# 富士通 PRIMEHPC FX10 チューニング連載講座 IV

# 疎行列演算の高速化：OpenMP によるノード内並列化（その 1）

中 島 研 吾

東京大学情報基盤センター

## 1. はじめに

有限要素法, 差分法等の科学技術アプリケーションは最終的には大規模な疎行列を係数とする連立一次方程式を解くことに帰着される．安定で効率的な解法の研究開発は重要な技術的課題であり，特に昨今では大規模並列計算をターゲットとした前処理付き反復法に関する研究が盛んである〔1,2〕．大規模なマルチコアクラスタ，メニィコアクラスタではノード間のメッセージパッシング（例：MPI），ノード内のスレッド並列（例：OpenMP）に基づくハイブリッド並列プログラミングモデルが広く使用されている〔1,2〕．

本稿は，ハイブリッド並列化に向けて，各ノードにおける OpenMP 並列化のための注意事項について説明する．本稿で扱う題材は有限体積法によるアプリケーションである．本稿の内容は単一のアプリケーションに特化した内容であるが，基本的な考え方は様々な分野に適用可能である．

本稿の題材は，これまで当センターの Hitachi SR11000/J1，Hitachi HA8000 クラスタシステム（T2K 東大）を対象とした並列プログラミング講習会，解説記事〔3,4,5,6〕で扱われて来たものと同様である．現在も Fujitsu PRIMEHPC FX10（Oakleaf-FX）によるお試しアカウント付き並列プログラミング講習会「科学技術計算のためのマルチコアプログラミング入門：Hybrid 並列プログラミングモデルへの道」として実施しているほか，本学大学院理学系研究科地球惑星科学専攻で「理学フロンティア科目」として開講されている「並列計算プログラミング・先端計算機演習」においても教材として取り上げられている．2012 年度夏期集中講義の詳細な資料を講義のホームページ[1]で見ることができる．次回講習会は 2013 年 7 月 2 日・3 日に開催される第 31 回講習会[2]であるが，本稿に記述されているような内容も含めて現在教材をアップデート中である．

本稿では特に有限体積法によってポアソン方程式を解くことによって得られる疎行列を係数行列とする大規模連立一次方程式を代表的な前処理手法である ICCG 法（Conjugate Gradient Method preconditioned by Incomplete Cholesky Factorization）の並列化に主眼を置いている．ハイブリッド並列化において重要なのは，各ノード内におけるスレッド並列化である．特に ICCG 法はメモリへの書き込みと参照が同時に発生するデータ依存性（Data Dependency）のあるプロセスを含んでいるため，並列化を実施するには，適切なデータの並べ替え（リオーダリング：reordering）によって並列性を抽出する必要がある．このような対策は OpenMP 向けの解説書でも詳しく取り上げられることは余りない．

以下，過去の記事〔3,4,5,6〕と重複する点もあるが，有限体積法の基礎的な考え方から 2 回に分けて詳細に説明する．なお，OpenMP の文法等については文献〔7,8〕を参照されたい．

---

[1] http://nkl.cc.u-tokyo.ac.jp/12e/05-Multicore/

[2] http://www.cc.u-tokyo.ac.jp/support/kosyu/31/

## 2. アプリケーションの概要

本稿で対象とするアプリケーション（以下「対象アプリケーション」）は図1に示す差分格子によってメッシュ分割された三次元領域において，以下のポアソン方程式を解くものである：

$$\Delta\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} + \frac{\partial^2\phi}{\partial z^2} = f \tag{1}$$

$$\phi = 0 @ z = z_{\max} \tag{2}$$

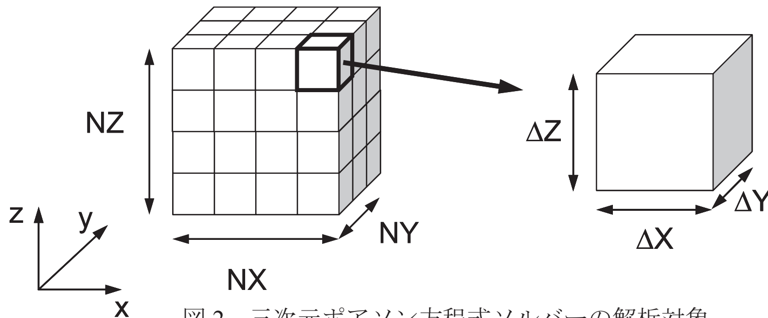



図2　三次元ポアソン方程式ソルバーの解析対象

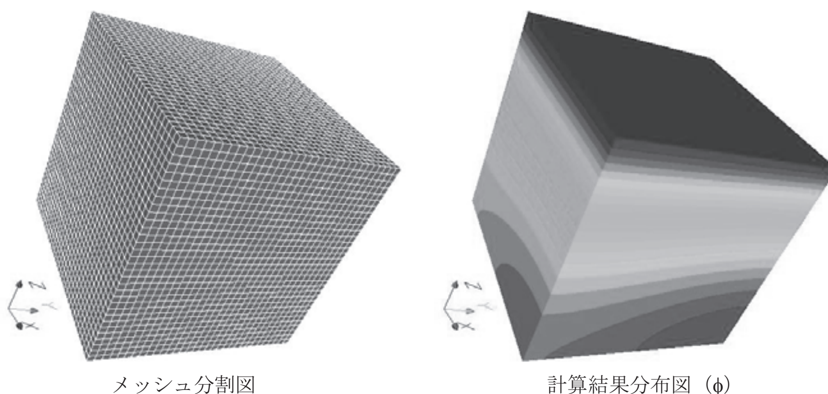
差分格子の各メッシュは直方体（辺長さはΔX，ΔY，ΔZ），X，Y，Z各方向のメッシュ数はNX，NY，NZ

ここで，式（1）の右辺のfは体積あたりのフラックス（flux，流束）項で，以下に示すような空間分布を有すると仮定している：

$$f = dfloat(i_0 + j_0 + k_0) \tag{3}$$

$$i_0 = XYZ(icel,1), \quad j_0 = XYZ(icel,2), \quad k_0 = XYZ(icel,3) \tag{4}$$

式（4）における*XYZ(icel, k)*（k=1,2,3）はX，Y，Z方向の差分格子のインデックスで各メッシュがX，Y，Z方向の何番目にあるかを示している．支配方程式（1）を境界条件（2）とフラックスの条件（3），（4）を適用して解いた計算結果の例（ϕの分布）を図2に示す〔5〕.



