2.2 ハイブリッド非構造格子 CFD ソルバのメモリ階層構造への対応および有限体積法における再帰参照と SIMD 化

宇宙航空研究開発機構
坂下　雅秀

ポストペタアプリ WG の会合で報告した内容の概要について、以下に示す。

第四回会合（2014 年 12 月 10 日）では、「ハイブリッド非構造格子 CFD ソルバのメモリ階層構造への対応」と題して、CFD ソルバにおける固有の問題点と、キャッシュメモリへの対応方法についてまとめた。キャッシュの有効利用とは、キャッシュに乗せたデータでいかに多くの計算を行うかに尽きるが、CFD ソルバの採用する、いわゆるステンシル系の計算では、そもそもデータの再利用回数が少ないため、なかなか効率の良い方法が存在しないのが現状であり、前途多難である。

第七回会合（2015 年 10 月 8 日）では、「有限体積法における再帰参照と SIMD 化」と題して、多くの CFD ソルバが採用する有限体積法において SIMD 化に際して問題となる再帰参照と、その対処方法についてまとめた。また、実機（FX100）が利用可能となったため、非構造格子 CFD ソルバにおけて、実際に FX100 を使用した SIMD 化チューニングの一例について報告した。この例では、当該ソルバがサポートするいくつかの機能を諦めることによって SIMD 化の促進を図り、SIMD 化前に比べて計算時間で 1.5 倍、理論ピーク性能比では 6.3%という期待以上の性能が得られた。

一般に、実用に供するアプリは多数のユーザの多様な要求に答えるため、多くの機能を実装する必要に迫られる運命にある。このため、SIMD 化したい主要な DO ループが SIMD 化の障害となりうる複雑な構造を持たざるを得なくなることが多い。そこで、特定ユーザを念頭に、思い切って機能を絞り込み、主要な DO ループを簡素化することによって SIMD 化を促進し性能向上を図ることは、有力な選択肢のひとつである。

一方で、単純な DO ループしか SIMD 化できず性能の出ない SIMD アーキテクチャのシステムでは実用的とはいえない。そこで、いっさいの機能を諦めることなく、FX100 でどこまで SIMD 化し性能を出せるかについても重要なテーマであるといえる。

ハイブリッド非構造格子CFDソルバ の メモリ階層構造への対応

宇宙航空研究開発機構
坂下雅秀

非構造格子CFDソルバにおける問題点

- 主要ループはすべてランダムアクセス
  FACOM230-75 APUの時代から性能が出ない
  ⇨ とりあえず、キャッシュ（オンチップメモリ）にのせてreordering
  ⇨ メモリ階層構造への対応
  ⇨ やらなければいけないことはPhiでもGPUでもSX-ACEでもあまり変わらない
- 時間積分に陰解法が使われることが多い
  並列化が困難
  陽解法にしてしまうという方法はある
  ⇨ Krylov部分空間法を使うことにして前処理は？

キャッシュに「のっていない」状態

- データ xが 計算に必要となる
- メモリ（オフチップメモリ）→キャッシュ→レジスタ
- データ yが 計算に必要となる
- メモリ→キャッシュ→レジスタ

:

- キャッシュがいっぱいになりデータxが追い出される

:

- 再びデータ xが 計算に必要となる
- メモリ→キャッシュ→レジスタ

メモリ―キャッシュ間の転送回数が多い状態

キャッシュに「のっている」状態

- データ xが 計算に必要となる
- メモリ→キャッシュ→レジスタ
- データ yが 計算に必要となる
- メモリ→キャッシュ→レジスタ

:

- 再びデータ xが 計算に必要となる
- xがキャッシュに残っている

メモリ―キャッシュ間の転送回数が少ない状態

- 要するにキャッシュに「のっていない」「のっている」とは転送回数が「多い」「少ない」ということ
- 当該データが1回しか使われないなら意味がない

回数で考えるメリット

- 直観的でわかりやすい

PAで出してくれたらいいな

回数の計算方法

- スループットを使うと，データ転送がオーバーラップして隠れている部分は回数にカウントされない
- キャッシュミス回数を使うと実際に転送されている回数が計算される

隠れているのならカウントされなくていい？

回数の計算

- L1⇔L2キャッシュ間データ転送量[byte]（PA）

  L1$⇔L2$間スループット[GB/s]×浮動小数点L2$ヒット[s]
- L2キャッシュ⇔メモリ間データ転送量[byte]（PA）

  メモリスループット[GB/s]×浮動小数点L2$ミス[s]
- 定義・参照データサイズ

  定義・参照データ（配列）のサイズ[byte]

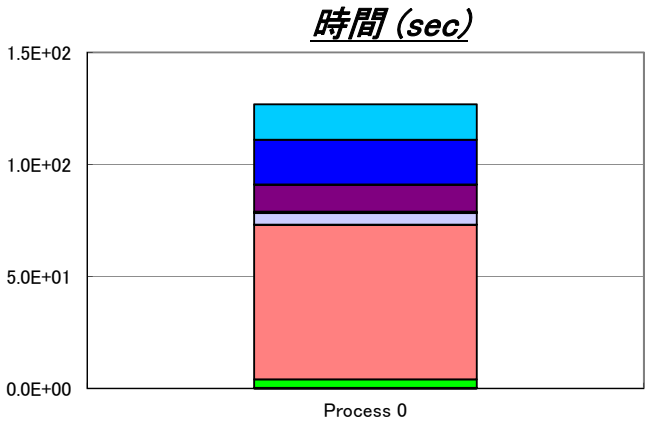
  直観的な指標なのでどんぶり勘定でOK

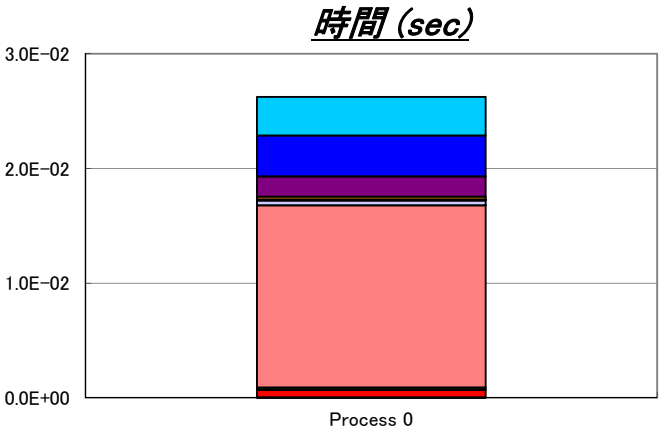
回数＝データ転送量÷定義・参照データサイズ

例題：行列・行列積の計算C=AB

```
    implicit NONE
    integer, parameter   :: N= 4096        !行列のサイズ
    real*8,allocatable :: A(:,:),  B(:,:),  C(:,:)
    integer :: i, j, k
    allocate(A(N,N),B(N,N),C(N,N))
(snip)
    do j=1,N
    do k=1,N
    do i=1,N
      C(i,j) = C(i,j) + A(i,k)*B(k,j)
    end do
    end do
    end do
(snip)
```


PA測定結果





行列・行列積の計算N=4096

L1$⇔L2$間転送回数≒65
$=4.08\times10^9\times69.0\div(8\times32\times4096^2)$
L2$⇔メモリ間転送回数≒0.2
$=4.05\times10^9\times0.20\div(8\times32\times4096^2)$

平均の計算 C(i)=0.5*(A(i)+B(i)), i=1, 10^7

L1$⇔L2$間転送回数≒0.6
$=8.61\times10^9\times0.016\div(8\times3\times10^7)$
L2$⇔メモリ間転送回数≒0.2
$=11.57\times10^9\times6.62\times10^{-4}\div(8\times3\times10^7)$

- 回数が多く性能に不満がある

  タイリング（＋アクセスの連続化）

  etc.

