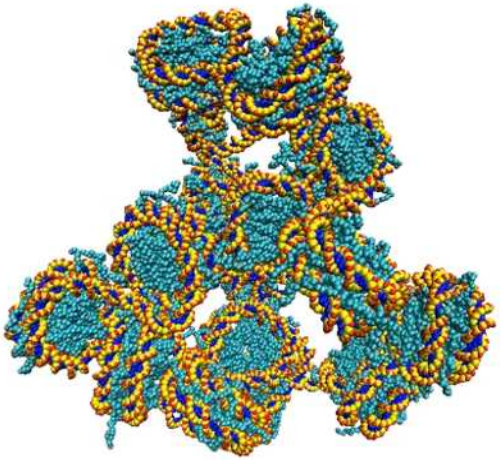
- GreatWave Massively Pallarell Computer (GWMPC)
  - 1,080 nodes, 32 cores/node, 34,560 cores
    - 1.092 PFLOPS (1.975 Hz x 16 FP x 32 cores x 1,080 CPUs)
  - CPU: SPARCTMXIfx (1.975GHz, 32 cores, 1 CPU/node)
  - Memory: 32GB/node
    - Memory BW: 480 GB/s/node
  - Interconnect: Tofu2 6D-Mesh/Torus (12.5GB/s, bidirectional)
- Application Computing Server (ACS)
  - Large memory server: 2 nodes, 1.5 TB/node
  - GPU server: 30 nodes, 4 GPU/node, Tesla K20X
- BigWaterfall Massively Parallel Computer (BWMPC)
  - 840 nodes, 40 cores/node, 33,600 cores
    - 2.58 PFLOPS (2.4 GHz x 32 FP X 20 cores x 1680 CPUs)
  - CPU: Intel Xeon Gold 6148(2.4 GHz, 20 cores, 2 CPUs/node)
  - Memory: DDR4-2666 96GB/node
    - Memory BW: 255 GB/s/node
  - Interconnect: InfiniBand EDR (12.6 GB/s, bidirectional)

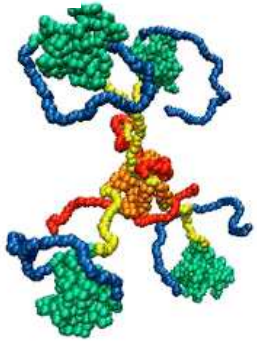
CafeMol: Coarse-grained biomolecular simulation software

H.Kenzaki, et al, J. Chem. Theor. Chem. (2011)

- CafeMol
  - Coarse-grained (CG) Protein/Nucleic acid/lipid models
  - CafeMol 3.0 source code and documentation are released at http://www.cafemol.org
  - Takada Lab (Kyoto Univ)







CG protein model: Go-like model

C. Clementi, H. Nymeyer, and J.N. Onuchic, J. Mol. Biol. (2000)

Based on the energy landscape theory

$$V_{protein} = V_{local} + V_{go} + V_{ex}$$

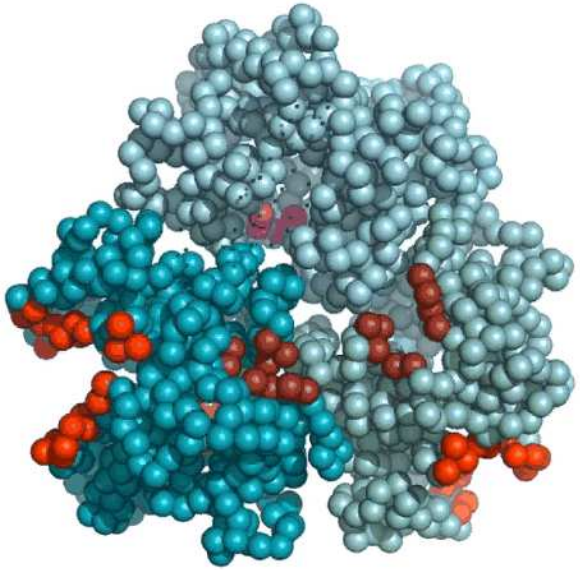
θ: bond angle
ϕ: dihedral angle
(0 means native state)

$$V_{local} = K_b \sum_i \left(r_{i,i+1} - r_{0i,i+1} \right)^2 + K_\theta \sum_i \left(\theta_i - \theta_{0i} \right)^2$$

$$+ K_\phi^1 \sum_i \left(1 - \cos\left(\phi_i - \phi_{0i} \right) \right) + K_\phi^3 \sum_i \left(1 - \cos 3\left(\phi_i - \phi_{0i} \right) \right)$$

$$V_{go} = \varepsilon_{go} \sum_{i,j}^{native} \left[5 \left(\frac{r_{0ij}}{r_{ij}} \right)^{12} - 6 \left(\frac{r_{0ij}}{r_{ij}} \right)^{10} \right]$$

$$V_{ex} = \varepsilon_{ex} \sum_{i,j}^{nonnative} \left(\frac{\sigma}{r_{ij}} \right)^{12}$$



X. Yao, H. Kenzaki, S. Murakami, and S. Takada, *Nature Comm.* (2010)

CG DNA model: 3SPN.1 force field (local, base pair and excluded volume interactions)

E.J. Sambrisiki, D.C. Schwartz, and J.J. de Pablo,Knotts, Biophys. J. (2009)

$$V_{dna} = V_{local} + V_{stack} + V_{bp} + V_{ex} + V_{qq} + V_{solv}$$

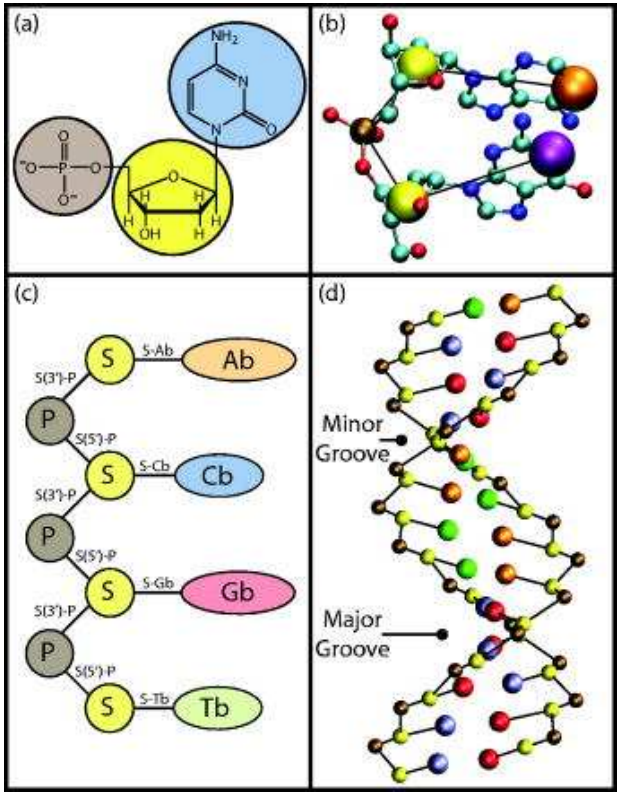
$$V_{local} = K_{b1}\sum_i \left(r_{i,i+1} - r_{0i,i+1}\right)^2 + K_{b2}\sum_i \left(r_{i,i+1} - r_{0i,i+1}\right)^4$$
$$+ K_\theta \sum_i \left(\theta_i - \theta_{0i}\right)^2 + K_\phi \sum_i \left(1 - \cos\left(\phi_i - \phi_{0i}\right)\right)$$

$$V_{stack} = 4\varepsilon_1 \sum_{i,j}^{N_{st}} \left[\left(\frac{\sigma_{0ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{0ij}}{r_{ij}}\right)^{6} \right]$$

$$V_{bp} = \sum_{i,j}^{N_{bp}} 4\varepsilon_{bpi} \left[5\left(\frac{r_{0ij}}{r_{ij}}\right)^{12} - 6\left(\frac{r_{0ij}}{r_{ij}}\right)^{10} \right]$$

$$V_{ex} = 4\varepsilon_1 \sum_{i,j}^{N_{ex}} \left[\left(\frac{\sigma_0}{r_{ij}}\right)^{12} - \left(\frac{\sigma_0}{r_{ij}}\right)^{6} \right] + \varepsilon_1 \,(if\ \ r_{ij} < d_{cut}),$$
$$= 0 \,(if\ \ r_{ij} > d_{cut})$$

θ: bond angle
ϕ: dihedral angle
(0 means B-type DNA)



CG DNA model: 3SPN.1 force field (electrostatic and solvation interactions)

$$V_{qq} = \sum_{i,j}^{N} \left(\frac{q_i q_j}{4\pi\varepsilon_0 \varepsilon(T,C) r_{ij}} \right) e^{-r_{ij}/\kappa_D}$$

Debye-Huckel theory

$$\varepsilon(T,C) = \varepsilon(T) a(C)$$

ε=78

$$\varepsilon(T) = 249.4 - 0.788T/K + 7.20\times10^{-4}(T/k)^2$$

$$a(C) = 1.000 - 0.2551C/M$$

$$+5.151\times10^{-2}(C/M)^2 - 6.889\times10^{-3}(C/M)^3$$

Debye length

$$\kappa_D = \left(\frac{\varepsilon_0 \varepsilon RT}{2N_A^2 e_q^2 I} \right)$$

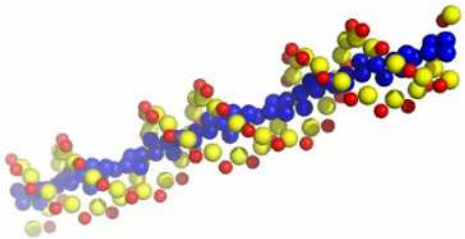
$$V_{solv} = \sum_{i<j}^{Nsolv} \varepsilon_s \left[1 - e^{-a(r_{ij}-r_s)} \right]^2 - \varepsilon_s$$

α^{-1} = 5.333A
r_s = 13.38A
$\varepsilon_0 = 0.504982\varepsilon$

$$\varepsilon_s = \varepsilon_N A_I$$

$$e_N = e_0 (1 - [1.40418 - 0.268231 N_{nt}]^{-1})$$

$$A_I = 0.474876 (1 + \{0.148378 + 10.9553 [Na^+]\}^{-1})$$



計算規模と並列化

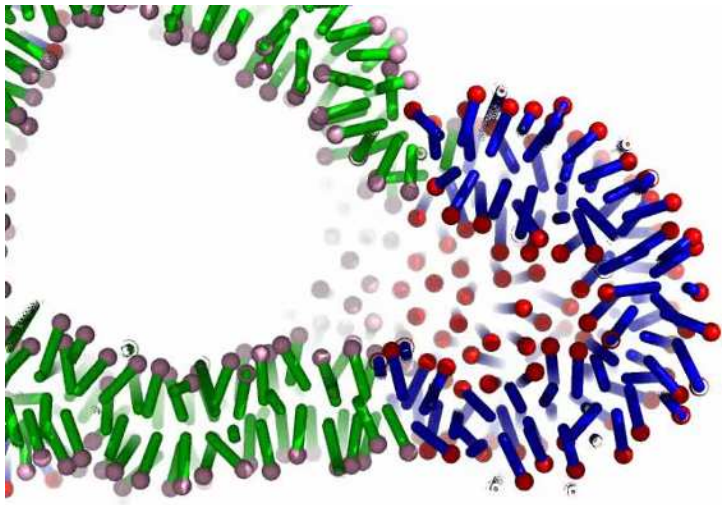
- 計算規模
  - 数十から数万粒子数程度
  - ヌクレオソーム1個の系：1,854粒子
- 並列化
  - 系の時間発展をMPIとOpenMPでハイブリッド並列化
    - 数十並列程度の並列化
    - 力の計算はネイバリングリスト方式
  - レプリカ交換によるMPI並列
    - 通信量が少ないので大規模並列化
    - 平衡量を求める計算に使える

ネイバリングリストの例 (2体の相互作用、i<jだけをリストアップ)

i	1				2		3			4		5			6		7		8
j	2	4	7	9	5	8	4	5	9	7	8	6	7	9	8	9	8	9	9

力場の計算はネイバリングリスト方式

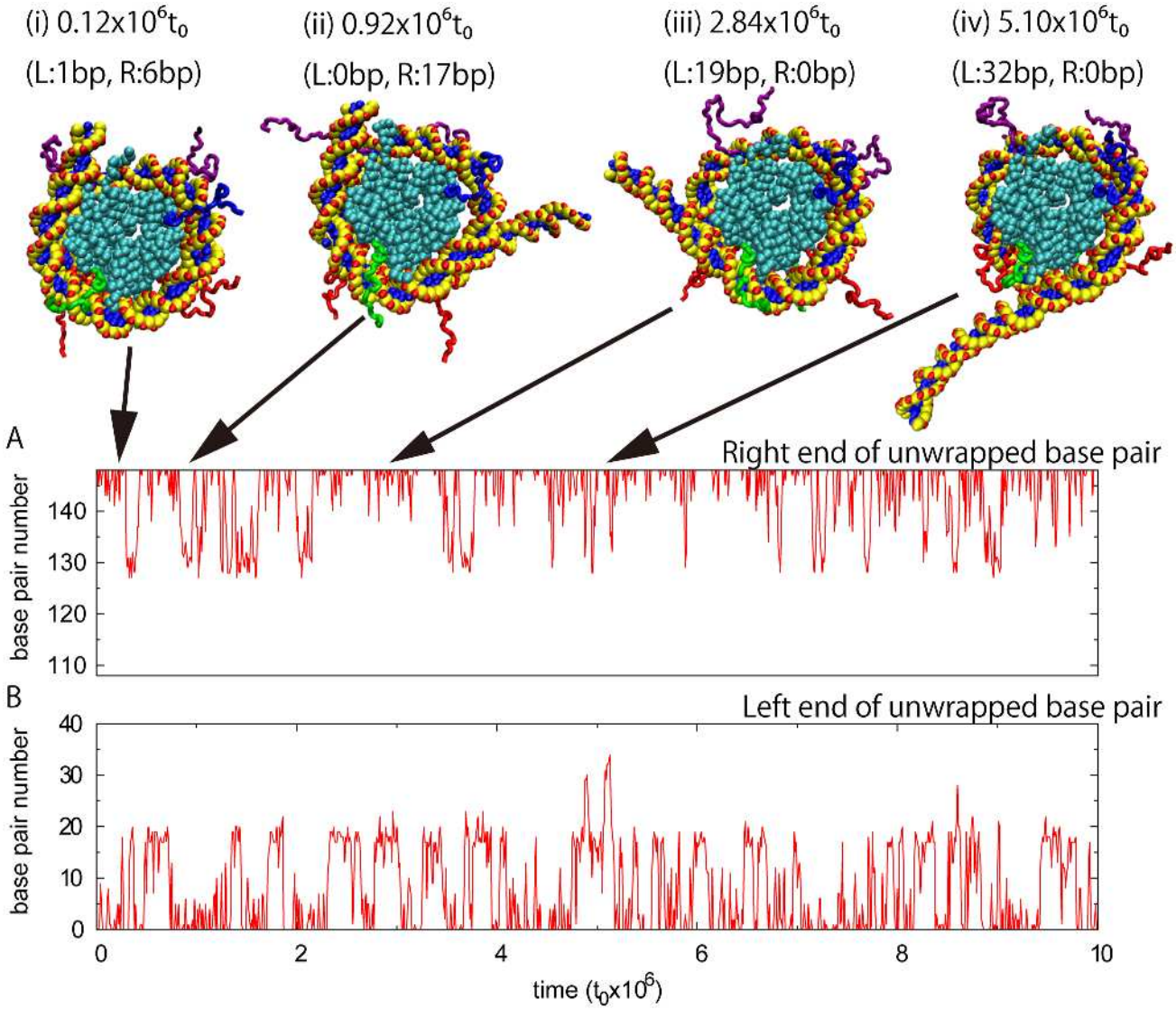
- 100stepに1回程度計算
- ネイバリングリストはノード毎に分割
  - ローカルな相互作用以外は基本的に2体の相互作用の形になるので、あらかじめ各プロセスで計算する1つめの粒子iのリストを作っておく
  - 各プロセスは割り当てられた粒子iのリストについて力場を計算する。
  - 最も大きな配列(粒子数x100〜10,000)
- 相互作用の到達範囲毎にネイバリングリストを作成
  - ローカルな相互作用
    - ボンド長、ボンド角、二面角
    - 計算を始める前にネイバリングリストを作成しておく
  - 近距離相互作用(<20A)
    - 郷ポテンシャル、排除体積、塩基対
  - 中距離相互作用(20-50A)
    - 疎水相互作用
    - 多体のため複雑なネイバリングリストに
  - 遠距離相互作用(>50A)
    - 静電相互作用
    - イオン強度により相互作用距離が変化



Simulation of nucleosome

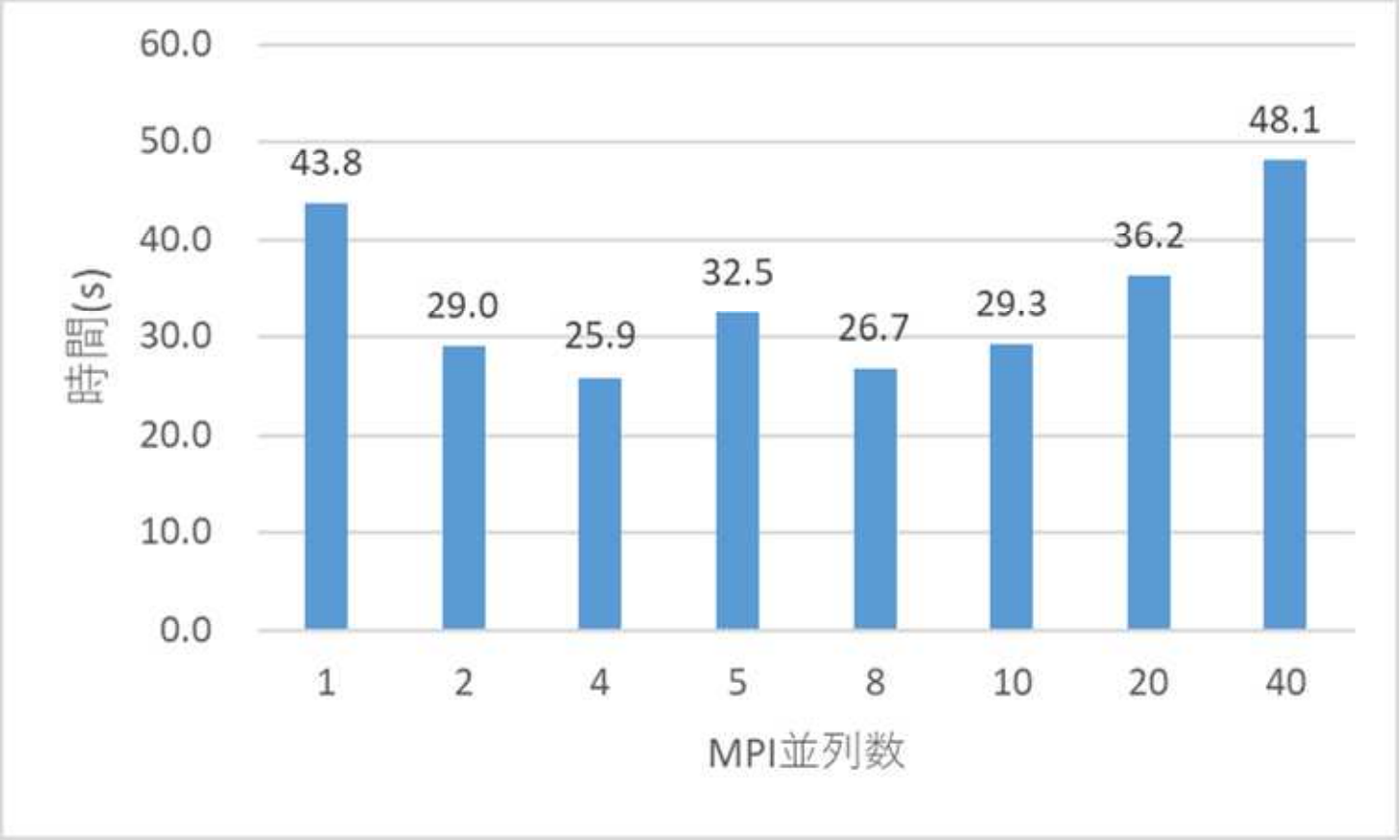
H.Kenzaki and S.Takada, PLoS. Comp. Biol. (2015)

Ion strength=300mM



Skylakeでの性能測定 nucleosome (10^5step, 100mM)

1 node (40 cores)利用で、MPI並列数xスレッド並列数=40を固定して、
MPI並列数を変えて性能測定
コンパイラは、インテルコンパイラ17.0.4
最適化オプションは-fopenmp –O3 –xCORE-AVX512



1nodeでは4MPI並列x10OpenMP並列のときに一番高い性能

FX100との比較 nucleosome (10^5step, 100mM)

	FX100(32 cores) -Kopenmp -Kfast -Kparallel 2MPIx16OpenMP	Skylake(40 cores) -fopenmp -O3 -xCORE-AVX512 4MPIx10OpenMP
force	36.7	18.1
_force(comm)	6.2	5.5
_force(local)	8.8	2.0
_force(go)	1.4	0.2
_force(pnl)	11.8	2.2
_force(ele)	5.8	7.9
random	7.0	4.0
neighbor	12.7	1.9
update	3.3	2.5
output	0.5	1.0
ope	59.9	21.9
comm	6.8	5.9
main_loop	65.7	27.8

Skylakeの方が2倍以上高速化
静電相互作用以外の相互作用とネイバリングリストで4-6倍

静電相互作用の計算方法

1. **ネイバリングリスト**iele2mp(2, lele)**を各プロセスで計算**
   i < jとなるペアだけを格納
2. simu_force.F90

```
!$omp parallel private(tn)
call simu_force_ele(force_mp_l(1, 1, tn))
```

スレッド毎にforce_mp**の異なる領域を用意**

3. simu_force_ele.F90

```
!$omp do private(…)
do iele1 = 1, lele
  imp1 = iele2mp(1, iele1)
  imp2 = iele2mp(2, iele2)
  v21(1:3) = xyz_mp(1:3, imp2) – xyz_mp(1:3, imp1)
  dist2 = v21(1)**2 + v21(2)**2 + v21(3)**2
  if(dist2 > cutoff2) cycle
  dist1 = sqrt(dist2)
  rdist1 = 1.0/dist1
  dvdw_dr = coef(iele)*rdist1*rdist1*(rdist1+rcdist)*exp(-dist1*rcdist)
  force_mp(1:3, imp1) = force_mp(1:3, imp1) – dvdw_dr*v21(1:3)
  force_mp(1:3, imp2) = force_mp(1:3, imp2) + dvdw_dr*v21(1:3)
end do
!$omp end do nowait
```

SIMD**化困難**
!$omp atomic**では遅くなる**

静電相互作用のSIMD化

1. ネイバリングリストiele2charge_k(ncharge, ncharge_mpi)を各プロセスで計算。
   電荷をもっている粒子のリストicharge2mp(ncharge)と座標xyz_ele(3, ncharge)を作っておく。
   タンパク質は4/20が、DNAは1/3の粒子が電荷をもっているため。
2. simu_force.F90

```
!$omp parallel private(tn)
call simu_force_ele2(force_mp_l(1, 1, tn))
```

3. simu_force_ele2.F90

```
!$omp do private(…)
do icharge1 = 1, ncharge
  imp1 = icharge2mp(icharge)
  for(1:3) = 0.0
  do iele = 1, lele_k(icharge)
    jcharge = iele2charge_k(iele, icharge)
    v21(1:3) = xyz_ele(1:3, icharge) – xyz_ele(1:3, jcharge)
    …
  for(1:3) = for(1:3) + dvdw  dr*v21(1:3)
  end do
  force_mp(1:3, imp1) = force_mp(1:3, imp1) + for(1:3)
end do
!$omp end do nowait
```

電荷を持った粒子だけの座標

SIMD化OK
ただし計算量は
本来必要な量の2倍
計算時間は7.9s→6.1s

静電相互作用のデータ構造変換

- Array of Structure(AoS)からStructure of Array(SoA)にデータ構造を変換。
  - xyz_ele(1:3, ncharge)をxyz_ele(ncharge, 1:3)に変換。

	Skylake(40 cores)	Skylake(40 cores) SIMD	Skylake(40 cores) SIMD+DATA
force	18.1	16.1	14.8
_force(comm)	5.5	5.1	5.2
_force(local)	2.0	2.1	2.0
_force(go)	0.2	0.2	0.2
_force(pnl)	2.2	2.2	2.2
_force(ele)	7.9	6.1	4.6
random	4.0	4.0	3.9
neighbor	1.9	1.8	1.9
update	2.5	2.5	2.5
output	1.0	1.1	1.0
ope	21.9	20.6	19.0
comm	5.9	5.6	5.6
main_loop	27.8	26.2	24.6

計算時間は6.1s→4.6s

まとめ

- Skylakeでのベンチマーク結果
  - 1ノードでは、4MPI並列x10スレッドの時に一番高い性能。
  - FX100に対して2倍以上の高速化。
    - 静電相互作用は余り変わらない。
    - 静電相互作用以外やネイバリングリングリストの計算で4-6倍の差。
- Skylakeでの静電相互作用のチューニング
  - SIMD化により1.3倍高速化。
  - データ構造を変えることにより1.3倍高速化。

FUJITSU

shaping tomorrow with you

SS研 メニーコア時代のアプリ性能検討WG

[A.I. 3]
CafemolのFX100とIntel Skylakeの性能差について

2018年7月20日

富士通株式会社
TCソリューション事業本部
渡邊 健太

目次

- 目的
- 評価環境
- 現状分析
- チューニング
- まとめ

目的

FUJITSU

[A.I. 3] FX100おける cafemol の性能を分析する

- Intel Skylakeとの性能差の原因を調査する
- FX100で高速化の検討

剣崎さんから評価対象のサブルーチンを選定いただいています。

[メールの一部を抜粋]

> ○調べていただきたいサブルーチン
> simu_force_bond.F90(タイマーの名称: _force(bond))
> simu_force_angle.F90(タイマーの名称: _force(angle))
> simu_force_nlocal_go.F90(タイマーの名称: _force(go))
>
> bondとangleは鎖で繋がった局所的な2体、3体の相互作用で、メモリアクセスが連続的
> になっています。
> goはランダムアクセスを含むような2体の相互作用となっています。
> どれもskylakeにくらべて6倍程度遅くなっています。

評価環境

FUJITSU

FX100 と Intel Skylake

		FX100	Intel Skylake
CPU	名称	SPARC64 XIfx	Intel Xeon Gold 6148
	動作周波数	**1.50 GHz**	**[Base] 2.40 GHz** **[MAX] Normal : 3.10 GHz** **AVX512 : 2.20 GHz**
	コア数	32コア + アシスタントコア	20コア
	キャッシュメモリ	L1I$, L1D$: 64KB/コア L2$: 24MB/CPU	L1I$, L1D$: 32KB/コア L2$: 1MB L3$: 27.5MB
	理論ピーク性能	768GFlops/CPU (倍精度)	1,408GFlops/CPU (AVX512,倍精度)
	メモリ帯域	(240+240)GB/s/CPU	128GB/s/CPU (DDR-2666, 6chnnels)
ノード	CPU数, コア数	**1CPU/ノード,** **(32コア + アシスタントコア)/ノード**	**2CPU/ノード, 40コア/ノード**
	理論ピーク性能	768GFlops/CPU (倍精度)	2,816GFlops/CPU (AVX512,倍精度)
	メモリ帯域	(240+240)GB/s	256GB/s
コンパイラー		富士通コンパイラ	Intel Compiler v18.1

現状分析 (1/5)

FUJITSU

■ 各計算機の性能を確認

■ 測定条件

[コンパイルオプション]
FX100 : -Kopenmp,fast,parallel,optmsg=2 –Qa,m,p,t,x
SKL　: -O3 –qopenmp -qopt-report=5

■ 測定結果

		FX100		Intel Skylake		
		①	②	③	④	
評価環境	測定者	渡邊	剣崎さん	渡邊	剣崎さん	性能値の比較
	計算機	FX100 (1.5GHz)	FX100 (1.975GHz)	Skylake (2.40GHz)	Skylake (2.40GHz)	② / ④
	並列数	2p x 16t	2p x 16t	4p x 10t	4p x 10t	
タイマーの結果	_force(bond)	0.6316	0.462	0.1142	0.0912	5.1
	_force(angle)	2.0688	1.5737	0.2970	0.2803	5.6
	_force(go)	1.8975	1.4399	0.2090	0.1935	7.4

#1　　#2

#1 クロック周波数の差と同じ性能差
#2 BIOS設定SNC(Sub-NUMA Clustering)が違う?
　③ : SNC disable
　④ : SNC enable

現状分析 (2/5)

FUJITSU

■ コンパイラの最適化状況

コンパイラ最適化	_force(bond)		_force(angle)		_force(go)	
	FX100	SKL	FX100	SKL	FX100	SKL
SIMD	×	×	×	×	×	×
SWP	×	-	×	-	×	-
FULLUNROLLING*1	○	○	○	○	○	○
UNSWITCHING*2	○	-	○	-	×	-
PREFETCH	×	×	○	×	×	×

*1 : doループにではなく、ループ内部の配列式に対して適応
*2 : ループ内で不変な不等式をループ外に出す最適化

■ 実行効率の差

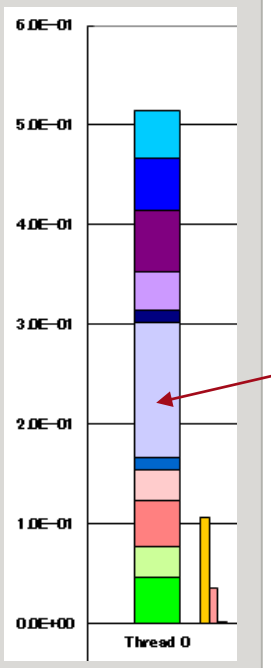
SIMD化されていないため、各サブルーチンの実行効率の差は下表となる

サブルーチン	① FX100(1.975GHz)	② SKL (3.1GHz) *No-SIMD	③ No-SIMD時のピーク性能比 FX100 / SKL	実行効率の差 ① / ② x ③
_force(bond)	0.462	0.0912	FX100 / SKL = 0.51	2.6
_force(angle)	1.5737	0.2803	FX100 : 252.8GFlops (1.975GHz x 32c x 4)	2.9
_force(go)	1.4399	0.1935	SKL : 496.0GFlops (3.100GHz x 40c x 4)	3.8

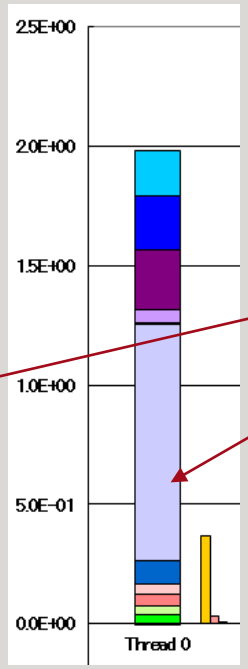
現状分析 (3/5)

FUJITSU

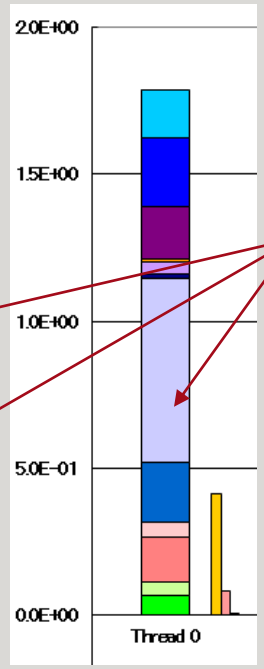
■ PA情報(実行時間内訳)



force_bond



force_angle



force_go

	_force(bond)	_force(angle)	_force(go)
4命令コミット	9.14%	9.68%	9.27%
2/3命令コミット	10.24%	11.42%	13.08%
1命令コミット	12.09%	12.42%	9.98%
浮動小数点演算待ち	26.53%	50.01%	35.12%
整数演算待ち	2.41%	4.77%	11.27%
浮動小数点ロード L1アクセス待ち	5.85%	2.26%	2.96%
浮動小数点ロード L2アクセス待ち	9.21%	2.52%	8.57%
整数ロード L1アクセス待ち	5.97%	1.90%	2.65%
整数ロード L2アクセス待ち	8.88%	1.81%	3.58%

- ✓ 命令スケジュールの改善で高速化できる部分(浮動小数点/整数演算待ち、浮動小数点/整数L1アクセス待ち)

_force(bond)	_force(angle)	_force(go)
40.76%	58.94%	52.00%

- ✓ 各サブルーチンの××命令コミットの割合は30%を超えており、命令数もボトルネックになっている。

現状分析 (4/5)

FUJITSU

PA情報(メモリ・キャッシュスループット情報)@FX100

メモリ・キャッシュスループット情報

サブルーチン	L1ビジー率	L2ビジー率	メモリビジー率	L2スループット	メモリスループット
_force(bond)	21%	7%	0%	32.08GB/s	0.00GB/s
_force(angle)	19%	2%	0%	8.13GB/s	0.00GB/s
_force(go)	23%	5%	0%	20.77GB/s	0.00GB/s

キャッシュミス情報

サブルーチン	L1Dミス率	L1Dミスdm率	L2ミス率	L2ミスdm率	μDTLBミス率	mDTLBミス率
_force(bond)	2.63%	91.05%	0.00%	70.27%	1.44%	0.00%
_force(angle)	0.51%	81.49%	0.00%	83.91%	0.33%	0.00%
_force(go)	1.15%	87.43%	0.00%	85.24%	0.34%	0.00%

- ✓ メモリビジー率・メモリスループットから、メモリアクセスはほとんど無い。
- ✓ L1キャッシュミス率は、理論値(3.15%)以下である。

現状分析 (5/5)

FUJITSU

■ 基本プロファイル情報の採取

[実行時のコマンド]
fipp -C –d fipp_dir -I hwm,cpu –P nouserfunc
[プロファイル情報のテキスト出力]

Procedures profile

****
Application - procedures
****

Cost	%	Operation (S)	Barrier	%	Start	End	
19792	100.0000	1873.0138	1695	8.5641	--	--	Application
2551	12.8890	241.4149	1600	62.7205	406	515	simu_tintegral.get_random_number._OMP_1_
2311	11.6764	218.7002	0	0.0000	--	--	__lll_lock_wait
2029	10.2516	192.0169	0	0.0000	5	143	simu_force_pnl_
2015	10.1809	190.6891	0	0.0000	--	--	__pthread_mutex_unlock_usercnt
1990	10.0546	188.3222	0	0.0000	45	87	simu_neighbor_list._OMP_1_
1276	6.4470	120.7537	0	0.0000	4	88	simu_force_ele2_
1023	5.1688	96.8112	0	0.0000	7	424	simu_force_pnl2_
806	4.0724	76.2771	0	0.0000	510	549	mt_stream.mt_genrand_int32_
538	2.7183	50.9133	0	0.0000	1032	1080	gf2xe.mult_i32_
496	2.5061	46.9388	0	0.0000	84	124	simu_neighbor_list_ele2._PRL_3_
443	2.2383	41.9240	0	0.0000	--	--	__g_dsin
416	2.1019	39.3690	0	0.0000	708	729	mt_stream.mt_genrand_double1_
413	2.0867	39.0839	0	0.0000	14	113	simu_force_nlocal_go_
378	1.9099	35.7725	0	0.0000	288	376	simu_force_fdih.calc_force_fdih_
231	1.1671	21.8582	0	0.0000	6	329	mloop_flexible_local_
186	0.9398	17.6023	0	0.0000	166	236	simu_force_fdih.calc_phi_
152	0.7680	14.3840	0	0.0000	--	--	__GI___gettimeofday_internal
146	0.7377	13.8166	0	0.0000	4	191	simu_force_dih_
139	0.7023	13.1556	0	0.0000	--	--	__g_dscn2
131	0.6619	12.3968	0	0.0000	13	770	simu_neighbor_assign_

コンパイルリスト (_force(bond))

```
55                !$omp do private(imp1,imp2,v21,dist,ddist,ddist2,for, &
56                !$omp&          force,efull,iunit,junit,isys,istat)
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    UNSWITCHING
                  <<< Loop-information  End >>>
57    1  p  2s       do ibd = ksta, kend
58    1
59    1  p  2v         imp1 = ibd2mp(1, ibd)
60    1  p  2v         imp2 = ibd2mp(2, ibd)
61    1
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    FULL UNROLLING
                  <<< Loop-information  End >>>
62    1  p  6v         v21(1:3) = xyz_mp_rep(1:3, imp2,irep) - xyz_mp_rep(1:3, imp1,irep)
63    1
64    1  p  2v         dist = sqrt(v21(1)**2 + v21(2)**2 + v21(3)**2)
65    1  p  2v         ddist = dist - bd_nat(ibd)
66    1  p  2v         ddist2 = ddist**2
67    1
68    1               ! calc force
69    1  p  2v         for = (coef_bd(1, ibd) + 2.0e0_PREC * coef_bd(2, ibd) * ddist2) * &
70    1                  (-2.0e0_PREC * ddist / dist)
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    FULL UNROLLING
                  <<< Loop-information  End >>>
71    1  p  6m         force(1:3) = for * v21(1:3)
72    1
73    2  p  2v         if(inmgo%i_multi_mgo == 0) then
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    FULL UNROLLING
                  <<< Loop-information  End >>>
74    2  p  6           force_mp(1:3, imp1) = force_mp(1:3, imp1) - force(1:3)
75    2  p  6           force_mp(1:3, imp2) = force_mp(1:3, imp2) + force(1:3)
```

→ メインループの最適化情報
SIMD/SWPはなし

→ インダイレクトアクセス

→ If文 ループ中で不変

```
76    2
77    2  p  2v       else
78    2               ! calc energy
79    2  p  2v         efull = (coef_bd(1, ibd) + coef_bd(2, ibd) * ddist2) * ddist2
80    2  p  2v         iunit = imp2unit(imp1)
81    2  p  2v         junit = imp2unit(imp2)
82    2  p  2m          ene_unit(iunit, junit) = ene_unit(iunit, junit) + efull
83    2
84    2  p  2v         isys = ibd2sysmbr_mgo(1, ibd)
85    3  p  2s         if(isys == 0) then
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    FULL UNROLLING
                  <<< Loop-information  End >>>
86    3  p  6s           force_mp(1:3, imp1) = force_mp(1:3, imp1) - force(1:3)
87    3  p  6s           force_mp(1:3, imp2) = force_mp(1:3, imp2) + force(1:3)
88    3  p  2s         else
89    3  p  2s           istat = ibd2sysmbr_mgo(2, ibd)
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    FULL UNROLLING
                  <<< Loop-information  End >>>
90    3  p  6s           force_mp_mgo(1:3, imp1, istat, isys) = force_mp_mgo(1:3,
imp1, istat, isys) - force(1:3)
91    3  p  6s           force_mp_mgo(1:3, imp2, istat, isys) = force_mp_mgo(1:3,
imp2, istat, isys) + force(1:3)
92    3  p  2          end if
93    2  p  2v       end if
94    1  p  2v     end do
95              !$omp end do nowait
```

→ If文

- ✓ サブルーチン内のメインループはIntel SkylakeでもSIMD化されていない
- ✓ FULLUNROLLINGはループ内の配列式に対して適応

コンパイルリスト (_force(angle))

```
      52                !$omp do private(imp1,imp2,imp3,v21,v32,c11,c22,c21, &
      53                !$omp&          co_theta,dba,t3,for,force_21,force_32, &
      54                !$omp&          efull,iunit,junit,isys,istat)
                      <<< Loop-information Start >>>
                      <<<  [OPTIMIZATION]
                      <<<    UNSWITCHING
                      <<<    PREFETCH      : 2
                      <<<     ba_nat: 2
                      <<< Loop-information  End >>>
      55    1  p   s       do iba=ksta,kend
      56    1
      57    1  p   m         if (coef_ba(1, iba) < ZERO_JUDGE .and. coef_ba(2, iba) <
ZERO_JUDGE) cycle
      58    1
      59    1  p   s         imp1 = iba2mp(1, iba)
      60    1  p   s         imp2 = iba2mp(2, iba)
      61    1  p   s         imp3 = iba2mp(3, iba)
      62    1
                    <<< Loop-information Start >>>
                    <<<  [OPTIMIZATION]
                    <<<    FULL UNROLLING
                    <<< Loop-information  End >>>
      63    1  p  3s          v21(1:3) = xyz_mp_rep(1:3, imp2, irep) - xyz_mp_rep(1:3, imp1,
irep)
      64    1  p  3s          v32(1:3) = xyz_mp_rep(1:3, imp3, irep) - xyz_mp_rep(1:3, imp2,
irep)
      65    1
      66    1  p   s         c11 = v21(1) * v21(1) + v21(2) * v21(2) + v21(3) * v21(3)
      67    1  p   s         c22 = v32(1) * v32(1) + v32(2) * v32(2) + v32(3) * v32(3)
      68    1  p   s         c21 = v32(1) * v21(1) + v32(2) * v21(2) + v32(3) * v21(3)
      69    1
      70    1  p   s         co_theta = - c21 / sqrt(c11 * c22)
      71    1
      72    2  p   s         if(co_theta > 1.0e0_PREC) then
      73    2  p   m           co_theta = 1.0e0_PREC
      74    2  p   s         else if(co_theta < -1.0e0_PREC) then
      75    2  p   s           co_theta = -1.0e0_PREC
      76    2  p   s         end if
      77    1
      78    1  p   s         dba = acos(co_theta) - ba_nat(iba)
      79    1
      80    1  p   s         t3 = c11 * c22 - c21**2
      81    2  p   s         if(t3 <= 1.0e0_PREC) then
      82    2  p   s           t3 = 1.0e0_PREC
      83    2  p   s         end if
      84    1
```

→ メインループの最適化情報
SIMD/SWPはなし

→ cycle文

→ インダイレクトアクセス

→ If文

→ If文

```
      85    1                ! calc force
      86    1  p   s          for = 2.0e0_PREC * coef_ba(1, iba) * dba / sqrt(t3) - &
      87    1                     coef_ba(2, iba) / sqrt(c11 * c22)
                    <<< Loop-information Start >>>
                    <<<  [OPTIMIZATION]
                    <<<    FULL UNROLLING
                    <<< Loop-information  End >>>
      88    1  p  3s          force_21(1:3) = for * (v21(1:3) * (c21 / c11) - v32(1:3))
      89    1  p  3s          force_32(1:3) = for * (v32(1:3) * (c21 / c22) - v21(1:3))
      90    1
      91    2  p   s         if(inmgo%i_multi_mgo == 0) then
                    <<< Loop-information Start >>>
                    <<<  [OPTIMIZATION]
                    <<<    FULL UNROLLING
                    <<< Loop-information  End >>>
      92    2  p  3             force_mp(1:3, imp1) = force_mp(1:3, imp1) - force_21(1:3)
      93    2  p  3             force_mp(1:3, imp2) = force_mp(1:3, imp2) + force_21(1:3) -
force_32(1:3)
      94    2  p  3             force_mp(1:3, imp3) = force_mp(1:3, imp3) + force_32(1:3)
      95    2
      96    2  p   s          else
                                    :
     114    2  p   s          end if
     115    1
     116    1  p   v        end do
     117                  !$omp end do nowait
```

→ If文 ループ中で不変

- ✓ サブルーチン内のメインループは Intel SkylakeでもSIMD化されていない
- ✓ FULLUNROLLINGはループ内の配列式に対して適応
- ✓ cycle文あり