メッシュ分割図　　計算結果分布図（ϕ）

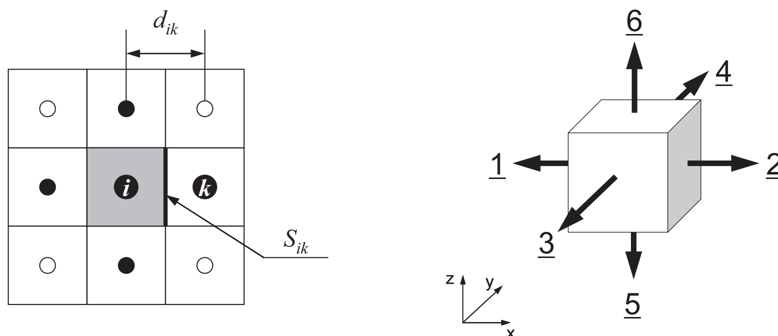
図2　計算結果例（$32^3$=32,768メッシュ）

形状は規則正しい差分格子であるが，プログラムの中では，一般性を持たせるために，有限

体積法に基づき，非構造格子型のデータとして考慮する．図 2 における任意のメッシュ $i$ の各面を通過するフラックスについて，式（1）により以下に示す式（5）のような釣り合い式が成立する：

$$\sum_{k=1}^{6}\left[\frac{S_{ik}}{d_{ik}}\left(\phi_k-\phi_i\right)\right]+V_i f_i=0 \tag{5}$$

ここで，$S_{ik}$：メッシュ $i$ と隣接メッシュ $k$ 間の表面積，$d_{ik}$：メッシュ $i$-$k$ 重心間の距離，$V_i$：メッシュ $i$ の体積，$f_i$：メッシュ $i$ の体積あたりフラックスである（図 3（a）参照）．三次元問題の場合，各直方体メッシュは 6 個の面を持っているため，隣接メッシュ数は（最大で）6 であり，式（5）左辺第一項は k=1~6 の和となっている．図 3（b）は，三次元問題における各メッシュの局所面番号（すなわち隣接メッシュの番号付けのルール）である．




（a）メッシュ $i$ と隣接メッシュ $k$ の関係　　（b）局所面番号，隣接メッシュナンバリング

図 3　隣接メッシュとの関係

式（5）を図 1 のような三次元領域に適用すると，メッシュ $i$ について式（6）が得られる：

$$\begin{aligned}&\frac{\phi_{k=1}-\phi_i}{\Delta x}\Delta y\Delta z+\frac{\phi_{k=2}-\phi_i}{\Delta x}\Delta y\Delta z+\frac{\phi_{k=3}-\phi_i}{\Delta y}\Delta z\Delta x+\frac{\phi_{k=4}-\phi_i}{\Delta y}\Delta z\Delta x+\\&\frac{\phi_{k=5}-\phi_i}{\Delta z}\Delta x\Delta y+\frac{\phi_{k=6}-\phi_i}{\Delta z}\Delta x\Delta y=f_i\Delta x\Delta y\Delta z\end{aligned} \tag{6}$$

これを整理すると，式（7）のようになり，三次元差分法の場合と同様の式が得られる：

$$\frac{\phi_{k=1}-2\phi_i+\phi_{k=2}}{\Delta x^2}+\frac{\phi_{k=3}-2\phi_i+\phi_{k=4}}{\Delta y^2}+\frac{\phi_{k=5}-2\phi_i+\phi_{k=6}}{\Delta z^2}=f_i \tag{7}$$

式（5）にもどって，これを整理すると，式（8）が得られる：

$$\left[\sum_{k=1}^{6}\frac{S_{ik}}{d_{ik}}\right]\phi_i-\left[\sum_{k=1}^{6}\frac{S_{ik}}{d_{ik}}\phi_k\right]=+V_i f_i \tag{8}$$

これは各メッシュ $i$ について成立する式であるので，全メッシュ数を $N$ とすると，$N$ 個の方程

式を連立させて，境界条件を適用し，連立一次方程式 $[A]\{\phi\}=\{b\}$ を解くことに帰着される．式（8）の左辺第一項は$[A]$の対角項，第二項は非対角項，式（8）の右辺は$\{b\}$に対応している．式（8）からもわかるように，各メッシュ $i$ に対応する非対角成分の数は最大 6 個であるので，係数行列$[A]$は疎（sparse）な行列となる．

## 3. ICCG 法について

式（8）を連立させて得られる連立一次方程式$[A]\{\phi\}=\{b\}$を解く方法として，本稿では反復法（Iterative Method），特に Krylov 部分空間法といわれる，非定常型の反復法を使用する〔9〕．係数行列$[A]$は，式（7）からも類推されるように，対称かつ正定（Symmetric Positive Definite）であり，このような行列に対しては，通常は共役勾配法（Conjugate Gradient Method，CG 法）が使用される〔7〕．Krylov 部分空間法の収束は係数行列の性質（固有値分布）に強く依存するため，前処理を適用して固有値が 1 の周辺に集まるように行列の性質を改善する．前処理行列を$[M]$とすると，$[\tilde{A}]=[M]^{-1}[A]$，$\{\tilde{b}\}=[M]^{-1}\{b\}$として（$[M]^{-1}$を右から乗ずる場合もある），すなわち$[\tilde{A}]\{\phi\}=\{\tilde{b}\}$という方程式を代わりに解くことになる．$[M]^{-1}$ が$[A]^{-1}$ をよく近似した行列であれば，$[\tilde{A}]=[M]^{-1}[A]$は単位行列に近くなり，それだけ解きやすくなる．前処理付き CG 法のアルゴリズムの概要はリスト 1 のようになる：

```
compute {r}^(0)={b}-[A]{x}^(0)
for i= 1, 2, …
    solve [M]{z}^(i-1)={r}^(i-1)
    ρ_{i-1}= {r}^(i-1) {z}^(i-1)
    if i=1
      {p}^(1)={z}^(0)
     else
      β_{i-1}= ρ_{i-1}/ρ_{i-2}
      {p}^(i)={z}^(i-1) + β_{i-1}{z}^(i)
    endif
    {q}^(i)=[A]{p}^(i)
    α_i = ρ_{i-1}/{p}^(i){q}^(i)
    {x}^(i)={x}^(i-1) + α_i{p}^(i)
    {r}^(i)={r}^(i-1) - α_i{q}^(i)
    check convergence |r|
End
```

