2.6 2次元プラズマ粒子(PIC)コードの測定評価および宇宙プラズマ５次元ブラソフコードVlasov5の測定評価

名古屋大学宇宙地球環境研究所
梅田　隆行

2.6.1 ２次元プラズマ粒子(PIC)コードの測定評価

2.6.1.1 はじめに

Particle-In-Cell(PIC)法は、1960 年代にプラズマ粒子と電磁場との相互作用のシミュレーション手法として考案されて以来、様々な分野において幅広く用いられている。オリジナルの PIC 法は、プラズマ物理及び電波科学分野において開発され、格子点(Cell)上に定義された電磁場をマックスウェル方程式により解き進め、その電磁場 Cell 中を荷電粒子が加減速を受けながら動きまわることから PIC と名付けられた。現実空間に存在する膨大な数の荷電粒子を有限の計算機資源で扱うことは不可能であるため、PIC 法では、ある程度まとまった数の荷電粒子の集団を 1 つの“超”粒子(super-particle)として扱っている。PIC 法では、ラグランジュ変数である粒子の位置及び速度と、オイラー変数である電磁場がデータ配列として混在しているため、スカラ型超並列計算機において高い性能を得るのは容易ではない。そのため、様々なスカラ型 CPU における性能特性を理解しておく必要がある。

2.6.1.2 プログラム概要

プラズマ粒子の運動は、以下の荷電粒子の運動(ニュートン・ローレンツ)方程式によって記述される。

$$\frac{\mathrm{d}\vec{r}_n}{\mathrm{d}t} = \vec{v}_n \tag{1}$$

$$\frac{\mathrm{d}\vec{v}_n}{\mathrm{d}t} = \frac{q_n}{m_n}\left(\vec{E} + \vec{v}_n \times \vec{B}\right) \tag{2}$$

また電磁場の時空間発展は以下のマックスウェル方程式によって記述される。

$$\nabla \times \vec{B} = \mu_0 \vec{J} + \frac{1}{c^2}\frac{\partial \vec{E}}{\partial t} \tag{3}$$

$$\nabla \times \vec{E} = -\frac{\partial \vec{B}}{\partial t} \tag{4}$$

さらに、荷電粒子が作る電流を計算するために、以下の電荷の連続の式を用いる。

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \vec{J} = 0 \tag{5}$$

運動方程式(1)及び(2)の時間積分が2次精度になるように、粒子の位置と速度は互いに Δt/2 ずれた時刻に定義される。また粒子の位置と電場を同じ時刻に、粒子の速度と磁場を同じ時刻にすることで、Maxwell 方程式(3)及び(4)による電場の更新と磁場の更新がそれぞれ2次精度になるようにしている。電磁場の各成分(Ex,Ey,Ez,Bx,By,Bz)は、空間差分が 2 次精度になるように、Yee 格子と呼ばれる Staggered 格子上に定義される。また、電流密度(Jx,Jy,Jz)は電場(Ex,Ey,Ez)と同じ空間格子上かつ粒子の速度(磁場)と同じ時刻に定義される。ラグランジュ

変数である荷電粒子の位置及び速度(x,y,z,vx,vy,vz)と格子状のオイラー変数である電磁場(Ex,Ey,Ez,Bx,By,Bz)が混在していることが、プラズマ PIC コードの特徴と言える。

プラズマ PIC コードのカーネルは3つに大きく分けることができる。1つは Maxwell 方程式(3)及び(4)による電磁場の更新で、これには FDTD 法が用いられる。電磁場更新カーネルの負荷が全体の計算負荷で占める割合は 1%未満であり、FDTD 法の性能特性及びチューニングについては、SS 研マルチコアクラスタ性能 WG 成果報告書を参照して頂き、ここでは割愛する。

2 つ目のカーネルは、式(3)による荷電粒子の速度計算であり、その概要をプログラム1に示す。粒子がどの格子の近くにあるかを計算し(2-3 行目)、隣接格子に対する重みを計算し(4-13 行目)、荷電粒子の位置における電磁場の値を隣接格子から重みに基づいて計算し(14-19 行目)、最後に荷電粒子の位置における電磁場の値を用いて粒子を加速する(20-22 行目)。ここで特徴的なのは、14-19 行目が粒子の位置に対するリストアクセスになっている点である。粒子番号 n は位置には全く依存しないため、配列のインデックス i,j はランダムとなり、結果として 14-19 行目の配列へのアクセスはランダムロードとなる。

3 つ目のカーネルは、式(5)による電流密度の計算であり、その概要をプログラム 2 に示す。粒子がどの格子の近くにあるかを計算し(2-3 行目)、隣接格子に対する重みを計算し(4-13 行目)、荷電粒子の位置における電流を重みに基づいて隣接格子へ割り振る(14 行目-)。電流密度計算カーネルにも速度計算カーネルと同様に粒子の位置に対するリストアクセスが存在するが、配列へのアクセスはランダムロード及びストアの両方となる。また、スレッド間で同じインデックスにアクセスする可能性もあるため、配列の属性は reduction となる。

以下では、これら速度計算カーネルと電流密度計算カーネルの2つについて性能測定を行った。

2.6.1.3 測定環境

性能測定には、名古屋大学の旧システムである FX10 と CX400/250 及び、新システムである FX100 と CX400/2500 を使用した。なお、本測定は単一ノードで行い、MPI を用いた並列性能の測定は行わない。コンパイラオプションは以下のとおりである。

- CX250/250(インテルコンパイラ): –O3 –ipo –ip –xAVX –openmp
- CX250/250(富士通コンパイラ): –Kfast,AVX,openmp
- CX250/2550(インテルコンパイラ): –O3 –ipo –ip –xCORE–AVX2 –openmp
- CX250/2550(富士通コンパイラ): –Kfast,CORE–AVX2,openmp
- FX10/FX100(富士通コンパイラ): –Kfast,visimpact,openmp

2.6.1.4 測定結果

計算サイズは 1000x1000 格子、144 粒子/格子とした。これは、約 3GB のメモリ使用量に相当する。計算ステップ数は 10 とし、スレッド数を変える強スケーリング測定を行った。