ソースコード (force_go)

```
      67                !$omp do private(imp1,imp2,rcut_off2,v21,dist2,roverdist2,roverdist4, &
      68                !$omp&           roverdist8,roverdist12,roverdist14,dgo_dr,for,imirror)
      69    1  p          do icon=ksta,kend
      70    1
      71    1  p            imp1 = icon2mp(1, icon)
      72    1  p            imp2 = icon2mp(2, icon)
      73    2  p            if (iclass_mp(imp1) == CLASS%RNA .AND. iclass_mp(imp2) ==
CLASS%RNA) then
      74    2  p              rcut_off2 = rcut_off2_rna
      75    2  p            else
      76    2  p              rcut_off2 = rcut_off2_pro
      77    2  p            endif
      78    1
      79    2  p            if(inperi%i_periodic == 0) then
                      <<< Loop-information Start >>>
                      <<<  [OPTIMIZATION]
                      <<<    FULL UNROLLING
                      <<< Loop-information  End >>>
      80    2  p  3           v21(1:3) = xyz_mp_rep(1:3, imp2, irep) - xyz_mp_rep(1:3, imp1,
irep)
      81    2  p            else
                      <<< Loop-information Start >>>
                      <<<  [OPTIMIZATION]
                      <<<    FULL UNROLLING
                      <<< Loop-information  End >>>
      82    2  p  3           v21(1:3) = pxyz_mp_rep(1:3, imp2, irep) - pxyz_mp_rep(1:3, imp1,
irep)
      83    2  p              call util_pbneighbor(v21, imirror)
      84    2  p            end if
      85    1
      86    1          !     v21(1:3) = xyz_mp_rep(1:3, imp2, irep) - xyz_mp_rep(1:3, imp1,
irep)
      87    1
      88    1  p            dist2 = v21(1)*v21(1) + v21(2)*v21(2) + v21(3)*v21(3)
      89    1
      90    1  p            roverdist2 = go_nat2(icon) / dist2
```

→ インダイレクトアクセス

→ If文

→ If文

```
      91    1  p            if(roverdist2 < rcut_off2) cycle
      92    1
      93    1  p            roverdist4 = roverdist2 * roverdist2
      94    1  p            roverdist8 = roverdist4 * roverdist4
      95    1  p            roverdist12 = roverdist4 * roverdist8
      96    1  p            roverdist14 = roverdist12 * roverdist2
      97    1
      98    1  p            dgo_dr = 60.0e0_PREC * coef_go(icon) / go_nat2(icon) *
(roverdist14 - roverdist12)
      99    1
      100   2  p            if(dgo_dr > DE_MAX) then
      101   2          !       write (*, *) "go", imp1, imp2, dgo_dr
      102   2  p              dgo_dr = DE_MAX
      103   2  p            end if
      104   1          !     if(dgo_dr > 5.0e0_PREC) dgo_dr = 5.0e0_PREC
      105   1
                      <<< Loop-information Start >>>
                      <<<  [OPTIMIZATION]
                      <<<    FULL UNROLLING
                      <<< Loop-information  End >>>
      106   1  p  3           for(1:3) = dgo_dr * v21(1:3)
      107   1  p  3           force_mp(1:3, imp2) = force_mp(1:3, imp2) + for(1:3)
      108   1  p  3           force_mp(1:3, imp1) = force_mp(1:3, imp1) - for(1:3)
      109   1
      110   1  p          end do
      111              !$omp end do nowait
```

→ cycle文

→ If文

- ✓ サブルーチン内のメインループはIntel SkylakeでもSIMD化されていない
- ✓ UNSWITCHINGは適応されていない
- ✓ FULLUNROLLINGはループ内の配列式に対して適応
- ✓ cycle文あり

現状分析まとめ

FUJITSU

- 3つのサブルーチンの1ノードのFX100とSKLの性能は、5~7倍SKLが良い
  - FX100 (1.5GHz)とFX100 (1.975GHz)の性能差から、CPUの演算性能(周波数)の差が実測性能に直結
  - PA情報からメモリアクセスはほとんど無い
  - FX100/SKLでSIMD化されていない。
    →1ノードのFX100/SKLのピーク演算性能差が2倍程度異なるため、**FX100とSKLの実行効率の差は2.6~3.6倍**になる

- PA情報(実行時間内訳)から、
  - 命令スケジューリングの改善で高速化できる部分は、40~58%ある。
    → **SWP化の検討**
  - 命令数(XX命令コミット)の削減で高速化できる部分は、約30%ある。
    → **SIMD化の検討**

プログラム全体に対する性能改善 1

FUJITSU

- 基本プロファイラから得られた “__lll_lock_wait”に対する対応
- 実行時環境変数を指定

```
export XOS_MMM_L_ARENA_LOCK_TYPE=0
```

  - 「0」を指定することで、malloc要求を並列処理することができる。
  - 「1」がデフォルト値、逐次処理される。

- 測定結果 (プログラム全体の経過時間)

指定なし	指定あり	改善率
130.75 (s)	115.63	13.1%

メインループおよび調査対象のサブルーチンの性能への影響はない。

プログラム全体に対する性能改善 2

FUJITSU

- オーバーヘッドが大きい時間計測ルーチンを置き換える
- タイマールーチンの置換え

オリジナル	変更後	
	FX100	SKL
MPI_wtime	gettod	clockx

- タイマー結果

サブルーチン	FX100			SKL		MPI_wtimeの性能差	タイマー置換え後の性能差
	FX100 (1.5GHz)		FX100 (1.975GHz)				
	MPI_wtime	gettod	gettod	MPI_wtime	clockx		
force_bond	0.6316	**0.4509**	**0.3424**	0.1142	**0.1005**	5.1	**3.4**
force_angle	2.0688	**1.8957**	**1.4398**	0.2970	**0.2809**	5.6	**5.1**
force_go	1.8975	**1.7119**	**1.3002**	0.2090	**0.1999**	7.4	**6.5**

✓ 経過時間が短いサブルーチンほど、タイマーのオーバーヘッドの影響が大きくなる

force_bondのチューニング

FUJITSU

チューニング方針

1. 手動でループアンスイッチングを適応
2. !ocl simd_redundant_vl()を配列式の直前に挿入
   → この指示行を指定しないとSWP化されない。コンパイラのメッセージは、
   「スケジューリング結果を得られなかったため、ソフトウェアパイプライニングを適用できません。」

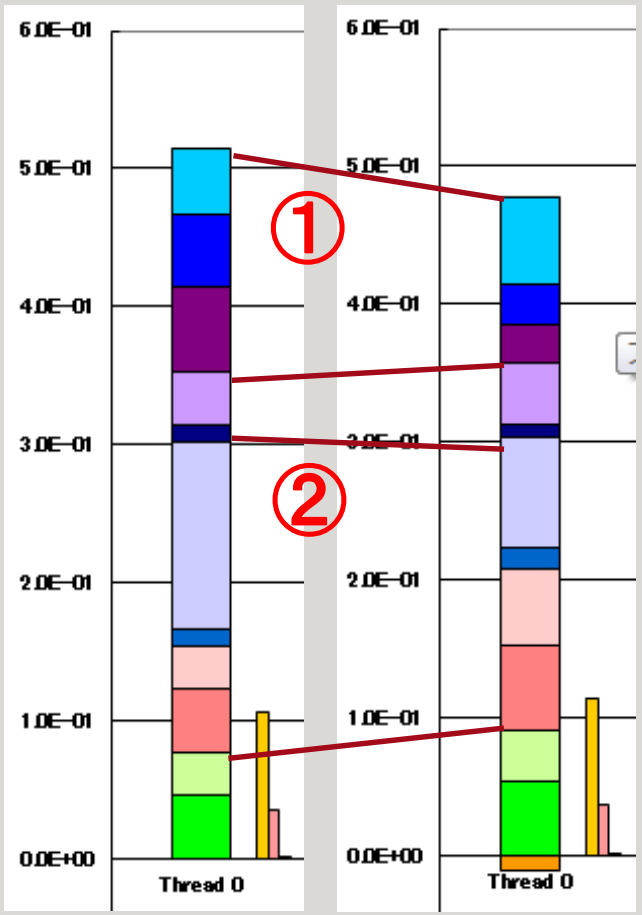
simd_redundant_vl指示行
ループ回転数がSIMD長で割り切れない場合でも、マスク付きSIMD命令を利用し、SIMD実行するループを生成する。

```
      57    1            if(inmgo%i_multi_mgo == 0) then        ← 手動ループアンスイッチング
      58    1
      59    1          !$omp do private(imp1,imp2,v21,dist,ddist,ddist2,for, &
      60    1          !$omp&          force,efull,iunit,junit,isys,istat)
                      <<< Loop-information Start >>>
                      <<<  [OPTIMIZATION]
                      <<<    SOFTWARE PIPELINING
                      <<< Loop-information  End >>>
      61    2  p          do ibd = ksta, kend
      62    2
      63    2  p            imp1 = ibd2mp(1, ibd)
      64    2  p            imp2 = ibd2mp(2, ibd)
      65    2
      66    2            !    write(*,*) "fj*** imp1=",imp1,"imp2=",imp2
      67    2           !ocl simd_redundant_vl(3)
                      <<< Loop-information Start >>>
                      <<<  [OPTIMIZATION]
                      <<<    SIMD(VL: 4)
                      <<<    FULL UNROLLING
                      <<< Loop-information  End >>>
      68    2  p  1v         v21(1:3) = xyz_mp_rep(1:3, imp2,irep) - xyz_mp_rep(1:3,
imp1,irep)
      69    2
      70    2  p            dist = sqrt(v21(1)**2 + v21(2)**2 + v21(3)**2)
      71    2  p            ddist = dist - bd_nat(ibd)
      72    2  p            ddist2 = ddist**2
      73    2
```

```
      74    2              ! calc force
      75    2  p           for = (coef_bd(1, ibd) + 2.0e0_PREC * coef_bd(2, ibd) * ddist2) * &
      76    2                 (-2.0e0_PREC * ddist / dist)
      77    2          !ocl simd_redundant_vl(3)
                      <<< Loop-information Start >>>
                      <<<  [OPTIMIZATION]
                      <<<    SIMD(VL: 4)
                      <<<    FULL UNROLLING
                      <<< Loop-information  End >>>
      78    2  p  1v         force(1:3) = for * v21(1:3)
      79    2          !ocl simd_redundant_vl(3)
      80    2  p  1v           force_mp(1:3, imp1) = force_mp(1:3, imp1) - force(1:3)
      81    2          !ocl simd_redundant_vl(3)
      82    2  p  1v           force_mp(1:3, imp2) = force_mp(1:3, imp2) + force(1:3)
      83    2  p          enddo
      84    1          !$omp end do nowait
      85    1
      86    1            else                                  ← 手動ループアンスイッチング
```

force_bondのチューニングの効果

■ 実行時間内訳の比較



■ チューニング効果
(#) タイマー(gettod)の経過時間から

Original	Tune	向上率
0.4509秒	0.4120秒	9.4%

① 命令コミットのコストは少し改善

→ ループ内の配列式をSIMD化したので命令数が減少した。

② 命令スケジューリングで改善できる部分のSWPによる効果が小さい

- SWPに必要な回転数 :44
(コンパイラメッセージから)
- 実際のループ回転数：約900

force_bondのSWPの効果

FUJITSU

- SWPの効果が小さい = 命令を重ねられていない原因
  - ループ最後のインダイレクトアクセスのあるリダクション演算部に対して、コンパイラは依存関係を見切れず、逐次スケジューリングになっていると考えられる。

```
   79    2              !ocl simd_redundant_vl(3)
   80    2   p   1v             force_mp(1:3, imp1) = force_mp(1:3, imp1) - force(1:3)
   81    2              !ocl simd_redundant_vl(3)
   82    2   p   1v             force_mp(1:3, imp2) = force_mp(1:3, imp2) + force(1:3)
```

  - しかし、データには重なりがあるため、!OCL NORECURRENCEが使えない

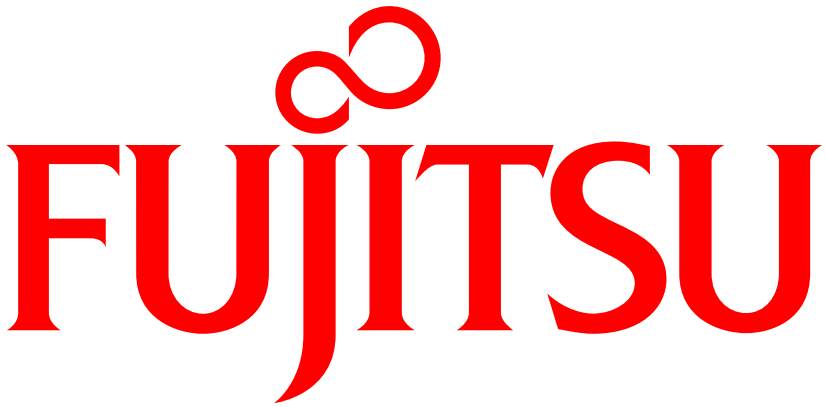
```
fj*** imp1= 1 imp2= 2
fj*** imp1= 1 imp2= 3
fj*** imp1= 3 imp2= 4
fj*** imp1= 4 imp2= 5
fj*** imp1= 4 imp2= 6
fj*** imp1= 6 imp2= 7
fj*** imp1= 7 imp2= 8
fj*** imp1= 7 imp2= 9
fj*** imp1= 9 imp2= 10
fj*** imp1= 10 imp2= 11
          :
```

データに重なりが無いようなレイアウトにすることでコンパイラ最適化(SWP)の効果を最大限に引き出せると考える

まとめ

FUJITSU

- FX100とSkylakeの5~7.4倍の性能差の分析
  - タイマールーチンの置換えにより、性能差は3.4~6.5倍
  - さらに、ピーク演算性能の差を考慮すると、その差は、1.7~3.3倍
  - FX100では、PA情報の採取から、命令数と演算/データアクセス待ちがボトルネックとなっていた。
- 高速化の検討
  - 手動ループアンスイッチングと!ocl simd_redundant_vlの指定により、SIMD化/SWP化を促進することで、約10%性能が改善した。
  - インダイレクトアクセスのデータに対して、データの重なりがないようレイアウトを変更できれば、より最適化の効果を得ることができると考える。
- 残り2ルーチンに対して
  - _force(bond)との一番の違いはcycle文があること。
  - cycle文直前でループ分割し、後半のループに対して、_force(bond)と同様のチューニングをすることが可能



shaping tomorrow with you

2019/2/25
SS研メニーコア時代のアプリ性能WG
第9回会合

生体分子粗視化シミュレータCafeMol タイマーとローカルな力のSIMD化

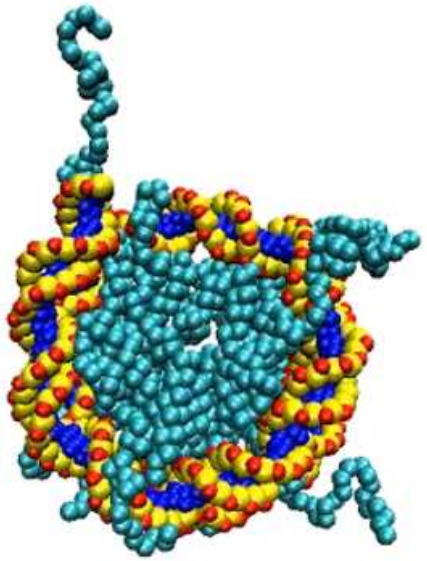
検崎博生
理研 情報システム部

概要

- 粗視化モデルとCafeMolについて
  - 生体分子シミュレーションと粗視化
  - CGタンパク質/DNAモデル
  - 計算方法と並列化
- タイマーによるオーバーヘッド
  - mpi_wtime, system_clock, gettod
- 1ノードでのMPI並列数とスレッド並列数
- ローカルな相互作用のSIMD化
  - ボンド長とボンド角相互作用
  - OpenMPのスレッド並列のチャンクサイズを変更

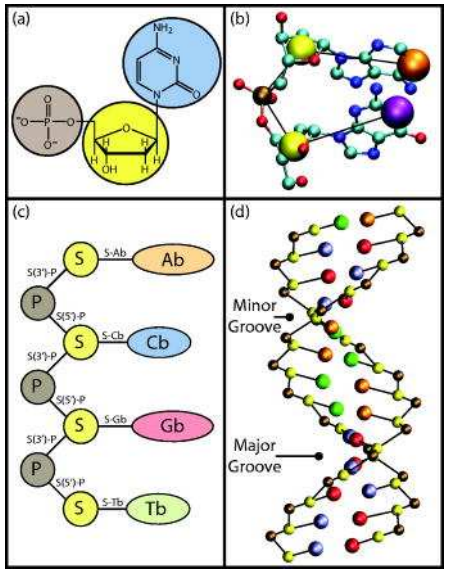
Model

- CafeMol H.Kenzaki, et al, J. Chem. Theor. Chem. (2011)
- Protein
  - AICG2+ model W Li, et al PNAS (2012)
  - FLP model for disordered regions in X-ray structure Knotts et al, J. Chem. Phys. (2007)
- DNA
  - 3SPN.1 model



ローカルなポテンシャル

θ: bond angle
ϕ: dihedral angle
(0 means native state)

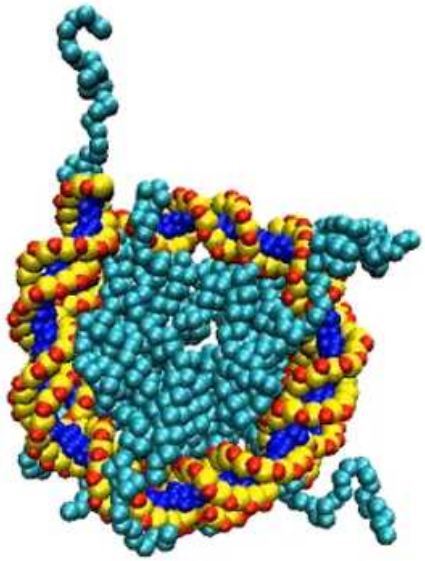


$$V_{local} = K_b \sum_i \left(r_{i,i+1} - r_{0i,i+1} \right)^2 + K_\theta \sum_i \left(\theta_i - \theta_{0i} \right)^2$$

$$+ K_\phi^1 \sum_i \left(1 - \cos\left(\phi_i - \phi_{0i} \right) \right) + K_\phi^3 \sum_i \left(1 - \cos 3\left(\phi_i - \phi_{0i} \right) \right)$$

ボンド長、ボンド角、二面角などの相互作用からなるが、モデルによって詳細はさまざまである。

今回の計算対象と計算方法



- 計算対象
  - ヌクレオソーム1個の系：1,854粒子
- 計算方法
  - ネイバリングリストを作っておく
    - ローカルな相互作用は最初に作っておく
    - それ以外は100stepに1回更新
  - MPIとOpenMPでネイバリングリストを分割して並列計算
    - プロセスとスレッド毎に力を保存して、最後に通信を行い足し合わせる
  - 速度や座標のアップデートは全プロセスで同じものを行う。

ネイバリングリストの例 (2体の相互作用)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
i	1				2		3			4		5			6		7		8
j	2	4	7	9	5	8	4	5	9	7	8	6	7	9	8	9	8	9	9

タイマーの比較 nucleosome (10^5step, 100mM)

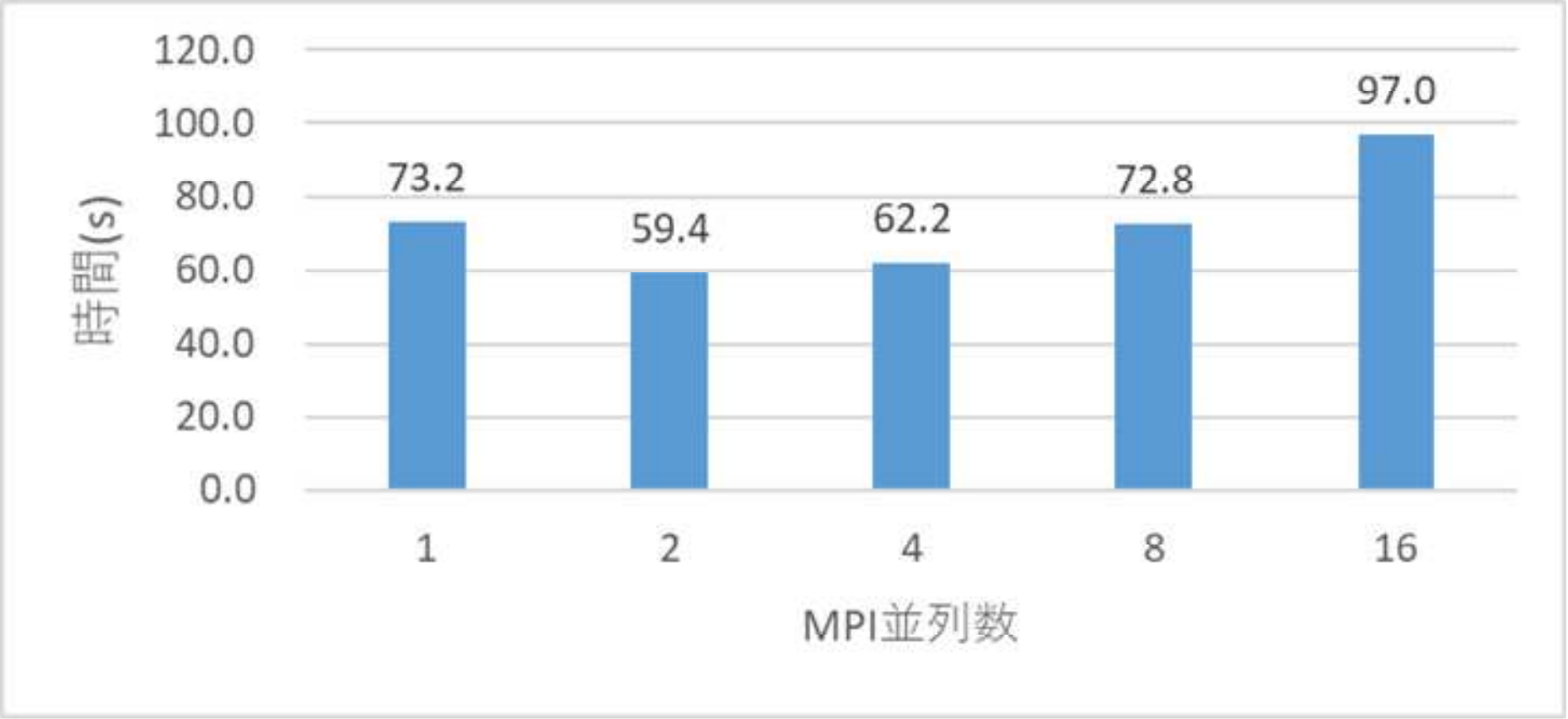
- FX100 1 node: SPARC64 XIfx (1.975GHz, 32 cores, 1 CPU/node)
- コンパイラは、Fujitsu Fortran Compiler 2.0.0-07
- 最適化オプションは-Kopenmp,fast,parallel,ocl,optmsg=2 -Qa,m,p,t,x
- 1ノード (32コア) で2MPI並列X16スレッド並列で実行

	FX100(32 cores) mpi_wtime	FX100(32 cores) system_cloc	FX100(32 cores) gettod
force	38.33	34.92	34.94
_force(comm)	6.12	6.15	6.19
_force(local)	10.42	8.72	8.68
__force(bond)	0.48	0.42	0.38
__force(bangle)	1.54	1.39	1.41
_force(go)	1.43	1.31	1.30
_force(pnl)	11.70	11.55	11.51
_force(ele)	5.87	5.69	5.68
ope	60.52	53.03	52.75
comm	6.75	6.60	6.65
main_loop	67.27	59.62	59.40

mpi_wtimeはオーバーヘッドが大き過ぎる。
system_clockとgettodは同程度。

FX100での性能測定 nucleosome (10^5step, 100mM)

- FX100 1 node: SPARC64 XIfx (1.975GHz, 32 cores, 1 CPU/node)
- **コンパイラは、**Fujitsu Fortran Compiler 2.0.0-07
- **最適化オプションは**-Kopenmp,fast,parallel,ocl,optmsg=2 -Qa,m,p,t,x
- **1ノード (32コア) でMPI並列数X16スレッド並列数=32に固定して実行**



1nodeでは2MPI並列x16OpenMP並列のときに一番高い性能

force(bond)の計算方法

1. **ネイバリングリスト**iele2mp(2, lele)**を各プロセスで計算**
   i < jとなるペアだけを格納
2. simu_force.F90

```
!$omp parallel private(tn)
call simu_force_bond(force_mp_l(1, 1, tn))
```

スレッド毎にforce_mp**の異なる領域を用意**

3. simu_force_bond.F90

```
!$omp do private(....)
  do ibd = ksta, kend
     imp1 = ibd2mp(1, ibd)
     imp2 = ibd2mp(2, ibd)
     v21(1:3) = xyz_mp_rep(1:3, imp2,irep) - xyz_mp_rep(1:3, imp1,irep)
     dist = sqrt(v21(1)**2 + v21(2)**2 + v21(3)**2)
     ddist = dist - bd_nat(ibd)
     ddist2 = ddist**2
     for = (coef_bd(1, ibd) + 2.0e0_PREC * coef_bd(2, ibd) * ddist2) * (-2.0e0_PREC * ddist / dist)
     force(1:3) = for * v21(1:3)
     force_mp(1:3, imp1) = force_mp(1:3, imp1) - force(1:3)
     force_mp(1:3, imp2) = force_mp(1:3, imp2) + force(1:3)
  end do
!$omp end do nowait
```

```
ibd=1 imp1=1 imp2=2
ibd=2 imp1=1 imp3=3
ibd=3 imp1=3 imp2=4
ibd=4 imp1=4 imp2=5
ibd=5 imp1=4 imp2=6
ibd=6 imp1=6 imp2=7
```

force_mp**への足しこみが原因で**SIMD**化できない**

force(bond)のSIMD化

- 方針
  - force_mpへの足し込みをループ内で被らないようする。
  - force_mpがスレッド毎に別なので、各スレッド毎に被らないようする。
  - ボンド長の場合、ネイバリングリストで3つ以上離れていれば被らない。
  - よって、チャンクサイズを1にし、スレッド並列数が3以上であれば、SIMD化が可能なはず。
- tune1
  - 今回のモデルに関係ない部分を削除しておく。
  - チェック用のifなど、if文は全て削除する。
- tune2
  - !omp do schedule(static,1)としてチャンクサイズを1に設定する。
  - この段階では、メモリアクセスが遅くなると予想できる。
- tune3
  - !ocl norecurrence()を付けてSIMD化を行う。
- ボンド各force(bangle)についても同様にSIMD化を行うが、FLPモデルについては、中で関数を読んでいるのでSIMD化に失敗。
  - 他の相互作用でもif文を外せないなどSIMD化できないことはよくある。

force(bond)とforce(bangle)のSIMD化

- tune1: 今回の計算に使わない部分を削除しソースを単純化
- tune2: OpenMPのスレッド並列の順序を変更
- tune3: !ocl norecurrence()を付けてSIMD化
- コンパイルオプションは-Kopenmp,fast,parallel,ocl,optmsg=2 -Qa,m,p,t,x
- 1ノード（32コア）で2MPI並列X16スレッド並列で実行

	FX100(32 cores) tune1	FX100(32 cores) tune2	FX100(32 cores) tune3
force	34.90	35.44	35.00
_force(comm)	6.18	6.21	6.22
_force(local)	8.71	8.99	8.79
__force(bond)	0.37	0.62	0.51
__force(bangle)	1.42	1.44	1.06
_force(go)	1.27	1.29	1.29
_force(pnl)	11.52	11.65	11.47
_force(ele)	5.68	5.67	5.68
ope	52.80	53.24	52.80
comm	6.65	6.66	6.65
main_loop	59.44	59.90	59.44

bondはSIMD化の副作用の方が大きく遅くなった。
bangleはSIMD化により30%程度速くなった。

まとめ

- Timerの比較
  - MPI_WTIMEはオーバーヘッドが大きい。
  - system_clockとgettodは同程度。
- MPI並列とOpenMP並列の兼ね合い。
  - 1ノードでは、2MPI並列x16スレッドの時に一番高い性能。
- FX100でのローカルな相互作用のSIMD化
  - スレッド並列の割当を変えることにより、SIMD化に成功。
    - 複雑な力場を使う場合はif文や関数呼び出しを含むので難しい。
  - 性能の向上は限定的。
    - スレッド並列の割当を変えるオーバーヘッドとの兼ね合い。

3.4　ADVENTURE における多倍長精度演算の利用に向けた検討

名古屋大学情報基盤センター　荻野　正雄

3.4.1　はじめに

ADVENTURE とは、設計用大規模計算力学システム開発プロジェクト(通称 ADVENTURE プロジェクト、https://adventure.sys.t.u-tokyo.ac.jp/)において開発されているオープンソース CAE システムである。大規模メッシュを用いて自然物や人工物を丸ごと詳細にモデル化し、多様な並列分散計算機環境のもとで固体の変形や熱・流体の流れ等の力学解析から可視化、設計最適化までを行える特徴がある。無料公開されていることもあり、教育・研究から産業応用まで幅広く利用されているソフトウェアである。ここでは、有限要素法による 3 次元電磁場解析モジュール ADVENTURE_Magnetic を取り上げる。電磁場には静磁場、低周波電磁場、高周波電磁場など様々な種類があり、対象とする問題の特性に応じて Maxwell 方程式から偏微分方程式を導出する。それらに辺要素有限要素法を適用することで、様々な線形方程式が得られる。例えば、時間調和渦電流問題や高周波電磁場問題では複素対称線形方程式を解くことになる。しかし、係数行列が悪条件になりやすく、共役直交共役勾配(COCG)法などの反復解法で高い精度の収束解を得ることが困難な場合がある。そこで現在、IEEE 754-2008 が規定する四倍精度浮動小数点数や D.H. Bailey や K. Briggs などが提案した倍精度数 2 つを用いる疑似四倍精度数などの多倍長精度演算に着目し、反復計算における丸め誤差を低減させることで、電磁場解析を効率化することを目指している。今回は、富士通 FX100 を含むいくつかの環境において、任意精度・多倍長精度ライブラリの性能評価を行った。また、ライブラリ利用における知識や技術の蓄積と共有を行い、さらに富士通 TCS 環境の課題を明らかにする。

3.4.2　任意精度・多倍長精度演算ライブラリの概要

表 3.4.1 に今回用いた任意精度・多倍長精度演算ライブラリ、実数及び複素数の浮動小数点数型を示す。

表 2.9.1　主な任意精度・多倍長精度浮動小数点数

Library	Precision	Real data type	Complex data type
C data type	extended	long double	long double _Complex
Fortran data type	quadruple	real(kind=16)	complex(kind=16)
libquadmath	quadruple	__float128	__complex128
Intel's _Quad	quadruple	_Quad	complex<_Quad>
QD	pseudo-quad	dd_real	complex<dd_real>
Fujitsu's fast_dd	pseudo-quad	dd_real	complex<dd_real>

ARPREC	arbitrary	mp_real	mp_complex
exflib	arbitrary	exfloat	complex<exfloat>
MPFR/GMP	arbitrary	mpfr_t	-
MPC		-	mpc_t

四倍精度としては、Fortran データ型、GCC 拡張である C 言語の libquadmath ライブラリ(https://gcc.gnu.org/onlinedocs/libquadmath/)、Intel コンパイラ拡張である C 言語の_Quad を用いる。libquadmath ライブラリは GCC 4.6.0 以降から利用可能である。QD ライブラリ(http://crd-legacy.lbl.gov/~dhbailey/mpdist/)は倍精度浮動小数点数を 2 つ用いる double-double 形式の疑似四倍精度演算ライブラリである。富士通 fast_dd は、富士通 TCS に含まれる高速 4 倍精度基本演算ライブラリであり、QD と同じく double-double 形式の疑似四倍精度である。比較用として、C データ型の拡張倍精度、任意精度演算ライブラリである ARPREC(http://crd-legacy.lbl.gov/~dhbailey/mpdist/)、exflib(http://www-an.acs.i.kyoto-u.ac.jp/~fujiwara/exflib/)、並びに MPFR(http://www.mpfr.org/)とその複素数版である MPC(http://www.multiprecision.org/)を用いる。

ここで、Intel _Quad、QD、fast_dd、並びに exflib では複素数の基本演算機能が提供されていないため、C++複素数テンプレートクラスと組み合わせて用いた。

また、任意精度・多倍長精度演算ライブラリは利用マニュアルが十分でないことが多く、サンプルコードからプログラミング方法を学ぶ必要がある。今回作成したいくつかのプログラムをスライド 1～8 に示す。

3.4.3 測定環境

性能測定には、汎用 PC、名古屋大学の FX100、及び名古屋大学の CX400 を使用した。それぞれの環境における CPU、OS、コンパイラを表 2.9.2 に示す。また、それぞれの環境において性能測定を行うことができたライブラリを表 2.9.3 に示す。性能測定できなかった理由は次の通りである。

- libquadmath は、FX100(fcc)は富士通コンパイラのため、FX100(gcc)は gcc-6.3.0 だが当該ライブラリ未提供のため、CX400 は gcc-4.4.7 のため
- Intel's _Quad は、PC ではライセンス未導入のため、FX100 は SPARC64 プロセッサのため
- fast_dd は、FX100 のみ利用可能であるため
- exflib は x64 バイナリ版のみを利用したため

表 3.4.2　実験に用いた計算機とソフトウェア

Name	CPU	OS	Compiler	Site
PC	Intel Core i7-8650U (Kabylake-R)	Ubuntu 18.04 TLS	gcc-7.3.0	-
FX100(fcc)	Fujitsu SPARC64 XIfx	XTC OS 2.1.1	fcc-2.0.0	Nagoya
FX100(gcc)			gcc-6.3.0	Nagoya
CX400	Intel Xeon E5-2697v3 (Haswell-EP)	RHEL	icc-14.0.3	Nagoya

表 3.4.3　使用ライブラリと計算環境における利用可否

Library	Version	PC	FX100(fcc)	FX100(gcc)	CX400
libquadmath	(gcc-4.6.0 or later)	○	-	-	-
Intel's _Quad	icc-14.0.3	-	-	-	○
QD	2.3.20	○	○	○	○
fast_dd	2.3.2	-	○	○	-
ARPREC	2.2.19	○	○	○	○
exflib	x64-bin-20180620	○	-	-	○
GMP MPFR MPC	4.0.1 6.1.2 1.1.0	○	○	○	○

3.4.4 計算精度に関する数値実験

a) 問題設定

実数を係数にもつ 2 次方程式$ax^2+bx+c=0$を考える。この式は解の公式により、$x_1=\left(-b-\sqrt{b^2-4ac}\right)/(2a)$と$x_2=\left(-b+\sqrt{b^2-4ac}\right)/(2a)$で得られる。このとき、桁落ちしやすい係数$a=1.01$、$b=2718281$、$c=0.01$を用いて$x_2$を計算したときの精度を評価する。この問題では加減算、乗算、除算、平方根の計算を行う。

真の解は$x_2=-3.67879553291216532392049 99\times10^{-9}$である。比較として、exflib ライブラリで 100 桁精度で計算した結果は$\tilde{x_2}=-3.67879553291216532392049 92803347\times10^{-9}$であった。

b) 実験結果

スライド 9 に実験結果を示す。任意精度演算ライブラリについては、4 倍精度相当となるように計算精度を設定している。表の結果に書かれている数値は、太字・下線のところまで正しく計算できていることを表している。表より、普

通に倍精度数で計算したのでは小数点以下 1 桁までしか正しく計算できていないことが分かる。これに対し、疑似四倍精度の QD と fast_dd は 16～17 桁まで正しく計算できており、Fortran や C の libquadmath といった四倍精度の結果(18～19 桁)に近い計算精度が得られている。これより、名大 FX100 環境などにおいて、多倍長精度計算ライブラリを正しく利用できていることが示された。

ここで、x86 系である PC の環境では long double 型は小数点以下 3 桁程度の精度で拡張倍精度として倍精度より高い精度で計算しているようであるがこの問題にとっては精度不足である。一方、FX100 の long double 型は四倍精度相当の結果が得られている。long double 型変数が占めるサイズを調べたところ 128 ビットであり、きちんと 128 ビット分の精度が出ていることになる。富士通の C 言語処理系でとりあえず四倍精度数を使いたい場合に有用である。

また、Intel's _Quad は倍精度と同じ程度の結果となった。これは、平方根を含め数学関数の四倍精度版が提供されていなかったためである。実験当時の Intel コンパイラの説明として_Quad は実験的なものと述べられていた。

なお、任意精度演算ライブラリの ARPREC と exflib は想定以上の精度が出ているが理由は不明である。

3.4.5 計算時間に関する数値実験

a) 問題設定

以下のロジスティック写像を考える。

$$x_{n+1} = a x_n (1 - x_n)$$

$a = 4$とすると、x_nは区間[0,1]全体で非周期に変動するカオスとなることが知られている。このとき、初期値$x_0 = 0.7501$とし、nが10^6になるまで計算したときの計算時間を評価する。また、x_nを実数としたとき、複素数としたときの 2 ケースを評価する。この問題では、乗算と減算のみを行う。

b) 実験結果

スライド 10 と 11 に実数と複素数のときの実験結果をそれぞれ示す。表において、QD(C++)と QD(C)はそれぞれプログラム記述言語が C++と C であることを表している。QD ライブラリ自体は C++で記述されており、C++プログラムから利用する場合向けに四則演算等の演算子オーバーロードをしている一方で、C プログラム向けに関数コールでの利用も可能となっている。C++であればプログラムは書きやすいが、計算時間の観点で最適なコードが生成されない可能性がある。そこで比較のために 2 つの実装を行った。

スライド 10 より、PC では QD(C++)が最速であった。これは C++の最適化が十分に行えているのだと推測される。また、QD(C++)による疑似四倍精度演算は倍精度演算に比べて 3 倍遅い程度であった。また、疑似四倍精度演算は四倍精度演算よりも 3 倍程度速く、前の実験で計算精度が同程度であったことから、実用性が高いと言える。FX100(fcc)では、同じく QD(C++)が最速であったが、倍精

度演算よりも 7 倍程度の計算時間であり、C++コードのさらなる最適化が望まれる。CX400 では Intel's _Quad が最速であり、QD(C++)もほぼ同程度であった。

次に、ターゲットとする複素数演算の性能を、スライド 11 で見てみる。実数のときと異なり、全ての環境において QD(C)が最速となった。よって、QD(C)の利用が良いように思えるが、プログラム例のスライド 4 を見てわかるように、QD ライブラリを C 言語から複素数で利用する場合は複素数の四則演算などの関数を自作する必要があり、さらにプログラミングコストが高くなり、可搬性も損なわれる。C++の演算子オーバーロードを利用すれば、疑似四倍精度数であっても、複素数であっても、z=x+y のように記述できる。しかし、実際には、実部と虚部それぞれの加算に展開され、加算は疑似四倍精度数の加算になり、その加算 1 回あたりでは Knuth による加算の無誤差変換に基づいて倍精度数の加減算が 11 回行われる。つまり、z=x+y という記述は 22 回の演算に展開される。z=x*y は 100 回以上、z=x/y は 250 回以上の演算にもなる。実際のプログラムではもっと長い計算式が書かれ、for ループブロック内であればループボディが非常に長くなることを意味する。スライド 14 と 15 に、QD(C++)を用いた疑似四倍精度の密行列ベクトル積コードを FX100 の富士通 C/C++コンパイラでコンパイルしたときの最適化メッセージを示す。これより、SIMD 化もソフトウェアパイプライン化も行われていないことが分かる。コンパイラによる最適化が望まれる。なお、インライン展開に関するメッセージが大量にでてくるため、メッセージを調査しづらい問題があった。

また、名大 FX100 において富士通 C/C++コンパイラの 32 レジスタ制限版を用いた実験も行った。その結果をスライド 12 と 13 に示す。このケースでは通常環境と性能差はなかった。しかし、上述したように現状でもコンパイラにはもう少し最適化を頑張ってほしいため、32 レジスタ制限環境の影響を受けないかどうかは判断できないと言える。

3.4.6 複素対称線形方程式の数値実験

複素対称線形方程式における多倍長精度計算の有効性を評価する数値実験を PC 環境で行った。疎行列データベース UF Sparse Matrix Collection で公開されている 2 つの小規模行列を用いて、IC 前処理付き COCG 法で解いた実験結果をスライド 16～18 に示す。これより、疑似四倍精度数を用いると倍精度数の場合と比べて反復回数を削減できることが分かる。このケースでは総計算時間は倍精度の場合が高速であったが、反復回数の削減効果は問題依存のため、その効果が大きい場合であれば、倍精度数よりも疑似四倍精度数を使う方が高速になる可能性があると言える。

スライド 19 と 20 に、高周波電磁場の有限要素解析で得られた複素対称行列による数値実験結果を示す。このケースでは、わずかにであるが疑似四倍精度が高速であった。

3.4.7 富士通 fast_dd について

富士通 TCS に含まれる富士通 fast_dd は QD と同様に double-double 形式の疑似四倍精度演算機能を提供するライブラリであるが、スライド 10～13 に示すように計算時間の面では QD ライブラリよりも低い性能となっている。しかし、富士通 fast_dd は 2 要素ずつ同時に計算するマルチ関数やベクトル関数を提供しており、それらが利用できるケースでの性能改善が目的の 1 つにある。前述の計算は単純な漸化式であったことからマルチ関数等の利用は行っていない。そこで、第 9 回 WG において富士通側報告として行われたマルチ関数・ベクトル関数の性能評価例をスライド 21～23 に示す。これより、複数要素の同時計算が行える場合では fast_dd は QD よりも高い性能が期待できることが明らかになった。

3.4.8 HOWTO

今回の数値実験を通して得られた多倍長精度演算ライブラリの利用に関する知識について、HOWTO としてまとめ、知識共有を行う。得られた知識は以下の通りである。

① 任意精度・多倍長精度演算ライブラリには、計算順序の入れ替えを行うと計算精度が低下するものがある。よって、例えば富士通 C/C++コンパイラでは"-Keval"オプションは指定すべきではない。"-Keval"は"-Kfast"に含まれているので"-Kfast"の利用も避けるべきである。しかし、環境によっては、コンパイラの標準オプションとして"-Kfast"が指定されている場合があるので注意が必要であり、"-Knoeval"を使うのが望ましい。

② 富士通 C/C++処理系で任意精度・多倍長精度演算ライブラリを利用するときは、コンパイラのインライン展開オプション"-x-"を指定した方が高速となる。

③ 富士通 C/C++コンパイラで QD ライブラリをビルドするときは、コンパイラに"-Kfp_contract"オプションを指定すれば FMA 命令をきちんと使ってくれる。①、②とあわせて、"-O3 -Kfp_contract -Knoeval -x-"が基本となる。

④ 名大 FX100 の gcc-6.3.0 環境では FMA 命令を使うよう指定して QD ライブラリをビルドすると、計算結果が誤ったものになる。SPARC64 XIfx の正しい FMA 命令が出されないのだと思われる。また、libquadmath はビルドされていない。

⑤ 富士通 fast_dd は SSL II の一部として提供されているが、GCC など富士通 TCS 以外のコンパイラで SSL II ライブラリを使うときは"-SSL2"ではリンク

できない。例えば今回指定したのは、インクルードパス指定は"-I/opt/FJSVmxlang/include"、リンク指定は"-L/opt/FJSVmxlang/lib64 -lssl2mtfoursimd_prexi -lssl2mtfoursimd_com -lssl2mtfoursimd_postxi -lfj90i -lfj90fmt
-lfj90f -lfj90rt -lfjrtcl -ltrt -L/usr/lib64 -lm -lelf"である。Intel MKL Link Line Advisor のようなリンク方法を調べる手助けがあるのが望ましい。

⑥ 富士通 C/C++処理系では long double 型は 128 ビットの 4 倍精度浮動小数点数である。

⑦ 任意精度・多倍長精度の浮動小数点数に定数を代入するときは、定数の精度に気を付ける必要がある。例えば QD ライブラリでは文字列定数を使えば、疑似四倍精度数に変換した上で代入してくれる。