- 回数が少なく性能に不満がある

  データの再利用が図れるようなループの融合

  etc.

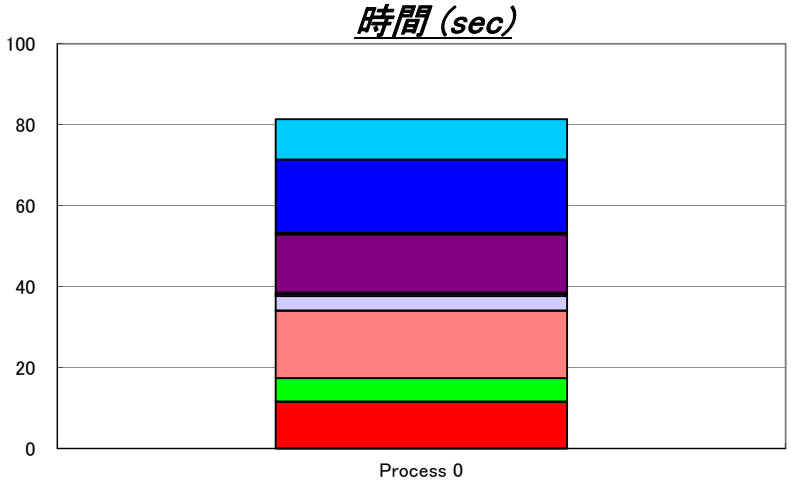
あとは

具体的な事例集があればなんとかなる？

行列・行列積のタイリング

```
    implicit NONE
    integer, parameter :: NN   =  4096     ! 行列のサイズ
    integer, parameter :: NT   =    256     ! タイル（小行列）の数
    integer, parameter :: NB   = NN/NT   ! タイル（小行列）のサイズ
    real*8,allocatable  :: A(:,:,:,:), B(:,:,:,:), C(:,:,:,:)
    integer :: i , j , k , l, m, n
    allocate( A(NB,NB,NT,NT), B(NB,NB,NT,NT), C(NB,NB,NT,NT) )
(snip)
    do n=1,NT;  do m=1,NT;  do l=1,NT
      do j=1,NB
      do k=1,NB
      do i=1,NB
        C(i,j,l,n) = C(i,j,l,n) + A(i,k,l,m)*B(k,j,m,n)
      end do
      end do
      end do
    end do; end do; end do
```

PA測定結果（N=4096、NB=16）



行列・行列積の計算N=4096,NB=16

L1$⇔L2$間転送回数≒4.3(←65)

$=1.11\times10^9\times16.7\div(8\times32\times4096^2)$

L2$⇔メモリ間転送回数≒2.4(←0.2)