図 1 は、名大旧システムにおけるプラズマ PIC コードのスレッド性能を示す。速度計算カーネルは、富士通コンパイラを用いた場合は高いスケーラビリティが得られているが、インテルコンパイラを用いた場合はスレッド数を増やすにつれて性能が低下している様子が見られる。一方で電流密度計算カーネルは、富士通コンパイラを用いた場合はスレッド数を増やすにつれて性能が低下している様子が見られるが、インテルコンパイラを用いた場合は 4 スレッドまで高い性能を発揮するものの、4 スレッドで性能が飽和する様子が見られた。

図2は、名大新システムにおけるプラズマ PIC コードのスレッド性能を示す。速度計算カーネルは、富士通コンパイラを用いた場合はおおむね 70%以上でスケールしており、インテルコンパイラを用いた場合は若干速度が向上したものの旧システムとほぼ同様の傾向を示した。電流密度計算カーネルは、富士通コンパイラとインテルコンパイラ共に旧システムとほぼ同様の傾向を示した。

次に、粒子のデータが完全にソートされ、電磁場データへのアクセスが連続的になった場合の性能測定を行った。図 3 は、名大旧システムにおけるプラズマ PIC コードのスレッド性能を示す。速度計算カーネルは、富士通コンパイラを用いた場合は 90%以上でスケールしており、インテルコンパイラを用いた場合でも、16 スレッドまでは 90%以上でスケールしたが 24 スレッドの時に性能が極端に低下した。電流密度計算カーネルは、富士通コンパイラを用いた場合は 70%以上でスケールしており、インテルコンパイラを用いた場合でも、16 スレッドまでは 90%近くでスケールしたが 16 スレッドの時に性能が飽和した。

図 4 は、名大新システムにおけるプラズマ PIC コードのスレッド性能を示す。速度計算カーネルも電流密度計算カーネルも、富士通コンパイラとインテルコンパイラ共に旧システムとほぼ同様の傾向を示した。

2.6.1.5 まとめ

本計測で得られた知見は、以下のとおりである。

1. 富士通コンパイラを用いた場合は、ランダムロードは 70%以上でスケールし、連続的なロードは 90%以上でスケールする。
2. ランダムストアのスレッド性能低下は、どちらのコンパイラを用いても、全てのシステムにおいて顕著であり、PIC コードにおけるデータの並び替えは性能面からは必須である。
3. 富士通コンパイラを用いた場合は、連続的なロード及びストアは 70-80%以上でスケールする。
4. インテルコンパイラを用いた場合は、連続的なロード及びストアは 90%以上でスケールするが、全コアを用いた場合のみ極端に性能が低下する。
5. シングルスレッドの性能は、インテルコンパイラを用いた場合のほうが高いが、富士通コンパイラを用いた場合ほうがスレッド性能が高いため、結果的にマルチスレッド性能は富士通コンパイラを用いた場合ほうが高くなる。
6. FX10 と FX100 は似た傾向を示す。ランダムロード及びストアは両者で性能が数割しか変わらないが、連続的なロードは約 3 倍、連続的なストアは約 1.7 倍 FX100 のほうが高速である。
7. IvyBridge と Haswell は傾向を示す。また両者で性能はほとんど変わらない。

プログラム1:速度計算カーネル

```
!$OMP DO
1 DO n=1,Np
2   i = nint(x(n))
3   j = nint(y(n))

4   fx1 = ...
5   fx2 = ...
6   fx3 = ...
7   fy1 = ...
8   fy2 = ...
9   fy3 = ...
10  hx1 = ...
11  hx2 = ...
12  hy1 = ...
13  hy2 = ...

14  pex = ...
15  pey = ey(i-1,j-1)*fx1*hy1 + ey(i  ,j-1)*fx2*hy1 + ey(i+1,j-1)*fx3*hy1 &
        + ey(i-1,j  )*fx1*hy2 + ey(i  ,j  )*fx2*hy2 + ey(i+1,j  )*fx3*hy2
16  pez = ...
17  pbx = ...
18  pby = ...
19  pbz = ...

20   vx(n) = vx(n) + ...
21   vy(n) = vy(n) + ...
22   vz(n) = vz(n) + ...
23 END DO
!$OMP END DO
```

プログラム 2:電流密度計算カーネル

```
!$OMP DO REDUCTION(+: jx,jy,jz)
1 DO n=1,Np
2   i = nint(x(n))
3   j = nint(y(n))

4   fx1 = ...
5   fx2 = ...
6   fx3 = ...
7   fy1 = ...
8   fy2 = ...
9   fy3 = ...
10  hx1 = ...
11  hx2 = ...
12  hy1 = ...
13  hy2 = ...

14 jx(i-1,j-1)=jx(i-1,j-1)+q(n)*vx(n)*hx1*fy1
15 jx(i  ,j-1)=jx(i  ,j-1)+q(n)*vx(n)*hx2*fy1
16 jx(i-1,j  )=jx(i-1,j  )+q(n)*vx(n)*hx1*fy2
17 jx(i  ,j  )=jx(i  ,j  )+q(n)*vx(n)*hx2*fy2
18 jx(i-1,j+1)=jx(i-1,j+1)+q(n)*vx(n)*hx1*fy3
19 jx(i  ,j+1)=jx(i  ,j+1)+q(n)*vx(n)*hx2*fy3
  :
  :
  :
35 END DO
!$OMP END DO
```