⑧ 富士通 fast_dd は疑似四倍精度定数の指定方法が不明だが、富士通 C/C++処理系のときは 4 倍精度である long double 型定数で代用できる。

⑨ 任意精度・多倍長精度演算ライブラリを使うときは、出力関数が対応しているかも気にする必要がある。

3.4.9 まとめ

今回は将来的に ADVENTURE へ多倍長精度演算を組み込むことを目的に、任意精度・多倍長精度演算ライブラリを名大 FX100 環境に移植し、数値実験によって性能評価を行った。当初は、想定する計算結果が得られないことが多く、丸め回数が少ない FMA 命令を使えているか、コンパイラによる計算順序入れ替えは起こっていないか、高精度な定数はどう指定するか、高精度な数学関数はあるのか、あたりを確認していけば正しく移植できることが見えてきた。疑似四倍精度数かつ複素数となると 1 回の四則演算あたりの計算量が多く、それを含む DO ループ／for ループの最適化は難しいと思われるが、ポスト京のコンパイラが効率良いコードを出してくれることに期待する。また、多倍長精度計算の専門家との協調作業が必要と思われるが、そのためには大量に出力されるコンパイラの最適化メッセージの改善も望まれる。

複素数版プログラム例 (FORTRAN)

```
program quadformula_complex_real16
  implicit none
  complex(kind=16) :: a, b, c, x2
  a = 1.01Q0
  b = 2718281.0Q0
  c = 0.01Q0
  x2 = (-b+sqrt(b*b-4.0*a*c))/(2.0*a)
  print *, real(x2)
end program quadformula_complex_real16
```

4倍精度複素数の宣言

4倍精度定数の代入

複素数版プログラム例 (libquadmath, C)

```
#include <stdio.h>
#include <stdlib.h>
#include <quadmath.h>

int main(void)
{
  __complex128 a, b, c, x2;
  char str[1024];

  a = 1.01Q;
  b = 2718281.0Q;
  c = 0.01Q;
  x2 = (-b+csqrtq(b*b-4.0*a*c))/(2.0*a);
  quadmath_snprintf(str, 1024, "%31.30Qe , %31.30Qe",
__real__ x2, __imag__ x2);
  printf("x2= %s¥n", str);
}
```

4倍精度複素数の宣言

4倍精度定数の代入

4倍精度数学関数の呼び出し

4倍精度専用の出力関数

複素数版プログラム例 (QD, C++)

```
#include <iostream>
#include <iomanip>
#include <complex>
#include <stdlib.h>
#include <qd/dd_real.h>
using namespace std;

int main(void)
{
  complex<dd_real> x2;
  unsigned int oldcw;
  fpu_fix_start(&oldcw);

  complex<dd_real> a("1.01", "0.0");
  complex<dd_real> b("2718281.0", "0.0");
  complex<dd_real> c("0.01", "0.0");

  x2 = (-b+sqrt(b*b-
complex<dd_real>(4.0,0.0)*a*c))/(complex<dd_real>(2.0,0.0)*a);
  cout << "x2= " << scientific << setprecision(31) << x2 <<
"¥n";
}
```

疑似4倍精度変数の宣言

80ビット拡張倍精度を使うIntel x86向け設定

4倍精度定数の代入

複素数版プログラム例 (QD, C)

```
...
#include <qd/dd_real.h>
#include <qd/c_dd.h>
#define TO_DOUBLE_PTR(a, ptr) ptr[0] = a.x[0]; ptr[1] = a.x[1];

/* (a_r + b_r) + (a_i + b_i) I */
void c_zdd_add(const double *a_r, const double *a_i, const double *b_r,
const double *b_i, double *c_r, double *c_i) {
  dd_real cr, ci;
  cr = dd_real(a_r) + dd_real(b_r);
  ci = dd_real(a_i) + dd_real(b_i);
  TO_DOUBLE_PTR(cr, c_r);
  TO_DOUBLE_PTR(ci, c_i);
}
...
int main(void){
  double a_r[2], a_i[2], b_r[2], b_i[2], c_r[2], c_i[2], x2_r[2], x2_i[2];
...
  c_dd_copy_d(1.01, a_r);
...
  c_zdd_mul(b_r, b_i, b_r, b_i, t1_r, t1_i); /* b*b */
  c_zdd_mul(a_r, a_i, c_r, c_i, t2_r, t2_i); /* a*c */
...
  c_dd_write(x2_r);
```

マクロ関数の定義

倍々精度複素数の
四則演算関数は自作

変数1個につき，実部と虚部それぞれ大きさ2個の倍精度配列

四則演算は関数呼び出し

複素数版プログラム例 (Fujitsu fastdd, C++)

```
#include <iostream>
#include <iomanip>
#include <stdlib.h>
#include <complex>
#include <fast_dd.h>

using namespace std;
int main(void)
{
  complex<dd_real> a, b, c, x1, x2;
  long double lx;

  a = 1.01L;
  b = 2718281.0L;
  c = 0.01L;

  x2 = (-b+sqrt(b*b-4.0*a*c))/(2.0*a);
  lx = to_long_double(real(x2));
  cout << "x2= " << scientific << setprecision(31) << lx <<
"¥n";
}
```

疑似4倍精度複素数の宣言

4倍精度定数の作り方が用意されていないので，long double定数を代入

4倍精度の出力機能は用意されていないのでlong doubleとかに変換してから出力する必要あり

複素数版プログラム例 (ARPREC, C++)

```
#include <iostream>
#include <iomanip>
#include <stdlib.h>
#include <arprec/mp_complex.h>

using namespace std;
int nr_digits = 32;

int main(void)
{
  mp::mp_init(nr_digits);
  mp_complex x2;
  mp_complex a("1.01E+0","0E+0");
  mp_complex b("2718281.0E+0","0E+0");
  mp_complex c("0.01E+0","0E+0");

  x2 = (-b+sqrt(b*b-4*a*c))/(2*a);
  cout << "x2= " << scientific << setprecision(31) << x2.real << "
" << x2.imag << "¥n";

  mp::mp_finalize();
}
```

精度の設定

任意精度変数の宣言

任意精度定数の代入

複素数版プログラム例 (exflib, C++)

精度の設定

```
#define PRECISION 32
#include <iostream>
#include <iomanip>
#include <complex>
#include <stdlib.h>
#include <exfloat.h>

using namespace std;
int main(void)
{
  complex<exfloat> a, b, c, x2;

  a = "1.01";
  b = "2718281.0";
  c = "0.01";

  x2 = (-b+sqrt(b*b-
complex<exfloat>(4,0)*a*c))/(complex<exfloat>(2,0)*a);
  cout << "x2= " << scientific << setprecision(31) << x2 <<
"¥n";
}
```

任意精度複素数の宣言

任意精度定数の代入

複素数版プログラム例 (MPC, C++)

```
#include <iostream>
#include <iomanip>
#include <stdlib.h>
#include <gmp.h>
#include <mpfr.h>
#include <mpc.h>

#define PRECISION 106
using namespace std;
int main(void)
{
  mpc_t a, b, c, _four, _two, x2, t1, t2, t3;

  mpc_init2(a,PRECISION);
  mpc_init2(b,PRECISION);
  mpc_init2(c,PRECISION);
  mpc_init2(x2,PRECISION);
  mpc_init2(t1,PRECISION);
  mpc_init2(t2,PRECISION);
  mpc_init2(t3,PRECISION);
  mpc_init2(_four,PRECISION);
  mpc_init2(_two,PRECISION);
```

精度の設定

任意精度変数の宣言

変数の初期化や任意精度定数の代入

```
  mpc_set_str(a, "(1.01 0.0)", 10, MPC_RNDNN);
  mpc_set_str(b, "(2718281.0 0.0)", 10, MPC_RNDNN);
  mpc_set_str(c, "(0.01 0.0)", 10, MPC_RNDNN);
  mpc_set_str(_four, "(4.0 0.0)", 10, MPC_RNDNN);
  mpc_set_str(_two, "(2.0 0.0)", 10, MPC_RNDNN);

  mpc_mul(t1, b, b, MPC_RNDNN);
  mpc_mul(t2, a, c, MPC_RNDNN);
  mpc_mul(t3, t2, _four, MPC_RNDNN);
  mpc_sub(t2, t1, t3, MPC_RNDNN);
  mpc_sqrt(t1, t2, MPC_RNDNN);
  mpc_sub(t2, t1, b, MPC_RNDNN);
  mpc_mul(t3, a, _two, MPC_RNDNN);
  mpc_div(x2, t2, t3, MPC_RNDNN);

  cout << "x2= ";
  mpc_out_str(stdout, 10, 0, x2, MPC_RNDNN);
  cout << "¥n";

  mpc_clear(a);
  mpc_clear(b);
  mpc_clear(c);
  mpc_clear(x2);
  mpc_clear(t1);
  mpc_clear(t2);
  mpc_clear(t3);
  mpc_clear(_four);
  mpc_clear(_two);
}
```

乗算

オブジェクトの解放

計算精度に関する数値実験結果

- 最適化オプション “-O3”, MPFRの丸めモードは最近接偶数丸め
- 任意精度ライブラリは4倍精度相当で計算

	言語	精度	結果
double	C	倍	-**3.6**8840623610090497961660940384 0e-09
long double	C	拡張倍	-**3.678**8385319362145543678892994 46e-09
long double@FX	C	128ビット	-**3.678795532912165323 9**60311671534e-09
real(kind=16)	F90	4倍	-**3.678795532912165323 9**60311671534e-09
libquadmath	C	4倍	-**3.678795532912165323**7603627688244e-09
_Quad	C++	4倍	-**3.6**8840623610090501233224886516 63e-09
QD	C++	疑似4倍	-**3.678795532912165 32**960567157986e-09
fast_dd	C++	疑似4倍	-**3.67879553291216 53**9458323614294 0e-09
ARPREC	C++	32桁	-**3.678795532912165323920499280 3347**e-09
exflib	C++	32桁	-**3.678795532912165323920499**457919e-09
MPFR	C	106ビット	-**3.67879553291216 53**0958523266365 9e-09

計算時間[s]に関する実験結果 (実数)

	言語	PC (gcc)	FX100 (fcc)	FX100 (gcc)	CX400 (icc)
double	C	0.0105	0.0116	0.00546	0.0060
long double	C	0.0110	0.7882	0.781	0.0095
real(kind=16)	F	0.1050	0.0000	0.781	0.0746
libquadmath	C	0.0921	N/A	N/A	N/A
_Quad	C++	N/A	N/A	N/A	**0.0714**
QD (C++)	C++	**0.0304**	**0.0732**	0.0333	0.0798
QD (C)	C	0.0409	0.2884	**0.0221**	0.1364
fast_dd	C++	N/A	0.1441	0.144	N/A
ARPREC	C++	0.8968	6.6298	7.493	0.6934
exflib	C++	0.0856	N/A	N/A	0.1060
MPFR	C	0.3329	1.3545	0.597	0.3939

赤字は四倍精度相当の計算結果になっているもので最速
青字は計算結果の傾向が異なるもの

計算時間[s]に関する実験結果 (複素数)

	言語	PC / gcc	FX100 / fcc	FX100 / gcc	CX400 / icc
double	C	0.0227	0.0234	0.0228	0.0087
long double	C	0.0457	1.4669	1.288	0.0155
complex(kind=16)	F	0.3250	0.0000	1.453	0.2133
libquadmath	C	0.2141	N/A	N/A	N/A
_Quad	C++	N/A	N/A	N/A	0.2004
QD (C++)	C++	0.1349	0.3232	0.342	0.1552
QD (C)	C	**0.1334**	**0.2862**	**0.241**	**0.1472**
fast_dd	C++	N/A	0.7464	0.803	N/A
ARPREC	C++	1.5565	13.6597	5.515	1.3135
exflib	C++	0.2591	N/A	N/A	0.2864
MPC	C	0.1928	0.5868	0.422	0.2342

赤字は四倍精度相当の計算結果になっているもので最速
青字は計算結果の傾向が異なるもの

計算時間[s]に関する実験結果2 (実数)

	言語	PC / gcc	FX100 / fcc	FX100 / fcc(32reg)	FX100 / gcc
double	C	0.00679	0.00548	0.00546	0.00546
long double	C	0.00645	0.788	0.785	0.781
real(kind=16)	F90	0.0730	0.313	0.324	0.781
libquadmath	C	0.0636	N/A	N/A	N/A
QD (C++)	C++	**0.0285**	0.0333	0.0322	**0.0692**
QD (C)	C	0.0323	**0.0221**	**0.217**	0.0851
fast_dd	C++	N/A	0.144	0.143	0.155
ARPREC	C++	0.408	7.493	7.247	2.90
exflib	C++	0.0727	N/A	N/A	N/A
MPFR	C	0.166	0.597	0.574	0.469

赤字は四倍精度相当の計算結果になっているもので最速
青字は計算結果の傾向が異なるもの

計算時間[s]に関する実験結果2 (複素数)

	言語	PC / gcc	FX100 / fcc	FX100 / fcc(32reg)	FX100 / gcc
double	C	0.00542	0.0205	0.0205	0.0228
long double	C	0.0248	1.455	1.452	1.288
complex(kind=16)	F90	0.186	0.787	0.584	1.453
libquadmath	C	0.138	N/A	N/A	N/A
QD (C++)	C++	0.0579	0.173	0.128	0.342
QD (C)	C	**0.0569**	**0.112**	**0.113**	**0.241**
fast_dd	C++	N/A	0.743	0.733	0.803
ARPREC	C++	0.704	13.549	13.801	5.515
exflib	C++	0.205	N/A	N/A	N/A
MPC	C	0.132	0.528	0.487	0.422

赤字は四倍精度相当の計算結果になっているもので最速
青字は計算結果の傾向が異なるもの

FX100における疑似四倍精度実行列ベクトル積演算(QD(C++))の最適化メッセージ例

```
38                for (i = 0; i < n; i++) {
39    i             q[i] = 0.0;
40              #pragma loop noalias
41          2       for (j = 0; j < n; j++) {
42    i     2         q[i] += A[i*n+j] * p[j];
43          2       }
44                }
...
 jwd6101s-i  "logisticmap_real_qd_quad.cpp", line 41: SIMD conversion is not
applied because a statement that prevents SIMD conversion exists.
  jwd8670o-i  "logisticmap_real_qd_quad.cpp", line 41: This loop is not
software pipelined because the loop contains a branch instruction which is
not for loop iteration.
  jwd8202o-i  "logisticmap_real_qd_quad.cpp", line 41: Loop unrolling
expanding 2 times is applied to this loop.
  jwd8101o-i  "logisticmap_real_qd_quad.cpp", line 42: Inline expansion is
applied to the user defined function '_ZmlRK7dd_realS1_'.
  jwd8101o-i  "logisticmap_real_qd_qu
applied to the user defined function
```

・SIMD化されず
・SWPLされず
・ループ展開2倍

インライン展開のメッセージが大量にでる

FX100における疑似四倍精度複素行列ベクトル積演算(QD(C++))の最適化メッセージ例

```
40                for (i = 0; i < n; i++) {
41    i             q[i] = c_zero;
42              #pragma loop noalias
43                for (j = 0; j < n; j++) {
44    i             q[i] += A[i*n+j] * p[j];
45                }
46              }
...
 jwd6101s-i  "logisticmap_complex_qd_quad.cpp", line 43: SIMD conversion is
not applied because a statement that prevents SIMD conversion exists.
  jwd8670o-i  "logisticmap_complex_qd_quad.cpp", line 43: This loop is not
software pipelined because the loop contains a branch instruction which is
not for loop iteration.
  jwd8101o-i  "/opt/FJSVmxlang/bin/../include/c++/std/stl/_string_io.c", line
44: Inline expansion is applied to the user defined function
'_ZNKSt8ios_base5flagsEv'.
  jwd8101o-i  "logisticmap_complex_qd_quad.cpp", line 44: Inline expansion is
applied to the user defined function '_ZStmlI7dd_realESt7complexIT_ERKS3_S5_'.
  jwd8101o-i  "logisticmap_complex_qd_quad.cpp", line 44: Inline expansion is
applied to the user defined function '_ZNSt7complexI7dd_realEpLERKS1_'.
```

- SIMD化されず
- SWPLされず
- ループ展開なし

Numerical experiments on the convergence of the iterative method

- Solving complex symmetric linear systems

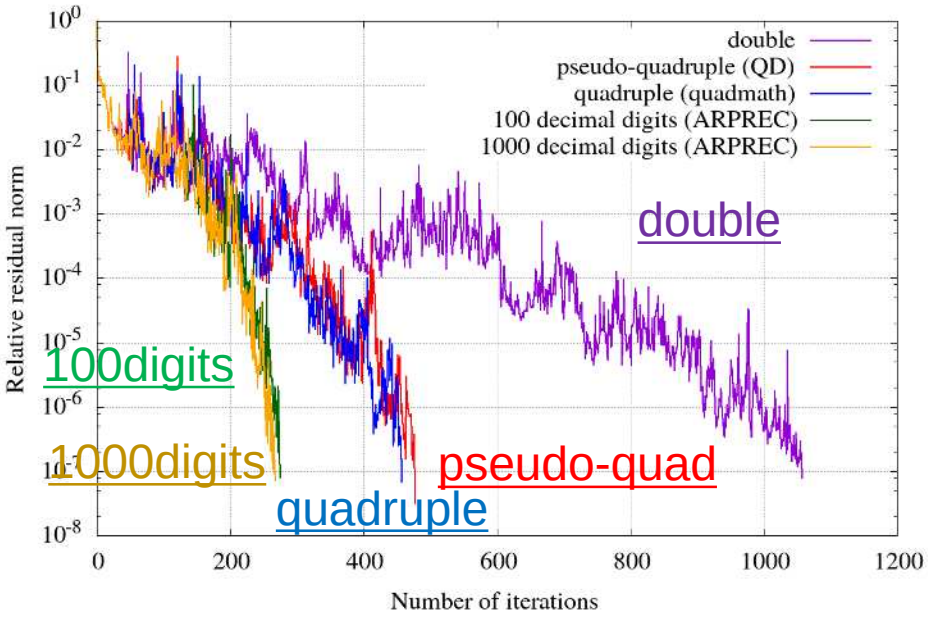
$$A\mathbf{x} = \mathbf{f}$$

  - coefficient matrices (from the SuiteSparse Matrix Collection)
    - **dwg961b**: n= 961, num. of nonzero= 19,591
    - **qc2534**: n= 2,534, num. of nonzero= 463,360

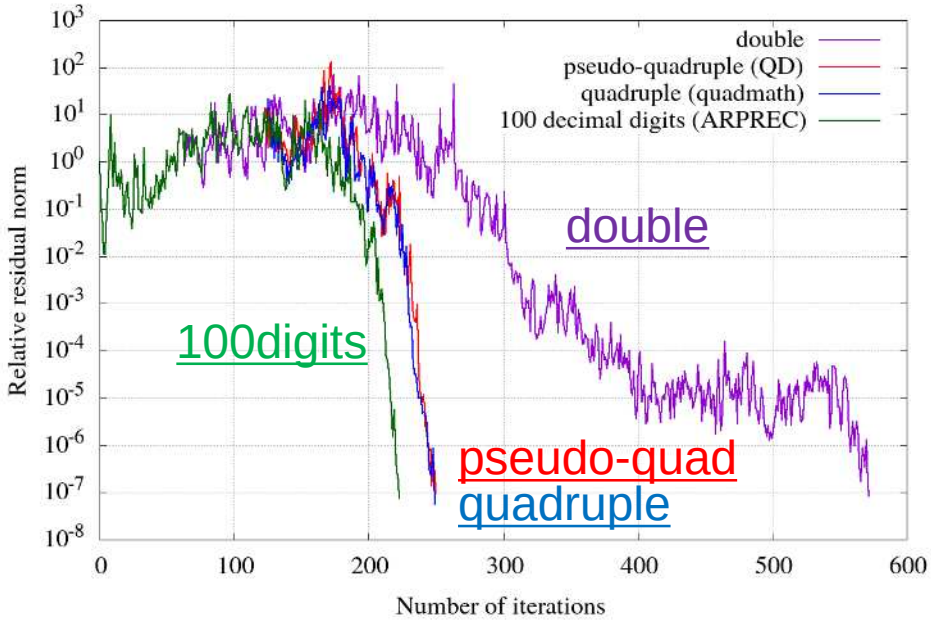
    an electromagnetic field problem and a complex symmetric matrix
  - $\mathbf{f} = A \times (1 + i, 1 + i, \ldots, 1 + i)^T$
- Iterative methods
  - COCG (Conjugate Orthogonal Conjugate Gradient) method w/ incomplete Cholesky preconditioning
- Multiple-precision libraries
  - libquadmath (GCC 5.4.0), QD 2.3.17, ARPREC 2.2.18
- Computers
  - Intel Core i7-6500U / Ubuntu 16.04 TLS / gcc-5.4.0

Comparison of number of iterations with different precision arithmetic

Convergence criterion: 10^{-7}



Convergence history for dwg961b



Convergence history for qc2534

- “100digits” means 100 decimal digits of precision calculated by ARPREC as reference.
- In solving complex symmetric linear systems,
  - multiple-precision got faster convergence compared with double
  - (pseudo-)quadruple is enough precision

Comparison of calculation time with different precision arithmetic

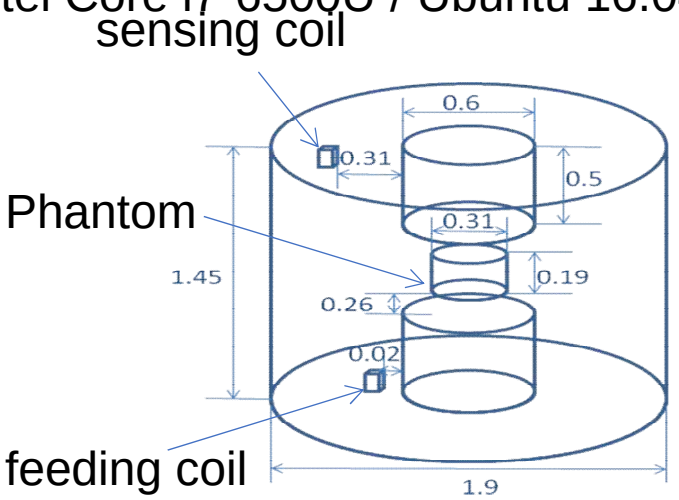
Convergence criterion: 10^{-7}

Matrix	Library	Precision name	Decimal digits	# of Iter.	Time [s]
dwg961b	non-use	double	15	1,058	0.480
	QD	pseudo-quadruple	32	479	**1.216**
	quadmath	quadruple	34	457	4.132
	ARPREC	pseudo-quadruple	32	383	32.503
qc2534	non-use	double	15	572	9.437
	QD	pseudo-quadruple	32	248	**20.841**
	quadmath	quadruple	34	249	73.272
	ARPREC	pseudo-quadruple	32	240	748.447

- QD is faster than GCC's libquadmath and ARPREC in total.
- QD takes at most 3 times longer than double precision in small-scale cases.

TEAM Workshop Problem 29

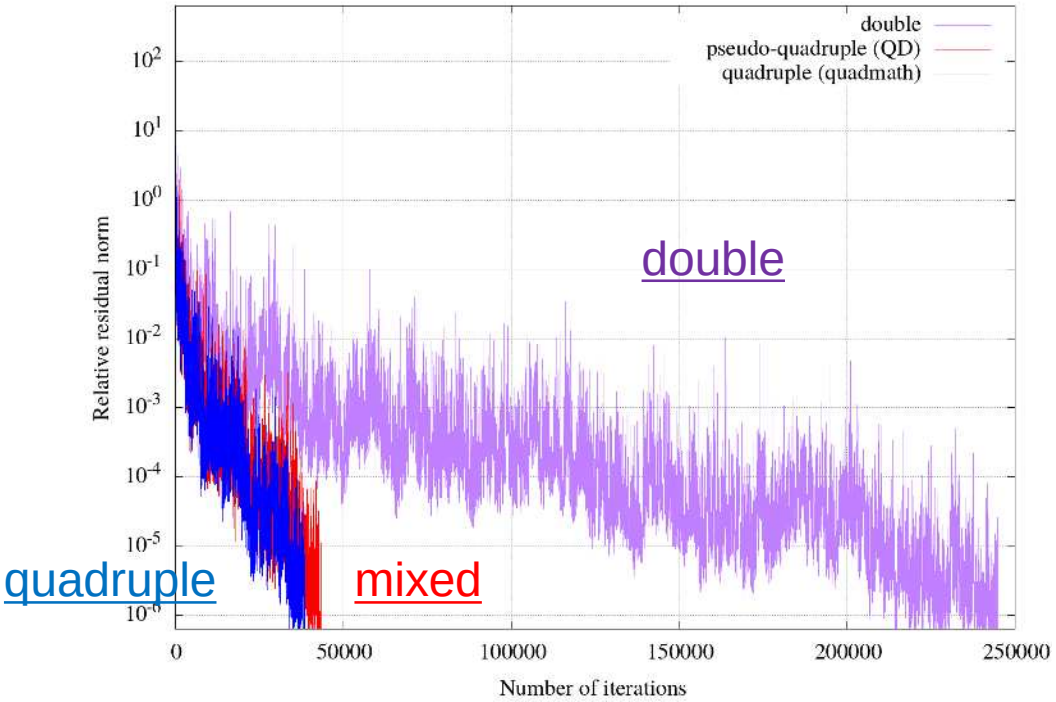
- Benchmark model for hyperthermia simulation
  - Problem size: 134,573
  - Frequency: 300 (MHz)
  - Relative permittivity of Phantom: 80.0
  - Electrical conductivity of Phantom: 0.52 (S/m)
- Linear solver
  - Iterative method: COCG w/ Symmetric SOR ($\omega = 1.05$) preconditioning
  - Convergence tolerance: relative residual norm < 10^{-7}
- Computers
  - Intel Core i7-6500U / Ubuntu 16.04 TLS / gcc-5.4.0



Model

Finite element mesh

Computational performance of 300 (MHz)



Precision	# of iter.	Time [hr.]
double	24,5105	5.78
pseudo-quadruple (QD)	43,598	**5.69**
quadruple (libquadmath)	38,643	14.09

4倍精度演算

FUJITSU

- IEEE 754-2008のbinary128形式
  - ソフトでエミュレーションするため低速

- double-double形式
  - 倍精度を2つ並べた形式
    - 倍精度演算器を使って計算できるためIEEE形式よりも高速
  - 加減乗算の演算方法は広く知られている
  - doube-double形式の実装例
    - フリーソフト：QDなど
    - 富士通：高速4倍精度基本演算ライブラリ(以降fast_dd)
    - 他ベンダー
      - IBMでは4倍精度としてdouble-double形式をサポートするシステム有り
      - NEC：ASLQUAD
    - この他に、ユーザ自身が実装する場合もあり

QD と fast_dd

FUJITSU

- QDの実装
  - C++版はヘッダファイル内に演算が書かれており、インライン展開される
    - 翻訳時に最適化が適用されることで高速化を実現
  - Fortran版は、wrapperのみでCの関数を呼び出している
  - 注意点
    - C++版は、インライン展開されないと高速化されず、また、演算順序を変更する最適化が適用されると結果が正しくなくなる
    - Fortranは1要素ずつ関数を呼び出すため高速化されない
- fast_ddの実装
  - ユーザの指定する最適化オプションによって性能や結果に影響が出ることを避けるため、インライン展開方式は採用しない
    - 演算は基本的に1要素ずつ関数を呼び出すことで実現
  - 高速化のために、マルチ関数（2要素ずつ計算）・ベクトル関数（入出力が1次元配列）を用意
  - ➔マルチ関数・ベクトル関数を使用して頂くことを期待
    - 7/20ご報告の評価で使われたロジスティック写像は漸化式でベクトル関数が使えないためQDのC++版（インライン）が高速

実行性能

FUJITSU

加算の性能

FX100 1.5Ghz
N=1024 (L1に乗るサイズ)

```
use fast_dd
INTEGER::N
TYPE(dd_real)::X(N),Y(N),Z(N)
DO I=1,N
  Z(I)=X(I)+Y(I)
END DO
```

※ マルチ関数、ベクトル関数は関数呼出し化した版を使用

		性能 mflops(注1)	備考
QD	C++	68.2	インライン、SWPLのみ
	Fortran	21.9	1要素ずつ関数呼出し
fast_dd 単体関数	C++	13.1	1要素ずつ関数呼出し
	Fortran	22.2	1要素ずつ関数呼出し
fast_dd マルチ関数	C++	43.0	2要素ずつ関数呼出し
	Fortran	114.0	2要素ずつ関数呼出し
fast_ddベクトル関数	C++	805.0	SIMD+SWPL
	Fortran	874.0	SIMD+SWPL

注1) double-double形式の加算1回を1flopとしたときの性能

4.1　核融合プラズマ乱流コード GKV の Post-FX100 性能推定

核融合科学研究所　沼波 政倫
名古屋大学情報基盤センター 片桐 孝洋
富士通株式会社　青木 正樹

4.1.1 はじめに

核融合プラズマ乱流コードである GKV（スライド p.3）は、主に磁場閉じ込め型の核融合装置における炉心プラズマの乱流輸送現象を扱うように設計・開発された。磁場閉じ込め装置は、強い磁場によってトーラス状の高温プラズマを閉じ込める。この強い磁場に起因して測度空間を 2 次元に近似した 5 次元位相空間上（3 次元実空間+2 次元速度空間）のプラズマ分布関数の発展を記述するボルツマン方程式（ジャイロ運動論方程式）が、この磁場閉じ込めプラズマの第一原理モデルとなる。GKV コードでは、ジャイロ運動論方程式を 5 次元格子上で離散化して解き進める。特に、乱流による揺らぎの磁力線垂直方向への波長が平衡分布のスケールよりも十分に短いことを仮定し、一本の磁力線に沿った局所フラックス・チューブ上でシミュレーションを行う。5 次元位相空間は、空間離散化として磁力線垂直方向の 2 次元空間に対してはフーリエ・スペクトル法、磁力線平行方向の実空間、および 2 次元の速度空間方向については、4 次または 5 次精度差分法を用いており、時間積分には 4 次精度陽的 Runge-Kutta-Gill 法を採用している。2 次元の速度空間に加えて、電磁揺動や極微細電子スケール乱流を扱うことができるようにするため、磁力線垂直方向の 2 次元実空間は非常に高い解像度を必要とし、大型ヘリカル装置（LHD）などの複雑な磁場形状を有する実験装置での解析も対象とするために、磁力線平行方向にも高解像度を要する。さらには、プラズマを構成する粒子の種類の分だけ、分布関数の種類が増加するため、最終的に総格子点数として 100 億点以上を必要とする極めて大きな自由度の流体計算となる。「京」による全系計算でようやく、電子と水素イオンのスケールの乱流を同時に扱うことが初めてできるようになったが、さらに質量や電荷の大きなイオンを含んだ現実的なプラズマを扱うためには、計算コストがさらに数十倍～数百倍に増大するため、ポスト京のような次世代エクサスケール計算機が必要である。京以降で現行の計算機、例えば核融合科学研究所のプラズマシミュレータ（FX100：SPARC64XIfx）から、ポスト京、あるいは、Post-FX100（A64fx）への移行を考えた場合、大きな技術的ギャップが存在する。例えば、SPARC から ARM へのアーキテクチャの変更や、レジスタ数が 128 から 32 へ削減、SIMD 幅の 256bit から 512bit への増加、プロセッサ（CMG）あたり演算コア数の 32（16）から 48（12）への変更等である。このうち、本性能推定ではユーザプログラムの実装方法への影響が大きいと思われる、レジスタ数の削減に関して、FX100 上で用意されたポスト FX100 向けチューニング

用コンパイラと Post-FX100 性能推定ツールを用いて、Post-FX100 に対する GKV コードの性能推定を実施した。

4.1.2 プログラム概要

GKV は、プラズマ分布関数の熱平衡部分と摂動部分に分けた場合の摂動部分の時間発展を追う Delta-f モデルに基づき、磁場閉じ込め炉心プラズマに対して、一本の磁力線に沿った局所領域（フラックス・チューブ）注目し、この局所領域を計算対象としてプラズマの乱流およびプラズマ分布関数の時間発展を追跡する。大域的な手法に基づく乱流コードと違い、平衡分布の発展は記述することができないが、電磁揺動や極微細電子スケール乱流を定量的に扱うことができるため、これまでに、大型ヘリカル装置（LHD）や JT-60U など、実際の大型プラズマ実験装置での放電プラズマを対象にした実験結果の再現や予測解析が行われてきた。前述の通り、空間離散化としては、磁力線垂直方向の 2 次元空間に対してフーリエ・スペクトル法、磁力線平行方向の 1 次元実空間、および 2 次元の速度空間方向については、4 次または 5 次精度差分法を採用し、時間積分には 4 次精度陽的 Runge-Kutta-Gill 法を採用している。GKV では、上述の 5 次元位相空間に加えて、プラズマ粒子種を区別するために 6 次元配列を用意し、MPI を利用した 5 次元領域分割を行う。この際、並列 2 次元 FFT のための転置通信、差分演算のための 1 対 1 通信、速度空間・粒子種積分のための総和通信といった MPI 通信を利用している。これまでに、「京」に対する最適化として、Tofu ネットワークにおけるプロセス配置最適化、OpenMP を利用した通信演算オーバーラップの実装等の最適化処理が施されており、種々の効率化の結果、「京」全系規模に渡る約 60 万コアの強スケーリング（演算性能 786.4TFlops、理論ピーク比 8.29%、実行並列化率 99.99994%）を達成し、前述のように「京」の全系計算で電子スケール乱流とイオンスケール乱流の同時計算が実現した。また、FX100 におけるアシスタント・コアを用いて、通信と演算のオーバーラップ最適化も行っている。1CMG あたり 16 演算コア + 1 アシスタント・コアを有する FX100 の特性を反映して、アシスタント・コアで通信処理を行った場合、演算コアのマスタースレッドが通信に参加する場合と比較して、6.25%（=1/16）近くの高速化を達成している（スライド p.4～5 参照）。

4.1.3　ポスト FX100 向けチューニング用コンパイラによるコスト分析

GKV コードの性能推定に向けて、GKV カーネルコードを対象に、現行の FX100 の１ノードを用いた単体コスト分析を行った。コンパイラはポスト FX100 向けチューニング用コンパイラを用い、レジスタ数 128（以降、128reg と記載）でのコンパイルでのオプションは、「-Kfast -Kparallel -Kocl -Cpp -X9」であり、レジスタ数 32（以降、32reg と記載）でのコンパイル・オプションは 128reg と同様にソフトウェア・パイプラインを動作させるため、上に加えて「-Kno

unroll」を追加指定した。スライド p.7 に、測定条件とコスト分布の例を示す。ここでは、１ノード１プロセス（MPI 並列なし）を前提として、取り扱う問題サイズを、通常の１MPI プロセスあたりに相当する（nx,ny,nz,nv,nm）=（85,64,24,8,48）とした（粒子種数は 1）。128reg でのコンパイラを用いて測定した GKV コードのコスト分布を、スライド p.7 下部に示す。このコスト分析から、「literm_k」および「exb_realspcal」のサブルーチンがそれぞれ全体の約 53.7%、約 13.6%を占めているため、以降のコスト評価は、この 2 つにおけるループを対象にする。

前者のサブルーチン「literm_k」では、4 次精度陽的 Runge-Kutta-Gill 法による時間積分を行う際の方程式の右辺における線形項計算の一部に相当するものであり、（nx,ny,nz,nv,nm）に関する 5 次元ループから構成されている（スライド p.9 参照）。128reg コンパイラと 32reg コンパイラでの実行時間の比較（スライド p.10）では、128reg から 32reg へのコンパイラの変更による性能の劣化はほとんど見られていない。また、詳細 PA 情報を用いた比較（スライド p.11）では、128reg コンパイラ、32reg コンパイラともに、メモリビジー率が高く（96%）、メモリバンド幅ネックのループであることが分かる。一方、L1D キャッシュのミス率は、理論値（3.125%）付近であった。

後者のサブルーチン「exb_realspcal」では、時間積分を行う際の方程式の右辺における非線形項計算の一部に相当しており、（nx,ny,nv,nm）に関する 4 次元ループ計算から構成されている（スライド p.13 参照）。128reg コンパイラでの実行時間と比較して、32reg コンパイラの場合は、4%程度の短縮となっており（スライド p.14）、「literm_k」の場合と同様に、32reg コンパイラによる性能劣化はほぼ見られていない。詳細 PA 情報による比較（スライド p.15）においても、128reg および 32reg コンパイラともにメモリビジー率が高く、メモリバンド幅ネックのループであることが分かる。一方、L1D ミス率がそれぞれ 5.12%、5.24%となっており、理論値（3.15%）よりも高いため、スラッシングが発生していると考えられる。この対応として、5 節で述べるループ分割によるチューニングを試みた。

4.1.4　Post-FX100 における性能の推定

GKV カーネルコードの Post-FX100 における性能を推定するにあたり、前節のコスト分析で対象にした 2 つのサブルーチンについて詳細な評価を行う。性能予測では、スライド p.17 に示すように、FX100 において 32reg コンパイラを利用し、詳細プロファイラの実行でＰＡ情報を取得する。そこから、性能予測ツールを用いて、ハード、命令セットおよびコンパイラによる性能向上効果を算出し、性能予測値を出力する。

もっとも高いコストのループであった「literm_k」についての性能推定結果をスライド p.19 に示す。FX100 での詳細 PA 実行時間は前述のとおり、128reg、3

2reg ともにほとんど変化はなく、メモリバンド幅ネックのループであった。一方、性能推定ツールによる Post-FX100 の推定値（スレッド数は 16 から 12 に変更）は、FX100 での結果（約 69 秒）から約 28%の高速化となる約 50 秒であった。メモリビジー時間がほぼ全経過時間を律速しており、メモリバンド幅ネックのループであることにより、演算コストではなくメモリ側のコストで性能が律速されていることが分かる。

一方、もう一つの高コストループであった「exb_realspcal」に対する前述のコスト分析では、レジスタ数の現象による影響はほとんどなく、「literm_k」と同じくメモリバンド幅ネックのループであることが判明していたが、L1 キャッシュミス率が理論値よりも大きかった。スライド p.21 に示した Post-FX100 での性能推定結果では、約 20%の高速化を示しているが、やはり、キャシュのスラッシングを反映して、L2 ビジーとなっているため、データを L1 キャッシュに適切に乗せるようなチューニングが必要となる。

4.1.5　高コストループのチューニング

GKV カーネルコードにおいて 2 番目に高い計算コストを生じていた「exb_realspcal」に対して、Post-FX100 向けの最適化に取り組んだ。前述のように、このループは、高い L1 キャッシュミス率のために L2 ビジーとなっていたため、L1 キャッシュに載せるためのチューニングが必要である。そこで、スライド p.23 に示すように、該当ループについて、（nx,ny）の 2 次元ループ内の演算を 2 つに分割し、それぞれを別々のループに分割することで、L1 キャッシュミスの低減を期待した。結果は、FX100 の PA 情報（スライド p.24）に示す通り、128reg および 32reg コンパイラともに、ループ分割しない場合に 5%以上あった L1D キャッシュミス率が、2.7%弱に低減しており、理論値（3.125%）以下に改善されたことが確かめられた。この PA 情報に基づいて、Post-FX100 での性能を推定した結果、ループ分割しない場合の FX100 での結果（約 4.7 秒）に対して、35%の高速化となる約 3.1 秒まで性能向上することが確かめられた。詳細 PA 情報からは、L2 アクセス待ちが改善されて、スラッシングが解消されていることが分かる（スライド p.26）。推定に関しても、L2 ビジーの上限に近い性能となっている。

4.1.6　まとめ

プラズマ乱流コード GKV のカーネルコードを対象として、Post-FX100 における性能の推定を行った。GKV は、プラズマ流体解析コードと異なり、一本の磁力線に沿った局所フラックス・チューブ上でシミュレーションを行う。5 次元位相空間は、空間離散化として磁力線垂直方向の 2 次元空間に対してはフーリエ・スペクトル法、磁力線平行方向の実空間、および 2 次元の速度空間方向については、4 次または 5 次精度差分法を用いており、時間積分には 4 次精度陽的 Runge

-Kutta-Gill 法を採用している。本性能推定は、GKV カーネルコードを対象に 1 ノード 1 プロセスでの単体性能をベースに進めた。まず、FX100 上でコスト分析を行った結果、「literm_k」および「exb_realspcal」のサブルーチンがそれぞれ全体の約 53.7%、約 13.6%を占めていたため、この 2 つを評価対象にした。

対象のループは双方ともに、128reg から 32reg へのコンパイラの変更による性能の劣化はほとんど見られず、メモリビジー率が高く、メモリバンド幅ネックのループであることが分かった。「literm_k」での L1D キャッシュのミス率は理論値付近であった一方、「exb_realspcal」では、理論値を大きく超えていたため、スラッシングが発生している可能性を確認した。

性能予測ツールを用いた Post-FX100 での性能推定の結果、「literm_k」は、上述のとおり、メモリバンド幅ネックのループであることにより、演算コストよりもメモリ側のコストで性能が律速され、FX100 から約 28%の高速化が確認された。もう一方の「exb_realspcal」については、同様にメモリバンド幅ネックのループであると同時に、L1 キャッシュミス率が理論値よりも大きかったため、Post-FX100 での性能推定の結果は、約 20%の高速化を示し、キャシュのスラッシングにより L2 ビジーとなり、そこが性能を律速していることが分かった。

次に、「exb_realspcal」におけるキャシュのスラッシングを回避するため、ループ内の演算を 2 つに分割し、それぞれを別々のループに分割することによる L1 キャッシュミスの低減を期待したチューニングを行った。チューニング後の性能は、L1D キャッシュミス率が理論値以下に改善され、Post-FX100 での性能は最適化前に対して 35%の向上を確認した。詳細 PA 情報から、L2 アクセス待ちが改善されて、スラッシングが解消されていることが分かった。

今回の性能推定では、高コストとなっているメモリバンド幅ネックのループはソフトウェア・パイプライン化や SIMD 化がなされていて、性能的な課題がない中、高コストループの一つであった「exb_realspcal」では、ループ分割を施した最適化後の性能では、L2 ビジーのループとなった。今後のさらなる最適化としては、ブロッキングにより性能向上の可能性がある。

核融合プラズマ乱流コードGKVの Post-FX100性能推定

沼波 政倫 (自然科学研究機構 核融合科学研究所)
片桐 孝洋 (名古屋大学 情報基盤センター)
青木 正樹 (富士通株式会社)

アウトライン

- プログラム概要
- ポストFX100向けチューニング用コンパイラによるコスト分析
- Post-FX100における性能の推定
- 高コストループのチューニング
- まとめ

GKVコード

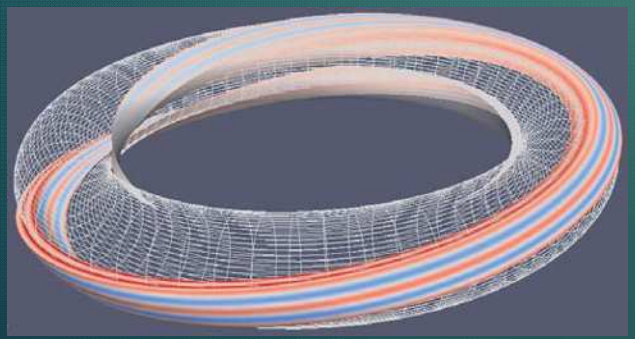
T.-H. Watanabe, NF (2006), M. Nunami, PFR (2010, 2015),
S. Maeyama, CPC (2013), M. Nakata, CPC (2015)

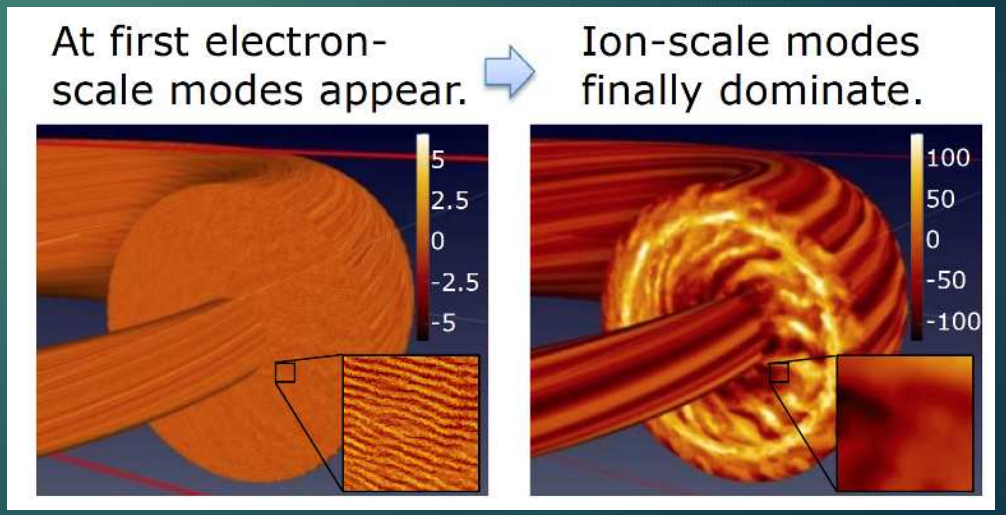
- GyroKinetic Vlasov code : One of major gyrokinetic codes in Japan
- Local delta-f model in a flux-tube geometry (shorter time than full-f model).
- Solve gyrokinetic Vlasov-Poisson-Ampere equations
  - Reduce computations by eliminating gyration.
  - CFD calculation in 5D phase space.
  - Eulerian (Continuum) solver: spectral in 2D (k_x,k_y)-space (2D-FFT), finite-difference in 3D (z, v||, μ)-space.
- High scalability for wide scale range simulations including ion- and electron-scale.

$$\frac{D\delta f_{sk}}{Dt} + v_{Ts} v_{//} \mathbf{b}^* \cdot \nabla \delta f_{sk} - v_{Ts} \mu \mathbf{b} \cdot \nabla B \frac{\partial \delta f_{sk}}{\partial v_{//}} = -i\mathbf{v}_{ds} \cdot \mathbf{k}_\perp (\delta f_{sk} + \frac{q_s F_{sM}}{T_s} \phi_k J_{0s})$$

$$+ i\mathbf{v}_{*s} \cdot \mathbf{k}_\perp \frac{q_s F_{sM}}{T_s} (\phi_k - v_{Ts} v_{//} A_{//k}) J_{0s} + v_{Ts} v_{//} \frac{q_s F_{sM}}{T_s} E_{//k} + C(\delta f_{sk})$$

$$\lambda_{Di}^2 k_\perp^2 \phi_k = \sum_s q_s \delta n_{sk}^{(p)} \qquad k_\perp^2 A_{//k} = \beta_i \sum_s q_s \delta u_{sk}^{(p)}$$





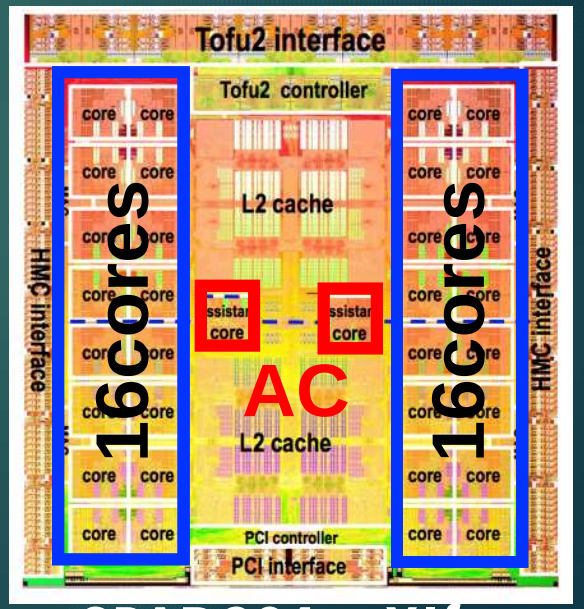
Comm.-Compt. overlap w/ Assistant Core

- Comm. – Compt. overlap using “OpenMP Master”+MPI
  - → MPI comm. on OMP Master thread and the independent calculations on the other threads overlaps.
  - → Utilizing the dynamic scheduling in OMP.
  - → Unable: “Static” scheduling in OMP.