リスト 1　前処理付き CG 法（共役勾配法，Conjugate Gradient Method）のアルゴリズム

前処理手法として広く使用されているのは，不完全 LU 分解（Incomplete LU Factorization，ILU）である．「完全」LU 分解は$[A]=[L][U]$のように係数行列を上下三角行列の積に分解し，前進後退代入によって連立一次方程式の解を求める直接法（direct method）の一種である．前項の最後で述べたように，本対象アプリケーションの場合は$[A]$は疎な行列であるが，$[L]$，$[U]$は必ずしもそうではなく，もともと 0 であったところに非ゼロ成分（fill-in）が生じる場合もある．不完全LU分解ではこの fill-in のレベルや数を制御して，前処理行列$[M]=[\tilde{L}][\tilde{U}](\approx[A])$を生成する．実用的には，ILU(0)（カッコ内は fill-in のレベル），すなわち fill-in を全く考慮しない場合でも，広範囲の問題に対応できる．ILU(0)では，元の行列$[A]$と前処理行列$[M]$の非ゼロ成分の位置が同じとなる．

対称行列における LU 分解に相当するものとして，コレスキー分解（Cholesky Factorization）がある．本稿では，コレスキー分解の一種である，修正コレスキー分解（Modified Cholesky Factorization）を適用する．修正コレスキー分解は，通常のコレスキー分解$[A]=[L][L]^T$の代わり

に以下に示すような$[A]=[L][D][L]^T$を用いる手法である．

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} & 0 \\ l_{51} & l_{52} & l_{53} & l_{54} & l_{55} \end{bmatrix} \begin{bmatrix} d_1 & 0 & 0 & 0 & 0 \\ 0 & d_2 & 0 & 0 & 0 \\ 0 & 0 & d_3 & 0 & 0 \\ 0 & 0 & 0 & d_4 & 0 \\ 0 & 0 & 0 & 0 & d_5 \end{bmatrix} \begin{bmatrix} l_{11} & l_{21} & l_{31} & l_{41} & l_{51} \\ 0 & l_{22} & l_{32} & l_{42} & l_{52} \\ 0 & 0 & l_{33} & l_{43} & l_{53} \\ 0 & 0 & 0 & l_{44} & l_{54} \\ 0 & 0 & 0 & 0 & l_{55} \end{bmatrix}$$

$$\sum_{k=1}^{i} l_{ik} \cdot d_k \cdot l_{jk} = a_{ij} \quad (j = 1,2,\cdots,i-1) \tag{9}$$

$$\sum_{k=1}^{i} l_{ik} \cdot d_k \cdot l_{ik} = a_{ii} \tag{10}$$

ここで$l_{ii} \cdot d_i = 1$とすると以下が成立する：

$$\begin{aligned} & i = 1,2,\cdots,n \\ & \quad j = 1,2,\cdots,i-1 \\ & \quad\quad l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} \cdot d_k \cdot l_{jk} \\ & \quad d_i = \left( a_{ii} - \sum_{k=1}^{i-1} l_{ik}{}^2 \cdot d_k \right)^{-1} = l_{ii}{}^{-1} \end{aligned} \tag{11}$$

本稿では, fill-inを考慮しない以下のような不完全（修正）コレスキー分解（Incomplete (Modified) Cholesky Factorization，IC）を適用し，前処理行列の成分$\tilde{d}_i, \tilde{l}_{ij}$を計算する（fill-inを考慮しないことから正確にはIC(0)とするべきである）．すなわち，$a_{ij} \neq 0$の場合にのみ$\tilde{l}_{ij} = a_{ij} - \sum_{k=1}^{j-1} \tilde{l}_{ik} \cdot \tilde{d}_k \cdot \tilde{l}_{jk}$を計算している：

$$\begin{aligned} & i = 1,2,\cdots,n \\ & \quad j = 1,2,\cdots,i-1 \\ & \quad \tilde{l}_{ij} = a_{ij} - \sum_{k=1}^{j-1} \tilde{l}_{ik} \cdot \tilde{d}_k \cdot \tilde{l}_{jk} \quad if\ a_{ij} \neq 0 \\ & \quad \tilde{d}_i = \left( a_{ii} - \sum_{k=1}^{i-1} \tilde{l}_{ik}{}^2 \cdot \tilde{d}_k \right)^{-1} = \tilde{l}_{ii}^{-1} \end{aligned} \tag{12}$$

$\tilde{l}_{ij} = a_{ij} - \sum_{k=1}^{j-1} \tilde{l}_{ik} \cdot \tilde{d}_k \cdot \tilde{l}_{jk}$の部分は非常に計算時間を要するプロセスであるため$\tilde{l}_{ij} = a_{ij}$と更に省略した形で計算が実施される場合もある．実は，対象アプリケーションの場合，図6に示すように，$\tilde{l}_{ij} = a_{ij} - \sum_{k=1}^{j-1} \tilde{l}_{ik} \cdot \tilde{d}_k \cdot \tilde{l}_{jk}$において$\tilde{l}_{ik}$か$\tilde{l}_{ik}$のいずれかが必ず0となるため，$\tilde{l}_{ij} = a_{ij}$となる．すなわち，前処理行列において非対角成分は元の行列$[A]$と同じであるため，$\tilde{d}_i = \left( a_{ii} - \sum_{k=1}^{i-1} \tilde{l}_{ik}{}^2 \cdot \tilde{d}_k \right)^{-1} = \tilde{l}_{ii}^{-1}$を計算するだけでよい．

対象アプリケーションでは，このような不完全（修正）コレスキー分解（IC）による前処理手法を適用した共役勾配法（CG法），すなわちICCG法を使用する．

前処理付きCG法の $[M]\{z\}=[\tilde{L}\tilde{D}\tilde{L}^T]\{z\}=\{r\}$ を解く（$\{z\}=[LDL^T]^{-1}\{r\}$）部分では：