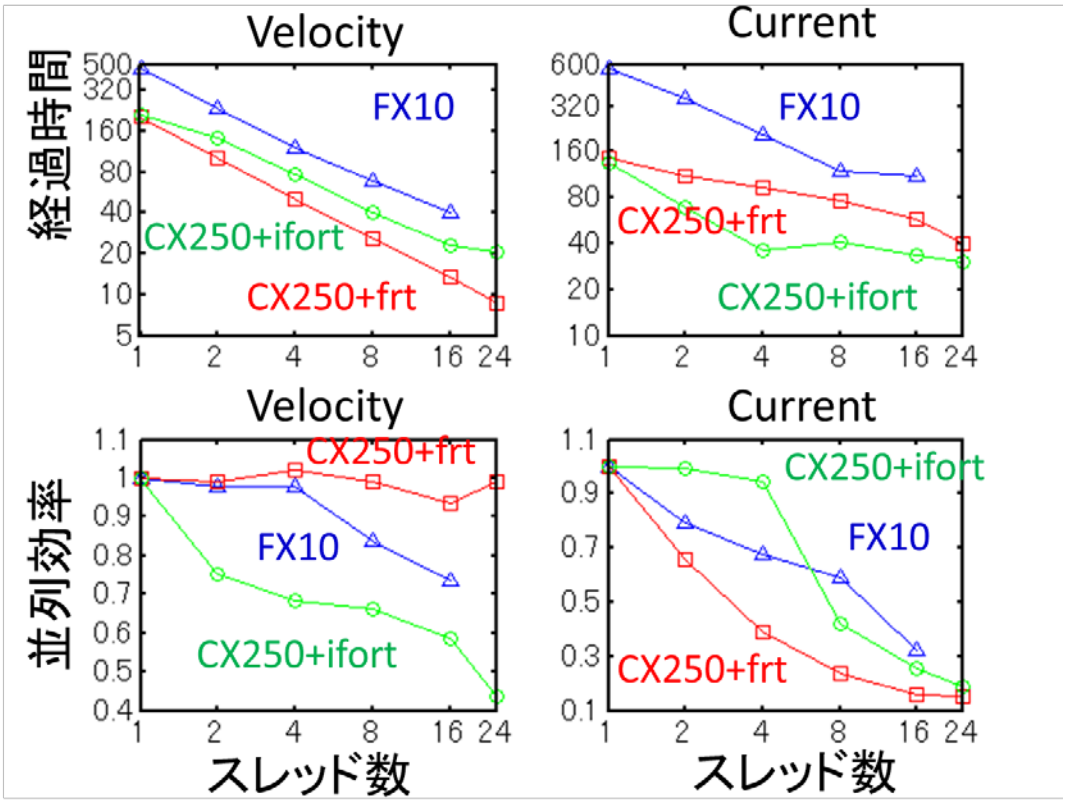


図 1：名大旧システムにおけるプラズマ PIC コードのスレッド性能。ランダムアクセスの場合

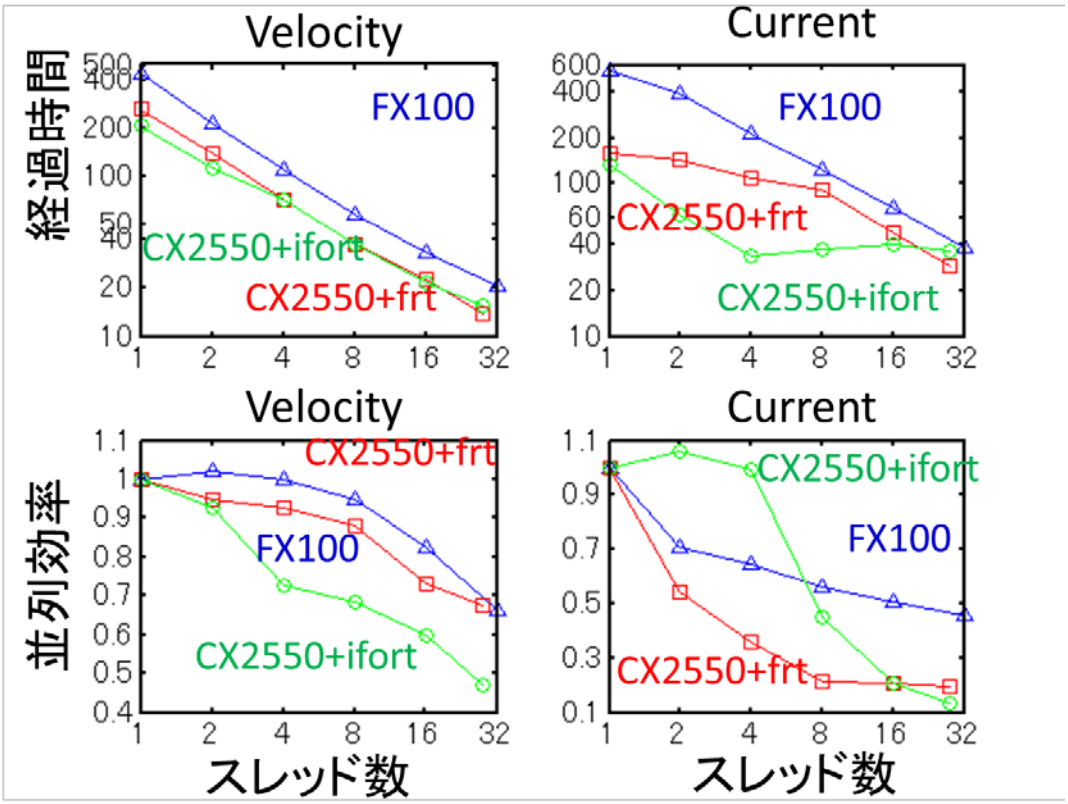


図 2：名大新システムにおけるプラズマ PIC コードのスレッド性能。ランダムアクセスの場合

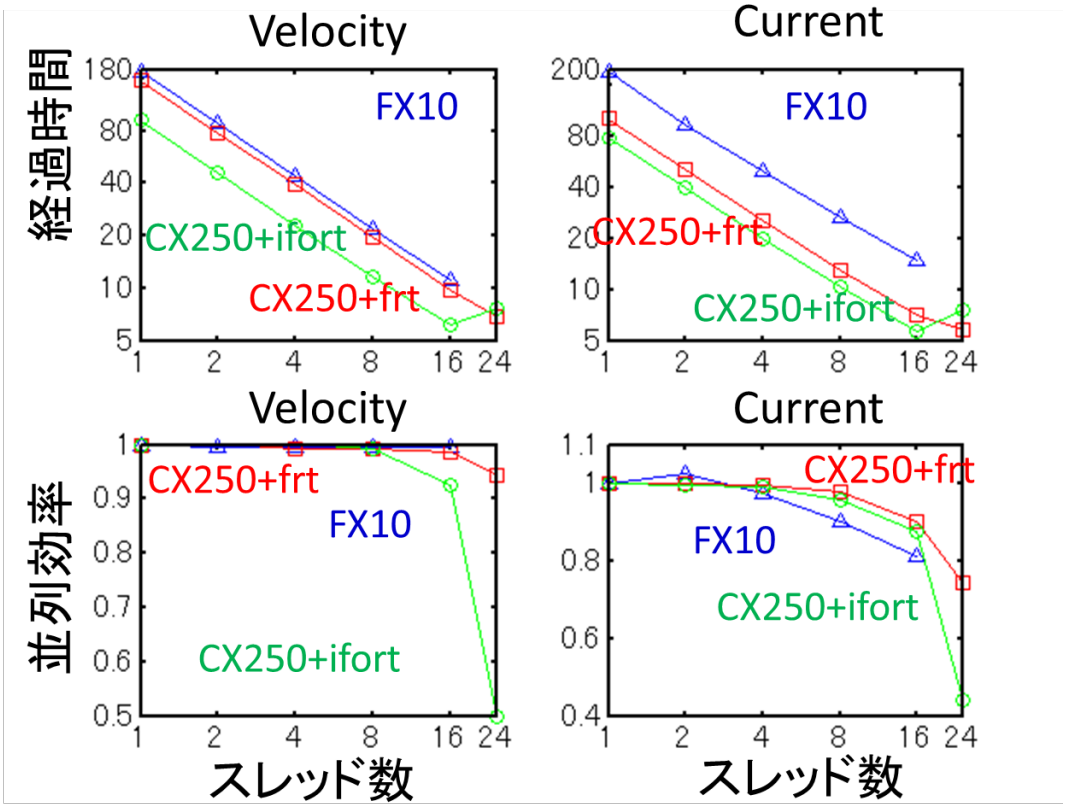


図 3：名大旧システムにおけるプラズマ PIC コードのスレッド性能。粒子データをソートした場合

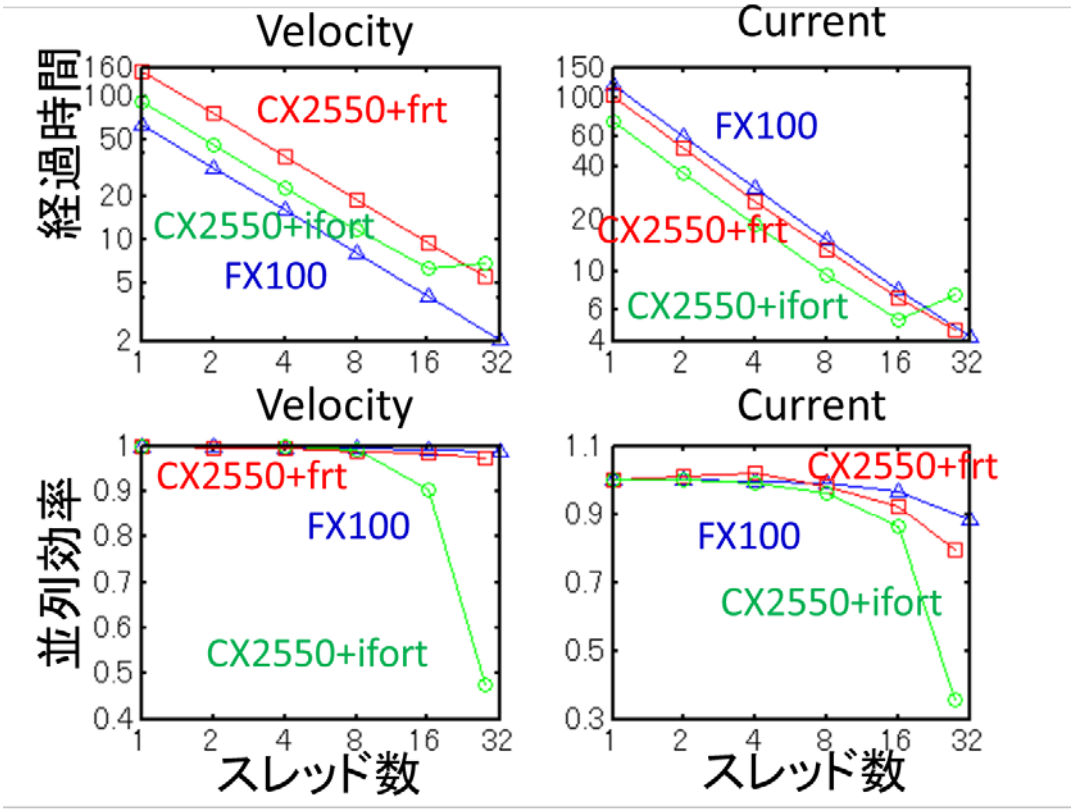


図 4：名大新システムにおけるプラズマ PIC コードのスレッド性能。粒子データをソートした場合

2.6.2 宇宙プラズマ５次元ブラソフコードVlasov5の測定評価

2.6.2.1 はじめに

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

2.6.2.2 プログラム概要

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

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

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

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

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

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

実空間の移流(position カーネル)の概要をプログラム 1 に、速度空間の移流(velocity_e カーネル)の概要をプログラム 2 に示す。両者は同じ演算手法を用いているが、数値流束データを格納するための一時配列の容量が異なることが分かる。分布関数データの定義が、モーメント計算(速度空間の積分)を高速に行うために f(vx,vy,vz,x,y)となっており、velocity_e カーネルでは数値流束データが l,n,m に依存するために 3 次元であるのに対して、position カーネルでは数値流束データが i,j に依存するために 5 次元になっている。

プログラム 1：position カーネルの概要

```
DO j=1,Ny
  DO i=1,Nx
    DO n=1,Nvz
      DO m=1,Nvy
        DO l=1,Nvx
          dfx(l,m,n,i,j)=... !x 方向のフラックスの計算
          dfy(l,m,n,i,j)=... !y 方向のフラックスの計算

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

プログラム 2：velocity_e カーネルの概要

