「A64FXシステム アプリ性能検証WG」活動報告

片桐孝洋

名古屋大学情報基盤センター

科学技術計算分科会 2022年度会合
「富岳」NEXTへの挑戦～現在から未来へ～
2023年1月20日(金) 分科会、神戸国際会議場
11:25-12:15（講演 40分，Q&A 10分）

発表内容

- A64FXシステムアプリ性能検討WGのご紹介
- 活動事例
- おわりに

発表内容

- A64FXシステムアプリ性能検討WGのご紹介
- 活動事例
- おわりに

活動方針

- スーパーコンピュータ「富岳」が本格運用に迫り、多種のスーパーコンピュータがメニーコアCPUを採用しメニーコア時代に突入している。
- アプリ研究開発者にとっては、いまだに大規模コアの有効利用には様々な困難が伴う。
- その問題解決のためには、コンパイラ等のシステムソフトウェアと協調して性能最適化を行う知識と技術が利用者に求められるほか、システムソフトウェア自体の自動性能チューニングも必要である。
- そこで本WGでは、A64FXのARMプロセッサ環境を対象に、コンパイラ、メッセージ通信ライブラリ、性能解析ツール等の改善点について議論し、そのノウハウの集約と共有を行う。

活動内容

- 以下の3段階に分けて、2年間のWG活動を実施
- 前半3～4回で、A64FXを対象として、メニーコアプロセッサの性能評価やシステムソフトウェア改善について議論する。具体的には、推進委員が所有するアプリケーション等を中心に、現在のA64FXシステムの問題点を洗い出す。また、新しいアプリケーション分野（例えばAI）についても検討する。
- 中盤3～4回で、前半で洗い出された問題点を考慮し、富士通が提供するコンパイラ等の問題点・課題点に対して評価検討を行う。最終的には、現存するA64FXシステムのための利用者マニュアル、コンパイラ機能を含めた改善について検討する。
- 後半2回で、取りまとめた結果を成果物としてまとめ、2022年度 SS研科学技術分科会のイベントにて成果報告を実施
- 成果物として、「利用者視点によるコンパイラ機能改善」、「A64FXプロセッサの利用技術」等を想定
- 活動の進め方
  - 活動期間：2020年11月～2022年10月
  - 会合開催：年4～5回、計10回

構成員

- 高木亮治（JAXA）担当幹事
- 片桐孝洋 (名大) *まとめ役
- 熊畑清(JAXA)、小野先生(九大)、藤田航平(東大地震研)、梅田隆行(名大 宇地研)、前山伸也(名大 プラズマ理論)、小野寺直幸(原子力研)、黒田明義(理研)、望月祐志(立教大学)
- [富士通]井上晃(Uvance Core Technology本部）*まとめ役、
- 山中栄次(UX＆FXシステム事業部)、石井邦憲 (同左)、原口正寿(同左)、渡邉雄二(同左)、渡辺健介(同左)、内海裕一郎(同左)、野瀬貴史(同左)、後藤理恵(同左)、斎藤佑馬(同左)、鎌塚 俊(未来社会&テクノロジ本部)、柴田 純(同左)、川島 崇裕(同左)、向井優太(同左)、小山謙太郎(同左)、松井秀司(官庁六事)、稲荷智英(同左)、高科勝俊(同左)、笠井良浩(同左)、福本尚人(富士通研究所)、川上健太郎(同左)
- [SS研事務局] 小野寺啓之、松本孝之、棚橋修一、犬束英輔

成果物


A64FXシステムアプリ性能検討WG
FUJITSU
shaping tomorrow with you
A64FXプログラミングガイド 入門編
2022年10月
富士通株式会社


目次
FUJITSU
A64FXプロセッサ
A64FXプロセッサ概要
A64FXプロセッサ諸元
セクタキャッシュ
高速ストア(zfill)
電力制御
A64FX向けプログラミング開発環境
プログラミング開発環境概要
C/C++ tradモードと clangモード
2つのOpenMPライブラリ
コンパイラ推奨オプション
富士通コンパイラと他コンパイラが生成するオブジェクトの互換性について
従来システムからの移行
コンパイラ実行時ライブラリの主なエントリ名
Fortranがサポートするタイマー
Fortran、C/C++ tradモードのデバッグ機能
ラージページ
CPU性能解析レポートを用いた性能チューニング

発表内容

- A64FXシステムアプリ性能検討WGのご紹介
- 活動事例
- おわりに

事例の紹介

- 全ての事例は時間的に紹介できないので、発表者（片桐）の主観で選んだ、事例を紹介
- 問答形式
  - アプリ開発者による問題提起
  - 富士通のグループによる調査結果提示

事例）
小回転ループで
性能低下の可能性

乱流燃焼コード
LS-FLOW HO (LSHO)
性能改善調査

JAXA 研究開発部門 第三研究ユニット
熊畑 清

2021.3.25(Thu)
Scientific System研究会 第3回 A64FXシステムアプリ性能検討WG

Outlines

- アプリ概要
  - LS-FLOW-HO
  - Flux Reconstruction Method

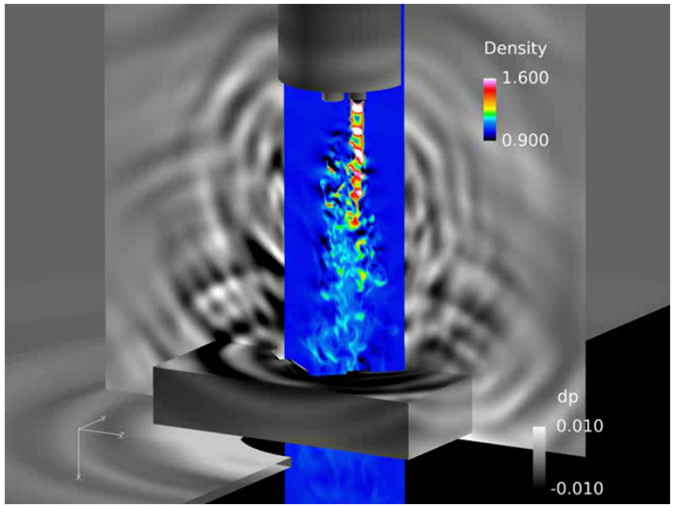
- 性能について直面した問題
  1. 配列代入
  2. スレッド間のインバランス
  3. インライン展開
  4. 重複した処理

- 性能の観点から見た，今後検討するべき点

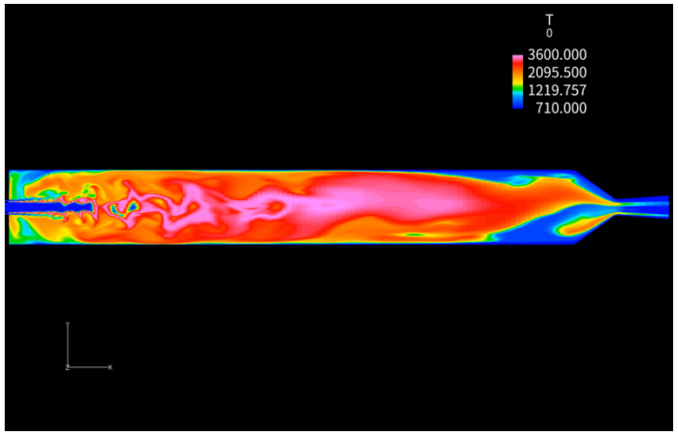
アプリ概要 LS-FLOW-HO

- JAXA 研究開発部門 第三研究ユニットで開発している高次精度の流体・燃焼解析ソルバー[1]
- 有限体積法に基づき，高次精度を達成するため Flux-Reconstructionメソッド[2][3]を採用
  - セルは次数に応じた数の内点(SP)と境界面での点(FP)を持つ
  - セル内SPの情報からセル内局所Fluxを求め，セル間で連続するように修正(Reconstruction)

ロケット発射台の空力音響解析



エンジン燃焼器の乱流燃焼解析



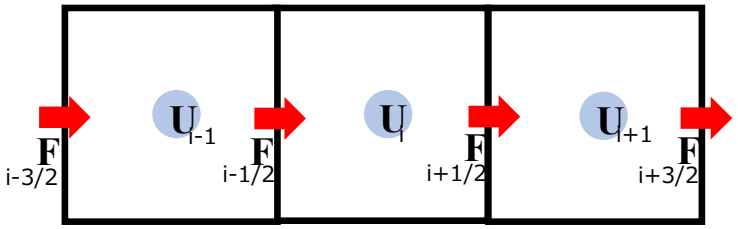
[1] Y.Abe et.al, 2018, Stable, non-dissipative, and conservative flux-reconstruction schemes in split forms.
[2] H.T.Huynh, 2007, A flux reconstruction approach to high-order schemes including discontinuous Galerkin methods.
[3] H.T.Huynh, 2009, A reconstruction approach to high-order schemes including discontinuous Galerkin for Diffusion.

並列数の実績は～1000ノードで完走．2880ノードで現在確認中
ただし本日の内容は全て単体性能の話です

Flux Reconstruction Method

有限体積法

空間をメッシュ化し複数のセル(あるいはControl Volume: CVと呼称)に分割.

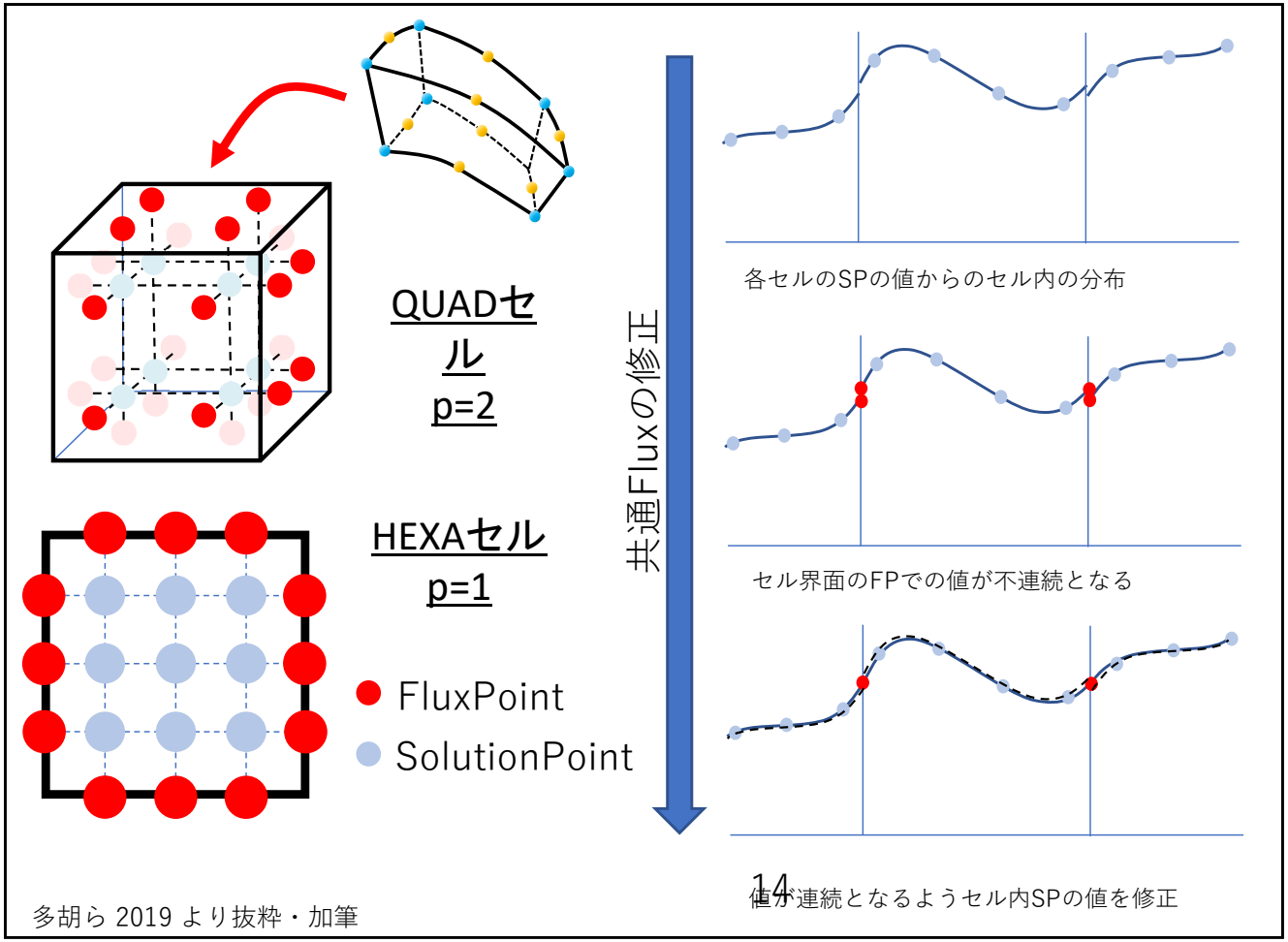


$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}}{\partial x} = \mathbf{0}$$

セル中心の物理量Uを，セル界面を出入りする量F(Flux)との釣り合い式に基づいて計算する

いかにしてセル境界面でのフラックスを精度良く求めるかが重要となる

Flux Reconstruction Method (有限要素法っぽさもある)



14

性能について直面した問題 1

配列代入

コンパイラが複数の配列代入をまとめてしまい
レジスタ不足となりスピルで速度低下している
ケース

プロファイラfippの結果に見られた問題区間

fippの結果に見られるRiemannFluxSimpleの謎

JSS3 (A64FX, frtpx) 2.2GHz 12スレッド				
手続き	開始行	終了行	コスト	コスト%
riemannfluxsimple_	1112	1226	12244	17.5882
computetemperature_sp_eb._OMP_2_	2907	2970	6288	9.0474
residualcorrection._OMP_1_	1957	2115	4956	7.1308
residualfacegradient.OMP_2_	2564	2706	4008	5.7668
residualself_ns2_fc._OMP_1_	4587	4871	3636	5.2316
computegradcomspace_br1_.OMP_1_	7250	7400	3312	4.7654
residualself_euler_div_.OMP_1_	420	660	3193	4.5942
Cal_aCoefficient_SRK			2459	3.5381
Cal_dadT_SRK			2199	3.164
ComputeTransProp_Chung			2115	3.0431
Cal_departureFunction_SRK			2036	2.9295
ComputeTransProp_Chung			1590	2.2877
riemannfluxsimple._PRL_1_	1140	1150	1394	2.0057
residualfacecommon_ns_fc_lad._OMP_1_	2778	2950	1368	1.9683
hllcflux_prim_pev_	1364	1538	1537	1.9525
thermalprop_sp_fc._OMP_2_	3669	3735	1139	1.6388
Spc_Hppp_Cal			1004	1.4446
computegradcomspace._OMP_1_	7154	7212	935	1.3453
residualreconst_temperature._OMP_1_	579	633	912	1.3122
Spc_Cppp_Cal			778	1.1194
newtonIter4T_SRK			705	1.0144
Cal_omegaFunction_SRK			678	0.0755

fippのApplication Procedureでのコスト1%以上の区間

RiemannFluxSimpleが全体の17%のコスト12244を占めている
と出ている

プロファイラ fippによるコスト分布
サブルーチン単位ではなく，手続き単位

※手続き
ループなり関数呼び出しといったコンパイラが認識する処理の単位

1. RiemannFluxSimple内のループではない部分で高コスト
2. コスト1%以上のものを合計しても82%にしかならない(支配的なものがない)
3. インライン展開されているものが多い

呼び出し側と当該区間でタイマーを入れた

タイマー※を挿入して測定を試みた

RiemannFluxSimpleを呼び出す側ループ

```
ResidualSelfCommon
                                                    タイマー
!$omp parallel default (none)                        range1
!$omp private(ff, j, k, ip) &                        ループ全体
!$omp shared(ct, pt, ho, nfp)
!$omp do
do ff=1 ho%gd%nTFace
                              range2                 range3
   call RiemannFluxSimple(pr, ho, nfp, ff)
        サブルーチンを呼んでから戻ってくるまで
Enddo
!$omp end do
!$omp end parallel
```

当該ルーチンRiemannFluxSimple

```
RiemannFluxSimple

subroutine RiemannFluxSimple(・・・)                 range3
   use param_mod                                サブルーチンを呼んでから
   use ho_mod                                   サブルーチン内の実行が始まるまで
   use common_mod
   implicit none
   yype (PARAM),intent(in) :: pr
    ・・・
   real(8) snd_spd_l(nfp)
   real(8) snd_spd_r(nfp)
    ・・・

   ul() = ho%uuf(:,1,ff)
   ur() = ho%uuf(:,2,ff)
    ・・・
```

※omp_get_wtimeによる測定

RiemannFluxSimple内のタイマー区間

```
subroutine RiemannFluxSimple(・・・)
   use param_mod
   use ho_mod
   use common_mod
   implicit none
   yype (PARAM),intent(in) :: pr
   ・・・
   real(8) snd_spd_l(nfp)
   real(8) snd_spd_r(nfp)
   ・・・
```

range3

```
   ul() = ho%uuf(:,1,ff)
   ur() = ho%uuf(:,2,ff)
```

range5

range4

```
   snd_spd_l(:) = ho%SndSpdf(:,1,ff)
   snd_spd_r(:) = ho%SndSpef(:,2,ff)
```

range6

```
   do j=1, nfp
     enthalpy_r(j) = ho%uuf(j*nv,1,ff)
     enthalpy_l(j) = ho%uuf(j*nv,2,ff)
   enddo
```

range7

```
   fna(:,:) = ho%fnaL(:,:,ff)
```

range8

続く

```
   if(pr%calorically_pfct .eq. 1) then
      RusanovFlux_Prim, RoeFlux_Prim,
      RoeFlux_entropyfix_Prim,
      SLAUFluxPrim等を呼び分ける
   else
      RusanovFlux_Prim, RoeFlux_Prim,
      RoeFlux_entropyfix_Prim,
      SLAUFlux_Prim_PEV, SHUSFlux_Prim_PEV,
      KepFlux_Prim_PEV, HLLCFlux_Prm_PEV
      等を呼び分ける
   endif
```

range9

```
   ho%fcom(:,ff) = fn(:)
```

range10

```
end subroutine RiemannFluxSimple
```

range11 = range7+range8+range9+range10

タイマーを入れた際に見られた現象

RiemannFluxSimple周辺にタイマーを挿入して実行すると，速くなった!

メインループ(時間発展ループ)の実行時間がタイマーを入れないと58秒，入れると48秒程度になっている
↓
タイマーの各区間の，有効/無効を変えて調査すると，range5/6へのタイマーの有無でRiemannFluxSimpleの速度が変わる

case	stdout	range1	range2	range3	range4	range5	range6	range7	range8	range9	range10	range11
no31	57.5	12.9	12.8	0.118	12.4			1.08	0.144	1.21	0.221	2.96
no32	47.6	2.94	2.81	0.11	2.41	0.115		0.106	0.104	1.2	0.189	1.91
no33	47.7	3.11	2.99	0.109	2.59	0.113	0.103	0.106	0.104	1.2	0.19	1.91
no34	47.6	2.94	2.81	0.109	2.41		0.106	0.106	0.104	1.2	0.19	1.91

メインループ

ということで**range5/6が怪しい**
↓
当該部分のコンパイルリスト(*lst)を確認したが差がない

コンパイルリスト(*.lstファイル)からのj情報
889回転以上で並列
8SIMD
SWP(IPC=3.0, ITR=80, MVE=5, POL=S)
PREFETCH(HARD) ur, ul

range5/6共に，タイマーの入れ方を変えてもコンパイルリストは変化しなかった

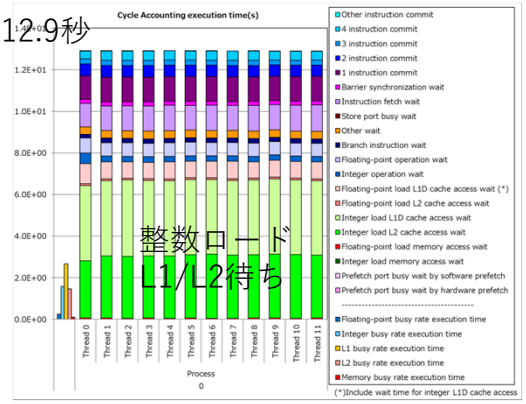
```
jwd5003p-i  "RiemannFluxSimple.F90", line 47: Array description is parallelized.
jwd6003s-i  "RiemannFluxSimple.F90", line 47: SIMD conversion is applied to array description.
jwd8204o-i  "RiemannFluxSimple.F90", line 47: This loop is software pipelined.
jwd8205o-i  "RiemannFluxSimple.F90", line 47: The software-pipelined loop is chosen at run time
when the iteration count is greater than or equal to 80.
```

CPU性能解析レポートを比較

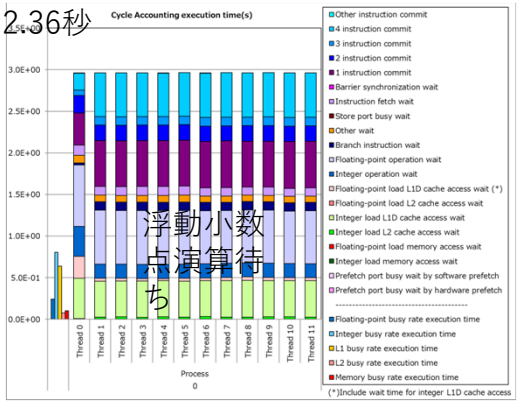
range1(RiemannFluxSimpleを呼んでから返ってくるまでの区間)をFAPP 17回採取で測定

	遅いケース	速いケース
実行時間	12.9秒	2.96秒
浮動小数点演算数	2.60E+11	2.61E+11
L1Iミス率が頻発	している	していない
L1DTLBミスが頻発	している	していない
実行中の 目立つ時間	整数ロードL1アクセス待ち 整数ロードL2アクセス待ち	支配的なものはなし 強いて言えば浮動小数点演算待ち
命令コミット時間合計	2.34秒	1.3秒
有効総命令	**1.27E+11**	**8.47E+11**
Fill命令	**7.18E+8**	**1.76E+5**
DCZVA命令	**1.16E+8**	**2.88E+4**

12.9秒



2.36秒



コンパイルリストは同一だったが，やはりコンパイラが出している命令が違う？
↓
コンパイラのアセンブラ出力を確認してみた

コンパイラのアセンブラ出力を比較

遅い方(range5/6タイマー無し)

```
.L695:
      .loc 1 47 0
..LDL5:
/*    ??? */    ldr     x0, [x19, 40]   //  (*)
      .loc 1 43 0
..LDL6:
/*     43 */    mov     x1, 0
      .loc 1 47 0
..LDL7:
/*     47 */    ldr     x7, [x0, 7664]  //  "ho"
/*     47 */    ldr     x3, [x0, 7616]  //  "ho"
/*     47 */    ldr     x2, [x0, 7648]  //  "ho"
/*     47 */    ldr     x28, [x0, 7632] //  "ho"
/*     47 */    ldr     x25, [x0, 7560] //  "ho"
      .loc 1 63 0 is_stmt 0
..LDL8:
/*     63 */    ldr     x4, [x0, 30648] //  "ho"
/*     63 */    ldr     x8, [x0, 30632] //  "ho"
/*     63 */    ldr     x5, [x0, 30600] //  "ho"
      .loc 1 47 0
..LDL9:
/*    ??? */    str     x3, [x19, 24]   //  (*)
      .loc 1 63 0
..LDL10:
/*     63 */    ldr     x3, [x0, 30616] //  "ho"
/*     63 */    ldr     x0, [x0, 30544] //  "ho"
      .loc 1 47 0
..LDL11:
/*     47 */    mul     x26, x7, x2
      .loc 1 43 0 is_stmt 1
..LDL12:
/*     43 */    adrp    x2, riemannfluxsimple._PRL_1_
/*     43 */    add     x2, x2, :lo12:riemannfluxsimple._PRL_1_
      .loc 1 47 0
..LDL13:
      .loc 1 63 0 is_stmt 0
..LDL14:
/*     63 */    mul     x8, x4, x8
```

ソース47行
ul(:)=ho%uuf(:,1,ff)
関係の処理

ソース63行
snd_spd_l(:)=ho%SndSpdf(:,1,ff)
関係の処理

47行の処理の途上で
63行の処理が始まっていて
混ざっている

長いので以下略

速い方(range5/6タイマー有)

```
.L735:
      .loc 1 47 0
..LDL6:
/*     47 */    ldr     x0, [x21, 7664] //  "ho"
/*     47 */    ldr     x1, [x21, 7648] //  "ho"
/*     47 */    cmp     x28, 0
/*     47 */    ldr     x25, [x21, 7560]        //  "ho"
/*     47 */    ldr     x2, [x21, 7616] //  "ho"
/*     47 */    ldr     x26, [x21, 7632]        //  "ho"
/*     47 */    mul     x1, x0, x1
/*     47 */    mul     x0, x0, x22
/*    ??? */    stp     x1, x2, [x19, 40]       //  (*)
/*     47 */    sub     x1, x2, 1
/*    ??? */    str     x1, [x19, 32]   //  (*)
      .loc 1 48 0 is_stmt 0
..LDL7:
/*     48 */    sub     x1, x2, 2
/*     48 */    msub    x1, x1, x26, x25
      .loc 1 47 0
..LDL8:
/*    ??? */    str     x0, [x19, 16]   //  (*)
h/*    ??? */   ldp     x0, x2, [x19, 32]       //  (*)
      .loc 1 48 0
..LDL9:
/*     48 */    sub     x20, x1, x2
      .loc 1 47 0
..LDL10:
/*     47 */    madd    x0, x0, x26, x2
/*     47 */    sub     x5, x25, x0
/*     47 */    ble     .L745
/*     47 */    cmp     x28, 889
/*     47 */    bge     .L744
/*    ??? */    ldr     x1, [x19, 8]    //  (*)
/*     47 */    ptrue   p0.d, ALL
/*     47 */    asr     x0, x1, 2
/*     47 */    add     x0, x1, x0, lsr #61
/*     47 */    asr     x9, x0, 3
/*     47 */    lsl     x8, x9, 3
```

ソース47行の処理が
集中して配置されている

47行の処理と
63行の処理は
分かれている

長いので以下略

状況からの推察

前提） 配列代入は，コンパイラによりループとして扱われる

推定） コンパイルリストでは，range5/6は別々に扱っているように出ているが，遅い方(オリジナル)では，一つにまとめられており，47行の処理と63行の処理が同時に扱われループボディが長くなりスピルが多発している？

速い方では，range5と6の間にタイマー用のサブルーチンコールが入るため，それぞれ別々に扱われ小さなループになっており，スピルしない？

コンパイルリストではこんな風に思えるが

889回転以上で並列
8SIMD
SWP(IPC=3.0, ITR=80, MVE=5, POL=S)
PREFETCH(HARD) ur, ul

ul(:) = ho%uuf(:,1,ff)
ur(:) = ho%uuf(:,2,ff)

loop1

889回転以上で並列
8SIMD
SWP(IPC=3.0, ITR=80, MVE=5, POL=S)
PREFETCH(HARD) ur, ul

snd_spd_l(:)=ho%Spdf(:,1,ff)
snd_spd_r(:)=ho%Spef(:,2,ff)

loop2

実際にはこうコンパイルされている

889回転以上で並列
8SIMD
SWP(IPC=3.0, ITR=80, MVE=5, POL=S)
PREFETCH(HARD) ur, ul

ul(:) = ho%uuf(:,1,ff)
ur(:) = ho%uuf(:,2,ff)
snd_spd_l(:) = ho%Spdf(:,1,ff)
snd_spd_r(:) = ho%Spef(:,2,ff)

loop1

コンパイルリストが同じになっているのがヒントか？

改変）配列代入をやめループで実装

```
range5
real(8) ul(nv*nfp)
real(8) ur(nv*nfp)

ul(:) = ho%uuf(:,1,ff)
ur(:) = ho%uuf(:,2,ff)
```

```
k=nv*nfp  ループで実装
do j=1,k
   ul(j) = ho%uuf(j,1,ff)
   ur(j) = ho%uuf(j,2,ff)
enddo
```

```
range6
real(8) snd_spd_l(nfp)
real(8) snd_spd_r(nfp)

snd_spd_l(:) = ho%SndSpdf(:,1,ff)
snd_spd_r(:) = ho%SndSpdf(:,2,ff)
```

```
do j=1,nfp ループで実装
   snd_spd_l(j) = ho%SndSpdf(j,1,ff)
   snd_spd_r(j) = ho%SndSpdf(j,2,ff)
enddo
```

range1(RiemannFluxSimpleを呼んでいる時間の合計)は13秒→3秒
メインループ(時間発展ループ)の時間が上記改変で58秒→47秒
要するにrange5とrange6が別とコンパイラに認識されれば良いので
他にも方法はあると思う

性能について直面した問題 2

スレッド間のインバランス

収束計算から抜ける反復計算が
スレッド毎に異なりスレッド間インバランスが
出ているケース

プロファイラfippの結果に見られた問題区間

fippの結果に見られるコスト2番目の区間

JSS3 (A64FX, frtpx) 2.2GHz 12スレッド				
手続き	開始行	終了行	コスト	コスト%
riemannfluxsimple_	1112	1226	12244	17.5882
computetemperature_sp_eb._OMP_2_	2907	2970	6288	9.0474
residualcorrection._OMP_1_	1957	2115	4956	7.1308
residualfacegradient.OMP_2_	2564	2706	4008	5.7668
residualself_ns2_fc._OMP_1_	4587	4871	3636	5.2316
computegradcomspace_br1_.OMP_1_	7250	7400	3312	4.7654
residualself_euler_div_.OMP_1_	420	660	3193	4.5942
Cal_aCoefficient_SRK			2459	3.5381
Cal_dadT_SRK			2199	3.164
ComputeTransProp_Chung			2115	3.0431
Cal_departureFunction_SRK			2036	2.9295
ComputeTransProp_Chung			1590	2.2877
riemannfluxsimple._PRL_1_	1140	1150	1394	2.0057
residualfacecommon_ns_fc_lad._OMP_1_	2778	2950	1368	1.9683
hllcflux_prim_pev_	1364	1538	1537	1.9525
thermalprop_sp_fc._OMP_2_	3669	3735	1139	1.6388
Spc_Hppp_Cal			1004	1.4446
computegradcomspace._OMP_1_	7154	7212	935	1.3453
residualreconst_temperature._OMP_1_	579	633	912	1.3122
Spc_Cppp_Cal			778	1.1194
newtonIter4T_SRK			705	1.0144
Cal_omegaFunction_SRK			678	0.0755

fippのApplication Procedureでのコスト1%以上の区間

RiemannFluxSimpleが全体の17%のコスト2907を占めている
と出ている

プロファイラ fippによるコスト分布
サブルーチン単位ではなく，手続き単位

※手続き
ループなり関数呼び出しといったコンパイラが認識する処理の単位

当該部分のソース

```
!$omp parallel do default(none)&
!$omp
private(icell,i,Rho,P,u,v,w,rhoi,y,t,h,cp,cv,Cs,mu,pr_l,iter, &
!$omp
ir,ix,iy,iz,ie,mwbar,rc,ek1,ee0,dlte,iflag,Ysp,n,hk,hk_tmp)&
!$omp shared(ho,istepHex,coefrec1d,pr,flmlt_sw,smallpos) &
!$omp reduction(max: tmax,itermax) &
!$omp reduction(min: tmin)
  do icell=1,ho%gd%nTCell      7100回転
     do i=1,ho%nsp(icell)      27回転
        ir = (i-1)*nv+1
        ix = ir+1
        iy = ix+1
        iz = iy+1
        ie = iz+1
        Rho  = ho%uup(ir,icell)
        U    = ho%uup(ix,icell)
        V    = ho%uup(iy,icell)
        W    = ho%uup(iz,icell)
        Rhoi = 1.d0/Rho
        do ichem=1,pr%nchem
           Y(ichem) = ho%uup(ie+ichem,icell)
        end do

        if(flmlt_sw.eq.1) then
           call flmlt_get_yi(Y(1), ho%zvar(i,icell),
                             ho%kai(i,icell), Y)
        elseif(flmlt_sw.eq.2) then

        elseif(flmlt_sw.eq.-1) then
           Y = pr%spc_mfrc_init
        endif
        call ComputeMwBar(Y, MwBar)
        call ComputeRconst(Y, MwBar, Rc)
        ! Update Rc here
        ho%Rc(i,icell)=Rc
```

Prefetch(HARD)

SIMD VL8
SWP(IPC=2.5, ITR-112, MVE=8, POL=S)
Prefetch(HARD)

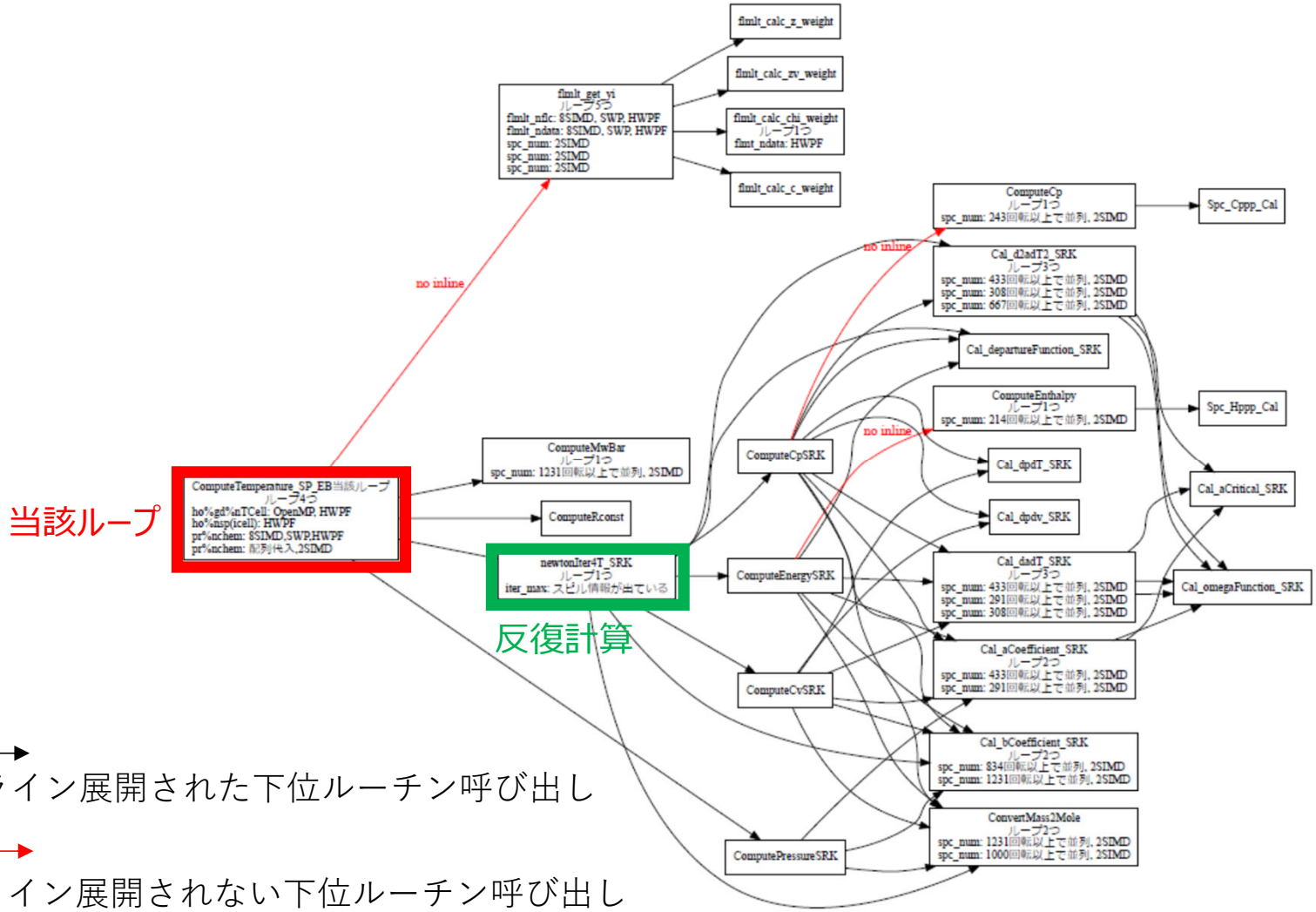
配列式で代入
SIMD VL2

続く

続く

```
        ! initial guess
        T    = ho%T(i,icell) ! this needs to initialize T in IC
        !// compute T from total energy
        ek1  = 0.5d0*Rho*(U*U + V*V + W*W)
        !// rho*e_s
        ee0  = ho%uu(ie,icell) - ek1
        call newtonIter4T_SRK(T,Rho,Y,MwBar,Rc,ee0,ek1,dlte,
             pr%iter_newton,pr%conv_newton,iflag,iter)

        if(iflag==1)then
           write(*,*)'not converge (newton iteration), icell= ',
icell
           stop
        end if

        ho%T(i,icell) = MAX(T, smallpos)
        tmax=dmax1(T,tmax)
        tmin=dmin1(T,tmin)
        itermax=MAX(iter,itermax)

        ! pressure from EOS
        call ComputePressureSRK( Rho, T, Y, Rc, P )
        P    = MAX(P, smallpos) !2020/11/30
        ho%uup(ie,icell) = P
        ho%pev(i,icell) = P ! no use?

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

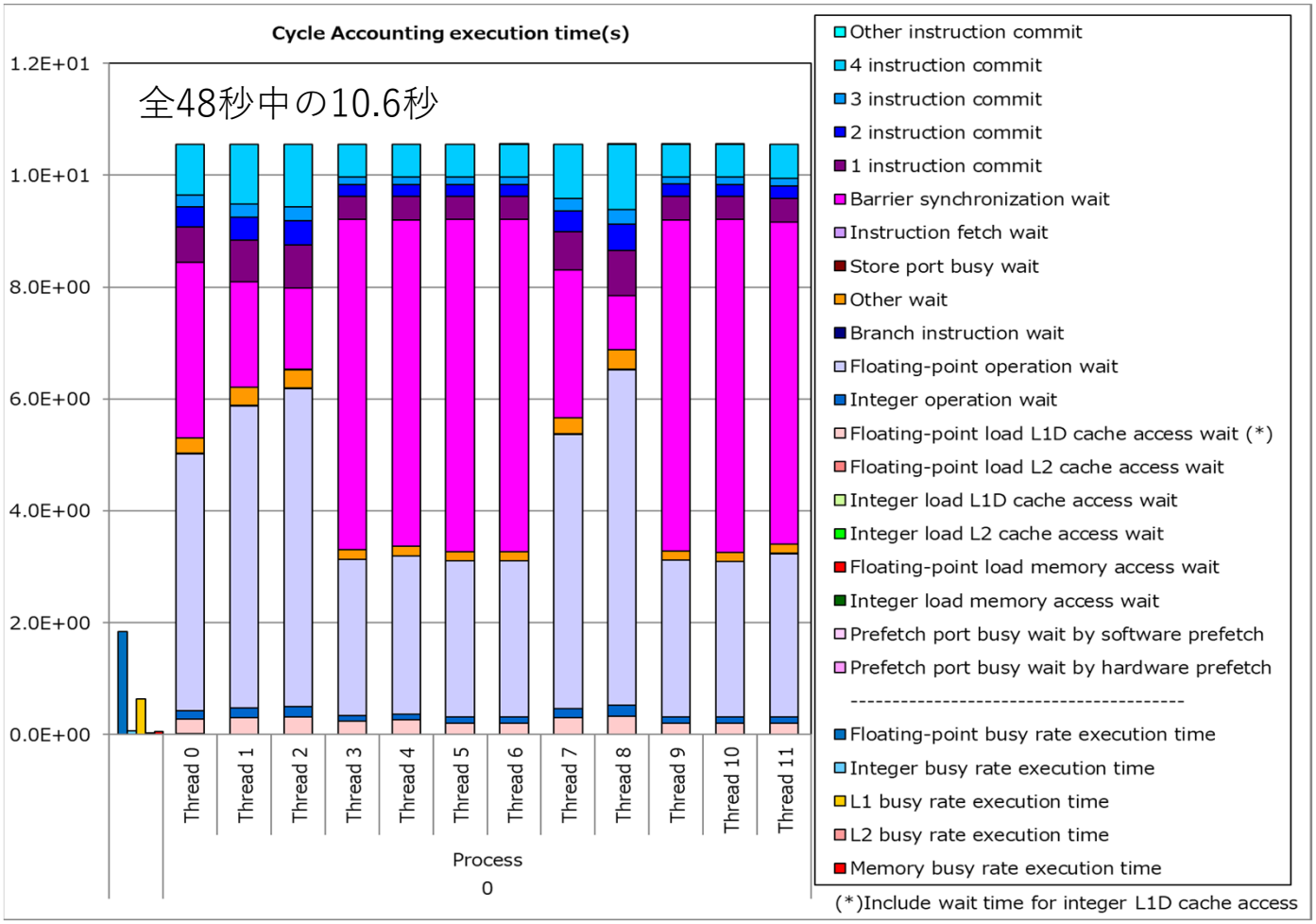
当該部分のコールツリー



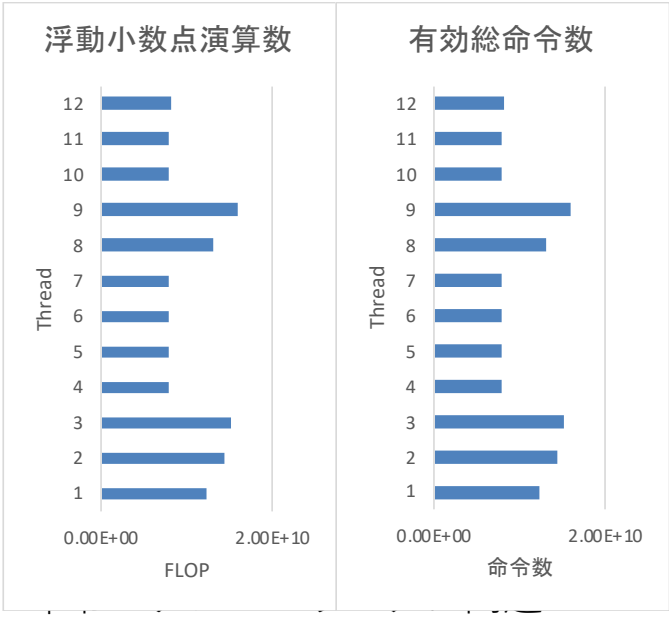
インライン展開された下位ルーチン呼び出し

インライン展開されない下位ルーチン呼び出し

当該部分のCPU性能解析レポート



スレッド毎の浮動小数点演算数と有効総命令数



反復計算部分ソース