- 前進代入 $[\tilde{L}]\{y\}=\{r\}$
- 後退代入 $[\tilde{D}\tilde{L}^T]\{z\}=\{y\}$

のようにして，$\{z\}$を計算する．

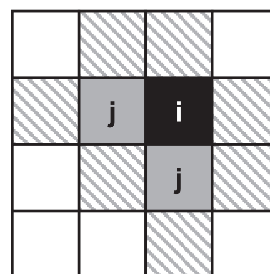



図4　二次元格子における不完全（修正）コレスキー分解

斜線を施したメッシュが $i$ または $j$ に接続する $k$ の候補であるが，$i$ と $j$ に同時に接続するメッシュは存在しない．三次元の場合も同様である．

## 4. 単体CPU用オリジナルプログラムの処理内容

続いて単体CPU用に開発されたオリジナルプログラム〔5〕の処理内容を簡単に説明する．興味を持たれた読者は，Fortran版[3]，C言語版[4]のプログラムをダウンロードして実際に中身を読んだり，計算を実行してみると良い．利用方法は夏期集中講義のホームページ[5]（既出）を参照されたい．

係数行列は対称であるが，対角成分，上三角成分，下三角成分に分けて記憶している．また行列の格納形式としては，非零成分のみを効率的に格納できるCompressed Row Storage（CRS）という方法が採用されている．各成分は表1のように記憶されている：

表1 Compressed Row Storage（CRS）による疎行列格納法

| 配列名 | 型 | 内 容 |
| --- | --- | --- |
| D(i) | 倍精度実数 | 対角成分，i=1~ICELTOT<br>（ICELTOT：全メッシュ数） |
| B(i) | 倍精度実数 | 右辺ベクトル，i=1~ICELTOT |
| indexL(i),indexU(i) | 整数 | 各行の上下三角成分数，i=0~ICELTOT<br>一次元圧縮配列（CRS） |
| itemL(k), itemU(k) | 整数 | 各行の上下三角成分に対応した列番号<br>一次元圧縮配列（CRS）<br>i=0~indexL(ICELTOT), indexU(ICELTOT) |
| AL(k), AU(k) | 倍精度実数 | 各行の上下三角成分<br>一次元圧縮配列（CRS）<br>i=0~indexL(ICELTOT), indexU(ICELTOT) |

表1に示したCRS法によって一次元圧縮配列に格納された疎行列成分を使用してCG法に現れる行列ベクトル積$[A]\{p\}=\{q\}$の計算を実施すると，リスト2のようになる．

[3] http://nkl.cc.u-tokyo.ac.jp/files/multicore-f.tar
[4] http://nkl.cc.u-tokyo.ac.jp/files/multicore-c.tar
[5] http://nkl.cc.u-tokyo.ac.jp/12e/05-Multicore/

```
       do i= 1, ICELTOT
         VAL= D(i)*p(i)
         do k= indexL(i-1)+1, indexL(i)
           VAL= VAL + AL(k)*p(itemL(k))
         enddo
         do k= indexU(i-1)+1, indexU(i)
           VAL= VAL + AU(k)*p(itemU(k))
         enddo
         q(i)= VAL
       enddo
```

リスト 2　CRS 法によって格納された疎行列成分による行列ベクトル積$[A]\{p\}=\{q\}$の計算

① ベクトル内積　$\rho=\{r\}\{z\}$

```
 RHO= 0.d0
 do i= 1, ICELTOT
   RHO= RHO + r(i)*z(i)
 enddo
```

```
  RHO= 0.d0
!$omp   parallel do private(i)
!$omp&  reduction(+:RHO)
  do i= 1, ICELTOT
    RHO= RHO + r(i)*z(i)
  enddo
!$omp end parallel do
```

② ベクトルの実数倍の加減　$\{p\}=\{z\}+\beta\{p\}$

```
 do i= 1, ICELTOT
   p(i)= z(i) + BETA*p(i)
 enddo
```

```
!$omp   parallel do private(i)
  do i= 1, ICELTOT
    p(i)= z(i) + BETA*p(i)
  enddo
!$omp end parallel do
```

③ 行列ベクトル積　$\{q\}=[A]\{p\}$

```
 do i= 1, ICELTOT
   VAL= D(i)*p(i)
   do k= indexL(i-1)+1, indexL(i)
     VAL= VAL + AL(k)*p(itemL(k))
   enddo
   do k= indexU(i-1)+1, indexU(i)
     VAL= VAL + AU(k)*p(itemU(k))
   enddo
   q(i)= VAL
 enddo
```

```
!$omp   parallel do
!$omp & private(i,j,VAL)
  do i= 1, ICELTOT
    VAL= D(i)*p(i)
    do k= indexL(i-1)+1, indexL(i)
      VAL= VAL + AL(k)*p(itemL(k))
    enddo
    do k= indexU(i-1)+1, indexU(i)
      VAL= VAL + AU(k)*p(itemU(k))
    enddo
    q(i)= VAL
  enddo
!$omp end parallel do
```

リスト 3　ベクトル積，ベクトルの実数倍の加減，行列ベクトル積の OpenMP による並列化

ICCG 法（リスト 1）の処理内容は，大きく分けて以下の 4 種類である：

① ベクトルの内積
② ベクトルの実数倍の加減
③ 行列ベクトル積
④ 前処理（前処理行列生成，前進後退代入）

このうち①～③については OpenMP のディレクティヴを挿入するだけで容易に並列化可能である（リスト 3）.

しかし，前処理行列生成部，前進後退代入の部分はこのように単純ではない．例えば，$\tilde{d}_i=\left(a_{ii}-\sum_{k=1}^{i-1}\tilde{l}_{ik}^{\,2}\cdot\tilde{d}_k\right)^{-1}=\tilde{l}_{ii}^{-1}$を計算している部分はリスト 4 のようになる（$\tilde{d}_i \Rightarrow$`DD(i)`）．また，

前進後退代入（$[M]\{z\}=[\tilde{L}\,\tilde{D}\,\tilde{L}^T]\{z\}=\{r\}$を解く部分）はリスト5のようになる（次ページ）．