Idomura, Int’l J.HPC Appl. 2014,
Maeyama, Parallel Compt. 2015

- Comm. – Compt. overlap using Assistant Core(AC) + MPI on FX100
  - → MPI comm. on AC and the indep. calculations on “ALL threads” overlaps.
  - → Enable: fully non-blocking communications, e.g., Send/Recv, AlltoAll, Allreduce.
  - → Enable: ”Static” scheduling or “chunk-size-optimized dynamic” sched.

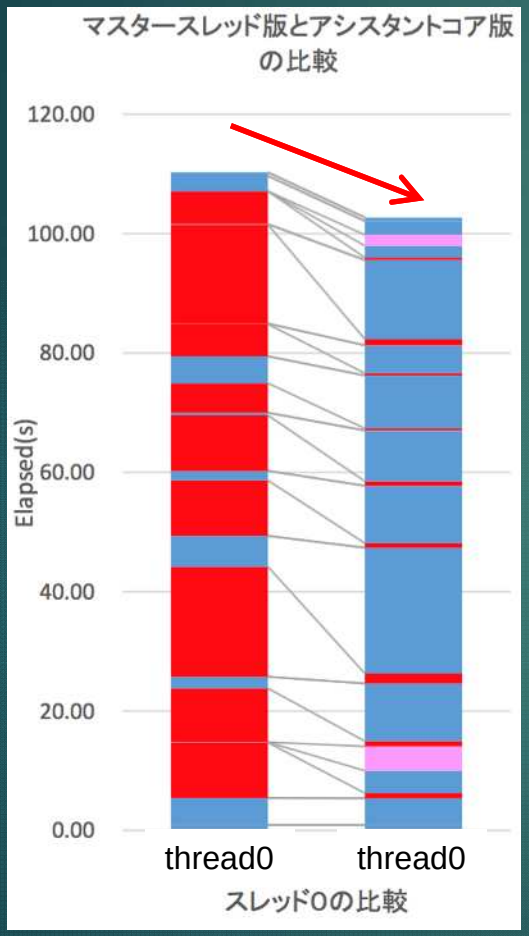
The code optimization and the performance evaluation are in progress at NIFS.



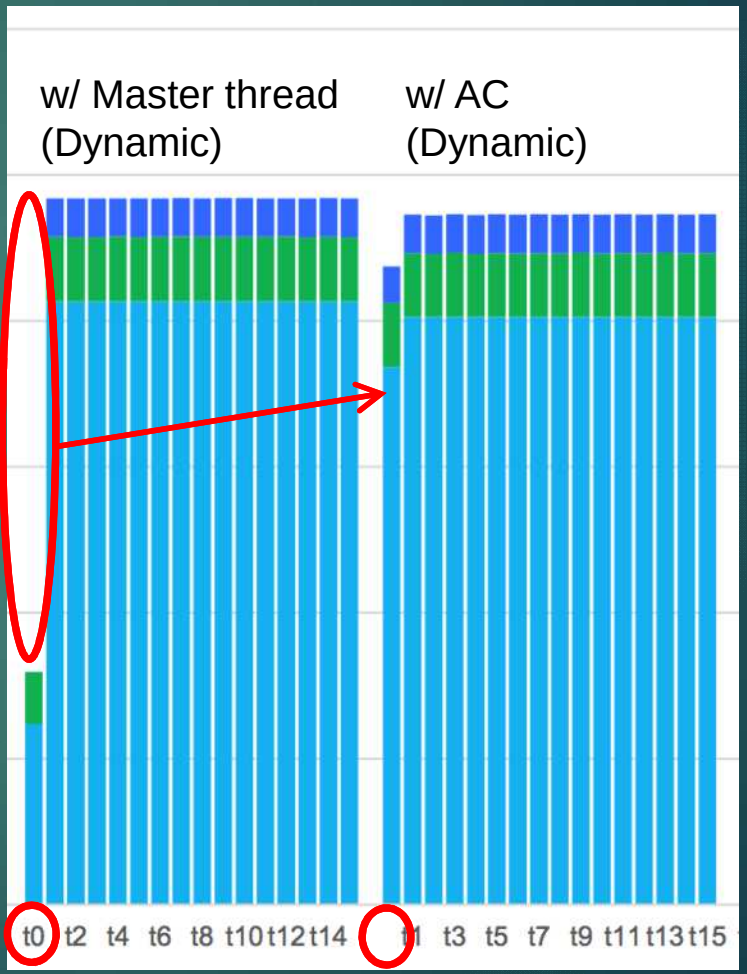
SPARC64TM XIfx

Comm.-Compt. overlap w/ Assistant Core

Master → AC



Integration of calc. costs



→ Almost ideal improvement of ~6.25% = (1/16) by AC.

評価環境

FUJITSU

- FX100 (**ハードウェア、ソフトウェアの情報**)

		FX100 諸元
CPU	名称	SPARC64 XIfx
	動作周波数	1.975GHz
	コア数	32コア＋アシスタントコア
	キャッシュメモリ容量	L1I$: 64KB/コア(4way) L1D$: 64KB/コア(4way) L2$: 24MB/CPU
	理論ピーク性能	1011GFlops/CPU(倍精度) 2022GFlops/CPU(単精度)
	主記憶-CPU間帯域	(240+240)GB/s/CPU
ノード	CPU数、コア数	1CPU/ノード (16コアx2CMG＋アシスタントコア)/ノード
	理論ピーク性能	1011GFlops/ノード(倍精度) 2022GFlops/ノード(単精度)
	主記憶容量	32GB/ノード
	主記憶-CPU間帯域	(240+240)GB/s
	ノード間通信帯域	100GB/s
OS の名称とバージョン		FX100向けOS 2.1.1

測定条件と分析する高コストループの選定

FUJITSU

FX100 の測定条件

項目	条件
コンパイラ	ポストFX100向けチューニング用コンパイラ
翻訳時オプション	ベース：-Kfast -Kparallel -Kocl -Cpp -X9 32reg：-Knounroll 、ループ分割時：-Kloop_nofusion
パラメータ	ループ回転数の指定 (並列化の軸は48回転)：nm=47 、 nz=24 (前回 16回転時：nm=15 、 nz=16)
スレッド数	16スレッド
実行条件	1ノード – 1プロセス (MPI 無し)

分析する高コストループの選定

- 16スレッド(128reg)のプロファイラ情報(Thread 0 の抜粋)から、上位2つのループ(合計で約67%のコスト)を分析する。

Cost	%	Operation (S)	Barrier	%	Nest	Kind	Exec	Start	End		
1199	100.0000	16.1980	14	1.1676	--	--	--	--	--	Thread 0	
644	53.7114	8.7002	0	0.0000	5	DO	AUTO	62	69	gkv_advnc.literm_k._PRL_1_	①
164	13.6781	2.2156	0	0.0000	4	DO	AUTO	280	302	gkv_exb.exb_realspcal._PRL_1_	②
80	6.6722	1.0808	0	0.0000	4	DO	AUTO	71	99	gkv_colli.colli_lb_model._PRL_1_	
24	2.0017	0.3242	0	0.0000	4	DO	AUTO	102	106	gkv_exb.exb_input._PRL_1_	

高コストループ①
literm_k の分析

- FX100 の分析
  - 最適化の状況
  - 128reg と 32regコンパイラ の比較

ループ①：最適化の状況 (高コスト部のソースリスト)

FUJITSU

■ 高コスト1位：gkv_advnc.literm_k._PRL_1_ (Kernel　literm_k 区間)

```
50           !$OMP parallel do private(iv,iz,my,mx)
           <<< Loop-information Start >>>
           <<<  [PARALLELIZATION]
           <<<    Standard iteration count: 2       (ネスト2と3のループはコンパイラでループ交換される)
           <<< Loop-information  End >>>
51  1 pp          do im = 0, nm                     ⇒ 並列化の軸となるループ：48回転 (0～48) nm は 47
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    INTERCHANGED(nest: 3)            (ネスト2と3のループはコンパイラでループ交換される)
           <<< Loop-information  End >>>
53  2  p            do iv = 1, 2*nv                ⇒ 16回転 (1～16) nv は 8
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    INTERCHANGED(nest: 2)
           <<< Loop-information  End >>>
59  3  p              do iz = -nz, nz-1            ⇒ 48回転 (-24～23) nz は 24
61  4  p                do my = ist_y, iend_y      ⇒ 21回転 (0～20) ist_y は 0 , iend_y は 20
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 4)
           <<<    SOFTWARE PIPELINING                 (ソフトウェアパイプライニングは、ループの回転数が84回以上で実行)
           <<< Loop-information  End >>>                                  (32regコンパイラでは20回以上で実行)
62  5  p 3v               do mx = -nx, nx          ⇒ 171回転 (-85～85) nx は85
63  5  p 3v                 lf(mx,my,iz,iv,im) =                          &
64  5                          - ui * kvd(mx,my,iz,iv,im) * ff(mx,my,iz,iv,im)    &
65  5                          - cs1 * fmx(iz,iv,im) * (                  &
66  5                              + ui * kvd(mx,my,iz,iv,im) * psi(mx,my,iz,im)  &
67  5                              - ui * kvs(mx,my,iz,iv,im)             &
68  5                                   * ( psi(mx,my,iz,im) - cs2 * vl(iv) * chi(mx,my,iz,im)))
69  5  p 3v               end do
70  4  p                end do
71  3  p              end do
72  2  p            end do
73  1  p          end do
75           !$OMP end parallel do
```

倍精度複素数 (complex)
倍精度実数　(real)

ループ①：128reg と 32regコンパイラ の比較(1/2)

FUJITSU

PAの実行時間で比較

128reg と 32regコンパイラで、性能差はほぼ出ていない事から、レジスタ数減少の影響は、小さいと考えられる。

literm_k	レジスタ数	スレッド数	実行時間(sec)	比率
FX100 (PA実行時間)	128reg	16	69.13	1.00
	32reg	16	69.09	0.99

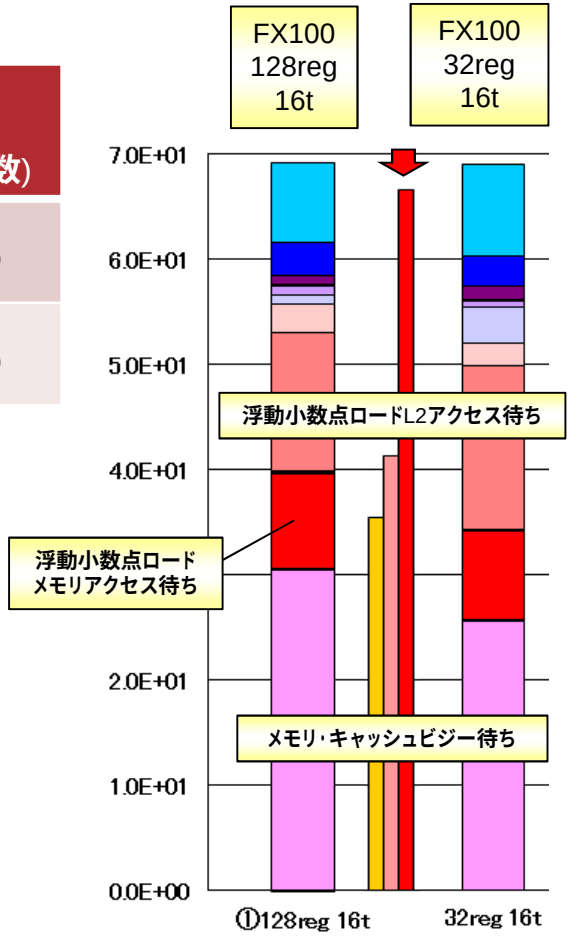
- 32regでは128regと同様にSWPを動作させるため、-Knounroll を指定

ループ①：128reg と 32regコンパイラ の比較(2/2)

FUJITSU

- **PA情報で比較 (Memory、Cache)**
  - 128reg と 32regコンパイラ共にメモリビジー率が高く、メモリバンド幅ネックのループである。
  - L1キャッシュのミス率も、理論値(3.15%)付近である。

literm_k	レジスタ数	スレッド数	メモリビジー率	メモリスループット (GB/sec)	L1D ミス率 (/ロード・ストア数)	L1D ミス dm率 (/L1D ミス数)
FX100 (PA情報)	128reg	16	96%	136.68	3.22%	7.31%
	32reg	16	96%	136.76	3.18%	7.62%



高コストループ②
exb_realspcal の分析

- FX100 の分析
  - 最適化の状況
  - 128reg と 32regコンパイラ の比較

ループ②：最適化の状況 (高コスト部のソースリスト)

FUJITSU

■ 高コスト2位：gkv_exb.exb_realspcal._PRL_1_ (Kernel exb_realspcal 区間)

```
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    INTERCHANGED(nest: 2)            (ネスト1と2のループはコンパイラでループ交換される)
                <<<    PREFETCH       : 2
                <<<      vl: 2
                <<< Loop-information  End >>>
274  1  p             do iv = 1, 2*nv                  ⇒ 16回転 (1～16) nv は 8
277  1          !$OMP parallel do private(mx,my)
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 2
                <<<  [OPTIMIZATION]
                <<<    INTERCHANGED(nest: 1)            (ネスト1と2のループはコンパイラでループ交換される)
                <<< Loop-information  End >>>
278  2 pp              do iz = -nz, nz-1                ⇒並列化の軸となるループ： 48回転 (-24～23) nz は 24
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH       : 20
                <<<      wkexw_xw: 10, wkdf2_xw: 10
                <<< Loop-information  End >>>
279  3  p               do mx = ist_xw, iend_xw        ⇒ 128回転 (0～127) ist_xw は 0 , iend_xw は 127
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 4)                       ※ 128reg ではフルアンロールが動作。32regコンパイラでは、
                <<<    FULL UNROLLING                    ※ アンロールではなくSWPL が適用される。(回転数が36回以上で実行)
                <<< Loop-information  End >>>
280  4  p   fv            do my = 0, nyw                ⇒ 65回転 (0～64) nyw は 64
288  4  p   fv              wkdf1_xw(my,mx,iz,iv) =                                  &
289  4                        cmplx(      real ( wkdf1_xw(my,mx,iz,iv), kind=DP )  &
290  4                                  * real ( wkeyw_xw(my,mx,iz) - cs1 * vl(iv) &
291  4                                         * wkbyw_xw(my,mx,iz),    kind=DP )  &
292  4                                -     real ( wkdf2_xw(my,mx,iz,iv), kind=DP )  &
293  4                                    * real ( wkexw_xw(my,mx,iz) - cs1 * vl(iv) &
294  4                                           * wkbxw_xw(my,mx,iz),    kind=DP )  &
295  4                                ,     aimag ( wkdf1_xw(my,mx,iz,iv)          )  &
296  4                                   * aimag ( wkeyw_xw(my,mx,iz) - cs1 * vl(iv) &
297  4                                           * wkbyw_xw(my,mx,iz)             )  &
298  4                                -     aimag ( wkdf2_xw(my,mx,iz,iv)          )  &
299  4                                   * aimag ( wkexw_xw(my,mx,iz) - cs1 * vl(iv) &
300  4                                           * wkbxw_xw(my,mx,iz)             )  &
301  4                                , kind=DP ) * cef
302  4  p   fv            end do
303  3  p               end do
304  2  p             end do
306  1          !$OMP end parallel do
307  1  p           end do
```

倍精度複素数 (complex)
倍精度実数 (real)

ループ②：128reg と 32regコンパイラ の比較(1/2)

FUJITSU

PAの実行時間で比較

128reg と 32regコンパイラで、性能差は大きく出ていない事から、レジスタ数減少の影響は、小さいと考えられる。

exb_realspcal	レジスタ数	スレッド数	実行時間(sec)	比率
FX100 (PA実行時間)	128reg	16	4.72	1.00
	32reg	16	4.57	0.96

- 32regでは、SWPを動作させるため、-Knounroll を指定
  128regは、フルアンローリング動作

レジスタ数で最適化が異なる
128reg：SIMD + FULL UNROLLING
32reg：SIMD + SWPL

ループ②：128reg と 32regコンパイラ の比較(2/2)

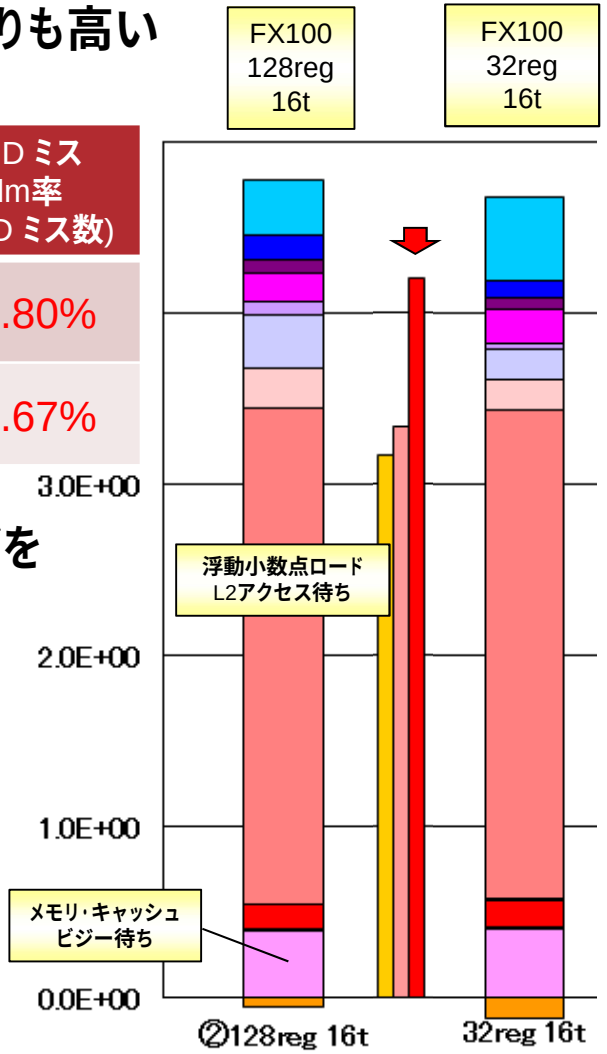
FUJITSU

PA情報で比較 (Memory、Cache)

- 128reg と 32regコンパイラ共にメモリビジー率が高く、メモリバンド幅ネックのループであるが、L1D ミス率が理論値(3.15%)よりも高いため、スラッシングが発生していると考えられる。

exb_realspcal	レジスタ数	スレッド数	メモリビジー率	メモリスループット(GB/sec)	L1D ミス率(/ロード・ストア数)	L1D ミス dm率(/L1D ミス数)
FX100 (PA情報)	128reg	16	89%	119.74	5.12%	48.80%
	32reg	16	85%	115.82	5.24%	50.67%

- スラッシングの対応として、ループ分割によるチューニングを実施する。(前回発表時と同様)



GKV
Post-FX100推定

- **ポストFX100のアーキテクチャ**
- **性能予測ツールの概要**
- **高コストループ① literm_k の分析**
- **高コストループ② exb_realspcal の分析**

性能予測ツールの概要

FUJITSU

アプリコード（カーネル部）

FX100：32regコンパイラを使用し
詳細プロファイラ実行でＰＡ情報を取得

PA情報：詳細プロファイラの出力結果(csvファイル)

性能予測ツール

性能予測値

- 性能予測ツール概略
  - FX100のＰＡ情報を入力とする
  - ハード、命令セット／コンパイラによる性能向上効果を算出する
  - 性能予測値を出力する

FUJITSU

高コストループ①
literm_k の分析

- Post-FX100推定
  - FX100 の実行時間と Post-FX100推定値の比較
  - 推定性能について

ループ①：Post-FX100推定

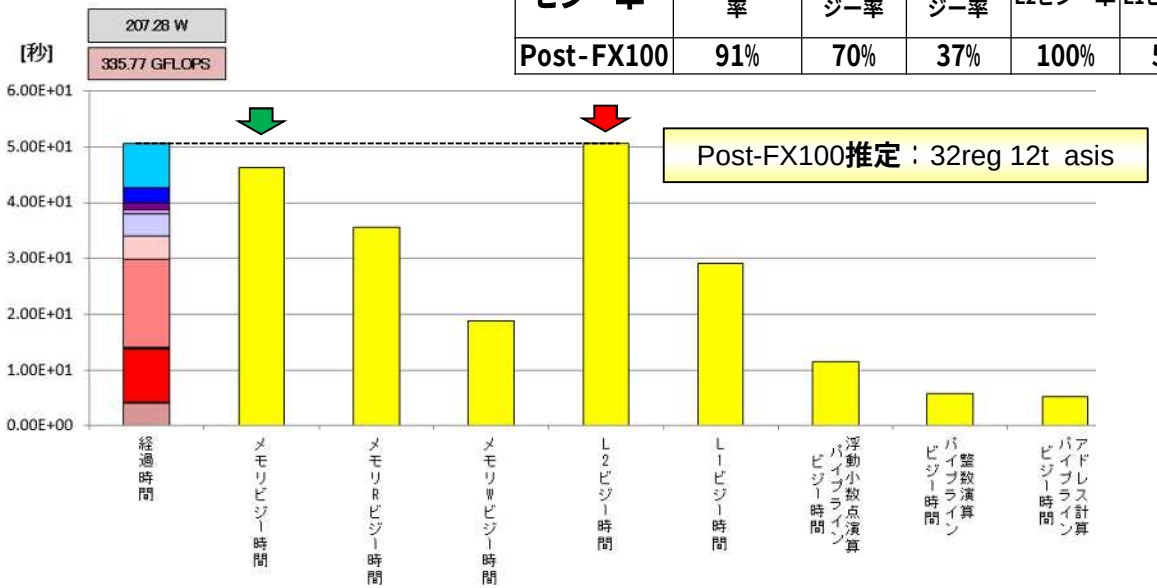
FUJITSU

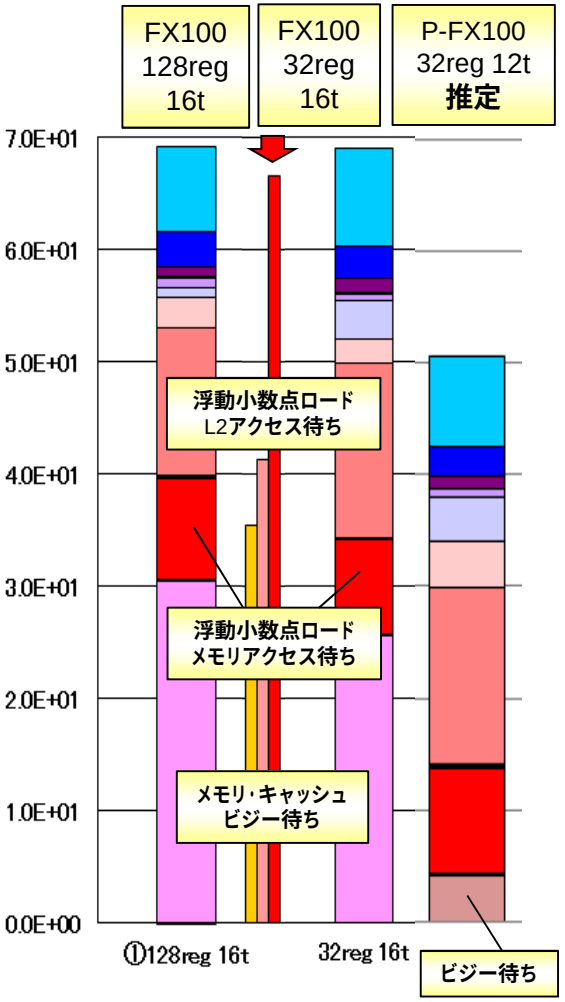
- FX100 の実行時間と Post-FX100推定値の比較

literm_k	レジスタ数	スレッド数	実行時間(sec)	比率
FX100 (PA実行時間)	128reg	16	69.13	1.00
	32reg	16	69.09	0.99
Post-FX100 (推定値)	32reg	12	50.46	0.72

- 推定性能について
  - メモリバンド幅ネックのループであり、演算のコストよりも、メモリ側のコストで性能が律速するため、Post-FX100への移行に関して、特に問題はない。

ビジー率	メモリビジー率	メモリRビジー率	メモリWビジー率	L2ビジー率	L1ビジー率
Post-FX100	91%	70%	37%	100%	57%





高コストループ②
exb_realspcal の分析

- Post-FX100推定
  - FX100 の実行時間と Post-FX100推定値の比較
  - 推定性能について

ループ②：Post-FX100推定

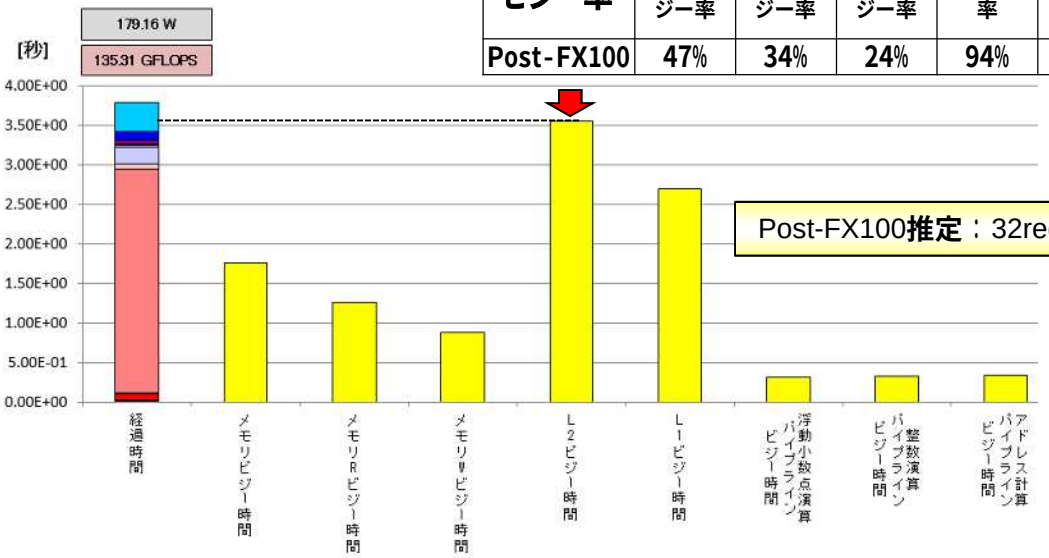
FUJITSU

- FX100 の実行時間と Post-FX100推定値の比較

exb_realspcal	レジスタ数	スレッド数	実行時間(sec)	比率
FX100 (PA実行時間)	128reg	16	4.72	1.00
	32reg	16	4.57	0.96
Post-FX100 (推定値)	32reg	12	3.78	0.80

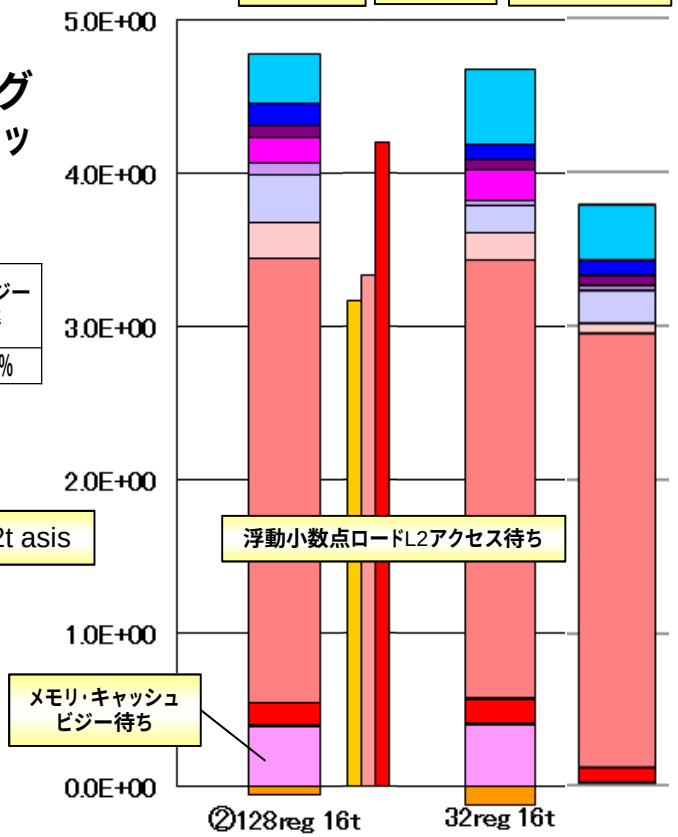
- 推定性能について
  - メモリバンド幅ネックのループであるが、キャシュのスラッシングにより、Post-FX100の推定ではL2ビジーとなるため、L1キャッシュに乗るようなチューニング(ループ分割など)を実施すると、性能改善が期待できる。(前回発表時と同様)

ビジー率	メモリビジー率	メモリRビジー率	メモリWビジー率	L2ビジー率	L1ビジー率
Post-FX100	47%	34%	24%	94%	72%



Post-FX100推定：32reg 12t asis

レジスタ数で最適化が異なる
128reg：SIMD+FULL UNROLLING
32reg：SIMD+SWPL



高コストループ②
exb_realspcal のチューニング

- チューニング(ループ分割)の実施
  - 最適化の状況
  - Post-FX100推定

ループ②：ループ分割：最適化の状況

FUJITSU

高コスト2位を手動でループ分割

```
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 2
                <<< Loop-information  End >>>
266  1 pp            do iz = -nz, nz-1           ! 手動でループ交換  ※コンパイラで自動的に交換された場合、自動並列化の回転軸が変化するため。
268  1             !$OMP parallel do private(mx,my)
269  2  p              do iv = 1, 2*nv            ! 手動でループ交換
271  2             !-------------------------------------------------------------------
272  3  p               do mx = ist_xw, iend_xw  ! 手動でループ分割
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 4)
                <<<    SOFTWARE PIPELINING        ! 32regではレジスタが不足するため -Knounroll を指定し、最適化を FULL UNROLLING から SWPL に変更
                <<< Loop-information  End >>>
273  4  p  v             do my = 0, nyw
274  4  p  v               wkdf1_xw(my,mx,iz,iv) =                              &
275  4                       cmplx(      real ( wkdf1_xw(my,mx,iz,iv), kind=DP )  &
276  4                                 * real ( wkeyw_xw(my,mx,iz) - cs1 * vl(iv) &
277  4                                        * wkbyw_xw(my,mx,iz),    kind=DP )  &
278  4                             ,    aimag ( wkdf1_xw(my,mx,iz,iv)          )  &
279  4                                * aimag ( wkeyw_xw(my,mx,iz) - cs1 * vl(iv) &
280  4                                        * wkbyw_xw(my,mx,iz)             )  &
281  4                             , kind=DP )
282  4  p  v             end do
283  3  p               end do
284  2             !-------------------------------------------------------------------
285  3  p               do mx = ist_xw, iend_xw  ! 手動でループ分割
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 4)
                <<<    SOFTWARE PIPELINING        ! 32regではレジスタが不足するため -Knounroll を指定し、最適化を FULL UNROLLING から SWPL に変更
                <<< Loop-information  End >>>
286  4  p  v             do my = 0, nyw
287  4  p  v               wkdf1_xw(my,mx,iz,iv) = (wkdf1_xw(my,mx,iz,iv) +     &
288  4                       cmplx(-     real ( wkdf2_xw(my,mx,iz,iv), kind=DP )  &
289  4                                 * real ( wkexw_xw(my,mx,iz) - cs1 * vl(iv) &
290  4                                        * wkbxw_xw(my,mx,iz),    kind=DP )  &
291  4                             ,-   aimag ( wkdf2_xw(my,mx,iz,iv)          )  &
292  4                                * aimag ( wkexw_xw(my,mx,iz) - cs1 * vl(iv) &
293  4                                        * wkbxw_xw(my,mx,iz)             )  &
294  4                             , kind=DP )) * cef
295  4  p  v             end do
296  3  p               end do
297  2             !-------------------------------------------------------------------
299  2  p              end do
300  1             !$OMP end parallel do
```

倍精度複素数 (complex)
倍精度実数　(real)

ループ②：ループ分割：Post-FX100推定(1/2)

FUJITSU

- ループ分割に関する評価
- FX100のPA情報の比較 (Memory、Cache)
  - ループ分割によりキャシュのスラッシングが解消し、L1Dミス率が理論値(3.125%)以下に改善された。

exb_realspcal	レジスタ数	スレッド数	ループ分割	L1D ミス率 (/ロード・ストア数)	L1D ミスdm率 (/L1D ミス数)
FX100 (PA情報)	128reg	16	なし	5.12%	48.80%
			あり	2.69%	17.76%
	32reg	16	なし	5.24%	50.67%
			あり	2.65%	14.04%

  - L1Dキャッシュスラッシングの目安

L1D ミス率(/ロード・ストア数)
単精度　1.5625%以上 倍精度　3.125%以上

単精度:1/64 (64回に1回ミスをする)
倍精度:1/32 (32回に1回ミスをする)

ループ②：ループ分割：Post-FX100推定(1/2)

FUJITSU

- ループ分割に関する評価
- Post-FX100推定の性能予測(推定時間)の比較
  - ループ分割により、推定性能が改善された。

exb_realspcal	レジスタ数	スレッド数	ループ分割	推定時間(sec)	比率
FX100 (PA実行時間)	128reg	16	なし	4.72	1.00
			あり	4.91	1.03
	32reg	16	なし	4.57	0.96
			あり	4.85	1.02
Post-FX100 (推定値)	32reg	12	なし	3.78	0.80
			あり	3.10	0.65

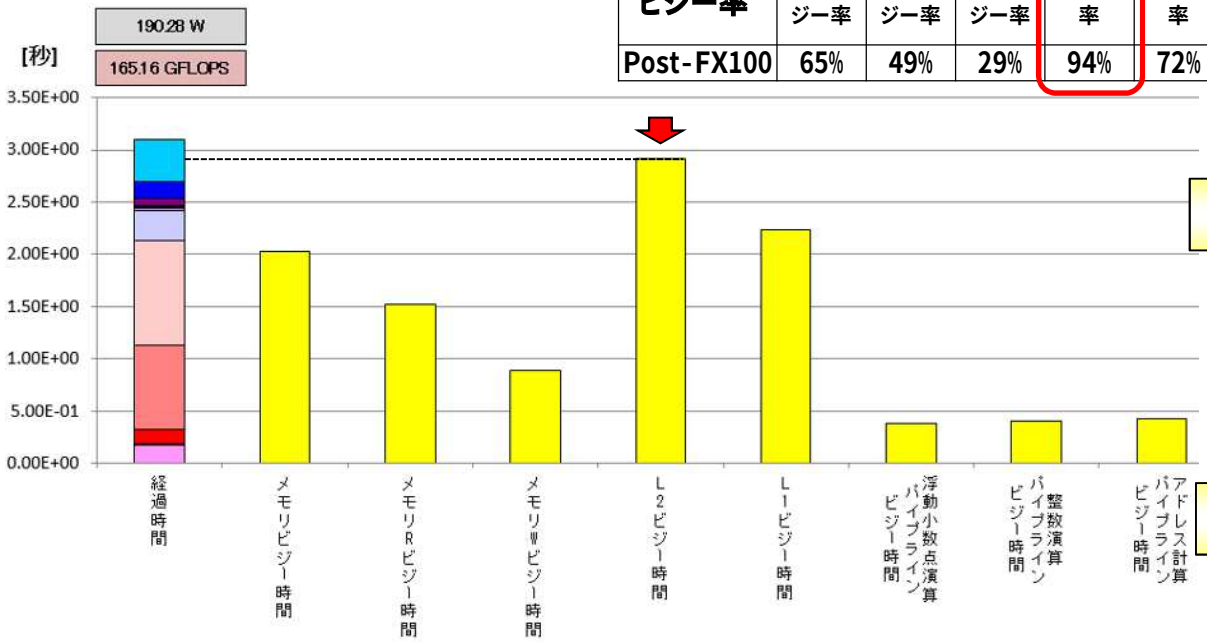
ループ②：ループ分割：Post-FX100推定(2/2)

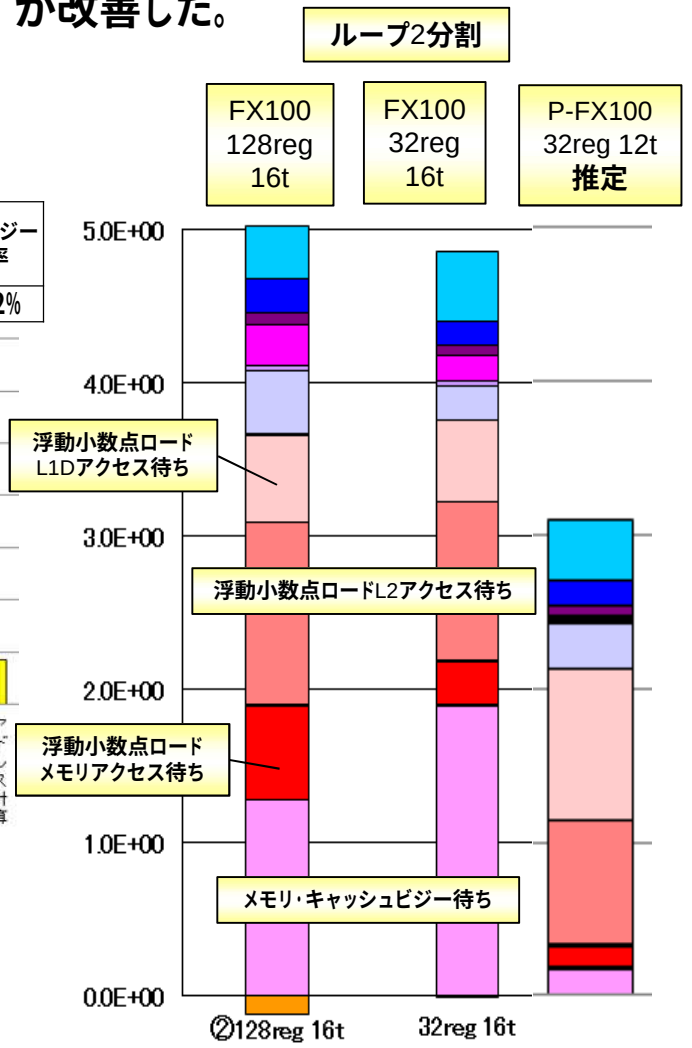
FUJITSU

- **ループ分割時のPost-FX100推定PA情報の比較**
  - **ループ分割により、スラッシングが解消され、L2アクセス待ち が改善した。**
  - **推定に関しても、L2ビジーの上限に近い性能となっている。**

Post-FX100推定：32reg 12t ループ2分割

ビジー率	メモリビジー率	メモリRビジー率	メモリWビジー率	L2ビジー率	L1ビジー率
Post-FX100	65%	49%	29%	94%	72%





まとめ

- プラズマ乱流コードGKVのPost-FX100における性能の推定を行った
- ポストFX100向けコンパイラによるコスト分析
  - 「literm_k」、「exb_realspcal」が全体の約67%を占める
  - レジスタ数の減少(128reg→32reg)による性能劣化はない
  - メモリビジー率が高く、基本的にメモリバンド幅ネック
  - 「exb_realspcal」では、スラッシングが発生
- 性能推定ツールによるPost-FX100における性能推定
  - 「literm_k」は、FX100から約28%の高速化となり、メモリ側コストで性能が律速
  - 「exb_realspcal」はキャシュ・スラッシングにより約20%の高速化にとどまる
- 高コストループのチューニング
  - ループを分割し、L1キャッシュミスの低減を期待したチューニングを実施
  - 最適化後、約35%の性能向上を確認
  - L2アクセス待ちが改善されて、スラッシングが解消