$=0.9\times10^9\times11.6\div(8\times32\times4096^2)$

ピーク性能比11%
↓
ピーク性能比17%

ちなみにIvy Bride E5 2643V2ではNB=64のときピーク性能比28%

行列・行列積の性質

- ループ1回当たり2演算を大量（NB^3回）に繰り返す
- 計算に必要なデータは3 NB^2語

⇩

配列要素a_{ij}（or b_{ij}）をNB回繰り返し利用
簡単な計算を大量に行うのでレジスタに余裕がある

⇩

データの利用回数が多いのでキャッシュにのせれば効果大
上手くいけばレジスタ上のデータも再利用してくれる？

タイリングの効果が出易い

（行列・行列積の計算は
演算量が計算に必要なデータ量に比べて多い計算）

オンチップメモリとタイリング

- 本来の？オンチップメモリの使い方

  計算とメモリアクセスのオーバーラップ

  （そもそも、オーバーラップするのに最低限必要なメモリ帯域が「（アプリケーションが）要求（する）メモリ帯域（B／F）」だったような？）

  ⇨ オンチップメモリにのせるには、まずは小分け（タイリング）する必要がある

  ⇩ ユーザの手で
  オーバーラップさせるのはハードルが高いので

- キャッシュメモリの場合、ハードおよびコンパイラが頼り

  ⇨ 性能が出ればそれでおしまい
  出なければ？

  ⇩

- タイル内のデータの利用回数を増やすことを考える

- プリフェッチはload/storeネックのアプリでは効果が薄い？

- 一番効果が出てほしいのは、そういうアプリだったりする

ハイブリッド非構造格子の性質

ハイブリッド非構造格子とは

ピラミッド（4面体）、プリズム（三角柱）、六面体等
種々の格子を組み合わせて形成された空間格子

性質

・複雑形状まわりの格子を精度よく生成し易い
・領域分割が難しい（METISなどPDSを利用）
⇨ タイリングには再帰的領域分割が必須

非構造格子の性質

・ランダムアクセス

そんなハイブリッド非構造格子CFDソルバで
性能を出すには？

非構造格子CFDにおける移流項の計算

- データの利用回数は10程度で少ない（2次精度）
  ⇨ タイリングによる効果が出難い

⇩

再帰的領域分割だけでは、あまり意味がない

「袖」まで計算すれば同じデータを利用するループが統合できて利用回数を増やせる（マルチコアクラスタWG）

- 演算量はループ1回あたり1,000程度で多い
  ⇨ 計算式が複雑で途中結果が大量に発生

⇩

物理的に意味のあるループに分割し整理

本当に早くなる？

試にやってみるにはハードルが高い

とりあえず移流項の計算部分を抜きだして
格子点数を変化させてサブリージョンの生成を代用
境界条件の計算は無視

性能をリミットする要因が明確になっていないので
果たして実アプリの傾向を継承しているのか不明

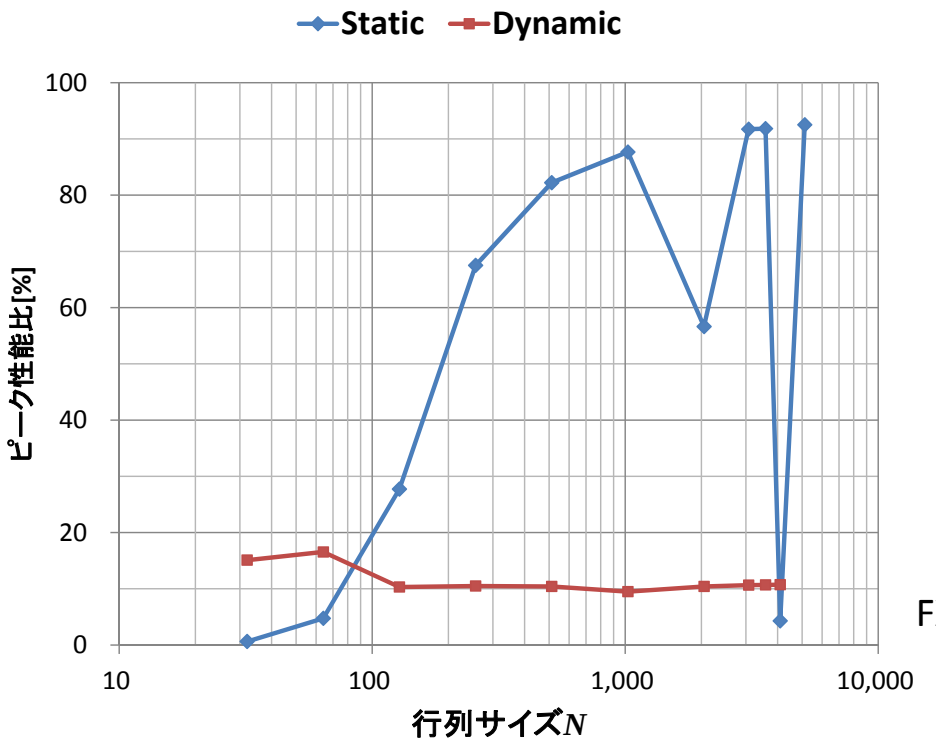
ループの統合は有効か？

一連の計算の物理的に意味のある
要素計算への分割は可能か？

くらいは見られる？

おまけ
（新システム評価結果の一部）

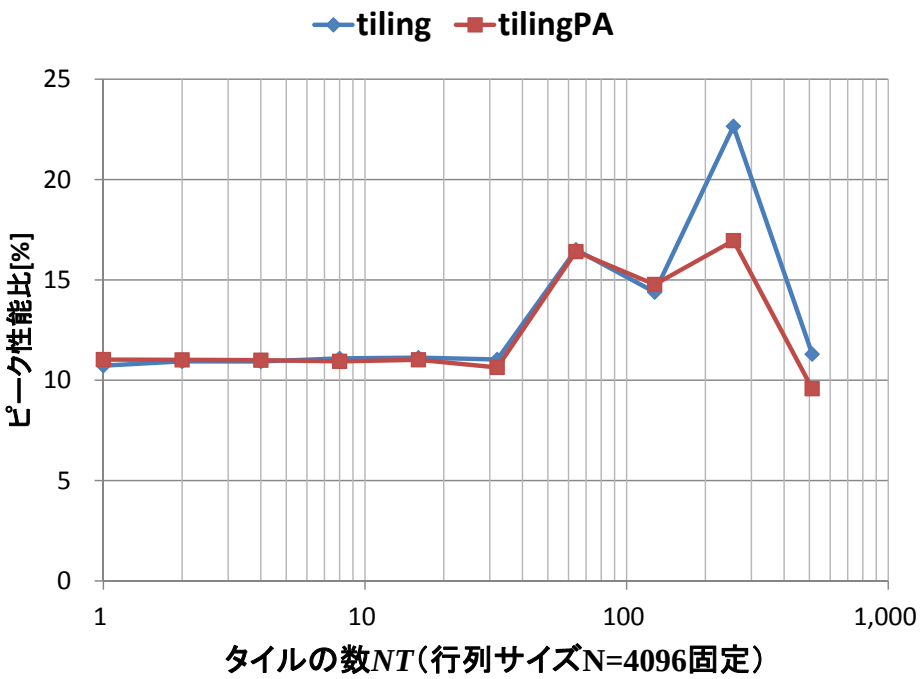
行列・行列積における
配列の静的確保（Static）と動的確保（Dynamic）



FX1で測定

タイリングを行った場合の行列・行列積の性能

演算量と実行時間から計算した結果（tiling）と PAによる測定結果（tiling PA）



FX1で測定

有限体積法における再帰参照とSIMD化

宇宙航空研究開発機構
坂下　雅秀

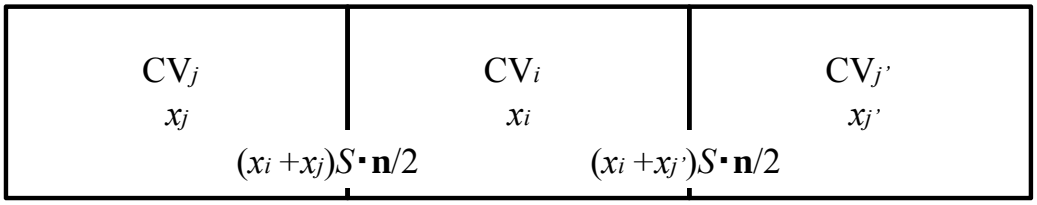
SS研ポストペタアプリ性能WG第7回会合資料2015年10月8日

有限体積法における典型的な定義参照関係：勾配の計算

例：　Green-Gauss公式（Gaussの発散定理）

$$\int_V \nabla x dV = \oint_S x \cdot d\mathbf{S} \quad \Rightarrow \quad \nabla x = \frac{1}{V}\oint_S x \cdot d\mathbf{S}$$

差分表記　$$\nabla x_i = \frac{1}{V_i}\sum_j \frac{1}{2}\left(x_i + x_j\right) S \cdot \mathbf{n}$$



$(x_i+x_j)S \cdot \mathbf{n}/2$はControl Volume（CV）表面で求まり
CV$_i$とCV$_j$両方の計算で参照される

コーディング・パターン

パターンI

```
do iface=1,num_face

   icl     =face2CV(1,iface)
   icr     =face2CV(2,iface)
   wdx     =0.5*(x(icl)+x(icr))*Sn(iface)
   dx(icl)=dx(icl)+wdx
   dx(icr)=dx(icr)+wdx
end do
```

全ての面 *iface* で$(x_i+x_j)S\mathbf{n}/2$を計算
<u>同時に</u>左右のCVに計算結果を配る

- もっとも素直なコーディング
- スレッド並列化不可（再起参照）
- キャッシュミス率低

パターンII

```
do icol=1,num_color
do jcol=col_list(1,icol),col_list(2,icol)
   iface  =face_list(jcol,icol)
   icl    =face2CV(1,iface)
   icr    =face2CV(2,iface)
   wdx    =0.5*(x(icl)+x(icr))*Sn(iface)
   dx(icl)=dx(icl)+wdx
   dx(icr)=dx(icr)+wdx
end do
end do
```

“パターンI”の再帰参照をcolorlingにより回避

- 非構造格子向き
- スレッド並列化可能
- 収束性低下？
- キャッシュミス率高

コーディング・パターン

パターンIII

```
 do iface=1,num_face
    icl        =face2CV(1,iface)
    icr        =face2CV(2,iface)
    wdx(iface)=0.5*(x(icl)+x(icr))*Sn(iface)
 end do
!$omp parallel do schedule(dynamic,1000)
 do icv=1,num_CV
 do ifc=1,num_face_into_CV(icv)
    iface=CV2face(ifc,icv)
    dx(icv)=dx(icv)+wdx(iface)
 end do
 end do
```