これらの処理では，DD，z を計算するためのループで，右辺に他のメッシュにおける DD，z が参照されている．図5（a）のような16メッシュから成る領域において，図5（b）のように4スレッドを使用した並列計算を実施しようとしても，例えば前進代入部分では，図5（c）において⇔で示されたメッシュに関して，メモリへの書き込みと参照が同時に生じ，データ依存性が発生する可能性がある（後退代入，前処理行列生成部も同様）．このようなデータ依存性を持つループを OpenMP で強制的に並列化すると，非常に長い計算時間を要したり，正しい計算結果を得られない場合がある．並列計算を実現するためには，こうしたデータ依存性の除去，すなわち右辺で参照される DD(k)や z(k)の内容が，DD(i)や z(i)を計算している間に「絶対に変わらない」ことを保証する必要がある．このような場合，要素間の接続に関する情報をもとに，要素を並び替える（=リオーダリング：reordering）ことによって，データ依存性を除去することが可能である〔4,6〕．次章では，「リオーダリング」の手法と実装について説明する．

```
      do i= 1, ICELTOT
        VAL= D(i)
        do k= indexL(i-1)+1, indexL(i)
          VAL= VAL - (AL(k)**2) * DD(itemL(k))
        enddo
        DD(i)= 1.d0/VAL
      enddo
```

リスト4　不完全（修正）コレスキー分解

```
      do i= 1, ICELTOT
        z(i)= r(i)
      enddo
```

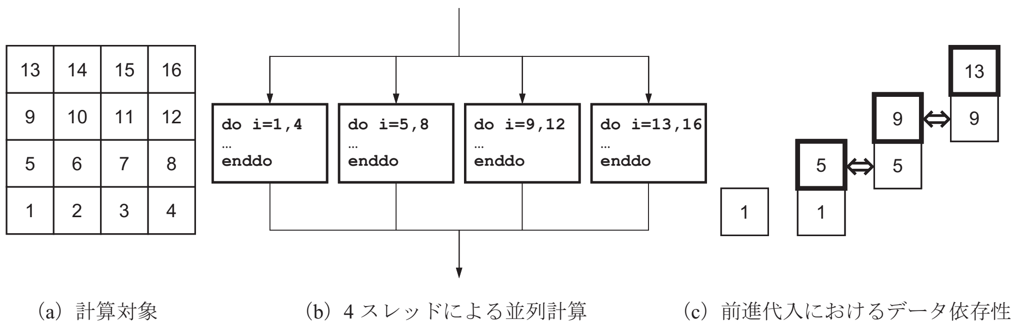
前進代入：$[\tilde{L}]\{y\}=\{r\}$

```
      do i= 1, ICELTOT
        WVAL= z(i)
        do k= indexL(i-1)+1, indexL(i)
          WVAL= WVAL -  AL(k) * z(itemL(k))
        enddo
        z(i)= WVAL * DD(i)
      enddo
```

後退代入：$[\tilde{D}\,\tilde{L}^T]\{z\}=\{y\}$

```
      do i= ICELTOT, 1, -1
        SW  = 0.0d0
        do k= indexU(i-1)+1, indexU(i)
          SW= SW + AU(k) * z(itemU(k))
        enddo
        z(i)= z(i) - DD(i)*SW
      enddo
```

リスト5　前進後退代入




（a）計算対象　（b）4スレッドによる並列計算　（c）前進代入におけるデータ依存性

図5　前進代入において並列計算ができない例

## 5. OpenMP による並列化

### (1) Multicoloring

並列計算のためには，データ依存性の除去が必要であり，図 5 に示した前進代入の場合を例にとると，右辺で参照される `z(itemL(k))` の内容が，`z(i)` を計算している間に「絶対に変わらない」ことを保証する必要がある．そこで「グラフ理論」における「色づけ（coloring）」の手法を適用して，各要素を，依存性を持たない互いに独立な要素ごとに分類する手法が広く用いられている〔4,6〕．図 5 に示した 16 要素の領域は図 6（a）に示すように「2 色」に「色づけ」できる．このような色づけ手法を，チェッカーボードにちなんで，「Red-Black Coloring」と呼ぶ．「red」に色づけされた要素 {`1,3,6,8,9,11,14,16`}（図 6（b），白黒印刷なので「gray」）は，お互いに隣接しておらず，依存関係を持たない独立な要素群である．「black」に色づけされた残りの要素（図 6（c））についても同様である．

| | | | |
|---|---|---|---|
| 13 | 14 | 15 | 16 |
| 9 | 10 | 11 | 12 |
| 5 | 6 | 7 | 8 |
| 1 | 2 | 3 | 4 |

（a）Red-Black Coloring

| | | | |
|---|---|---|---|
| 13 | 14 | 15 | 16 |
| 9 | 10 | 11 | 12 |
| 5 | 6 | 7 | 8 |
| 1 | 2 | 3 | 4 |

（b）red に色づけされた要素

| | | | |
|---|---|---|---|
| 13 | 14 | 15 | 16 |
| 9 | 10 | 11 | 12 |
| 5 | 6 | 7 | 8 |
| 1 | 2 | 3 | 4 |

（b）black に色づけされた要素

図 6　Red-Black Coloring の例

ここで，

```
character*5 COLOR(N)
  COLOR(i)= 'RED  ' (if i= 1,3,6,8, 9,11,14,16)
  COLOR(i)= 'BLACK' (if i= 2,4,5,7,10,12,13,15)
```

なる配列 `COLOR(:)` を導入すると，リスト 5 に示した前進代入の部分はリスト 6 のように書き換えることによって，データ依存性を持たないループとすることができる．図 8 からもわかるように，前進後退代入では「red」の計算をしている間は，右辺には「black」の要素のみが現れ，逆に「black」の計算をしている間は，右辺には「red」の要素のみが現れる．ある「色（red, black）」に属する要素の計算中は，同じ「色」に属する要素は右辺では参照されないことになり，データ依存性が回避される．<u>同じ「色」に属する要素に関する計算は並列に実施できる．</u>

実際の計算では，図 7 に示すように要素を「色」の順に「リオーダリング」するとプログラムも簡便になり，計算効率も高くなる（リスト 7）．リスト 7 の `COLORindex(:)` は各「色」に含まれている要素数を表すための一次元圧縮配列である．図 7 の例の場合は，

```
COLORindex(0)= 0
COLORindex(1)= 8
COLORindex(2)=16
```

であり，1～8 番目の要素が第 1 色（red）に属し（`COLORindex(0)+1=1`，`COLORindex(1)=8`），9 番目～16 番目の要素が第 2 色（black）に属している（`COLORindex(1)+1=9`，`COLORindex(2)=16`）．このように，データ依存性を排除するには，「色づけ（coloring）」と「並び替え（リオーダリング：reordering）」を併用した手法が有効である．一般の複雑な形状に対しては，互いに独立な要素群を抽出するためには，2 色より多い色数が必要となるため，多数