```
DO j=1,Ny
  DO i=1,Nx
    DO n=1,Nvz
      DO m=1,Nvy
        DO l=1,Nvx
          dfx(l,m,n)=... !vx 方向のフラックスの計算
          dfy(l,m,n)=... !vy 方向のフラックスの計算
          dfz(l,m,n)=... !vz 方向のフラックスの計算

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

2.6.2.3 測定環境

性能測定には、名古屋大学のFX100を使用した。なお、本測定は単一ノードで行い、並列性能の測定は行わない。コンパイラオプションは以下のとおりである。

- -Kfast,visimpact,openmp,preex,simd=2 -x250

2.6.2.4 配列の転置

Position カーネルで用いる一時配列の容量を減らすためには、プログラム3で示すような速度と位置を入れ替えた配列 ftmp(x,y,vx,vy,vz)を用いる必要があるが、これには配列の転置を行う必要がある。そこで本プロジェクトでは、配列の転置について、プログラム4を用いて性能評価を行った。プログラム4をそのまま 32 スレッドで走らせると、Nx=166,Ny=86,Nvx=Nvy=Nvz=46 のときに 1 ステップあたり順方向で 2.6 秒、逆方向で 2.5 秒程度掛かる。このプログラムを-Kopenmp を除いてコンパイルすると、順方向逆方向共にソースリストに以下のように表示される。

```
<<<     INTERCHANGED(nest: 2)
           do jj=nys-3,nye+3
<<<     INTERCHANGED(nest: 4)
           do ii=nxs-3,nxe+3
<<<     INTERCHANGED(nest: 1)
           do nn=nvzs-3,nvze+3
<<<     INTERCHANGED(nest: 3)
           do mm=nvys-3,nvye+3
```

したがって、n, j, m, i, l の順にループを入れ替えたほうが高速になる。またこの場合、COLLAPSE(2)が最も高速であり、このときに 1 ステップあたり順方向で 0.3 秒、逆方向で 0.23 秒となり、約 10 分の 1 に短縮された。

プログラム 3：転置した分布関数を用いた position カーネルの概要

```
DO n=1,Nvz
  DO m=1,Nvy
    DO l=1,Nvx
      DO j=1,Ny
        DO i=1,Nx
          dfx(i,j)=... !x 方向のフラックスの計算
          dfy(i,j)=... !y 方向のフラックスの計算

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

プログラム 4：配列転置

```
!順方向
!$OMP DO COLLAPSE(3)
do j=1,Ny
 do i=1,Nx
  do n=1,Nvz
   do m=1,Nvy
    do l=1,Nvx
     gtmp(i,j,l,m,n)=g(l,m,n,i,j)
     end do
   end do
  end do
 end do
end do
!$OMP END DO
```

```
!逆方向
!$OMP DO COLLAPSE(3)
do n=1,Nvz
 do m=1,Nvy
  do l=1,Nvx
   do j=1,Ny
    do i=1,Nx
    g(l,m,n,i,j)=gtmp(i,j,l,m,n)
    end do
   end do
  end do
 end do
end do
!$OMP END DO
```

2.6.2.5 **最適化指示子(OCL)**

幾つかのプログラムパターンについて、ループアンロールの最適化指示子(unroll(n))及び、ソフトウェアパイプラインの最適化指示子(noswp)について、その有効性を確かめた。まず、パターン1－3の全ての場合において、アンロール回数を 10 にするのが概ね高速であることが分かった。次に、ソフトウェアパイプラインの最適化指示子(noswp)について有り無しの場合を比較したところ、ループ内の配列がループの添え字を直接参照している場合(パターン1)や、ループ内の配列がループの添え字と単純な整数演算を参照している場合(パターン 2)は、ソフトウェアパイプラインを用いた(指示子を入れない)ほうが速く、ループ内の配列がループの添え字と変数との演算になっている場合(パターン 3)は、ソフトウェアパイプラインを用いない(指示子を入れた)ほうが高速であることが分かった。

プログラム 5：最適化指示子確認用のプログラムパターン

```
!パターン１：配列の単純参照
do n=nvzs,nvze
  do m=nvys-1,nvye
!OCL UNROLL(10)
    do l=nvxs,nvxe
      vvy(l,m) = vx(l)*bbx+vy(m)*bby+vz(n)*bbz
      mm0(l,m) = m-floor(vvy(l,m))
      mmv(l,m) = sign(1.0d0,vvy(l,m))
    end do
  end do
end do
```

```
!パターン２：配列の参照が単純な演算
do n=nvzs-1,nvze+1
  do m=nvys-1,nvye+1
!OCL UNROLL(10)
    do l=nvxs-1,nvxe+1
      hp2=f(l,mm0(l,m)+2,n,i,j)
      hp1=f(l,mm0(l,m)+1,n,i,j)
      hp0=f(l,mm0(l,m)  ,n,i,j)
      hm1=f(l,mm0(l,m)-1,n,i,j)
      hm2=f(l,mm0(l,m)-2,n,i,j)
      dfy(l,m)=pic5(hp2,hp1,hp0,hm1,hm2,vvy(l,m))
    end do
  end do
end do