全ての面 *iface* で$(x_i+x_j)\mathrm{Sn}/2$ を計算
<u>作業用配列に保存したのち</u>各CVで和を計算

- 構造格子向き（二番目のloopが展開可能）
- 一番目のloopは高速・二番目のloopは低速
- openMP(OMP)の動的割り当て適用により性能改善
- スレッド並列化可能

パターンIV

```
do icv=1,num_CV
do ifc=1,num_face_into_CV(icv)
   iface  =CV2face(1,ifc,icv)
   icr    =CV2face(2,ifc,icv)
   dx(icv)=dx(icv)
          +0.5*(x(icv)+x(icr))*Sn(iface)
end do
end do
```

全てのCVで一度に$\Sigma(x_i+x_j)S\mathbf{n}/2$を計算

- 計算量2倍
- スレッド並列化可能
- キャッシュミス率低

非構造格子CFDソルバの処理の流れ

Navier-Stokes方程式 → 連立方程式：$\mathbf{A}x=\mathbf{b}$

・右辺（ベクトル）$\mathbf{b}$の計算

勾配∇x_i計算
勾配制限関数Ψ_iの計算 ⇨ CV$_i$表面における物理量x_f

$$x_f = x_i + \Psi_i \nabla x_i (\mathbf{r}_i - \mathbf{r}_f)$$

移流項（advection flux）の計算

粘性項（viscous flux）の計算

・時間積分

係数行列$\mathbf{A}$は配列に保存されずここで計算される（matrix free form）

LU-SGS（部分因数分解による近似LU分解）陰解法

移流項（advection）の計算

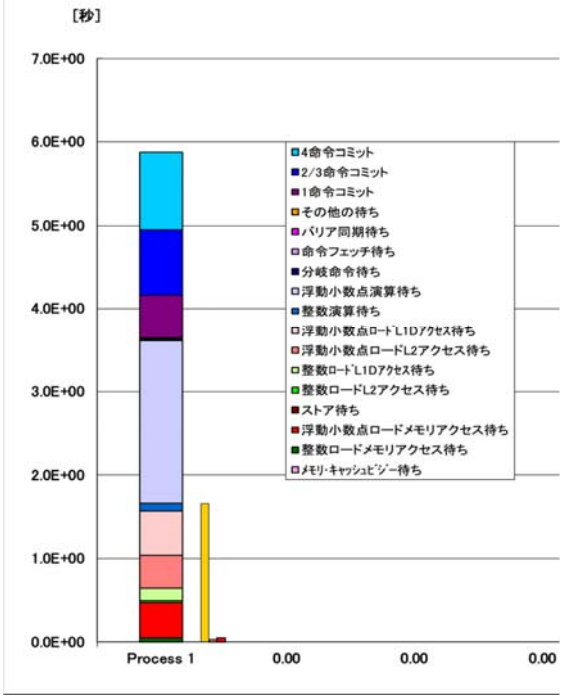
No.	修正箇所	時間	ピーク比	SIMD	SWP
0	(ASIS)	5.9sec	3.9%		
1	内部サブルーチンを手動でインライン展開	5.9sec	3.9%		
2	If文の削除（置き換え） 最内側ループの インライン展開	3.0sec	8.1%		○
3	ループ分割（パターンIII） SWPのOCL文追加	2.2sec	前半：11.4% 後半： 6.5%	○	○

その他、-prefetch_indirectの追加、-Nrtrapの削除など、最適化オプションの変更を行ったが、変化は0.1%以下。

移流項(advection)の計算

No.	修正箇所	時間	ピーク比	SIMD	SWP
0	(ASIS)	5.9sec	3.9%		

修正前(ASIS)



7

移流項(advection)の計算

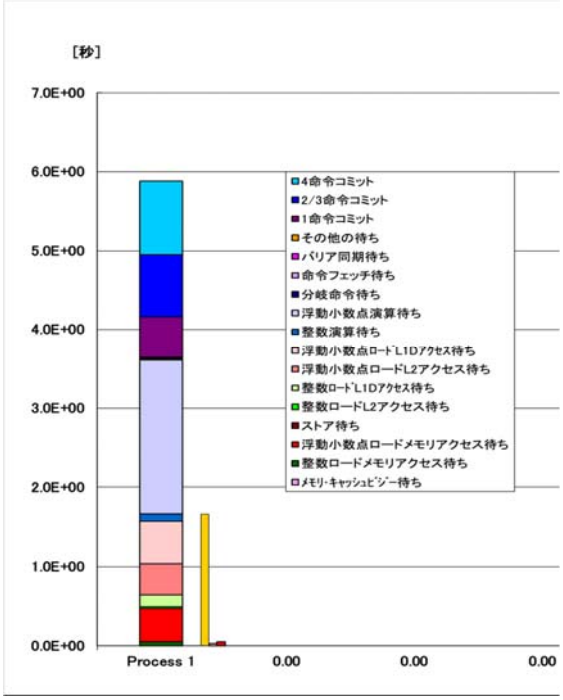
No.	修正箇所	時間	ピーク比	SIMD	SWP
1	内部サブルーチンを手動でインライン展開	5.9sec	3.9%		

```
faceloop1: do i_face = 1, n_face
    call pre_advection_scheme(再構築)
    call hllew(リーマン解法)
    call post_advection_scheme(流束和分)
end do faceloop1
```

赤字の部分を手動でインライン展開

もともと、コンパイルオプションで展開するようにしているが、その方法だと、最適化がかからない。

しかし、展開するだけでは、SWPやSIMDが有効とならないため、性能は変わらない



8

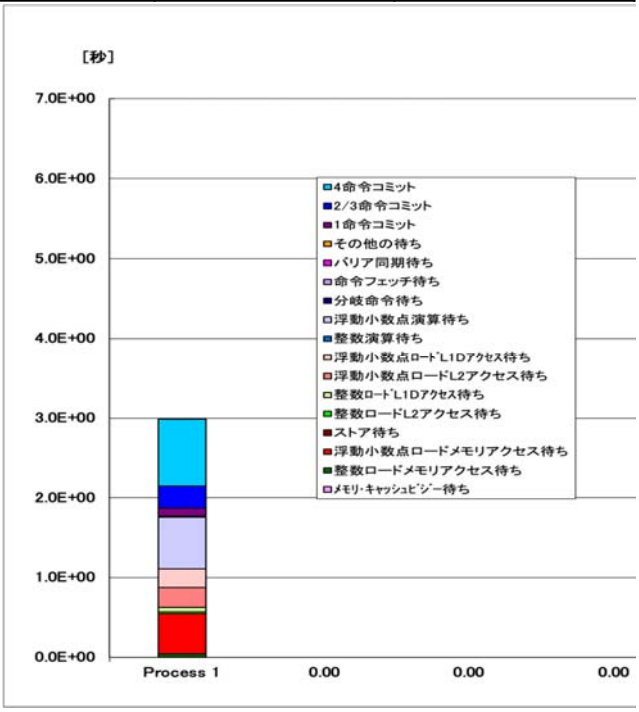
移流項の計算（pre_advection_scheme）

No.	修正箇所	時間	ピーク比	SIMD	SWP
2	If文の削除 最内側ループ展開	3.0sec	8.1%		○

```
flag_neg = max(0,sign(1,pria_update(1)))*max(0,sign(1,pria_update(5)))
!ocl unroll(5)
do i = 1,5
      pria_update_tmp(i,i_face) = flag_neg*pria_update(i)    &
&                                 + (1-flag_neg)*pria(i)
end do

flag_neg = max(0,sign(1,prib_update(1)))*max(0,sign(1,prib_update(5)))
!ocl unroll(5)
do i = 1,5
      prib_update_tmp(i,i_face) = flag_neg*prib_update(i)   &
&                                 + (1-flag_neg)*prib(i)
end do
```