の色を使用したマルチカラー法（Multicoloring，MC 法）が使用される．ここで示した Red-Black 法は色数＝2 で，MC 法のうち，最も単純な形状に対して適用される特別なケースである．

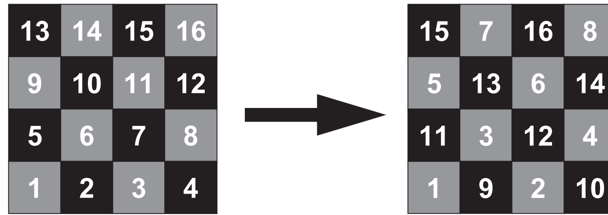



図 7　Red-Black 法の例

```
do i= 1, ICELTOT
  if (COLOR(i).eq.'RED  ') then
    WVAL= z(i)
    do k= indexL(i-1)+1, indexL(i)
      WVAL= WVAL -  AL(k) * z(itemL(k))
    enddo
    z(i)= WVAL * DD(i)
  endif
enddo

do i= 1, ICELTOT
  if (COLOR(i).eq.'BLACK') then
    WVAL= z(i)
    do k= indexL(i-1)+1, indexL(i)
      WVAL= WVAL -  AL(k) * z(itemL(k))
    enddo
    z(i)= WVAL * DD(i)
  endif
enddo
```

リスト 6　　前進代入（Red-Black Coloring によりデータ依存性を除去）

```
do icol= 1, 2
  do i= COLORindex(icol-1)+1, COLORindex(icol)
    WVAL= z(i)
    do k= indexL(i-1)+1, indexL(i)
      WVAL= WVAL -  AL(k) * z(itemL(k))
    enddo
    z(i)= WVAL * DD(i)
  enddo
enddo
```

リスト 7　　前進代入（Red-Black 法の導入）

### (2) Cuthill-McKee, Reverse Cuthill-McKee

レベルセットによる並べ替え法である Cuthill-McKee 法（以下 CM 法），Reverse Cuthill-McKee 法（RCM 法）〔9〕は元々，疎行列のバンド幅やプロフィルを減少させることで，ガウスの消去法や LU 分解における fill-in を減少させ，効率的な計算を実施するための手法である〔9〕．リオーダリングは，本稿のように独立な要素を抽出するだけでなく，様々な用途に使用されている．

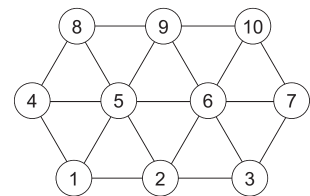



図 8 グラフの例

CM 法は図 8 に示すようなグラフ（点群とそれらを結ぶ辺の集合）において，点を分類する手法である．点群をレベル（level）という，coloring における「色」に相当する考え方に基づき分

類している．以下に CM 法の概要を示す（図 9）：

アルゴリズム 1：CM 法の基本的なアルゴリズム

| |
|---|
| ① 各点に隣接する点の数を「次数（degree)」，最小次数の点を「レベル=1」の点とする． |
| ② 「レベル=k-1」に属する点に隣接する点を「レベル=k」とする．全ての点にレベルづけがされるまで「k」を 1 つずつ増やして繰り返す． |
| ① 全ての点がレベルづけされたら，レベルの順番に再番号づけする（各レベル内では「次数」の番号が少ない順)． |

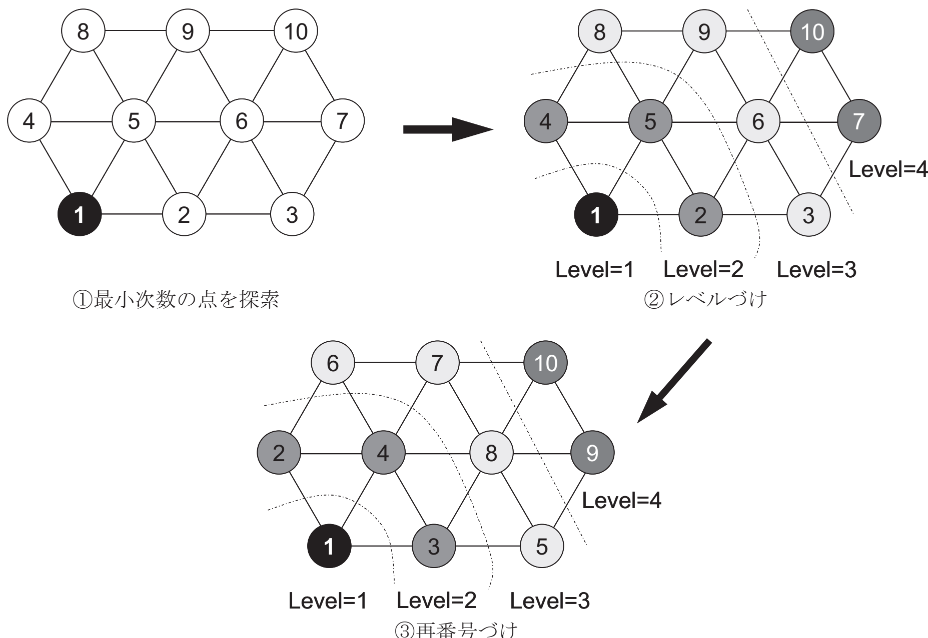
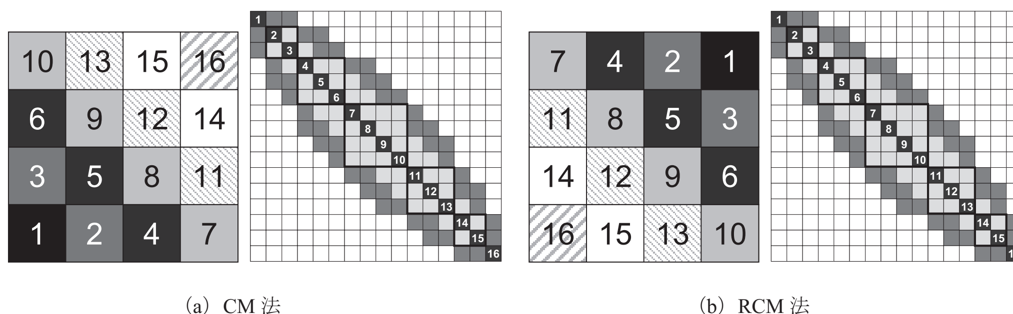



図 9　Cuthill-McKee 法（CM 法）の例

RCM 法は図 9 に示す CM 法で得られた番号を逆順にふり直すもので，LU 分解時の fill-in が減少するため，不完全コレスキー分解時にも有効と考えられる〔9〕．しかし，図 5（a）に示した 16 要素の形状の場合は，fill-in の数はともに 44 である（図 10（a)，(b）参照)．レベル数はともに 7 であり，図 10（a)，(b）に示すように対角線上に並んだ要素群が同じレベルに分類される．これは差分格子における Hyberline（三次元では Hyperplane）と同じである〔4,6,9〕.