!パターン３：配列の参照が変数演算
do n=nvzs-1,nvze+1
  do m=nvys-1,nvye+1
!OCL NOSWP
!OCL UNROLL(10)
    do l=nvxs-1,nvxe+1
      hp2=f(l,mm0(l,m)+mmv(l,m)*2,n,i,j)
      hp1=f(l,mm0(l,m)+mmv(l,m)  ,n,i,j)
      hp0=f(l,mm0(l,m)           ,n,i,j)
      hm1=f(l,mm0(l,m)-mmv(l,m)  ,n,i,j)
      hm2=f(l,mm0(l,m)-mmv(l,m)*2,n,i,j)
      dfy(l,m)=pic5(hp2,hp1,hp0,hm1,hm2,vvy(l,m))
    end do
  end do
end do
```

以上の OCL の最適化により、プログラム全体として 1 ステップあたり 17.7 秒から 15.7 秒まで短縮された。

2.6.2.6 まとめ

本性能測定で得られた知見は以下の通りである。

- 配列の転置については、OpenMP のオプションを外してコンパイルしたときに、コンパイラが最適化した通りにループを並び替えるのが最も高速であることが分かった。COLLAPSE などのディレクティブはコンパイラの最適化を阻害する場合がるため注意が必要である。

- ループアンロールの最適化指示子については、ループ長が割り切れる数若しくは、ループ長を SIMD 数で割った値の約数にアンロール回数を設定するのが概ね高速であることが分かった。
- ソフトウェアパイプラインの最適化指示子については、配列の添え字が変数の演算で参照されている場合は NOSWP を指定したほうが高速になる場合が多いことが分かった。

FUJITSU
shaping tomorrow with you

超次元配列の転置

転置のソースリスト

FUJITSU

swap(順)

```
26                  !$OMP PARALLEL DEFAULT(PRIVATE) &
27                  !$OMP SHARED(nxs,nxe,nys,nye,nvxs,nvxe,nvys,nvye,nvzs,nvze,gg,gtmp)
28                  !$OMP DO COLLAPSE(3)
29    1   p             do jj=nys-3,nye+3            (-1 ～ 84 回転)
30    2   p               do ii=nxs-3,nxe+3          (-1 ～ 164回転)
31    3   p                 do nn=nvzs-3,nvze+3      (-1 ～ 44 回転)
32    4   p                   do mm=nvys-3,nvye+3    (-1 ～ 44 回転)
                    <<< Loop-information Start >>>
                    <<<  [OPTIMIZATION]
                    <<<    SIMD(VL: 4)
                    <<< Loop-information  End >>>
33    5   p   8v                do ll=nvxs-3,nvxe+3  (-1 ～ 44 回転)
34    5   p   8v                  gtmp(ii,jj,ll,mm,nn)=gg(ll,mm,nn,ii,jj)
35    5   p   8v                end do
36    4   p                   end do
37    3   p                 end do
38    2   p               end do
39    1   p             end do
40                  !$OMP END DO
41                  !$OMP END PARALLE
```

ストライドアクセス　連続アクセス

swap(逆)

```
45                  !$OMP PARALLEL DEFAULT(PRIVATE) &
46                  !$OMP SHARED(nxs,nxe,nys,nye,nvxs,nvxe,nvys,nvye,nvzs,nvze,gg,gtmp)
47                  !$OMP DO COLLAPSE(3)
48    1   p             do nn=nvzs-3,nvze+3          (-1 ～ 44 回転)
49    2   p               do mm=nvys-3,nvye+3        (-1 ～ 44 回転)
50    3   p                 do ll=nvxs-3,nvxe+3      (-1 ～ 44 回転)
51    4   p                   do jj=nys-3,nye+3      (-1 ～ 84 回転)
                    <<< Loop-information Start >>>
                    <<<  [OPTIMIZATION]
                    <<<    SIMD(VL: 4)
                    <<< Loop-information  End >>>
52    5   p   8v                do ii=nxs-3,nxe+3    (-1 ～ 164回転)
53    5   p   8v                  gg(ll,mm,nn,ii,jj)=gtmp(ii,jj,ll,mm,nn)
54    5   p   8v                end do
55    4   p                   end do
56    3   p                 end do
57    2   p               end do
58    1   p             end do
59                  !$OMP END DO
60                  !$OMP END PARALLEL
```

ストライドアクセス　連続アクセス

ループ交換前後のソースリスト

FUJITSU

```
26          !$OMP PARALLEL DEFAULT(PRIVATE) &
27          !$OMP SHARED(nxs,nxe,nys,nye,nvxs,nvxe,nvys,nvye,nvzs
28          !$OMP DO COLLAPSE(3)
29 1 p          do jj=nys-3,nye+3
30 2 p            do ii=nxs-3,nxe+3
31 3 p              do nn=nvzs-3,nvze+3
32 4 p                do mm=nvys-3,nvye+3
33 5 p 8v               do ll=nvxs-3,nvxe+3
34 5 p 8v                 gtmp(ii,jj,ll,mm,nn)=gg(ll,mm,nn,ii,jj)
35 5 p 8v               end do
36 4 p                end do
37 3 p              end do
38 2 p            end do
39 1 p          end do
40          !$OMP END DO
41          !$OMP END PARALLEL

45          !$OMP PARALLEL DEFAULT(PRIVATE) &
46          !$OMP SHARED(nxs,nxe,nys,nye,nvxs,nvxe,nvys,nvye,nvzs
47          !$OMP DO COLLAPSE(3)
48 1 p          do nn=nvzs-3,nvze+3
49 2 p            do mm=nvys-3,nvye+3
50 3 p              do ll=nvxs-3,nvxe+3
51 4 p                do jj=nys-3,nye+3
52 5 p 8v               do ii=nxs-3,nxe+3
53 5 p 8v                 gg(ll,mm,nn,ii,jj)=gtmp(ii,jj,ll,mm,nn)
54 5 p 8v               end do
55 4 p                end do
56 3 p              end do
57 2 p            end do
58 1 p          end do
59          !$OMP END DO
60          !$OMP END PARALLEL
```

ループ交換前

```
26          !$OMP PARALLEL DEFAULT(PRIVATE) &
27          !$OMP SHARED(nxs,nxe,nys,nye,nvxs,nvxe,nvys,nvye,nvzs,nvze,gg,gtmp)
28          !$OMP DO COLLAPSE(2)
29 1 p          do nn=nvzs-3,nvze+3
30 2 p            do jj=nys-3,nye+3
                    do mm=nvys-3,nvye+3
                      do ii=nxs-3,nxe+3
33 4          !OCL NOUNROLL
34 5 p  v               do ll=nvxs-3,nvxe+3
35 5 p  v                 gtmp(ii,jj,ll,mm,nn)=gg(ll,mm,nn,ii,jj)
36 5 p  v               end do
37 4 p                end do
38 3 p              end do
39 2 p            end do
40 1 p          end do
41          !$OMP END DO
42          !$OMP END PARALLEL