```
do iter=1,iter_max

   call ComputeEnergySRK( Rho, T, Y, Rc, ee1 )
   fnc = ee0 - ee1

   call ComputeCpSRK( Rho, T, Y, Rc, Cp )
   call ComputeCvSRK( Rho, T, Y, Cp, Rc, Cv )
   d2adT2 = Cal_d2adT2_SRK( T, X, Rc )
   dfnc = -Cv + T*d2adT2*K1

   T    = T-fnc/dfnc
   T    = MAX(T, small_temp)
   res  = dabs((fnc/dfnc)/T)

   if(res < eps) then
      goto 10001
   end if
end do
```

①
上位の
ComputeTemperaturerルーチンの
OpenMP化ループから呼ばれる
ルーチンなので，スレッドごとに反復
計算している

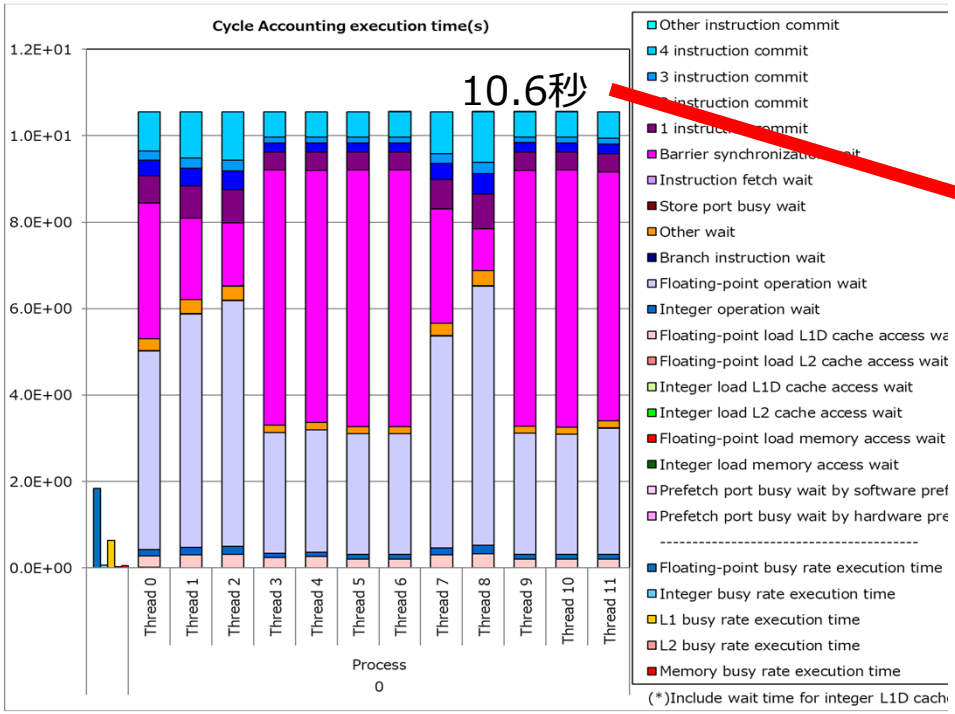
②
**スレッドごとに，抜けるのに
要する反復数が異り
インバランス発生**

③
上位の
ComputeTemperature routineの
OpenMP化ループのスケジュールを変更

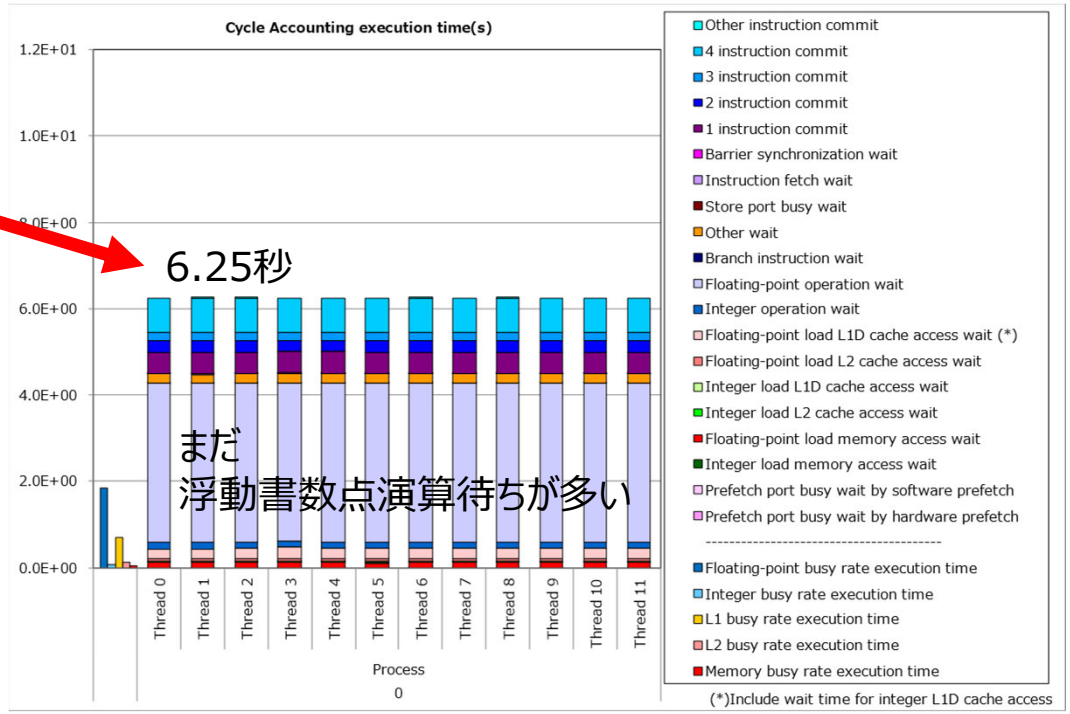
```
!$omp parallel do default(none)&

!$omp parallel do default(none) &
!$omp schedule(dynamic, 1)
```

スレッドスケジュールの修正効果

オリジナル



修正後



※)
ただし，現在のソースでは反復計算を途中で抜けずに，必ず固定回数回るようにしているため，OpenMPのスレッド調整をせずともスレッドのインバランスを回避している

性能について直面した問題 3

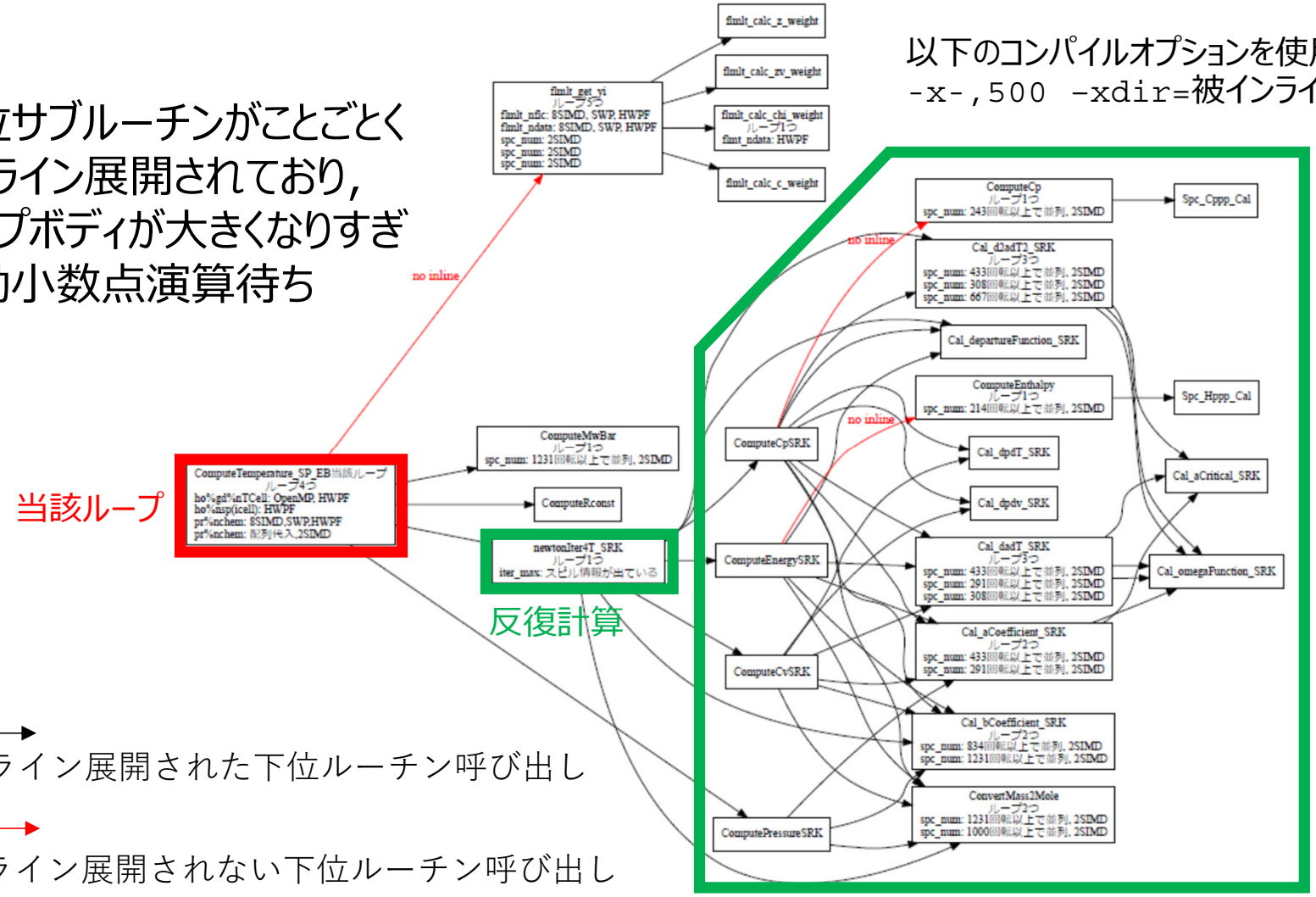
インライン展開

インライン展開が多くループボディが過大になっているケース

再掲) 当該部分のコールツリー

①
下位サブルーチンがことごとく
インライン展開されており，
ループボディが大きくなりすぎ
浮動小数点演算待ち

以下のコンパイルオプションを使用
`-x-,500 -xdir=被インラインソースを置くディレクトリ`



インライン展開された下位ルーチン呼び出し

インライン展開されない下位ルーチン呼び出し

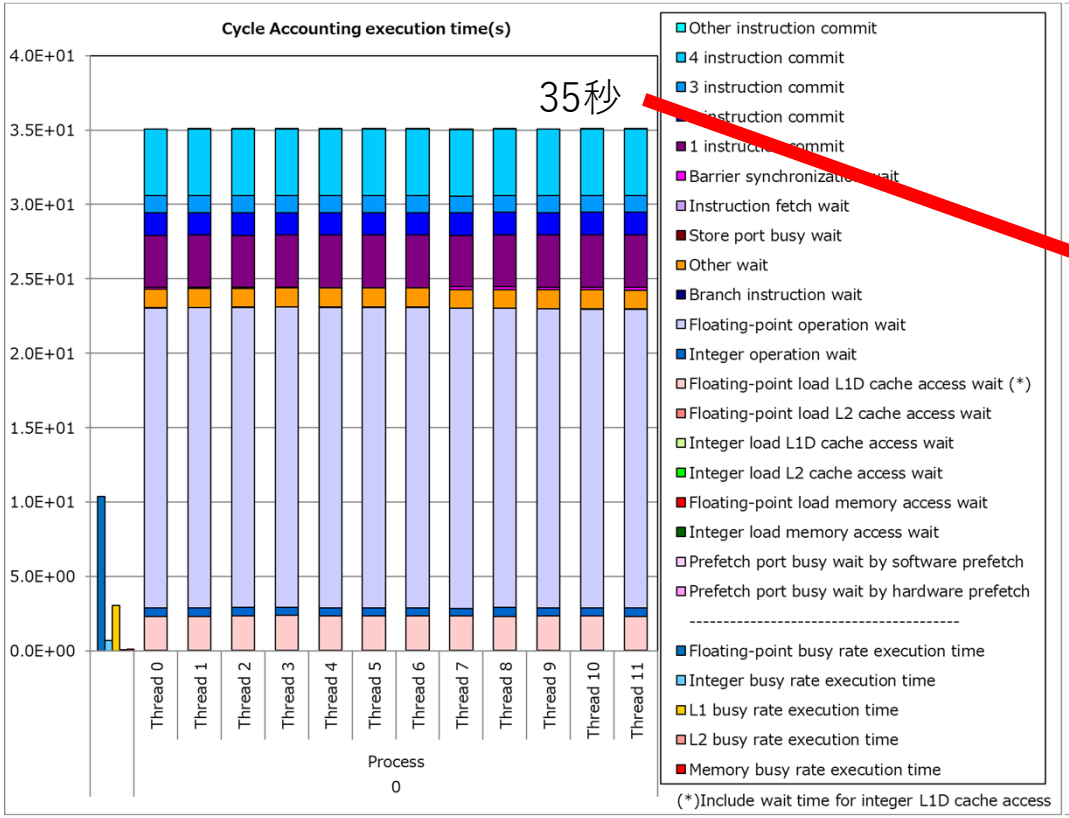
②
緑枠のルーチンを

インライン展開
させない

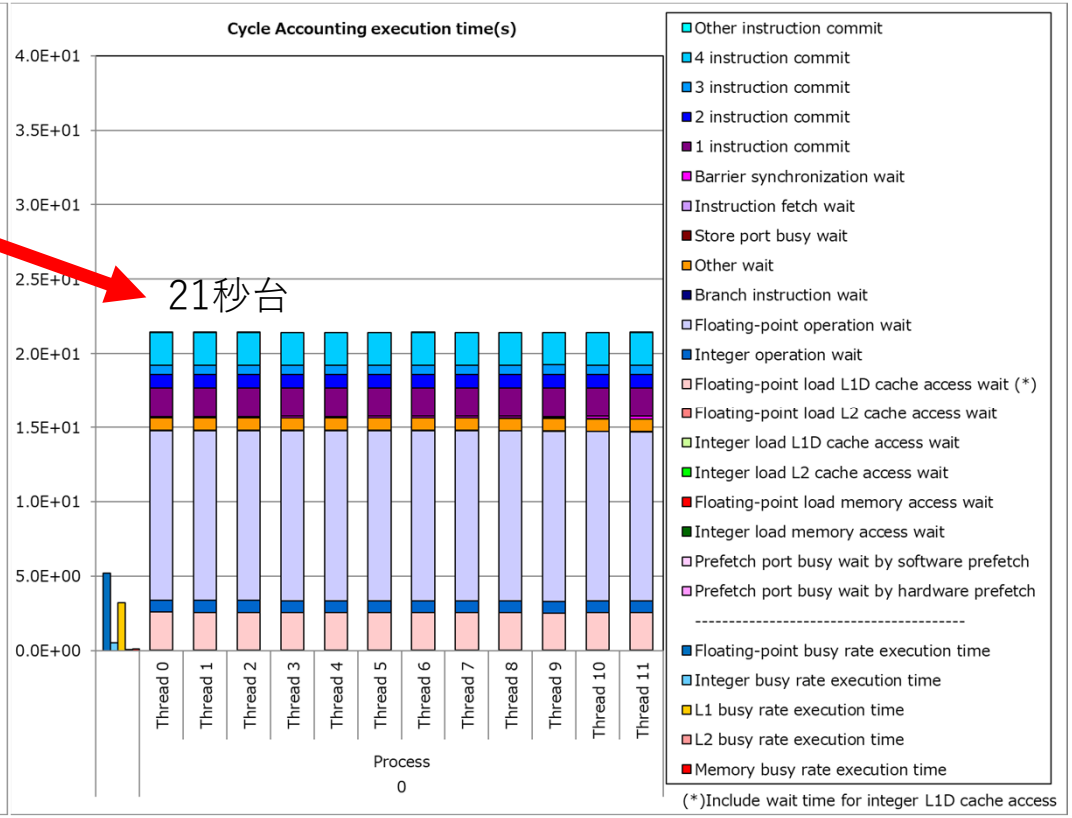
ようにしてみた

インライン展開の抑制の効果

オリジナル



修正後



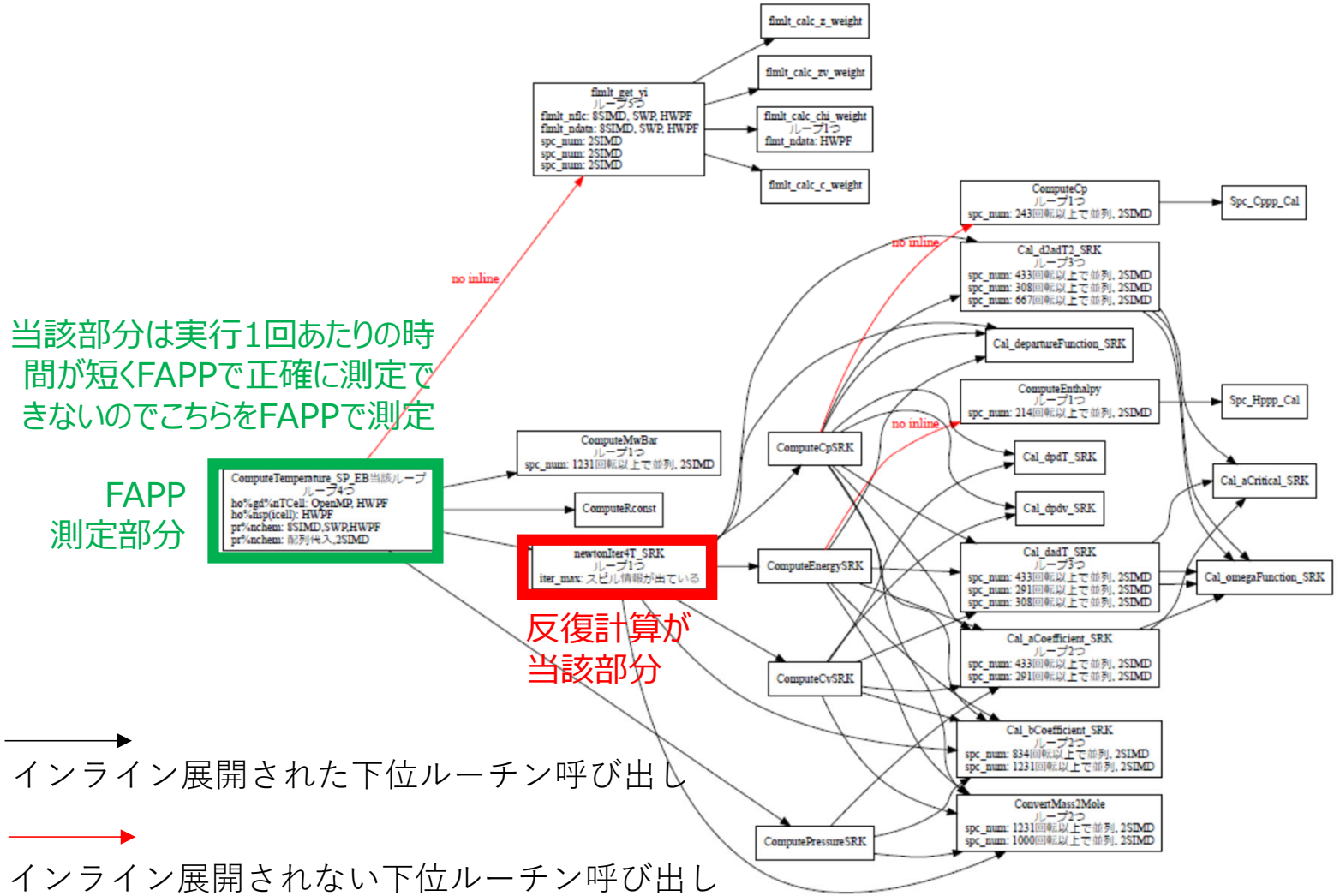
見ている区間は23ページで示したのと同じComputeTemperature_SP_EBだが，ソースのVersionと入力データが異なるため，時間が違う

性能について直面した問題 4

重複した処理

同一ループ内で同じ処理を複数回やっている
ケース

再々掲）当該部分のコールツリー



同じサブルーチンを異るサブルーチンから呼んでいる(計算内容は同じ)結果は同じ）など

重複した計算があり非効率

当該部分ソース

同色の行は同一の処理を意味

```
v =1d0/rho
X =ConvertMass2Mole(Y)
b =Cal_bCoefficient(X, Rc)
K1=Cal_departureFunction_SRK(v, b)

do iter=1,iter_max
   ee1 = ComputeEnergySRK_noinline(Rho, T, Y, Rc)
         Enthalpy0 = ComputeEnthalpy(T, Y)
                 v = 1d0/rho
                 X = ConvertMass2Mole(Y)
                 a = Cal_aCoefficient_SRK(T, X, Rc)
                 b = Cal_bCoefficient_SRK(X, Rc)
              dadT = Cal_dadT_SRK(T, X, Rc)
              K1   = Cal_departureFunction_SRK(v, b)
          return (Enthalpy0 - Rc * T + K1 * (a - dadT*T))

   Cp  = ComputeCpSRK_noinline(Rho, T, Y, Rc)
          Cp0 = ComputeCp(T, T)
            v = 1d0/rho
            X = ConvertMass2Mole(Y)
            a = Cal_aCoefficient_SRK(T, X, Rc)
            b = Cal_bCoefficient_SRK(X, Rc)
         dadT = Cal_dadT_SRK(T, X, Rc)
           K1 = Cal_departureFunction_SRK(v, b)
         dpdT = Cal_dpdT_SRK(v, dadT, b, Rc)
         dpdv = cal_dpdv_SRK(v, T, a, b, Rc)
         return (Cp0-Rc-K1*d2adT2-T*apdT**2/dpdv)
```

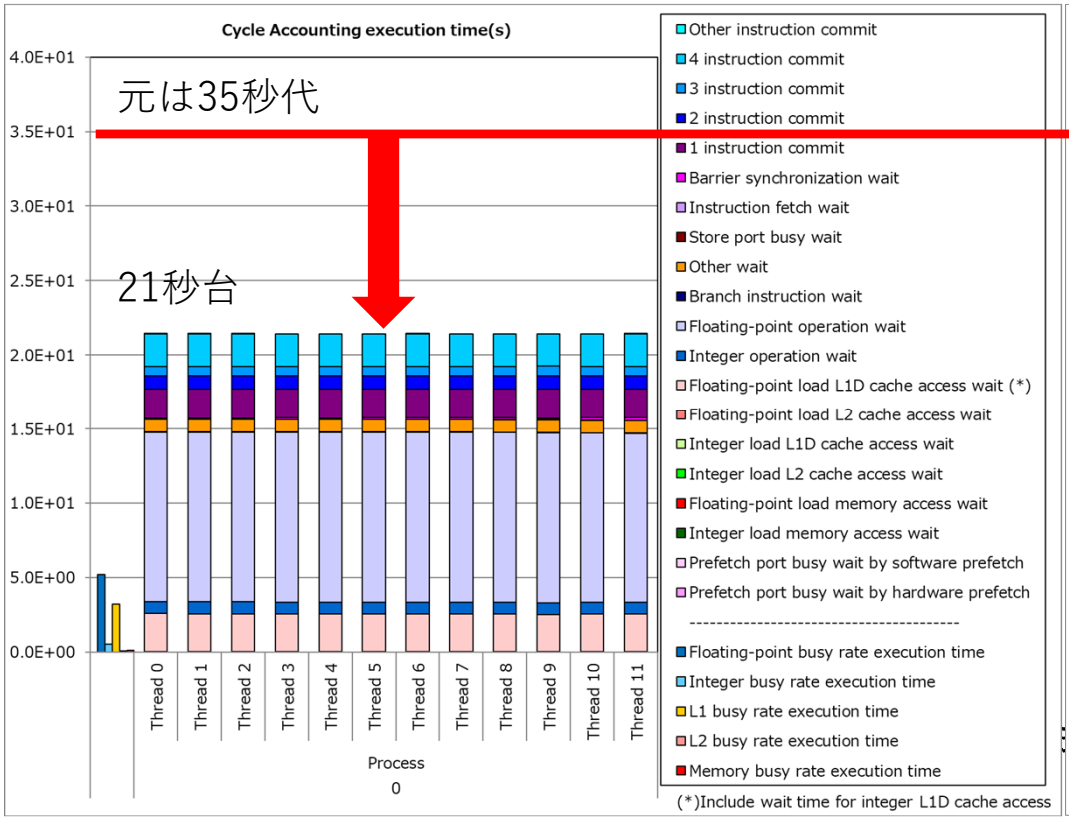
```
Cv = ComputeCvSRK_noinline(Rho, T, Y, Cp, Rc)
        v = 1d0/rho
        X = ConvertMass2Mole(Y)
        a = Cal_aCoefficient_SRK(T, X, Rc)
        b = Cal_bCoefficient_SRK(X, Rc)
     dadT = Cal_dadT_SRK(T, X, Rc)
     dpdT = Cal_dpdT_SRK(v, dadT, b, Rc)
     dpdv = cal_dpdv_SRK(v, T, a, b, Rc)
     return (Cp+T*dpdT**2/dpdv)

   d2adT2 = Cal_d2adT2_SRK(T, X, Rc)
            do i=1,spc_num
                 arci = Cal_aCritical(i, Rc)
                 ai(i) = sqrt(arci)
            enddo
            . . .

   Tの更新
enddo
```

Tに依存しない変数は
iterのループ内で不変

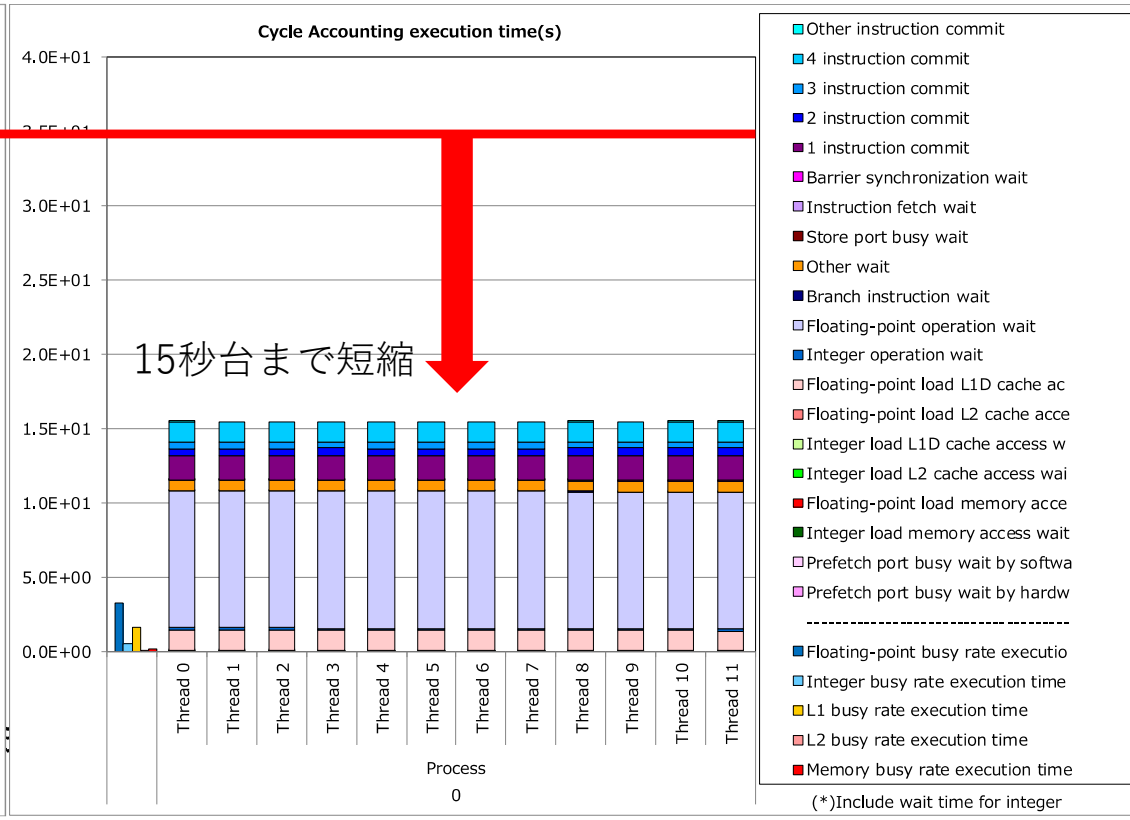
- 図中の□内は，その上の行のサブルーチンの処理内容を抜粋したもの
- 同じ色の行は重複した処理を示している
- また，例えば黄色いハッチングは，温度Tを更新する反復ループないで，Tに依存しないので，ループないから外に出せる

重複処理の統合の効果

オリジナル



修正後

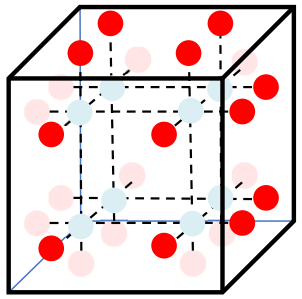


「割り算を逆数の掛け算にする」「サブルーチン内でローカル作業配列を都度作らない」「サブルーチン内で不変な値を先に計算」なども試みたが，ほぼ効果なし

むすび) 性能の観点から見た検討するべき点

- **ソースの多くの箇所で，インライン展開を多用しており，ループボディが大きくなる傾向がある**

- **最内回転数がごく短いループが多く，ループ最適化がうまく作用しない**
  - 物理量の数(密度，流速三成分，エネルギー)
  - 物質の数(水素と酸素など，問題によって異なるがそれほど多くない)
  - セルが持っている点の数(問題によって異なるがそれほど多くない)



```
do i=0, nCell セルで回るループ(多い)
   do j=0, nPoint    セルが持つ点で回るループ
      do k=0, nspc   物理量のループ
         ......
      enddo
   enddo
enddo
```

A64FXシステムアプリ性能検討WG

FUJITSU
shaping tomorrow with you

JAXA)熊畑様で発生したLS-FLOW-HOにおける配列代入の性能問題について

２０２１年９月１５日

ミッションクリティカルシステム事業本部 ＨＰＣシステム事業部
報告者：原口正寿

配列代入の性能問題について

FUJITSU

【課題】 下図のようなプログラムでcallがある場合とない場合で性能差が発生