If文をmax/min/sign関数で置き換える
面ループの中のdoループはocl文でunrolling
←最内側ループは!ocl unrollで展開する
←乱流モデルの変数は固定にする
（例えば、全部2方程式を解く）



移流項（advection）の計算

No.	修正箇所	時間	ピーク比	SIMD	SWP
3	ループ分割（パターンIII） SWPのOCL文追加	2.2sec	前半：11.4% 後半：　6.5%	○	○

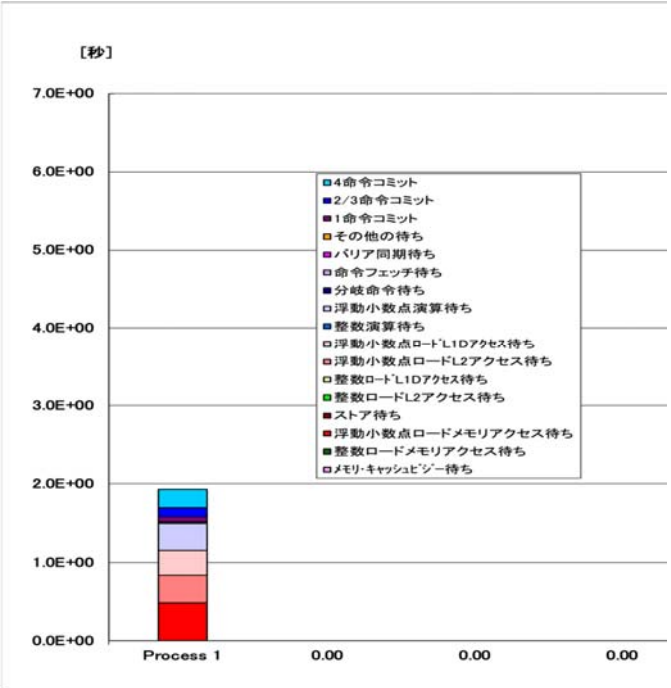
```
faceloop11: do i_face = 1, n_face
pre_advection_scheme（再構築）      ← 一時的に配列に保存
end do faceloop11

faceloop12: do i_face = 1, n_face
hllew（リーマン解法）               ← 一時的に配列に保存
end do faceloop12

cellloop1: do i_cell = 1, n_cell
post_advection_scheme（流束和分）
end do cellloop1
```

再帰参照を回避すため、
CVループに変更

ループを分割することで、SIMDとSWPが有効になる



制限関数（limiter）の計算

前半：最大・最小を求める

No.	修正箇所	時間	ピーク比	SIMD	SWP
0	(ASIS)	0.8sec	0%		
1	外側ループでSWP化	0.5sec	0%		○
2	面ループ→CVループ （パターンIV）	0.2sec	0%	○	○

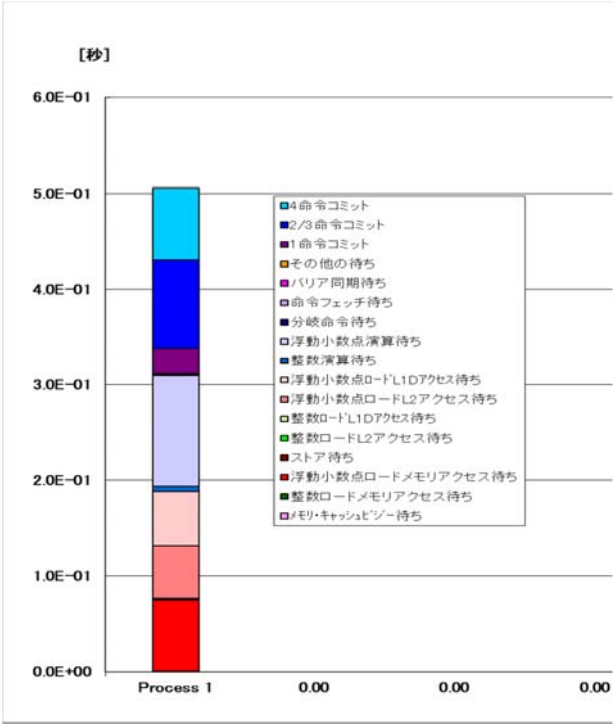
後半：制限関数の計算

No.	修正箇所	時間	ピーク比	SIMD	SWP
0	(ASIS)	3.4sec	7.0%		
1	ループ融合を無効化 ループ展開	2.6sec	9.3%		○
2	ループ分割（パターンIII）	1.9sec	13.7%	○	○

制限関数（limiter）の計算前半

No.	修正箇所	時間	ピーク比	SIMD	SWP
0	(ASIS)	0.8sec	0%		

修正前（ASIS）



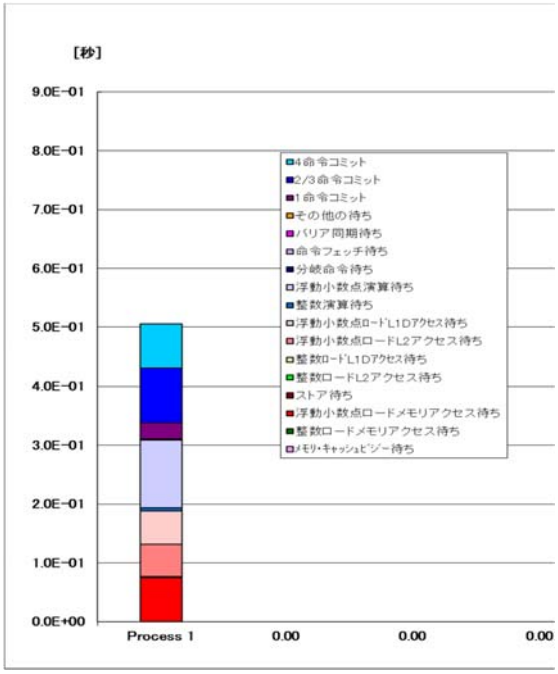
制限関数（limiter）の計算前半

No.	修正箇所	時間	ピーク比	SIMD	SWP
1	ループ融合を無効化 最内側ループの展開	0.5sec	0%		○

```
do i_face = 1, num_face

!ocl LOOP_NOFUSION
!ocl unroll(5)
      do i = 1, num_var          ! num_var=5, parameter属性, 引数渡し
        cell_states_max(i,ia_cell) = max( cell_states_max(i,ia_cell), prib(i) )
      end do
!ocl LOOP_NOFUSION
!ocl unroll(5)
      do i = 1, num_var          ! num_var=5, parameter属性, 引数渡し
        cell_states_min(i,ia_cell) = min( cell_states_min(i,ia_cell), prib(i) )
      end do

end do
```