4.2　宇宙プラズマ５次元ブラソフコード Vlasov5 の性能測定および推定

名古屋大学宇宙地球環境研究所　　梅田　隆行
富士通株式会社　　内藤 俊也

4.2.1　はじめに

希薄で無衝突状態にある宇宙プラズマは、磁気流体力学、イオン運動論、電子運動論などの様々な時空間スケールで物理現象を記述できる。Vlasov コードは、宇宙プラズマの第一原理である Vlasov(無衝突 Boltzmann)方程式と Maxwell 方程式により、プラズマ中の全ての電磁・静電波動と荷電粒子の運動との相互作用を解くシミュレーション手法である(文末資料 p.1、以降ページ番号のみ表記する)。Vlasov 方程式は、位置及び速度の関数として表される位相空間分布関数の保存則である。現実の世界では、位置及び速度は共に 3 次元であるが、既存のコンピュータシステムにおいて 6 次元問題を扱うのは困難であるため、本プロジェクトで用いる Vlasov5 では実空間（位置）を 2 次元、速度空間を 3 次元とした 5 次元問題を扱う。本稿では、前マルチコアクラスタ性能 WG 以降で行ったプログラムの改良点についてまとめ、さらにポスト FX100 での性能を推定した。

4.2.2 プログラム概要

Vlasov コードは、Maxwell 方程式による電磁場の更新と Vlasov 方程式による分布関数の更新から成る。Maxwell 方程式は電磁場の解析で広く用いられている FDTD（Finite Difference Time Domain）法により、その時間発展を解き進める。Maxwell 方程式の計算負荷は Vlasov 方程式の計算負荷に対して通常 0.1%未満であるため、今回は性能評価の対象外とした。

Vlasov 方程式による分布関数の更新は、演算子分離法により以下のように実空間（位置方向）の移流、電場による速度空間の移流及び磁場による速度空間の回転の３つの部分に分解され、それぞれの式が①実空間の移流（position カーネル）、速度空間の②移流（velocity_e カーネル）及び③回転（velocity_b カーネル）に対応する(p.2)。

$$\frac{\partial f_s}{\partial t} + \mathbf{v} \cdot \frac{\partial f_s}{\partial \mathbf{r}} = 0 \tag{1}$$

$$\frac{\partial f_s}{\partial t} + \frac{q_s}{m_s}\mathbf{E} \cdot \frac{\partial f_s}{\partial \mathbf{v}} = 0 \tag{2}$$

$$\frac{\partial f_s}{\partial t} + \frac{q_s}{m_s}(\mathbf{v} \times \mathbf{B}) \cdot \frac{\partial f_s}{\partial \mathbf{v}} = 0 \tag{3}$$

式(1)及び(2)は線形移流方程式であり、演算子“非”分離型の保存型解法を用いる。また式(3)は回転流方程式であり、back-substitution 法と呼ばれる演算子分離型の解法を用いる。またこれらの方程式を解く上で、物理量の保存、解の無振動性、正値性の保証を満たす独自の 5 次精度保存型無振動スキームを用いている(p.3)。

速度空間の移流（velocity_e カーネル）の概要をプログラム 1 に示す(p.4)。実空間 x,y 方向の添え字を i,j、速度空間 vx,vy,vz の添え字を l,m,n とする。このプログラムでは、vx,vy,vz の各方向の 1 次元数値流束を計算したのち 3 次元数値流束を合成している(p.4)。分布関数データの定義は、モーメント計算(速度空間の積分)を高速に行うために f(vx,vy,vz,x,y)となっており、velocity_e カーネルでは数値流束データが l,n,m に依存するために 3 次元配列となる。一方、実空間の移流（position カーネル）では数値流束データが i,j に依存するために(l,m,n,i,j)の 5 次元配列となるべきであるが、分布関数データを転置して f(x,y,vx,vy,vz)と定義することにより、2 次元配列となる。またこの配列の転置により、position カーネルはプログラム 2 の velocity_e カーネルとほとんど同じとなる(p.6)。

プログラム 1：velocity_e カーネルの概要

```
1   DO j=1,Ny
2     DO i=1,Nx
3       DO n=0,Nvz
4         DO m=0,Nvy
5           DO l=0,Nvx
6             dfx(l,m,n)=... !vx 方向のフラックスの計算
7             dfy(l,m,n)=... !vy 方向のフラックスの計算
8             dfz(l,m,n)=... !vz 方向のフラックスの計算
9           END DO
10        END DO
11      END DO
12      DO n=1,Nvz
13        DO m=1,Nvy
14          DO l=1,Nvx
15            f(l,m,n,i,j)=f(l,m,n,i,j)-dfx(l,m,n)+dfx(l-1,m,n) &
16                                      -dfy(l,m,n)+dfy(l,m-1,n) &
17                                      -dfz(l,m,n)+dfz(l,m,n-1)
18          END DO
19        END DO
20      END DO
21    END DO
22 ND DO
```

4.2.3 測定環境およびプログラム

性能測定には、名古屋大学の FX100 を使用した。なお、本測定は単一ノードで行い、2 プロセス・16 スレッドを利用した。またノード間の並列性能の測定は行わない。コンパイラオプションは以下のとおりである。

- -Kfast,openmp,ocl,simd=2 -x250

測定には２つのプログラムを使用した。プログラム２は WG 開始時点での as-is コードであり、プログラム１の 3-11 行目の vx,vy,vz の各速度方向の数値流束を計算するプログラムである。変数 lv,l0,mv,m0,nv,n0,sx,sy,sz は、実空間の変数である電場ベクトルの符号に依存して+1 または-1 のどちらかの値をとり、1 行目のループの外で計算される。また inv3 は parameter 定数 1/3 である。プログラム３は、プログラム２の 30-47 行目の累積和の計算を複数の 1 次元ループに分割したものである。

4.2.4　累積計算のループ分割

プログラム２では､35-46 行目にある 3 次元配列への累計計算において配列要素へのアクセスに依存関係があり、30 行目のループに対するコンパイラの最適化を阻害していた(p.7)。プログラム３では、36,41,46,51 行目のループ内において dfx,dfy,dfz の各配列にアクセスするのはそれぞれ 1 回であり、配列要素へのアクセスに対して依存関係が無いため、各ループについてコンパイラの最適化が促進される(p.8)。ただし、30 行目のループにおける gfxy,gfyz,gfzx,gfxyz の計算結果を、分割した 36,41,46,51 行目のループに渡す必要がある。このため、プログラム３ではこれらの変数を 1 次元配列として定義する必要が生じた。

プログラム２を 2 プロセス・プロセスあたりの 16 スレッドで実行すると、1 タイムステップあたりの計算に約 3.19 秒掛かった。この最適化によって、プログラム３では計算時間が約 2.48 秒に短縮され､約 1.23 倍の高速化に成功した。ループ分割には配列が増えることによって計算性能が劣化するデメリットもあるが、本例の場合にはコンパイラの最適化による性能改善のメリットの方が大きかった(p.9,10)。

4.2.5 128reg/32reg コンパイラ比較および、ツールによるポスト FX100 の性能推定

FX100 に搭載されている SPARC64XIfx プロセッサのレジスタ数は 128 であるが、ポスト FX100 に採用される A64FX プロセッサではレジスタ数が 32 に削減される。本節では、FX100 上に用意された、レジスタ数の削減を考慮した 32 レジスタ版富士通コンパイラ（以下、32reg コンパイラとよぶ）を用いてプログラム３を測定した。さらに、推定ツールにより、プログラム３のポスト FX100 上での性能を推定した(p.11)。

32reg コンパイラを用いて翻訳したプログラムは、 128reg コンパイラを用いて翻訳したプログラムに対して、プログラム全体で速度が約 68%に低下した(p.12)。なお、128reg コンパイラを用いた場合にはすべてのカーネルにおいて ocl による手動ループアンローリングを用いた時が最も高速であったが、32reg コンパイラを用いた場合は velocity_e カーネルについてのみ大幅な速度低下が生じたため、ocl による手動ループアンローリングを外した。なお、この ocl による手動ループアンローリングについては、ポストペタアプリ性能 WG 成果報告書を参照されたい。負荷の高いループおよび行は、関数 pic5 を呼び出していた(p.13,14,15)。また関数 pic5 内では組み込み関数を呼び出す行の負荷が高い可能性があることが分かった(p.16)。性能劣化が一番激しかった velocity_b カーネルでは、浮動小数点演算待ちが増加しており、レジスタ数が少なくなることで演算待ちが生じた可能性がある(p.17)。

32reg コンパイラによる測定結果を性能推定ツールに入力することにより、ポスト FX100 におけるコードの性能を推定することができる。このツールによると、32reg コンパイラによる測定結果に対して、ポスト FX100 での実行時間は約 1.28 倍になると推定された。ただし、性能推定ツールではノードあたり 2 プロセス・12 スレッドであるのに対し、ポスト FX100 に採用される A64FX プロセッサは 48 コアあるため、実行時間は約 0.64 倍になると推定される(p.18)。本プログラムは演算ネックであり、性能推定ツールが実際の性能より低めの推定値を出す傾向にある。ループ内でスカラ関数 pic5 を呼ぶプログラム構造から、ベクトル(配列)関数を呼ぶプログラム構造に変更し、関数内でループ分割を行うことにより(p.19)、約 1.7 倍の高速化が見込める。これは、128reg コンパイラによる測定結果に対して、ポスト FX100 の 48 コアでの実行時間は約 0.55 倍になることを意味する。

4.2.6 まとめ

性能チューニングによって、式(2)を解くプログラム全体において、約 2.37 倍の高速化がなされた。以下は、本 WG の活動で得られた知見である。

- 配列に対する累積計算は、ループ分割により配列に対するアクセスの依存性を無くすことができ、ループの最適化及び計算の高速化が可能である。
- レジスタ数が 128 から 32 に削減されることにより、プログラム全体で速度が約 68%に低下する可能性がある。これにより、A64FX プロセッサの 48 コアでは、SPARC64 XIfx の 32 コアに対してプログラム全体の実行時間が約 0.94 倍(=0.64/0.68)になると推定される。
- 翻訳指示文(OCL)によるループアンローリング数の調整や、ソフトウェアパイプライニングの抑制により、性能が改善する可能性がある。

- ループ内でスカラ関数を呼ばずに、サブルーチンによりループそのものをベクトル演算することにより、ループ分割などの最適化手法を適用することが、今後の課題として挙げられる。これにより、約 1.7 倍の高速化が見込める。

プログラム2

```
1  do n=0,nvz+1
2    do m=0,nvy+1
3      do l=0,nvx+1
4        ffi(l)=1.0d0/ff(l,m,n,i,j)
5      end do
6      do l=0,nvx+1
7        hm2=ff(l-lv-lv,m,n,i,j)
8        hm1=ff(l-lv   ,m,n,i,j)
9        hp0=ff(l      ,m,n,i,j)
10       hp1=ff(l+lv   ,m,n,i,j)
11       hp2=ff(l+lv+lv,m,n,i,j)
12       gfx(l)=pic5(hp2,hp1,hp0,hm1,hm2,ex(i,j))
13     end do
14     do l=0,nvx+1
15       hm2=ff(l,m-mv-mv,n,i,j)
16       hm1=ff(l,m-mv   ,n,i,j)
17       hp0=ff(l,m      ,n,i,j)
18       hp1=ff(l,m+mv   ,n,i,j)
19       hp2=ff(l,m+mv+mv,n,i,j)
20       gfy(l)=pic5(hp2,hp1,hp0,hm1,hm2,ey(i,j))
21     end do
22     do l=0,nvx+1
23       hm2=ff(l,m,n-nv-nv,i,j)
24       hm1=ff(l,m,n-nv   ,i,j)
25       hp0=ff(l,m,n      ,i,j)
26       hp1=ff(l,m,n+nv   ,i,j)
27       hp2=ff(l,m,n+nv+nv,i,j)
28       gfz(l)=pic5(hp2,hp1,hp0,hm1,hm2,ez(i,j))
29     end do
30     do l=0,nvx+1
31       gfxy  = abs((gfx(l)*ffi(l))* gfy(l))                *0.5d0
32       gfyz  = abs((gfy(l)*ffi(l))* gfz(l))                *0.5d0
33       gfzx  = abs((gfz(l)*ffi(l))* gfx(l))                *0.5d0
34       gfxyz = abs((gfx(l)*ffi(l))*(gfy(l)*ffi(l))*gfz(l))*inv3
35       dfx(l+l0,m   ,n   ) = dfx(l+l0,m   ,n   ) +(gfx(l)+(gfxyz-gfxy-gfzx)*sx)
36       dfx(l+l0,m   ,n+nv) = dfx(l+l0,m   ,n+nv)         -(gfxyz-gfzx)     *sx
37       dfx(l+l0,m+mv,n   ) = dfx(l+l0,m+mv,n   )         -(gfxyz-gfxy)     *sx
38       dfx(l+l0,m+mv,n+nv) = dfx(l+l0,m+mv,n+nv)          + gfxyz          *sx
39       dfy(l   ,m+m0,n   ) = dfy(l   ,m+m0,n   ) +(gfy(l)+(gfxyz-gfyz-gfxy)*sy)
40       dfy(l+lv,m+m0,n   ) = dfy(l+lv,m+m0,n   )         -(gfxyz-gfxy)     *sy
41       dfy(l   ,m+m0,n+nv) = dfy(l   ,m+m0,n+nv)         -(gfxyz-gfyz)     *sy
42       dfy(l+lv,m+m0,n+nv) = dfy(l+lv,m+m0,n+nv)          + gfxyz          *sy
43       dfz(l   ,m   ,n+n0) = dfz(l   ,m   ,n+n0) +(gfz(l)+(gfxyz-gfzx-gfyz)*sz)
44       dfz(l   ,m+mv,n+n0) = dfz(l   ,m+mv,n+n0)         -(gfxyz-gfyz)     *sz
45       dfz(l+lv,m   ,n+n0) = dfz(l+lv,m   ,n+n0)         -(gfxyz-gfzx)     *sz
46       dfz(l+lv,m+mv,n+n0) = dfz(l+lv,m+mv,n+n0)          + gfxyz          *sz
47     end do
48   end do
49 end do
```

プログラム3

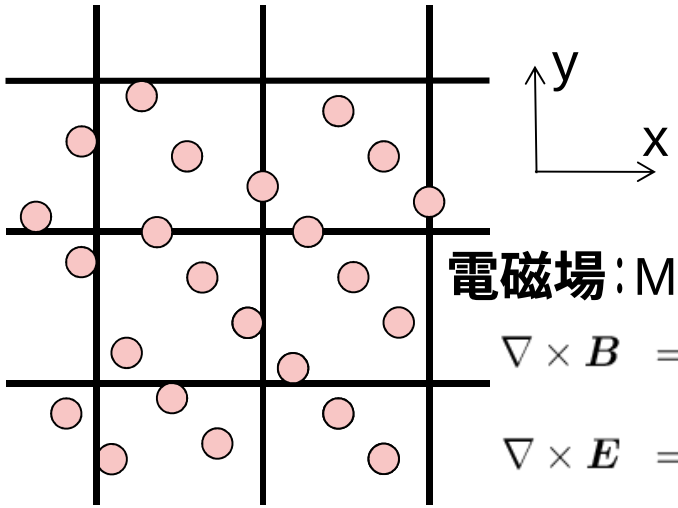
```
30   do l=0,nvx+1
31     gfxy(l)  = abs((gfx(l)*ffi(l))* gfy(l))               *0.5d0
32     gfyz(l)  = abs((gfy(l)*ffi(l))* gfz(l))               *0.5d0
33     gfzx(l)  = abs((gfz(l)*ffi(l))* gfx(l))               *0.5d0
34     gfxyz(l) = abs((gfx(l)*ffi(l))*(gfy(l)*ffi(l))*gfz(l))*inv3
35   end do
36   do l=0,nvx+1
37                            dfx(l+l0,m                ,n                )      =
dfx(l+l0,m   ,n   )+gfx(l)+(gfxyz(l)-gfxy(l)-gfzx(l))*sx
38                            dfy(l                ,m+m0,n                )      =
dfy(l   ,m+m0,n   )+gfy(l)+(gfxyz(l)-gfyz(l)-gfxy(l))*sy
39                            dfz(l                ,m                ,n+n0)      =
dfz(l   ,m   ,n+n0)+gfz(l)+(gfxyz(l)-gfzx(l)-gfyz(l))*sz
40   end do
41   do l=0,nvx+1
42     dfx(l+l0,m   ,n+nv) = dfx(l+l0,m   ,n+nv)       -(gfxyz(l)-gfzx(l))*sx
43     dfy(l+lv,m+m0,n   ) = dfy(l+lv,m+m0,n   )       -(gfxyz(l)-gfxy(l))*sy
44     dfz(l   ,m+mv,n+n0) = dfz(l   ,m+mv,n+n0)       -(gfxyz(l)-gfyz(l))*sz
45   end do
46   do l=0,nvx+1
47     dfx(l+l0,m+mv,n   ) = dfx(l+l0,m+mv,n   )       -(gfxyz(l)-gfxy(l))*sx
48     dfy(l   ,m+m0,n+nv) = dfy(l   ,m+m0,n+nv)       -(gfxyz(l)-gfyz(l))*sy
49     dfz(l+lv,m   ,n+n0) = dfz(l+lv,m   ,n+n0)       -(gfxyz(l)-gfzx(l))*sz
50   end do
51   do l=0,nvx+1
52     dfx(l+l0,m+mv,n+nv) = dfx(l+l0,m+mv,n+nv)       + gfxyz(l)*sx
```

運動論方程式：流体方程式の元となる第一原理
粒子間の衝突が無視できるプラズマ（電離気体）と電磁場を扱う

■ 粒子(Particle-In-Cell)コード

$$\frac{\mathrm{d}\boldsymbol{r_n}}{\mathrm{d}t} = \boldsymbol{v_n}$$
$$\frac{\mathrm{d}\boldsymbol{v_n}}{\mathrm{d}t} = \frac{q_n}{m_n}(\boldsymbol{E} + \boldsymbol{v_n} \times \boldsymbol{B})$$

r = (x,y,z) と v = (vx,vy,vz) のN体
x(n), y(n), z(n), vx(n), vy(n), vz(n)
と電磁場 E(i,j,k), B(i,j,k) を扱う



電磁場：Maxwell方程式

$$\nabla \times \boldsymbol{B} = \mu_0 \boldsymbol{J} + \frac{1}{c^2}\frac{\partial \boldsymbol{E}}{\partial t}$$
$$\nabla \times \boldsymbol{E} = -\frac{\partial \boldsymbol{B}}{\partial t}$$

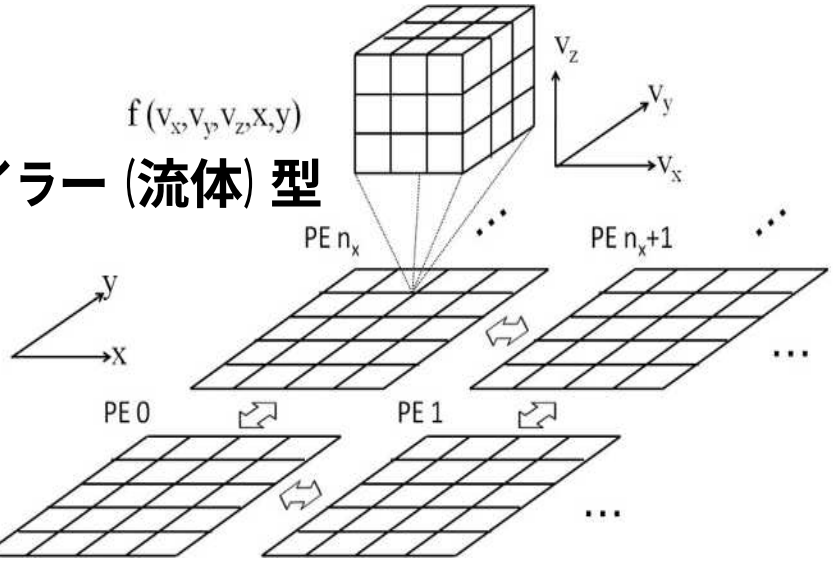
電磁場格子の中を、粒子が動き回る
ノード間並列化（ロードバランス）に課題

■ 分布関数(Vlasov)コード

$$\frac{\partial f}{\partial t} + \boldsymbol{v}\frac{\partial f}{\partial \boldsymbol{x}} + \frac{q}{m}\left(\boldsymbol{E} + \boldsymbol{v} \times \boldsymbol{B}\right)\frac{\partial f}{\partial \boldsymbol{v}} = 0$$

分布関数 f(i,j,k,l,m,n)と
電磁場 E(i,j,k), B(i,j,k) を扱う

オイラー（流体）型



電磁場格子上に分布関数が定義されている
高いスケーラビリティ

コードの概要：Vlasov5D

解いている方程式：

- Maxwell**方程式（電磁場の変化と伝搬）**

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \qquad \nabla \times \mathbf{B} = \mu_0 \mathbf{J} + \frac{1}{c^2}\frac{\partial \mathbf{E}}{\partial t}$$

負荷は0.1%**未満**

- Vlasov（**無衝突**Boltzmann）**方程式（プラズマの運動）**

$$\frac{\partial f_s}{\partial t} + \boldsymbol{v} \cdot \frac{\partial f_s}{\partial \boldsymbol{x}} + \frac{q_s}{m_s}\left(\boldsymbol{E} + \boldsymbol{v} \times \boldsymbol{B}\right) \cdot \frac{\partial f_s}{\partial \boldsymbol{v}} = 0$$

f(x, y, ~~z~~, vx, vy, vz)

3つに分解

$$\frac{\partial f_s}{\partial t} + \boldsymbol{v} \cdot \frac{\partial f_s}{\partial \boldsymbol{x}} = 0$$

（位置更新）

position

$$\frac{\partial f_s}{\partial t} + \frac{q_s}{m_s}\boldsymbol{E} \cdot \frac{\partial f_s}{\partial \boldsymbol{v}} = 0$$

（速度更新：移流）

velocity_e

$$\frac{\partial f_s}{\partial t} + \frac{q_s}{m_s}\left(\boldsymbol{v} \times \boldsymbol{B}\right) \cdot \frac{\partial f_s}{\partial \boldsymbol{v}} = 0$$

（速度更新：回転）

velocity_b

コードの構造　まとめ

- **移流計算**：velocity_e, position
  - velocity_e：**速度微分、演算が速度空間依存**
  - position：**位置微分、演算が実空間依存**

    配列の転置(速度,位置)⇒(位置,速度)により、
    velocity_e**と同じ数値解法**(Umeda+ CPC 2009)**を適用**
- **回転計算**：velocity_b
  - **速度微分、演算が速度空間依存**
  - **剛体回転のため、特殊な演算子分離法**(back-substitution**法**:Schmitz&Grauer CPC 2006)**を使用**
  - vz(loop4)⇒vy(loop5)⇒vx(loop6)⇒vz(loop7)⇒vy(loop8)⇒vz(loop9)**の順に計算。**4**と**9**、**5**と**8**、**6**と**7**は同じプログラム構造を持つ**
- **共通解法**：**5次精度の、保存型・無振動・正値スキーム**
  (Umeda EPS 2008; Umed+ CPC 2012)

$$\frac{\partial f_s}{\partial t}+\frac{q_s}{m_s}\mathbf{E}\cdot\frac{\partial f_s}{\partial \mathbf{v}}=0$$

velocity_e：電場による加速を計算

```
do nn=nvzs-1,nvze+1
 do mm=nvys-1,nvye+1
  do ll=nvxs-1,nvxe+1
   ffi(ll)=1.0d0/ff(ll,mm,nn,ii,jj)
  end do
 do ll=nvxs-1,nvxe+1
   hm2=ff(ll-lv*2,mm,nn,ii,jj)
   hm1=ff(ll-lv  ,mm,nn,ii,jj)
   hp0=ff(ll     ,mm,nn,ii,jj)
   hp1=ff(ll+lv  ,mm,nn,ii,jj)
   hp2=ff(ll+lv*2,mm,nn,ii,jj)
   gfx(ll)=pic5(hp2,hp1,hp0,hm1,hm2,vvx)
  end do
 do ll=nvxs-1,nvxe+1
   hm2=ff(ll,mm-mv*2,nn,ii,jj)
   hm1=ff(ll,mm-mv  ,nn,ii,jj
   hp0=ff(ll,mm     ,nn,ii,jj)
   hp1=ff(ll,mm+mv  ,nn,ii,jj)
   hp2=ff(ll,mm+mv*2,nn,ii,jj)
   gfy(ll)=pic5(hp2,hp1,hp0,hm1,hm2,vvy)
 end do
```

割り算

vx方向流束計算

vy方向流束計算

```
   do ll=nvxs-1,nvxe+1
    hm2=ff(ll,mm,nn-nv*2,ii,jj)
    hm1=ff(ll,mm,nn-nv  ,ii,jj)
    hp0=ff(ll,mm,nn     ,ii,jj)
    hp1=ff(ll,mm,nn+nv  ,ii,jj)
    hp2=ff(ll,mm,nn+nv*2,ii,jj)
    gfz(ll)=pic5(hp2,hp1,hp0,hm1,hm2,vvz)
   end do
   do ll=nvxs-1,nvxe+1
    a =
    b =
    c =          (ffi,gfx,gfy,gfzを使用)
    d =
    dfx(ll+l0,mm   ,nn   ) &
           = dfx(ll+l0,mm   ,nn   ) + a
    dfx(ll+l0,mm+mv,nn   ) &
           = dfx(ll+l0,mm+mv,nn   ) + b
    dfx(ll+l0,mm   ,nn+nv) &
           = dfx(ll+l0,mm   ,nn+nv) + c
    dfx(ll+l0,mm+mv,nn+nv) &
           = dfx(ll+l0,mm+mv,nn+nv) + d
        :
        :
   end do
  end do
end do
```

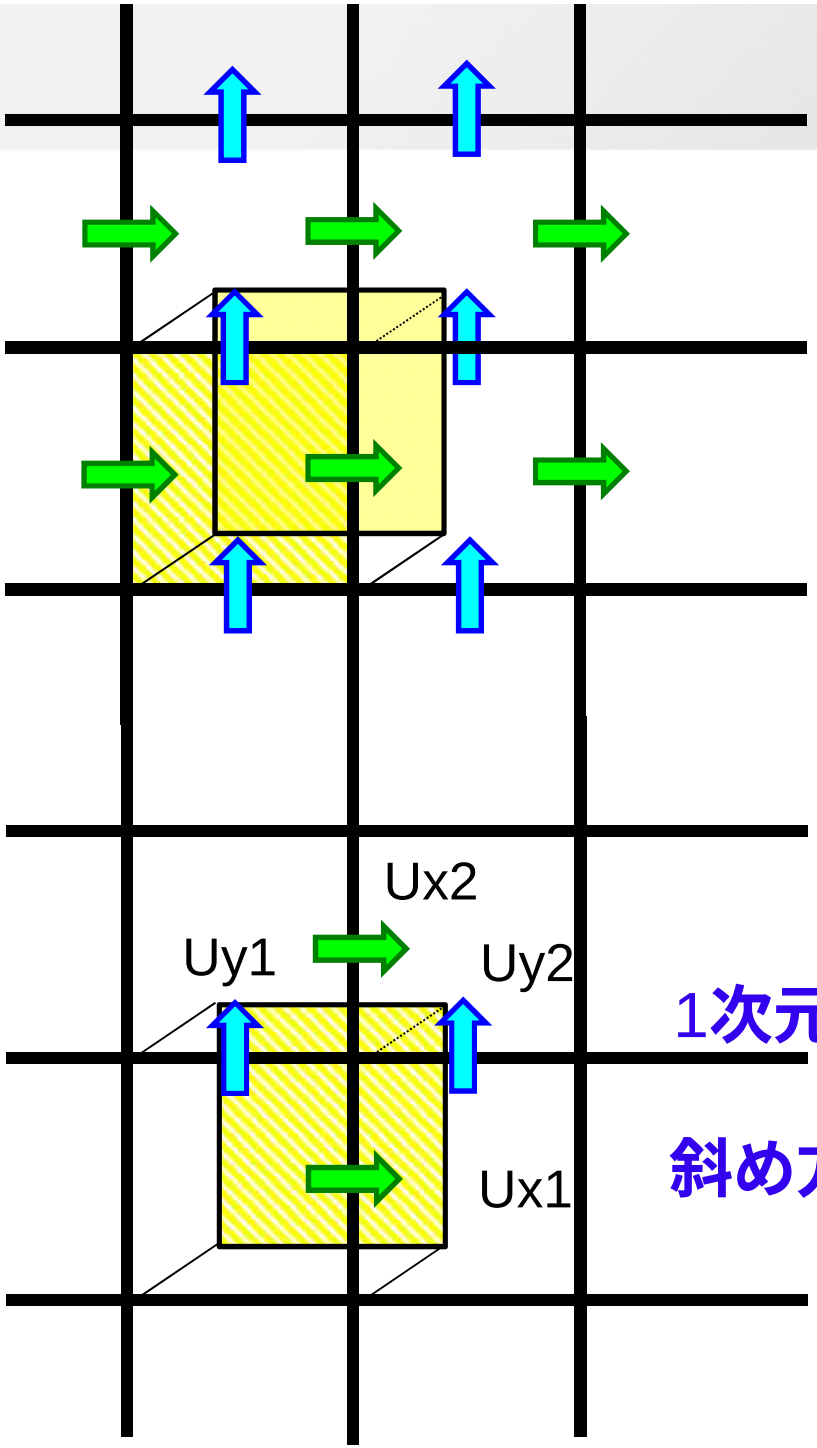
vz方向流束計算

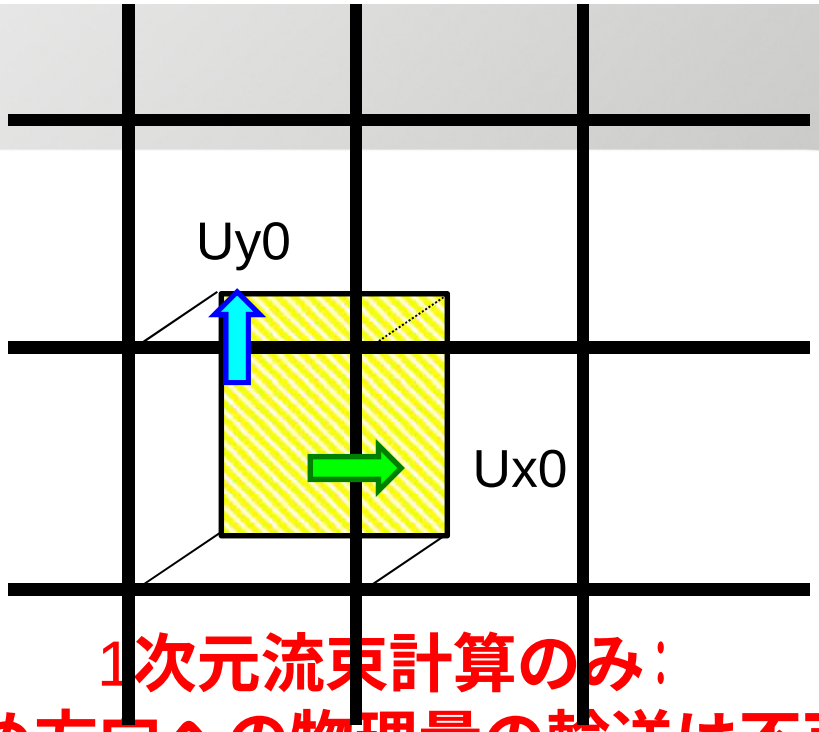
1次元流束から
3次元流束を合成

5次・保存型・無振動・正値スキームにより1次元流束を計算

重い演算（割り算・流束計算）を1次元ループ分割

ループ間のデータ共有を1次元配列で行う





1次元流束計算のみ：
斜め方向への物理量の輸送は不可
⇒多段時間積分(R-K法etc…)

1次元流束から多次元流束を合成し、
周辺格子点の情報を累積
斜め方向への物理量の輸送を可能に

[Umeda et al. CPC 2009]

Ux0=Ux1+Ux2

Uy0=Uy1+Uy2

“position” subroutine

$$\frac{\partial f_s}{\partial t} + \boldsymbol{v} \cdot \frac{\partial f_s}{\partial \boldsymbol{r}} = 0$$

```
f(l,m,n,i,j)=f(l,m,n,i,j) &
    -dfx(l,m,n,i,j)+dfx(l,m,n,i-1,j) &
    -dfy(l,m,n,i,j)+dfy(l,m,n,i,j-1)
```

線形移流　流束の配列が大きい
同じ計算方法 ⇒転置 ftmp(i,j,l,m,n)

```
ftmp(i,j,l,m,n)=f(i,j,l,m,n) &
    -dfx(i,j)+dfx(i-1,j) &
    -dfy(i,j)+dfy(i,j-1)
```

“velocity_e” subroutine

$$\frac{\partial f_s}{\partial t} + \frac{q_s}{m_s}\boldsymbol{E} \cdot \frac{\partial f_s}{\partial \boldsymbol{v}} = 0$$

```
f(l,m,n,i,j)=f(l,m,n,i,j) &
    -dfx(l,m,n)+dfx(l-1,m,n) &
    -dfy(l,m,n)+dfy(l,m-1,n) &
    -dfz(l,m,n)+dfz(l,m,n-1)
```

“velocity_b” subroutine

$$\frac{\partial f_s}{\partial t} + \frac{q_s}{m_s}(\boldsymbol{v} \times \boldsymbol{B}) \cdot \frac{\partial f_s}{\partial \boldsymbol{v}} = 0$$

回転
演算を各方向で分離
(directional splitting)

```
f(l,m,n,i,j)=f(l,m,n,i,j)-dfz(l,m,n)+dfz(l,m,n-1)
f(l,m,n,i,j)=f(l,m,n,i,j)-dfy(l,m,n)+dfy(l,m-1,n)
f(l,m,n,i,j)=f(l,m,n,i,j)-dfx(l,m,n)+dfx(l-1,m,n)
f(l,m,n,i,j)=f(l,m,n,i,j)-dfx(l,m,n)+dfx(l-1,m,n)
f(l,m,n,i,j)=f(l,m,n,i,j)-dfy(l,m,n)+dfy(l,m-1,n)
f(l,m,n,i,j)=f(l,m,n,i,j)-dfz(l,m,n)+dfz(l,m,n-1)
```

プログラム2(as-is)性能分析

● プロファイラ ループコスト

```
*************************************************************************************
    Application - loops
*************************************************************************************
           Cost         %    Nest    Kind     Exec    Start      End
-------------------------------------------------------------------------------------
          89927  100.0000      --      --      --       --       --  Application
-------------------------------------------------------------------------------------
          19623   21.8210       5   DO      OpenMP      137      157  velocity.velocity_e_5d._OMP_1_
           4733    5.2632       5   DO      OpenMP      251      258  velocity.velocity_b_5d._OMP_2_
           4214    4.6860       5   DO      OpenMP      221      229  position.position_5d._OMP_2_
           3967    4.4114       5   DO      OpenMP      126      133  velocity.velocity_e_5d._OMP_1_
           3525    3.9198       5   DO      OpenMP      432      439  velocity.velocity_b_5d._OMP_2_
           3428    3.8120       5   DO      OpenMP      157      159  position.position_5d._OMP_1_
           3350    3.7252       5   DO      OpenMP      115      122  velocity.velocity_e_5d._OMP_1_
           3306    3.6763       5   DO      OpenMP      286      293  velocity.velocity_b_5d._OMP_2_
           3107    3.4550       5   DO      OpenMP      396      403  velocity.velocity_b_5d._OMP_2_
           3094    3.4406       5   DO      OpenMP      104      111  velocity.velocity_e_5d._OMP_1_
```

velocity_eは1タイムステップあたり2回呼ばれる

137,221：flux累積和

104,115,126,251,286,396,432：関数pic5呼出

157：転置

```
velocity_e                *ループ長：42      oclによる手動unrolling
137     5   p  10s        do ll=nvxs-1,nvxe+1
138     5   p  10v          gfxy  = abs((gfx(ll)*ffi(ll))* gfy(ll))                *0.5d0
139     5   p  10v          gfyz  = abs((gfy(ll)*ffi(ll))* gfz(ll))                *0.5d0
140     5   p  10v          gfzx  = abs((gfz(ll)*ffi(ll))* gfx(ll))                *0.5d0
141     5   p  10v          gfxyz = abs((gfx(ll)*ffi(ll))*(gfy(ll)*ffi(ll))*gfz(ll))*inv3
142     5
143     5   p  10v          dfx(ll+l0,mm   ,nn   ) = dfx(ll+l0,mm   ,nn   ) +(gfx(ll)+(gfxyz-gfxy-gfzx)*sx)
144     5   p  10v          dfx(ll+l0,mm   ,nn+nv) = dfx(ll+l0,mm   ,nn+nv)          -(gfxyz-gfzx)     *sx
145     5   p  10v          dfx(ll+l0,mm+mv,nn   ) = dfx(ll+l0,mm+mv,nn   )          -(gfxyz-gfxy)     *sx
146     5   p  10v          dfx(ll+l0,mm+mv,nn+nv) = dfx(ll+l0,mm+mv,nn+nv)          + gfxyz           *sx
147     5
148     5   p  10m          dfy(ll   ,mm+m0,nn   ) = dfy(ll   ,mm+m0,nn   ) +(gfy(ll)+(gfxyz-gfyz-gfxy)*sy)
149     5   p  10m          dfy(ll+lv,mm+m0,nn   ) = dfy(ll+lv,mm+m0,nn   )          -(gfxyz-gfxy)     *sy
150     5   p  10m          dfy(ll   ,mm+m0,nn+nv) = dfy(ll   ,mm+m0,nn+nv)          -(gfxyz-gfyz)     *sy
151     5   p  10m          dfy(ll+lv,mm+m0,nn+nv) = dfy(ll+lv,mm+m0,nn+nv)          + gfxyz           *sy
152     5
153     5   p  10m          dfz(ll   ,mm   ,nn+n0) = dfz(ll   ,mm   ,nn+n0) +(gfz(ll)+(gfxyz-gfzx-gfyz)*sz)
154     5   p  10m          dfz(ll   ,mm+mv,nn+n0) = dfz(ll   ,mm+mv,nn+n0)          -(gfxyz-gfyz)     *sz
155     5   p  10m          dfz(ll+lv,mm   ,nn+n0) = dfz(ll+lv,mm   ,nn+n0)          -(gfxyz-gfzx)     *sz
156     5   p  10m          dfz(ll+lv,mm+mv,nn+n0) = dfz(ll+lv,mm+mv,nn+n0)          + gfxyz           *sz
157     5   p  10v        end do
```

コンパイラによる最適化なし

最適化阻害

プログラム3(ループ分割)性能分析

```
velocity_e                *ループ長：42  oclによる手動unrolling
                 <<<    SIMD(VL: 4)
                 <<<    SOFTWARE PIPELINING
                 <<<    PREFETCH(HARD) Expected by compiler :
                 <<<      ff, (unknown), (unknown)
129      5   p  10v          do ll=nvxs-1,nvxe+1
130      5   p  10v            ffi = 1.0d0/ff(ll,mm,nn,ii,jj)
131      5   p  10v            gfxy(ll)  = abs((gfx(ll)*ffi)* gfy(ll))            *0.5d0
132      5   p  10v            gfyz(ll)  = abs((gfy(ll)*ffi)* gfz(ll))            *0.5d0
133      5   p  10v            gfzx(ll)  = abs((gfz(ll)*ffi)* gfx(ll))            *0.5d0
134      5   p  10v            gfxyz(ll) = abs((gfx(ll)*ffi)*(gfy(ll)*ffi)*gfz(ll))*inv3
135      5   p  10v          end do
                 <<<    SIMD(VL: 4)
                 <<<    SOFTWARE PIPELINING
                 <<<    PREFETCH(SOFT) : 24
                 <<<     SEQUENTIAL : 24
                 <<<      (unknown): 24
138      5   p  10v          do ll=nvxs-1,nvxe+1
139      5   p  10v            dfx(ll+l0,mm   ,nn   ) = dfx(ll+l0,mm   ,nn   ) +(gfx(ll)+(gfxyz(ll)-gfxy(ll)-gfzx(ll))*sx)
140      5   p  10v            dfy(ll   ,mm+m0,nn   ) = dfy(ll   ,mm+m0,nn   ) +(gfy(ll)+(gfxyz(ll)-gfyz(ll)-gfxy(ll))*sy)
141      5   p  10v            dfz(ll   ,mm   ,nn+n0) = dfz(ll   ,mm   ,nn+n0) +(gfz(ll)+(gfxyz(ll)-gfzx(ll)-gfyz(ll))*sz)
142      5   p  10v          end do
                 <<<    SIMD(VL: 4)
                 <<<    SOFTWARE PIPELINING
145      5   p  10v          do ll=nvxs-1,nvxe+1
146      5   p  10v            dfx(ll+l0,mm   ,nn+nv) = dfx(ll+l0,mm   ,nn+nv)          -(gfxyz(ll)-gfzx(ll))          *sx
147      5   p  10v            dfy(ll+lv,mm+m0,nn   ) = dfy(ll+lv,mm+m0,nn   )          -(gfxyz(ll)-gfxy(ll))          *sy
148      5   p  10v            dfz(ll   ,mm+mv,nn+n0) = dfz(ll   ,mm+mv,nn+n0)          -(gfxyz(ll)-gfyz(ll))          *sz
149      5   p  10v          end do
                 <<<    SIMD(VL: 4)
                 <<<    SOFTWARE PIPELINING
152      5   p  10v          do ll=nvxs-1,nvxe+1
153      5   p  10v            dfx(ll+l0,mm+mv,nn   ) = dfx(ll+l0,mm+mv,nn   )          -(gfxyz(ll)-gfxy(ll))          *sx
154      5   p  10v            dfy(ll   ,mm+m0,nn+nv) = dfy(ll   ,mm+m0,nn+nv)          -(gfxyz(ll)-gfyz(ll))          *sy
155      5   p  10v            dfz(ll+lv,mm   ,nn+n0) = dfz(ll+lv,mm   ,nn+n0)          -(gfxyz(ll)-gfzx(ll))          *sz
156      5   p  10v          end do
                 <<<    SIMD(VL: 4)
                 <<<    SOFTWARE PIPELINING
159      5   p  10v         do ll=nvxs-1,nvxe+1
160      5   p  10v           dfx(ll+l0,mm+mv,nn+nv) = dfx(ll+l0,mm+mv,nn+nv)         + gfxyz(ll)                  *sx
161      5   p  10v           dfy(ll+lv,mm+m0,nn+nv) = dfy(ll+lv,mm+m0,nn+nv)         + gfxyz(ll)                  *sy
162      5   p  10v           dfz(ll+lv,mm+mv,nn+n0) = dfz(ll+lv,mm+mv,nn+n0)         + gfxyz(ll)                  *sz
163      5   p  10v         end do
```