```
arrA1(:) = AA%array1(:, 1, n2)
arrA2(:) = AA%array1(:, 2, n2)
call timer()
arrB1(:)  = AA%array2(:,1,n2)
arrB2(:)  = AA%array2(:,2,n2)
```

高速

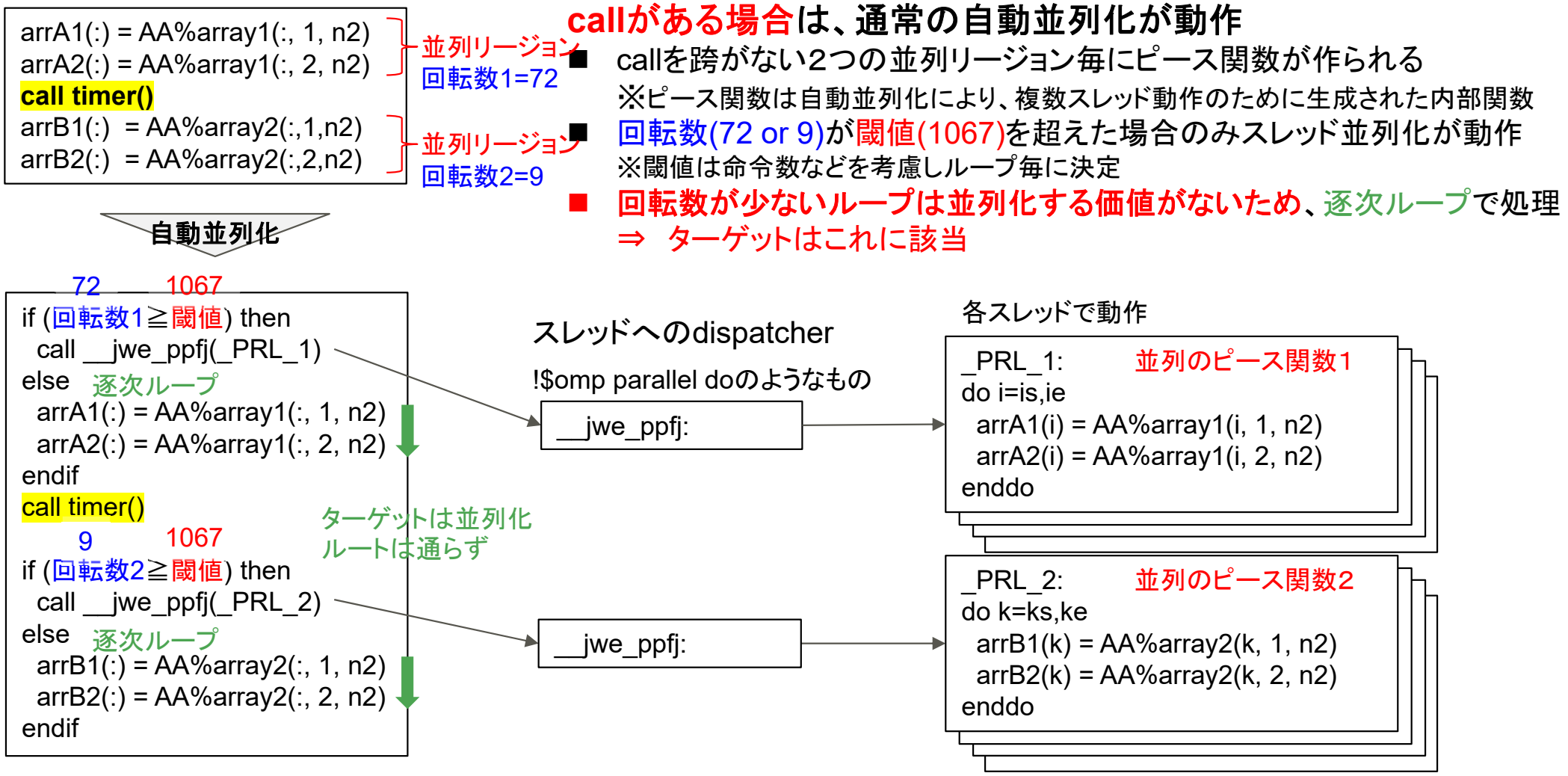
```
arrA1(:) = AA%array1(:, 1, n2)
arrA2(:) = AA%array1(:, 2, n2)
! call timer()
arrB1(:)  = AA%array2(:,1,n2)
arrB2(:)  = AA%array2(:,2,n2)
```

低速

【原因】 callを跨ぐ配列記述が1つに纏められることが原因ではなく、
自動並列化が動作することで、性能低下が生じていました。

本資料では、熊畑様のプログラムで発生している自動並列化の問題点と回避策を提示します。

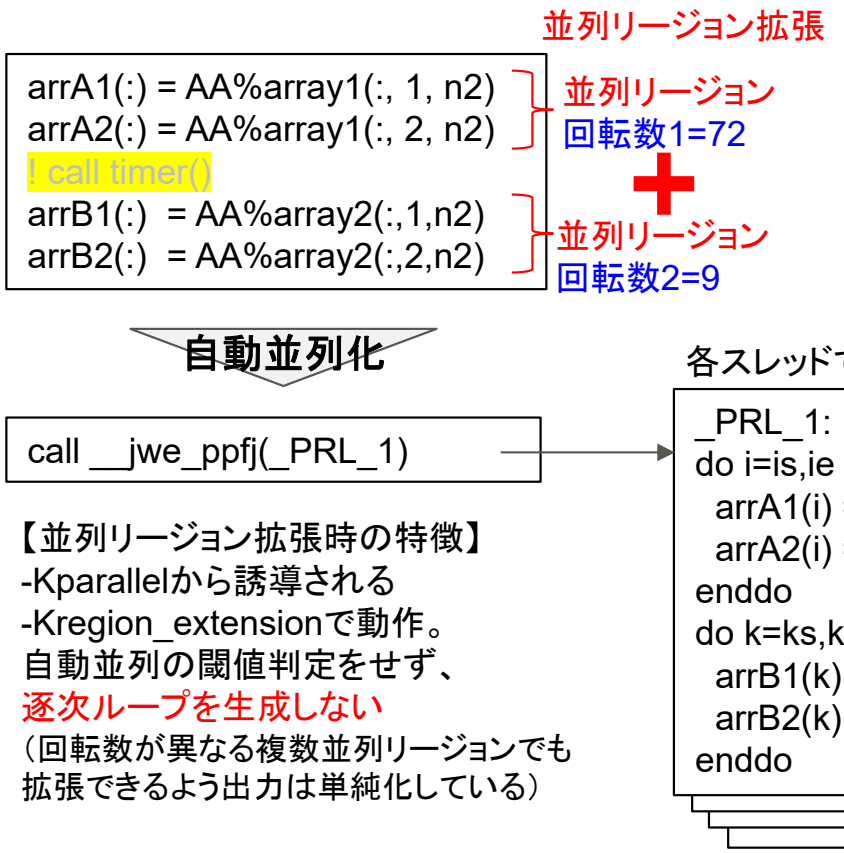
callがあるケース：通常の自動並列化



callがある場合は、通常の自動並列化が動作

- callを跨がない2つの並列リージョン毎にピース関数が作られる
  ※ピース関数は自動並列化により、複数スレッド動作のために生成された内部関数
- 回転数(72 or 9)が閾値(1067)を超えた場合のみスレッド並列化が動作
  ※閾値は命令数などを考慮しループ毎に決定
- **回転数が少ないループは並列化する価値がないため**、逐次ループで処理
  ⇒ ターゲットはこれに該当

callがないケース：並列リージョン拡張



callがない場合は、並列リージョン拡張が動作

- 複数の並列リージョンを1つのピース関数にまとめ、並列化のオーバーヘッドを削減
- 通常の自動並列化のように逐次ループを作らないため、小回転ループで性能低下の可能性あり

⇒ ターゲットはこれに該当

【並列リージョン拡張時の特徴】
-Kparallelから誘導される
-Kregion_extensionで動作。
自動並列の閾値判定をせず、
逐次ループを生成しない
（回転数が異なる複数並列リージョンでも拡張できるよう出力は単純化している）

【結論】
小回転ループに対し、並列リージョン拡張により、スレッド並列化が動作。
並列化のオーバヘッド(主要因)とfalse sharingによる性能低下が発生。

※並列リージョン拡張が動作したことをメッセージやリスタで表示していない問題もあり

【回避策】並列リージョン拡張機能の抑止：-Knoregion_extensionオプションの付加

※この問題に限っては、最適化指示子「!OCL SERIAL」指定のほうが高速

事例）

- OCL指示子の導入による SIMD化の促進
- ループ分割

ABINIT-MP

FMOプログラムABINIT-MPの高速化と超大規模系への対応

○望月祐志[1,2]（ABINIT-MP取り纏め役）
中野達也[3]、坂倉耕太[4]、渡邊啓正[5]、土居英男[1]
片桐孝洋[6]
1：立教大、2：東大生研、3：国立衛研、4：FOCUS
5：HPCシステムズ、6：名大

謝辞：本活動はJHPCNのjh210036-NAH&jh2200010課題とも連動しています

FMO法とABINIT-MPプログラム

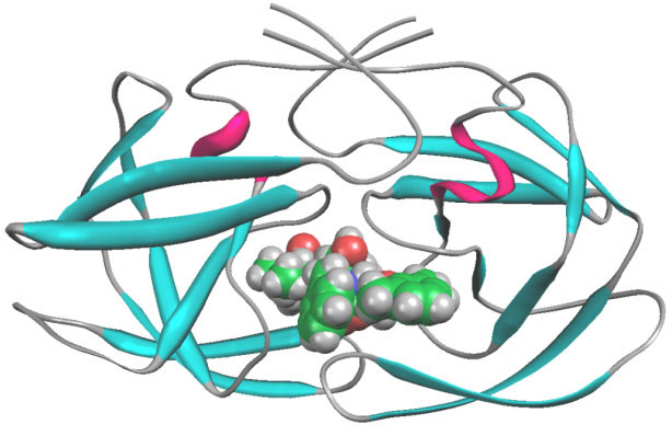
フラグメント分子軌道（FMO）法

◇巨大分子系

生体高分子や凝集系では一般的

⇒ タンパク質、DNA （水和状態）

数千～数万原子、数千～数十万軌道



【HIVプロテアーゼとロピナビル】

◇分割&統合系のアプローチの一つ

北浦らが23年程前に2体展開で提案

⇒ フラグメントとその対で系のエネルギーを評価 （FMO2）

⇒ 環境静電ポテンシャル（ESP）、直接結合切断（BDA）

⇒ 階層的な並列処理（フラグメントリスト&内部処理）

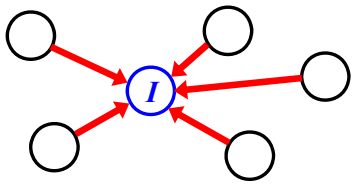
⇒ 定量性を高める電子相関の導入も直截

フラグメント間の相互作用エネルギー（IFIE）

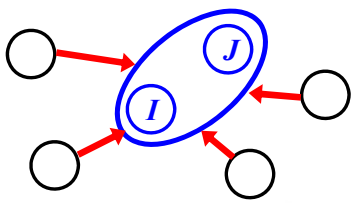
⇒ 計算対象の解析ツール

⇒ 生物物理や創薬に向く

⇒ 材料系にも適用可能



モノマー （アミノ酸単位など）



ダイマー

FMO計算のためのプログラム

◇GAMESS-US [米国Gordonグループ]; Fedorov、Gordon、北浦ら

GAUSSIANに抗し得る有力なフリーソフト、世界規模 (Fortran)

⇒ 様々な機能をFMO化、多彩な計算、GDDI並列

◇ABINIT-MP; 望月、中野ら

実用機能は十分、東大系PJ・CREST-PJなどで開発 (Fortran)

⇒ IOレス、MPI、OpenMP/MPI混成並列、スパコンと好相性

◇PAICS; 石川

FMO-MP2(RI)に特化 (C)

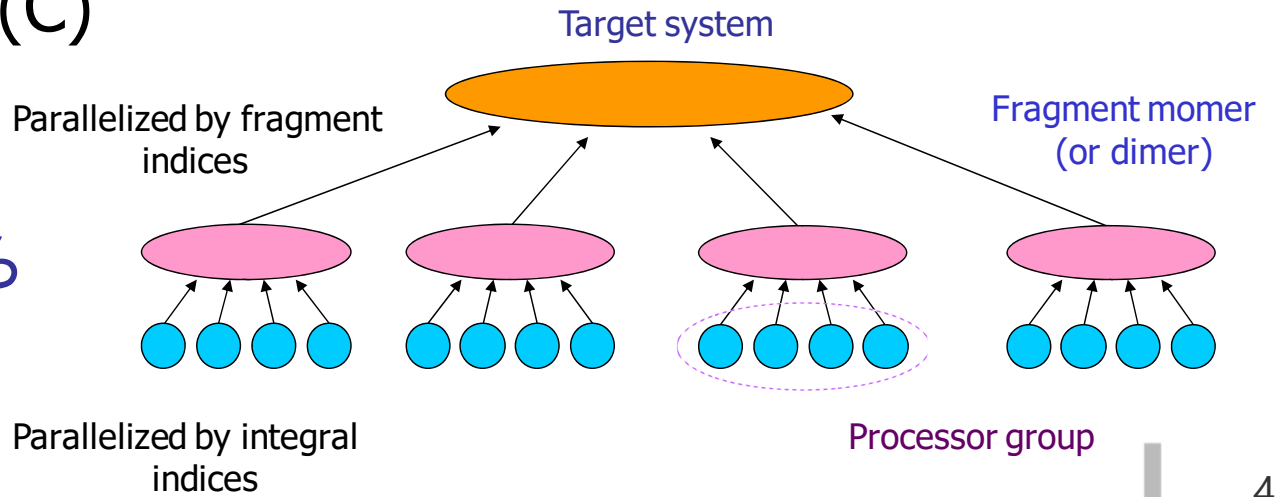
⇒ MPI並列

◇OpenFMO; 稲富、鬼頭ら

FMO-HF (C)

⇒ 超並列指向

2階層の並列処理で計算資源を有効利用



Ref.; “The ABINIT-MP program”, Y. Mochizuki et al. in “*Recent Advances of the Fragment Molecular Method*” (Springer, 2021).

ABINIT-MPの主な機能（オープンシリーズとして整備中）

- エネルギー
  - → FMO4: HF, MP2
  - → FMO2: HF～CCSD(T), LRD
  - → FMO2: CIS/CIS(D)
- エネルギー微分
  - → FMO4: HF, MP2
  - → FMO2: MP2構造最適化, MD
- その他機能
  - → SCIFIE, PB, sp2-BDA, α(ω)
  - → 電子密度生成, CAFI, FILM
- 並列化環境（PC~スパコン）
  - → MPI, OpenMP/MPI混成
  - → 最深部はBLAS処理

http://www.cenav.org/abinit-mp-open_ver-2-rev-4/

Open Ver. 1 （ポスト「京」のPJで整備）
- Rev. 5 (2016年12月)
- Rev. 10 (2018年2月)
- Rev. 15 (2019年3月)
- Rev. 22 (2020年6月)；当面は併存

Open Ver. 2 （「富岳」の時代に移行）
- Rev. 4 (2021年9月)
- Rev. 8 (2023年3月予定)

（注記：Ver. 2系では、BioStation Viewer へのデータファイルの書出しを廃止した）

Ref.; T. Nakano et al., *Chem. Phys. Lett.* **318** (2000) 684. & T. Nakano et al., *Chem. Phys. Lett.* **351** (2002) 475.

基本のHF計算

$$\mathbf{F}^x\mathbf{C}^x=\mathbf{S}^x\mathbf{C}^x\boldsymbol{\varepsilon}^x \qquad \mathbf{F}^x=\mathbf{H}^x+\mathbf{G}^x \quad \Leftrightarrow \text{ HFの一般化固有値問題 （要反復計算）}$$

$$H^x_{\mu\nu}=H^{core\ x}_{\mu\nu}+V^x_{\mu\nu}+\sum_k B_k\langle\mu|\theta_k\rangle\langle\theta_k|\nu\rangle \qquad V^x_{\mu\nu}=\sum_{K\neq x}\left(u^K_{\mu\nu}+v^K_{\mu\nu}\right) \quad \Leftrightarrow \text{ 1電子部分の修飾}$$

$$u^K_{\mu\nu}=\sum_{A\in K}\left\langle\mu\left|\left(-Z_A/|\mathbf{r}-\mathbf{A}|\right)\right|\nu\right\rangle \qquad v^K_{\mu\nu}=\sum_{\lambda\sigma\in K}P^K_{\lambda\sigma}(\mu\nu|\lambda\sigma) \quad \Leftrightarrow \text{ 環境静電ポテンシャル（ESP)}$$

$$P_{\mu\nu}=2\sum_{i=1}^{occ}C^*_{\mu i}C_{\nu i} \qquad G^x_{\mu\nu}=\sum_{\lambda\sigma\in x}P^x_{\lambda\sigma}\left[(\mu\nu|\lambda\sigma)-\frac{1}{2}(\mu\sigma|\lambda\nu)\right] \quad \Leftrightarrow \text{ 2電子部分 O(N}^4\text{)}$$

（並列処理）

高速化のための幾つかの工夫

$$V^L_{\mu\nu}\cong\sum_{\lambda\in L}(\mathbf{P}^L\mathbf{S}^L)_{\lambda\lambda}(\mu\nu,\lambda\lambda) \quad \text{for } R_{\min}(X,L)\geq L_{\text{aoc}} \quad \Leftrightarrow \text{ ESP-AOC近似 （実用精度高し）}$$

$$V^L_{\mu\nu}\cong\sum_{A\in L}\left\langle\mu\left|\left(Q_A/|\mathbf{r}-\mathbf{A}|\right)\right|\nu\right\rangle \quad \text{for } R_{\min}(X,L)\geq L_{ptc} \qquad Q_A=\sum_{\lambda\in A}(\mathbf{P}^L\mathbf{S}^L)_{\lambda\lambda} \quad \Leftrightarrow \text{ ESP-PTC近似 （速い）}$$

$$E'_{IJ}\cong E'_I+E'_J+\mathrm{Tr}\left(\mathbf{P}^I\mathbf{u}^J\right)+\mathrm{Tr}\left(\mathbf{P}^J\mathbf{u}^I\right)+\sum_{\mu\nu\in I}\sum_{\lambda\sigma\in J}\mathbf{P}^I_{\mu\nu}\mathbf{P}^J_{\lambda\sigma}(\mu\nu|\lambda\sigma) \quad \Leftrightarrow \text{ Dimer-ES近似 （HF計算無）}$$

種々の工夫により、FMO2の計算コストのシステムサイズ依存性は2乗より低い

FMO-HF計算の流れ

分子をフラグメントに分割し、電子を割り当てる

↓

モノマーの初期電子密度を計算する

↓

与えられた電子密度を用いて、周囲のモノマーによる環境静電ポテンシャル下での、モノマーのエネルギーと電子密度を計算する

↓

与えられた電子密度と得られた電子密度の差が閾値よりも小さい

No → 与えられた電子密度を用いて、周囲のモノマーによる環境静電ポテンシャル下での、モノマーのエネルギーと電子密度を計算する

Yes

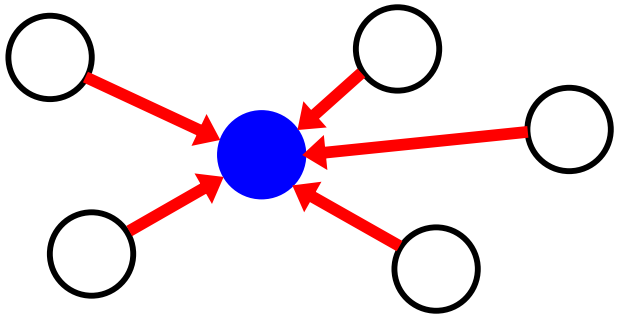
↓

収束したモノマーの電子密度(Self-Consistent Charge; SCC)を用いて、周囲のモノマーによる環境静電ポテンシャル下での、ダイマーのエネルギーと電子密度を計算する

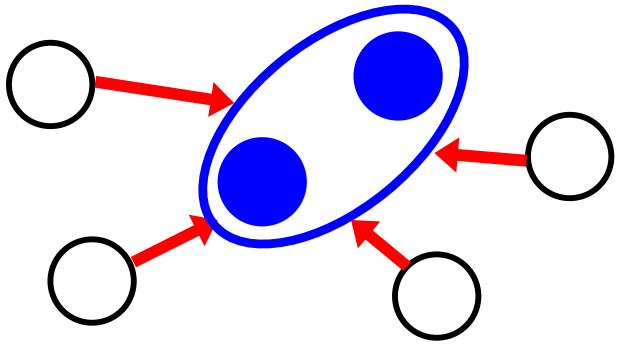
↓

分子の全エネルギーおよび全電子密度を計算

モノマーの段階で自己無撞着電荷を課すのが特徴



モノマー



ダイマー

Ref.; S. Obara et al., *J. Chem. Phys.* **84** (1986) 3963.

2電子積分の生成について#1

- 小原のVertical Recurrence Relation (VRR)がベース
- ジェネレータでF90コード群を自動生成 （微分も）

中野氏

TABLE I. Recurrence expressions[a] for the electron repulsion integrals over *s* and *p* Cartesian Gaussian functions.

$$(ss, ss)^{(0)} = (\zeta+\eta)^{-1/2} K(\zeta_a, \zeta_b, \mathbf{A}, \mathbf{B}) K(\zeta_c, \zeta_d, \mathbf{C}, \mathbf{D})\, F_0(T)$$

$$(p_i s, ss)^{(0)} = (P_i - A_i)(ss, ss)^{(0)} + (W_i - P_i)(ss, ss)^{(1)}$$

$$(p_i s, p_k s)^{(0)} = (Q_k - C_k)(p_i s, ss)^{(0)} + (W_k - Q_k)(p_i s, ss)^{(1)} + \frac{\delta_{ik}}{2(\zeta+\eta)}(ss, ss)^{(1)}$$

$$(p_i p_j, ss)^{(0)} = (P_j - B_j)(p_i s, ss)^{(0)} + (W_j - P_j)(p_i s, ss)^{(1)} + \frac{\delta_{ij}}{2\zeta}\{(ss, ss)^{(0)} - \frac{\rho}{\zeta}(ss, ss)^{(1)}\}$$

$$(p_i p_j, p_k s)^{(0)} = (Q_k - C_k)(p_i p_j, ss)^{(0)} + (W_k - Q_k)(p_i p_j, ss)^{(1)} + \frac{1}{2(\zeta+\eta)}\{\delta_{ik}(sp_j, ss)^{(1)} + \delta_{jk}(p_i s, ss)^{(1)}\}$$

$$(p_i p_j, p_k p_l)^{(0)} = (Q_l - D_l)(p_i p_j, p_k s)^{(0)} + (W_l - Q_l)(p_i p_j, p_k s)^{(1)} + \frac{1}{2(\zeta+\eta)}\{\delta_{il}(sp_j, p_k s)^{(1)} + \delta_{jl}(p_i s, p_k s)^{(1)}\} + \frac{\delta_{kl}}{2\eta}\{(p_i p_j, ss)^{(0)} - \frac{\rho}{\eta}(p_i p_j, ss)^{(1)}\}$$

$$(i, j, k, l = x, y, z)$$

[a] For the definition of the variables, see the text.

SCHEME I.

```
(First step)
    DO ICS = 1,n_CS     loops for the contracted shells
    DO JCS = 1,ICS
    DO IPS = 1,m_ICS    loops for the primitive shells
    DO JPS = 1,m_JCS
    The calculation of the parameters P, ζ, and K(ζ,ζ',R,R')
    for each pair of primitive shells
    CONTINUE

(Second Step)
    DO IPPS = 1,N_PPS   a loop for the first pair of primitive shells
    DO JPPS = 1, IPPS   a loop for the second pair of primitive shells
    The evaluation of ERI's
    CONTINUE
```

- Gauss型関数の角運動量の昇降を利用
- 並列化は短縮シェルの対のループで

$$G_{ijk:\alpha}(r) = N x^i y^k z^l \exp(-\alpha r^2)$$

2電子積分の生成について#2

- 軌道タイプの組み合わせに応じて個別のルーチンに
- 高い軌道角運動量のルーチンの最深部ループは長い
- 微分も同様に組み合わせ毎 （_gradが付く）

sub_dddd.F90	sub_dfsf.F90	sub_dspp.F90	sub_fffs.F90	sub_fsfd.F90	sub_pfdf.F90	sub_ppsp.F90	sub_sdps.F90	sub_sppd.F90
sub_dddf.F90	sub_dfsp.F90	sub_dsps.F90	sub_ffpd.F90	sub_fsff.F90	sub_pfdp.F90	sub_ppss.F90	sub_sdsd.F90	sub_sppf.F90
sub_dddp.F90	sub_dfss.F90	sub_dssd.F90	sub_ffpf.F90	sub_fsfp.F90	sub_pfds.F90	sub_psdd.F90	sub_sdsf.F90	sub_sppp.F90
sub_ddds.F90	sub_dpdd.F90	sub_dssf.F90	sub_ffpp.F90	sub_fsfs.F90	sub_pffd.F90	sub_psdf.F90	sub_sdsp.F90	sub_spps.F90
sub_ddfd.F90	sub_dpdf.F90	sub_dssp.F90	sub_ffps.F90	sub_fspd.F90	sub_pfff.F90	sub_psdp.F90	sub_sdss.F90	sub_spsd.F90
sub_ddff.F90	sub_dpdp.F90	sub_dsss.F90	sub_ffsd.F90	sub_fspf.F90	sub_pffp.F90	sub_psds.F90	sub_sfdd.F90	sub_spsf.F90
sub_ddfp.F90	sub_dpds.F90	sub_fddd.F90	sub_ffsf.F90	sub_fspp.F90	sub_pffs.F90	sub_psfd.F90	sub_sfdf.F90	sub_spsp.F90
sub_ddfs.F90	sub_dpfd.F90	sub_fddf.F90	sub_ffsp.F90	sub_fsps.F90	sub_pfpd.F90	sub_psff.F90	sub_sfdp.F90	sub_spss.F90
sub_ddpd.F90	sub_dpff.F90	sub_fddp.F90	sub_ffss.F90	sub_fssd.F90	sub_pfpf.F90	sub_psfp.F90	sub_sfds.F90	sub_ssdd.F90
sub_ddpf.F90	sub_dpfp.F90	sub_fdds.F90	sub_fpdd.F90	sub_fssf.F90	sub_pfpp.F90	sub_psfs.F90	sub_sffd.F90	sub_ssdf.F90
sub_ddpp.F90	sub_dpfs.F90	sub_fdfd.F90	sub_fpdf.F90	sub_fssp.F90	sub_pfps.F90	sub_pspd.F90	sub_sfff.F90	sub_ssdp.F90
sub_ddps.F90	sub_dppd.F90	sub_fdff.F90	sub_fpdp.F90	sub_fsss.F90	sub_pfsd.F90	sub_pspf.F90	sub_sffp.F90	sub_ssds.F90
sub_ddsd.F90	sub_dppf.F90	sub_fdfp.F90	sub_fpds.F90	sub_pddd.F90	sub_pfsf.F90	sub_pspp.F90	sub_sffs.F90	sub_ssfd.F90
sub_ddsf.F90	sub_dppp.F90	sub_fdfs.F90	sub_fpfd.F90	sub_pddf.F90	sub_pfsp.F90	sub_psps.F90	sub_sfpd.F90	sub_ssff.F90
sub_ddsp.F90	sub_dpps.F90	sub_fdpd.F90	sub_fpff.F90	sub_pddp.F90	sub_pfss.F90	sub_pssd.F90	sub_sfpf.F90	sub_ssfp.F90
sub_ddss.F90	sub_dpsd.F90	sub_fdpf.F90	sub_fpfp.F90	sub_pdds.F90	sub_ppdd.F90	sub_pssf.F90	sub_sfpp.F90	sub_ssfs.F90
sub_dfdd.F90	sub_dpsf.F90	sub_fdpp.F90	sub_fpfs.F90	sub_pdfd.F90	sub_ppdf.F90	sub_pssp.F90	sub_sfps.F90	sub_sspd.F90
sub_dfdf.F90	sub_dpsp.F90	sub_fdps.F90	sub_fppd.F90	sub_pdff.F90	sub_ppdp.F90	sub_psss.F90	sub_sfsd.F90	sub_sspf.F90
sub_dfdp.F90	sub_dpss.F90	sub_fdsd.F90	sub_fppf.F90	sub_pdfp.F90	sub_ppds.F90	sub_sddd.F90	sub_sfsf.F90	sub_sspp.F90
sub_dfds.F90	sub_dsdd.F90	sub_fdsf.F90	sub_fppp.F90	sub_pdfs.F90	sub_ppfd.F90	sub_sddf.F90	sub_sfsp.F90	sub_ssps.F90
sub_dffd.F90	sub_dsdf.F90	sub_fdsp.F90	sub_fpps.F90	sub_pdpd.F90	sub_ppff.F90	sub_sddp.F90	sub_sfss.F90	sub_sssd.F90
sub_dfff.F90	sub_dsdp.F90	sub_fdss.F90	sub_fpsd.F90	sub_pdpf.F90	sub_ppfp.F90	sub_sdds.F90	sub_spdd.F90	sub_sssf.F90
sub_dffp.F90	sub_dsds.F90	sub_ffdd.F90	sub_fpsf.F90	sub_pdpp.F90	sub_ppfs.F90	sub_sdfd.F90	sub_spdf.F90	sub_sssp.F90
sub_dffs.F90	sub_dsfd.F90	sub_ffdf.F90	sub_fpsp.F90	sub_pdps.F90	sub_pppd.F90	sub_sdff.F90	sub_spdp.F90	sub_ssss.F90
sub_dfpd.F90	sub_dsff.F90	sub_ffdp.F90	sub_fpss.F90	sub_pdsd.F90	sub_pppf.F90	sub_sdfp.F90	sub_spds.F90	sub_xxxx.F90
sub_dfpf.F90	sub_dsfp.F90	sub_ffds.F90	sub_fsdd.F90	sub_pdsf.F90	sub_pppp.F90	sub_sdfs.F90	sub_spfd.F90	suberi.F90
sub_dfpp.F90	sub_dsfs.F90	sub_fffd.F90	sub_fsdf.F90	sub_pdsp.F90	sub_ppps.F90	sub_sdpd.F90	sub_spff.F90	swz_init.F90
sub_dfps.F90	sub_dspd.F90	sub_ffff.F90	sub_fsdp.F90	sub_pdss.F90	sub_ppsd.F90	sub_sdpf.F90	sub_spfp.F90	
sub_dfsd.F90	sub_dspf.F90	sub_fffp.F90	sub_fsds.F90	sub_pfdd.F90	sub_ppsf.F90	sub_sdpp.F90	sub_spfs.F90	

1st Refs.; Y. Mochizuki et al., *Chem. Phys. Lett.* **396** (2004) 473. & Y. Mochizuki et al., *Theor. Chem. Acc.* **112** (2004) 442.
2nd Ref.; Y. Mochizuki et al., *Chem. Phys. Lett.* **457** (2008) 396.

相関エネルギー補正のMP2のアルゴリズム2種

Loop over *i*-batch [parallelizable when needed]
  Loop over σ [to be parallelized for worker processes]
    Loop over λ
      Generate ($\mu\nu$,$\lambda[\sigma]$) list [canonical relation for μν]
      Do 1/4 transformation of $\mu \rightarrow i$ [screening & DAXPY]
      Do 2/4 transformation of $\nu \rightarrow a$ [DDOT]
      Do 3/4 transformation of $\lambda \rightarrow j$ [screening & DAXPY]
    End of loop over λ
    Do 4/4 transformation for $\sigma \rightarrow b$ [screening & DAXPY]
  End of loop over σ ["all-reduce" must be done for $(ia|jb)$]
  Calculate partial MP2 energy
End of loop over *i*-batch

Loop over *ij*-batch ! size depending on available memory
  Loop over σ ! to be parallelized
    Loop over λ
      Preparing ($\mu\nu|\lambda\sigma$) ! for canonical $\mu\nu$-pair
      Forming ($i\nu|\lambda\sigma$) ! DGEMM, fixed $\lambda\sigma$, running over μ
      Forming ($ia|\lambda\sigma$) ! DGEMM, fixed $\lambda\sigma$, running over ν
      Forming ($ia|j\sigma$) ! DGEMM, fixed σ, direct-product for fixed λ
    End of loop over λ
    Forming ($ia|jb$) ! DGEMM, direct-product for fixed σ
  End of loop over σ ! all-reduce operation as barrier
  Calculate partial MP2 energy with respect to *ij*-batch
End of loop over *ij*-batch

MP2相関エネルギー補正

$$E_{\mathrm{MP2}} = \sum_{ijab} \frac{(ia \mid jb)\left[2(ia \mid jb) - (ib \mid ja)\right]}{\varepsilon_i + \varepsilon_j - \varepsilon_a - \varepsilon_b}$$

添字の変換：$4N^5$コスト

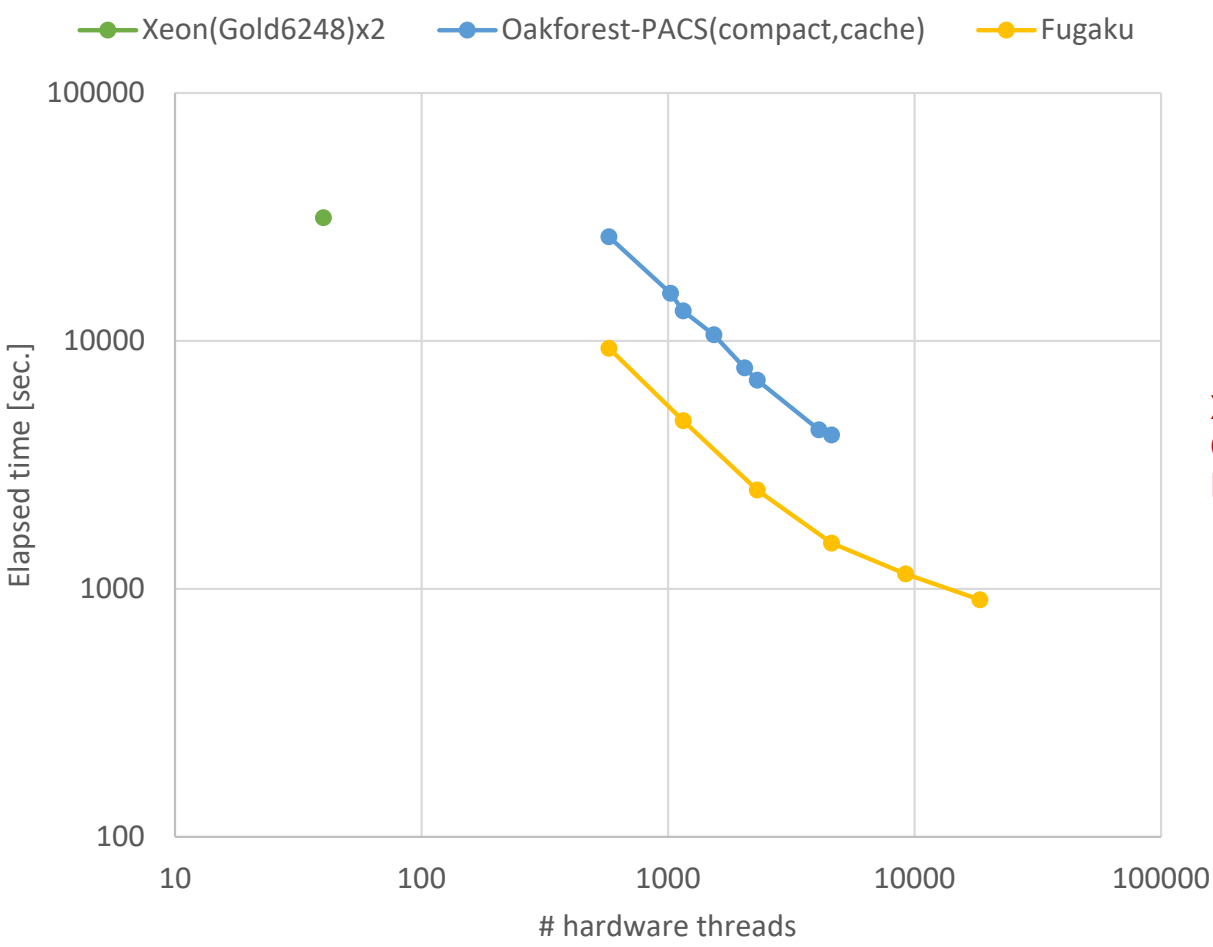
$$(ia \mid jb) = \sum_{\sigma} C_{\sigma b} \left(\sum_{\lambda} C_{\lambda j} \left(\sum_{\nu} C_{\nu a} \left(\sum_{\mu} C_{\mu i} (\mu\nu \mid \lambda\sigma) \right) \right) \right)$$

【第一版】
- DAXPYとDDOTを使い、閾値判断を優先
- 実効的演算数を下げる方針、15年程前のチップでは有効

【第二版】
- DGEMMの高性能に期待
- 2段目と4段目をDGEMM処理がデフォルト
- 最近のスパコンでは4段ともDGEMMが最も速い

FMO-MP2/6-31G*ジョブのスケーリング

Ver. 1 Rev. 22を使用



Xeon; 40 cores
OFP; up to 4608 cores
Fugaku; up to 18432 cores

September 2020

- PDB ID: 6LU7 = SARS-CoV-2 Mpro + N3 ligand の系
- MP2の積分変換は全てDGEMMで実行
- Dimer-ESのCMM近似は (>5のリージョンで使用)
- 「富岳」はOakforest-PACSよりも2.8倍ほど速い

改良の第一弾：Ver. 2 Rev. 4

2020年度の「富岳」の利用で認識したポイント

■プログラム改良の必要性

・高速化と大規模化
  ⇒ 多数のサンプル構造の扱い （Ver. 1 Rev. 22に比して数倍を目標）
  ⇒ 大型の水和モデルの扱い （水を含めて数万フラグメントを視野）

・想定プラットフォーム
  ⇒ 「富岳」を頂点とする富士通A64FX系のスパコン群
  ⇒ NEC SX-Aurora TSUBASAやIntel Xeonのスパコン群

・BioStation ViewerのIFは不要
  ⇒ 数千フラグメント系はフツーのPCでは利用不可 （メモリ、GPU的に）
  ⇒ CPFのための情報配列の保持が重い （フラグメント数の自乗依存性）

・機能追加
  ⇒ 統計評価/機械学習向けの記述子を含むデータの出力
  ⇒ より詳細な相互作用解析と励起状態系のプロパティの評価

■HPCI拠点でのABINIT-MPのライブラリ整備

・新規感染症発生時の対応
  ⇒ 多数の拠点で同時並行的に解析

Ver. 2系の整備開発

■Ver. 2 Rev. 4（2021年9月16日リリース済）

・高速化（A64FX向け）
　⇒ 2電子積分生成のSIMD化、MPI通信量の削減、プリント量の抑制
　⇒ Ver. 1 Rev. 22比で1.2～1.5倍の加速 （MP2レベル、系と基底に依存）
・大規模系の対応
　⇒ 結果データ可視化用の配列を削除 （BioStation ViewerのIFを廃止）
　⇒ 1.1万フラグメントのタンパク質の液滴モデルが扱い可能 （MP3レベル）
・機能の追加
　⇒ 分極率の算定、多層近似での領域限定相関計算、機械学習データのダンプ

■Ver. 2 Rev. 8（2023年3月リリースの予定）

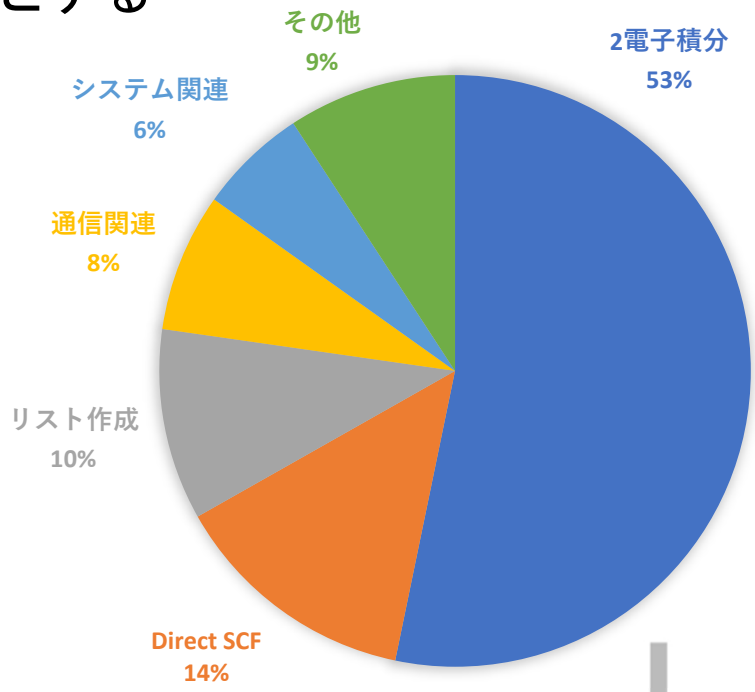
・高速化（A64FX向け）
　⇒ 積分周りの改良の継続、Fock行列構築のif分岐除去、モノマーSCCの加速
　⇒ 作業版では1.5倍～1.7倍の加速を達成
・大規模系の対応
　⇒ 液滴モデルで1.4万フラグメントの扱いを目標 （MP2/6-31G*）
・機能の追加
　⇒ PIEDAの詳細化 （分散安定化の分離、静電相互作用の再評価など）
　⇒ 励起エネルギーとイオン化エネルギーの算定 （領域限定）

A64FXでのコスト分析 （Ver. 1 Rev. 22）

・Ala_9GlyのFMO-MP2/6-31G*のテストジョブ
・12スレッド8プロセス（2ノード実行:FX1000）

- プログラム全体のコスト分布
  - 基本プロファイラによるプロセス0番、スレッド0番のコスト分布
  - 2電子積分処理が全体の約半分を占める
    ただし、81種の処理の総和であるため、
    1種あたりのコストは1%前後と非常に小さい
  - 通信に関連したコストは8%程度と小さい
  - 性能改善に向けたソース分析は以下を対象とする
    - 2電子積分
    - Direct SCF
    - リスト作成

2電子積分：81種のサブルーチン(sub_*)のコスト総和
Direct SCF：サブルーチンdirect_scf_gmatのコスト
リスト作成：3種のサブルーチン(get_tei_rs_fix, get_tei_pq_fix, get_ixijcs_to_proc_pqfix)のコスト総和
通信関連：通信に関連した処理(putofu_*, opal_*, mca_*)のコスト総和
システム関連：ライブラリやOSなどに関連した処理のコスト総和
その他：上記以外の処理の総和



2電子積分の生成の改善#1

- 改良指針（井上G@富士通の助言）
  - **OCL指示詞の導入によるSIMD化の促進、一部スカラ変数化も必要**
  - **コンパイラオプションの変更**

- リファレンス
  - オリジナルコード、Ala_9GlyのMP2ジョブ
  - 2ノード実行、12スレッド（OpenMP）×8プロセス（MPI）
  - コンパイラオプション：-O3 -Knosimd -Koptmsg=2 –V
  - 6-31G*//cc-pVZ： 153.0s/134.5s//337.4s/306.1s (MP2;AXPY/GEMM)

- 手動でのSIMD化と結果（その１）
  - オリジナルコード＋SSSS, PSSS, SPSS, SSPS, SSSP, PPSS, PSPS, PSSP, SPPS, SPSP, SSPP, DSSS, SDSS, SSDS, SSSD（スカラ変数化）
  - OCL指示詞の追加
  - コンパイラオプション：-O3 -Knosimd -Kocl
  - 6-31G*//cc-pVZ：142.5s/124.5s(7.4%)//294.4s/254.0s(17.0%)

  (cc-pVDZの場合、オリジナルコードと比較して全体で17.0%の高速化)

2電子積分の生成の改善#2

- 手動での最適化と結果（その2）
  - コンパイラオプション：-Kfast-Knosimd -Kocl
  - 6-31G*/cc-pVZ：116.2s(24.1%)/239.6s(29.0%)（DGEMMで4段のみ）
  - 6-31G*/cc-pVZ：78.3s(48.8%)/165.1s(51.1%)（Buffered direct SCF）

- 補足説明
  - SIMD化はOCL指示詞の挿入だけで足らず、スカラ変数化なども必要
  - 地道な修正作業を多くのルーチンで実施
    （Intel Xeon系では加速の効果は視られず）
  - Buffered direct SCFでは生成した2電子積分を可能な限りメモリに保持
    （積分計算の回数自体を減らす/メモリ管理はシビアでやや不安定）
  - MP2では積分をバッファするよりも変換用の作業配列に割り当てるべき
    （バッチ数が増えると積分計算の回数が増える）

SIMD化した積分ルーチンの例(sssp)

```
subroutine sub_sssp(zetam,pm,dkabm,etam,qm,dkcdm, &
     ma,mb,mc,md,ngij,ngkl,a,b,c,d,sint,tv)
  !
  !        Nov.05,'02
  !                T.NAKANO & Y. ABE
  !
  use constant
  use auxiliary_integral_table
  use integral_parameter
  implicit none
  real(8),intent(in)::zetam(*),pm(3,*),dkabm(*), &
       etam(*),qm(3,*),dkcdm(*)
  integer,intent(in)::ma,mb,mc,md,ngij,ngkl
  real(8),intent(in)::a(3),b(3),c(3),d(3),tv
  real(8),intent(out)::sint(*)
  !-----------------------------------------
  integer npq,nrs,ix
  real(8) p(3),q(3),qd(3),pq(3),wq(3),f(0:max_m), &
       dkab,zeta,dkcd,eta,ze,rz,re,rho,a0,tt
  integer ts,i,j,k,l,m
  real(8) delta,t_inv
  real(8) ssss(0:1),f0,f1,qd1,qd2,qd3,wq1,wq2,wq3

  sint(1:3) = 0.0_8
```

OCL指示子

```
!ocl eval
!ocl fp_relaxed
!ocl fp_contract
!ocl noswp
!ocl eval_concurrent
!ocl SIMD
```

```
do npq=1,ngij
  if (abs(dkabm(npq)) > tv) then
    do nrs=1,ngkl
      if (abs(dkabm(npq)*dkcdm(nrs)) > tv) then
        ze  = 1.0_8/(zetam(npq)+etam(nrs))
        a0  = dkabm(npq)*dkcdm(nrs)*sqrt(ze)
        rz  = etam(nrs)*ze
        re  = zetam(npq)*ze
        rho = zetam(npq)*rz

        do i=1,3
!         qd(i) = qm(i,nrs)-d(i)
          pq(i) = qm(i,nrs)-pm(i,npq)
!         wq(i) =-re*pq(i)
        end do

        qd1 = qm(1,nrs)-d(1)
        qd2 = qm(2,nrs)-d(2)
        qd3 = qm(3,nrs)-d(3)
        wq1 =-re*pq(1)
        wq2 =-re*pq(2)
        wq3 =-re*pq(3)
```

スカラ変数化

以下、次頁

SIMD化した積分ルーチンの例(続き)

```
          tt = (pq(1)*pq(1)+pq(2)*pq(2)+pq(3)*pq(3))*rho

          if (tt <= 38.0_8) then  ! Tf = 2*m+36 (for m=1)
            ts    = 0.5_8+tt*fmt_inv_step_size
            delta = ts*fmt_step_size-tt

!           f(0) = ((fmt_table(3,ts)*inv6*delta  &
!                +   fmt_table(2,ts)*inv2)*delta &
!                +   fmt_table(1,ts))*delta      &
!                +   fmt_table(0,ts)
!           f(1) = ((fmt_table(4,ts)*inv6*delta  &
!                +   fmt_table(3,ts)*inv2)*delta &
!                +   fmt_table(2,ts))*delta      &
!                +   fmt_table(1,ts)
            f0   = ((fmt_table(3,ts)*inv6*delta  &
                 +   fmt_table(2,ts)*inv2)*delta &
                 +   fmt_table(1,ts))*delta      &
                 +   fmt_table(0,ts)
            f1   = ((fmt_table(4,ts)*inv6*delta  &
                 +   fmt_table(3,ts)*inv2)*delta &
                 +   fmt_table(2,ts))*delta      &
                 +   fmt_table(1,ts)
          else
            t_inv = inv2/tt
!           f(0) = sqrt(pi_over2*t_inv)
!           f(1) = t_inv*f(0)
            f0   = sqrt(pi_over2*t_inv)
            f1   = t_inv*f0
          end if
```