（a）CM 法　　（b）RCM 法

図 10　CM 法，RCM 法の適用例（■：非ゼロ成分，■：fill-in（ともに 44))

本稿では，前述の CM 法のアルゴリズムを，同じレベルに属する点，要素同士が独立となるように，以下のように少し修正している：

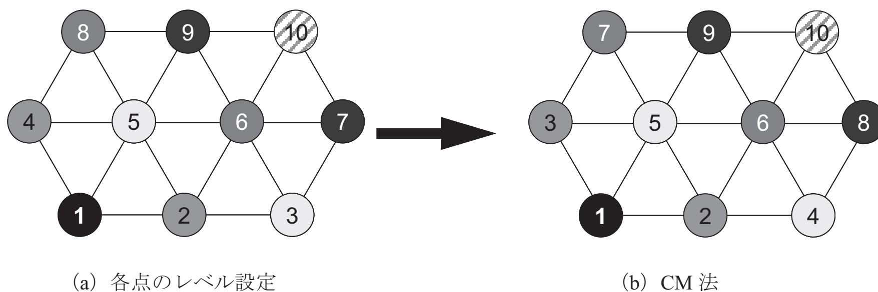
アルゴリズム 2：修正された CM 法のアルゴリズム

① 各要素に隣接する要素数を「次数」とし，最小次数の要素を「レベル=1」の要素とする．
② 「レベル=k-1」の要素に隣接する要素を「レベル=k」とする．同じレベルに属する要素はデータ依存性が発生しないように，隣接している要素同士が同じレベルに入る場合は一方を除外する（現状では先に見つかった要素を優先している）．全ての点要素にレベルづけがされるまで「k」を 1 つずつ増やして繰り返す．
③ 全ての要素がレベルづけされたら，レベルの順番に再番号づけする．

図 8 に示した例に，この修正された CM 法（以下単に「CM 法」と呼ぶ）を適用すると図 11 のようになる．レベルの総数が 4 から 6 に増加している．図 9 では，{2,4,5}（旧番号）は同じレベルに属していたが，図 11 では {5} は違うレベルに属している．

図 5（a）の例に関しては，図 12 に CM 法のレベル番号づけの手順を示す．MC 法と CM 法の大きな違いは，CM 法では，同一レベル（色）における各要素の独立性だけでなく，計算順序を考慮して，レベル間の依存性を考慮している点にある．

MC 法は高い並列性能とスレッド間の負荷分散を容易に達成可能であるが, ICCG 法におけるリオーダリングに適用した場合，悪条件問題で収束が悪化する．それに対して CM 法，RCM 法は悪条件問題でも収束性が MC 法と比較して良い〔1〕.