46          !$OMP PARALLEL DEFAULT(PRIVATE) &
47          !$OMP SHARED(nxs,nxe,nys,nye,nvxs,nvxe,nvys,nvye,nvzs,nvze,gg,gtmp)
48          !$OMP DO COLLAPSE(2)
49 1 p          do nn=nvzs-3,nvze+3
50 2 p            do jj=nys-3,nye+3
51 3 p              do mm=nvys-3,nvye+3
52 4 p                do ii=nxs-3,nxe+3
53 5 p  v               do ll=nvxs-3,nvxe+3
54 5 p  v                 gg(ll,mm,nn,ii,jj)=gtmp(ii,jj,ll,mm,nn)
55 5 p  v               end do
56 4 p                end do
57 3 p              end do
58 2 p            end do
59 1 p          end do
60          !$OMP END DO
61          !$OMP END PARALLEL
```

SWPするため

ループ交換後

転置(逆)チューニング前後のキャッシュ効率

FUJITSU

- L1Dキャッシュとライン数
  - 64KB/コア　1ライン256バイト(32要素)・256ライン
- チューニング前後のキャッシュミスとキャッシュヒット

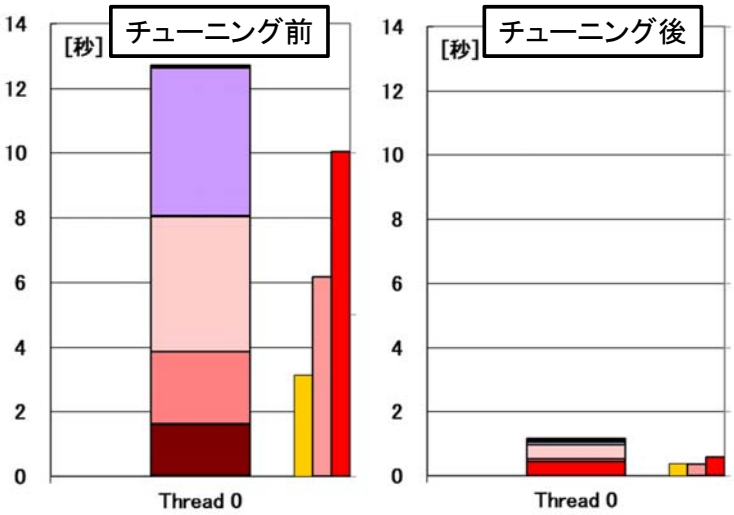
チューニング前		チューニング後	
gg (3 2 1 5 4)	←ネスト数	gtemp (4 2 5 3 1)	←ネスト数
(-1, -1, -1, -1, -1)	キャッシュミス	(-1, -1, -1, -1, -1)	キャッシュミス
(-1, -1, -1, 0, -1)	キャッシュミス	(-1, -1, 0, -1, -1)	キャッシュミス
(-1, -1, -1, 1, -1)	キャッシュミス	(-1, -1, 1, -1, -1)	キャッシュミス
:	:	:	:
(-1, -1, -1, 164, -1)	キャッシュミス	(-1, -1, 44, -1, -1)	キャッシュミス
(-1, -1, -1, -1, 0)	キャッシュミス	(0, -1, -1, -1, -1)	キャッシュヒット
(-1, -1, -1, 0, 0)	キャッシュミス	(0, -1, 0, -1, -1)	キャッシュヒット
(-1, -1, -1, 1, 0)	キャッシュミス	(0, -1, 1, -1, -1)	キャッシュヒット
:	:	:	:
(-1, -1, -1, -1, 1)	キャッシュミス	(1, -1, 1, -1, -1)	キャッシュヒット
(-1, -1, -1, 0, 1)	キャッシュミス	(1, -1, 0, -1, -1)	キャッシュヒット
(-1, -1, -1, 1, 1)	キャッシュミス	(1, -1, 1, -1, -1)	キャッシュヒット
:	:	:	:
gtemp (5 4 3 2 1)	←ネスト数	gg (5 3 1 4 2)	←ネスト数
連続アクセス		連続アクセス	

- チューニング後は、連続アクセスとなることによりキャッシュヒットが増加

転置(逆)チューニング結果

FUJITSU

■ ループ交換によるチューニング



■ グラフは、32スレッド実行のスレッド0のサイクルアカウンティング結果

■ L1Dミス数とL2ミス数が1/10以下に減少して、性能が向上

	L1D ミス率(/ロード・ストア数)	ロード・ストア数	L1D ミス数	L2 ミス率(/ロード・ストア数)	L2 ミス数
チューニング前	58.10%	4.38E+08	2.55E+08	51.28%	2.25E+08
チューニング後	4.44%	4.46E+08	**1.98E+07**	3.39%	**1.51E+07**

キャッシュ効率のチューニング

FUJITSU

■ キャッシュ利用効率を上げられないか？と言う観点でチューニングを試した→ループ交換でキャッシュ利用効率を上げること確認

■ 転置(順)のループ交換チューニングの性能

イタレーション回数	ループ交換無し（秒）	ループ交換有り（秒）	性能比(無し/有り)
1	3.093	0.750	4.12
2	2.603	0.291	8.95
3	2.606	0.291	8.96
4	2.602	0.291	8.94
5	2.604	0.291	8.95

■ 転置(逆)のループ交換チューニングの性能

イタレーション回数	ループ交換無し（秒）	ループ交換有り（秒）	性能比(無し/有り)
1	2.528	0.234	10.80
2	2.550	0.234	10.90
3	2.531	0.234	10.82
4	2.531	0.234	10.82
5	2.539	0.234	10.85

FUJITSU
shaping tomorrow with you

ループアンロール

コードの違い

FUJITSU

- 主なループの違い (nvxs=2、nvxe=41)
- velocity_e (42回転) OCL指定あり
  - sb