コンパイラによる最適化※loop A

コンパイラによる最適化※loop B

コンパイラによる最適化※loop C

コンパイラによる最適化※loop D

コンパイラによる最適化※loop E

ループ間のデータ受け渡しのために新たな1次元配列を使用

プログラム3(ループ分割)性能分析

● プロファイラ ループコスト：プルグラム３

```
*******************************************************************************
   Application - loops
*******************************************************************************
          Cost        %    Nest    Kind     Exec    Start      End
-------------------------------------------------------------------------------
         77742  100.0000      --      --      --       --       --  Application
-------------------------------------------------------------------------------
          4730   6.0842       5  DO      OpenMP      258      265  velocity.velocity_b_5d._OMP_2_
          3601   4.6320       5  DO      OpenMP      119      126  velocity.velocity_e_5d._OMP_1_
          3582   4.6075       5  DO      OpenMP      439      446  velocity.velocity_b_5d._OMP_2_
          3432   4.4146       5  DO      OpenMP      293      300  velocity.velocity_b_5d._OMP_2_
          3391   4.3619       5  DO      OpenMP      157      159  position.position_5d._OMP_1_
          3356   4.3168       5  DO      OpenMP       97      104  velocity.velocity_e_5d._OMP_1_
          3237   4.1638       5  DO      OpenMP      130      136  velocity.velocity_e_5d._OMP_1_
          3227   4.1509       5  DO      OpenMP      403      410  velocity.velocity_b_5d._OMP_2_
          3091   3.9760       5  DO      OpenMP      108      115  velocity.velocity_e_5d._OMP_1_
          2849   3.6647       5  DO      OpenMP      139      143  velocity.velocity_e_5d._OMP_1_
```

velocity_eは1タイムステップあたり2回呼ばれる

97,108,119,
258,293,439,403：
関数pic5呼出

157：転置
130：loop A
139：loop B

プログラム	As-is (2)	ループ分割 (3)	比率
position	1.29	1.03	0.80
velocity_e	3.19	2.48	0.78

mpi_wtimeで計測した1タイムステップあたりの平均経過時間
velocity_bは変更なし

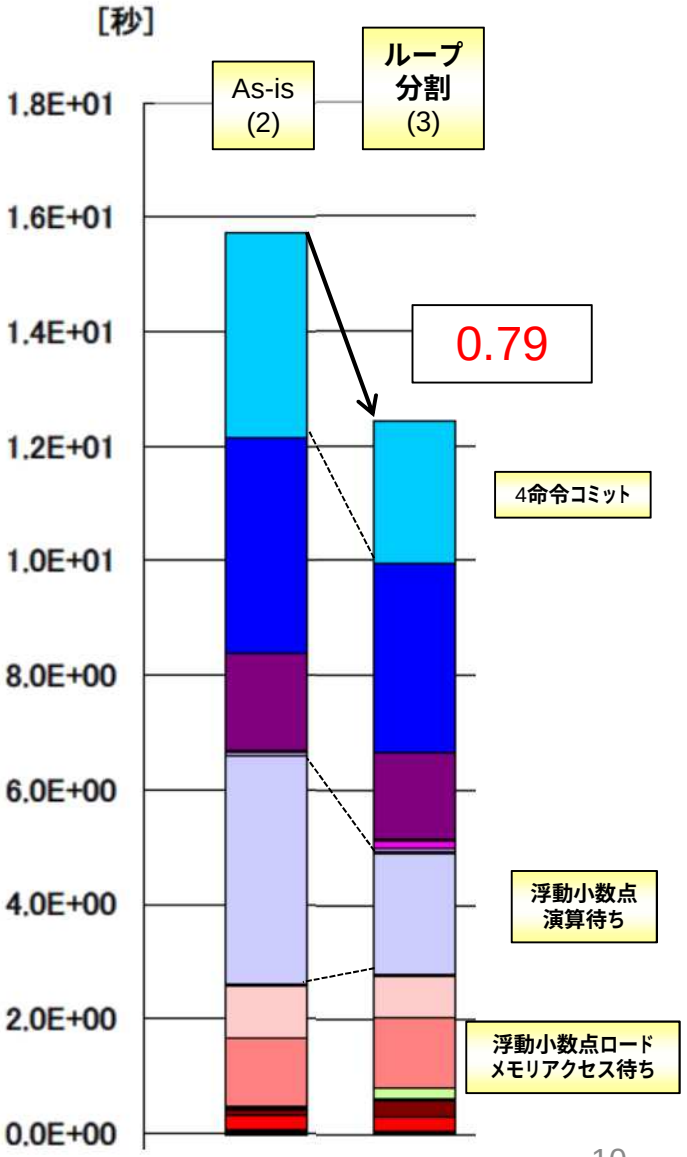
分析区間velocity_e：プログラムの比較

PAの実行時間とPA情報で比較

velocity_e	プログラム	スレッド数	実行時間 (sec)	比率
FX100 (PA実行時間)	As-is (2)	16	15.68	1.00
	ループ分割 (3)		12.34	0.79

- 最も負荷が高いループは、配列アクセスの依存関係で最適化がなされなかった。
- 配列アクセスの依存性を無くすループ分割により最適化が促進され、約20%の計算時間の短縮に成功
- SIMD以外の最適化がかかっているループ(loop A, B)のコストが少し目立つ。
- Loop B-Eは配列インデックスの計算がある

⇒!OCL NOSWPを指定したほうがよい？？



128reg/32regコンパイラ比較、FX100の測定条件

■測定条件

項目	条件
コンパイラ	FX100用128regコンパイラVer.2.0.0 / ポストFX100向けチューニング用32regコンパイラ
翻訳時 オプション	-Kfast,ocl,simd=2 -x250
パラメータ	mpi_x = 1, mpi_y = 1, ns =2 nx = 128, ny = 64 nvx = 40, nvy = 40, nvz = 40 (実行サイズ約28GB)
スレッド数	16スレッド
実行条件	1ノード – 2プロセス
プログラム	ループ分割版(プログラム3)

主要3区間の実行時間

- **評価対象区間は、**velocity_e、velocity_b、position **の計3つのサブルーチン**

レジスタ数 スレッド数	128reg 16t (秒)	32reg 16t (秒)	比率
position	5.40	6.27	1.16
velocity_e	12.39	17.97※	1.45
velocity_b	11.67	18.98	1.62
計	29.46	43.02	1.46

velocity_eは1タイムステップあたり2回呼ばれる

PAで計測した5タイムステップの経過時間

※関数pic5呼出ループにおいて、!OCL UNROLLを外した場合
(外さない場合は86.31秒：約7倍)

主要ループコスト

● プロファイラ ループコスト 128reg16t（velocity_eに!OCL UNROLLあり）

```
*******************************************************************************
    Application - loops
*******************************************************************************
          Cost          %    Nest    Kind     Exec    Start      End
-------------------------------------------------------------------------------
         90857   100.0000      --      --       --       --       --   Application
-------------------------------------------------------------------------------
          6613     7.2785       5   DO       OpenMP      115      122   velocity.velocity_e_5d._OMP_1_   *loop3
          6212     6.8371       5   DO       OpenMP      104      111   velocity.velocity_e_5d._OMP_1_   *loop2
          6076     6.6874       5   DO       OpenMP       93      100   velocity.velocity_e_5d._OMP_1_   *loop1
          4341     4.7778       5   DO       OpenMP      252      259   velocity.velocity_b_5d._OMP_2_   *loop4
          3897     4.2892       5   DO       OpenMP      433      440   velocity.velocity_b_5d._OMP_2_   *loop9
          3642     4.0085       5   DO       OpenMP      134      138   velocity.velocity_e_5d._OMP_1_
          3423     3.7675       5   DO       OpenMP      160      162   position.position_5d._OMP_1_
          3269     3.5980       5   DO       OpenMP      287      294   velocity.velocity_b_5d._OMP_2_   *loop5
          3145     3.4615       5   DO       OpenMP      148      152   velocity.velocity_e_5d._OMP_1_
          3091     3.4020       5   DO       OpenMP      141      145   velocity.velocity_e_5d._OMP_1_
```

velocity_eは1タイムステップあたり2回呼ばれる

● プロファイラ ループコスト 32reg16t（velocity_eに!OCL UNROLLなし）

```
*******************************************************************************
    Application - loops
*******************************************************************************
          Cost          %    Nest    Kind     Exec    Start      End
-------------------------------------------------------------------------------
        115508   100.0000      --      --       --       --       --   Application
-------------------------------------------------------------------------------
          8260     7.1510       5   DO       OpenMP      115      122   velocity.velocity_e_5d._OMP_1_   *loop3
          7297     6.3173       5   DO       OpenMP       93      100   velocity.velocity_e_5d._OMP_1_   *loop1
          7249     6.2758       5   DO       OpenMP      104      111   velocity.velocity_e_5d._OMP_1_   *loop2
          5921     5.1261       5   DO       OpenMP      252      259   velocity.velocity_b_5d._OMP_2_   *loop4
          4507     3.9019       5   DO       OpenMP      397      404   velocity.velocity_b_5d._OMP_2_   *loop8
          4460     3.8612       5   DO       OpenMP      360      367   velocity.velocity_b_5d._OMP_2_   *loop7
          4413     3.8205       5   DO       OpenMP      287      294   velocity.velocity_b_5d._OMP_2_   *loop5
          4328     3.7469       5   DO       OpenMP      433      440   velocity.velocity_b_5d._OMP_2_   *loop9
          4128     3.5738       5   DO       OpenMP      323      330   velocity.velocity_b_5d._OMP_2_   *loop6
          3118     2.6994       5   DO       OpenMP      160      162   position.position_5d._OMP_1_
```

loop4とloop9
loop5とloop8
loop6とloop7
は同じ構造

高コストループ

loop4とloop9
loop5とloop8
loop6とloop7
は同じ構造

```
velocity_e

89      3   p                 do nn=nvzs-1,nvze+1
90      4   p                   do mm=nvys-1,nvye+1
                                  *loop1 ⇒ ループ長：42
93      5   p    v              do ll=nvxs-1,nvxe+1
94      5   p    v                hm2=ff(ll-lv-lv,mm,nn,ii,jj)
95      5   p    v                hm1=ff(ll-lv   ,mm,nn,ii,jj)
96      5   p    v                hp0=ff(ll      ,mm,nn,ii,jj)
97      5   p    v                hp1=ff(ll+lv   ,mm,nn,ii,jj)
98      5   p    v                hp2=ff(ll+lv+lv,mm,nn,ii,jj)
99      5   pi   v                gfx(ll)=pic5(hp2,hp1,hp0,hm1,hm2,fex)
100     5   p    v              end do
                                  *loop2 ⇒ ループ長：42
104     5   p    v              do ll=nvxs-1,nvxe+1
105     5   p    v                hm2=ff(ll,mm-mv-mv,nn,ii,jj)
106     5   p    v                hm1=ff(ll,mm-mv   ,nn,ii,jj)
107     5   p    v                hp0=ff(ll,mm      ,nn,ii,jj)
108     5   p    v                hp1=ff(ll,mm+mv   ,nn,ii,jj)
109     5   p    v                hp2=ff(ll,mm+mv+mv,nn,ii,jj)
110     5   pi   v                gfy(ll)=pic5(hp2,hp1,hp0,hm1,hm2,fey)
111     5   p    v              end do
                                  *loop3 ⇒ ループ長：42
115     5   p    v              do ll=nvxs-1,nvxe+1
116     5   p    v                hm2=ff(ll,mm,nn-nv-nv,ii,jj)
117     5   p    v                hm1=ff(ll,mm,nn-nv   ,ii,jj)
118     5   p    v                hp0=ff(ll,mm,nn      ,ii,jj)
119     5   p    v                hp1=ff(ll,mm,nn+nv   ,ii,jj)
120     5   p    v                hp2=ff(ll,mm,nn+nv+nv,ii,jj)
121     5   pi   v                gfz(ll)=pic5(hp2,hp1,hp0,hm1,hm2,fez)
122     5   p    v              end do
```

関数pic5のインライン展開

関数pic5を呼び出すループが高コスト
形状はほぼ同じ。ループ長やメモリへ
のアクセスが異なる。

```
velocity_b

241     3   p           do mm=nvys,nvye
242     4   p             do nn=nvzs-1,nvze
                            *loop4 ⇒ ループ長：40 oclによる手動unrolling
252     5   p  10v         do ll=nvxs,nvxe
253     5   p  10v           hm2=ff(ll,mm,nn0(ll)-nnz(ll)-nnz(ll),ii,jj)
254     5   p  10v           hm1=ff(ll,mm,nn0(ll)-nnz(ll)        ,ii,jj)
255     5   p  10v           hp0=ff(ll,mm,nn0(ll)                ,ii,jj)
256     5   p  10v           hp1=ff(ll,mm,nn0(ll)+nnz(ll)        ,ii,jj)
257     5   p  10v           hp2=ff(ll,mm,nn0(ll)+nnz(ll)+nnz(ll),ii,jj)
258     5   pi 10v           dfz1(ll,nn)=pic5(hp2,hp1,hp0,hm1,hm2,vvz(ll))
259     5   p  10v         end do
260     4   p             end do

276     3   p           do nn=nvzs,nvze
277     4   p             do mm=nvys-1,nvye
                            *loop5 ⇒ ループ長：40 oclによる手動unrolling
287     5   p  10v         do ll=nvxs,nvxe
288     5   p  10v           hm2=ff(ll,mm0(ll)-mmy(ll)-mmy(ll),nn,ii,jj)
289     5   p  10v           hm1=ff(ll,mm0(ll)-mmy(ll)        ,nn,ii,jj)
290     5   p  10v           hp0=ff(ll,mm0(ll)                ,nn,ii,jj)
291     5   p  10v           hp1=ff(ll,mm0(ll)+mmy(ll)        ,nn,ii,jj)
292     5   p  10v           hp2=ff(ll,mm0(ll)+mmy(ll)+mmy(ll),nn,ii,jj)
293     5   pi 10v           dfy1(ll,mm)=pic5(hp2,hp1,hp0,hm1,hm2,vvy(ll))
294     5   p  10v         end do
295     4   p             end do

310     3   p           do nn=nvzs,nvze
311     4   p             do mm=nvys,nvye
                            *loop6 ⇒ ループ長：40 oclによる手動unrolling
323     5   p  10v         do ll=nvxs-1,nvxe
324     5   p  10v           hm2=ff(llO(ll)-llx(ll)-llx(ll),mm,nn,ii,jj)
325     5   p  10v           hm1=ff(llO(ll)-llx(ll)        ,mm,nn,ii,jj)
326     5   p  10v           hp0=ff(llO(ll)                ,mm,nn,ii,jj)
327     5   p  10v           hp1=ff(llO(ll)+llx(ll)        ,mm,nn,ii,jj)
328     5   p  10v           hp2=ff(llO(ll)+llx(ll)+llx(ll),mm,nn,ii,jj)
329     5   pi 10v           dfx1(ll,mm)=pic5(hp2,hp1,hp0,hm1,hm2,vvx(ll))
330     5   p  10v         end do
331     4   p             end do
```

関数pic5のインライン展開

主要行コスト

● プロファイラ 行コスト 128reg16t

```
*******************************************************************************
Application - lines
*******************************************************************************
          Cost          %      Line
-------------------------------------
         90857   100.0000        --   Application
-------------------------------------
          5491     6.0436       121   velocity.velocity_e_5d._OMP_1_
          5311     5.8454        99   velocity.velocity_e_5d._OMP_1_
          5266     5.7959       110   velocity.velocity_e_5d._OMP_1_
          2471     2.7197       477   velocity.velocity_b_5d._OMP_2_
          2443     2.6888       477   velocity.velocity_e_5d._OMP_1_
          2275     2.5039       160   position.position_5d._OMP_1_
          2260     2.4874       475   velocity.velocity_b_5d._OMP_2_
          2091     2.3014       276   position.position_5d._OMP_3_
          1916     2.1088       166   velocity.velocity_e_5d._OMP_1_
          1420     1.5629       475   velocity.velocity_e_5d._OMP_1_
```

● プロファイラ 行コスト 32reg16t

```
*******************************************************************************
Application - lines
*******************************************************************************
Cost          %      Line
-------------------------------------
        115508   100.0000        --   Application
-------------------------------------
          6605     5.7182       121   velocity.velocity_e_5d._OMP_1_
          6269     5.4273       110   velocity.velocity_e_5d._OMP_1_
          6230     5.3936        99   velocity.velocity_e_5d._OMP_1_
          4257     3.6855       477   velocity.velocity_b_5d._OMP_2_
          3280     2.8396       475   velocity.velocity_b_5d._OMP_2_
          2890     2.5020       329   velocity.velocity_b_5d._OMP_2_
          2830     2.4500       366   velocity.velocity_b_5d._OMP_2_
          2725     2.3591       403   velocity.velocity_b_5d._OMP_2_
          2692     2.3306       477   velocity.velocity_e_5d._OMP_1_
          2632     2.2786       293   velocity.velocity_b_5d._OMP_2_
```

99,110,121:関数pic5呼出
(1タイムステップあたり2回)

477,475:関数pic5内
(velocity_bでは1タイプステップあたり6回、
velocity_eでは1タイプステップあたり3×2回)

293,329,366,403:関数pic5呼出
(loop5,6,7,8)

高コスト行(関数pic5内)

関数pic5内

```
470         hmax1=max(max(hm1,hp0),min(hm1*2.0d0-hm2,hp0*2.0d0-hp1))
471         hmin1=min(min(hm1,hp0),max(hm1*2.0d0-hm2,hp0*2.0d0-hp1))
472         hmax2=max(max(hp1,hp0),min(hp0*2.0d0-hm1,hp1*2.0d0-hp2))
473         hmin2=min(min(hp1,hp0),max(hp0*2.0d0-hm1,hp1*2.0d0-hp2))
474
475         hmax=max(hmax1,hmax2)
476
477         hmin=max(gmin,min(hmin1,hmin2))
```

- **問題点、疑問点など**
  - loop4とloop9のループコストが目立つが、128reg,32regコンパイラで共に、行コストには表れない。
  - loop4とloop9は同じ形状であるにもかかわらず、loop4のコストが高い。

- 関数pic5内で、470-473行目のほうが計算が重そうにもかかわらず、477,475行目のコストが128reg,32regコンパイラで共に目立つ。
- 32regコンパイラにおいて、477行目の性能劣化が顕著。

⇒ループ内で関数を呼び出さずに、関数内にループ計算するように構成変更すれば、コンパイラの最適化が促進？

分析区間velocity_b：128reg と 32regコンパイラ の比較

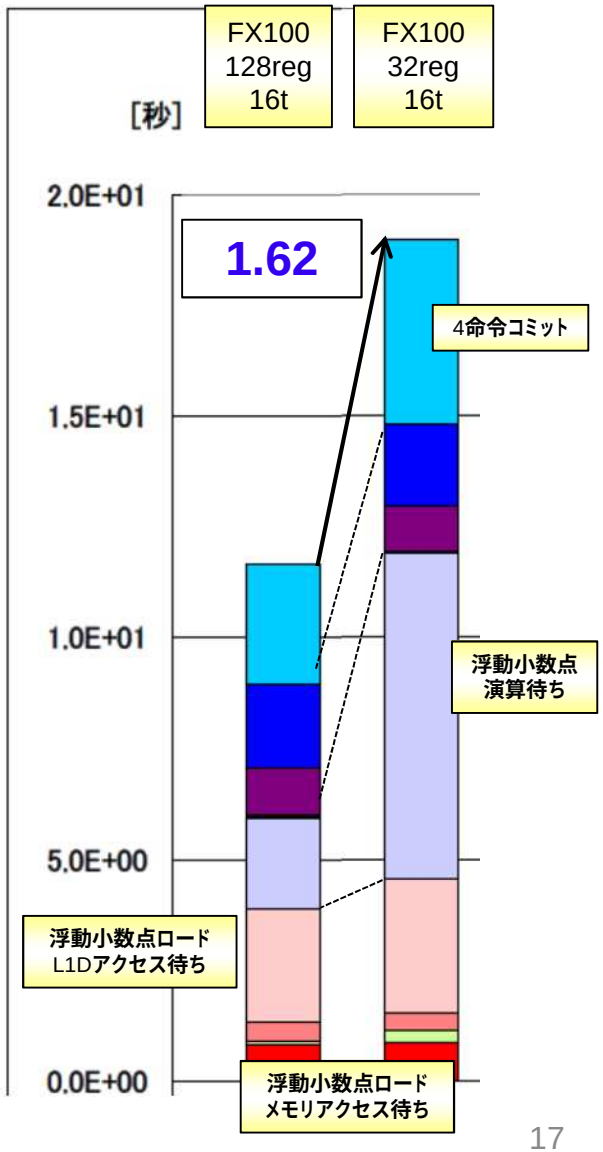
PAの実行時間とPA情報で比較

- 128reg と 32regコンパイラの性能差は約57%で、レジスタ数減少の影響は大きい。

velocity_b	レジスタ数	スレッド数	実行時間 (sec)	比率
FX100 (PA実行時間)	128reg	16	11.67	1.00
	32reg		18.98	1.62

- レジスタ数減少の影響で、ループはアンローリングされているが、レジスタが少ないため、スピルが発生し、演算レイテンシ（浮動小数点演算待ち）が増加。

⇒ループアンローリング数の調整による性能改善の可能性？？



ポストFX100、主要3区間の実行時間推定

- **評価対象区間は、**velocity_e、velocity_b、position **の計3つのサブルーチン**

	FX100 PA実行 (a)	FX100 PA実行 (b)	ポスト FX100 推定(c)	
レジスタ数 スレッド数	128reg 16t (秒)	32reg 16t (秒)	32reg **12t** (秒)	比率 c/b
position	5.40	6.27	7.43	1.185
velocity_e	12.39	17.97※	23.31※	1.297
velocity_b	11.67	18.98	24.42	1.287
計	29.46	43.02	55.16	1.282

- **ポストFX100推定値は、**32reg**実行結果に対して約**1.28**倍 ⇒ 24スレッドなら約0.64倍**
- **演算ネックの区間は、実際の性能より低めの推定値が出る傾向にある**

※関数pic5**呼出ループにおいて、**!OCL UNROLL**を外した場合**

32regチューニングのイメージ

As-is

```
do nn=nvzs-1,nvze+1
  do mm=nvys-1,nvye+1
    do ll=nvxs-1,nvxe+1
      hm2=ff(ll-lv-lv,mm,nn,ii,jj)
      hm1=ff(ll-lv   ,mm,nn,ii,jj)
      hp0=ff(ll      ,mm,nn,ii,jj)
      hp1=ff(ll+lv   ,mm,nn,ii,jj)
      hp2=ff(ll+lv+lv,mm,nn,ii,jj)
      gfx(ll)=pic5(hp2,hp1,hp0,hm1,hm2,fex)
    end do
:
:
:
function pic5(hp2,hp1,hp0,hm1,hm2,vv)
 real(kind=8), intent(in) :: hp2,hp1,hp0,hm1,hm2,vv
 real(kind=8), intent(out):: pic5

:
:
  pic5 =
end function
```

ループ分割により、velocity_bの実行時間の推定値が14.06秒、性能比c/bが0.74・c/aが1.2となった実績あり

関数のベクトル化＋ループ分割

```
do nn=nvzs-1,nvze+1
  do mm=nvys-1,nvye+1
    do ll=nvxs-1,nvxe+1
      hm2(ll)=ff(ll-lv-lv,mm,nn,ii,jj)
      hm1(ll)=ff(ll-lv   ,mm,nn,ii,jj)
      hp0(ll)=ff(ll      ,mm,nn,ii,jj)
      hp1(ll)=ff(ll+lv   ,mm,nn,ii,jj)
      hp2(ll)=ff(ll+lv+lv,mm,nn,ii,jj)
      ffx(ll)=fex
    end do
    call pic(hp2,hp1,hp0,hm1,hm2,ffx,nvxe-nvxs+3,gfx)
:
:
:
subroutine pic5(hp2,hp1,hp0,hm1,hm2,vv,nv,gf)
  integer(kind=4) :: intent(in) :: nv
  real(kind=8), intent(in) :: hp2(nv),hp1(nv),hp0(nv),hm1(nv),hm2(nv),vv(nv)
  real(kind=8), intent(out):: gf(nv)

  do ll=1,nv
:
  end do
  do ll=1,nv
:
  end do
  do ll=1,nv
    gf(ll) =
  end do
end subroutine
```

ループ分割による性能チューニングは今後の課題

※ループ分割に伴う作業配列の増加により性能が低下する可能性もある

4.3 プラズマ流体解析コード GT5D の Post-FX100 性能推定

日本原子力研究開発機構　井戸村 泰宏、伊奈 拓也
富士通株式会社　内藤 俊也、三吉 郁夫

4.3.1 はじめに

核融合プラズマ流体解析コード GT5D（スライド p2、以降、ページ番号のみ表記する）は磁場閉じ込め型の核融合炉における炉心プラズマの閉じ込め性能を決定する乱流輸送現象の評価を目的として開発された。核融合プラズマの第一原理モデルは 5 次元位相空間（3 次元空間×2 次元速度）におけるプラズマ粒子分布の発展を記述する移流・拡散方程式（ボルツマン方程式）で与えられる（p3）。核融合プラズマ流体解析ではこの方程式を 5 次元格子上で離散化して CFD スキームを適用するため、3 次元流体モデルに比べて～100^2倍程度大きい自由度の流体計算となる。「京」を用いてようやく ITER 規模の炉心プラズマの数値実験を断熱的電子モデルのイオン系乱流に対して行うことが可能になったが、運動論的電子モデルを含む電子系乱流の数値実験ではイオンに比べて熱速度が数十倍となる電子の運動を計算する必要があり、計算コストがさらに数十倍になるため、ポスト京のようなエクサスケールマシンが必要となる。しかしながら、現行機の SPARC64XIfx（FX100）から将来の A64fx（ポスト京、あるいは、Post-FX100）への移行では、以下に示すような技術的なギャップが存在する。

- SPARC から ARM へのアーキテクチャ変更
- レジスタ数の 128 から 32 への削減
- Out-of-Order 資源の増強
- SIMD 幅の 256bit から 512bit への増加
- プロセッサ（CMG）あたり演算コア数の 32（16）から 48（12）への変更
- メモリ機構の HMC（240+240GB/s）から HCM2（1024GB/s）への変更

このうち、本性能推定ではユーザプログラムの実装方法への影響が大きいと思われる、レジスタ数の削減、および、メモリ機構の変更に関して、FX100 上で用意された 32 レジスタ版富士通コンパイラ（32reg コンパイラ）と Post-FX100 性能推定ツールを用いて GT5D の主要カーネルであるクリロフソルバの性能推定を実施した（p1）。

4.3.2 プログラム概要

GT5D は炉心プラズマ全体を計算領域としてプラズマ乱流やプラズマ分布の大域的な非線形発展を追跡する。この特徴を活かして、これまで非局所的な乱流輸送現象の研究や乱流輸送の装置サイズスケーリングの研究が行われてきた。計算手法としては 5 次元移流・拡散方程式に差分法を適用し、時間積分に半陰解法を用いている（p3）。移流項を陰的に計算する差分・陰解法カーネルは共役残差（GCR）法、あるいは、省通信一般化最小残差（CA-GMRES）法によって実装されており（p5）、コードの主要コストを占める（p4）。並列化手法としては MPI と OpenMP を用いるハイブリッド並列モデルを採用している。さらに、OpenM

P によって実装した通信スレッドによる通信隠蔽処理や階層的な多次元領域分割モデルの採用により Oakforest-PACS 全系に至る強スケーリングを達成し、約 1500 億格子を用いる ITER 規模の数値実験が可能になった（p2）。

4.3.3　128/32 レジスタ版コンパイラを用いた性能測定

p4 に 32reg/128reg コンパイラを用いて測定した GT5D のコスト分布を示す。ここで、問題サイズは$(N_x, N_y, N_z, N_v, N_w)=(320, 320, 32, 96, 20)=6x10^9$とし、MPI 並列数は $(n_x, n_y, n_w) =(4, 4, 20)=320$（160node x 2process x 16threads）とした。GT5D は移流項の 4 次元差分計算、衝突項の 2 次元差分計算、ポアソン方程式の 2 次元有限要素法、1 次元 FFT 等、異なる数値計算法のカーネルを含むが、多くの計算カーネルでは 32reg コンパイラによる性能劣化は見られなかった。一方、全体コストの 8 割以上を占めるクリロフソルバ、および、移流項の差分計算カーネルでは大きな性能劣化が見られた。ここで、クリロフソルバの SpMV は移流項の差分計算（非対称行列、17 点ステンシル）で与えられることから、同様の性能劣化が発生しているものと推測される。以下では、さらにクリロフソルバ内部のコスト分布を詳細に議論する。

p5 に GT5D で実装されている GCR 法と CA-GMRES 法の演算特性を示す。GCR 法は SpMV カーネルと 2 つの DAXPY カーネルで構成され、演算密度 f/b=0.37 の低演算密度のソルバとなっている。一方、CA-GMRES 法の主要カーネルは SpMV と密行列演算カーネル（DSYRK、DGEMV）から構成され、f/b=1.38 の高演算密度のソルバとなっている。FX100 におけるカーネル性能評価では、DAXPY の実効性能が高い GCR 法の実効性能がルーフラインモデルの 85%となるのに対し、CA-GMRES 法の実効性能は 61%となる。しかしながら、処理時間で見ると CA-GMRES 法は GCR 法の約半分となっている。ここで、ルーフラインモデルは各環境における stream triad 性能を用いた。

p6 に GCR 法、CA-GMRES 法の各カーネルに対する 32reg/128reg コンパイラ比較を示す。ここで、問題サイズは$(N_x, N_y, N_z, N_v)=(160, 160, 32, 96)$とし、MPI 並列数は $(n_x, n_y) =(2, 2)=4$（2node x 2process x 16threads）とした。GCR 法においては、２つの DAXPY カーネルの 32reg コンパイラによる性能劣化は見られないが、SpMV カーネルの処理時間は 32reg コンパイラで約 2 倍に増大する。一方、CA-GMRES 法においては、密行列演算カーネルの性能劣化は少ないが、GCR 法と共通の SpMV カーネルについては処理時間が 32reg コンパイラで約 2 倍に増大する。ここで、GCR 法と CA-GMRES 法の SpMV カーネルの処理時間の違いは、GCR 法の SpMV カーネルのみに含まれているベクトル内積演算処理によるものである。

p7 に SpMV カーネルのサンプルコードを示す。この差分計算は 4 次元移流項の 4 次精度中心差分（森西スキーム）で与えられるため、4 方向 5 点差分で合計 17 ステンシルを参照する。ここで、ステンシル計算で参照するデータアクセスパターンはストライドアクセスとなっており、12MB の L2 キャッシュに対して最内

3 重ループのインデックス、l、k、j まではオンキャッシュとなり、さらに、最外の i についても OpenMP による dynamic ループ分割によってスレッド間でキャッシュ上のデータ再利用を図ることによりオンキャッシュとなるため、L2 キャッシュの参照が主要なコストを占める[サイエンティフィックシステム研究会､ポストペタアプリ性能評価 WG 成果報告書 2.8]。また、このカーネルはループボディが比較的大きく、レジスタ数の削減が処理性能に影響していると考えられる。この問題を回避するために、ユーザ側の最適化コードでは 4 次元差分の 4 重ループの最内の 1 方向ループを 2 分割してループあたりのレジスタ数を削減した。ここで、分割点は i,j 方向と k,l 方向、各 8 ステンシルずつに分割した。このループ分割により変数 df へのロード/ストアが 1 回ずつ増えるが、最内ループのインデックス l は 4 次元配列の最内、すなわち、連続アクセスなので、増大するロード/ストアはキャッシュ上で処理され、そのペナルティは小さいと考えられる。p6 に最適化後の SpMV カーネルの処理性能を示すが、処理時間の増大は 1 割以下に抑制されており、32reg/128reg コンパイラでほぼ同等の性能が得られることがわかった。この修正カーネルを用いて 32 レジスタの Intel 環境でも性能比較を行ったが、Intel 環境では最適化前後で性能に違いが見られなかった。これは､Intel コンパイラではこのような最適化が自動で行われていることを示唆している。

4.3.4　SpMV カーネルの詳細分析、最適化

p9 では、性能劣化が見られた SpMV カーネルについて詳細 PA を用いた分析を実施した。上記の事例と同様に 32reg コンパイラで性能劣化が見られるが、この例では、後で議論する Post-FX100 推定用に 12 スレッドでも割り切れるように問題サイズを(N_x,N_y,N_z,N_v)=(48,48,16,128)の逐次処理(16thread､もしくは、12thread) としたこと、および、富士通側でコンパイラオプションを調整したことの影響により、32reg/128reg 性能差は 16%程度に抑制されている。PA 情報から、128reg コンパイラでは SpMV カーネルの L2 スループットが大きいが、32reg コンパイラでは L2 スループットが約 15%低減し、ロード/ストア命令数が約 1 割増加している。これはレジスタスピルによる命令数の増加を示唆している。

この問題の回避に向けて、ループ分割による最適化を実施した。p10、11 に p7 に示す SpMV カーネルの最内 2 重ループを切り出したサンプルコードを示すが、富士通側の最適化では､分割点を i 方向と j,k,l 方向､それぞれ､4 ステンシル、12 ステンシルと非対称に分割している。また、ループ分割の次元に関しても最内 2 次元ループを対象とした。さらに、比較的高い L2 ミス dm 率が示唆するメモリレイテンシを隠蔽するために、ループ分割後の各 2 次元ループに対して、unroll 数を 2、4 と指定してソフトウェアプリフェッチを手動で指定した。この結果、p12 に示すように、128reg コンパイラとほど同等の処理時間が得られ、L2 スループットも回復した。

4.3.5　GCR ソルバの Post-FX100 性能推定

最適化後の GCR ソルバを用いて Post-FX100 性能推定を実施し、2 つの DAXPY カーネルと SpMV カーネルの処理性能を比較した。ここでは FX100、Post-FX100 それぞれの 1CMG を用いた処理性能を比較した。p1 に各環境のスペックを示すが、FX100 の 1CMG は 16core、0.5TFlops、メモリ帯域幅 120+120GB/s (in/out)、メモリ B/F=0.48、L2 キャッシュ B/F=2.0、L1 キャッシュ B/F=4.0、L2 キャッシュ 12MB となるのに対し、Post-FX100 の 1CMG は 12core、0.675TFlops、256GB/s、メモリ B/F=0.38、L2 キャッシュ B/F=1.3、L1 キャッシュ B/F=4.0、L2 キャッシュ 8MB となる。このため FX100/Post-FX100 理論性能比は演算性能で 1.35 倍、メモリバンド幅で 1.06 倍となる。性能推定ツールでは FX100 の PA 情報に基づいて Post-FX100 性能推定を行う。演算に関しては、SIMD 化されている命令については命令数を半分にカウントしている。メモリに関しては、メモリアーキテクチャの違いを反映してロード/ストアの処理コストや L1/L2 キャッシュアクセスのコストを換算しているがが、キャッシュサイズの違いは考慮していない。

p14、15 に DAXPY カーネルの性能を示す。DAXPY カーネルでは性能劣化が見られなかったことから性能評価は asis カーネルで実施した。2 つの DAXPY カーネルでほぼ同じ処理特性が得られており、どちらもメモリスループットがボトルネックとなっている。推定結果における Post-FX100 における 1CMG あたりの処理時間は FX100 の 0.75 倍となっている。一方、ルーフラインモデルで評価した理論性能比は DAXPY1 (f/b=0.08) で 0.93 倍、DAXPY2 (f/b=0.21) で 0.92 倍となる。ここで、ルーフラインモデルは上記理論性能を用いた。この推定結果は理論性能を上回る結果となっているが、これには Post-FX100 におけるメモリアーキテクチャの変更が大きく影響している。FX100 の HMC がロード/ストアそれぞれ 1:1 の設計となっており、ロード 2:ストア 1 となる stream triad の実効メモリバンド幅が 6 割程度にとどまっていた[サイエンティフィックシステム研究会、ポストペタアプリ性能評価 WG 成果報告書 2.1]のに対し、Post-FX100 の HBM2 では非対称なロード/ストアでもメモリバンド幅をフルに活用でき、stream triad の実効メモリバンド幅が約 8 割に向上している[http://pr.fujitsu.com/jp/news/2018/08/22-1.html]。

p16 に SpMV カーネルの性能を示す。カーネルの演算特性としては、FX100 と同様に L2 スループットがボトルネックとなっているが、SIMD 幅の拡大を反映して水色や青で示す命令コミットの処理時間が削減されている。推定結果における Post-FX100 における 1CMG あたりの処理時間は FX100 とほぼ同じ処理時間となっている。ルーフラインモデルに基づく理論性能比では SpMV (f/b=1.03) に対して 0.87 倍となることから、この推定結果は理論性能を下回る結果となっている。このカーネルは L2 ビジー時間が主要なコストを占めることから、実行性能の劣化は L2 キャッシュ帯域幅が FX100 の B/F=2.0 (1TB/s) から Post-FX100 の B/F=1.3 (877.5GB/s) に低下したことが影響していると考えられる。

4.3.6　まとめ

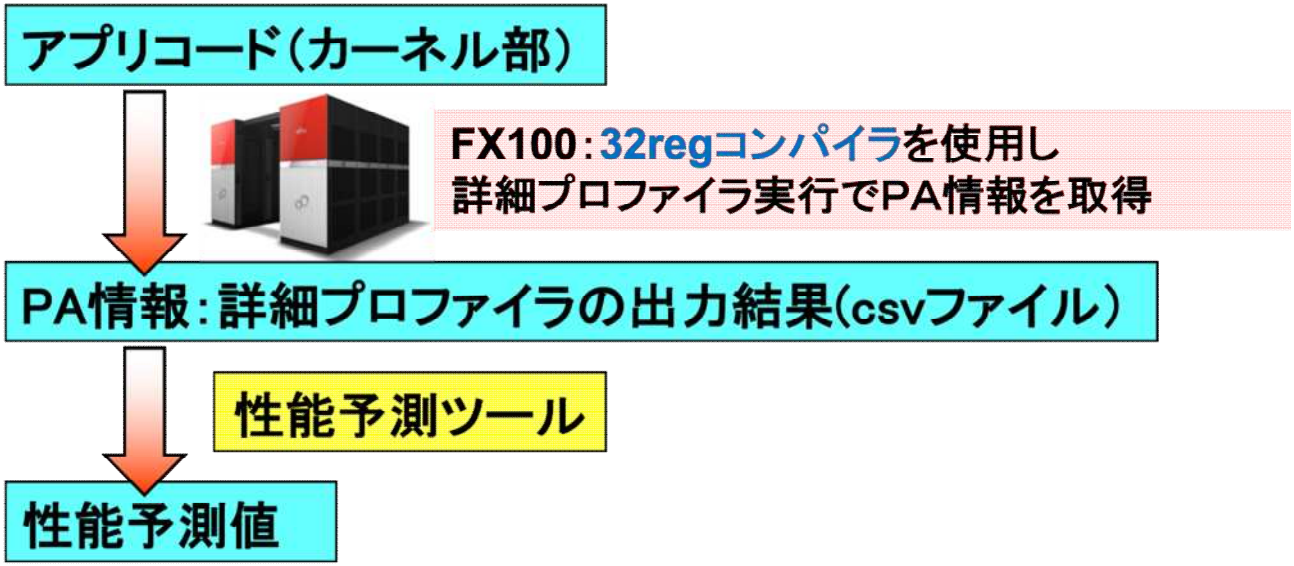
プラズマ流体解析コード GT5D を対象として Post-FX100 の性能推定を行った。GT5D コードは移流項の 4 次元差分計算、衝突項の 2 次元差分計算、ポアソン方程式の 2 次元有限要素法、1 次元 FFT 等、多彩な演算カーネルを含むコードであるが、4 次元差分計算以外は 32reg コンパイラによる性能劣化の影響は少ないことがわかった。これは、多くの既存アプリで大幅なコード修正なしに Post-FX100 の性能を引き出せる可能性を示唆している。実際に、GT5D 以外にも原子力機構の 3 次元多相多成分熱流動解析コードの評価を行ったところ、全てのカーネルで性能劣化が見られなかった。

4 次元差分計算の SpMV カーネルでは PA 情報からレジスタスピルによるものと推定される性能劣化が発生していたが、ループ分割による最適化を適用することで性能劣化を回避することに成功した。ただし、この実装に関してはユーザ側最適化と富士通側最適化でループ分割を行う次元や分割点に関して異なる実装が行われ、さらに、富士通側最適化では分割後のループに対する unroll、prefech 等の手動最適化が適用された。これらの最適化方法には任意性があるが、このような最適化の自動処理に向けた Post-FX100 コンパイラの開発が進展しており、今後、Post-FX100 の使い勝手が大幅に向上すると期待される。

Post-FX100 の性能推定では GCR ソルバの SpMV カーネルと 2 つの DAXPY カーネルの性能推定を実施した。DAXPY カーネルに関してはメモリアーキテクチャの変更により非対称なロード/ストア負荷のカーネルに対して性能向上が期待できることがわかった。一方、ほとんどのデータがオンキャッシュで処理され L2 スループットの負荷が大きい SpMV カーネルに関しては 1CMG あたりの処理時間が FX100 と Post-FX100 で同程度となり、ルーフラインモデルの理論性能予測を下回る推定結果となった。この性能劣化は L2 キャッシュ帯域幅の低下によるものと考えられる。

今回の性能推定は FX100 の PA 情報に基づく性能推定ツールを用いたため、Out-of-Order 機構や ARM-SVE の新たな命令セットの効果等を見ることができなかった。これらの課題については、今後、ポスト京シミュレータを用いてさらに精度の高い性能推定を行うことが必要である。

性能評価環境



測定環境:名大FX100+128reg/32reg**コンパイラ**

- SPARC64XIfx: 1TFlops, 240+240GB/s, 16x2core, 256bitSIMD, HMC
- **評価単位**CMG: 0.5TFlops, 240GB/s, 16core
- mpifrtpx -X9 -Kfast -Kparallel -Karray_private -Kpreex -Kopenmp -Kocl -Kmfunc=2 -Kprefetch_cache_level=all -CcdRR8 -Cpp -Koptmsg=2 -Y0,/center/fjlocal/opt/Simulation_compiler/bin -YI,/center/fjlocal/opt/Simulation_compiler/include

推定対象:ポストFX100

- A64FX: 2.7TFlops, 256x4GB/s, 12x4core, 512bitSIMD, HBM2
- **評価単位**CMG: 0.675TFlops, 256GB/s, 12core

プラズマ流体解析コードGT5Dの概要

- Gyrokinetic Toroidal 5D Eulerian code

  5次元ジャイロ運動論モデルによる核融合炉心プラズマの第一原理乱流計算を行い、乱流輸送特性を評価

- 計算手法

  MPI+OpenMPハイブリッド並列モデル

  5次元位相空間の移流・拡散→差分法

  時間積分→半陰的ルンゲ・クッタ法

  差分・陰解法カーネル

  →省通信クリロフ部分空間法+通信隠蔽

- 計算規模 (ITER規模)