!ocl loop_nofusionで、ループ融合を無効化
!ocl unrollによりループ展開
→面ループがSWP化される
→実は、unrollだけでいいかも

制限関数（limiter）の計算前半

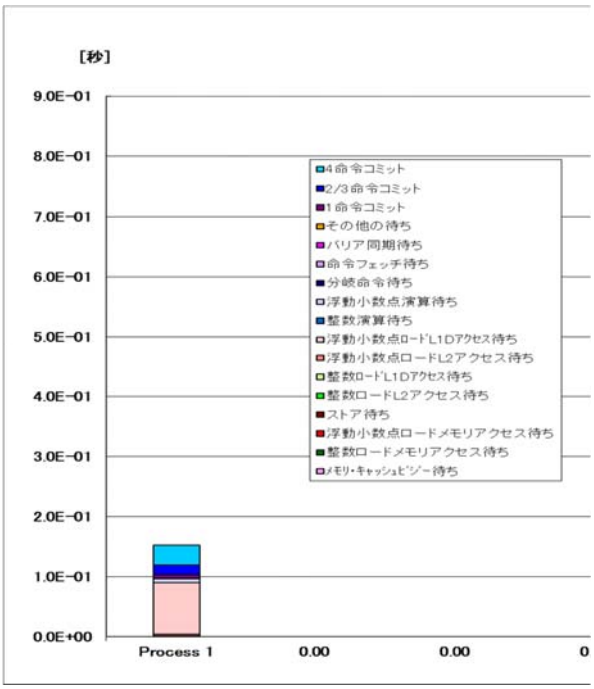
No.	修正箇所	時間	ピーク比	SIMD	SWP
2	面ループ→CVループ （パターンIV）	0.2sec	0%	○	○

```
do i_cell = 1, num_cell

      flag1 = max(0,sign(1,n_cell2face(i_cell)-1))
      flag2 = max(0,sign(1,n_cell2face(i_cell)-2))
      （省略）

      i0 = i_cell
      i1 = (1-flag1)*cell2cell(1,i_cell) + flag1*i_cell
      i2 = (1-flag2)*cell2cell(2,i_cell) + flag2*i_cell
      （省略）

!ocl unroll(5)
      do i = 1, 5
      cell_states_max(i,i_cell) = max ( cell_states(i,i0), &
                                  cell_states(i,i1), &
                                  cell_states(i,i2), &
                                  cell_states(i,i3), &
                                  cell_states(i,i4), &
                                  cell_states(i,i5), &
                                  cell_states(i,i6))
      end do
end do
```

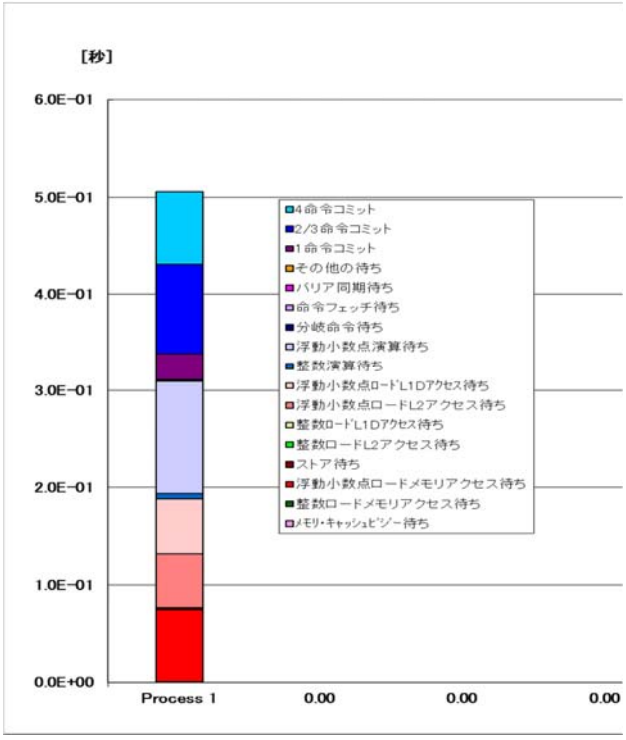


CVループに変更することで再帰参照を回避

制限関数（limiter）の計算後半

No.	修正箇所	時間	ピーク比	SIMD	SWP
0	(ASIS)	3.4sec	7%		

修正前（ASIS）



制限関数（limiter）の計算後半

No.	修正箇所	時間	ピーク比	SIMD	SWP
1	外側ループでSWP化	2.6sec	9.3%		○

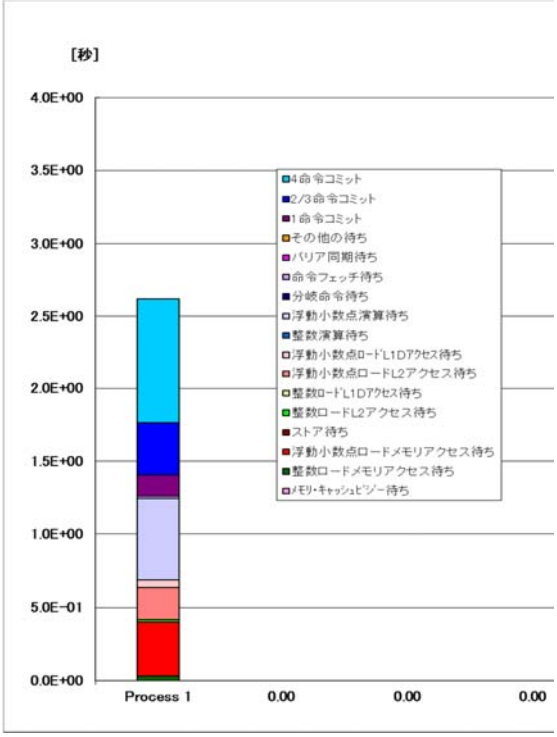
```
!ocl SWP
faceloop5: do i_face = 1, num_face

      （省略）

!ocl unroll(5)
!ocl NOSWP
!ocl NOSIMD
        do i = 1, num_var              ! Num_var=5, parameter属性
            （省略）
            slope_limiter(i,ib_cell) = min( slope_limiter(i,ib_cell), phi_tmpb )
        end do

end do faceloop5
```