```
!-------------------------------------------
!  ERI code generator Ver.20020228
!  2002/02/28
!  T. Nakano
!
!  (sssp)
!
!         ssss(0:1)=f(0:1)*a0
          ssss(0)=f0*a0
          ssss(1)=f1*a0
!         do l=1, 3
!           sint(l) = sint(l)+qd(l)*ssss(0)+wq(l)*ssss(l)
!         end do
          sint(1) = sint(1)+qd1*ssss(0)+wq1*ssss(1)
          sint(2) = sint(2)+qd2*ssss(0)+wq2*ssss(1)
          sint(3) = sint(3)+qd3*ssss(0)+wq3*ssss(1)
!-------------------------------------------
        end if
      end do
    end if
  end do
end subroutine sub_sssp
```

Ver. 2 Rev. 4での速度向上の例#1

HIV-1 protease / FMO-MP2/6-31G* / Benchmark 100 nodes @ Fugaku

Ver. 1 Rev. 22

```
====================
   ## TIME PROFILE
====================

        Elapsed time: Monomer SCF      =          452.4 seconds
        Elapsed time: Monomer MP2      =           17.4 seconds
        Elapsed time: Monomer (Total)  =          472.7 seconds
        Elapsed time: Dimer ES         =           99.7 seconds
        Elapsed time: Dimer SCF        =          278.6 seconds
        Elapsed time: Dimer MP2        =          269.1 seconds
        Elapsed time: Dimer (Total)    =          695.3 seconds
        Elapsed time: FMO (Total)      =         1168.0 seconds

## Time profile

Number of cores (total)    =    200
Number of cores (fragment) =      1

THREADS (FRAGMENT)         =     24


Total time =          1172.8 seconds
```

Ver. 2 Rev. 4

```
====================
   ## TIME PROFILE
====================

        Elapsed time: Monomer SCF      =          354.7 seconds
        Elapsed time: Monomer MP2      =           16.0 seconds
        Elapsed time: Monomer (Total)  =          373.4 seconds
        Elapsed time: Dimer ES         =          109.5 seconds
        Elapsed time: Dimer SCF        =          221.7 seconds
        Elapsed time: Dimer MP2        =          242.4 seconds
        Elapsed time: Dimer (Total)    =          673.4 seconds
        Elapsed time: FMO (Total)      =         1046.8 seconds

## Time profile

Number of cores (total)    =    200
Number of cores (fragment) =      1

THREADS (FRAGMENT)         =     24


Total time =          1050.2 seconds
```

- Ver. 2 Rev. 4はA64FX向け積分SIMD化、「不要配列」の整理などを反映済み
- より大型の系ではMP2ジョブで2-5割程度の速度向上
- cc-pVDZの方が短縮長が長いために加速効果が出やすい （他系でも評価）

Ver. 2 Rev. 4での速度向上の例#2

6VXX / FMO-MP2/6-31G* / Benchmark 8 racks @ Fugaku

Ver. 1 Rev. 22

```
  ====================
     ## TIME PROFILE
  ====================

        Elapsed time: Monomer SCF      =          2028.7 seconds
        Elapsed time: Monomer MP2      =            15.0 seconds
        Elapsed time: Monomer (Total)  =          2068.6 seconds
        Elapsed time: Dimer ES         =           353.9 seconds
        Elapsed time: Dimer SCF        =           362.4 seconds
        Elapsed time: Dimer MP2        =           302.6 seconds
        Elapsed time: Dimer (Total)    =          1603.4 seconds
        Elapsed time: FMO (Total)      =          3672.1 seconds

## Time profile

Number of cores (total)    =   3072
Number of cores (fragment) =      1

THREADS (FRAGMENT)         =     48


Total time =          3759.3 seconds
```

Ver. 2 Rev. 4

```
  ====================
     ## TIME PROFILE
  ====================

        Elapsed time: Monomer SCF      =          1801.6 seconds
        Elapsed time: Monomer MP2      =            14.2 seconds
        Elapsed time: Monomer (Total)  =          1839.1 seconds
        Elapsed time: Dimer ES         =           314.2 seconds
        Elapsed time: Dimer SCF        =           335.7 seconds
        Elapsed time: Dimer MP2        =           294.6 seconds
        Elapsed time: Dimer (Total)    =          1188.5 seconds
        Elapsed time: FMO (Total)      =          3027.7 seconds

## Time profile

Number of cores (total)    =   3072
Number of cores (fragment) =      1

THREADS (FRAGMENT)         =     48


Total time =          3090.8 seconds
```

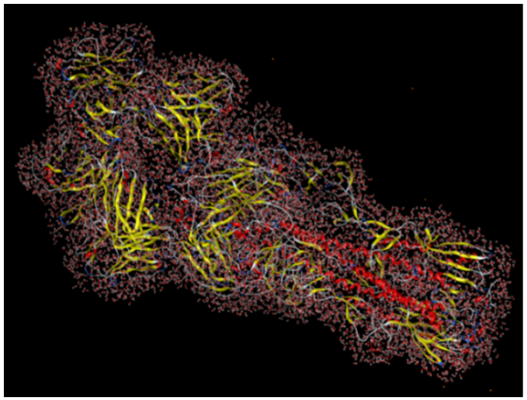
- Ver. 2 Rev. 4はA64FX向け積分SIMD化、「不要配列」の整理などを反映済み
- 対Ver. 1 Rev.22で1.2倍の加速 （cc-pVDZ; 8769.9秒→6356.6秒で1.4倍）

Ref.; 望月ら, *J. Comp. Chem. Jpn.* **20** (2021) 132.

超大規模系への対応例

従前の2倍の系が計算可能に

- インフルHA+Fab抗体×2(PDB id: 1KEN)の水和モデル
- フラグメント総数は11307、水と対イオンを含む
- 「不老」の1ラック、FMO-MP2/cc-pVDZは9.2時間で完走
- 「富岳」の8ラック、FMO-MP3/cc-pVDZは6.7時間で完走
- FMO-MP2ではモノマーSCCが半分弱のコスト ⇒ 要対応
- ダイマー部分で「謎の時間」が顕在化 ⇒ 要対応
- 水クラスターでは2万フラグメントのMP2ジョブも完走確認



```
====================
   ## TIME PROFILE
====================

        Elapsed time: Monomer SCF      =        14546.6 seconds
        Elapsed time: Monomer MP2      =           32.5 seconds
        Elapsed time: Monomer (Total)  =        14741.5 seconds
        Elapsed time: Dimer ES         =         4021.8 seconds
        Elapsed time: Dimer SCF        =         7215.9 seconds
        Elapsed time: Dimer MP2        =         2492.4 seconds
        Elapsed time: Dimer (Total)    =        18240.6 seconds
        Elapsed time: FMO (Total)      =        32982.1 seconds

## Time profile

Number of cores (total)     =     384
Number of cores (fragment) =       1

THREADS (FRAGMENT)          =      48

Total time =          33120.9 seconds
```

「不老」-2021年6月時点

```
====================
   ## TIME PROFILE
====================

        Elapsed time: Monomer SCF      =         7114.0 seconds
        Elapsed time: Monomer MP3      =          343.1 seconds
        Elapsed time: Monomer (Total)  =         7532.4 seconds
        Elapsed time: Dimer ES         =          534.8 seconds
        Elapsed time: Dimer SCF        =          891.4 seconds
        Elapsed time: Dimer MP3        =         4265.7 seconds
        Elapsed time: Dimer (Total)    =        16306.3 seconds
        Elapsed time: FMO (Total)      =        23838.7 seconds

## Time profile

Number of cores (total)     =    3072
Number of cores (fragment) =       1

THREADS (FRAGMENT)          =      48

Total time =          24203.2 seconds
```

「富岳」-2021年9月時点

HPCI拠点でのライブラリ整備（2021年12月時点）

赤:A64FX, 緑:SX-Aurora TSUBASA, 紫:Xeon

■登録サイトと版

- 北大「Grand Chariot」： Ver. 1 Rev. 22 & Ver. 2 Rev. 4
- 東北大「AOBA-A」： Ver. 1 Rev. 22 (Vectorized version)
- JCAHPC「Oakforest-PACS」： Ver. 1 Rev. 22 (system decommissioned)
- 東大「Wisteria / Odyssey & Aquarius」： Ver. 1 Rev. 22 & Ver. 2 Rev. 4
- 東工大「TSUBAME3.0」： Ver. 1 Rev. 22 & Ver. 2 Rev. 4
- 海洋機構「Earth Simulator 4」： Ver. 1 Rev. 22 (Vectorized version)
- 分子研「RCCS」： Ver. 1 Rev. 22 & Ver. 2 Rev. 4
- 名大「不老 Type I」： Ver. 1 Rev. 22 & Ver. 2 Rev. 4
- 阪大「SQUID」： Ver. 1 Rev. 22 (Vectorized version)
- R-CCS「富岳」： Ver. 1 Rev. 22 & Ver. 2 Rev. 4
- 計算科学振興財団「FOCUSスパコン」： Ver. 1 Rev. 22 & Ver. 2 Rev. 4
- 九大「ITO Subsystem-A」： Ver. 1 Rev. 22 & Ver. 2 Rev. 4

- Ver. 1 Rev. 22はBioStation Viewerとの関係で当面は併存
- ベクトル化Ver. 2 Rev. 4（準備中）も順次追加登録の予定
- 2022年度中にはVer. 2 Rev. 4からRev. 8に更新の予定
- HPCI拠点のスパコン機種変更に応じた対応も適宜行う方針

Ver. 2 Rev. 8へ向けての改良

Fock行列の構築の改善

- 改良指針（井上G@富士通の助言）
  - **14個のif分岐（添字の同値性判断）が最適化を阻害、アクセスも不連続に**
- 手動での最適化と結果（ローカル版で評価中）
  - 基底関数添字の同値性を$(1/2)^n$ (n=1,2,3)で繰り込み
  - if制御は積分閾値の篩い落としのみ
  - 系にも拠るが最大で30%の加速
    （Fock処理の修正部分のみ）
- Fock行列の対角化（今後要対応）
  - LAPACKルーチンを利用
  - 正準直交化を導入
    （線形従属性問題も回避）

```
do p=ixi1,ixi2
  do q=ixj1,ixj2
    do r=ixk1,ixk2
      do s=ixl1,ixl2
        ix=ix+1
        val = sint(ix)
        if((abs(val) <= tv)) cycle
        fock(q,p)=fock(q,p)+dc(s,r)*val*2.d0！クーロン項
        fock(s,r)=fock(s,r)+dc(q,p)*val*2.d0
        fock(r,p)=fock(r,p)-dc(s,q)*val*0.5d0！交換項
        fock(s,p)=fock(s,p)-dc(r,q)*val*0.5d0
        fock(r,q)=fock(r,q)-dc(s,p)*val*0.5d0
        fock(s,q)=fock(s,q)-dc(r,p)*val*0.5d0
      end do
    end do
  end do
end do
```

追加の改善

- 2電子積分ルーチンのSIMD化とループ分割（2021/12/9版）
  - SIMD化に加えてループ分割を実施
    （SSSS, PSSS, SPSS, SSPS, SSSP, PPSS, PSPS, PSSP,
    SPPS, SPSP, SSPP, DSSS, SDSS, SSDS, SSSD）
  - 積分タイプにも拠るが20%～50%の加速
  - ループ分割はレジスタスピルの低減に効果？
  - 「不老」でのピーク性能比はMP2で2.4～2.8% （MP4だと9.5～10%）

- モノマーSCCの外挿法の変更（2022/7/13版）
  - アンダーソン外挿をFock行列ベースから密度行列ベースへ （オプション）
  - 系と基底関数にも拠るが反復回数を1,2割削減 （むしろ増えることも）

- その他
  - IFIEでの「不要プリント」の抑制 （オプション）
  - モノマーSCCでのHF部分の簡易積分バッファリング （試行中）
  - 通信量削減のためにダイマーHFの収束判定をスキップ （テスト中）
  - 積分アルゴリズムの変更 （HRR方式、SX向けベクトル化版の試行）

改良ssppルーチン#1

```
subroutine sub_sspp(zetam,pm,dkabm,etam,qm,dkcdm, &
     ma,mb,mc,md,ngij,ngkl,a,b,c,d,sint,tv)
  !
  !        Nov.06,'02
  !                T.NAKANO & Y. ABE
  !
  use constant
  use auxiliary_integral_table
  use integral_parameter
  implicit none
  real(8),intent(in)::zetam(*),pm(3,*),dkabm(*), &
       etam(*),qm(3,*),dkcdm(*)
  integer,intent(in)::ma,mb,mc,md,ngij,ngkl
  real(8),intent(in)::a(3),b(3),c(3),d(3),tv
  real(8),intent(out)::sint(*)
  !-----------------------------------------------
  integer npq,nrs,i,k,m,ix
  real(8) :: pq(3),ze,rz,re,rho,tt

  integer ts
  real(8) delta,t_inv,zssss
  real(8)::f0,f1,f2
  real(8) ssss(0:2),ssps(1:3,0:1),sspp(1:3,1:3,0:0)

  real(8) :: xqc(ngij*ngkl,3),xqd(ngij*ngkl,3),xwq(ngij*ngkl,3)
  real(8) :: xtt(ngij*ngkl),xre(ngij*ngkl),xeta2(ngij*ngkl),xa0(ngij*ngkl)
  integer :: npqrs
```

改良ssppルーチン#2

```
ix = 0
 !ocl eval
 !ocl fp_relaxed
 !ocl fp_contract
 !ocl noswp
 !ocl eval_concurrent
 !ocl SIMD
 do npq=1,ngij
   if (abs(dkabm(npq)) <= tv) cycle
   do nrs=1,ngkl
     if (abs(dkabm(npq)*dkcdm(nrs)) <= tv) cycle
     ix = ix + 1

     ze   = 1.0_8/(zetam(npq)+etam(nrs))
     rz   = etam(nrs)*ze
     re   = zetam(npq)*ze
     rho  = zetam(npq)*rz
     do i=1,3
       pq(i) = qm(i,nrs)-pm(i,npq)
       xqc(ix,i) = qm(i,nrs)-c(i)
       xqd(ix,i) = qm(i,nrs)-d(i)
       xwq(ix,i) = -re*pq(i)
     end do

     xtt(ix) = (pq(1)*pq(1)+pq(2)*pq(2)+pq(3)*pq(3))*rho
     xre(ix) = re
     xa0(ix)   = dkabm(npq)*dkcdm(nrs)*sqrt(ze)
     xeta2(ix) = 0.5_8/etam(nrs)

   enddo
 enddo
```

改良ssppルーチン#3

```
sint(1:9)=0.0_8
!ocl eval
!ocl fp_relaxed
!ocl fp_contract
!ocl noswp
!ocl eval_concurrent
!ocl SIMD
do npqrs = 1,ix
  tt = xtt(npqrs)
  if (tt <= 40.0_8) then ! Tf=2*m+36
    ts=0.5_8+tt*fmt_inv_step_size
    delta=ts*fmt_step_size-tt
    f0=((fmt_table(03,ts)*inv6*delta &
           +fmt_table(02,ts)*inv2)*delta &
           +fmt_table(01,ts))*delta &
           +fmt_table(00,ts)
    f1=((fmt_table(04,ts)*inv6*delta &
           +fmt_table(03,ts)*inv2)*delta &
           +fmt_table(02,ts))*delta &
           +fmt_table(01,ts)
    f2=((fmt_table(05,ts)*inv6*delta &
           +fmt_table(04,ts)*inv2)*delta &
           +fmt_table(03,ts))*delta &
           +fmt_table(02,ts)
  else
    t_inv=inv2/tt
    f0=sqrt(pi_over2*t_inv)
    f1=t_inv*f0
    f2=t_inv*3.0_8*f1
  end if
```

```
  ssss(0)=f0*xa0(npqrs)
  ssss(1)=f1*xa0(npqrs)
  ssss(2)=f2*xa0(npqrs)

  do m=0, 1
    ssps(1,m)=xqc(npqrs,1)*ssss(m)+xwq(npqrs,1)*ssss(m+1)
    ssps(2,m)=xqc(npqrs,2)*ssss(m)+xwq(npqrs,2)*ssss(m+1)
    ssps(3,m)=xqc(npqrs,3)*ssss(m)+xwq(npqrs,3)*ssss(m+1)
  end do

  do k=1, 3
    sspp(k,1,0)=xqd(npqrs,1)*ssps(k,0)+xwq(npqrs,1)*ssps(k,1)
    sspp(k,2,0)=xqd(npqrs,2)*ssps(k,0)+xwq(npqrs,2)*ssps(k,1)
    sspp(k,3,0)=xqd(npqrs,3)*ssps(k,0)+xwq(npqrs,3)*ssps(k,1)
  end do

  zssss=xeta2(npqrs)*(ssss(0)-xre(npqrs)*ssss(1))

  sspp(1,1,0)=sspp(1,1,0)+zssss
  sspp(2,2,0)=sspp(2,2,0)+zssss
  sspp(3,3,0)=sspp(3,3,0)+zssss

  sint(1) = sint(1)+sspp(1,1,0)
  sint(2) = sint(2)+sspp(1,2,0)
  sint(3) = sint(3)+sspp(1,3,0)
  sint(4) = sint(4)+sspp(2,1,0)
  sint(5) = sint(5)+sspp(2,2,0)
  sint(6) = sint(6)+sspp(2,3,0)
  sint(7) = sint(7)+sspp(3,1,0)
  sint(8) = sint(8)+sspp(3,2,0)
  sint(9) = sint(9)+sspp(3,3,0)

enddo

end subroutine sub_sspp
```

積分ルーチンのループ分割例

Sub_ssssの例

```
do npq=1,ngij
  if (abs(dkabm(npq)) > tv) then
   do nrs=1,ngkl
    if (abs(dkabm(npq)*dkcdm(nrs)) > tv) then
     ze  = 1.0_8/(zetam(npq)+etam(nrs))
     a0  = dkabm(npq)*dkcdm(nrs)*sqrt(ze)
     rz  = etam(nrs)*ze
     rho = zetam(npq)*rz
     tt  = ((qm(1,nrs)-pm(1,npq))*(qm(1,nrs)-pm(1,npq)) &
         +(qm(2,nrs)-pm(2,npq))*(qm(2,nrs)-pm(2,npq)) &
         +(qm(3,nrs)-pm(3,npq))*(qm(3,nrs)-pm(3,npq)))*rho
     if (tt <= 36.0_8) then ! Tf = 2*m+36 (for the case of m=0)
      ts    = 0.5_8+tt*fmt_inv_step_size
      delta = ts*fmt_step_size-tt
      ssss(0) = (((fmt_table(3,ts)*inv6*delta &
          +fmt_table(2,ts)*inv2)*delta &
          +fmt_table(1,ts))*delta &
          +fmt_table(0,ts))*a0
     else
      ssss(0) = sqrt(pi_over4/tt)*a0
     end if
     sint(1) = sint(1)+ssss(0)
    end if
   end do
  end if
end do
```

```
do npq=1,ngij
  if (abs(dkabm(npq)) <= tv) cycle
  do nrs=1,ngkl
   if (abs(dkabm(npq)*dkcdm(nrs)) <= tv) cycle
   ix = ix + 1
   ze  = 1.0_8/(zetam(npq)+etam(nrs))
   xa0(ix)  = dkabm(npq)*dkcdm(nrs)*sqrt(ze)
   rz  = etam(nrs)*ze
   rho = zetam(npq)*rz
   xtt(ix)  = ((qm(1,nrs)-pm(1,npq))*(qm(1,nrs)-pm(1,npq)) &
       +(qm(2,nrs)-pm(2,npq))*(qm(2,nrs)-pm(2,npq)) &
       +(qm(3,nrs)-pm(3,npq))*(qm(3,nrs)-pm(3,npq)))*rho
  enddo
enddo
sint(1)     = 0.0_8
do npqrs=1,ix
  tt = xtt(npqrs)
  if (tt <= 36.0_8) then ! Tf = 2*m+36 (for the case of m=0)
   ts    = 0.5_8+tt*fmt_inv_step_size
   delta = ts*fmt_step_size-tt
   ssss(0) = (((fmt_table(3,ts)*inv6*delta &
       +fmt_table(2,ts)*inv2)*delta &
       +fmt_table(1,ts))*delta &
       +fmt_table(0,ts))*xa0(npqrs)
  else
   ssss(0) = sqrt(pi_over4/tt)*xa0(npqrs)
  end if
  sint(1) = sint(1)+ssss(0)
enddo
```

- 満田氏はSIMD化、ループ分割の試行、さらに詳細な性能評価を実施
- 積分ルーチンの今後のチューニングに関しての有効指針を提示

「A64FXを用いたフラグメント分子軌道計算プログラムの性能評価」満田ら; 情処学会2022春 - 学生奨励賞受賞 発表

すべてのタイミングは2022年7月後半～8月前半に測定（MP2は全段でDGEMMを指定）.

「不老」でのジョブ時間の比較#1

Ala9Gly - FMO-MP2 (MP2 - all DGEMM) / total job time / 10 fragments

12 threads - 8 process @ 2 nodes

	Ver. / Rev.	Date	Sec.	Min.	Acc.
6-31G*	V1 R22	2020/6/3	134.4	2.24	1.00
	V2 R4	2021/9/16	116.6	1.94	1.15
	V2 R4改	2021/12/9	96.4	1.61	1.39
	V2 R4改	2022/7/13	91.9	1.53	1.46
	V2 R4改	2022/8/9	76.6	1.28	1.75
cc-pVDZ	V1 R22	2020/6/3	303.6	5.06	1.00
	V2 R4	2021/9/16	240.4	4.01	1.26
	V2 R4改	2021/12/9	187.8	3.13	1.62
	V2 R4改	2022/7/13	189.5	3.16	1.60
	V2 R4改	2022/8/9	159.2	2.65	1.91

Chignolin - FMO-MP2 (MP2 - all DGEMM) / total job time / 10 fragments

12 threads - 8 process @ 2 nodes

	Ver. / Rev.	Date	Sec.	Min.	Acc.
6-31G*	V1 R22	2020/6/3	719.6	11.99	1.00
	V2 R4	2021/9/16	647.3	10.79	1.11
	V2 R4改	2021/12/9	547.8	9.13	1.31
	V2 R4改	2022/7/13	500.9	8.35	1.44
cc-pVDZ	V1 R22	2020/6/3	1738.8	28.98	1.00
	V2 R4	2021/9/16	1491.5	24.86	1.17
	V2 R4改	2021/12/9	1230.1	20.50	1.41
	V2 R4改	2022/7/13	1226.9	20.45	1.42

- SIMD化とループ分割の併用は有効
- 基底関数の短縮の長いcc-pVDZの方が加速は顕著
- Ala_9GlyではHFでの積分バッファリングを試行（最下段）
- 同じ10残基（フラグメント）でも効果が異なる

「不老」でのジョブ時間の比較#2

Trp-Cage - FMO-MP2 (MP2 - all DGEMM) / total job time / 20 fragments

24 threads - 20 process @ 10 nodes

	Ver. / Rev.	Date	Sec.	Min.	Acc.
6-31G*	V1 R22	2020/6/3	469.6	7.83	1.00
	V2 R4	2021/9/16	413.9	6.90	1.13
	V2 R4改	2021/12/9	344.2	5.74	1.36
	V2 R4改	2022/7/13	294.2	4.90	1.60
cc-pVDZ	V1 R22	2020/6/3	1059.9	17.67	1.00
	V2 R4	2021/9/16	876.1	14.60	1.21
	V2 R4改	2021/12/9	706.7	11.78	1.50
	V2 R4改	2022/7/13	622.0	10.37	1.70

Crambin - FMO-MP2 (MP2 - all DGEMM) / total job time / 43 fragments

24 threads - 43 process @ 22 nodes

	Ver. / Rev.	Date	Sec.	Min.	Acc.
6-31G*	V1 R22	2020/6/3	572.6	9.54	1.00
	V2 R4	2021/9/16	509.4	8.49	1.12
	V2 R4改	2021/12/9	444.5	7.41	1.29
	V2 R4改	2022/7/13	457.7	7.63	1.25
cc-pVDZ	V1 R22	2020/6/3	1507.6	25.13	1.00
	V2 R4	2021/9/16	1232.6	20.54	1.22
	V2 R4改	2021/12/9	1005.2	16.75	1.50
	V2 R4改	2022/7/13	973.0	16.22	1.55

- モノマーSCCのアンダーソン外挿（密度）は効く場合も多々ある（「必ず」ではない）
- DNA-ウラニルの複合体の水和系では81回の反復が46回に削減されたことも

「不老」でのジョブ時間の比較#3

200～400残基のタンパク質が応用計算の主対象になっている

HIV-Protease - FMO-MP2 (MP2 - all DGEMM) / total job time / 203 fragments

24 threads - 204 process @ 102 nodes

	Ver. / Rev.	Date	Sec.	Min.	Acc.
6-31G*	V1 R22	2020/6/3	971.0	16.18	1.00
	V2 R4	2021/9/16	866.0	14.43	1.12
	V2 R4改	2021/12/9	777.8	12.96	1.25
	V2 R4改	2022/7/13	741.9	12.37	1.31
cc-pVDZ	V1 R22	2020/6/3	2101.4	35.02	1.00
	V2 R4	2021/9/16	1737.2	28.95	1.21
	V2 R4改	2021/12/9	1526.5	25.44	1.38
	V2 R4改	2022/7/13	1564.4	26.07	1.34

SARS-CoV-2 Mainprotease - FMO-MP2 (MP2 - all DGEMM) / total job time / 395 fragments

24 threads - 384 process @ 192 nodes

	Ver. / Rev.	Date	Sec.	Min.	Acc.
6-31G*	V1 R22	2020/6/3	996.8	16.61	1.00
	V2 R4	2021/9/16	874.2	14.57	1.14
	V2 R4改	2021/12/9	793.6	13.23	1.26
	V2 R4改	2022/7/13	741.6	12.36	1.34
cc-pVDZ	V1 R22	2020/6/3	2230.6	37.18	1.00
	V2 R4	2021/9/16	1882.0	31.37	1.19
	V2 R4改	2021/12/9	1633.3	27.22	1.37
	V2 R4改	2022/7/13	1475.6	24.59	1.51

- 全体としてcc-pVDZ基底の方が優位に加速が得られる場合が多い
- 「富岳」でも少しテストしているが、加速は（やや）大きめに出る

まとめ、今後の方向性

まとめ、今後の方向性

■高速化

・2電子積分のSIMD化とループ分割、Fock行列構築の改良、モノマーSCC加速
　⇒ MP2ジョブでは(従前比で)1.5～1.7倍 (改造部分のみは1.8倍～2.2倍)
　⇒ 22年度内に2倍を達成したい

■大規模化

・可視化用の「不要配列」の削除
　⇒ 1.1万フラグメントのMP3計算も可 (残基数2.2千の水和タンパク質)

■今後

・2電子積分のHRRアルゴリズムの検討
　⇒ 検討&テスト中 (演算数は低減、ループの最適化は今後)
・複数の積分計算ルーチンの使い分け
　⇒ VRRとHRRの混成使用 (系や基底に拠る自動選択の可能性)
　⇒ 最終目標の加速は(従前比で)3倍
・タンパク質の水和モデルでの水のクラスター化の前処理
　⇒ 実効フラグメント数を削減 (3.3千残基のSタンパク質の計算で試行)
・GPUでの実行もそろそろ要検討すべきか
　⇒ ポスト「富岳」を意識

Ref.; M. Head-Gordon et al., *J. Chem. Phys.* **89** (1988) 5777.

Head-GordonのHRR積分生成

TABLE I. Generation of $(dd|ss)$ integrals from $[ss|ss]$ integrals. The functional notation VRR( ) and HRR( ) indicates applying the VRR and the HRR with the required lower ERI classes listed as arguments. All contracted classes have $m = 0$.

Target class: $(dd|ss)$

Applying the horizontal recurrence relation:

$(dd|ss) = \mathrm{HRR}\{(fp|ss), (dp|ss)\}$
$(fp|ss) = \mathrm{HRR}\{(gs|ss), (fs|ss)\}$
$(dp|ss) = \mathrm{HRR}\{(fs|ss), (ds|ss)\}$

Classes to be formed by contraction of primitive integrals:

$(gs|ss), (fs|ss), (ds|ss)$

Applying the vertical recurrence relation:

$[gs|ss]^{(0)} = \mathrm{VRR}\{[fs|ss]^{(0,1)}, [ds|ss]^{(0,1)}\}$
$[fs|ss]^{(0,1)} = \mathrm{VRR}\{[ds|ss]^{(0,2)}, [ps|ss]^{(0,2)}\}$
$[ds|ss]^{(0,2)} = \mathrm{VRR}\{[ps|ss]^{(0,3)}, [ss|ss]^{(0,3)}\}$
$[ps|ss]^{(0,3)} = \mathrm{VRR}\{[ss|ss]^{(0,4)}\}$
$[ss|ss]^{(0,4)}$ (to be formed directly)

TABLE II. Generation of $(pp|ss)$ ERIs and first derivatives. ERIs with subscript α, β, and γ are formed with exponent-scaled contraction coefficients from centers 1, 2, and 3, respectively.

Target classes: $(pp|ss), (dp|ss)_\alpha, (sp|ss),$
$(pd|ss)_\beta, (ps|ss), (pp|ps)_\gamma$

Applying the horizontal recurrence relation:

$(pd|ss)_\beta = \mathrm{HRR}\{(dp|ss)_\beta, (pp|ss)_\beta\}$
$(dp|ss)_\alpha = \mathrm{HRR}\{(fs|ss)_\alpha, (ds|ss)_\alpha\}$
$(dp|ss)_\beta = \mathrm{HRR}\{(fs|ss)_\beta, (ds|ss)_\beta\}$
$(pp|ps)_\gamma = \mathrm{HRR}\{(ds|ps)_\gamma, (ps|ps)_\gamma\}$
$(pp|ss) = \mathrm{HRR}\{(ds|ss), (ps|ss)\}$
$(pp|ss)_\beta = \mathrm{HRR}\{(ds|ss)_\beta, (ps|ss)_\beta\}$
$(sp|ss) = \mathrm{HRR}\{(ps|ss), (ss|ss)\}$

Classes to be formed by contraction of primitive integrals:

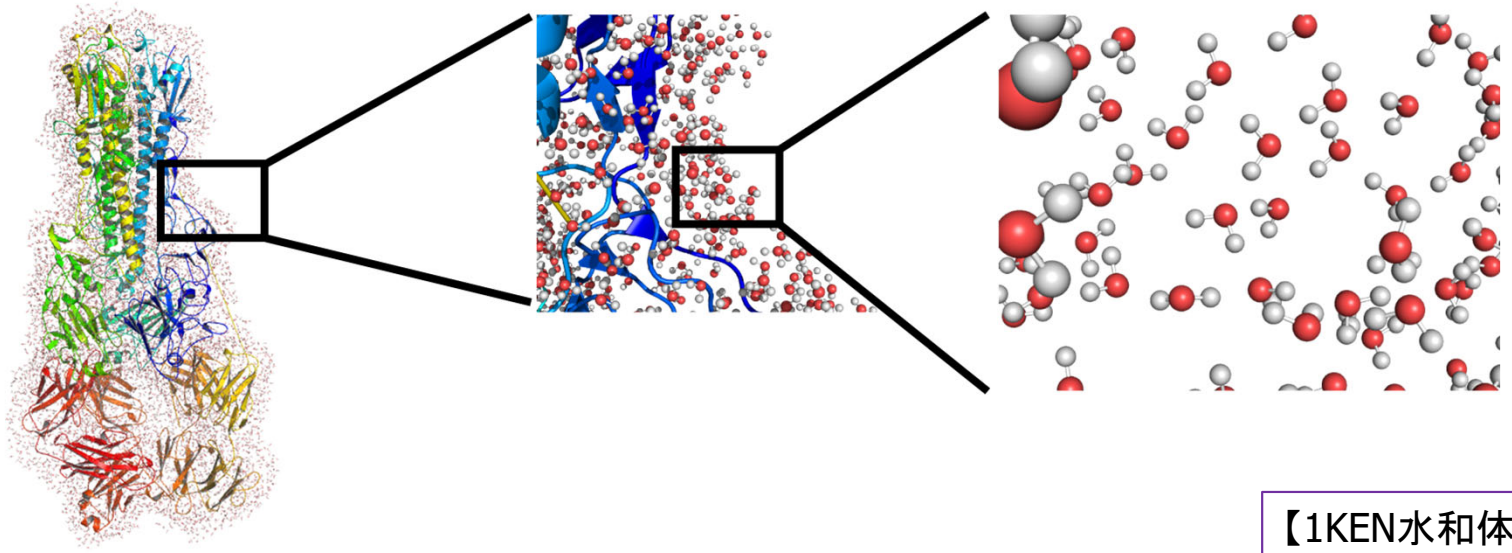
$(fs|ss)_\alpha, (fs|ss)_\beta, (ds|ps)_\gamma, (ds|ss), (ds|ss)_\alpha,$
$(ds|ss)_\beta, (ps|ps)_\gamma, (ps|ss), (ps|ss)_\beta, (ss|ss)$

Applying the vertical recurrence relation:

$[fs|ss]^{(0)} = \mathrm{VRR}\{[ds|ss]^{(0,1)}, [ps|ss]^{(0,1)}\}$
$[ds|ps]^{(0)} = \mathrm{VRR}\{[ds|ss]^{(0,1)}, [ps|ss]^{(1)}\}$
$[ds|ss]^{(0,1)} = \mathrm{VRR}\{[ps|ss]^{(0,2)}, [ss|ss]^{(0,2)}\}$
$[ps|ps]^{(0)} = \mathrm{VRR}\{[ps|ss]^{(0,1)}, [ss|ss]^{(1)}\}$
$[ps|ss]^{(0,2)} = \mathrm{VRR}\{[ss|ss]^{(0,3)}\}$
$[ss|ss]^{(0,3)}$ (to be formed directly)

- 水平漸化式(HRR)の方が演算量は少ない（展開項が短い場合は例外）
- 積分コードの自動生成プログラムはアリ
- チューニングの余地を含め検討中

タンパク質から遠方の水のクラスタリング



【1KEN水和体の例】

6142個の水分子を
2824個のクラスタに

1 water cluster :	900
2 waters cluster :	1110
3 waters cluster :	360
4 waters cluster :	328
5 waters cluster :	126

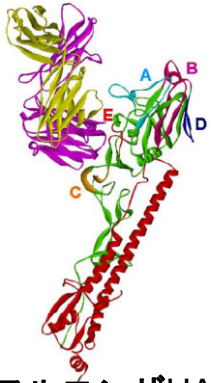
処理内容

1. 距離の近い水分子をグループ化する。(すでにグループ化されている水分子はグループに追加しない)
1. 2量体グループに近距離の2量体グループの水分子を追加する。
2. 3量体グループに単量体、もしくは2量体グループの水分子を追加する。
3. 2, 3の操作を複数回行い、ある程度4量体グループが作成できれば終了。

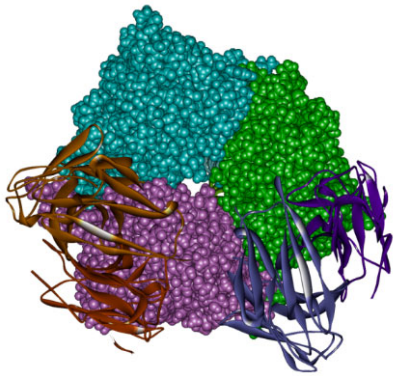
ABINIT-MPによるFMO計算のロードマップ

2004年 2008年 2010年 2013年 2015年 2020年

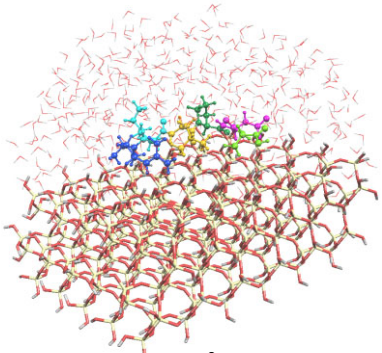
核内受容体(ER)
~300残基



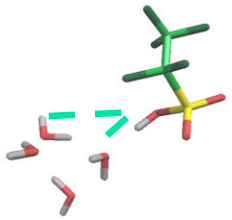
インフルエンザHA
抗原抗体系~1000残基



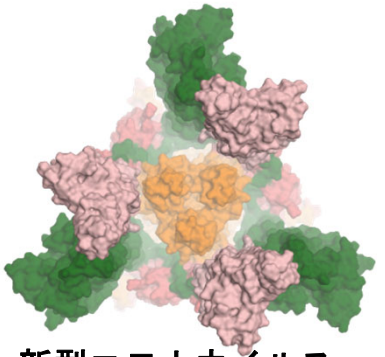
インフルエンザHA３量体
抗原抗体系~2400残基



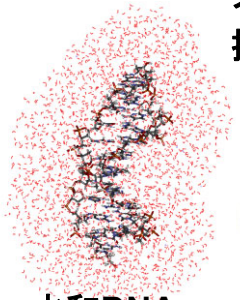
結晶ーペプチド複合系
~$(SiO_2)_{250}$-6残基-水和



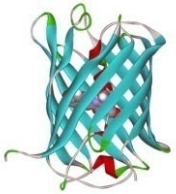
粗視化MD用
パラメータ
~数万サンプル



新型コロナウイルス
抗原抗体系~5300残基

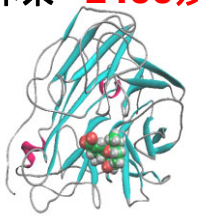


水和DNA
12塩基対＋2500wtr

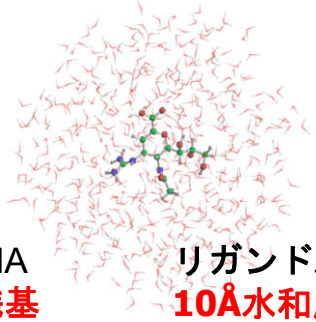


mFruits

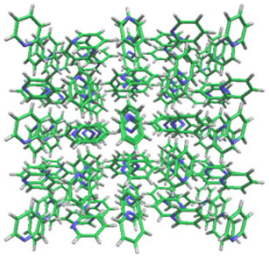
EGFR
チロシンキナーゼ



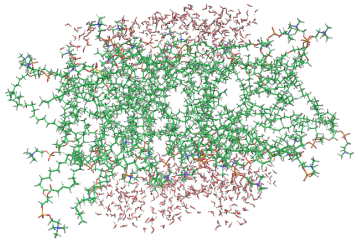
インフルエンザNA
タミフル~400残基



リガンド水和
10Å水和層



分子固体
~千個単位



粗視化→原子復元構造
~1万原子×サンプル数

計算	MP2	CIS/CIS(D)	MP3 CD	CCSD(T)	FMO4	Dimer-ES CMM		LRD	
構造	PDB一点計算/ モデル埋戻し	FMO-MD		MP2(p-opt)			FMO-DPD		MD生成 多構造
解析	IFIE	CAFI	FILM		BSSE	FMO4-IFIE		PIEDA	SVD	統計/ML
電荷、溶媒効果		ESP/RESP				NPA	SCIFIE	PB(SA)	大型液滴

事例）

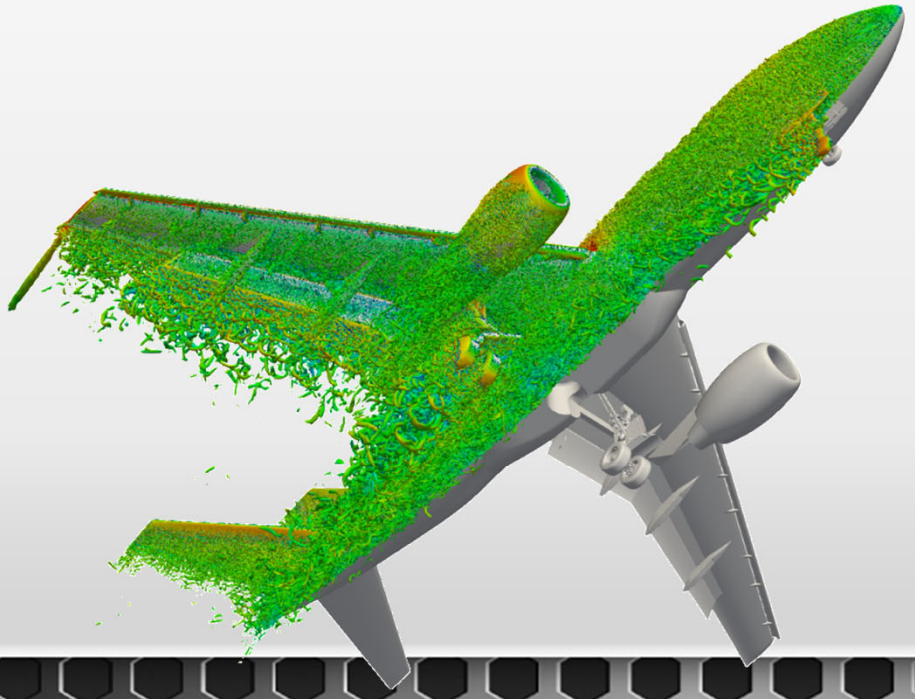
・ループ自動分割 機能

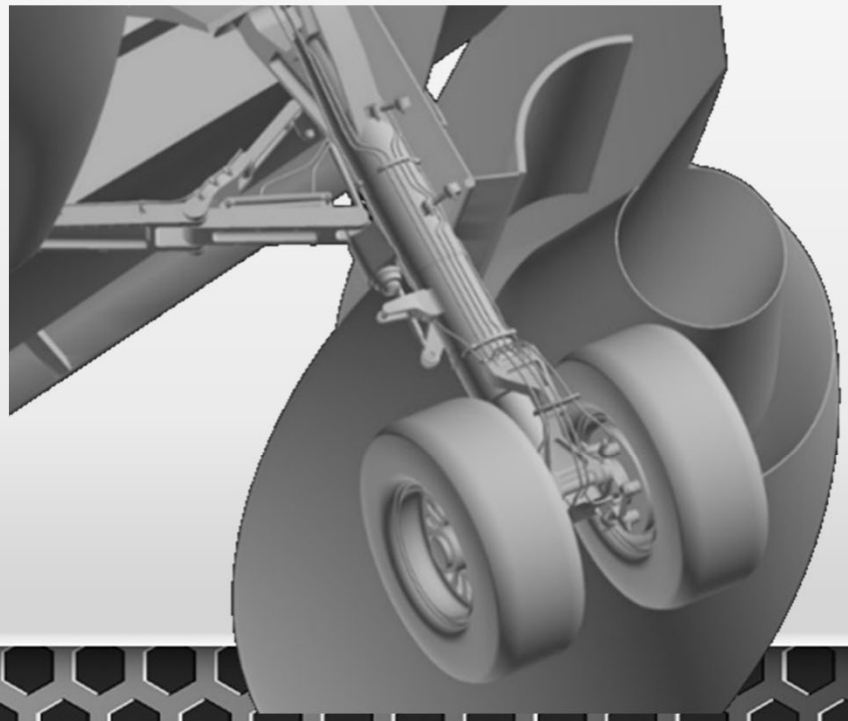
CFDプログラムのPRIMEHPC FX1000向け高速化チューニングについて(その2)

高木亮治
宇宙科学研究所
宇宙航空研究開発機構

流体解析(CFD)ソルバー

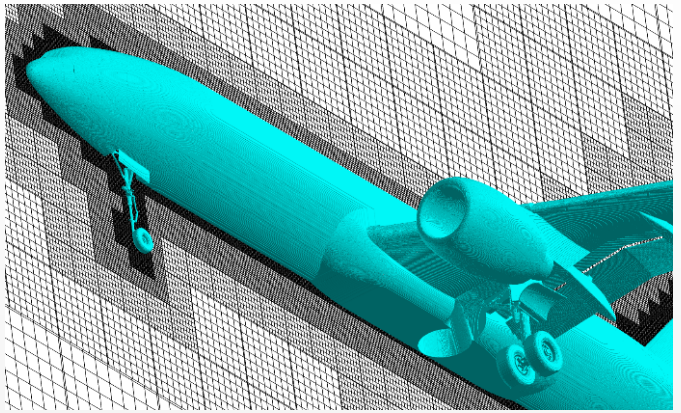
- 工学的CFD:設計・開発(・運用)で活用
  - 複雑な物理現象の高精度解析 ⇒ 第一原理的モデル
  - 複雑な形状への対応 ⇒ 複雑形状への計算格子自動作成
    - ✓従来手法:形状設計なのに形状変更に数ヶ月
    - ✓階層型等間隔直交構造格子法(BCM)＋埋め込み境界(IB)法
  - パラスタの必要性 ⇒ 高速計算、簡易モデル化

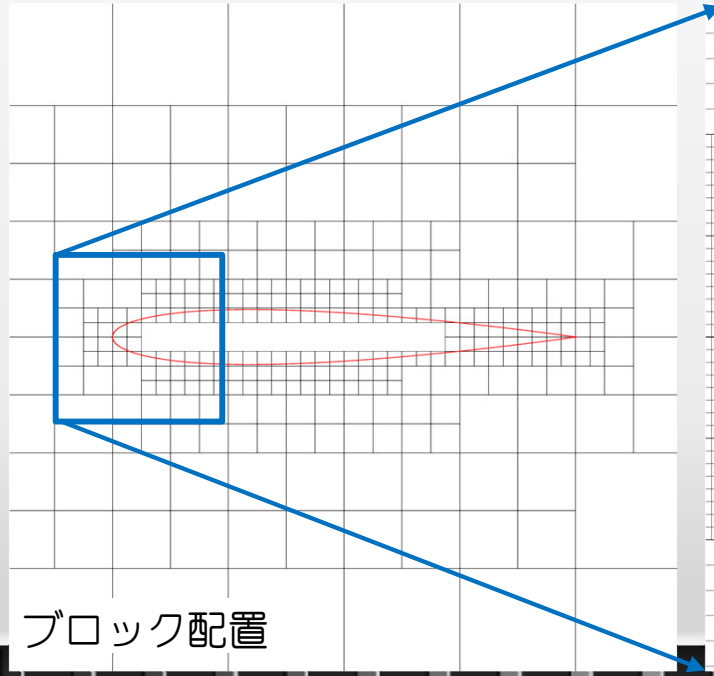




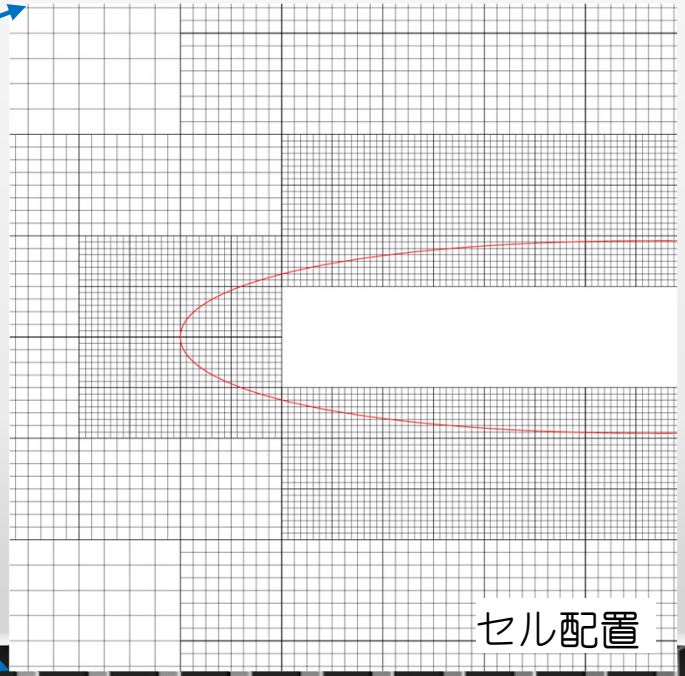
複雑形状への計算格子自動生成

- 階層型等間隔直交構造格子法(BCM)
  - 計算領域を八分木(3D)、四分木(2D)でブロック分割
  - 各ブロックは等間隔直交構造格子

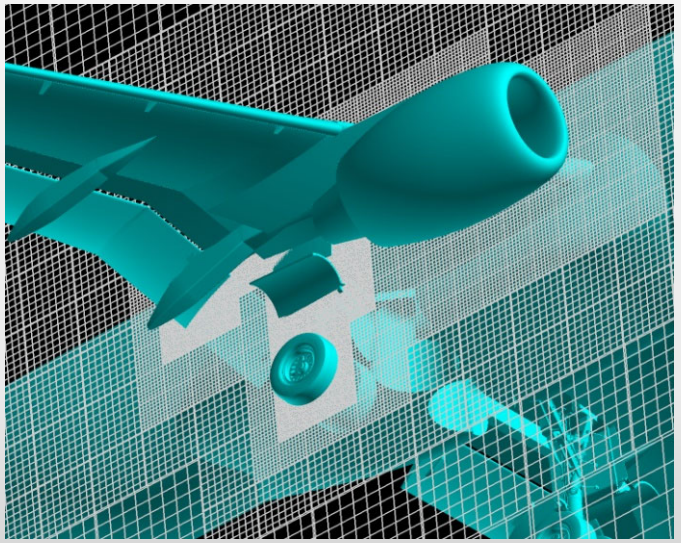




ブロック配置

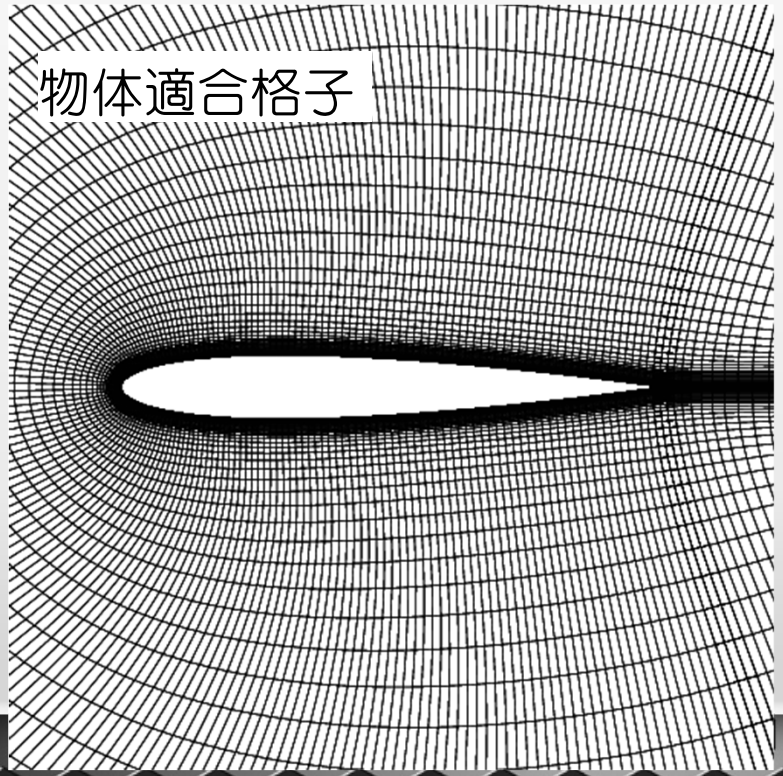


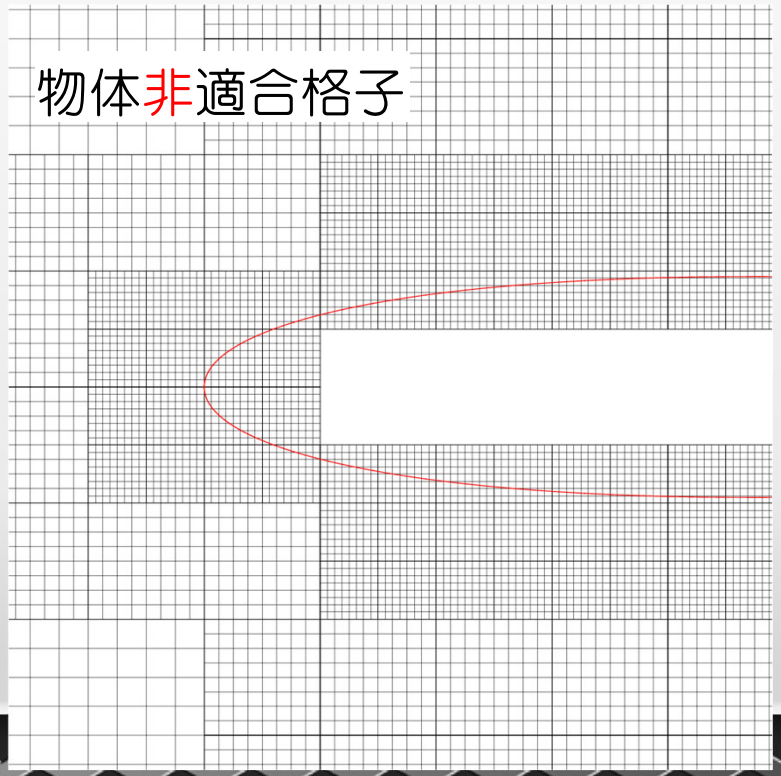
セル配置



埋め込み境界(IB)法

- 物体適合格子(従来手法)
  - 物体を計算格子で表現⇒支配方程式の座標変換で記述
- 物体非適合格子
  - 物体(主に壁)をアルゴリズムで表現



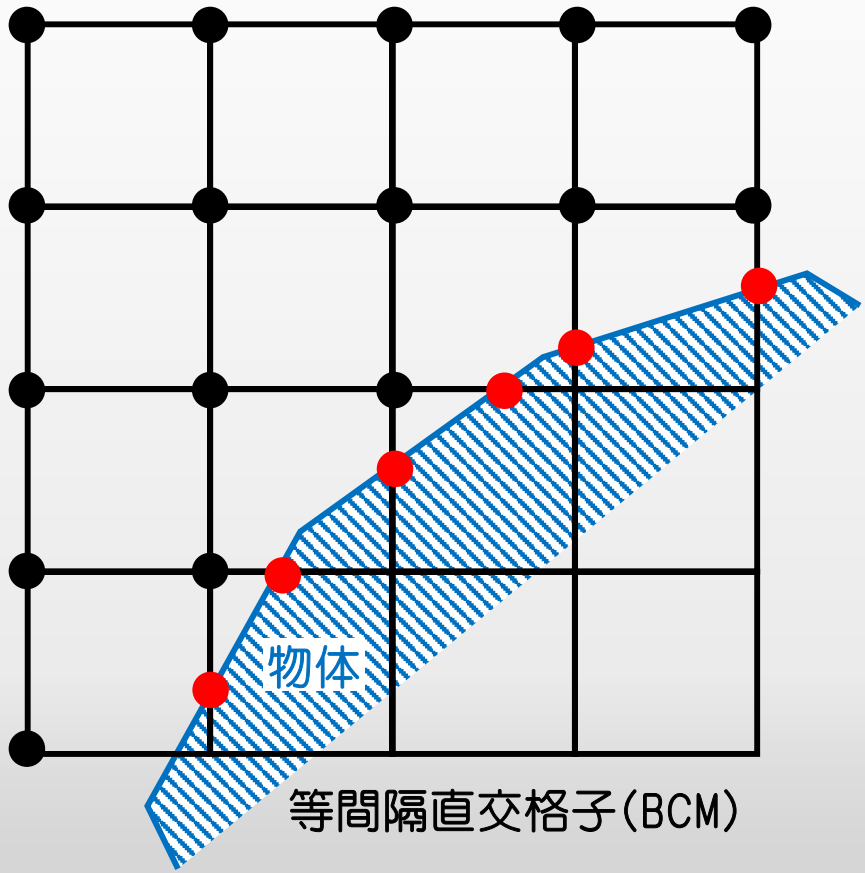


BCM+IBに特徴的な処理

- 流体の計算:右図の●
  - ステンシル計算(q(i, j, k, n) = F(q(i±1, …), q(i±2, …), …))
  - I, j, k～0(<100)、n=5
- 壁の処理(壁での流束計算):右図の●
  - if文の分岐処理