```
!OCL NOSWP
!OCL UNROLL(8)
  do ll=nvxs-1,nvxe+1
     hp2=gg(ll,mm,nn+nv+nv,ii,jj)
     hp1=gg(ll,mm,nn+nv    ,ii,jj)
     hp0=gg(ll,mm,nn       ,ii,jj)
     hm1=gg(ll,mm,nn-nv    ,ii,jj)
     hm2=gg(ll,mm,nn-nv-nv,ii,jj)
     t_dgz(ll,mm)=pic5(hp2,hp1,hp0,hm1,hm2,fez)
  end do
```

  - fx

```
!OCL NOSWP
!OCL UNROLL(8)
 do ll=nvxs-1,nvxe+1
    hp2=gg(ll,mm,nn+2,ii,jj)
    hp1=gg(ll,mm,nn+1,ii,jj)
    hp0=gg(ll,mm,nn  ,ii,jj)
    hm1=gg(ll,mm,nn-1,ii,jj)
    hm2=gg(ll,mm,nn-2,ii,jj)
    t_dgz(ll,mm)=pic5s(hp2,hp1,hp0,hm1,hm2,fez)
  end do
```

- ユーザ作成関数pic5sの中で、fezの符号に応じてhp2~hm2を並び替えている。

velocity_eの性能とソース抜粋

依頼1-1 FUJITSU

■ タイマー性能結果（単位：秒）

	区間4-1	区間4-2
velocity_e_sb	13.309	6.969
velocity_e_fx	14.036	6.982

```
do jj=nys-3,nye+3
  do ii=nxs-3,nxe+3
    処理1
  do nn=nvzs-1,nvze+1
    do mm=nvys-1,nvye+1
      do ll=nvxs-1,nvxe+1
        処理2
      end do
      do ll=nvxs-1,nvxe+1
        処理3
      end do
      do ll=nvxs-1,nvxe+1
        処理4
      end do
      do ll=nvxs-1,nvxe+1
        処理5
      end do
    end do
    do mm=nvys-1,nvye+1
      do ll=nvxs-1,nvxe+1
        処理6
      end do
    end do
  end do
  do nn=nvzs,nvze
    do mm=nvys,nvye
      処理7
    end do
  end do
end do
end do
```

!OCL NOSWP
!OCL UNROLL(8)

4-1

!OCL LOOP_NOFUSION
!OCL NOSWP
!OCL UNROLL(8)

!OCL NOSWP
!OCL UNROLL(8)

4-2

4-3

- OCL指定している区間4-1,4-2を分析以降のページで説明
- 最内ループの回転数は42回転
- 区間4-2はsbとfxで同じ
- 最内のネスト5でタイマーを採取するとオーバーヘッドが多くなり正しく取れない。

UNROLL数を変更した測定結果

FUJITSU

■ e_sb とe_fxの区間4-1(42回転)の測定結果比較（単位：秒）

区間4-1	-Knoswp		
-Kunroll=N	リスタ	e_sb	e_fx
1 (nounroll)	SIMD 有効 SWP 無効	16.541	17.364
2	〃	12.974	13.455
3	〃	14.082	14.198
4	〃	13.713	14.315
5	〃	12.080	13.763
6	〃	14.122	15.072
7	〃	13.509	14.604
8	〃	13.237	13.915
9	〃	12.611	13.878
10	〃	12.175	12.951

■ UNROLL展開数は、10,5または2が速い

- 42回転を4(SIMD)×10回転＋2回転(余り)となり、10回転をUNROLLすることで、2,5,10展開の場合は余りが出ないため速い結果であると考えられる。

UNROLL数を変更した測定結果

FUJITSU

■ e_sb とe_fxの区間4-2(42回転)の測定結果比較（単位：秒）

区間4-2	-Knoswp		
-Kunroll=N	リスタ	e_sb	e_fx
1 (nounroll)	SIMD 無効 SWP 無効	11.273	11.287
2	〃	8.540	8.583
3	〃	7.544	7.575
4	〃	7.069	7.151
5	〃	7.171	7.219
6	〃	6.885	6.848
7	〃	6.909	6.872
8	〃	7.124	7.152
9	〃	7.318	7.331
10	〃	6.845	6.850

■ UNROLL展開数は、6,7,10が速い

- SIMDが無効で回転数が42回転であるため、unroll数は割り切れる6,7が高速であったと考えられる。また、unroll数が10も高速であるが、これは命令の並列性などのメリットが余りループ(2回転)のデメリットを上回ったと考えられる。

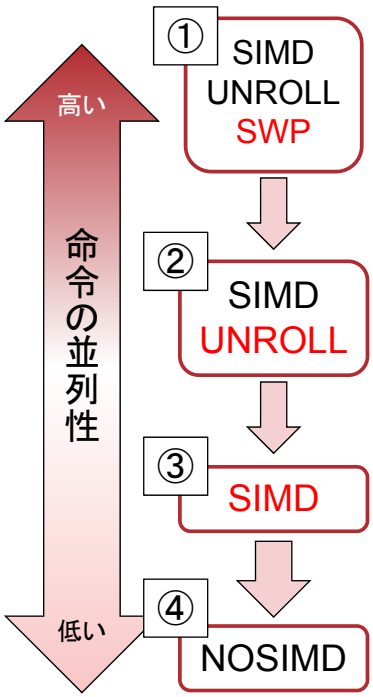
ループ最適化処理（複数生成）について

FUJITSU

■ 翻訳時にループ回転数が分からない場合の最適化処理

- 翻訳時にループ回転数を考慮した最適化処理を複数生成（以下の ▭ ）する
  この場合、実行時に最適化処理が確定する

- 例えばSIMD(4展開),UNROLL(8展開),SWPが行えるループ回転数が230であった場合
  以下のように最適化処理が動作する
  ① 230÷96 = 2 余り38 ⇒この処理を2回転
  回転数96は翻訳時最適化メッセージ(*)で確認
  ② 38÷(4(SIMD)×8(UNROLL)) = 1 余り6
  ⇒ この処理を1回転
  ③ 6÷4(SIMD) = 1 余り2 ⇒ この処理を1回転
  ④ この処理を2回転



* : jwd8205o-I ループの回転数が96回以上の時,ソフトウェアパイプライニングを適用したループが実行時に選択されます.