!ocl unrollにより最内側ループを展開
内部ループのSWP化を無効化
外部ループでSWP化



制限関数（limiter）の計算後半

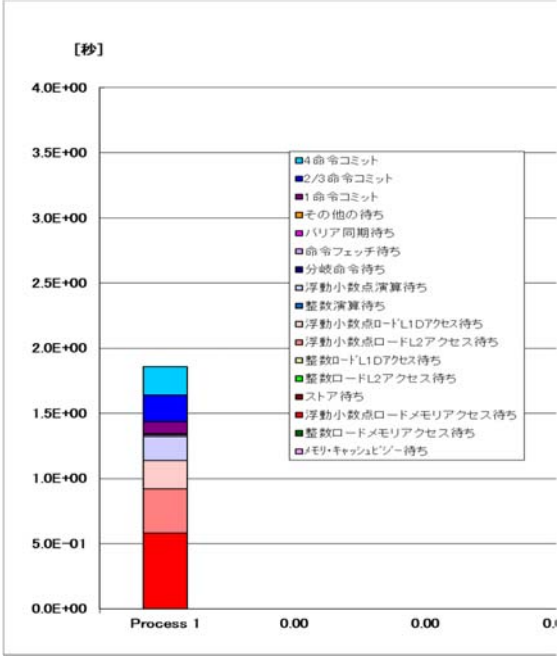
No.	修正箇所	時間	ピーク比	SIMD	SWP
2	ループ分割 （パターンIII）	1.9sec	13.7%	○	○

```
!ocl SWP
faceloop5: do i_face = 1, num_face
      （省略）
!ocl unroll(5)
       do i = 1, 5
              （省略）
              phi_tmp(2,i,i_face) =
       end do
end do faceloop5

!ocl loop_nofusion
!ocl SWP
 celllloop5: do i_cell = 1, num_cell
      （省略）
!ocl unroll(5)
       do i = 1, 5
         slope_limiter(i,i_cell) = &
&          min(  (1-fb1)*phi_tmp(1,i,i1) + fb1*phi_tmp(2,i,i1),  &
&              (1-fb2)*phi_tmp(1,i,i2) + fb2*phi_tmp(2,i,i2),  &
&              (1-fb3)*phi_tmp(1,i,i3) + fb3*phi_tmp(2,i,i3),  &
&              (1-fb4)*phi_tmp(1,i,i4) + fb4*phi_tmp(2,i,i4),  &
&              (1-fb5)*phi_tmp(1,i,i5) + fb5*phi_tmp(2,i,i5),  &
&              (1-fb6)*phi_tmp(1,i,i6) + fb6*phi_tmp(2,i,i6) )
       end do
end do celllloop5
```

面のループ

CVのループ



勾配（gradient）の計算

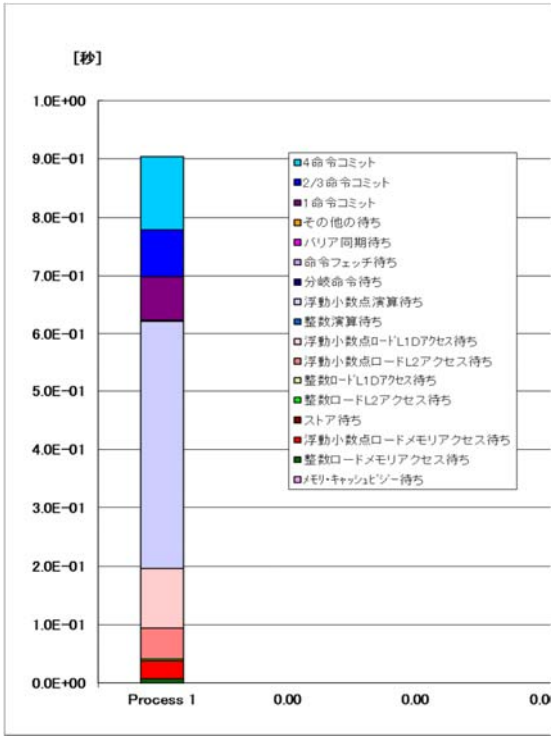
No.	修正箇所	時間	ピーク比	SIMD	SWP
0	(ASIS)	0.9sec	3.53%		
1	面ループを面ループとCVループに分割（パターンIII）	前半：0.4sec 後半：0.9sec	前半：5.8% 後半：4.2%	前半：○ 後半：○	前半：○ 後半：×
2	後半にprefetch	前半：0.4sec 後半：0.6sec	前半：5.8% 後半：5.5%	前半：○ 後半：○	前半：○ 後半：×

とりあえず、このルーチンだけ、高速化できなかった。

勾配（gradient）の計算

No.	修正箇所	時間	ピーク比	SIMD	SWP
0	(ASIS)	0.9sec	3.53%		

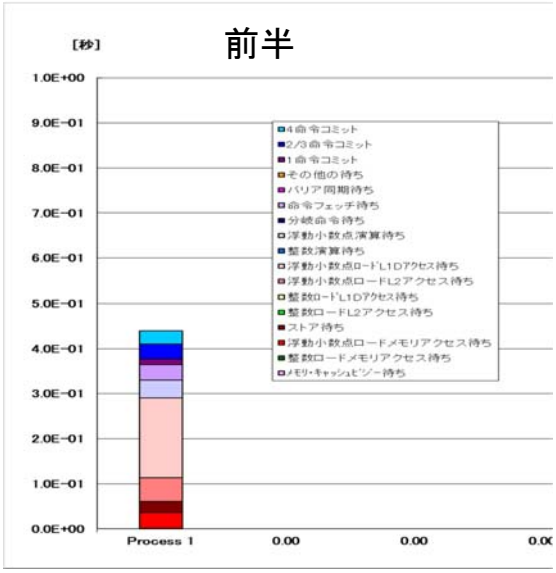
修正前（ASIS）



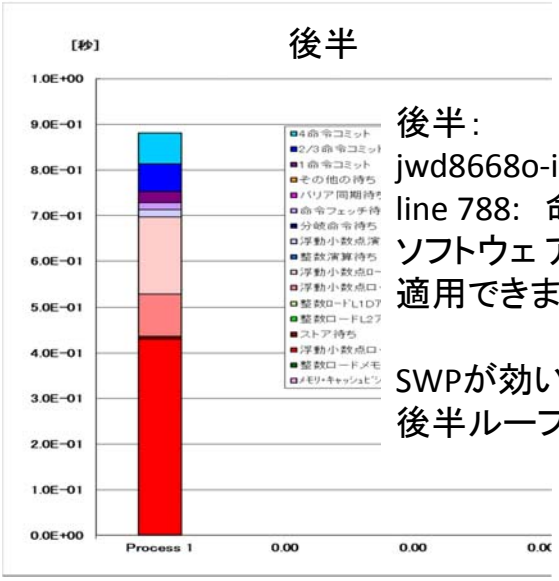
勾配（gradient）の計算

No.	修正箇所	時間	ピーク比	SIMD	SWP
1	ループ分割 （パターンIII）	前半：0.4sec 後半：0.9sec	前半：5.8% 後半：4.2%	前半：○ 後半：○	前半：○ 後半：×