```
do k=istart, iend
do j=istart, iend
do i=istart, iend
  流体の計算
  if(壁がある) then
      壁の処理
  endif
enddo; enddo; enddo
```

通常の物体適合格子ソルバーにはない処理



If文の高速化手法

- if文を含むループ:
  - マスク処理: 対流項(solver_rhs_cflux)
    - ✓if文の真偽率で性能は変化しないが演算量が増大
  - if文の真と偽でループを分割: 粘性項(solver_rhs_vflux)
    - ✓計算領域全体:物体無しとして計算
    - ✓壁の処理を分離して別ループ(9ループ、リストアクセス)
    - ✓if文の真偽率で性能が変化
- 対流項:ステンシルが長く、場合分けが多い ⇒ マスク処理
- 粘性項:隣接ステンシルだけ ⇒ ループを分割
- if文を削除した多重ループ ⇒ 一般的な構造格子ソルバーと同じ

多重ループの高速化

- FX1000の課題
  - レジスタ不足(FX100:256→FX1000:32)
    - ✓ループボディが大きい
  - メモリアクセス性能を出し切るためには
    - ✓連続アクセスかつ長いループ長が必要

- 高速化: ループ1重化＋ブロッキング＋ループ分割:
  - ループ長の確保→ループ1重化
  - レジスタ不足→ループ分割(←ループ長の減少)
  - 分割ループ間でのメモリアクセスの削減→ブロッキング

多重ループの高速化

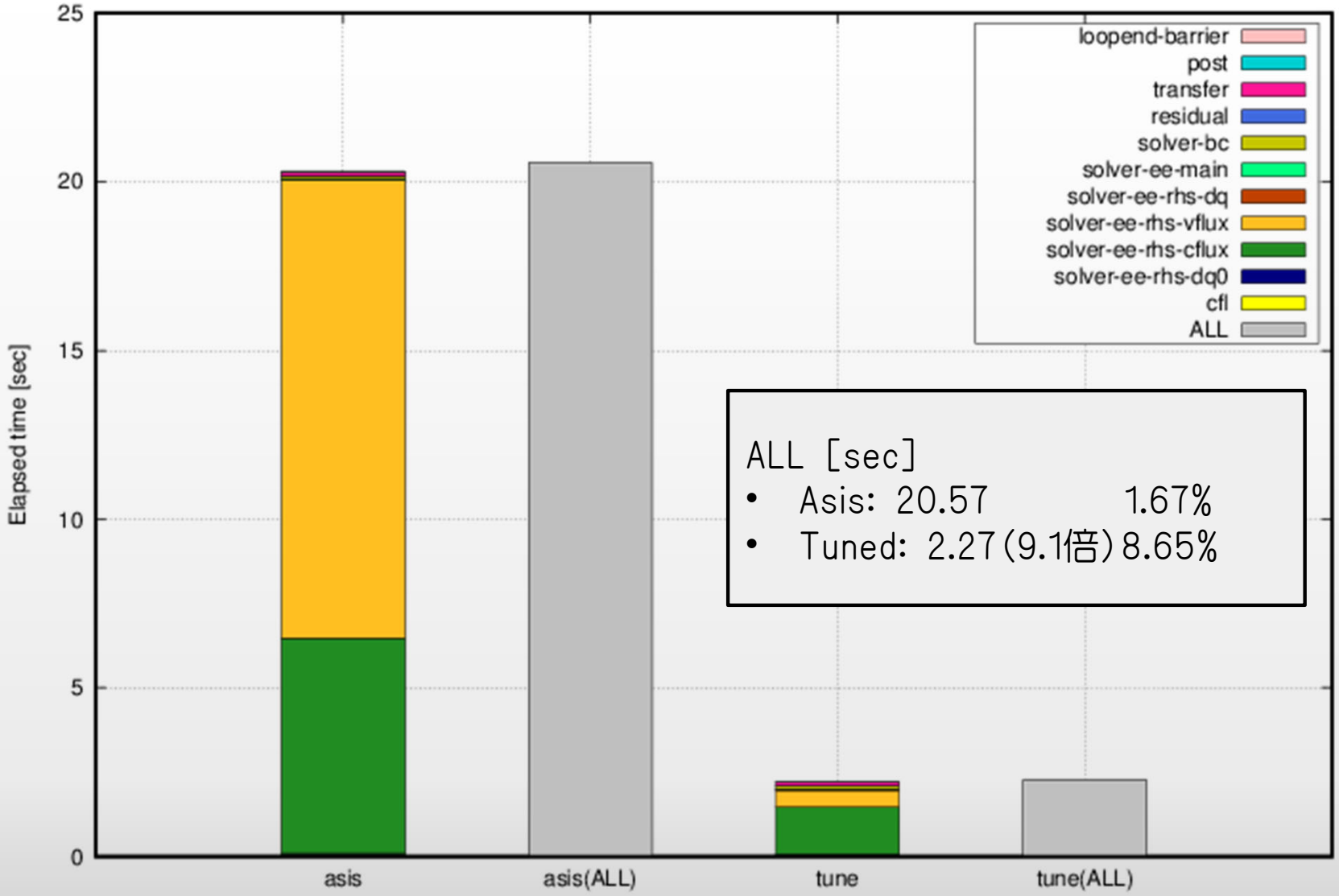
```
!$omp parallel do collapse(2)
do k=1,N
do  j=1,N
do  i=1,N
  a3d(i,j,k) = b3d(i,j,k)+S*c3d(i,j,k)
enddo; enddo; enddo
!$omp end parallel
```

```
real(8), dimension(:,:,:) :: a3d,b3d,c3d
call kernel(a3d,b3d,c3d)

contains
 subroutine kernel(a,b,c)
  real(8), dimension(*) :: a,b,c
  !$omp parallel
  !$omp do
  do lb=lbsrt,lbend,lbsize
   !ocl loop_fission_target(LS)
   !ocl loop_fission_threshold(40)
   !ocl prefetch_sequential(soft)
   do l=lb,lb+lbsize-1
    a(l) = b(l) + S*c(l)
   enddo; enddo
  !$omp end do
  !$omp do
   !ocl loop_fission_target(LS)
   !ocl loop_fission_threshold(40)
   !ocl prefetch_sequential(soft)
   do l=lbsrt_rest,lbend_rest
    a(l) = b(l) + S*c(l)
   enddo
   !$omp end do
   !$omp end parallel
```

- 3重ループの1重化:サブルーチンで変換
  ← 連続アクセス&ループ長の確保
- ループ分割:!oclの利用(富士通コンパイラ)
  ← レジスタスピル対策
- ブロッキング:手書き
  ← メモリアクセスの削減(L2$の利用)