  768x64x768x128x32 ~ 1500億格子

  ~10^5ステップ→約5Mnode-h/ショット

- 参考文献

  Idomura et al., Comput.Phys.Commun.2008

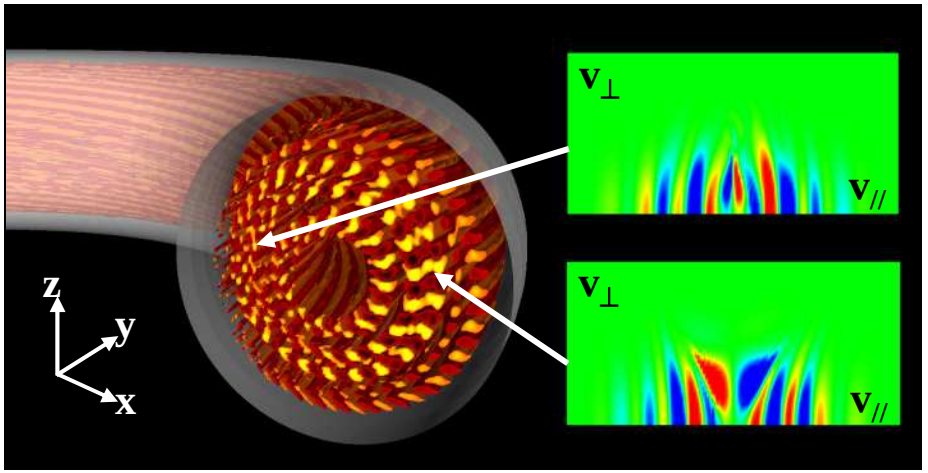
  Idomura et al., Nucl. Fusion 2009

  Idomura et al., Int. J. HPC Appl. 2014

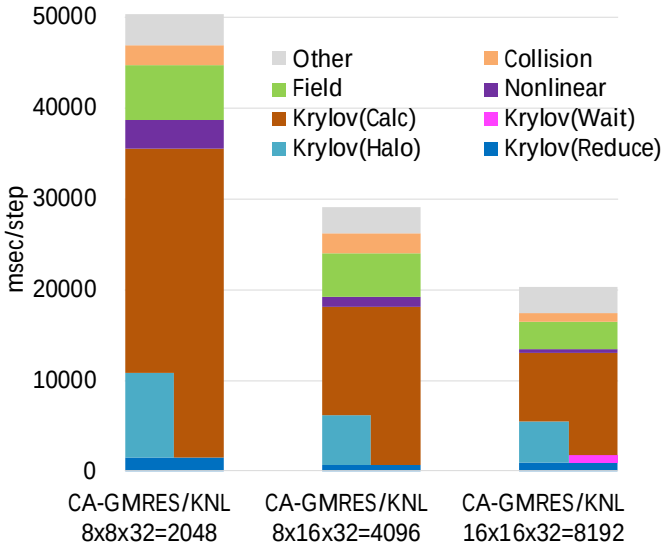
  Idomura, J. Comput. Phys. 2016

  Idomura et al., ScalA17@SC17

5次元位相空間 (速度2次元) の摂動粒子分布



Oakforest-PACSにおけるGT5D@ITER規模の強スケーリング



計算モデル

[Idomura,JCP07,CPC08,NF09,PFR14,JCP16]

- **プラズマ分布の移流･拡散方程式** $(R, \zeta, Z, v_{\parallel}, v_{\perp})$

$$\frac{\partial f}{\partial t} = \mathcal{G}[f] + \mathcal{C}[f]$$

$$\mathcal{G}[f] = -\mathbf{U}_1[\phi] \cdot \frac{\partial f}{\partial \mathbf{R}} - U_2[\phi] \frac{\partial f}{\partial v_{\parallel}}$$

熱運動や乱流場による移流

$$\mathcal{C}[f] = -\frac{\partial}{\partial \mathbf{v}} \cdot (\mathbf{D}_1 f) + \frac{\partial^2}{\partial \mathbf{v} \partial \mathbf{v}} : (\mathbf{D}_2 f)$$

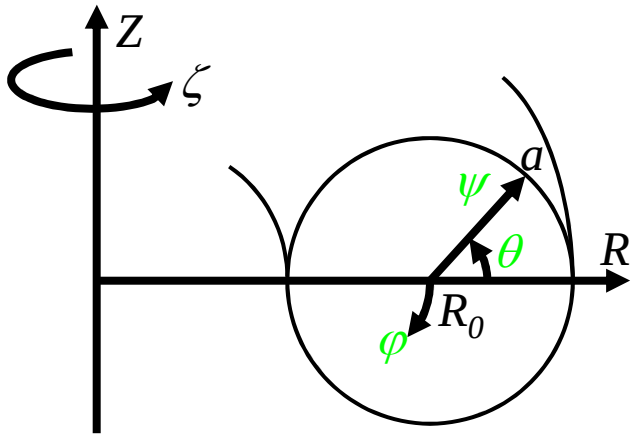
クーロン衝突効果

  - **中心差分** (4次精度森西) **+半陰的時間積分** (2次精度加法半陰的R-K)
  - **線形移流項の差分･陰解法の非対称行列に対するクリロフソルバ**

- **乱流場を決めるポアソン方程式** (ψ, θ, φ)

$$-\nabla_{\perp} \cdot (P_1 \nabla_{\perp} \phi) + P_2 (\phi - \langle \phi \rangle) = \rho[f]$$

  - **2次元有限要素法** (ψ, θ)
  - FFT (φ)



GT5Dの全体コスト比較

問題サイズ：$N_x x N_y x N_z x N_v x N_w = 320x320x32x96x20 = 6x10^9$

並列化：$n_x x n_y x n_w = 4x4x20 = 160$ (160node x 2process x 16threads)

ms/step	128reg	32reg	32reg/128reg
dn（**密度**）	20	22	1.09
field（**電場**）	247	247	1.00
drift（**移流速度**）	93	98	1.05
l4d_nl（**非線形移流**）	826	1131	1.37
l4d_l（**線形移流**）	466	806	1.73
rk（**時間積分**）	197	198	1.00
bc（**境界条件**）	922	1289	1.40
col（**粒子衝突**）	125	138	1.10
GCR（**クリロフソルバ**）	22966	29196	1.27
total	25862	33124	1.28

- **クリロフソルバが全体コストの8割以上を占める**
- **クリロフソルバ、および、移流項の差分計算カーネルで顕著な性能劣化**
- **他カーネルではあまり性能劣化は見られない**

GT5Dにおけるクリロフソルバの演算特性

[Idomura,ScalA17@SC17]

行列特性：**非対称行列**,17**点ステンシル** (5**点差分**×4**次元**)

問題サイズ：N_xxN_yxN_zxN_v=160x160x32x96

並列化：n_xxn_y=2x1 or 2x2 (2node x 2 or 1process)

Algorithm	Kernel	f/b	ICEX t	ICEX P	ICEX R_{RL}	ICEX R_{ICEX}	FX100 t	FX100 P	FX100 R_{RL}	FX100 R_{ICEX}	KNL t	KNL P	KNL R_{RL}	KNL R_{ICEX}
GCR	SpMV1	1.03	16.00	81.1	0.76	1.00	7.97	162.7	0.68	2.01	8.47	153.2	0.35	1.89
	DAXPY1	0.08	18.66	8.4	0.88	1.00	6.86	22.9	0.93	2.72	4.12	38.1	0.95	4.53
	DAXPY2	0.21	18.41	21.4	0.90	1.00	6.88	57.2	0.97	2.68	4.12	95.4	0.97	4.47
	TOTAL	0.37	53.07	34.8	0.85	1.00	21.71	85.1	0.85	2.44	16.72	110.5	0.65	3.17
CA-GMRES(22)	SpMV2	1.29	13.55	90.0	0.69	1.00	6.85	178.0	0.63	1.98	7.73	157.7	0.30	1.75
	DSYRK	3.00	3.73	264.2	1.03	1.00	3.26	303.1	0.63	1.15	1.80	547.7	0.55	2.07
	DGEMV	0.23	3.38	23.8	0.89	1.00	1.85	43.6	0.66	1.83	1.05	76.7	0.69	3.22
	TOTAL	1.38	22.19	105.8	0.77	1.00	13.12	179.0	0.61	1.69	11.35	207.0	0.37	1.96

f/b:**演算密度**, t:msec/iteration, P:GFLOPS, R_{RL}:Roofline**比**, R_{ICEX}:ICEX**比**

- **一般化共役残差法**GCR
  - **差分計算** (SpMV) +**ベクトル演算** (AXPY)
  - **低演算密度**
  - FX100**ではルーフライン理論性能の約**85%**の実効性能**
- **省通信一般化最小残差法**CA-GMRES
  - **差分計算** (SpMV) +**密行列演算** (SYRK) +**行列ベクトル積** (GEMV)
  - **高演算密度**
  - FX100**ではルーフライン理論性能の約**60%**の実効性能**

32reg/128regでのクリロフソルバ性能比較

GCR

msec/iteration	128reg	32reg	32reg/128reg	32reg(opt)	32reg/128reg(opt)
SpMV1	7.90	16.38	2.07	8.68	1.10
DAXPY1	7.22	7.22	1.00	7.22	1.00
DAXPY2	7.07	7.06	1.00	7.05	1.00
TOTAL	22.19	30.66	1.38	22.95	1.03

CA-GMRES

msec/iteration	128reg	32reg	32reg/128reg	32reg(opt)	32reg/128reg(opt)
SpMV2	6.55	13.21	2.02	6.80	1.04
DSYRK	3.37	3.95	1.17	3.95	1.17
DGEMV	1.93	1.89	0.98	1.92	1.00
TOTAL	11.84	19.05	1.61	12.67	1.07

- DAXPY、**密行列演算** (DSYRK,DGEMV) **では性能劣化はあまり見られない**
- SpMV (**差分計算**) **で大きい性能劣化が見られる**
  - **ループ分割によるループボディ調整で性能を改善**

→32reg**による性能劣化は**Out of Order**機能なしにほぼ解消？**

ユーザ側で実施した32reg向けSpMV最適化

オリジナル

```
real*8 f(-1:nv+2,-1:nz+2,-1:ny+2,-1:nx+2)
!$OMP DO COLAPSE(2) SCHEDULE(dynamic,1)
      do i = 1,nx    (nx=80)
      do j = 1,ny    (ny=80)
      compute vxyv~vxl2 from vx(l,j,i),vy(l,j,i),vz(l,j,i),vv(l,j,i)
      do k = 1,nz    (nz=32)
      do l = 1,nv    (nv=96)
        df(l,k,j,i)=
     &         vxyv(l)*f(l,k,j,i)
     &       + vvr(l) *f(l+1,k,j,i) - vvl(l) *f(l-1,k,j,i)
     &       - vvr2(l)*f(l+2,k,j,i) + vvl2(l)*f(l-2,k,j,i)
     &       +vzl2(l)*(f(l,k-2,j,i)-f(l,k+2,j,i))
     &       +vzl(l) *(f(l,k+1,j,i)-f(l,k-1,j,i))
     &       + vyr(l) *f(l,k,j+1,i) - vyl(l) *f(l,k,j-1,i)
     &       - vyr2(l)*f(l,k,j+2,i) + vyl2(l)*f(l,k,j-2,i)
     &       + vxr(l) *f(l,k,j,i+1)  - vxl(l) *f(l,k,j,i-1)
     &       - vxr2(l)*f(l,k,j,i+2) + vxl2(l)*f(l,k,j,i-2)
      enddo
      enddo
      enddo
      enddo
```

最適化

```
real*8 f(-1:nv+2,-1:nz+2,-1:ny+2,-1:nx+2)
!$OMP DO COLAPSE(2) SCHEDULE(dynamic,1)
      do i = 1,nx    (nx=80)
      do j = 1,ny    (ny=80)
      compute vxyv~vxl2 from vx(l,j,i),vy(l,j,i),vz(l,j,i),vv(l,j,i)
      do k = 1,nz    (nz=32)
      do l = 1,nv    (nv=96)
        df(l,k,j,i)=
     &         vxyv(l)*f(l,k,j,i)
     &       + vvr(l) *f(l+1,k,j,i) - vvl(l) *f(l-1,k,j,i)
     &       - vvr2(l)*f(l+2,k,j,i) + vvl2(l)*f(l-2,k,j,i)
     &       +vzl2(l)*(f(l,k-2,j,i)-f(l,k+2,j,i))
     &       +vzl(l) *(f(l,k+1,j,i)-f(l,k-1,j,i))
      enddo
      do l = 1,nv    (nv=96)
        df(l,k,j,i) = df(l,k,j,i)+
     &       + vyr(l) *f(l,k,j+1,i) - vyl(l) *f(l,k,j-1,i)
     &       - vyr2(l)*f(l,k,j+2,i) + vyl2(l)*f(l,k,j-2,i)
     &       + vxr(l) *f(l,k,j,i+1)  - vxl(l) *f(l,k,j,i-1)
     &       - vxr2(l)*f(l,k,j,i+2) + vxl2(l)*f(l,k,j,i-2)
      enddo
      enddo
      enddo
      enddo
```

この最適化はIntel環境 (32reg) の性能には影響しない

FUJITSU

SpMVカーネルの分析、最適化

■ GCRカーネル

行列特性：非対称行列,17点ステンシル（5点差分×4次元）

問題サイズ：N_xxN_yxN_zxN_v=48x48x16x128

（16と12で割り切れるようにN_xxN_y=48x48と調整）

コンパイラオプション：

(共通)　-xpre_diff -Kfast -Kopenmp -Kocl,optmsg=2 -V -X9 -Kmfunc=2 -Cpp -Kloop_nofusion
　　-CcdRR8 -Kswp_strong -Kprefetch_nostrong -Nlst=t,lst=i,lst=x -Kprefetch_sequential=soft

(DAXPY)　-Kprefetch_cache_level=all

(SpMV)　-Kprefetch_cache_level=1

128reg と 32regコンパイラ の比較

FUJITSU

PAの実行時間で比較

- 128reg と 32regコンパイラで、性能差が16%程度出ており、レジスタ数の影響は、やや大きい。

DIFF	コード	レジスタ数	スレッド数	実行時間(sec)	比率
FX100 (PA実行時間)	asis	128reg	16	0.37	1.00
		32reg		0.43	1.16

PA情報で比較

- 128regはL2ビジー率が高めである。
- 128reg,32reg共にL2デマンドミス率が高い。
- 32regは、 128regに比べてスピル･フィル命令数が増加し、命令コミットが増加している。
- 128reg、32reg共にSWPLが適用される。

DIFF	コード	レジスタ数	スレッド数	実行時間 (sec)	L1 ビジー率	L2 ビジー率	L2 スループット (GB/sec)	メモリ ビジー率	メモリ スループット (GB/sec)
FX100 (PA情報)	asis	128reg	16	0.37	69%	67%	263.09	69%	96.82
	asis	32reg		0.43	75%	57%	225.21	58%	80.64

DIFF	コード	レジスタ数	スレッド数	ロード･ ストア数	L1D ミス率	L1D ミス dm 率	L2 ミス率	L2 ミス dm 率
FX100 (PA情報)	asis	128reg	16	2.79E+10	1.35%	16.83%	0.39%	44.34%
	asis	32reg		3.08E+10	1.22%	14.52%	0.35%	50.10%

チューニング：ループ分割位置

FUJITSU

■ ループ分割位置について (2次元-2分割)

```
2020  3  p       do k = 1,nz          ⇒ nz は  16
2021  4  p  2v     do l = 1,nv         ⇒ nv は 128
2023  4  p  2v       df(l,k,j,i)=
2024  4          &          vxyv(l)*f(l,k,j,i)
2025  4          &        + vvr(l) *f(l+1,k,j,i)
2026  4          &        - vvl(l) *f(l-1,k,j,i)
2027  4          &        - vvr2(l)*f(l+2,k,j,i)
2028  4          &        + vvl2(l)*f(l-2,k,j,i)
2029  4          &        + vzl2(l)*(f(l,k-2,j,i)-f(l,k+2,j,i))
2030  4          &        + vzl(l) *(f(l,k+1,j,i)-f(l,k-1,j,i))
2031  4          &        + vyr(l) *f(l,k,j+1,i)
2032  4          &        - vyl(l) *f(l,k,j-1,i)
2033  4          &        - vyr2(l)*f(l,k,j+2,i)
2034  4          &        + vyl2(l)*f(l,k,j-2,i)
!-------------------------------------------------------------!
2035  4          &        + vxr(l) *f(l,k,j,i+1)
2036  4          &        - vxl(l) *f(l,k,j,i-1)
2037  4          &        - vxr2(l)*f(l,k,j,i+2)
2038  4          &        + vxl2(l)*f(l,k,j,i-2)
2040  4  p  2v       dfc_(l)=dfc_(l)+df(l,k,j,i)*fc(l,k,j,i)
2041  4  p  2v     enddo
2042  3  p       enddo
```

```
2020  3  p       do k = 1,nz          ⇒ nz は  16
2021  4  p  2v     do l = 1,nv         ⇒ nv は 128
2023  4  p  2v       df(l,k,j,i)=
2024  4          &          vxyv(l)*f(l,k,j,i)
2025  4          &        + vvr(l) *f(l+1,k,j,i)
2026  4          &        - vvl(l) *f(l-1,k,j,i)
2027  4          &        - vvr2(l)*f(l+2,k,j,i)
2028  4          &        + vvl2(l)*f(l-2,k,j,i)
2029  4          &        + vzl2(l)*(f(l,k-2,j,i)-f(l,k+2,j,i))
2030  4          &        + vzl(l) *(f(l,k+1,j,i)-f(l,k-1,j,i))
2031  4          &        + vyr(l) *f(l,k,j+1,i)
2032  4          &        - vyl(l) *f(l,k,j-1,i)
2033  4          &        - vyr2(l)*f(l,k,j+2,i)
2034  4          &        + vyl2(l)*f(l,k,j-2,i)
                   enddo
                 enddo
!-------------------------------------------------------------!
                 do k = 1,nz          ⇒ nz は  16
                   do l = 1,nv         ⇒ nv は 128
2035  4          &        + vxr(l) *f(l,k,j,i+1)
2036  4          &        - vxl(l) *f(l,k,j,i-1)
2037  4          &        - vxr2(l)*f(l,k,j,i+2)
2038  4          &        + vxl2(l)*f(l,k,j,i-2)
2040  4  p  2v       dfc_(l)=dfc_(l)+df(l,k,j,i)*fc(l,k,j,i)
2041  4  p  2v     enddo
2042  3  p       enddo
```

分割 1

分割 2

チューニング：最適化の状況

FUJITSU

■ 最適化の状況 (DIFF 区間) (ループ2次元-2分割 + 手動L2プリフェッチ・チューニング)

```
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    PREFETCH       : 8
                  <<<      f: 4, df: 4
                  <<< Loop-information  End >>>
2040  3  p              do k = 1,nz
2041  3            !ocl noprefetch
2042  3  p         !ocl prefetch_read(f(1,k,j+3,i),level=2,strong=1)
2043  3  p         !ocl prefetch_read(f(33,k,j+3,i),level=2,strong=1)
2044  3  p         !ocl prefetch_read(f(65,k,j+3,i),level=2,strong=1)
2045  3  p         !ocl prefetch_read(f(97,k,j+3,i),level=2,strong=1)
2047  3            !-------------------------------------------------
2048  3  p         !ocl prefetch_write(df(1,k,j+1,i),level=2,strong=1)
2049  3  p         !ocl prefetch_write(df(33,k,j+1,i),level=2,strong=1)
2050  3  p         !ocl prefetch_write(df(65,k,j+1,i),level=2,strong=1)
2051  3  p         !ocl prefetch_write(df(97,k,j+1,i),level=2,strong=1)
2053  3            !-------------------------------------------------
2067  3            !ocl unroll(2)
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 4)
                  <<<    SOFTWARE PIPELINING
                  <<< Loop-information  End >>>
2068  4  p  2v           do l = 1,nv
2069  4  p  2v              df(l,k,j,i)=
2070  4                &          vxyv(l)*f(l,k,j,i)
2071  4                &        + vvr(l) *f(l+1,k,j,i)
2072  4                &        - vvl(l) *f(l-1,k,j,i)
2073  4                &        - vvr2(l)*f(l+2,k,j,i)
2074  4                &        + vvl2(l)*f(l-2,k,j,i)
2075  4                &        + vzl2(l)*(f(l,k-2,j,i)-f(l,k+2,j,i))
2076  4                &        + vzl(l) *(f(l,k+1,j,i)-f(l,k-1,j,i))
2077  4                &        + vyr(l) *f(l,k,j+1,i)
2078  4                &        - vyl(l) *f(l,k,j-1,i)
2079  4                &        - vyr2(l)*f(l,k,j+2,i)
2080  4                &        + vyl2(l)*f(l,k,j-2,i)
2081  4  p  2v           enddo
2082  3  p              enddo
2083  2            !------------
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    PREFETCH       : 24
                  <<<      f: 16, df: 4, fc: 4
                  <<< Loop-information  End >>>
```

分割 1

```
2084  3  p              do k = 1,nz
2085  3            !ocl noprefetch
2086  3  p         !ocl prefetch_read(f(1,k,j+1,i-2),level=2,strong=1)
2087  3  p         !ocl prefetch_read(f(1,k,j+1,i-1),level=2,strong=1)
2088  3  p         !ocl prefetch_read(f(1,k,j+1,i+1),level=2,strong=1)
2089  3  p         !ocl prefetch_read(f(1,k,j+1,i+2),level=2,strong=1)
2090  3  p         !ocl prefetch_write(df(1,k,j+1,i),level=2,strong=1)
2091  3  p         !ocl prefetch_read(fc(1,k,j+1,i),level=2,strong=1)
2092  3            !-------------------------------------------------
2093  3  p         !ocl prefetch_read(f(33,k,j+1,i-2),level=2,strong=1)
2094  3  p         !ocl prefetch_read(f(33,k,j+1,i-1),level=2,strong=1)
2095  3  p         !ocl prefetch_read(f(33,k,j+1,i+1),level=2,strong=1)
2096  3  p         !ocl prefetch_read(f(33,k,j+1,i+2),level=2,strong=1)
2097  3  p         !ocl prefetch_write(df(33,k,j+1,i),level=2,strong=1)
2098  3  p         !ocl prefetch_read(fc(33,k,j+1,i),level=2,strong=1)
2099  3            !-------------------------------------------------
2100  3  p         !ocl prefetch_read(f(65,k,j+1,i-2),level=2,strong=1)
2101  3  p         !ocl prefetch_read(f(65,k,j+1,i-1),level=2,strong=1)
2102  3  p         !ocl prefetch_read(f(65,k,j+1,i+1),level=2,strong=1)
2103  3  p         !ocl prefetch_read(f(65,k,j+1,i+2),level=2,strong=1)
2104  3  p         !ocl prefetch_write(df(65,k,j+1,i),level=2,strong=1)
2105  3  p         !ocl prefetch_read(fc(65,k,j+1,i),level=2,strong=1)
2106  3            !-------------------------------------------------
2107  3  p         !ocl prefetch_read(f(97,k,j+1,i-2),level=2,strong=1)
2108  3  p         !ocl prefetch_read(f(97,k,j+1,i-1),level=2,strong=1)
2109  3  p         !ocl prefetch_read(f(97,k,j+1,i+1),level=2,strong=1)
2110  3  p         !ocl prefetch_read(f(97,k,j+1,i+2),level=2,strong=1)
2111  3  p         !ocl prefetch_write(df(97,k,j+1,i),level=2,strong=1)
2112  3  p         !ocl prefetch_read(fc(97,k,j+1,i),level=2,strong=1)
2113  3            !-------------------------------------------------
2158  3            !ocl unroll(4)
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 4)
                  <<<    SOFTWARE PIPELINING
                  <<< Loop-information  End >>>
2159  4  p  4v           do l = 1,nv
2160  4  p  4v              df(l,k,j,i)= df(l,k,j,i)
2161  4                &        + vxr(l) *f(l,k,j,i+1)
2162  4                &        - vxl(l) *f(l,k,j,i-1)
2163  4                &        - vxr2(l)*f(l,k,j,i+2)
2164  4                &        + vxl2(l)*f(l,k,j,i-2)
2166  4  p  4v              dfc_(l)=dfc_(l)+df(l,k,j,i)*fc(l,k,j,i)
2167  4  p  4v           enddo
2168  3  p              enddo
```

分割 2

チューニングによる性能改善

FUJITSU

- PA情報で比較 (チューニングの効果)
- ループ2分割 ＋ L2プリフェッチ によるチューニングの実施
  - 2次元で2分割した各ループの unroll 数を調整(2と4)して SWPL を促進
  - 1ループ中の処理を小さくすることにより、スピル･フィルを削減
  - L2デマンドミス率が高いため、メモリレイテンシの隠蔽で、L2プリフェッチを実施

DIFF	コード	レジスタ数	スレッド数	実行時間 (sec)	L1 ビジー率	L2 ビジー率	L2 スループット (GB/sec)	メモリ ビジー率	メモリ スループット (GB/sec)
FX100 (PA情報)	asis	128reg	16	0.37	69%	67%	263.09	69%	96.82
	asis	32reg		0.43	75%	57%	225.21	58%	80.64
	⇒tune			0.38	71%	68%	261.57	72%	99.65

DIFF	コード	レジスタ数	スレッド数	ロード･ ストア数	L1D ミス率	L1D ミス dm 率	L2 ミス率	L2 ミス dm 率
FX100 (PA情報)	asis	128reg	16	2.79E+10	1.35%	16.83%	0.39%	44.34%
	asis	32reg		3.08E+10	1.22%	14.52%	0.35%	50.10%
	⇒tune			3.23E+10	1.21%	21.58%	0.37%	0.74%

- チューニングにより、L2デマンドミス率が減少し、L2スループットが高い状態となった。

GCRソルバのPost-FX100性能推定

■ GCRカーネル

行列特性：非対称行列,17点ステンシル（5点差分×4次元）
問題サイズ：N_xxN_yxN_zxN_v=48x48x16x128
（16と12で割り切れるようにN_xxN_y=48x48と調整）
コンパイラオプション：
(共通)　-xpre_diff -Kfast -Kopenmp -Kocl,optmsg=2 -V -X9 -Kmfunc=2 -Cpp -Kloop_nofusion -CcdRR8 -Kswp_strong -Kprefetch_nostrong -Nlst=t,lst=i,lst=x -Kprefetch_sequential=soft
(DAXPY)　-Kprefetch_cache_level=all
(SpMV)　-Kprefetch_cache_level=1

DAXPY1

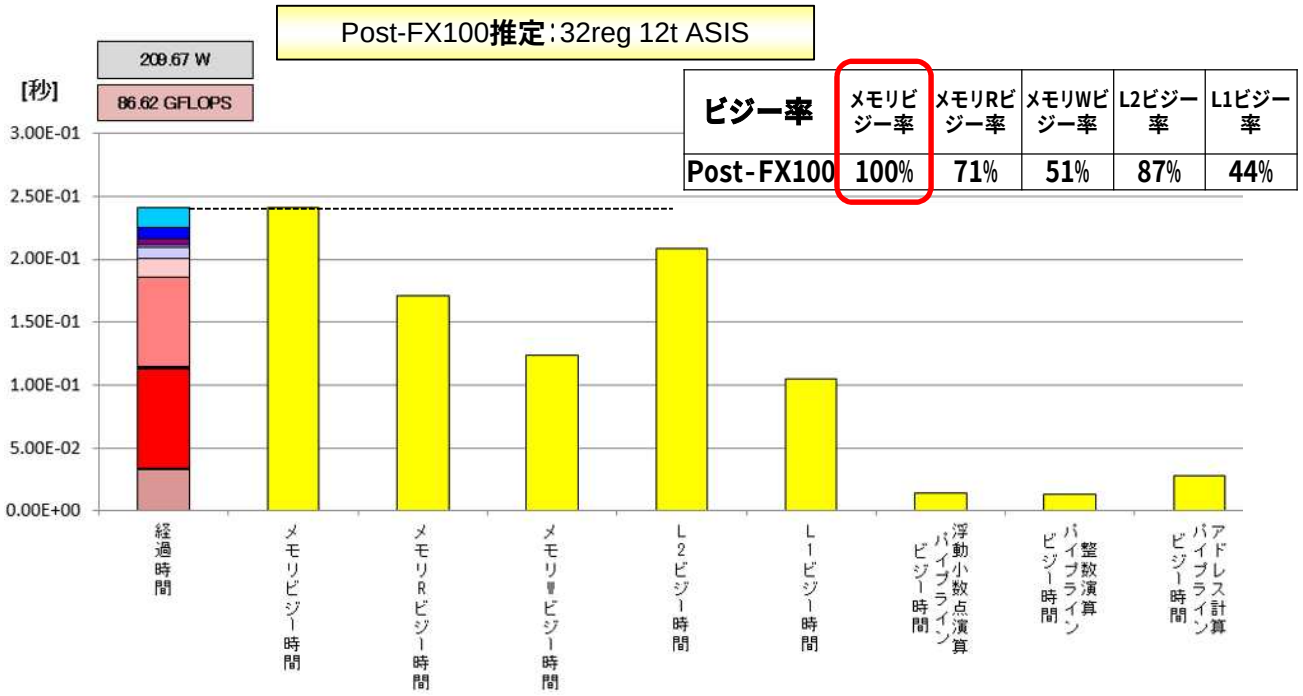
FUJITSU

FX100 の実行時間と Post-FX100推定値の比較

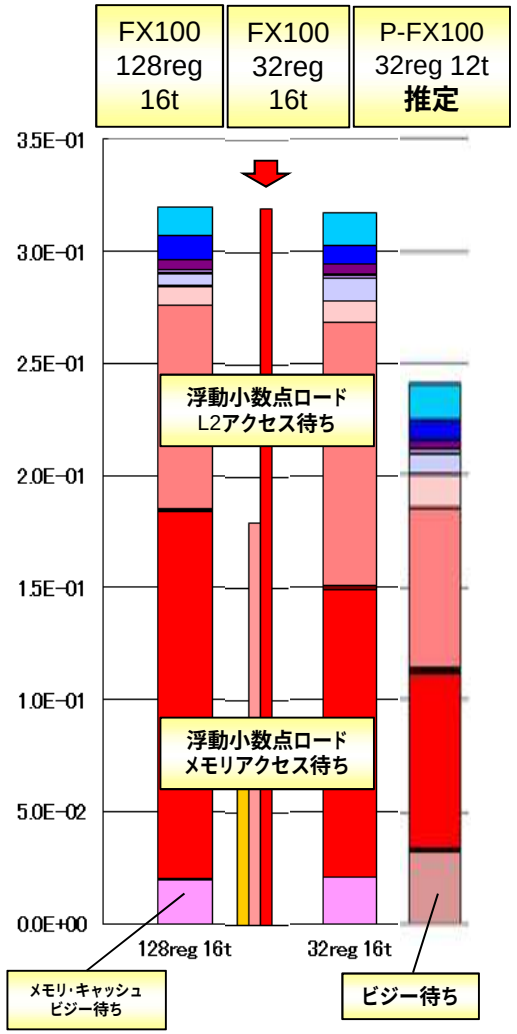
	レジスタ数	スレッド数	実行時間(sec)	比率
FX100 (PA実行時間)	128reg	16	0.32	1.00
	32reg		0.32	1.00
P-FX100 (推定値)		12	0.24	0.75

推定性能について

- メモリバンド幅ネックのループである。



ビジー率	メモリビジー率	メモリRビジー率	メモリWビジー率	L2ビジー率	L1ビジー率
Post-FX100	100%	71%	51%	87%	44%



DAXPY2

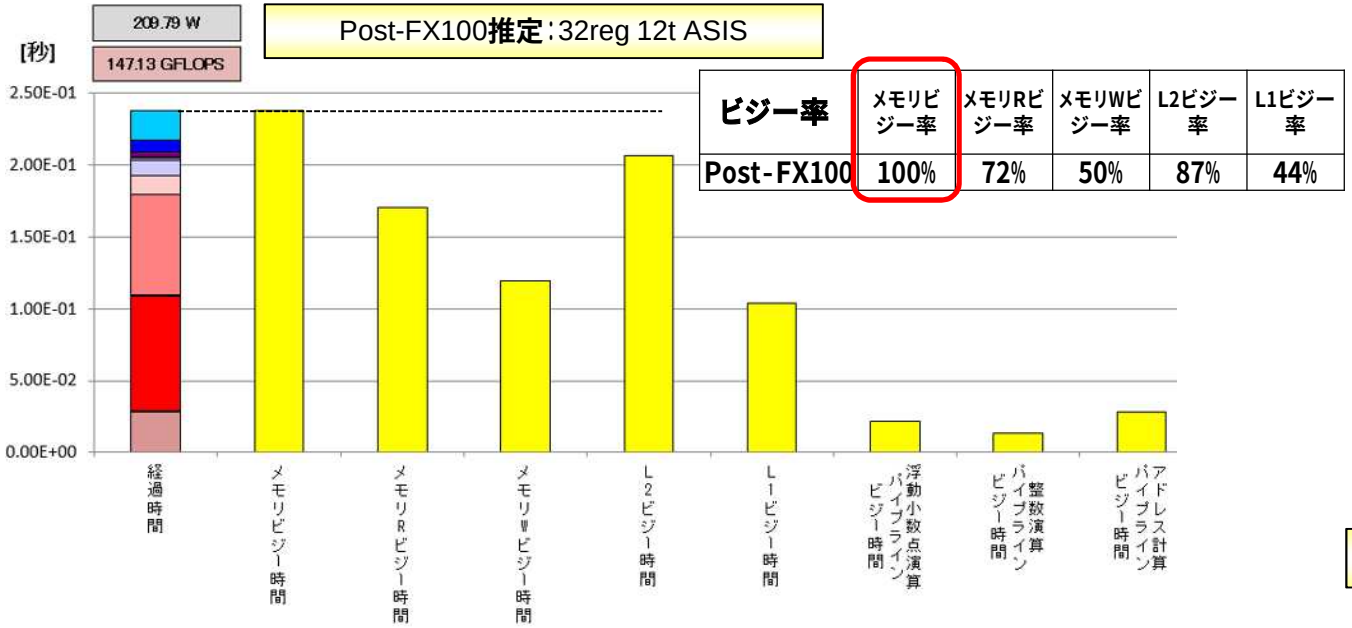
FUJITSU

FX100 の実行時間と Post-FX100推定値の比較

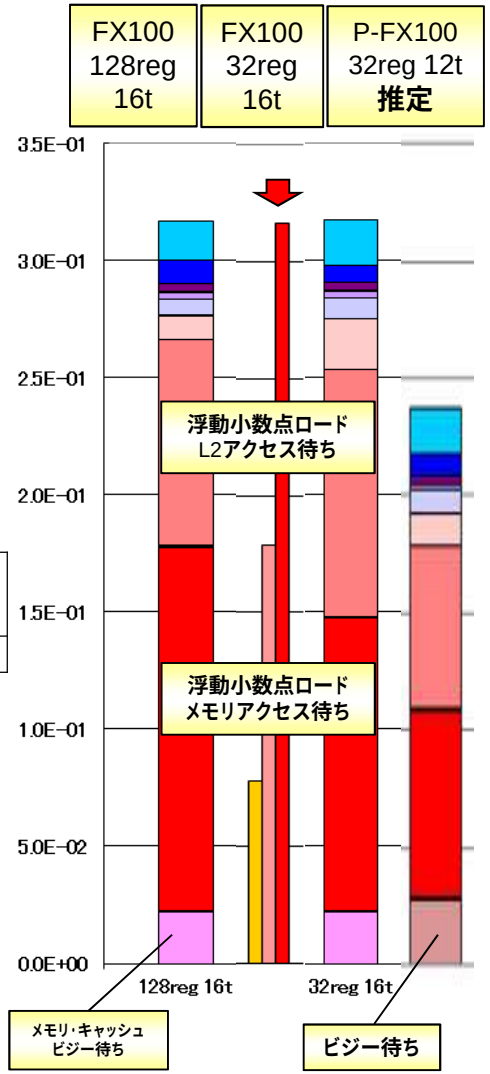
	レジスタ数	スレッド数	実行時間(sec)	比率
FX100 (PA実行時間)	128reg	16	0.32	1.00
	32reg		0.32	1.00
Post-FX100 (推定値)		12	0.24	0.75

推定性能について

- VEC1 と同様に、メモリバンド幅ネックのループである。



ビジー率	メモリビジー率	メモリRビジー率	メモリWビジー率	L2ビジー率	L1ビジー率
Post-FX100	100%	72%	50%	87%	44%



SpMV

FUJITSU

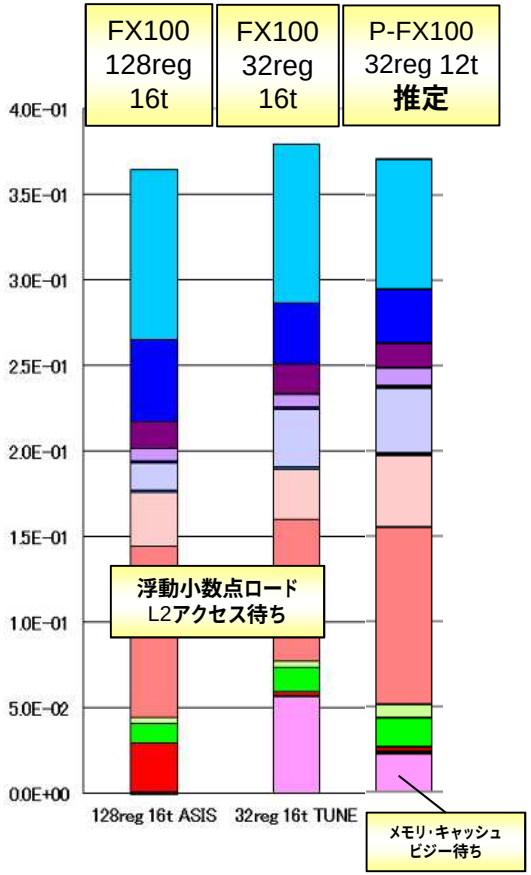
FX100 の実行時間と Post-FX100 推定値の比較

	コード	レジスタ数	スレッド数	実行時間(sec)	比率
FX100 (PA実行時間)	asis	128reg	16	0.37	1.00
	tune (2次元分割)	32reg		0.38	1.02
Post-FX100 (推定値)			12	**0.37**	1.00

推定性能について

- L2ビジー傾向のループとなっている。128regと同等な性能となった。

ビジー率	メモリビジー率	メモリRビジー率	メモリWビジー率	L2ビジー率	L1ビジー率	浮動小数点演算パイプラインビジー率	整数演算パイプラインビジー率	アドレス計算パイプラインビジー率
Post-FX100	51%	41%	18%	84%	70%	20%	11%	30%



4.4　N 体カーネルの性能推定

理化学研究所計算科学研究センター　似鳥　啓吾
富士通株式会社次世代ＴＣ開発本部　青木　正樹

4.4.1 概要

前回の WG（ポストペタアプリ性能 WG、P-PAP）では倍精度演算でニュートン重力の時間導関数を含めた N 体計算を、時間積分を含めて性能評価を行った。対象としたアーキテクチャは富士通 FX10/FX100、Intel KNC 等である。このコードでは

- 組み込み関数サポートを含むコンパイラ性能
- 演算数の少ないときのコア間同期性能

を評価するようなベンチマークとなっている。今回はもう少し単純な、単精度で導関数を含まない重力のみのカーネルの評価を行った。このような条件にした理由は

- SVE で単精度 SIMD がフルサポートされるようになったこと
- レジスタ数が 32 本になった影響を見極めておきたかったこと
- 実機での評価はまだ先になりそうなのでできるだけ単純な条件にしておきたかったこと

が挙げられる。よって本 WG での評価では単一コアで命令列がいかに効率よく実行パイプラインを埋めることができるかに主眼を向けたものとなる。N 体計算のカーネルは主記憶帯域への要求は低いものの命令間の依存関係という点では DGEMM や SGEMM のような行列行列乗算よりも性能を出すのが難しいケースもある。

4.4.2 Intel KNL での計測

第 2 回会合（2017/4/19）では、Intel KNL 上で AVX-512 命令を用いたコードの性能について発表した。よく知られているとおり、ポスト「京」での SIMD 幅も 512-bit であるため、コード移植の準備や比較対象のアーキテクチャとして KNL は適している。

この作業の中で最も驚きだったのが、Intel コンパイラが外側ループに対して#pragma simd を付加しただけで理想的に SIMD 化されたコードを出してしまったことである。$O(N^2)$ の N 体計算カーネルには重力を受け取る粒子についての外側の i-ループと重力の源となる粒子についての内側の j-ループがある。通常コンパイラが自動ベクトル化を行うのは最内ループに対してであるが、計算効率の観点からは外側ループの並列度が利用可能なときはこちらを利用するのが望ましい。ディレクティブをひとつ付けただけでメモリオペランドの放送を含めてコンパイラが外側ループの SIMD 化をしてしまったこと、組み込み関数版も作ってはみたもののアセンブリ出力も性能もコンパイラベクトル化のものとほぼ同一であったことを書き留めておく。

KNL の命令セットでひとつ特徴的なのは、逆数や逆数平方根の近似命令に対して単一命令で 28-bit 精度のものが用意されていることである。これらの命令は AVX-512ER というカテゴリに含まれ、Skylake Xeon から利用可能なのは 14-bit 精度の近似命令のみである。倍精度に対しては 1 回の Newton 法反復、単精度の場合は反復なしでフルの精度が得られる。追

加の回路と引き換えに単に命令数が減った分だけ速くなるだけでなく、数値定数格納のためのレジスタや out-of-order リソースの消費削減も期待できる。

計測された性能であるが、KNL の 1 コアに対して 1/2/4 スレッドを用いた場合ループ回転のサイクル数にして 14.55/13.00/11.34 であった。ループ中の命令数が 18、内 AVX 命令が 14 であったので、理想値は 2 命令発行のアーキテクチャの場合 9 サイクル、3 命令発行でレジスタ間の mov 命令を排除できたとすると 6.5 サイクルであり、良好なスケジューリングが得られているといえる。

少し別の話題にはなるが、この WG では SIMD レジスタを用いたテーブル参照の手法も紹介した。ARM SVE にも同様の tbl という命令が存在する。周期境界条件の分子動力学や宇宙論 N 体計算では相補誤差関数やガウス関数が相互作用カーネルに含まれるため、このような手法が威力を発揮する。

4.4.3 SVE 命令セットのコンパイラ出力

第 6 回会合（2018/4/16）では、富士通より SVE 命令用にコンパイルされた結果の提供を受け、その中身を簡単に紹介した。実機やシミュレータでの性能評価には至っていない。前回の Intel コンパイラでは外側ループに対して SIMD 化が行われたが、富士通コンパイラにはそのような機能は未対応とのことで、内側の j-ループに対して SIMD 化とソフトウェアパイプライニングが適用された。粒子データは 3 つの座標と 1 つの質量の 4 単精度語をパックした AoS (Array of Structures)になっていたため、粒子データのレジスタへの読み込みが Gather load になってしまっている。外側ループの SIMD 化ではそれぞれの語をレジスタないしメモリオペランドとして水平に放送していたので問題にならなかった箇所ではあるが、ここは SIMD 化の方針によっては SoA (Structure of Arrays)にユーザー側で変更すべき箇所であろう。逆数平方根の部分は、数値定数を 3 つ保持して有効桁数を一度に 3 倍にする方法が用いられていた。これは HPC-ACE のときの単精度逆数平方根の扱いと同様である。ARM SVE には数値定数のためにレジスタを消費せず有効桁数を 2 倍にする補助命令も用意されているがこちらは使われていなかった。

4.4.4 富士通での性能評価

第 8 回会合（2018/10/19）では富士通からの性能評価の報告があった。ただしこれは A64FX の実機やシミュレータでの評価ではなく、FX100 でレジスタ数を 32 本に制限してコンパイル・実行したもの、もしくはそのプロファイリング情報から Post-FX100 での性能を推計したものとなる。単純にレジスタ数を削減した結果は 128 レジスタ時にはほぼ見られなかった浮動小数点演算待ちが大幅に増加し、性能は 49%まで低下してしまっている。これはレジスタ不足によりソフトウェアパイプライニングが適用されなくなったため（SVE は swp されてる）で、その後ソフトウェアパイプライニングを適用すべく手動でループを 2 分割した版が示された。この結果ループ分割前から 22%の性能向上が得られ 128 レジスタ時の 60%の性能となっている。

なお、単純に FX100 でレジスタ数を削減しただけでは A64FX での正確な性能予測は不可能であり、命令セットの変更によるアドレス計算の削減や強化された out-of-order リソー

スといった性能改善要素を加味する必要がある（現時点では直接数字を出すことはできないが、実際にシミュレータを用いた計測で推計値よりも性能が改善することが確認された)。

Out-of-order や緩やかなソフトウェアパイプライニングが性能改善をもたらした場合、ループ分割の是非も逆転する可能性がある。この文書を執筆開始した 2019 年 2 月現在ではフラッグシップ２０２０プロジェクトにおいて DGEMM や STREAM といった重要なカーネルの実機評価の数値も徐々に公開され始めているので、遠くない将来に一般ユーザーにとっても実機のデータが入手可能になることを期待したい。