前半



後半

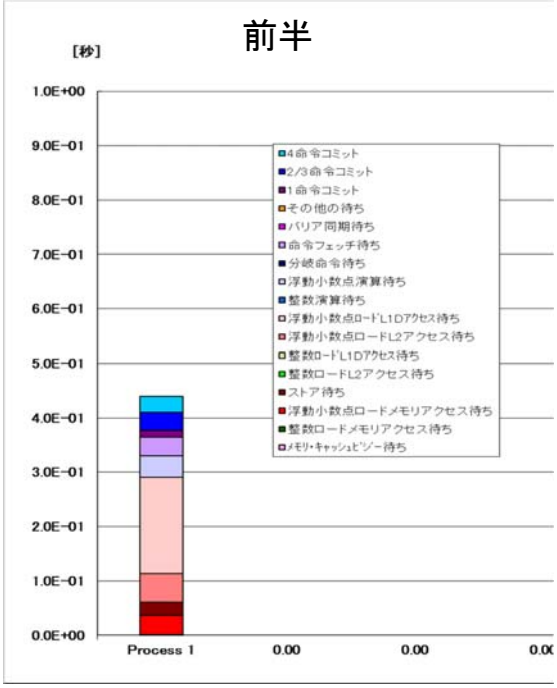


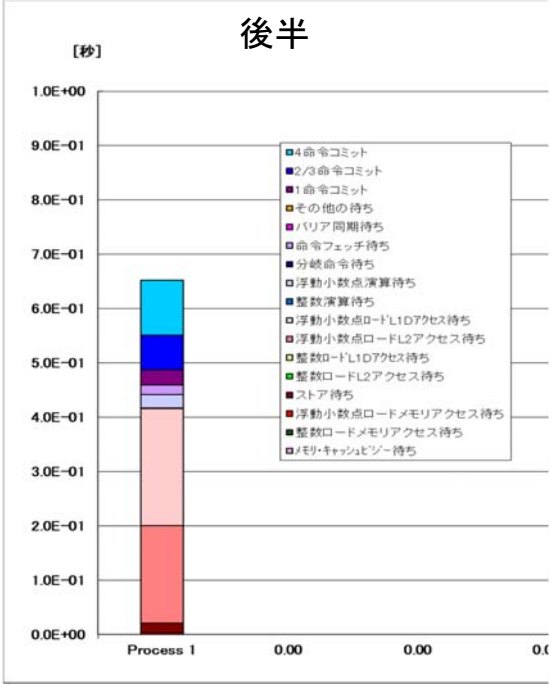
後半：
jwd8668o-i "gradient.f90",
line 788: 命令数が過剰なため，
ソフトウェアパイプライニングを
適用できません

SWPが効いていないため（？）
後半ループでメモリ待ちが目立つ

勾配(gradient)の計算

No.	修正箇所	時間	ピーク比	SIMD	SWP
2	後半にprefetch	前半:0.4sec 後半:0.6sec	前半:5.8% 後半:5.5%	前半:○ 後半:○	前半:○ 後半:×





Prefetch指定により
メモリ待ち改善

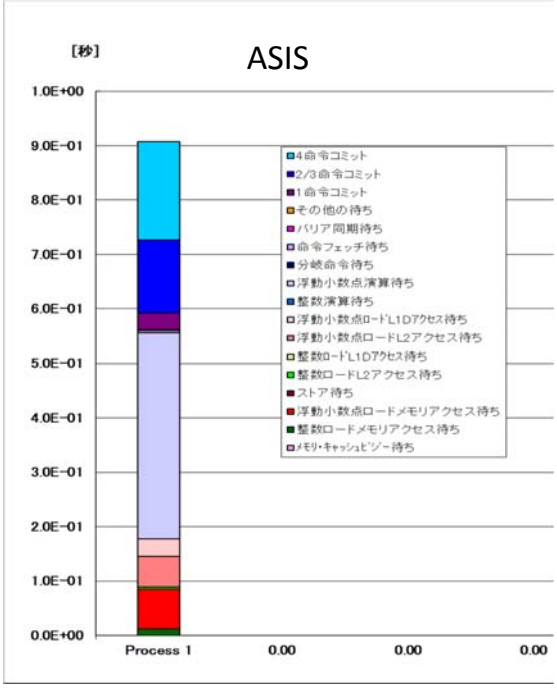
SWP化されれば、
もう少しはマシに
なる？

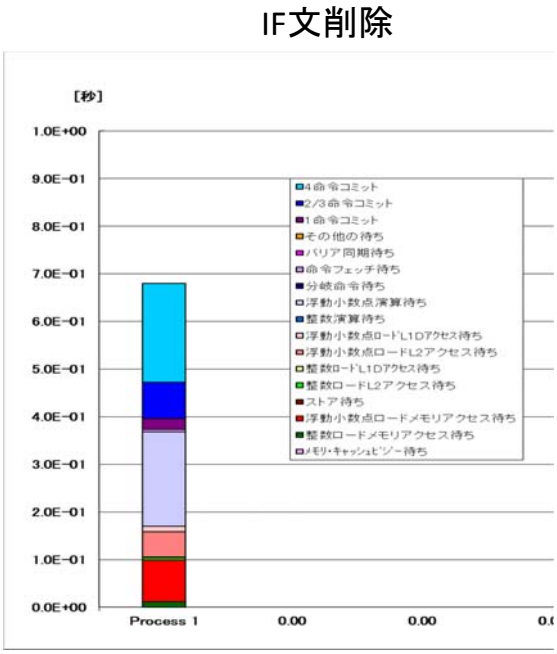
粘性項(viscous)の計算

No.	修正箇所	時間	ピーク比	SIMD	SWP
0	(ASIS)	0.9sec	7.0%		
1	If文削除	0.7sec	9.3%		○
2	ループ分割 (パターンIII)	前半:0.4sec 後半:0.1sec	前半:14.6% 後半: 4.6%	前半:○ 後半:○	前半:○ 後半:○

粘性項(viscous)の計算

No.	修正箇所	時間	ピーク比	SIMD	SWP
0	(ASIS)	0.9sec	7.0%		
1	If文削除	0.7sec	9.3%		○

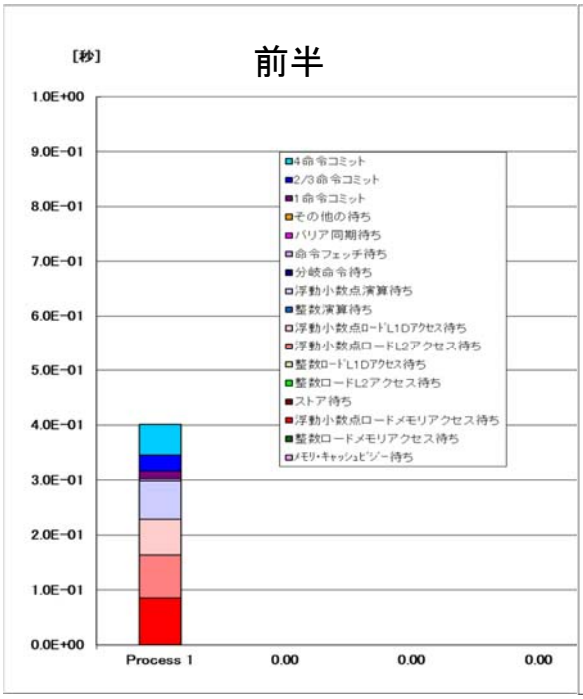




SWP化によって
演算待ち時間短縮

粘性項(viscous)の計算

No.	修正箇所	時間	ピーク比	SIMD	SWP
2	ループ分割 (パターンIII)	前半:0.4sec 後半:0.1sec	前半:14.6% 後半: 4.6%	前半:○ 後半:○	前半:○ 後半:○



SIMD化によって
さらに高速化