```
j=1;k=1
!$omp parallel do
do i=1,N*N*N
  a3d(i,j,k) = b3d(i,j,k)+S*c3d(i,j,k)
enddo
!$omp end parallel
```

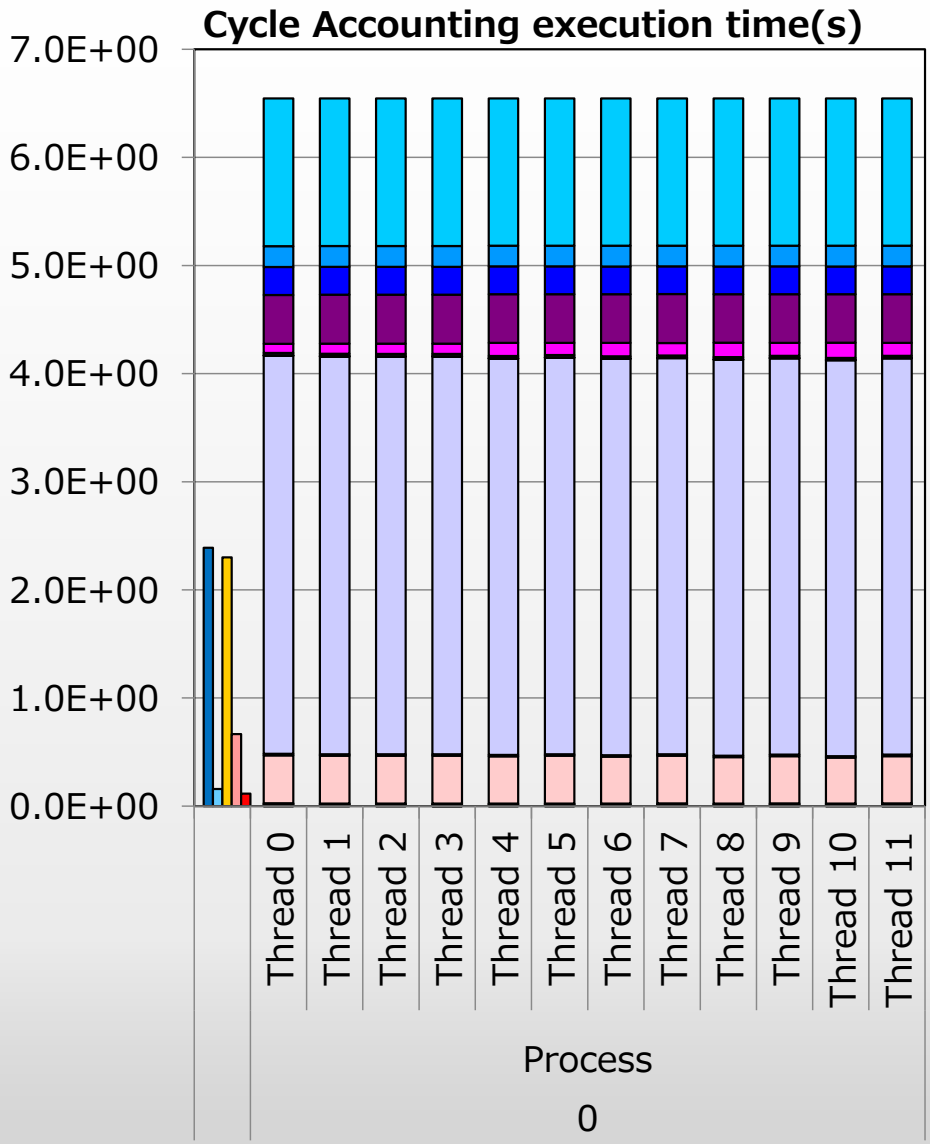
高速化結果

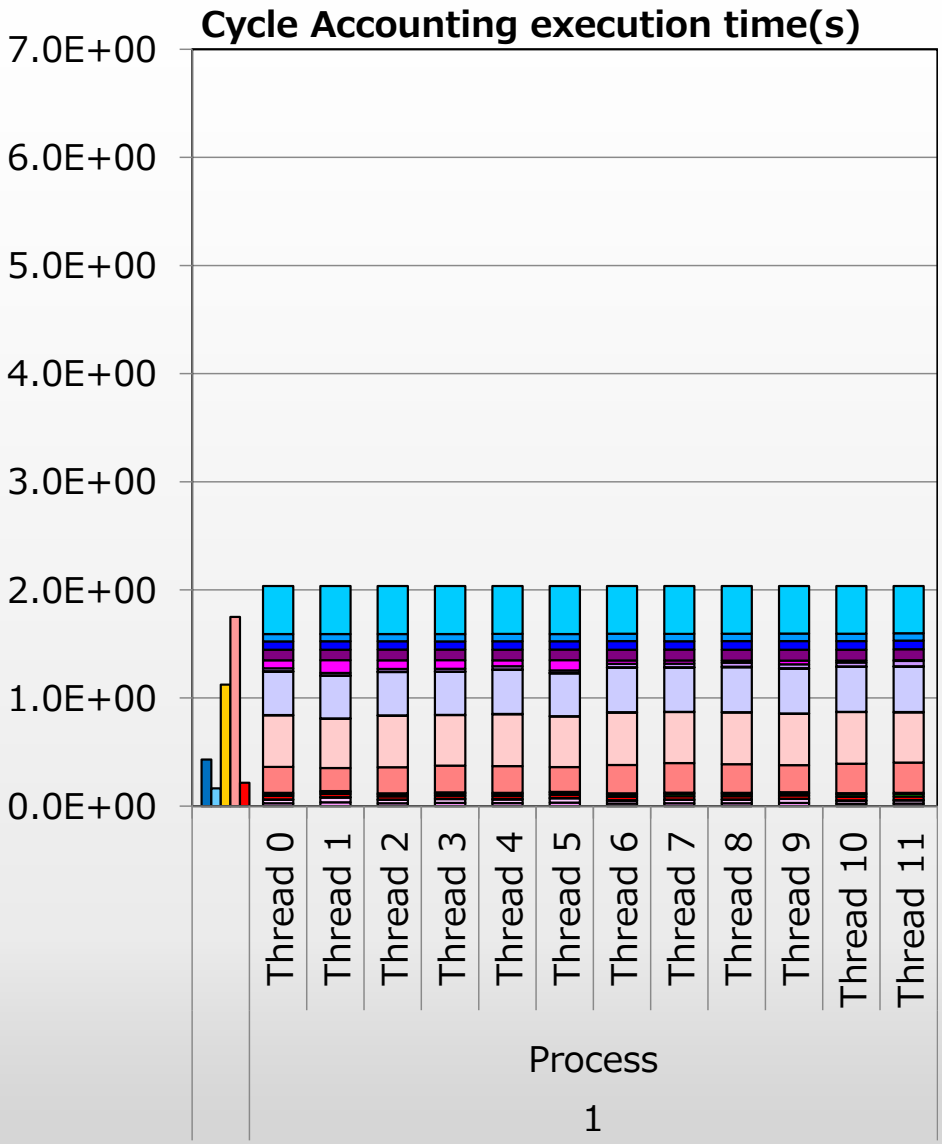


ALL [sec]
- Asis: 20.57 1.67%
- Tuned: 2.27 (9.1倍) 8.65%

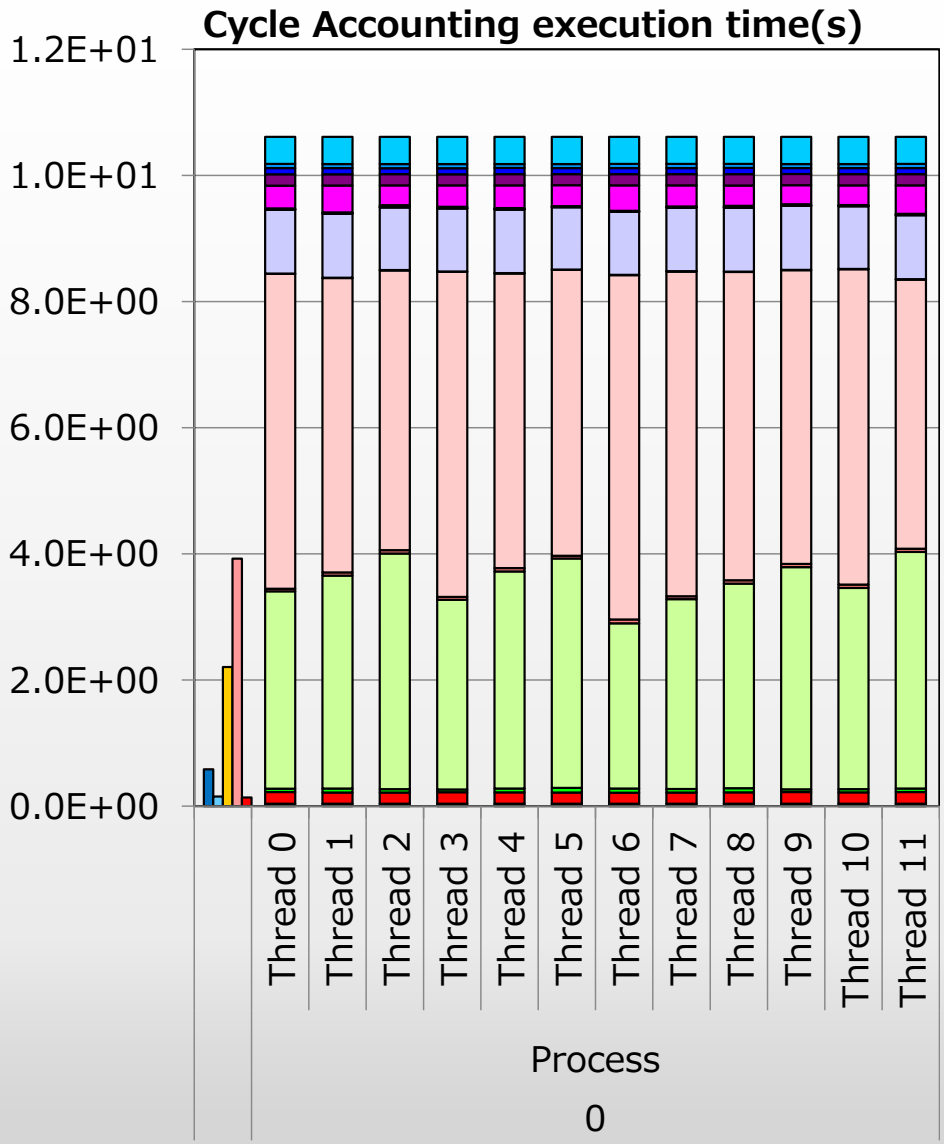
-Kfast,parallel,openmp,optmsg=2,ocl,noalias=s,preex,simd=2,autoobjstack,temparraystack

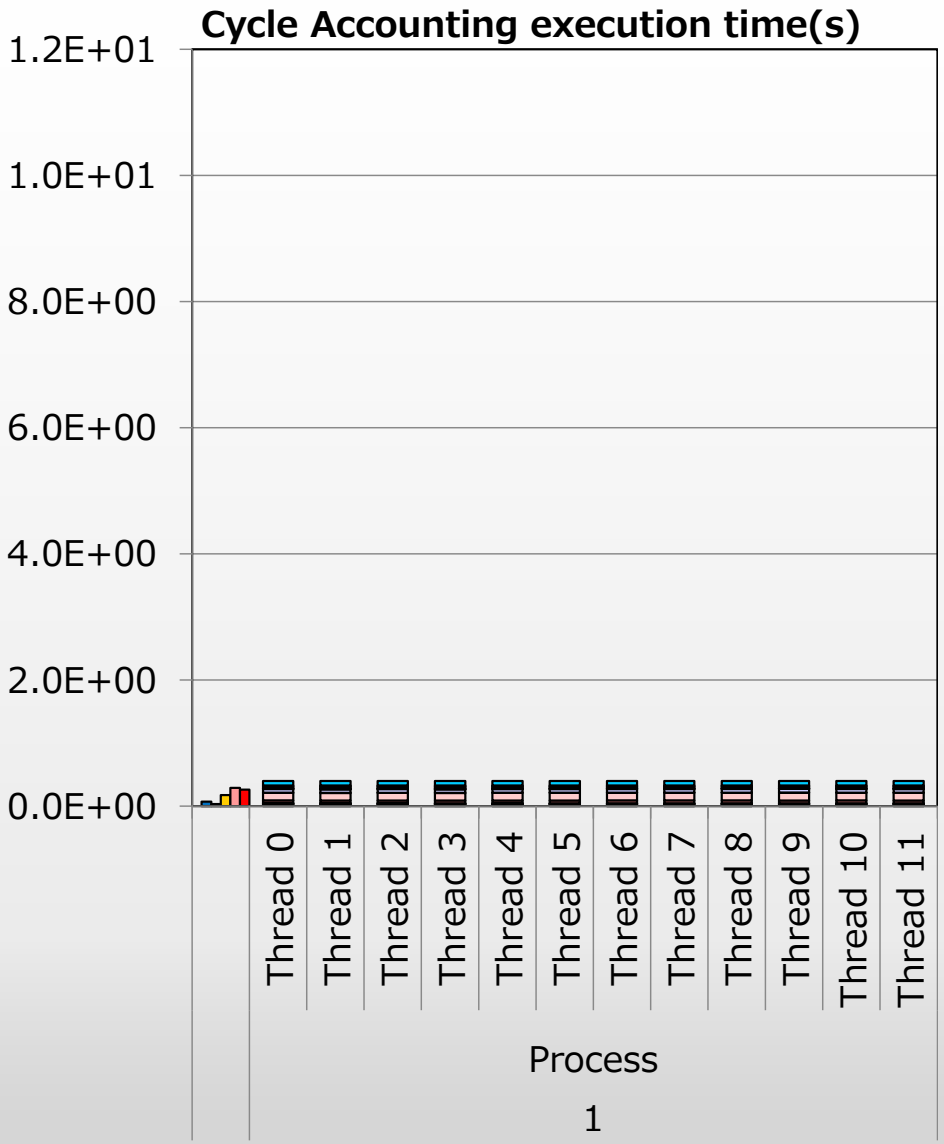
cflux(asis->tuned)



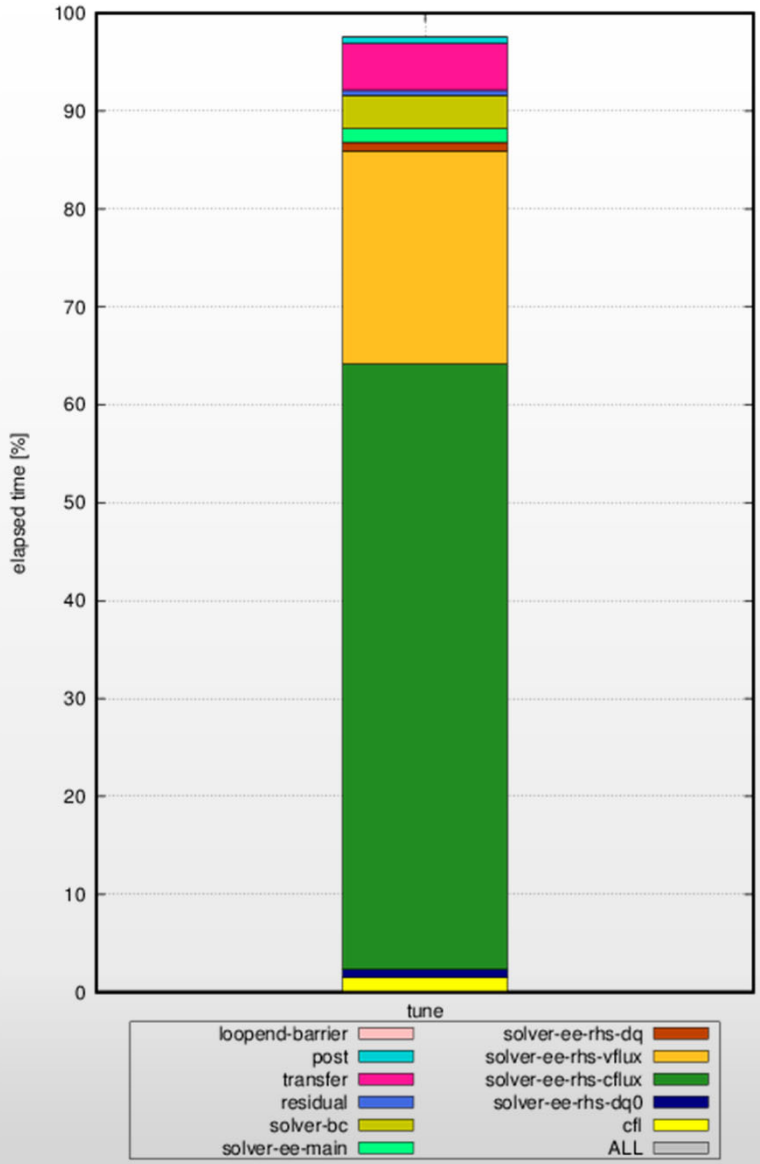


vflux(asis->tuned)





現状のホットカーネル

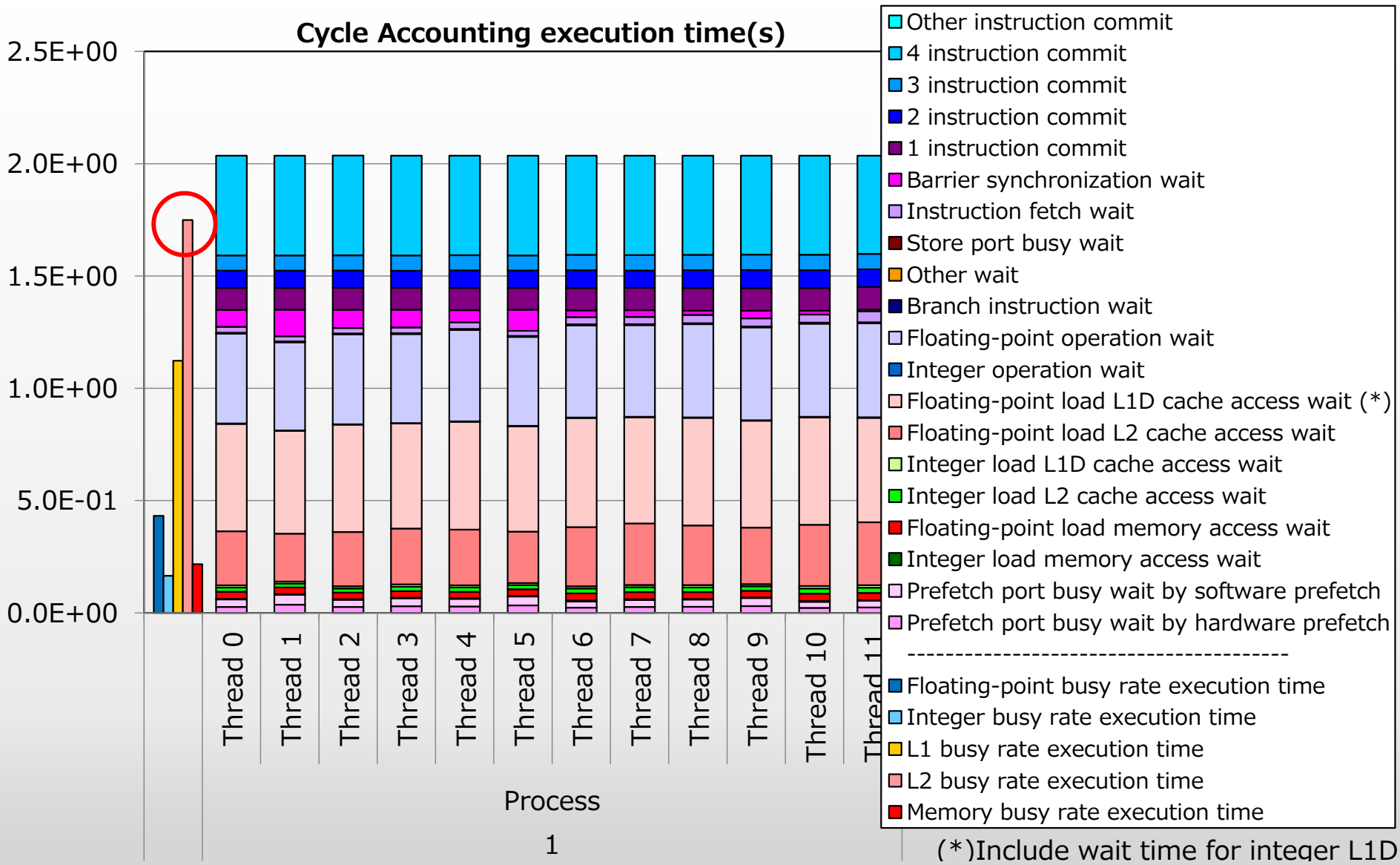


- 現状cfluxがホットカーネル
  - 2番目がvflux
- cfluxは15%程度は出て欲しい

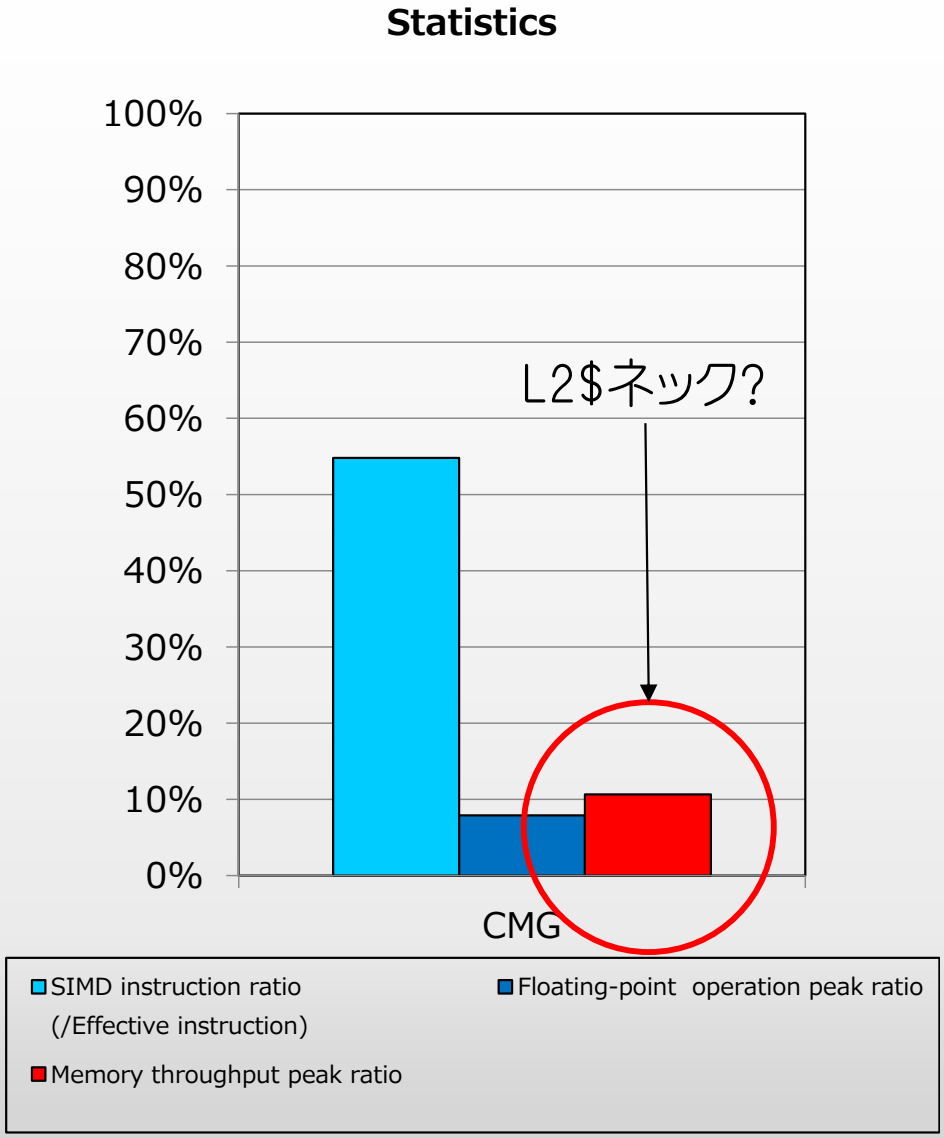
効率　[%]	cflux	vflux
演算	7.9	9.2
メモ	10.66	65.28
SIMD	54.81	50.89
ループ分割数	7+33	17

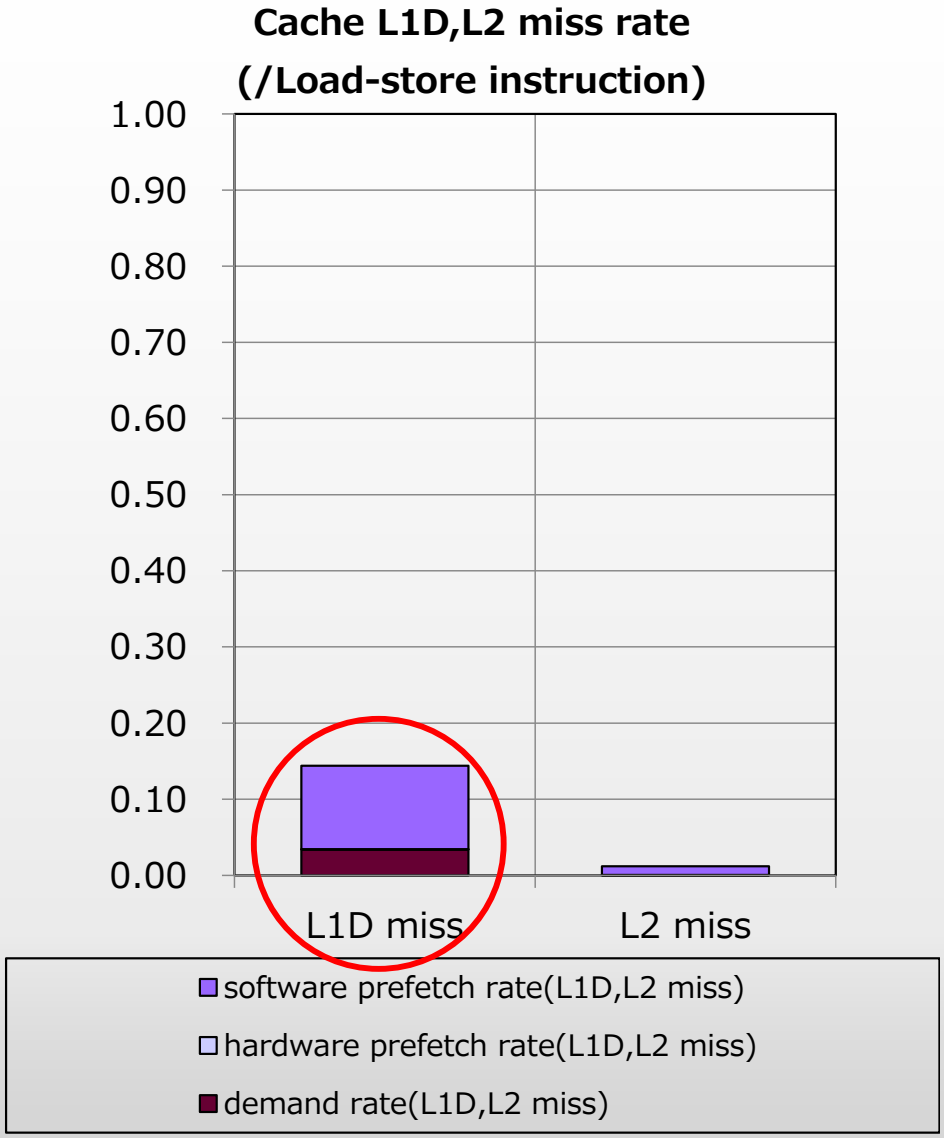
- 1ノード
- 4プロセス×12スレッド@CPU
- セル数:16.7M(20.1M)、64^3×64ブロック
- メモリ:10GB
- 壁率:0.18[%]

cflux(tuned)

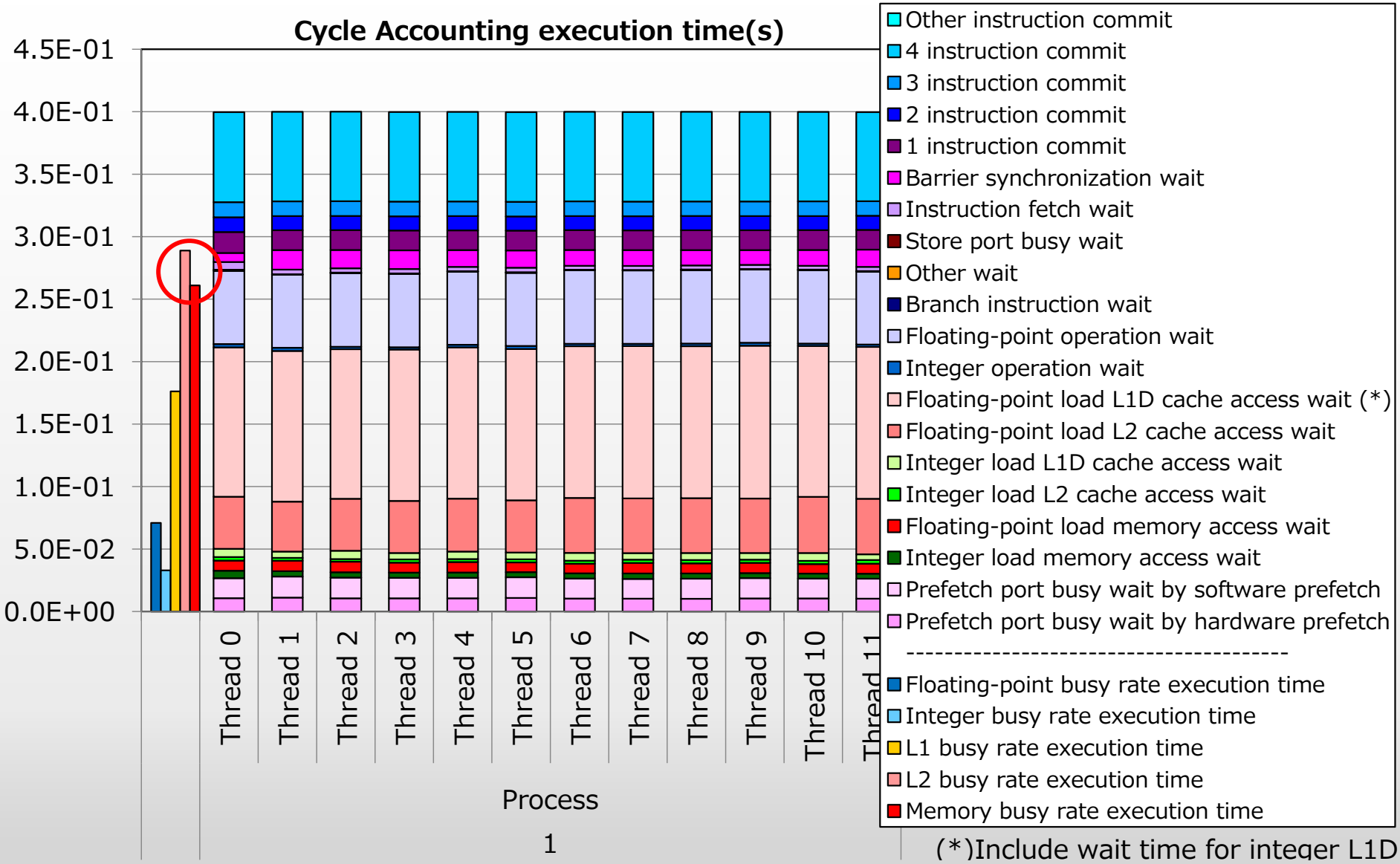

Cycle Accounting execution time(s)
2.5E+00
2.0E+00
1.5E+00
1.0E+00
5.0E-01
0.0E+00
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11
Process
1
Other instruction commit
4 instruction commit
3 instruction commit
2 instruction commit
1 instruction commit
Barrier synchronization wait
Instruction fetch wait
Store port busy wait
Other wait
Branch instruction wait
Floating-point operation wait
Integer operation wait
Floating-point load L1D cache access wait (*)
Floating-point load L2 cache access wait
Integer load L1D cache access wait
Integer load L2 cache access wait
Floating-point load memory access wait
Integer load memory access wait
Prefetch port busy wait by software prefetch
Prefetch port busy wait by hardware prefetch
Floating-point busy rate execution time
Integer busy rate execution time
L1 busy rate execution time
L2 busy rate execution time
Memory busy rate execution time
(*)Include wait time for integer L1D

cflux(tuned)

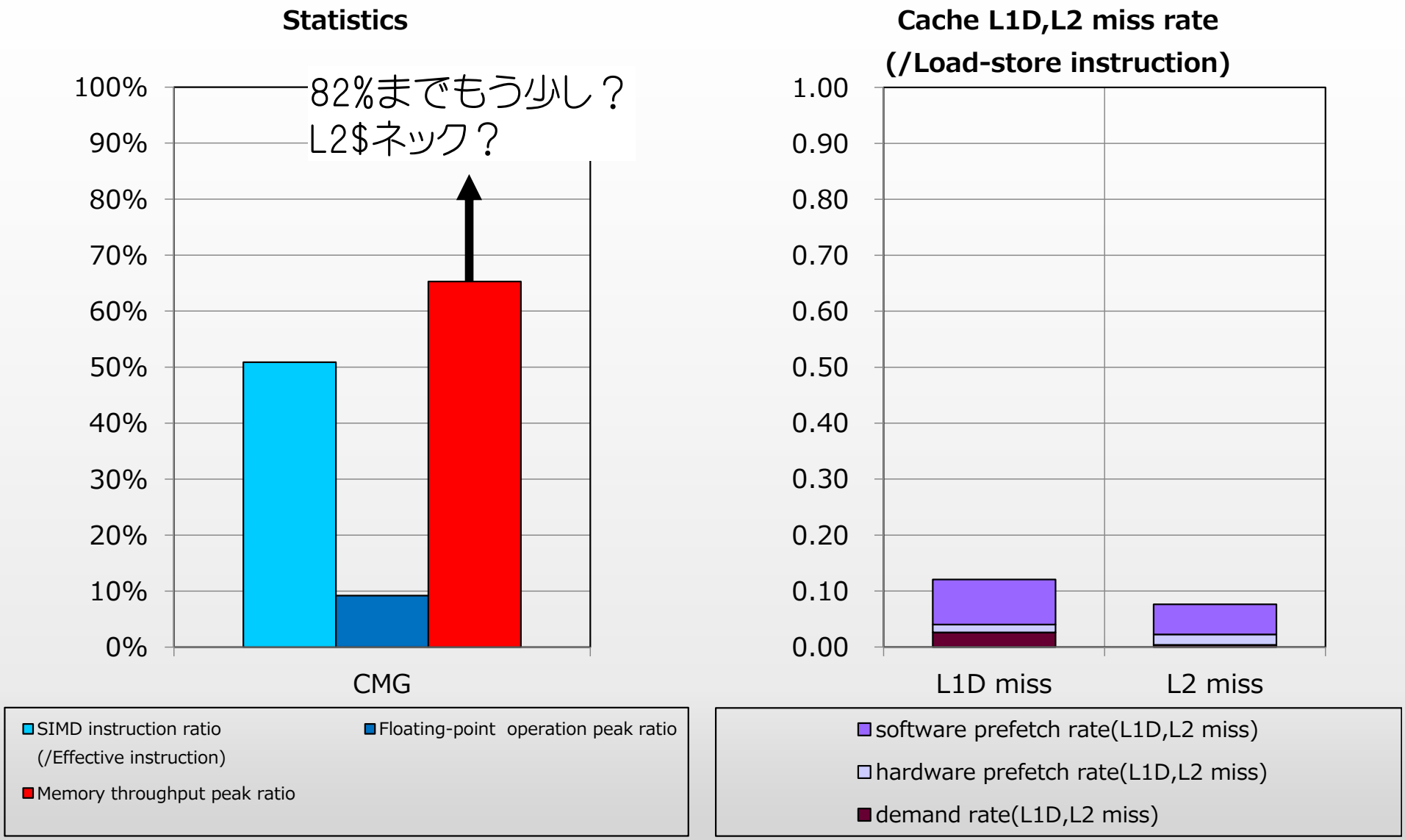

Statistics
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
L2$ネック?
CMG
SIMD instruction ratio (/Effective instruction)
Floating-point operation peak ratio
Memory throughput peak ratio


Cache L1D,L2 miss rate (/Load-store instruction)
1.00
0.90
0.80
0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.00
L1D miss
L2 miss
software prefetch rate(L1D,L2 miss)
hardware prefetch rate(L1D,L2 miss)
demand rate(L1D,L2 miss)

vflux(tuned)


Cycle Accounting execution time(s)
4.5E-01
4.0E-01
3.5E-01
3.0E-01
2.5E-01
2.0E-01
1.5E-01
1.0E-01
5.0E-02
0.0E+00
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11
Process
1
Other instruction commit
4 instruction commit
3 instruction commit
2 instruction commit
1 instruction commit
Barrier synchronization wait
Instruction fetch wait
Store port busy wait
Other wait
Branch instruction wait
Floating-point operation wait
Integer operation wait
Floating-point load L1D cache access wait (*)
Floating-point load L2 cache access wait
Integer load L1D cache access wait
Integer load L2 cache access wait
Floating-point load memory access wait
Integer load memory access wait
Prefetch port busy wait by software prefetch
Prefetch port busy wait by hardware prefetch
Floating-point busy rate execution time
Integer busy rate execution time
L1 busy rate execution time
L2 busy rate execution time
Memory busy rate execution time
(*)Include wait time for integer L1D

vflux(tuned)


Statistics
82%までもう少し？
L2$ネック？
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
CMG
SIMD instruction ratio (/Effective instruction)
Floating-point operation peak ratio
Memory throughput peak ratio
Cache L1D,L2 miss rate (/Load-store instruction)
1.00
0.90
0.80
0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.00
L1D miss
L2 miss
software prefetch rate(L1D,L2 miss)
hardware prefetch rate(L1D,L2 miss)
demand rate(L1D,L2 miss)

cflux

- アプリのB/F:
  - 算出根拠:
    - ✓ -gでの浮動小数点演算数:112.75［GFLOP］
    - ✓ プロファイラでのメモリアクセス量:852.52［GB］
    - ✓ L2$アクセス:852.52［GB］－ 89.6［GB］(L2$ミス)=762.92［GB］
      - ◘ L2$ミス:3.5×$10^8$×256［B］=89.6［GB］
  - メモリB/F:852.52/112.75 = 7.56
  - L2$ B/F:762.92/112.75 = 6.77

H/WのB/F:

- メモリ:1,024/3379.2 = 0.30
- L2$:3,600+/3379.2 = 1.06
- L1$:11,000+/3379.2 = 3.26

- 効率の予測:(実測は6%程度)
  - メモリ:0.30/7.56 = 3.97［%］(×0.8？)
  - L2$:1.06/6.77 = 15.7［%］(×？)

さらなる高速化は可能か？

- 現状はL2$ネック ⇒　L1$, L2$のチューニング？
- プリフェッチの工夫→自動ループ分割とバッティング？
  - ループの手動分割？
- ループ内での計算順序の入替（定義と参照を近づける）
- ループ1重化の課題:袖（現状2→4）による演算量増加
  - 実行効率的には良いかもしれないが経過時間的には×
- ループ長の最適値:32^3, 64^3, 96^3
- マスク処理を完全に分離（tuneC:壁なし3重ループ）だと11%程度:ループボディの演算量？

チューニング手法(現状)

- asis(if文は手動マスキング):

```
do k=1-ov,N+ov
do j=1-ov,N+ov
do i=1-ov,N+ov
  q3d(i,j,k,:) = qf3d(i,j,k,:)*flag + qs3d(i,j,k,:)*(1-flag)
  処理2
enddo; enddo; enddo
```

N = 64 (, 16,32)
ov = 2 (, 4)

- 現在(自動ループ分割(I)＋手動ブロッキング(I)):

```
do I=I_min,I_max
  q1d(I) = qf1d(I)*flag + qs1d(I)*(1-flag)
  処理2
enddo
```

I: 64^3 (, 16^3,32^3)

チューニング手法(他候補)

- tuneA(2重ループを1重化→自動ループ分割(ij)、結果的にブロッキング):

```
do k=1-ov,N+ov
  do ij=ij_min,ij_max
    q2d(ij,k,:) = qf2d(ij,k,:)*flag + qs2d(ij,k,:)*(1-flag)
    処理2
  enddo
enddo; enddo
```

ij=64^2 (, 16^2,32^2)

- tuneB(手動ループ分割、結果的にブロッキング、ループ長が課題？):

```
do k=1-ov,N+ov
do  j=1-ov,N+ov
  do  i=1-ov,N+ov
    q3d(i,j,k,:) = qf3d(i,j,k,:)*flag + qs3d(i,j,k,:)*(1-flag)
  enddo
  do  i=1-ov,N+ov
    処理2
  enddo
enddo; enddo
```

チューニング手法(他候補)

- tuneC:

```
do k=1-ov,N+ov
do j=1-ov,N+ov
do i=1-ov,N+ov
  q3d(i,j,k,:) = qf3d(i,j,k,:)…
  処理2
enddo; enddo; enddo
do l=l_min,l_max
  i=List(l,1); j=List(l,2); k=List(k,3)
  q3d(i,j,k,:) = qs3d(i,j,k,:)…
  処理2
enddo; enddo; enddo
```

流体の処理 (asisの演算量削減？)

壁の処理

まとめ

- BCM+IB法を用いたCFDプログラムにFX1000向け高速化を実施した。
  - IB法に固有のif文を削除し、多重ループの高速化を実施
  - ループ1重化＋ブロッキング＋ループ分割を適用し9倍程度の高速化を実現した。
- 更なる高速化(cflux)を実施したいがどうすれば良いか？

tuned(cflux & vflux)

cflux

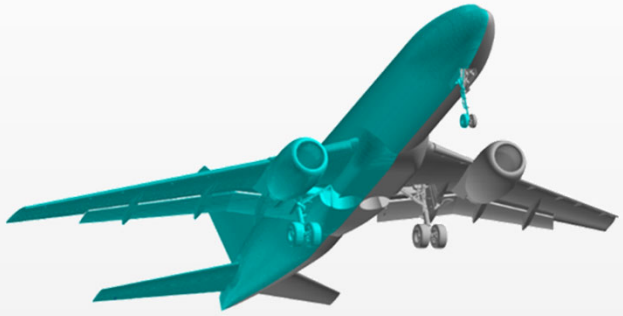
Statistics		Execution time (s)	GFLOPS	Floating-point operation peak ratio (%)	Memory throughput (GB/s)	Memory throughput peak ratio (%)	Effective instruction	Floating-point operation	SIMD instruction rate (%) (/Effective instruction)	SVE operation rate (%)	Floating-point pipeline Active element rate (%)	IPC	GIPS
Process	Thread												
1	0	2.04E+00	5.56	7.91%	2.30		4.89E+09	1.13E+10	54.79%	100.00%	78.44%	1.09	2.40
1	1	2.04E+00	5.56	7.91%	2.26		4.89E+09	1.13E+10	54.81%	100.00%	78.44%	1.09	2.40
1	2	2.04E+00	5.56	7.90%	2.27		4.89E+09	1.13E+10	54.81%	100.00%	78.43%	1.09	2.40
1	3	2.04E+00	5.56	7.91%	2.27		4.89E+09	1.13E+10	54.82%	100.00%	78.45%	1.09	2.40
1	4	2.04E+00	5.56	7.91%	2.26		4.89E+09	1.13E+10	54.82%	100.00%	78.43%	1.09	2.40
1	5	2.04E+00	5.56	7.91%	2.26		4.89E+09	1.13E+10	54.82%	100.00%	78.44%	1.09	2.40
1	6	2.04E+00	5.56	7.91%	2.27	10.66%	4.89E+09	1.13E+10	54.82%	100.00%	78.42%	1.09	2.40
1	7	2.04E+00	5.56	7.91%	2.27		4.89E+09	1.13E+10	54.82%	100.00%	78.44%	1.09	2.40
1	8	2.04E+00	5.56	7.91%	2.27		4.89E+09	1.13E+10	54.82%	100.00%	78.42%	1.09	2.40
1	9	2.04E+00	5.56	7.91%	2.28		4.89E+09	1.13E+10	54.82%	100.00%	78.43%	1.09	2.40
1	10	2.04E+00	5.56	7.91%	2.28		4.89E+09	1.13E+10	54.82%	100.00%	78.42%	1.09	2.40
1	11	2.04E+00	5.51	7.84%	2.27		4.85E+09	1.12E+10	54.82%	100.00%	78.43%	1.08	2.38
CMG 1 total		2.04E+00	66.64	7.90%	27.28	10.66%	5.86E+10	1.36E+11	54.81%	100.00%	78.43%	1.09	28.80

vflux

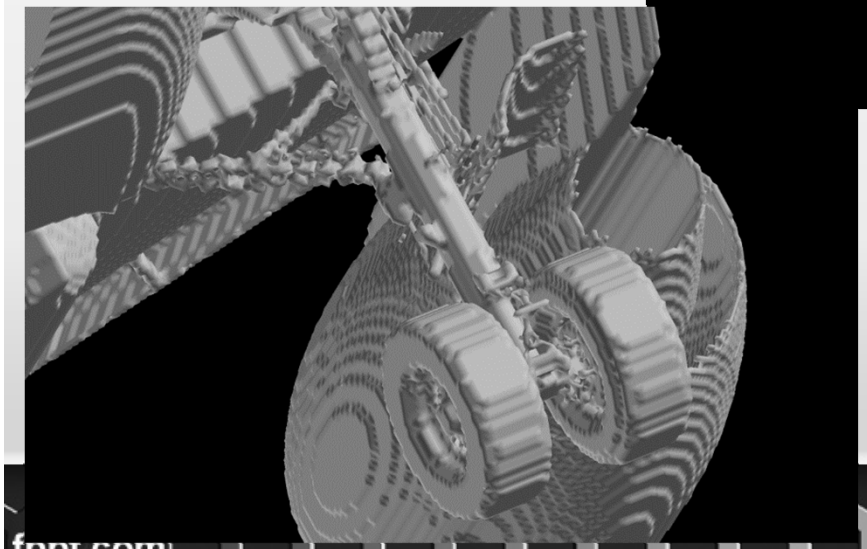
Statistics		Execution time (s)	GFLOPS	Floating-point operation peak ratio (%)	Memory throughput (GB/s)	Memory throughput peak ratio (%)	Effective instruction	Floating-point operation	SIMD instruction rate (%) (/Effective instruction)	SVE operation rate (%)	Floating-point pipeline Active element rate (%)	IPC	GIPS
Process	Thread												
1	0	4.00E-01	6.43	9.21%	14.09		7.94E+08	2.57E+09	50.36%	99.55%	79.78%	0.91	1.99
1	1	4.00E-01	6.43	9.20%	13.88		7.85E+08	2.57E+09	50.94%	99.55%	79.79%	0.90	1.96
1	2	4.00E-01	6.43	9.20%	13.92		7.85E+08	2.57E+09	50.94%	99.55%	79.79%	0.90	1.96
1	3	4.00E-01	6.43	9.21%	13.87		7.85E+08	2.57E+09	50.93%	99.55%	79.78%	0.90	1.96
1	4	4.00E-01	6.43	9.20%	13.91		7.85E+08	2.57E+09	50.94%	99.55%	79.78%	0.90	1.96
1	5	4.00E-01	6.43	9.21%	13.93		7.85E+08	2.57E+09	50.94%	99.55%	79.78%	0.90	1.97
1	6	4.00E-01	6.43	9.20%	13.91	65.28%	7.85E+08	2.57E+09	50.95%	99.55%	79.78%	0.90	1.96
1	7	4.00E-01	6.43	9.21%	13.97		7.85E+08	2.57E+09	50.95%	99.55%	79.78%	0.90	1.96
1	8	4.00E-01	6.43	9.20%	13.93		7.85E+08	2.57E+09	50.95%	99.55%	79.78%	0.90	1.96
1	9	4.00E-01	6.43	9.20%	13.90		7.85E+08	2.57E+09	50.95%	99.55%	79.78%	0.90	1.96
1	10	4.00E-01	6.43	9.20%	13.90		7.85E+08	2.57E+09	50.95%	99.55%	79.78%	0.90	1.96
1	11	4.00E-01	6.39	9.15%	13.92		7.81E+08	2.56E+09	50.94%	99.55%	79.76%	0.89	1.95
CMG 1 total		4.00E-01	77.13	9.20%	167.11	65.28%	9.43E+09	3.08E+10	50.89%	99.55%	79.78%	0.90	23.58

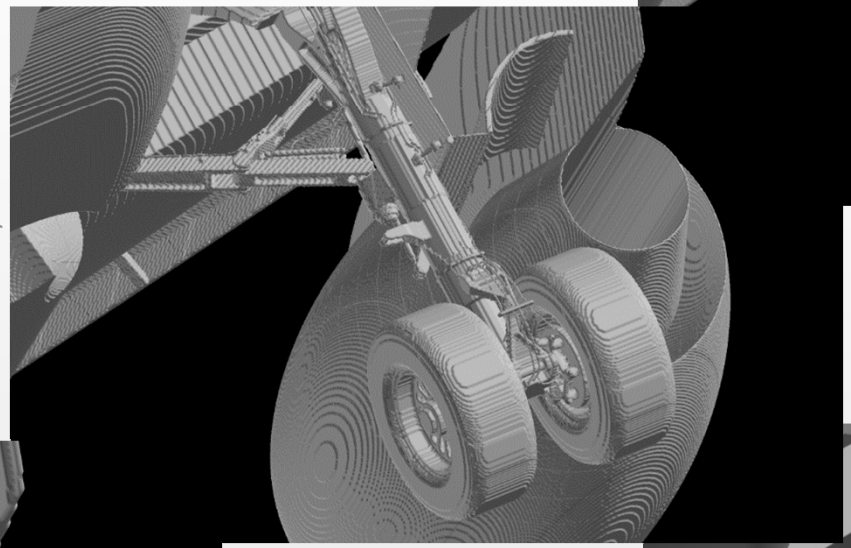
形状再現性

- 格子増加による物体形状再現性の向上
  - セルサイズ未満の構造は無視

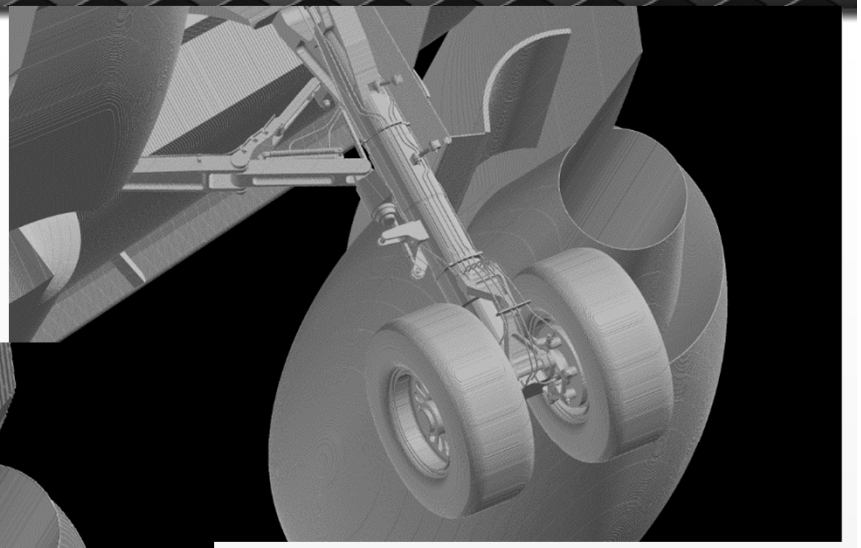


6,400万点



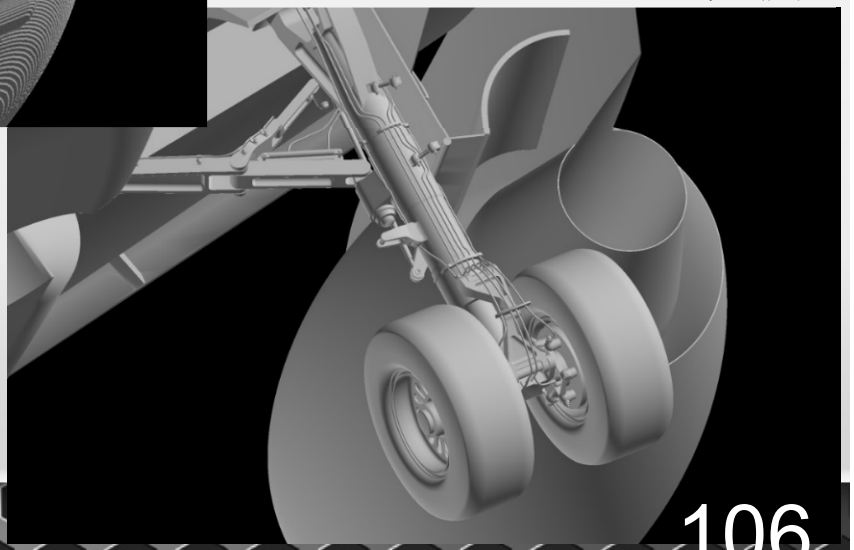


6.7億点

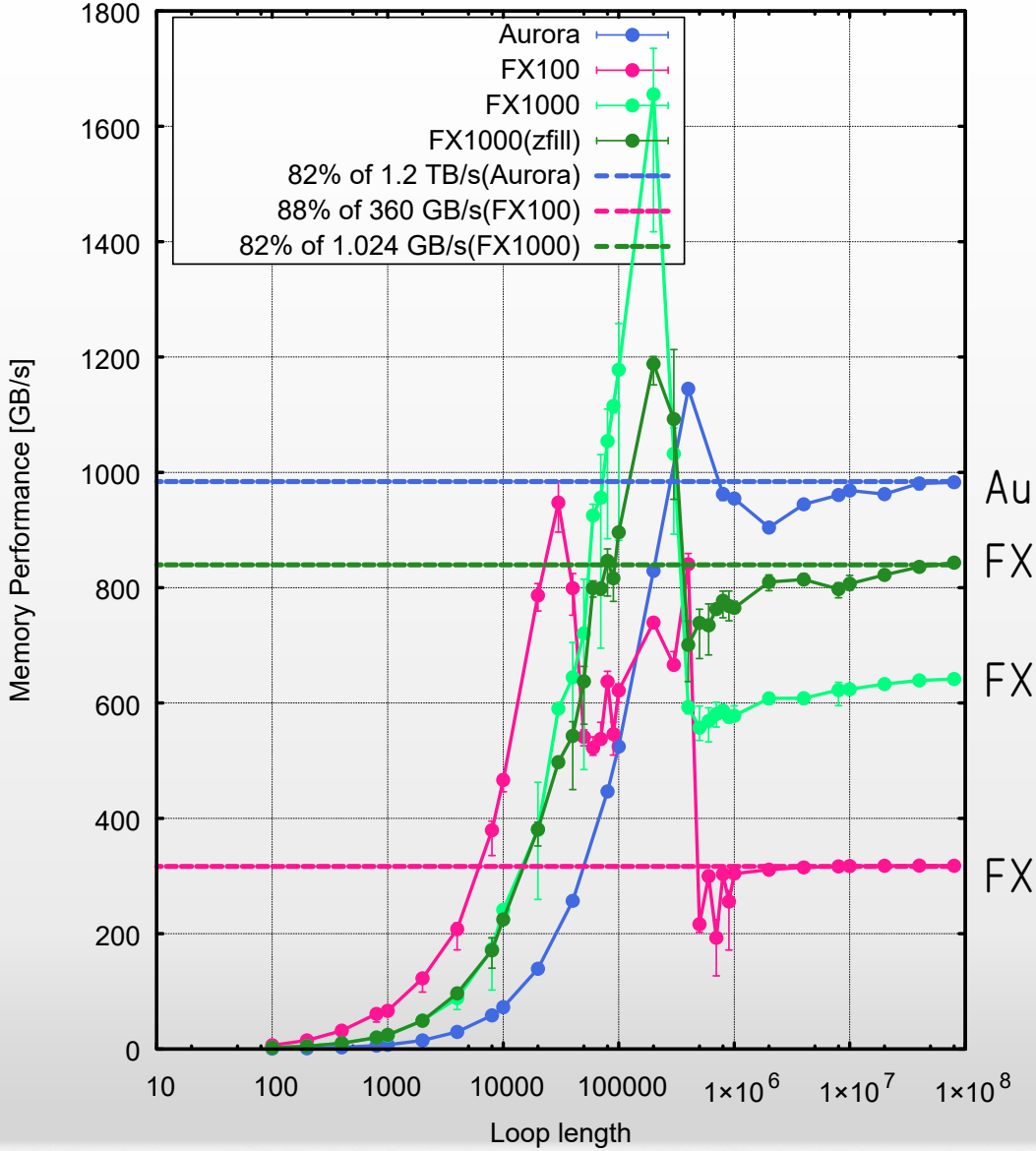


150億点

CAD形状



FX1000の実メモリ性能



-Kfast, zfill, parallel, openmp, ocl, prefetch_sequential=soft, prefetch_stride,optmsg=2, noalias=s, mfunc=2, preex, autoobjstack, temparraystack

Aurora

FX1000(zfill)

FX1000

FX100

STREAM TRIAD: a(n)=b(n)+S×c(n)

マルチブロック、多次元配列、袖

- 多次元配列
- ovlp(袖):a(1-ovlp:ndim+ovlp,1-ovlp:ndim+ovlp,1-ovlp:ndim+ovlp), …
- MB:キャッシュの影響は排除し、メモリ性能を測定
- 総格子点数:2,700万点(10x10x10 x 27,000ブロック～300x300x300 x 1ブロック)

実アプリの実装モデル

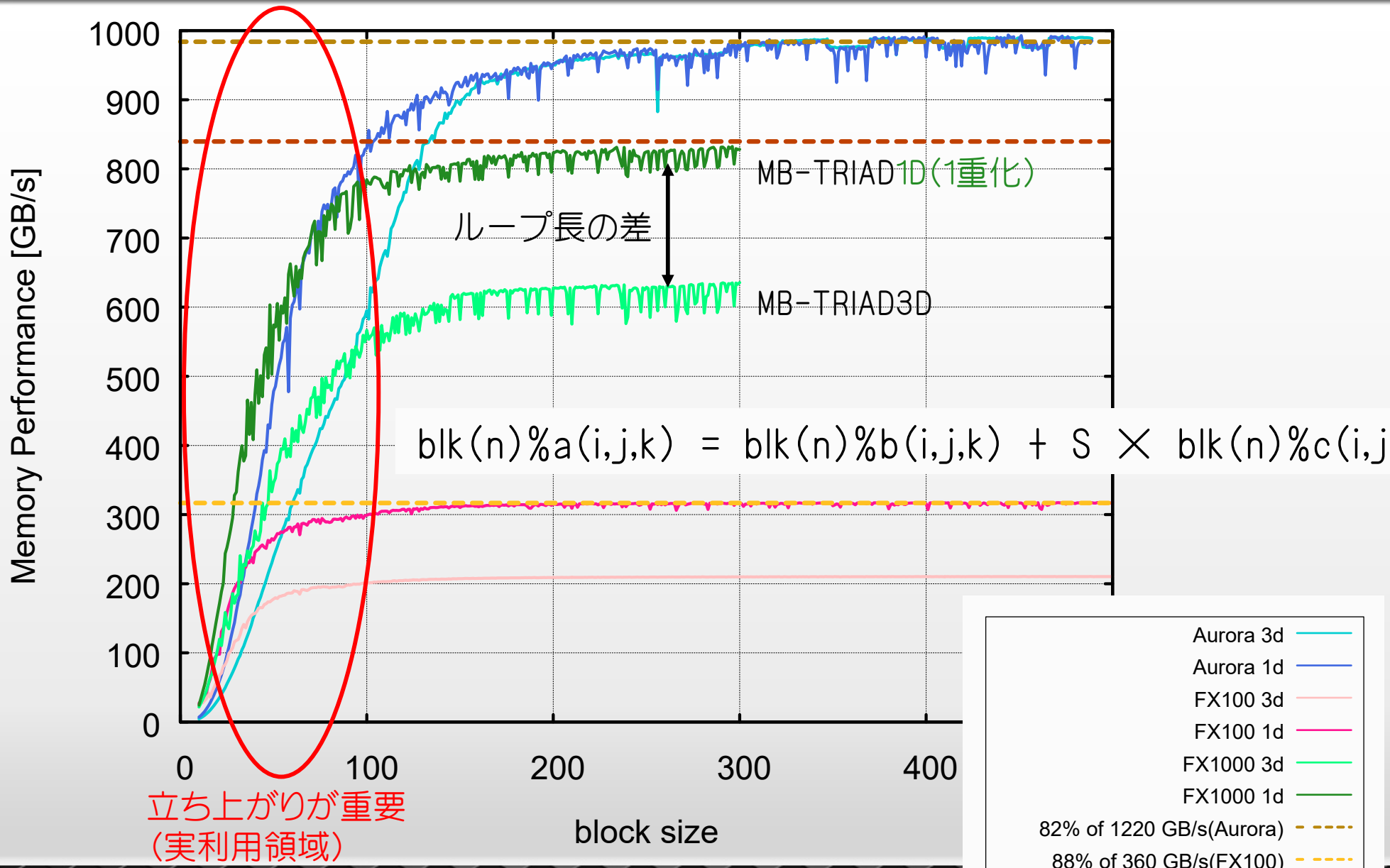
MB-TRIAD1D(1次元化)	MB-TRIAD3D

type blkDataType
  real(8), dimension(:,:,:), allocatable :: a,b,c
end type blkDataType
	type blkDataType
	  real(8), dimension(:,:,:), allocatable :: a,b,c
	end type blkDataType

type(blkDataType), dimension(:), allocatable :: blk do nb=1,nbmax call kernel(blk(nb)%a,blk(nb)%b,blk(nb)%c,…) enddo	←構造体ではなく配列渡し (コンパイラの問題)

subroutine kernel(a,b,c,…)
  real(8), dimension(*) :: a,b,c
  do n
  a(n) = b(n) + scalar*c(n)
  enddo
	subroutine kernel(a,b,c,…)
	  real(8), dimension(:,:,:) :: a,b,c
	  do k; do j; do i;
	  a(i,j,k) = b(i,j,k) + scalar*c(i,j,k)
	  enddo; enddo; enddo

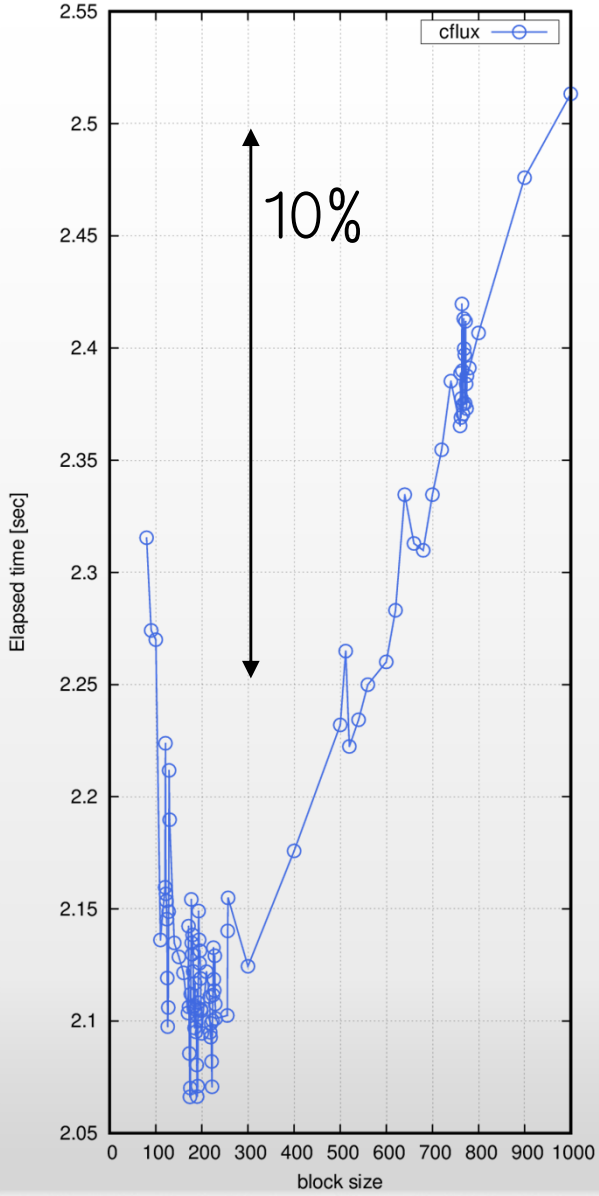
FX1000の実メモリ性能

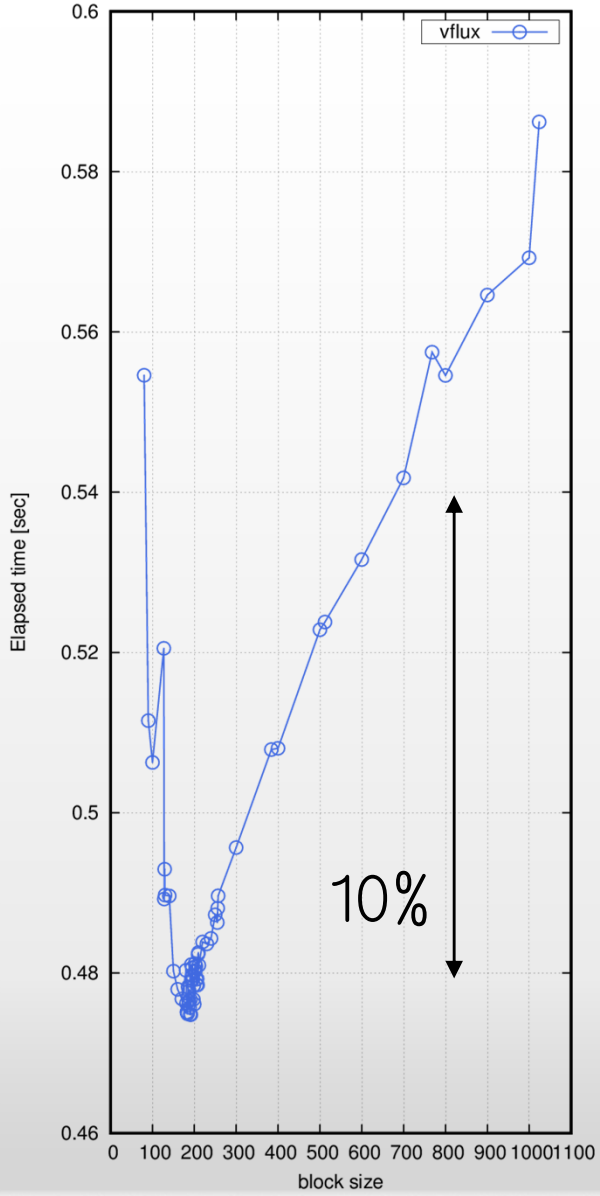


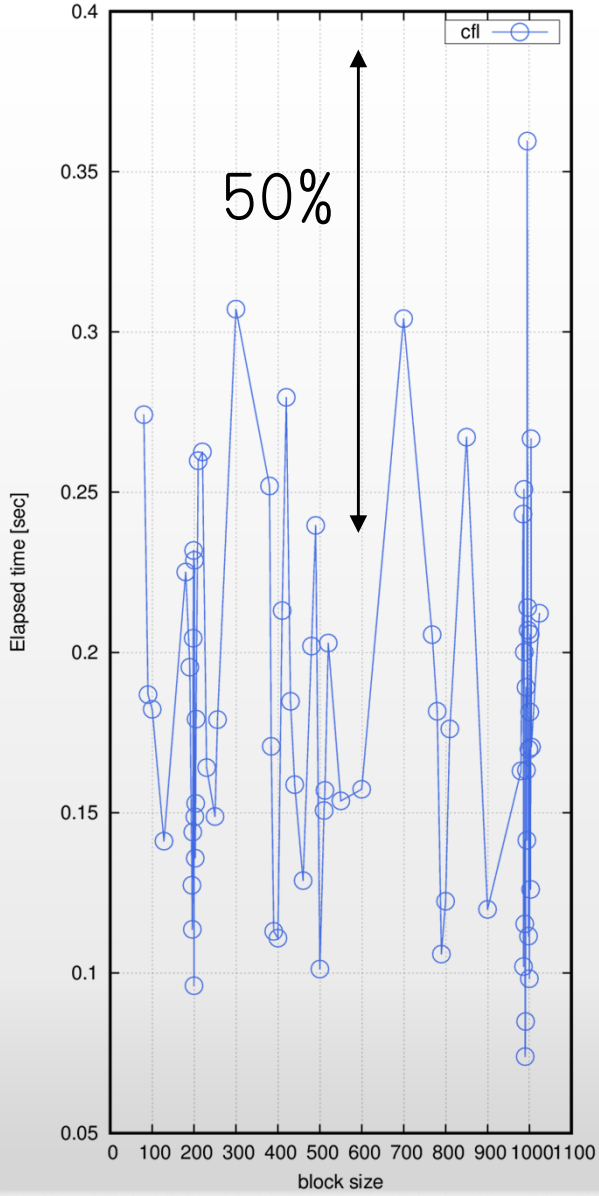
多重ループの高速化(FX1000の課題)

- レジスタ不足(FX100:256→FX1000:32)
  - S/W、SIMD、ZFILL(XFILL)の同時適用が困難
  - ループボディが大きい
- メモリアクセス性能を出し切るためには
  - 連続アクセスかつ長いループ長が必要
    - ✓多重ループの1次元のループ長(<O(100))では不足
      - □XFILL@FX100では256以上
    - ✓キャッシュも含め短ループは性能が出ない
    - ✓実利用領域は短ループが多い
  - 多次元配列の多重ループの場合1重化が必要

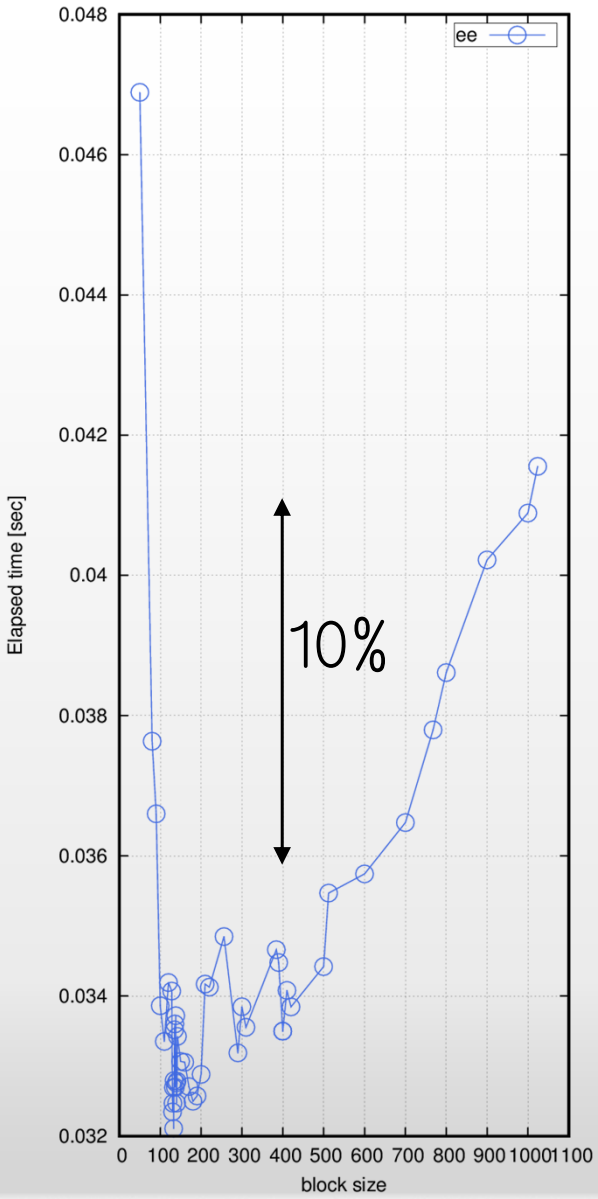
ブロックサイズの最適化(64)

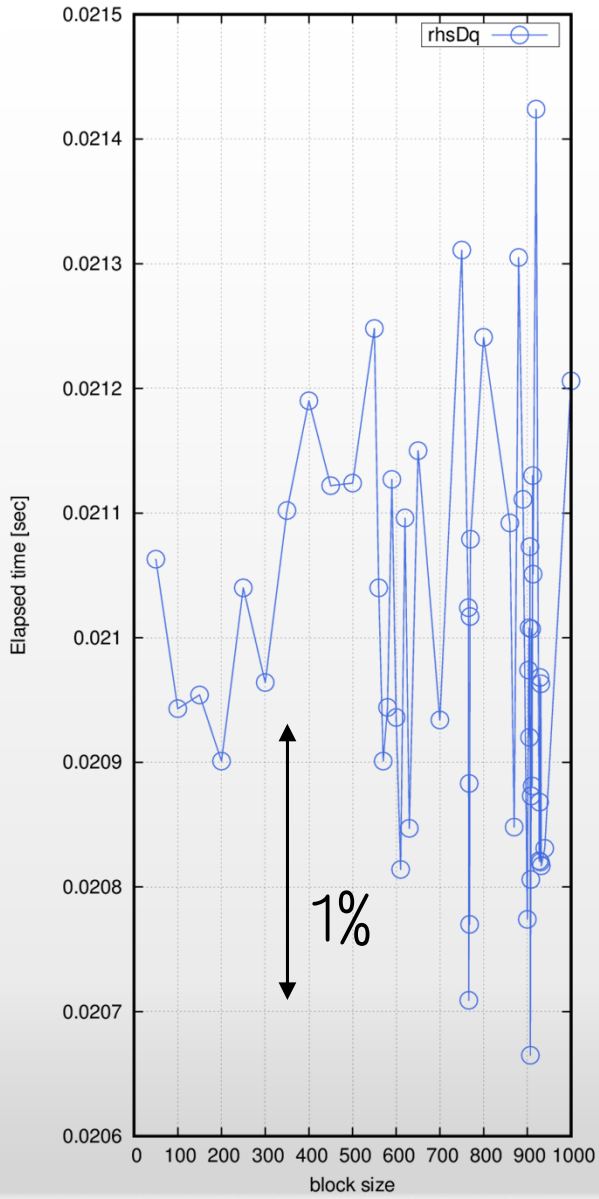






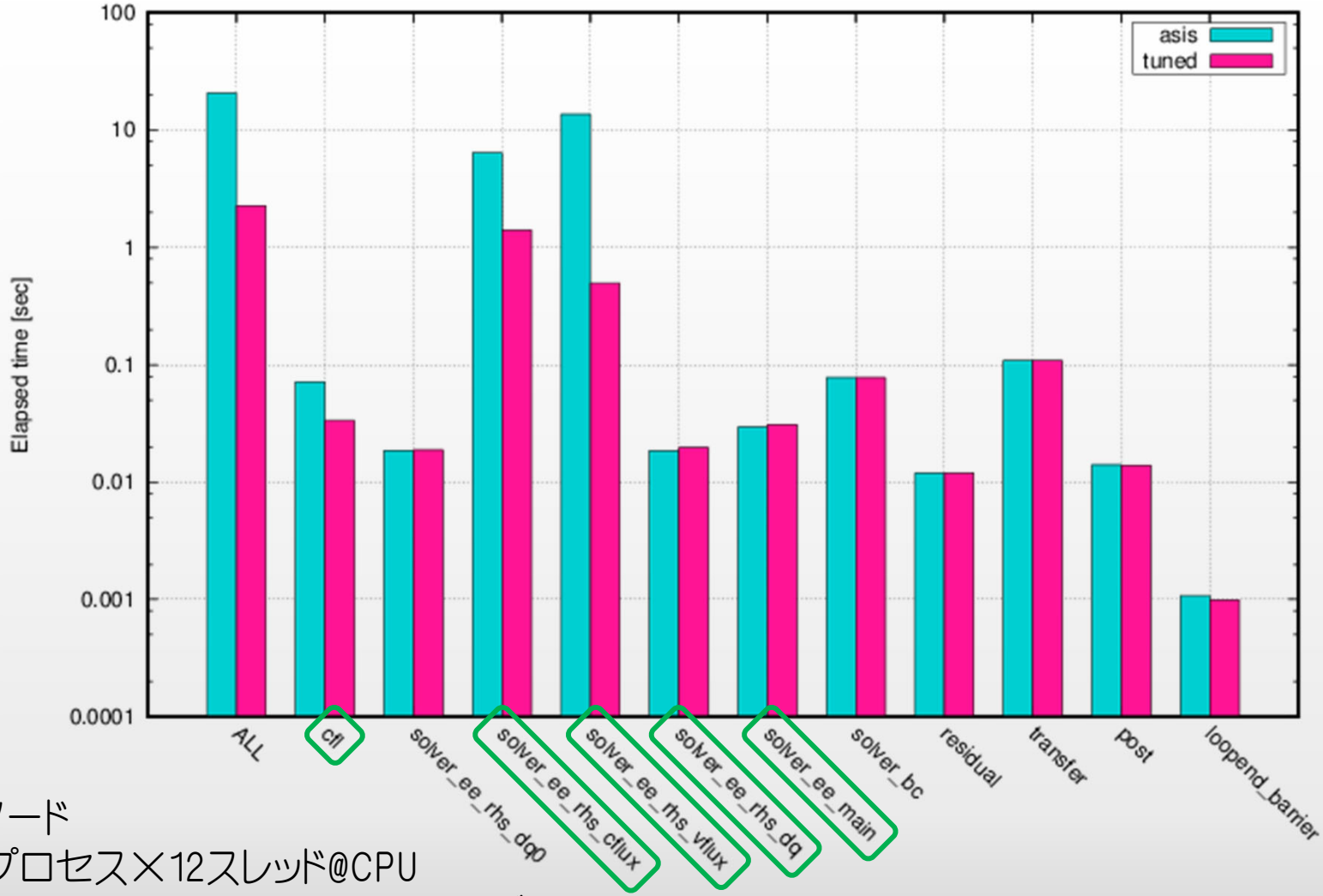
ブロックサイズの最適化(64)





- cflux:174
- vflux:192
- cfl:990
- ee:133
- dq:907

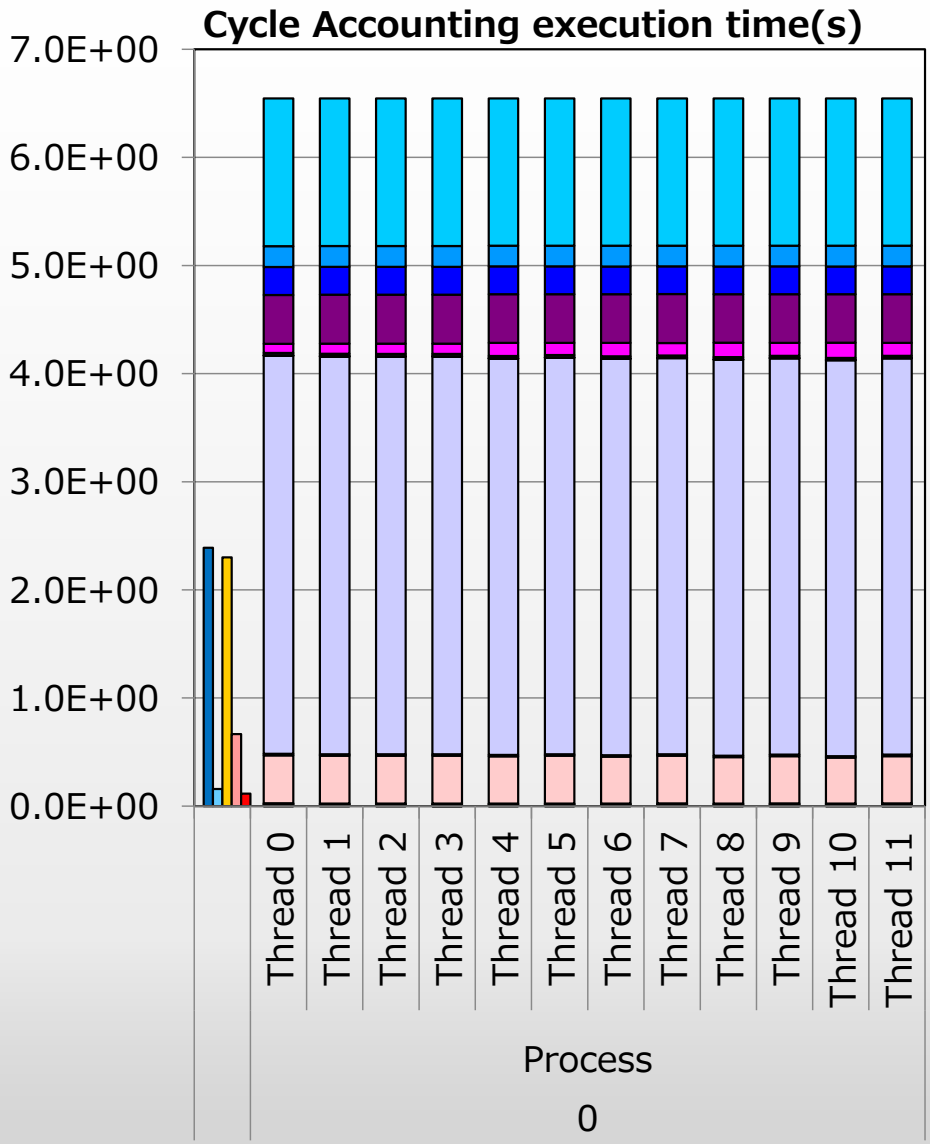
高速化結果

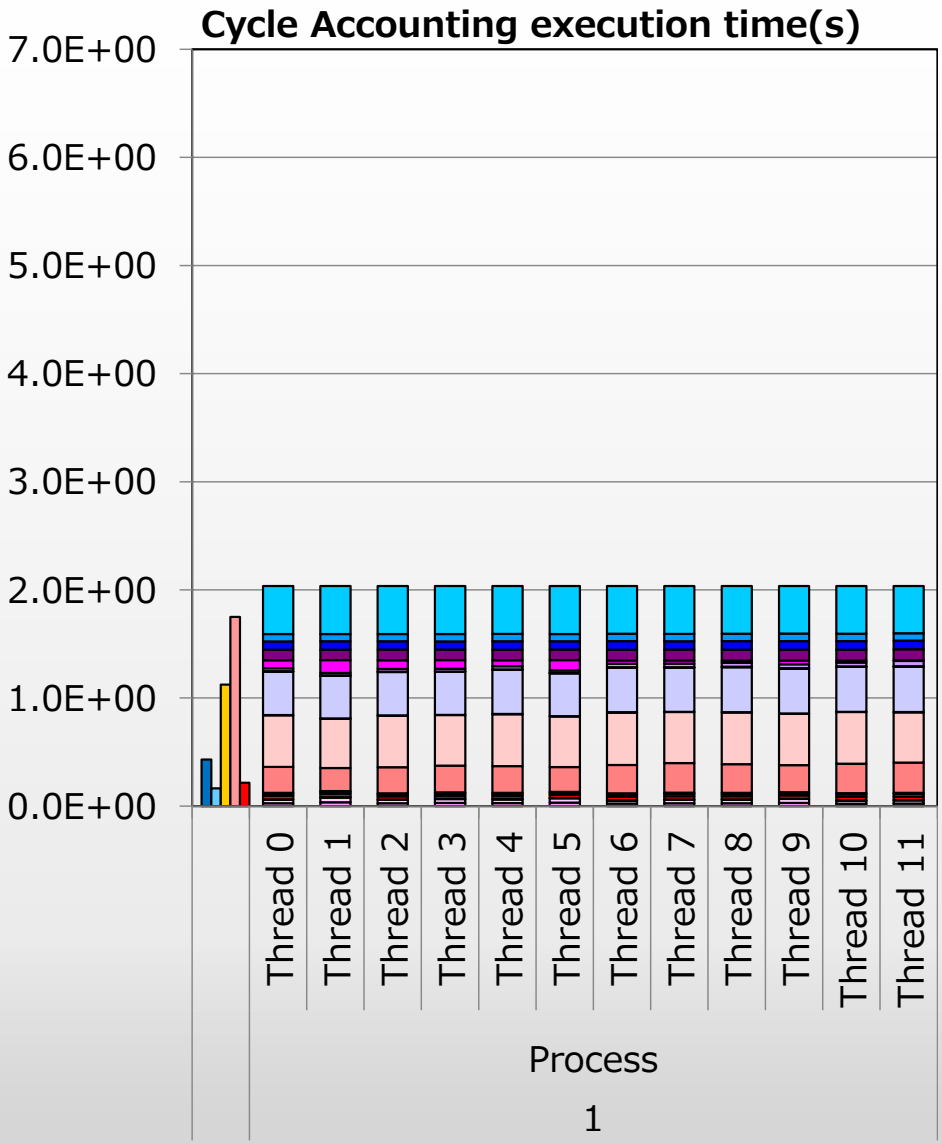


- 1ノード
- 4プロセス×12スレッド@CPU
- セル数:16.7M(20.1M)、64^3×64ブロック、10GB
- 壁率:0.18[%]

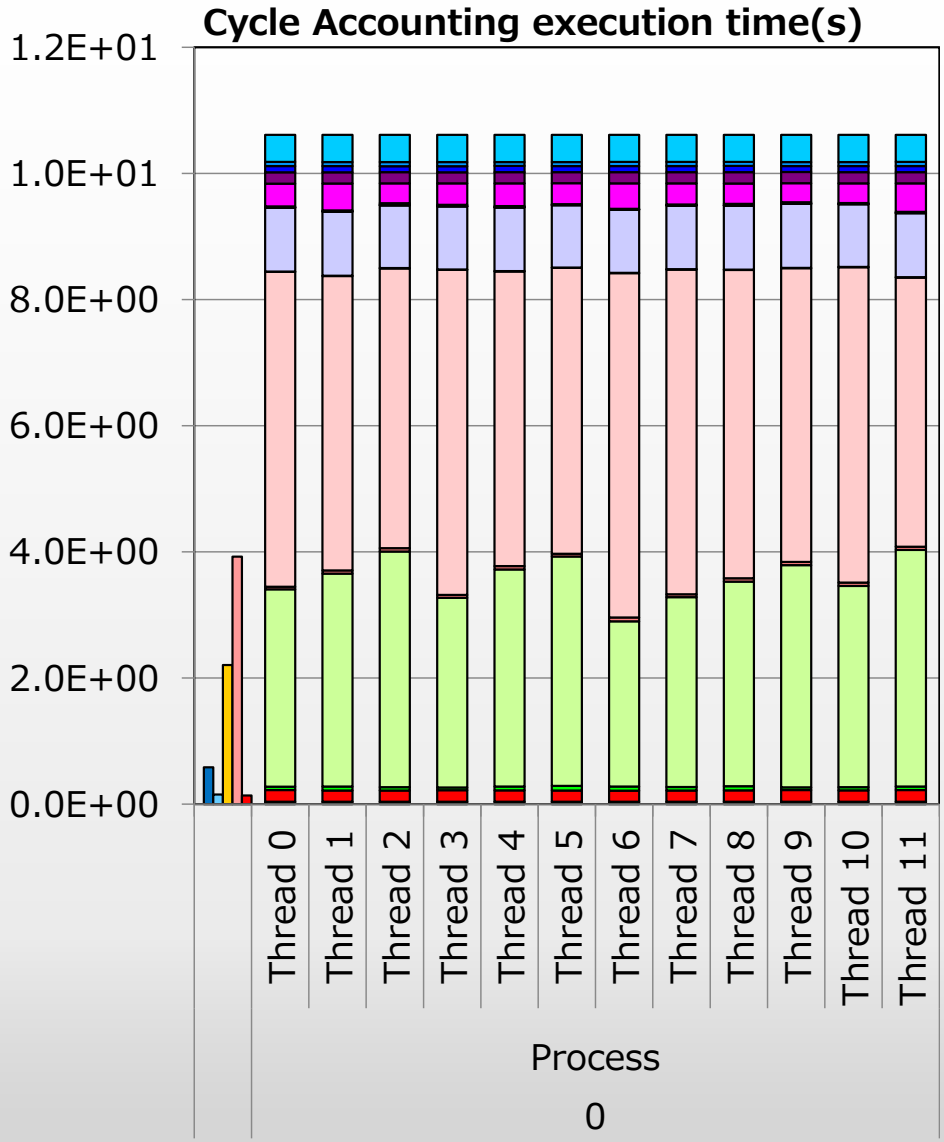
ブロックサイズの最適化を実施

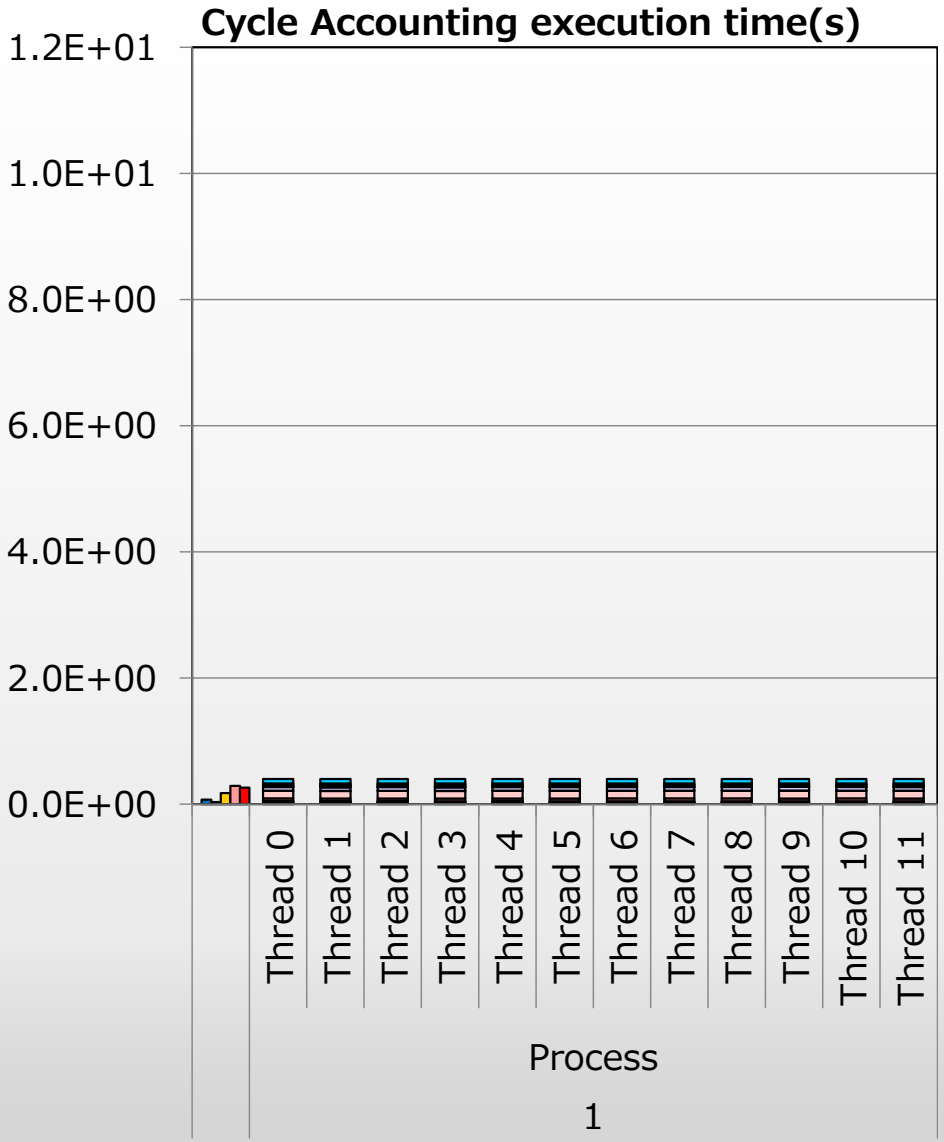
cflux(asis->tuned)





vflux(asis->tuned)





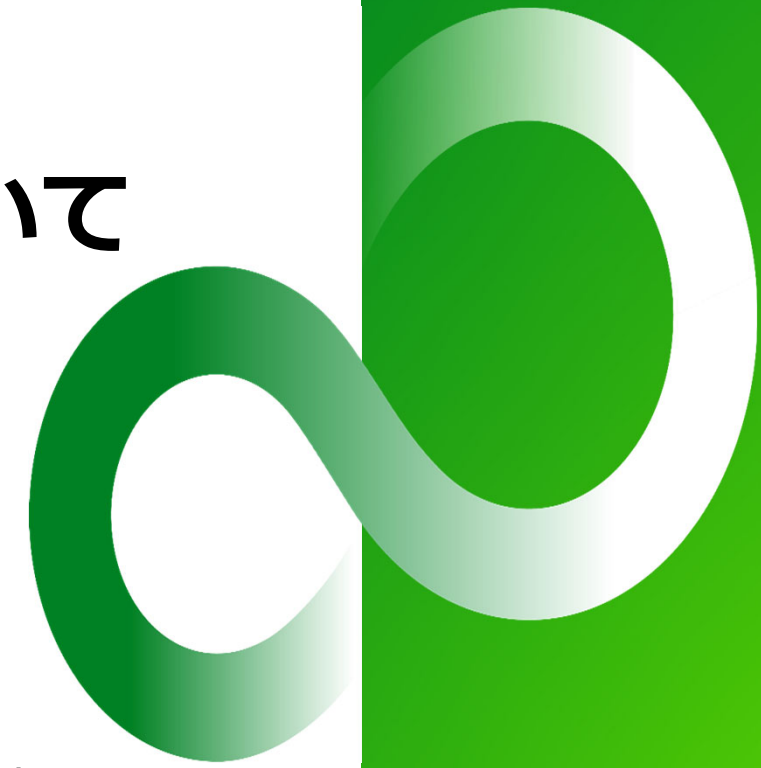
loop_fission_thresholdの影響

- !ocl loop_fission_threshold(N)
  - N:デフォルトは50
- パラスタ
  - N=30～54(連続的に)
  - ブロックサイズ:100～8,000(離散的に)
- 大きな影響はない？

	default(N=50)	cflux最速(N=34)	vflux最速(N=53)
cflux	2.136(166)	2.019(191)	-
vflux	0.3889(160)	-	0.3885(160)

A64FXシステムアプリ性能検討WG

ループ分割機能のAIについて



2022年8月23日
ミッションクリティカルシステム事業本部
UNIX&FXシステム事業部 FX言語ソフトウェア部
渡邉 雄二

ループ分割機能に関するAI

FUJITSU

- 高木先生
  - コンパイラのループ分割を試したが、L2ビジーで思ったよりも性能が向上していない。
  - ➔AI：コンパイラのループ分割機能でL2ビジーを改善できるかどうか確認する。

FUJITSU

コンパイラのループ分割機能

コンパイラのループ分割機能の概要

FUJITSU

- ループ分割機能の起動オプション
  - *-Kloop_fission*
    - コンパイラのループ分割機能を起動する。
    - -O2から誘導される。

- 2 つのループ分割機能
  - 位置指定分割
    - *fission_point*指示行で指定した分割位置でループを分割する機能
      - 多重ループの分割も可能
  - ループ指定分割
    - *loop_fission_target*指示行で指定したループをコンパイラが自動的に分割方法(分割数、分割位置)を判断して分割する機能
      - ループ間のデータ授受のための一時領域へのstore/loadを少なくしつつ、ソフトウェアパイプライン化できるように分割
      - 分割対象は最内ループのみ

位置指定分割機能の概要

- *fission_point*指示行を指定した位置でループを分割

```
do i = 1,n
  x=a(i)+b(i)
!ocl fission_point
  c(i)=d(i)+x
  e(i)=f(i)+x
enddo
```

指示行の位置で分割

```
do i = 1,n
 tmp(i)=a(i)+b(i)
enddo
do i = 1,n
 c(i)=d(i)+tmp(i)
 e(i)=f(i)+tmp(i)
enddo
X=tmp(n)
```

- 指示行の仕様
  - *!ocl fission_point[(n)]*
    - 内側からn番目までのループの分割を指示する。nの範囲は1～6、nを省略すると最内ループを分割(n=1)

```
do k = 1,n
  do j = 1,n
    do i = 1,n
      a(i,j,k)=b(i,j,k)+c(i,j,k)
!ocl fission_point(2)
      d(i,j,k)=e(i,j,k)+f(i,j,k)
    enddo
  enddo
enddo
```

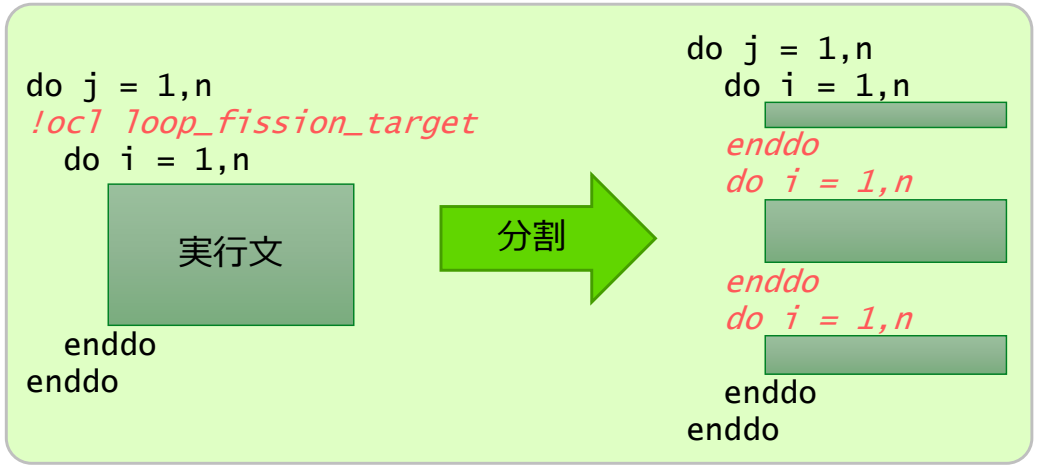
内側から2つのループを分割