N体計算ベンチマーク
（AVX-512で遊んでみた話）

2017/4/19

SS研メニーコア時代のアプリ性能検討WG第2回会合

理化学研究所計算科学研究機構

似鳥啓吾

はじめに

- 実機で試せる新アーキテクチャということでAVX-512 on KNL を試してみた
- 以下の２つをやってみた
  1. 単純な重力多体計算カーネル
     - コードはhttps://github.com/nitadori/gravity-512/に
  2. SIMDレジスタを用いたテーブル参照

AVX-512について

- インテルのSIMD命令セットとしては大幅なメジャーアップデート
- KNightsCornerでの拡張命令はx86 CPUのブランチ的な存在だったが、AVX-512ではSSE/AVXとの互換性も確保しつつSkylake Xeonからも採用
  - 現状ではKNightsLandingsで実機テストが可能
  - Skylake Xeon E5 v5はそろそろ発売？Amazon cloudで最初に触れるらしい
- レジスタ幅が512-bitに、倍精度なら8語、単精度なら16語
- レジスタの本数が32本へと倍増、マスクレジスタ新設
- オペランド修飾が充実、特にメモリオペランドから放送ができる

重力多体問題計算カーネル

```
#pragma omp parallel for
#pragma simd
        for(int i=0; i<N; i++){
                float ax=0.0f;
                float ay=0.0f;
                float az=0.0f;
                float po=0.0f;

                float xi = posm[i].x;
                float yi = posm[i].y;
                float zi = posm[i].z;

                for(int j=0; j<N; j++){
                        float dx = xi - posm[j].x;
                        float dy = yi - posm[j].y;
                        float dz = zi - posm[j].z;

                        float r2 =  eps2 + dx*dx + dy*dy + dz*dz;
                        float ri = 1.0f / sqrtf(r2);

                        float mri  = posm[j].m * ri;
                        float ri2  = ri * ri;
                        float mri3 = mri * ri2;

                        ax -= mri3 * dx;
                        ay -= mri3 * dy;
                        az -= mri3 * dz;
                }

                accp[i].ax = ax;
                accp[i].ay = ay;
                accp[i].az = az;
        }
}
```

- 外側ループの並列度に対して#pragma simdを付けただけ
- 内側ループの自動ベクトル化よりは高い性能が期待できる
- 内側ループには逆数平方根

$$\frac{d^2}{dt^2}\boldsymbol{r}_i = \sum_{j\neq i}^{N} m_j \frac{\boldsymbol{r}_j - \boldsymbol{r}_i}{\|\boldsymbol{r}_j - \boldsymbol{r}_i\|^3}$$

アセンブリ出力

```
..B1.42:                        # Preds ..B1.42 ..B1.41
                                # Execution count [2.50e+01]
        vsubps    (%rcx,%r10){1to16}, %zmm10, %zmm18
        vmovaps   %zmm4, %zmm14
        vsubps    4(%rcx,%r10){1to16}, %zmm9, %zmm19
        addq      $1, %r11
        vfmadd231ps %zmm18, %zmm18, %zmm14
        vsubps    8(%rcx,%r10){1to16}, %zmm8, %zmm21
        vfmadd231ps %zmm19, %zmm19, %zmm14
        vfmadd231ps %zmm21, %zmm21, %zmm14
        vrsqrt28ps %zmm14, %zmm15{%k1}{z}
        vmulps    12(%rcx,%r10){1to16}, %zmm15, %zmm16
        vmulps    %zmm15, %zmm15, %zmm17
        addq      $16, %r10
        vmulps    %zmm17, %zmm16, %zmm20
        vfnmadd231ps %zmm18, %zmm20, %zmm13
        vfnmadd231ps %zmm19, %zmm20, %zmm12
        vfnmadd231ps %zmm21, %zmm20, %zmm11
        cmpq      %r15, %r11
        jb        ..B1.42       # Prob 82%
```

- インテルコンパイラがほぼ完璧なコードを吐いてしまった
  - 人生であまりない経験
- 放送ロードがメモリオペランドに入る
- 単精度の逆数平方根は1命令
- 18命令（内AVX命令は14）
- 理想的にレイテンシが隠蔽されると9 cycle/loop程度か

組込み関数版

```
for(int j=0; j<N; j++){
    // _mm_prefetch((const char *)&posm[j+1], _MM_HINT_T0);

    __m512 dx = _mm512_sub_ps(xi, _mm512_set1_ps(posm[j].x));
    __m512 dy = _mm512_sub_ps(yi, _mm512_set1_ps(posm[j].y));
    __m512 dz = _mm512_sub_ps(zi, _mm512_set1_ps(posm[j].z));

    __m512 r2 = _mm512_fmadd_ps(dx, dx, eps2);
    r2 = _mm512_fmadd_ps(dy, dy, r2);
    r2 = _mm512_fmadd_ps(dz, dz, r2);

    __m512 ri = _mm512_rsqrt28_ps(r2);

    __m512 mri  = _mm512_mul_ps(ri, _mm512_set1_ps(posm[j].m));
    __m512 ri2  = _mm512_mul_ps(ri, ri);
    __m512 mri3 = _mm512_mul_ps(mri, ri2);

    ax = _mm512_fnmadd_ps(mri3, dx, ax);
    ay = _mm512_fnmadd_ps(mri3, dy, ay);
    az = _mm512_fnmadd_ps(mri3, dz, az);
}
```

- 書いては見たものの、#pragma simdによる自動ベクトル化版と全く変わらない出力
- この一件をもって「SIMD化なんかコンパイラが全部やってくれる」と一般化された期待をするべきではないと思いますが、隔世の感はあります

KNLでのシングルコア性能

- 1.3 GHz 1コアでSMTを1/2/4にして計測
- 1スレッドでもout-of-orderがうまく機能している
  - KNCでは最低2スレッドが必要でした
- 4スレッドにすることで27.7%の性能向上

core/thread	nsec/loop	cycle/loop
1C1T	11.19	14.55
1C2T	10.00	13.00
1C4T	8.76	11.34

テーブル参照の話

- 実数1入力1出力の数学関数高速化においては常套手段
- 特に入力区間や必要精度がわかっているときに強力
  - 例：分子動力学での相補誤差関数や指数関数
- 必ずしも最近のマシンで性能が出るわけではない
  - キャッシュに収めてなおかつヒットさせないとパイプラインがストールする
  - SIMD化が実質不可能、gatherは遅い
- Wide SIMDのレジスタに対する水平シャッフルでテーブル参照を行ってみた
  - AVX-512だと単精度なら16エントリのテーブル
  - 2つのレジスタをソースに32エントリとできる命令まである
- KNLで簡単なテストを作ってみた
  - `float exp2f(float x)`, $y=2^x$ for $x \in [1.0, 2.0)$
  - 3次の多項式

SIMDでテーブル参照

- こんなことができる命令がある
  `b[i] = a[idx[i]];`
  - `a`と`b`は`float[16]`のSIMDレジスタ
  - `idx`は`int[16]`のレジスタ(下４-bitのみを使う `0<=idx[i]<16`)
- AVX-512の該当命令は：`vpermps`、組込み関数だと
  `__m512 _mm512_permutexvar_ps (__m512i idx, __m512 a)`

もっと強力な命令もある

- vpermi2ps, vpermt2p
- __m512 _mm512_permutex2var_ps (__m512 a, __m512i idx, __m512 b)
- **データにレジスタ２本、インデクスに１本**
  - **aとbを繋いでidxで要素を参照**
    c[i] = idx&16 ? b[idx[i&15]] : a[idx[i&15]];
  - **単精度32エントリのテーブルとして利用可能**
  - **FMA3同様、３つの入力オペランドのうち１つが破壊される**

Cではこんなコード

- `float exp2f(float x)`, $y=2^x$ for $x \in [1.0, 2.0)$
- 区間を32分割して3次の多項式で近似
- 単精度の丸め誤差程度までの精度が得られた
  - 結果的に16エントリでも十分だった

```
float exp2app_32(const float x){
    static const float tab[32][4] = {
#       include"tab32x4.h"
    };
    union{
        float    f;
        unsigned u;
    } m32;
    m32.f = x;
    m32.u >>= 18;
    const int idx = m32.u & 31;
    m32.u <<= 18;
    const float h = x - m32.f;

    const float *a = tab[idx];
    const float y = a[0] + h * (a[1] + h * (a[2] + h * (a[3])));

    return y;
}
```

テーブルの生成

- Sollyaというツールの Remezアルゴリズム(minimax近似)を用いた
  - http://sollya.gforge.inria.fr/

```
prec = 60;
f = 2^x;
for n from 0 to 31 do {
    p = remez(f(x+1+n/32), 3, [0;1/32]);
    // p;
    print("{",
            coeff(p,0), ",",
            coeff(p,1), ",",
            coeff(p,2), ",",
            coeff(p,3), "}," );
    print("//", dirtyinfnorm(p - f(x+1+n/32), [0;1/32]));
};
```

実装

- さすがにこのぐらい複雑なものになると組込み関数が必要
- 原理的には特定の大きさのテーブル参照に対してはコンパイラからの生成も可能

```
for(int i=0; i<n; i+=16){
    __m512  xv = _mm512_loadu_ps(&x[i]);
    __m512i sr = _mm512_srli_epi32((__m512i)xv, 18);
    __m512i sl = _mm512_slli_epi32(sr, 18);
    __m512  dx = _mm512_sub_ps(xv, (__m512)sl);

//  __m512i idx = _mm512_and_epi32(sr, _mm512_set1_epi32(31));
    __m512i idx = sr;
    __m512 a0 = _mm512_permutex2var_ps(z00, idx, z01);
    __m512 a1 = _mm512_permutex2var_ps(z10, idx, z11);
    __m512 a2 = _mm512_permutex2var_ps(z20, idx, z21);
    __m512 a3 = _mm512_permutex2var_ps(z30, idx, z31);

    __m512 yv =
        _mm512_fmadd_ps(dx,
            _mm512_fmadd_ps(dx,
                _mm512_fmadd_ps(dx, a3, a2), a1), a0);

    _mm512_storeu_ps(&y[i], yv);
}
```

```
vmovups   (%r14,%rax,4), %zmm8
vmovaps   %zmm7, %zmm13
vmovaps   %zmm5, %zmm12
vmovaps   %zmm3, %zmm11
vpsrld    $18, %zmm8, %zmm10
vpslld    $18, %zmm10, %zmm9
vpermt2ps %zmm6, %zmm10, %zmm13
vsubps    %zmm9, %zmm8, %zmm14
vpermt2ps %zmm4, %zmm10, %zmm12
vpermt2ps %zmm2, %zmm10, %zmm11
vpermi2ps %zmm0, %zmm1, %zmm10
vfmadd231ps %zmm14, %zmm10, %zmm11
vfmadd231ps %zmm14, %zmm11, %zmm12
vfmadd213ps %zmm13, %zmm12, %zmm14
vmovups   %zmm14, (%rbx,%rax,4)
addq      $16, %rax
cmpq      %r12, %rax
jl        ..B1.30
```

性能

- On 1 core of Intel(R) Xeon Phi(TM) CPU 7210 @ 1.30GHz
- 32768 elements, 256 KiB read and 256 KiB write

- 16-entry, 1-thread
  - 0.487948 nsec/element, 7.807161 nsec/loop
- 16-entry, 4-thread
  - 0.424348 nsec/element, 6.789574 nsec/loop
- 32-entry, 1-thread
  - 0.661530 nsec/element, 10.584481 nsec/loop
- 32-entry, 4-thread
  - 0.634580 nsec/element, 10.153279 nsec/loop

まとめ

- SIMD幅がどんどん広がると使い勝手が悪化していくというのが一般論だけど
- 広大なレジスタ空間（zmmが32本あると2 KiB、4スレッドで8 KiB）を活用するという楽しみ方もあります

- とはいっても、Xeon PhiではなくてSkylake XeonのAVX-512の方が本命だと思います

重力多体問題
（FujitsuコンパイラSVE出力）

2018/4/16

SS研メニーコアWG

理研CCS　似鳥啓吾

概要

- 富士通様から重力多体問題計算カーネルのSVE向けコンパイル結果を提供していただきました
- 動かして性能計測はまだですが、コンパイル結果を見ていこうと思います

ARM SVEについて

- Scalable Vector Extension
- ARM社のSIMD拡張命令
- 特徴：
  - Vector Length Agnostic (VLA)：ベクトル長不可知論
  - 128-bitの倍数のSIMD幅なら最大2048-bitまでSIMD幅は「実装依存」
  - 同じバイナリが複数の異なるチップで動く
    - SIMD長の実装に依りループの反復数が変化する
    - （適切にコンパイルされていれば）結果はSIMD長に依らず一致
- Intelと比較して
  - 同じAVX-512でもKNLとSkylakeでサポートしている命令が違う
  - ただしKNxは市場から消え去りそう
- Fujitsu SPARC HPC-ACEと比較して
  - 倍精度にあったあの命令が単精度には無い、みたいなことが起こらない
  - すべての命令を検証するのは大変そう・・・

入力C++コード

- 普通の2重ループ
- ただしデータ構造はAoS
  - 外側ループに対するSIMD化と放送ロードの使用を想定しました
  - 内側ループに対してSIMD化するとstructure loadになります

```
struct PosM{
    float x, y, z;
    float m;
};

struct AccP{
    float ax, ay, az;
    float pot;
};
```

```
#pragma omp parallel for
#pragma loop simd
    for(int i=0; i<N; i++){
        float ax=0.0f;
        float ay=0.0f;
        float az=0.0f;
        float po=0.0f;

        float xi = posm[i].x;
        float yi = posm[i].y;
        float zi = posm[i].z;

        for(int j=0; j<N; j++){
            float dx = xi - posm[j].x;
            float dy = yi - posm[j].y;
            float dz = zi - posm[j].z;

            float r2 =  eps2 + dx*dx + dy*dy + dz*dz;
            float ri = 1.0f / sqrtf(r2);

            float mri  = posm[j].m * ri;
            float ri2  = ri * ri;
            float mri3 = mri * ri2;

            ax -= mri3 * dx;
            ay -= mri3 * dy;
            az -= mri3 * dz;
        }

        accp[i].ax = ax;
        accp[i].ay = ay;
        accp[i].az = az;
    }
```

コンパイラのメッセージ

- Optimization messages
- jwd6004s-i "test1.cpp", line 34: リダクション演算を含むループ制御変数'j'のループをSIMD化しました。
- jwd8204o-i "test1.cpp", line 34: ループにソフトウェアパイプライニングを適用しました。
- jwd8205o-i "test1.cpp", line 34: ループの回転数が96回以上の時、ソフトウェアパイプライニングを適用したループが実行時に選択されます。
- jwd8209o-i "test1.cpp", line 39: 多項式の演算順序を変更しました。
- jwd8206o-i "test1.cpp", line 40: 除算を逆数の乗算に変更しました。

- 内側ループに対してSIMD化とソフトウェアパイプライニングが行われました

コンパイル結果

- SWPがかかったものはちょっととても読めないので、端数用の単純ループ
  - SWPの方もspillはなかった
- 残念ながらGather load命令になっていまっている
  - ld1wという命令
  - まぁこれは元のデータ構造の側の問題でもある
  - movprfxについては後述

```
.L181:                  // :entr:term:mod:swpl
/*     35 */    add x16, x8, w5, sxtw #2
/*     36 */    add w15, w5, 1
/*     37 */    add w14, w5, 2
/*     42 */    add w9, w5, 3
/*     35 */    ld1w    {z3.s}, p0/z, [x16, z19.s, sxtw #2] //  ___3
/*     36 */    add x15, x8, w15, sxtw #2
/*     49 */    add w5, w5, 64
/*     49 */    subs    w1, w1, 1
/*     36 */    ld1w    {z4.s}, p0/z, [x15, z19.s, sxtw #2] //  ___3
/*     37 */    add x14, x8, w14, sxtw #2
/*     37 */    ld1w    {z5.s}, p0/z, [x14, z19.s, sxtw #2] //  ___3
/*     42 */    add x9, x8, w9, sxtw #2
/*     42 */    ld1w    {z26.s}, p0/z, [x9, z19.s, sxtw #2] //  ___3
/*     35 */    fsub    z3.s, z7.s, z3.s
/*     36 */    fsub    z4.s, z16.s, z4.s
/*     37 */    fsub    z5.s, z17.s, z5.s
/*     39 */    movprfx z6.s, p0/z, z3.s
/*     39 */    fmad    z6.s, p0/m, z3.s, z18.s
/*     36 */    fmla    z6.s, p0/m, z4.s, z4.s
/*     37 */    fmla    z6.s, p0/m, z5.s, z5.s
/*     40 */    frsqrte z24.s, z6.s
/*     40 */    fmul    z6.s, z6.s, z24.s
/*     40 */    fcmeq   p1.s, p0/z, z24.s, #0.0
/*     40 */    fmsb    z6.s, p0/m, z24.s, z20.s
/*     40 */    movprfx z25.s, p0/z, z21.s
/*     40 */    fmad    z25.s, p0/m, z6.s, z22.s
/*     40 */    fmul    z6.s, z6.s, z24.s
/*     40 */    fmad    z6.s, p0/m, z25.s, z24.s
/*     40 */    sel z6.s, p1, z23.s, z6.s
/*     42 */    fmul    z26.s, z26.s, z6.s
/*     43 */    fmul    z6.s, z6.s, z6.s
/*     44 */    fmul    z6.s, z6.s, z26.s
/*     46 */    fmls    z0.s, p0/m, z3.s, z6.s
/*     48 */    fmls    z1.s, p0/m, z5.s, z6.s
/*     48 */    fmls    z2.s, p0/m, z4.s, z6.s
/*     49 */    bne .L181
```

SVEでのFMA (fused multiply-add)

- 命令フィールドの都合からか3 operandだけ取る
  - 加減算もしくは乗算オペランドのどちらかが破壊される
    - Intel AVX2のFMA3と同様
  - 加減算オペランドのレジスタを破壊する命令
    - FMLA, FMLS, FNMLA, FNMLS
  - 乗算オペランドのレジスタを破壊する命令
    - FMAD, FMSB, FNMAD, FNMSB
- どちらのオペランドも破壊されて欲しくないとき
  - movprfx命令を直前に書いて破壊されるレジスタを避難させる
  - ただし実行コストを伴わない
  - 後続命令を修飾して4オペランドFMAのように振る舞わせる

- こういった知識は通常のアプリプログラミングでは不要。コンパイラの出力を読むためにはある程度慣れておく必要がある。

逆数平方根の部分

- 有効桁数を3倍にする公式が使われている
  - 初期値推定の精度は8-bit程度であると予想される
  - この点はHPC-ACEと同じ
- SVEにはRSQRTSという収束補助の命令がある
  - 1/2倍するのに余計な命令を消費しない
  - 3.0のような数値定数の保存にレジスタを消費しない
  - 倍精度のときはこれを3回呼ぶのだと思う

```
// /*    40 */ frsqrte z24.s, z6.s
z24 =  rsqrte(z6);
// /*    40 */ fmul   z6.s, z6.s, z24.s
z6 = z6 * z24;
// /*    40 */ fcmeq  p1.s, p0/z, z24.s, #0.0
p1 = z24 == 0.0;
// /*    40 */ fmsb   z6.s, p0/m, z24.s, z20.s
z6 = z20 - z6 * z24;
// /*    40 */ movprfx z25.s, p0/z, z21.s
// /*    40 */ fmad   z25.s, p0/m, z6.s, z22.s
z25 = z22 + z6 * z21;
// /*    40 */ fmul   z6.s, z6.s, z24.s
z6 = z6 * z24;
// /*    40 */ fmad   z6.s, p0/m, z25.s, z24.s
z6 = z25 * z6 + z24;
// /*    40 */ sel z6.s, p1, z23.s, z6.s
z6 = p1 ? z23 : x6;

y =  rsqrte(x);
h = 1.0f - (x*y)*y;
p = 0.5 + 0.375 * h
yh = y*h;
y =  y + yh*p;
```

逆数平方根命令略史

- AMD 3DNow!（20周年）
  - float2 SIMDに対して15-bit精度
  - 収束補助命令が2つ
  - 21個の新命令に対して2が逆数、3つが逆数平方根まわりだったことになる
- Intel SSE
  - float4 SIMDに対して12-bit精度
  - このあとず〜〜っとAVX-512になるまで倍精度からは近似命令を呼べなかった
  - IntelとAMDでビットレベルの互換性はなかった
- HPC-ACE
  - double2もしくはfloat2に対して8-bit精度
- HPC-ACE2
  - double4もしくはfloat4。float8は？
- AVX-512
  - 14-bit精度になった、倍精度からも単精度からも利用可能
  - KNLではいきなり28-bit精度が得られる拡張が利用可能

要望

- 外側ループに対してSIMD化したい
  - AoSのまま放送ロードで対応できる
  - VLAの思想的にも、総和を含むSIMD化はベクトル長ごとに結果が異なってしまう
- できればディレクティブで
  - Intelコンパイラにはでたことなので
  - OpenMP 4.0かOCLかはどちらでもいいけど

- 今のコンパイラで既に実現可能ならコードを見てみたい
  - まだできないなら実機を我々が触れる時期までに
  - これができるようになるとFDPS on Post-Kが素晴らしく使いやすいものになります

SS研メニーコア時代のアプリ性能検討WG資料

FUJITSU
shaping tomorrow with you

N体 カーネルの評価報告

2018年10月19日
富士通株式会社

目次

FUJITSU

- 測定環境
  - 評価環境
  - FX100 の測定条件と評価コード
- FX100 の性能分析
  - 最適化の状況
  - 128reg と 32reg コンパイラ の比較
  - チューニング
- Post-FX100 推定
  - Post-FX100 のアーキテクチャ
  - 性能予測ツールの概要
  - 評価区間の推定 (チューニング)
- まとめ

FUJITSU

測定環境

- 評価環境
- FX100 の測定条件と評価コード

評価環境

FUJITSU

- FX100（ハードウェア、ソフトウェアの情報）

		FX100 諸元
CPU	名称	SPARC64 Xifx
	動作周波数	1.975GHz
	コア数	32コア＋アシスタントコア
	キャッシュメモリ容量	L1I$: 64KB/コア(4way) L1D$: 64KB/コア(4way) L2$ ：24MB/CPU
	理論ピーク性能	1011GFlops/CPU(倍精度) 2022GFlops/CPU(単精度)
	主記憶-CPU間帯域	(240+240)GB/s/CPU
ノード	CPU数、コア数	1CPU/ノード (16コアx2CMG＋アシスタントコア)/ノード
	理論ピーク性能	1011GFlops/ノード(倍精度) 2022GFlops/ノード(単精度)
	主記憶容量	32GB/ノード
	主記憶−CPU間帯域	(240+240)GB/s
	ノード間通信帯域	100GB/s
OS の名称とバージョン		FX100向けOS 2.1.1

FX100 の測定条件と評価コード

FUJITSU

FX100 の測定条件

項目	条件
コンパイラ	Post-FX100向けチューニング用コンパイラ
翻訳時オプション	ASIS：-Kfast,ocl,openmp 、ループ分割時 ：-Kloop_nofusion
スレッド数	16スレッド
実行条件	1ノード – 1プロセス
PA 採取条件	評価区間をイタレーション1,000で回転させてPA情報を採取

評価コード

- 似鳥様からご紹介頂いた
  https://github.com/nitadori/gravity-512
  よりダウンロードしたソースを利用
- 評価区間は gravity_simple
- 規模 N は 1,024 (オリジナル：N = 16 * 1,024)
- 単精度 (C++)

FUJITSU

FX100 の性能分析

- 最適化の状況
- 128reg と 32reg コンパイラ の比較
- チューニング

最適化の状況 (評価区間のソースリスト)

FUJITSU

■ ソースコードの概要 (gravity_simple)

```
 :
26      void gravity_simple (
27          const int N,
28          const float eps2,
29          const PosM * __restrict posm,
30          AccP * __restrict accp)
31      {
32      #pragma omp parallel for
33      // #pragma simd          (現状 omp simd は未対応)
34  p       for (int i=0; i<N; i++) {
35  p           float ax=0.0f;
36  p           float ay=0.0f;
37  p           float az=0.0f;
38              float po=0.0f;
39
40  p           float xi = posm[i].x;
41  p           float yi = posm[i].y;
42  p           float zi = posm[i].z;
43
```

```
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 4)
                <<<    SOFTWARE PIPELINING
                <<< Loop-information  End >>>
44  p  4v    for (int j=0; j<N; j++) {
45  p  4v        float dx = xi - posm[j].x;
46  p  4         float dy = yi - posm[j].y;
47  p  4         float dz = zi - posm[j].z;
48
49  p  4         float r2 = eps2 + dx*dx + dy*dy + dz*dz;
50  p  4v        float ri  = 1.0f / sqrtf(r2);
51
52  p  4v        float mri   = posm[j].m * ri;
53  p  4v        float ri2    = ri * ri;
54  p  4v        float mri3 = mri * ri2;
55
56  p  4v        ax -= mri3 * dx;
57  p  4v        ay -= mri3 * dy;
58  p  4v        az -= mri3 * dz;
59  p  4v    }
60
61  p        accp[i].ax = ax;
62  p        accp[i].ay = ay;
63  p        accp[i].az = az;
64  p      }
65       }
```

ループ回転数128回以上で適用

ループ回転数
N は 1024

128reg と 32reg コンパイラ の比較

FUJITSU

PAの実行時間で比較

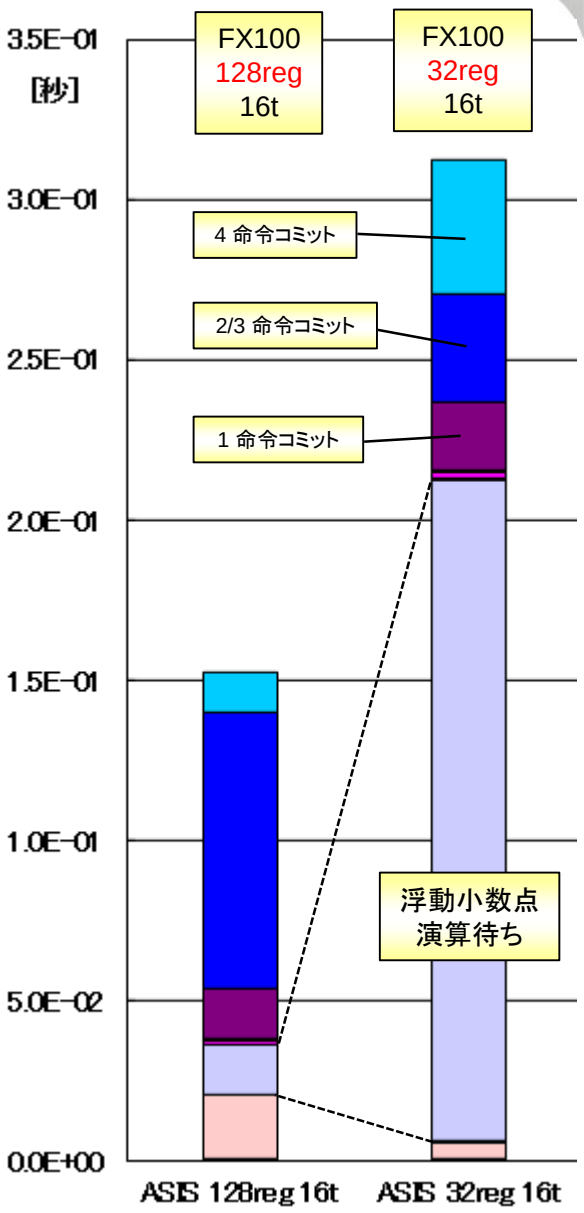
- 128reg と 32reg コンパイラの性能向上比は 0.49倍であり、レジスタ数減少による性能影響は大きい。

gravity _simple	レジスタ数	スレッド数	実行時間 (sec)	性能 向上比
FX100	128	16	0.153	1.00
	32		0.313	0.49

最適化状況

- 32reg では、レジスタ不足でソフトウェアパイプライニングが適用されないため、浮動小数点演算待ち が増加し、性能に影響が出ている。⇒ 32reg ではループ分割が必要。

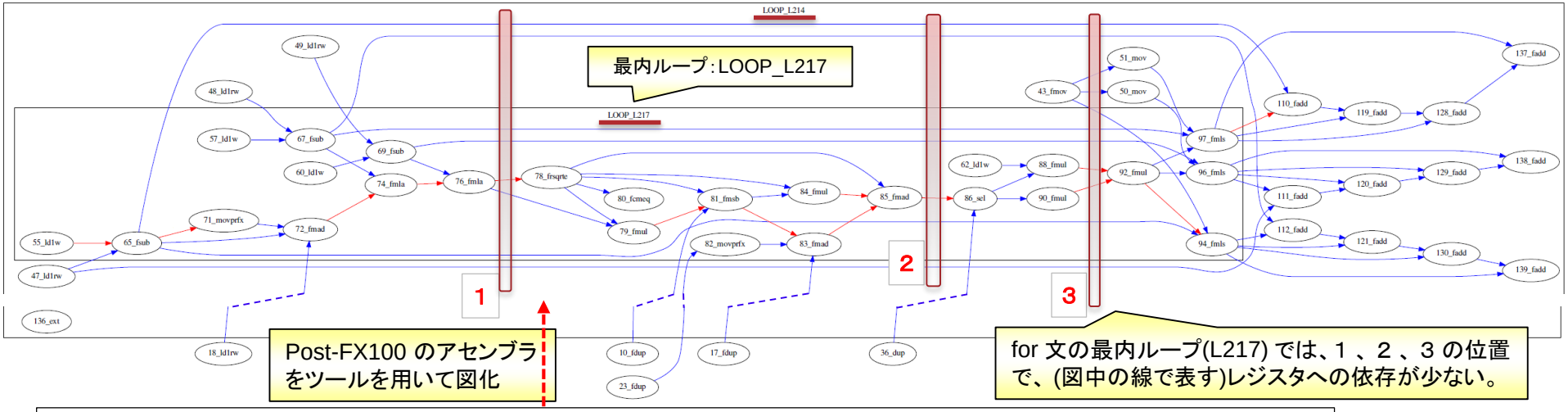
レジスタ数	SIMD	UNROLL	swpl 適用	Reg Fp
128	4	4	128回以上	85 / 128
32	4	2	浮動小数点 レジスタ不足	–



チューニング：ループ分割の検討

FUJITSU

■ ループの依存関係図から分割点を検討



Post-FX100 のアセンブラをツールを用いて図化

for 文の最内ループ(L217) では、1 、2 、3 の位置で、(図中の線で表す)レジスタへの依存が少ない。

```
33  void gravity_simple (const float eps2)
34  {
35  #pragma omp parallel for
36  // #pragma simd
37      for (int i=0; i<N; i++) {
38          float ax=0.0f;
39          float ay=0.0f;
40          float az=0.0f;
41          float po=0.0f;
42
43          float xi = posm_x[i];
44          float yi = posm_y[i];
45          float zi = posm_z[i];
46
47          for (int j=0; j<N; j++) {
48              float dx = xi - posm_x[j];
49              float dy = yi - posm_y[j];
50              float dz = zi - posm_z[j];
51
52              float r2 = eps2 + dx*dx + dy*dy + dz*dz;
                                                  ⇒ 候補 1
53              float ri = 1.0f / sqrtf(r2);
54                                                ⇒ 候補 2
55              float mri   = posm_m[j] * ri;
56              float ri2   = ri * ri;
57              float mri3 = mri * ri2;
58                                                ⇒ 候補 3
59              ax -= mri3 * dx;
60              ay -= mri3 * dy;
61              az -= mri3 * dz;
62          }
63
64          accp_ax[i] = ax;
65          accp_ay[i] = ay;
66          accp_az[i] = az;
67      }
68  }
```

外側ループ：L214

最内ループ：L217

以下の観点から、候補2 で評価を実施する
・使用レジスタ数が少ない
・アンロールが必要となるリダクション部分と使用レジスタ数が多い 1/sqrtf部分 を分離
・命令数バランスを考慮

チューニング：ソース (最内ループを2分割)

チューニングのソースコードの概要 (gravity_simple)

```
 :
38      #pragma omp parallel for                    追加した work 配列
                    private(wk_dx,wk_dy,wk_dz,wk_ri)
39      // #pragma simd              (現状 omp simd は未対応)
40  p     for (int i=0; i<N; i++) {
41  p       float ax=0.0f;                        ループ回転数 N は 1024
42  p       float ay=0.0f;
43  p       float az=0.0f;
44
45  p       float xi = posm[i].x;
46  p       float yi = posm[i].y;
47  p       float zi = posm[i].z;
48
49          // loop-div-1
        <<< Loop-information Start >>>
        <<<  [OPTIMIZATION]                         ループ回転数32回以上で適用
        <<<    SIMD(VL: 4)
        <<<    SOFTWARE PIPELINING
        <<< Loop-information  End >>>
50  p  2v    for (int j=0 ; j<N ; j++) {
51             // work (x,y,z)                      ループ回転数
52  p  2v       wk_dx[j] = xi - posm[j].x;         N は 1024
53  p  2v       wk_dy[j] = yi - posm[j].y;
54  p  2v       wk_dz[j] = zi - posm[j].z;
55                                                 区間1
56  p  2        float r2 =  eps2 + wk_dx[j]*wk_dx[j]
                    + wk_dy[j]*wk_dy[j] + wk_dz[j]*wk_dz[j];
57  p  2v       wk_ri[j] = 1.0f / sqrtf (r2);
58  p  2v    }
```

```
59          // loop-div-2
        <<< Loop-information Start >>>
        <<<  [OPTIMIZATION]                         ループ回転数48回以上で適用
        <<<    SIMD(VL: 4)
        <<<    SOFTWARE PIPELINING
        <<< Loop-information  End >>>
60  p  3v    for (int j=0 ; j<N ; j++) {
61  p  3v       float ri = wk_ri[j];               ループ回転数
62                                                 N は 1024
63  p  3        float mri  = posm[j].m * ri;
64  p  3v       float ri2   = ri * ri;
65  p  3v       float mri3 = mri * ri2;            区間2
66
67             // work (x,y,z)
68  p  3v       ax -= mri3 * wk_dx[j];
69  p  3v       ay -= mri3 * wk_dy[j];
70  p  3v       az -= mri3 * wk_dz[j];
71  p  3v    }
72
73  p        accp[i].ax = ax;
74  p        accp[i].ay = ay;
75  p        accp[i].az = az;
76  p     }
77      }
```

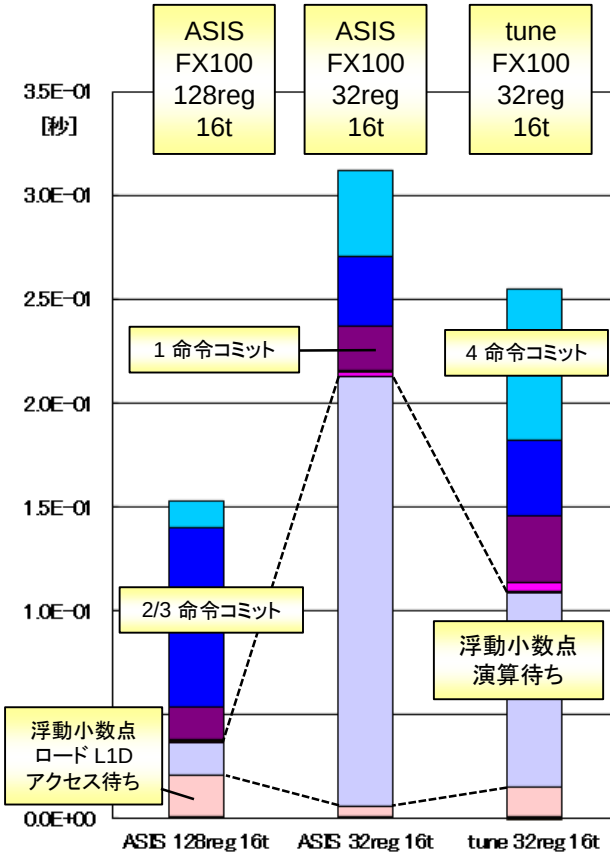
赤字は ASIS からの修正点

コードチューニング

FUJITSU

- PAの実行時間で比較
  - ループ分割のチューニングにより、32reg ASIS とループ分割を比較すると 1.22倍 の性能向上が見られる。

gravity _simple	スレッド数	レジスタ数	ソース	実行時間 (sec)	性能向上比
FX100	16	128	ASIS	0.153	-
		32	ASIS	0.313	1.00
			ループ分割	0.255	1.22



- 最適化状況
  - ループ分割によりソフトウェアパイプライニングが適用されたことで、浮動小数点演算待ちが減少して性能改善した。作業領域の受け渡しや、アドレス計算などが増加したため、命令数は増加している。

ソース	レジスタ数	ループ分割	SIMD	UNROLL	swpl 適用	Reg Fp
ASIS	128	—	4	4	128回以上	85 / 128
	32		4	2	レジスタ不足	—
ループ分割		区間1	4	2	32回以上	30 / 32
		区間2	4	3	48回以上	29 / 32

FUJITSU

Post-FX100 推定

- Post-FX100 のアーキテクチャ
- 性能予測ツールの概要
- 評価区間の推定 (チューニング)

Post-FX100 のアーキテクチャ

FUJITSU

機能		Post-FX100	PRIMEHPC FX100	京
ISA		ARMv8-A + SVE	SPARC V9 + HPC-ACE2	SPARC V9 + HPC-ACE
SIMD	bit幅	512	256	128
	単精度要素数	16	8（命令に制限有）	2
	倍精度要素数	8	4	2
汎用整数レジスタ（本）		31+SP（v8-Aと同じ）	32+32（g,l,i,o+xg）	32+32（g,l,i,o+xg）
ベクトルレジスタ（本）		32	128	128
プレディケートレジスタ（本）		16	（ベクトルレジスタを共用）	（ベクトルレジスタを共用）

12

性能予測ツールの概要

FUJITSU

アプリコード(カーネル部)

FX100:32regコンパイラを使用し
詳細プロファイラ実行でPA情報を取得

PA情報:詳細プロファイラの出力結果(csvファイル)

性能予測ツール

性能予測値

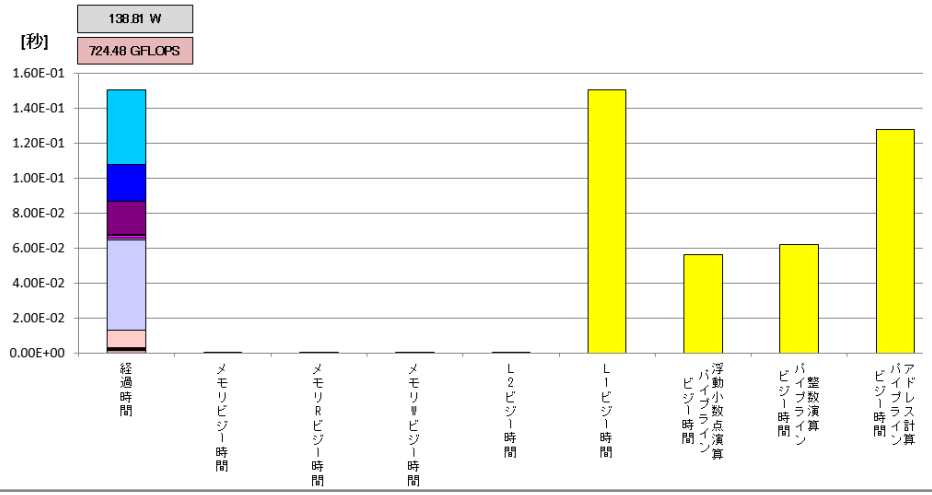
- 性能予測ツール概略
  - FX100のPA情報を入力とする
  - ハード、命令セット／コンパイラによる性能向上効果を算出する
  - 性能予測値を出力する

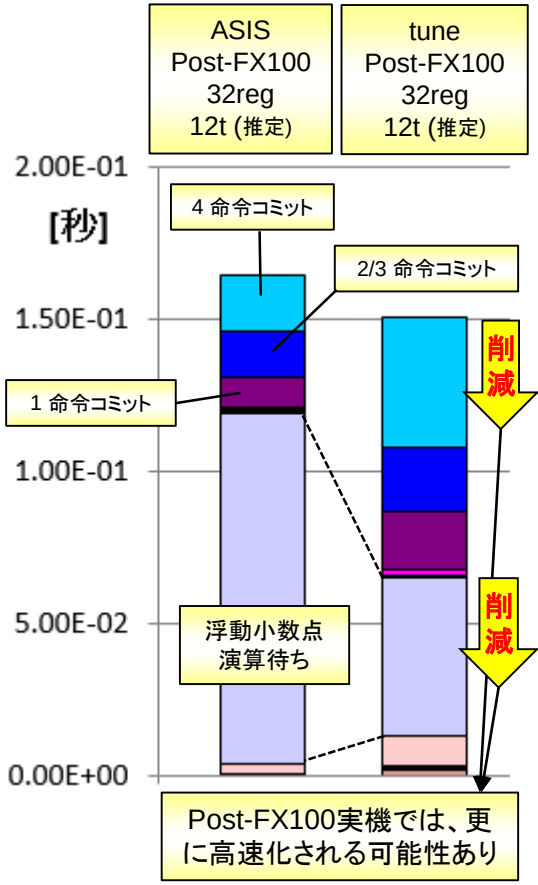
評価区間の推定 (コードチューニング)

- Post-FX100 推定値の比較 (ASIS と ループ分割)

gravity_simple	ソース	レジスタ数	スレッド数	実行時間 (sec)	性能向上比
Post-FX100 (単精度推定値)	ASIS	32	12	0.165	1.00
	ループ分割			0.151	1.09

- **推定性能について**
  - ループ分割を実施したことにより、FX100の結果同様ASIS性能に対して性能が改善した。
  - FX100の性能向上比と比較して、Post-FX100の向上比が小さい原因は、ループ分割による作業配列アクセス命令（アドレス計算）が増加し、命令コミットが増加したためである。
  - ISA対応のシミュレーションでチューニングコードを性能測定した結果、チューニング性能（推定）に対して約20%の性能向上を確認した。





FUJITSU

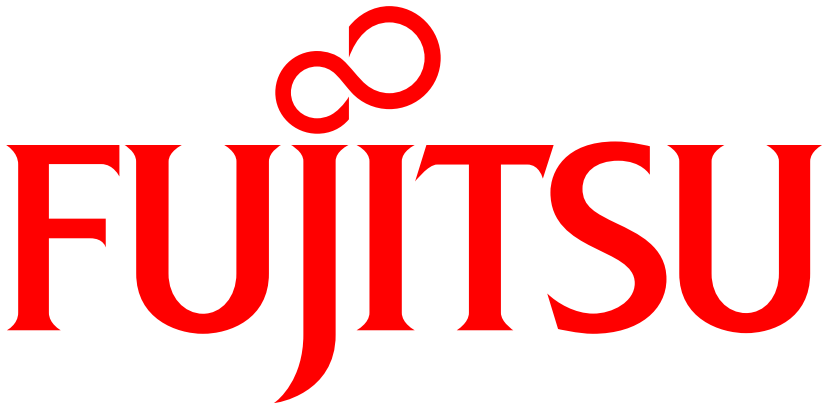
まとめ

まとめ

FUJITSU

- ASIS の問題点
  - 演算ネックのループのため、アーキレジスタ数減少の影響は出ている。
    - 32reg コンパイラでは、レジスタ不足により、ソフトウェアパイプライニングが適用されず、浮動小数点演算待ちが増加し、性能に影響が出る。

- チューニングによる対応
  - 32reg コンパイラでは、ループ分割が必要となる。
    - Post-FX100 では、コンパイラの分割機能により、自動で分割される。（予定）
  - ループ分割によりソフトウェアパイプライニングが適用されて性能は改善した。
    - ISA対応のシミュレーションでチューニングコードを性能測定した結果、チューニング性能（推定）に対して約20%の性能向上を確認できた。



shaping tomorrow with you