```
do k = 1,n
  do j = 1,n
    do i = 1,n

a(i,j,k)=b(i,j,k)+c(i,j,k)
    enddo
  enddo
  do j = 1,n
    do i = 1,n

d(i,j,k)=e(i,j,k)+f(i,j,k)
    enddo
  enddo
enddo
```

ループ指定分割機能の概要

FUJITSU

- *loop_fission_target*指示行で指定したループをコンパイラが自動的に分割方法(分割数、分割位置)を判断して分割
  - ループ間のデータ授受のための一時領域へのstore/loadを少なくしつつ、ソフトウェアパイプライン化できるように分割
  - 分割対象は最内ループのみ



- 指示行の仕様
  - *!ocl loop_fission_target[({cl|ls})]*
    - 分割対象のループと自動分割のアルゴリズム(cl、ls)を指定する。省略時はcl(コンパイル時間を重視)
  - オプショナルな指示行
    - *!ocl loop_fission_threshold(n)*
      - 分割後のループの大きさを指定する。省略値は50。nの範囲は1～100で、大きくする程ループのボディが大きく分割数が少なくなる。
    - *!ocl loop_fission_stripmining[({n|level})]*　＊：キャッシュ溢れ改善のための最適化、ブロッキングの一種
      - 一時領域によるキャッシュ溢れの削減のために分割対象ループにstripminig*を適用する。levelにはキャッシュ溢れの削減対象のキャッシュを指定(l1またはl2)、nで最内ループの回転数の直接指定も可 (nの範囲は2～100,000,000)

FUJITSU

JAXA 高木先生のプログラム

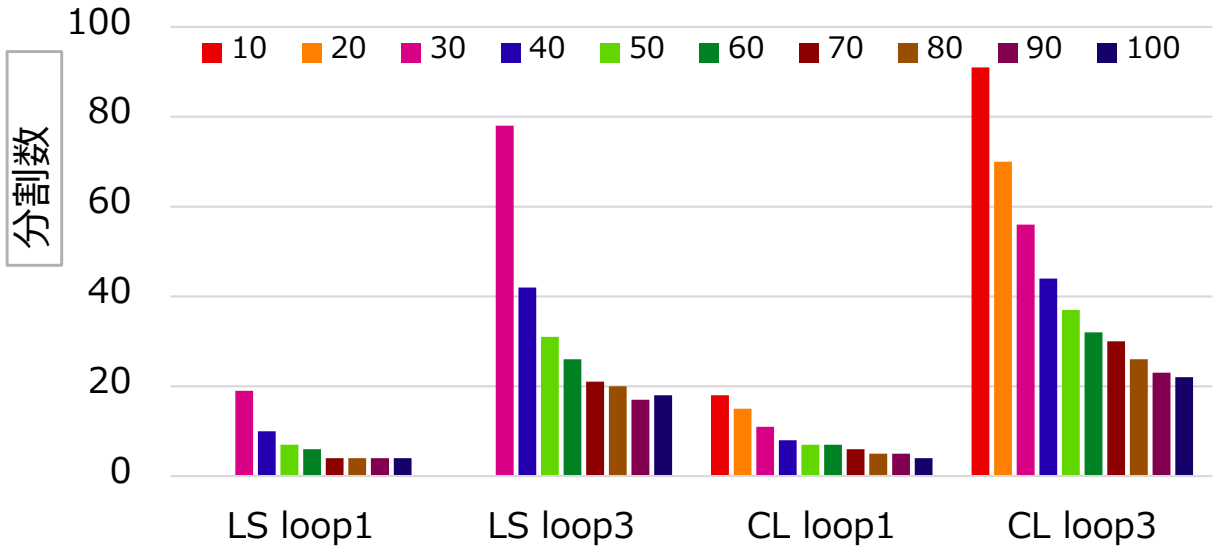
AI：コンパイラのループ分割機能でL2ビジーを改善できるかどうか確認する

ループ指定分割の状況：threshold値と分割数

FUJITSU

- ループのコストから右の２つのループを調査
- ループ指定分割による分割状況は以下の通り

対象ループ	分割アルゴリズム	分割状況										
		threshold値	10	20	30	40	50	60	70	80	90	100
Loop1 (58行)	LS	分割数	分割しない	分割しない	19	10	7	6	4	4	4	4
	CL	分割数	18	15	11	8	7	7	6	5	5	4
Loop3 (116行)	LS	分割数	分割しない	分割しない	78	42	31	26	21	20	17	18
	CL	分割数	91	70	56	44	37	32	30	26	23	22



threshold値と分割数

```
solverConflux.f90
Loop1:
 772       !$omp do
 773       do lb=lbsrt,lbend,lbsize
 774          !ocl simd,swp
 775          !!ocl swp_ireq_rate(200%)
 776          !!ocl swp_freq_rate(200%)
 777          !!ocl swp_preq_rate(200%)
 778          !ocl loop_fission_target(LS)
 779          !ocl
loop_fission_threshold(fissionTH_cflux)
 780          !ocl prefetch_sequential(soft)
 781          do l=lb,lb+lbsize-1          回転数200
 782             !
 783 #include “cflux_dq.inc“   !!  58行
 784             !
 785          enddo
 786       enddo
..
Loop3:
 801       !$omp do
 802       do lb=lbsrt,lbend,lbsize
 803          !ocl simd,swp
 804          !!ocl swp_ireq_rate(200%)
 805          !!ocl swp_freq_rate(200%)
 806          !!ocl swp_preq_rate(200%)
 807          !ocl loop_fission_target(LS)
 808          !ocl
loop_fission_threshold(fissionTH_cflux)
 809          !ocl prefetch_sequential(soft)
 810          do l=lb,lb+lbsize-1          回転数200
 811             !
 812 #include “cflux_kernel.inc“   !! 116行
 813             !
 814          enddo
 815       enddo
```

実行性能とL2ビジーの改善

FUJITSU

○ 実行性能

○ デフォルト(threshold:50)ではCLの方が若干性能が良く、最速はCLのthreshold:90

	分割無し	自動分割			
アルゴリズム	ー	**CL**		LS	
threshold	ー	デフォルト(50)	**90**	デフォルト(50)	30
実行時間(秒)	6.20	***4.85***	***4.73***	4.96	4.84
性能比	1.00	1.28	1.31	1.25	1.28

PAの測定区間：
blk_rhsConFlux_tune
関数の呼出し前後

○ L2ビジー改善

○ ループ指定分割のstripmining機能でL2ビジーの改善を試行

○ L1ミスが削減されL2ビジー率は減少したが、実行時間は増加

○ 最内ループの回転数の減少によるソフトウェアパイプラインの効率低下とstripminingの制御に伴う実行命令数の増加が影響していると推測

アルゴリズム	threshold	stripmining	Execution time (s)	性能比	Effective instruction	Floating-point operation	L1 busy rate (%)	L2 busy rate (%)	L1D miss	L2 miss	ループ回転数 (SIMD化前)
CL	90	無し	4.73	1.00	9.36E+10	8.78E+10	31.39%	*32.68%*	*1.03E+09*	4.04E+08	*200*
		L1	*4.96*	*0.95*	*1.07E+11*	8.78E+10	29.76%	*21.74%*	*7.22E+08*	4.01E+08	*32*

Vlasov5d（プラズマ分布関数コード）及びPIC2d(プラズマ粒子コード)の性能測定

- Vlasov5d: 5次元分布関数を扱うコード
  - 最内ループ長が短い(ex. 40^3*128^2 ~ 24GB)
    ⇒ !OCL SWP_POLICY(SMALL)を指定
  - FX100では、ソフトウェアパイプライン無し＋ループアンローリングが有効だった
    ⇒ FX1000では、ソフトウェアパイプラインの積極活用がより有効
  - !OCL SWP_FREG_RATE(r)により、浮動小数点レジスタ利用率を調整することで実効性能が15%まで向上 ⇒ ただし、調整は職人技
- PIC2d: 粒子が2次元格子内を自由に動き回る(particle-in-cell)コード、粒子の位置から電磁場格子へのデータアクセスがランダム
  - 粒子データをソートして連続的なメモリアクセスにする、また外側ループを格子、最内ループを粒子で回すことで、実効性能20%を達成
  - 最内ループの計算量が多く、コンパイラのソフトウェアパイプラインが効かない
    ⇒ 手動ループ分割により、実効性能30%を達成
  - !OCL FISSION_POINTの挿入により、ソース改変が最小限でループ分割可能

事例）
外側ループのSIMD化

GF-based NN predictor kernel

- 周辺グリッドの情報をローカルな節点に畳み込む計算
- 節点に関する3重ループ (ループ長: 長) の内部に周辺グリッドに関する3重ループ (ループ長: 短)が含まれているシンプルな構造
- 精度は全て単精度
- 外側のk, jループに対して OMP collapse(2)をかけ, iループ方向にSIMD化をかけたい

nef=3
nd=3(コンパイル時既知)

```
do k=1+nef-1,nez+1-nef+1
do j=1+nef-1,ney+1-nef+1
do i=1+nef-1,nex+1-nef+1
  we1=wei(1,i,j,k)
  we2=wei(2,i,j,k)
  …
  we8=wei(8,i,j,k)

  grs1=0.0
  grs2=0.0
  grs3=0.0
  do k1=1,nd*2+1
  do j1=1,nd*2+1
  do i1=1,nd*2+1
    rs1=rs(i1+i-nd-1,j1+j-nd-1,k1+k-nd-1,1)
    rs2=rs(i1+i-nd-1,j1+j-nd-1,k1+k-nd-1,2)
    rs3=rs(i1+i-nd-1,j1+j-nd-1,k1+k-nd-1,3)
    cocs1=cocs(i1,j1,k1,1)
    cocs2=cocs(i1,j1,k1,2)
    cocs3=cocs(i1,j1,k1,3)
    ww1=we1+we2*cocs1+we3*cocs2+we4*cocs3
    ww2=we5+we6*cocs1+we7*cocs2+we8*cocs3
    grs1=grs1+rs1*ww1*coe1s(i1,j1,k1,1)
    grs1=grs1+rs2*ww2*coe1s(i1,j1,k1,2)
    grs1=grs1+rs3*ww2*coe1s(i1,j1,k1,3)
    grs2=grs2+rs1*ww2*coe2s(i1,j1,k1,1)
    grs2=grs2+rs2*ww1*coe2s(i1,j1,k1,2)
    grs2=grs2+rs3*ww2*coe2s(i1,j1,k1,3)
    grs3=grs3+rs1*ww2*coe3s(i1,j1,k1,1)
    grs3=grs3+rs2*ww2*coe3s(i1,j1,k1,2)
    grs3=grs3+rs3*ww1*coe3s(i1,j1,k1,3)
  enddo
  enddo
  enddo

  zs(i,j,k,1)=grs1
  zs(i,j,k,2)=grs2
  zs(i,j,k,3)=grs3
enddo
enddo
enddo
```

富士通コンパイラでの最適化

- 実行環境
  - PRIMEHPC FX700 @ 理化学研究所
    - プロセッサ名: A64FX
    - プロセッサ数: 1 (48)
    - 動作周波数: 2.2 GHz
    - 理論演算性能: 3.3792 TFLOPS (倍精度)
    - メモリバンド幅: 1024 GB/s
- コンパイラ
  - `singularity shell /cloud_opt/singularity/x86_64/fx700xos_v1.1.0.sif`
  - `mpifrtpx frtpx (FRT) 4.2.0 20200615`
- コンパイルオプション
  - `-Klargepage -Kfast,openmp,ocl,parallel -Kautoobjstack,temparraystack -Kloop_nofission -Nfjomplib -Koptmsg=2 -V -Nlst=t`
- 実行時環境変数
  - `export OMP_NUM_THREADS=12`（x4 MPIプロセス）
  - `export PARALLEL=12`

```
!$OMP PARALLEL DO default(none), collapse(2)
do k=1+nef-1,nez+1-nef+1
do j=1+nef-1,ney+1-nef+1
!OCL NORECURRENCE
!OCL SIMD
do i=1+nef-1,nex+1-nef+1
  we1=wei(1,i,j,k)
  we2=wei(2,i,j,k)
  …
  we8=wei(8,i,j,k)

  grs1=0.0
  grs2=0.0
  grs3=0.0
  do k1=1,nd*2+1
  do j1=1,nd*2+1
  do i1=1,nd*2+1
    rs1=rs(i1+i-nd-1,j1+j-nd-1,k1+k-nd-1,1)
    rs2=rs(i1+i-nd-1,j1+j-nd-1,k1+k-nd-1,2)
    rs3=rs(i1+i-nd-1,j1+j-nd-1,k1+k-nd-1,3)
    cocs1=cocs(i1,j1,k1,1)
    cocs2=cocs(i1,j1,k1,2)
    cocs3=cocs(i1,j1,k1,3)
    ww1=we1+we2*cocs1+we3*cocs2+we4*cocs3
    ww2=we5+we6*cocs1+we7*cocs2+we8*cocs3
    grs1=grs1+rs1*ww1*coe1s(i1,j1,k1,1)
    grs1=grs1+rs2*ww2*coe1s(i1,j1,k1,2)
    grs1=grs1+rs3*ww2*coe1s(i1,j1,k1,3)
    grs2=grs2+rs1*ww2*coe2s(i1,j1,k1,1)
    grs2=grs2+rs2*ww1*coe2s(i1,j1,k1,2)
    grs2=grs2+rs3*ww2*coe2s(i1,j1,k1,3)
    grs3=grs3+rs1*ww2*coe3s(i1,j1,k1,1)
    grs3=grs3+rs2*ww2*coe3s(i1,j1,k1,2)
    grs3=grs3+rs3*ww1*coe3s(i1,j1,k1,3)
  enddo
  enddo
  enddo

  zs(i,j,k,1)=grs1
  zs(i,j,k,2)=grs2
  zs(i,j,k,3)=grs3
enddo
enddo
enddo
!$OMP END PARALLEL DO
```

富士通コンパイラでの最適化レポート

- **実際にSIMD化がかかったのは内側のi1ループ**
  - **ループ長が7**しかないため性能が出ず

.lstファイル

```
1023                    !$OMP PARALLEL DO default(none), collapse(2)
1024                    !$OMP& shared(nef,nez,ney,nex,wei,rs,zs,cocs,nnum,
1025                    !$OMP& ns1,ns2,coe1s,coe2s,coe3s),
1026                    !$OMP& private(i,j,k,i1,is,k1,j1,iit,ww1,ww2,
1027                    !$OMP&
grs1,grs2,grs3,itt,rs1,rs2,rs3,cocs1,cocs2,cocs3,
1028                    !$OMP& we1,we2,we3,we4,we5,we6,we7,we8)
1029     1   p                do k=1+nef-1,nez+1-nef+1
1030     2   p                do j=1+nef-1,ney+1-nef+1
1031     2                    !OCL NORECURRENCE
1032     2                    !OCL SIMD
1033     3   p                do i=1+nef-1,nex+1-nef+1
1034     3   p                  we1=wei(1,i,j,k)
1035     3   p                  we2=wei(2,i,j,k)
1036     3   p                  we3=wei(3,i,j,k)
1037     3   p                  we4=wei(4,i,j,k)
1038     3   p                  we5=wei(5,i,j,k)
1039     3   p                  we6=wei(6,i,j,k)
1040     3   p                  we7=wei(7,i,j,k)
1041     3   p                  we8=wei(8,i,j,k)
1042     3
1043     3   p                  grs1=0.0
1044     3   p                  grs2=0.0
```

```
                                <<< Loop-information Start >>>
                                <<<  [OPTIMIZATION]
                                <<<    FULL UNROLLING
                                <<< Loop-information  End >>>
1047     5   p   f                do j1=1,nd*2+1
                                <<< Loop-information Start >>>
                                <<<  [OPTIMIZATION]
                                <<<    SIMD(VL: 8)
                                <<< Loop-information  End >>>
1048     6   p   fv                 do i1=1,nd*2+1
1049     6   p   fv                   rs1=rs(i1+i-nd-1,j1+j-nd-1,k1+k-nd-
1,1)
1050     6   p   fv                   rs2=rs(i1+i-nd-1,j1+j-nd-1,k1+k-nd-
1,2)
1051     6   p   fv                   rs3=rs(i1+i-nd-1,j1+j-nd-1,k1+k-nd-
1,3)
1052     6   p   fv                   cocs1=cocs(i1,j1,k1,1)
1053     6   p   fv                   cocs2=cocs(i1,j1,k1,2)
1054     6   p   fv                   cocs3=cocs(i1,j1,k1,3)
1055     6   p   fv
ww1=we1+we2*cocs1+we3*cocs2+we4*cocs3
1056     6   p   fv
ww2=we5+we6*cocs1+we7*cocs2+we8*cocs3
1057     6   p   fv                   grs1=grs1+rs1*ww1*coe1s(i1,j1,k1,1)
1058     6   p   fv                   grs1=grs1+rs2*ww2*coe1s(i1,j1,k1,2)
1059     6   p   fv                   grs1=grs1+rs3*ww2*coe1s(i1,j1,k1,3)
1060     6   p   fv                   grs2=grs2+rs1*ww2*coe2s(i1,j1,k1,1)
1061     6   p   fv                   grs2=grs2+rs2*ww1*coe2s(i1,j1,k1,2)
1062     6   p   fv                   grs2=grs2+rs3*ww2*coe2s(i1,j1,k1,3)
1063     6   p   fv                   grs3=grs3+rs1*ww2*coe3s(i1,j1,k1,1)
1064     6   p   fv                   grs3=grs3+rs2*ww2*coe3s(i1,j1,k1,2)
1065     6   p   fv                   grs3=grs3+rs3*ww1*coe3s(i1,j1,k1,3)
1066     6   p   fv                 enddo
1067     5   p   f                  enddo
1068     4   p                      enddo
1069     3   p                      zs(i,j,k,1)=grs1
1070     3   p                      zs(i,j,k,2)=grs2
1071     3   p                      zs(i,j,k,3)=grs3
1072     3   p                    enddo
1073     2   p                    enddo
1074     1   p                    enddo
1075                     !$OMP END PARALLEL DO
1076                          end
```

```
!$OMP PARALLEL DO default(none), collapse(2)
do k=1+nef-1,nez+1-nef+1
do j=1+nef-1,ney+1-nef+1
!OCL NORECURRENCE
!OCL SIMD
do i=1+nef-1,nex+1-nef+1
  we1=wei(1,i,j,k)
  we2=wei(2,i,j,k)
  …
  we8=wei(8,i,j,k)

  grs1=0.0
  grs2=0.0
  grs3=0.0
  do k1=1,nd*2+1
  do j1=1,nd*2+1
  do i1=1,nd*2+1
    rs1=rs(i1+i-nd-1,j1+j-nd-1,k1+k-nd-1,1)
    rs2=rs(i1+i-nd-1,j1+j-nd-1,k1+k-nd-1,2)
    rs3=rs(i1+i-nd-1,j1+j-nd-1,k1+k-nd-1,3)
    cocs1=cocs(i1,j1,k1,1)
    cocs2=cocs(i1,j1,k1,2)
    cocs3=cocs(i1,j1,k1,3)
    ww1=we1+we2*cocs1+we3*cocs2+we4*cocs3
    ww2=we5+we6*cocs1+we7*cocs2+we8*cocs3
    grs1=grs1+rs1*ww1*coe1s(i1,j1,k1,1)
    grs1=grs1+rs2*ww2*coe1s(i1,j1,k1,2)
    grs1=grs1+rs3*ww2*coe1s(i1,j1,k1,3)
    grs2=grs2+rs1*ww2*coe2s(i1,j1,k1,1)
    grs2=grs2+rs2*ww1*coe2s(i1,j1,k1,2)
    grs2=grs2+rs3*ww2*coe2s(i1,j1,k1,3)
    grs3=grs3+rs1*ww2*coe3s(i1,j1,k1,1)
    grs3=grs3+rs2*ww2*coe3s(i1,j1,k1,2)
    grs3=grs3+rs3*ww1*coe3s(i1,j1,k1,3)
  enddo
  enddo
  enddo

  zs(i,j,k,1)=grs1
  zs(i,j,k,2)=grs2
  zs(i,j,k,3)=grs3
enddo
enddo
enddo
!$OMP END PARALLEL DO
```

Arm C Language Extensions (ACLE)での実装

- **Intrinsic (ACLE)での記述により手動でSIMD化**

```
#pragma omp parallel for collapse(3)
for (k = 1+nef-1 - 1; k < nez+1-nef+1; ++k) {
for (j = 1+nef-1 - 1; j < ney+1-nef+1; ++j) {
for (i = 1+nef-1 - 1; i < nex+1-nef+1; i += 32) {
  pred1 = svwhilelt_b32_s32(i, nex+1-nef+1);
  pred2 = svwhilelt_b32_s32(i+16, nex+1-nef+1);

  we1_v1 = svld1_f32(pred1, wei + (nex+1)*(ney+1)*(nez+1)*0 + (nex+1)*(ney+1)*k + (nex+1)*j + i);
  we2_v1 = svld1_f32(pred1, wei + (nex+1)*(ney+1)*(nez+1)*1 + (nex+1)*(ney+1)*k + (nex+1)*j + i);
  …
  we8_v1 = svld1_f32(pred1, wei + (nex+1)*(ney+1)*(nez+1)*7 + (nex+1)*(ney+1)*k + (nex+1)*j + i);
  we1_v2 = svld1_f32(pred2, wei + (nex+1)*(ney+1)*(nez+1)*0 + (nex+1)*(ney+1)*k + (nex+1)*j + i+16);
  we2_v2 = svld1_f32(pred2, wei + (nex+1)*(ney+1)*(nez+1)*1 + (nex+1)*(ney+1)*k + (nex+1)*j + i+16);
  …
  we8_v2 = svld1_f32(pred2, wei + (nex+1)*(ney+1)*(nez+1)*7 + (nex+1)*(ney+1)*k + (nex+1)*j + i+16);

  …
```

	Elapsed time (s)	TFLOPS	Peak to FP64
i1ループをSIMD化(コンパイラによる自動並列化)	0.3452	0.07 x4 = 0.28	8.3%
iループをSIMD化(ACLE)	**0.0598**	**0.42 x4 = 1.68**	**49.7%**

- **最内ループ以外のループのSIMD化で約5.8倍の高速化**

```
!$OMP PARALLEL DO default(none), collapse(2)
do k=1+nef-1,nez+1-nef+1
do j=1+nef-1,ney+1-nef+1
!OCL NORECURRENCE
!OCL SIMD
do i=1+nef-1,nex+1-nef+1
  we1=wei(1,i,j,k)
  we2=wei(2,i,j,k)
  …
  we8=wei(8,i,j,k)

  grs1=0.0
  grs2=0.0
  grs3=0.0
  do k1=1,nd*2+1
  do j1=1,nd*2+1
  do i1=1,nd*2+1
    rs1=rs(i1+i-nd-1,j1+j-nd-1,k1+k-nd-1,1)
    rs2=rs(i1+i-nd-1,j1+j-nd-1,k1+k-nd-1,2)
    rs3=rs(i1+i-nd-1,j1+j-nd-1,k1+k-nd-1,3)
    cocs1=cocs(i1,j1,k1,1)
    cocs2=cocs(i1,j1,k1,2)
    cocs3=cocs(i1,j1,k1,3)
    ww1=we1+we2*cocs1+we3*cocs2+we4*cocs3
    ww2=we5+we6*cocs1+we7*cocs2+we8*cocs3
    grs1=grs1+rs1*ww1*coe1s(i1,j1,k1,1)
    grs1=grs1+rs2*ww2*coe1s(i1,j1,k1,2)
    grs1=grs1+rs3*ww2*coe1s(i1,j1,k1,3)
    grs2=grs2+rs1*ww2*coe2s(i1,j1,k1,1)
    grs2=grs2+rs2*ww1*coe2s(i1,j1,k1,2)
    grs2=grs2+rs3*ww2*coe2s(i1,j1,k1,3)
    grs3=grs3+rs1*ww2*coe3s(i1,j1,k1,1)
    grs3=grs3+rs2*ww2*coe3s(i1,j1,k1,2)
    grs3=grs3+rs3*ww1*coe3s(i1,j1,k1,3)
  enddo
  enddo
  enddo

  zs(i,j,k,1)=grs1
  zs(i,j,k,2)=grs2
  zs(i,j,k,3)=grs3
enddo
enddo
enddo
!$OMP END PARALLEL DO
```

A64FXシステムアプリ性能検討WG

東大）藤田先生のリゾルバーの性能問題について

2022年02月24日, 2022年06月14日(追加報告)

ミッションクリティカルシステム事業本部
UNIX&FXシステム事業部 FX言語ソフトウェア部

東大)藤田様からの外側ループのSIMD化(振り返り)

FUJITSU

- GF-based NN predictor
  - カーネルは外側のdo k, do j, do i のループと、内側のdo k1, do j1, do i1 からなる６重ループ
  - 外側 k, j ループに対して OMP collapse(2)をかけ、iループ方向にSIMD化をしたい。
- 富士通コンパイラの最適化レポート
  - 最内ループ'i1'でSIMD化したがループ長7のため性能が出ず
- ACLE(intrinsic)の記述により手動でSIMD化(iのloop)
  → As is 対して 約5.8倍の高速化
- Intelコンパイラでの最適化
  - 狙い通りに ！DIR＄　SIMDでのループ'I' でSIMD化
- **富士通コンパイラで'I'のループでSIMD化したい。**

```
 19 !$OMP PARALLEL DO default(none),collapse(2)
 20 !$OMP& shared(nef,nez,ney,nex,nd,wei,rs,zs,cocs,
 21 !$OMP& coe1s,coe2s,coe3s),
 22 !$OMP& private(i,j,k,i1,j1,k1,ww1,ww2,
 23 !$OMP&
grs1,grs2,grs3,rs1,rs2,rs3,cocs1,cocs2,cocs3,
 24 !$OMP& we1,we2,we3,we4,we5,we6,we7,we8)
 25       do k=1+nef-1,nez+1-nef+1
 26       do j=1+nef-1,ney+1-nef+1
 27 !!OCL NORECURRENCE
 28 !!OCL SIMD
 29       do i=1+nef-1,nex+1-nef+1
 30         we1=wei(1,i,j,k)
… (略) …
 38 c
 39         grs1=0.0
 40         grs2=0.0
 41         grs3=0.0
 42 !!OCL NOSIMD
 43 !!OCL UNROLL('full')
 44         do k1=1,nd*2+1
 45 !!OCL NOSIMD
 46 !!OCL UNROLL('full')
 47         do j1=1,nd*2+1
 48 !!OCL SIMD
 49 !!OCL UNROLL('full')
 50         do i1=1,nd*2+1
 51           rs1=rs(i1+i-nd-1,j1+j-nd-1,k1+k-nd-1,1)
```

多重ループの外側ループでのSIMD化について

FUJITSU

○【課題】 多重ループの外側でSIMD化したい

問題を縮小化した例

```
do i=1,n
   we1=wei(1,i)
   we2=wei(2,i)
   s = 0.0
   do k1=1,nd*2+1
      do j1=1,nd*2+1
         do i1=1,nd*2+1
            rs1=rs(i1+i-nd-1,j1-nd-1,k1-nd-1)
            s=s+rs1*we1+we2*cocs(i1,j1,k1)
         end do
      end do
   end do
   zs(i)=s
end do
```

解析の範囲とはしている。

【阻害原因】 富士通のSIMD化機能は最内のループを対象としているためSIMDできません。

OpenMP SIMDも同様です。SIMD化の解析処理では、上記のk1,j1,i1のループを一重化やループ交換を目的としてSIMD化の解析対象としています。
しかしながら、最終的に最内を対象としているためSIMD化できていません。

外側ループのSIMD化については、規模の大きい新規開発となるため、今のところ改善の計画はありません。

外側SIMD以外のチューニングの結果

Short vectorの対処

FUJITSU

- ループ長が短い場合の対処として以下が考えられます
  - !OCL CLONE (n==定数) を活用
    - ループの回転数に偏りがあり回転数が少ない場合に、専用のループを作成します。

```
!OCL CLONE(n==4)
do i=1,n
  a(i) = …
end do
```

→

```
if (n==4) then
  do i=1,4
    a(i) = …
  end do
else
 do i=1,n
    a(i) = …
  end do
endif
```

調査の過程で、
OCL CLONEとOpenMPの排他の
課題を認識しました。

  - !OCL ITERATIONS(max=m, min=m, avg=a)の活用
    - ループ回転数のmax/min/avgを指定します。
      ループのunrollingやsoftware pipeliningの重ね具合を調整します。
  - !OCL SIMD_NOREDUNDUNT_VLの活用
    - 例えば回転数が９であり、SIMD長が８の時に１回転分が余ります。
      その部分がSIMD化されると遅くなります。
    - この場合に、余りループのSIMD化を抑止します。

性能改善の結果

FUJITSU

○ ループの回転数を単純に定数に置換

試行	経過時間	性能差	備考
基本オリジナル性能	1.288s	-	5回測定
ソースで変数ndを４で置換	**0.941s**	**1.368倍**	
上記+内側3重ループをfull-unroll	4.953s	0.26倍	SIMDできず遅い(レジスタ不足)
上記+内側2重ループをfull-unroll	4.275s	0.3倍	SIMDはできたが遅い(レジスタ不足)

○ OpenMPとの組み合わせでOCL CLONEが適応できなかったので、
可能性としての評価をソース書換で実施

試行	経過時間	性能差	
ソースで関数全体でclone	**0.938s**	**1.372倍**	
ソースでSIMD範囲だけでclone	0.971s	1.325倍	

○ その他のOCLの適用

試行	経過時間	性能差	
ocl iterations(avg=9)	1.287s	0.999倍	
ocl iterations(max=9)	1.211s	1.06倍	少し効果あり
ocl simd_noredunduntant_vl	2.651s	0.49倍	劣化

○ 結論
- ○ コンパイラオプションやOCLでの改善は最大で1.3倍程度の見込みに留まり、
先生の作成されたACLE(intrinsic)版の効果の約5.8倍には大きく及びませんでした。
- ○ OpenMP と　!OCL CLONEの課題を認識しました。

事例）
AI計算の高速化

AI計算の高速化

昔のChainerとconvolution性能の評価

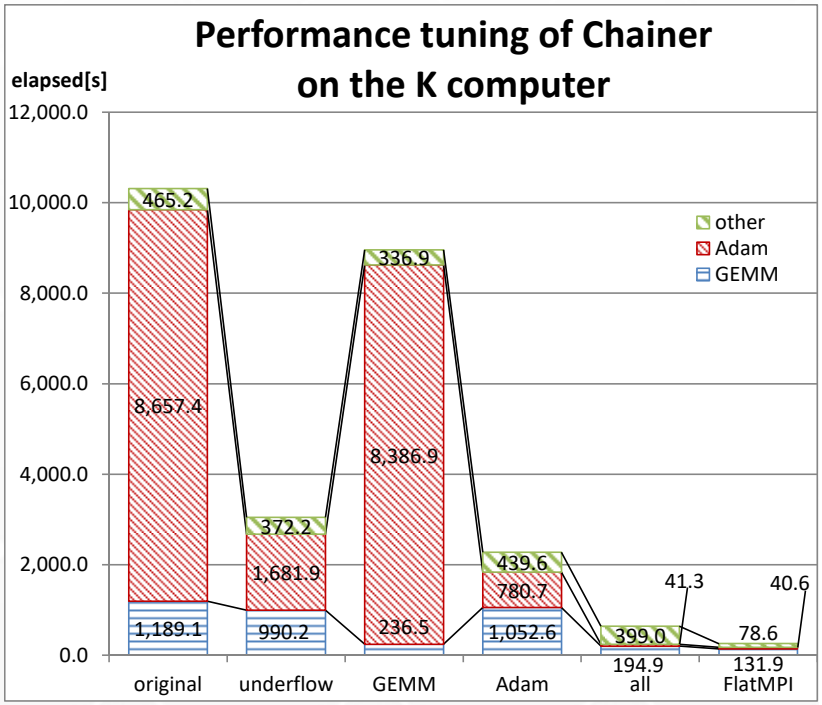
R-CCS 運用技術部門 ソフトウェア開発技術ユニット 黒田明義

2022年(令和4年)2月24日(木)

富岳に向けてのChainerの高速化

富岳に向けてのChainerの高速化

- **背景**
  - 2014年から行なわれてきたFS2020富岳開発プロジェクトで扱ってきた9本のターゲットアプリに加え，**2018年の理研改組を機**に，富岳の**汎用CPU特性を生かした大規模AI計算の実現**を目指して，（当時の南ユニットにて）調査並びにインポート，チューニングが行なわれた.
  - **TensorFlowはbazel**による構築が必要で，「京」での導入は困難を極めたが，**Chainer**は依存するライブラリが少なく，Pythonのsetup.pyを使って比較的容易に導入できた.
- **「京」/FX100でのMNIST問題の高速化**
  - 主にChainerを用いて性能チューニングを行い，CPUでのAI計算での実行や高速化の問題点を洗い出した.
  - 専用の高速化ライブラリ（DNNL）を持たない「京」のCPUでは，**numpyの高速化が鍵**であり，**SSL2数値ライブラリの結合は必要**であった. でないとOpenBLASで動作する(青色).
  - **optimizer処理の1/sqrt計算**にて，**SWPL**がきかず**浮動小数点アンダーフロー**の対応が必要であった. これらのnumpy処理をまとめて**Fortranにおきかえて高速化**した(赤色).
  - PythonによるNN処理ルーチン特有の**スレッド非並列をMPIによるプロセス分割**を行い高速化できた. しかし分散並列のこの手法はメモリ使用量が増えるため万能ではない(緑色).
  - **「京」で36.4倍，ピーク比で35.9%**にまで高速化され，富岳でもAI計算を十分できる見込みが立った.



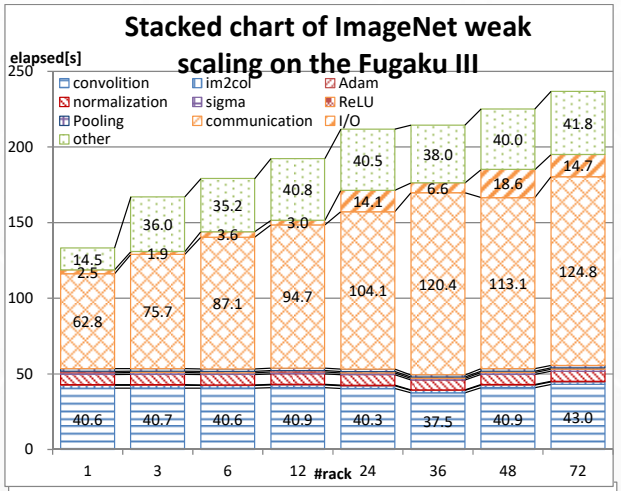
富岳に向けてのChainerの高速化

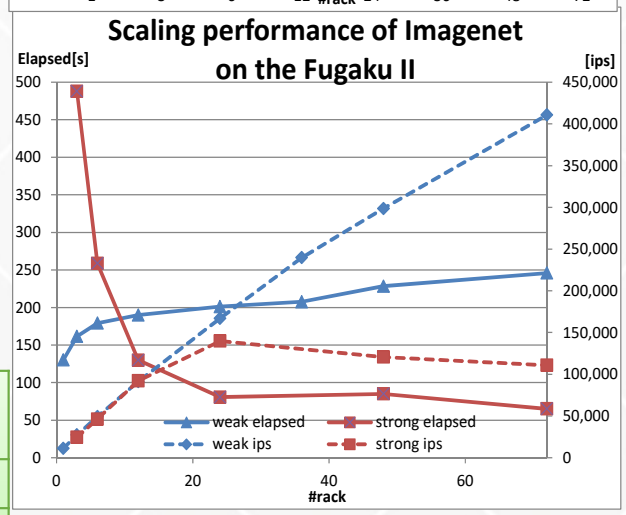
- **富岳でのResnet-50性能**
  - **関連する様々なマシン環境**にてDeep Learningのフレームワーク性能を評価し，性能阻害要因を調べ，高速化の可能性や指針を調査した.
  - FX100では，numpy GEMM処理でOpenBLASさえもインポートされず，最終的に1000倍もの高速化率を達成，**富岳ではResmet-50性能で66ipsを達成し，ピーク性能比でVoltaに匹敵する性能を実現した.**
  - 現在，**富士通研が開発したDNN高速化ライブラリ(MKL DNN/oneDNN)**を組み込んだ**PyTorch/TensorFlowにて，110ips**を達しており，富岳並びにFX1000で使えるようになっている.

Performance of Chainer			ASIS	TUNED		
system name	unit	model	elapsed	elapsed	peak ratio	speedup rate
K computer	CPU	MNIST	10,300.0 [s]	283.4 [s]	35.9%	36.4 x
FX100	CPU	MNIST	49,200.0 [s]	47.8 [s]	13.5%	1030.0 x
ThunderX2	CORE	MNIST	5,340.0 [s]	1,100.0 [s]	35.7%	4.8 x
			performance	performance	peak ratio	speedup rate
ThunderX2	CORE	Resnet-50		0.75 [ips]	54.4%	
ThunderX2	CPU	Resnet-50		11.95 [ips]	31.0%	
Fugaku	CPU	Resnet-50	16.30 [ips]	55.20 [ips] *	22.6%	3.4 x
Appolo70+Volta	GPU x2	Resnet-50		471.90 [ips]	39.2%	
	* estimation: 44.7-63.6 [ips]					

富岳に向けてのChainerの高速化

- **おまけ**
  - 「Deep Learningの並列性能の調査
    - Chainerを用い，**「富岳」での学習の並列性能**を調査した．
    - Weak Scalingな問題に対して，**演算時間(右上図：青赤)はスケール**することが分かった．
    - しかし，高並列で**通信やI/O(右上図：橙)がネック**になる．特に多ファイルから構成される**Python libraryのimportがMDSアクセス集中**を生み出し，問題となることが分かった．
    - LLIOが使えない「富岳」の開発段階にて，**BIOを利用したステージング**を提案し，このアクセス集中を回避できることが分かり，72rack規模までは拡張可能であることを初めて示した．
    - この手法は，現在，「富岳」の共用前評価環境での標準的な利用方法として提供された．
  - 大規模実問題ベンチマークMLPerf HPC性能
    - MLPerf HPCの評価・計測を実施した．主要演算の高速化には富士通研が開発した高速化ライブラリoneDNNを用いている．
    - I/O関係では，BIO上の/tmpやLLIOに変わり，BoB単位で共有できるSSD領域の利用を提案・導入し，ステージング時間を1/10程度まで削減した．
    - 通信関係では，BoB内データ共有のモデル並列に対し，BoB内隣接通信と全体の集団通信を同時に最適化するランク配置を提案・作製し測定に用いた．
    - **3次元画像解析問題のCosmoFlowにて，ABCIには及ばなかったものの2位の性能を達成し，CPUとしての単体性能並びに並列性能の両面で他を凌駕がし，日本がHPCでのAIを牽引していることを示した．「富岳」での16,384並列の並列性能を示したことは内外にその存在価値をアピールできたといえる．**





MLPerf HPC			frequency (GHz)	performance	top performance	difference (magnification)
benchmark	division	#nodes				
CosmoFlow	closed	512	2.2	228.77 min	34.42 min	0.15
CosmoFlow	closed	8192	2.2	101.49 min		0.34
CosmoFlow	open	16384	2.2	30.07 min	13.21 min	0.44

まとめ

- **numpy sqrtの問題から**
  - 本来sqrt計算は，富岳は**京ほどではないにせよ得意**なはずである.
  - 「京」ほどではないの意味は，**演算レイテンシの増加**により，レジスタ(OoO資源も)不足によるSWPLがうまくいかず，演算を隠せなくなっている**HWの仕様**によるものであり反省すべきである.
  - コンパイラの問題としては，難しいのは承知していますが，もっと**指示子の権限を強くする方法**が欲しい．norecurenceを入れてもポインタでループを回すとSIMD化やSWPL，スレッド並列はきかない.
  - しかし一番の問題は，**ユーザがOSSをブラックボックスとして使用**していることであり，**ユーザの意識向上**や**OSSの高速化をどうするか**が大きな課題になる.
- **convolution性能について**
  - CPUでも効率的には**GPUに劣らないこと**が分った.
  - cuDNNなどの高速化ライブラリでは**どのアルゴリズムでも最終的に小さなGEMMの処理に帰着**していた.
  - 但し，SIMD長が大きくなってしまったシステムでは**一般の方法(GEMM+前処理)では効率は出ない**．**Pythonではスレッド並列もちゃんと書かないと難しい**.
  - CPUではGPU単体と比べてピーク性能が低い分，並列化でカバーする必要があるが，(**Python module**と学習データの)I/O(と通信)がネックになりスケールしにくい.
  - また精度の問題から全体のbacth sizeは大きくすることができず，**モデル並列が必要**になる.

その他の実施事項

- AI計算の高速化
  - 富岳に向けてのChainerの高速化
  - NumPy sqrtの問題
  - convolution性能
- OpenMP TASK構文による通信・演算オーバーラップの検討
- GPUに対する性能評価
- C++最適化に関する性能評価
- 低B/Fアーキテクチャ向けの三重対角行列解法
- 混合精度演算と消費電力を考慮した自動チューニング

発表内容

- A64FXシステムアプリ性能検討WGのご紹介
- 活動事例
- おわりに

おわりに

- 「A64FXシステムアプリ性能検討WG」では、ユーザのプログラムを持ち込み問題を提起し、富士通コンパイラGに課題解析を依頼
- 成功事例
  - 適するコンパイラオプションの提案
  - SIMD化促進のための技法提案
    - OCL指示子、スカラ化などの書き方、等
  - ループ分割支援
    - 自動ループ分割機能、ループ分割の指定
- 今後の課題
  - さらなるループ分割の自動化
  - SIMD化の対象拡大
    - 最外ループのSIMD化をする機能
  - C++の最適化機能向上
    - GNU等の一般的なコンパイラが実現しているインライン関数や条件分岐文の最適化が機能せず、手動による関数のインライン化やマスク処理が必要、など

など