(a) 各点のレベル設定　　(b) CM 法

図 11　修正された CM 法によるリオーダリング

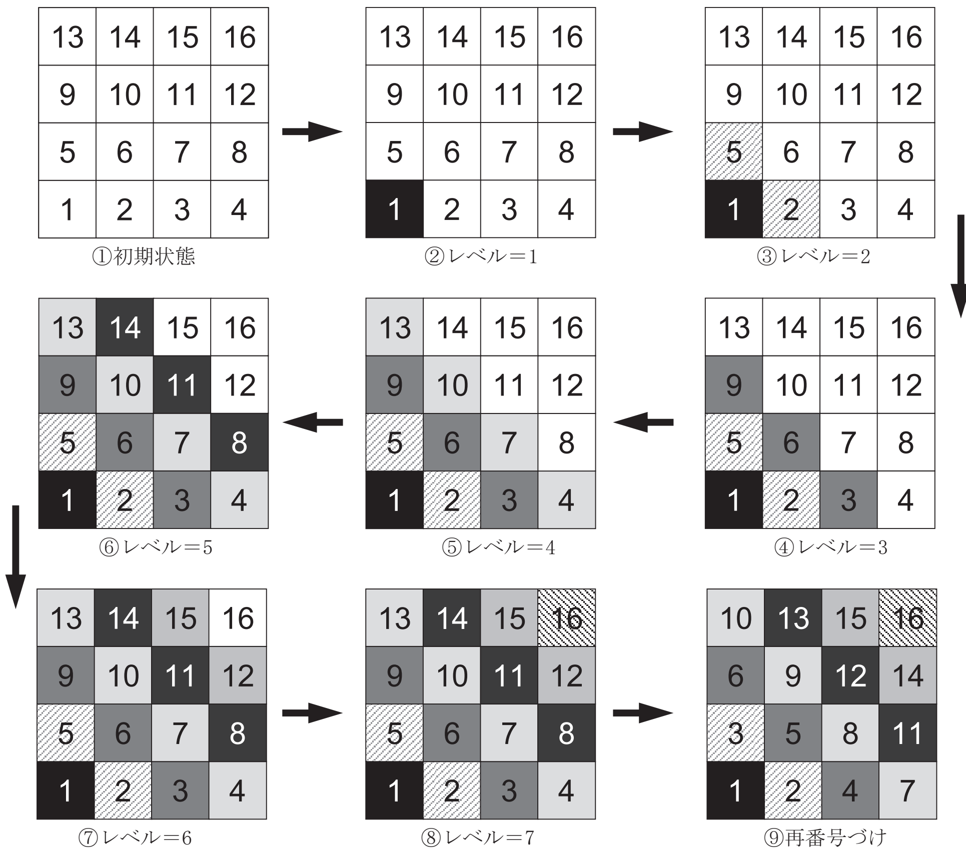



図 12　修正された CM 法によるリオーダリング

### (3) OpenMP 版プログラムの実装

表 2　新たに追加された変数

| 変数，配列名 | 型 | 内 容 |
|---|---|---|
| NCOLORtot | 整数 | 色数，レベル数 |
| COLORindex | 整数 | 各色，レベルに含まれる要素数の一次元圧縮配列，i=0~NCOLORtot，COLORindex(icol-1)+1 から COLORindex(icol)までの要素が icol 番目の色（レベル）に含まれる． |
| NEWtoOLD(i) | 整数 | 新番号⇒旧番号への参照配列，i=1~ICELTOT |
| OLDtoNEW(i) | 整数 | 旧番号⇒新番号への参照配列，i=1~ICELTOT |
| **PEsmpTOT** | 整数 | スレッド数 |
| **SMPindex(k)** | 整数 | スレッド用補助配列（データ依存性があるループに使用），**k= 0~NCOLORtot*PEsmpTOT** |
| **SMPindexG(k)** | 整数 | スレッド用補助配列（データ依存性が無いループに使用），**k= 0~PEsmpTOT** |

表 1 に対して新たに追加された変数名と内容を表 2 に示す．配列「SMPindex」は，不完全修正コレスキー分解，前進後退代入など，データ依存性があるループにおいて，各色（レベル）内のデータを並列に処理するための配列である．図 13 の例では，データ依存性を持たないように 5 色に色づけ，リオーダリングした後に，各色（レベル）内要素を 8 スレッドで並列に処理するためデータ分割（8 等分）している．各スレッドへのデータ分割は，リオーダリング後の

新しい番号順に実施されている．同じスレッドに属する要素は連続した番号を持つように配置される．要素が各スレッドの受け持ち要素数は基本的に各色（レベル）内の要素数の「PEsmpTOT分の 1」である（PEsmpTOT＝スレッド数）．リスト 8 はデータ依存性を持つループを「SMPindex」を使用して並列化する場合の使用方法である．「do ip= 1, PEsmpTOT」のループが並列化され，各々同時に実行される．リスト 9， 10 は，不完全修正コレスキー分解と前進後退代入のループの OpenMP による並列化である．リスト 7 の場合より 1 レベル深いループ構造になっている．

配列「SMPindexG」は，ベクトルの内積，ベクトル行列積などデータ依存性が無いループにおいて，データを並列に処理するために使用される配列である．各スレッドへのデータ分割は，リオーダリング後の新しい番号順に実施されている．同じスレッドに属する要素は連続した番号を持つように配置される．各スレッドの受け持ち要素数は基本的に全要素数の「PEsmpTOT分の 1」である．

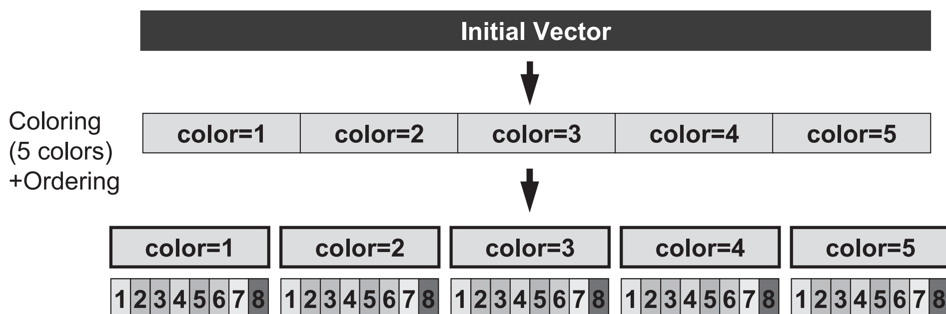



図 13　配列「SMPindex」の基本的な考え方

```
$!omp parallel  ...
      do ic= 1, NCOLORtot
!$omp do
        do ip= 1, PEsmpTOT                          ←並列化されるループ（始まり）
          ip1= (ic-1)*PEsmpTOT+ip
          do i= SMPindex(ip1-1)+1, SMPindex(ip1)
            (...)
          enddo
        enddo                                       ←並列化されるループ（終わり）
      enddo
!$omp end parallel
```

リスト 8　配列「SMPindex」の使用方法

```
!$omp parallel  private(ip,ip1,i,VAL,j)
      do ic= 1, NCOLORtot
!$omp do
        do ip= 1, PEsmpTOT
          ip1= (ic-1)*PEsmpTOT + ip
          do i= SMPindex(ip1-1)+1, SMPindex(ip1)
            VAL= D(i)
            do j= 1, INL(i)
              VAL= VAL - (AL(j,i)**2) * W(IAL(j,i),DD)
            enddo
            W(i,DD)= 1.d0/VAL
          enddo
        enddo
      enddo
!$omp end parallel
```

リスト 9　不完全修正コレスキー分解のループの OpenMP による並列化

```
!$omp parallel private(ip,ip1,i,WVAL,j)
      do ic= 1, NCOLORtot
!$omp do
        do ip= 1, PEsmpTOT
          ip1= (ic-1)*PEsmpTOT + ip
          do i= SMPindex(ip1-1)+1, SMPindex(ip1)
            WVAL= W(i,Z)
            do k= indexL(i-1)+1, indexL(i)
              WVAL= WVAL -  AL(k) * W(itemL(k),Z)
            enddo
            W(i,Z)= WVAL * W(i,DD)
          enddo
        enddo
      enddo
!$omp end parallel

!$omp parallel private(ip,ip1,i,SW,j)
      do ic= NCOLORtot, 1, -1
!$omp do
        do ip= 1, PEsmpTOT
          ip1= (ic-1)*PEsmpTOT + ip
          do i= SMPindex(ip1-1)+1, SMPindex(ip1)
            SW  = 0.0d0
            do k= indexU(i-1)+1, indexU(i)
              SW= SW + AU(k) * W(itemU(k),Z)
            enddo
            W(i,Z)= W(i,Z) - W(i,DD) * SW
          enddo
        enddo
      enddo
!$omp end parallel do
```

リスト 10　前進後退代入（[M]{z}={r}）のループの OpenMP による並列化

リスト 11 はデータ依存性を持たないループを「SMPindexG」を使用して並列化する場合の使用方法である.「do ip= 1, PEsmpTOT」のループが並列化され，各々同時に実行される. リスト 12，リスト 13，リスト 14 はそれぞれ，ベクトルの内積，ベクトルの実数倍の加減，行列ベクトル積のループの OpenMP による並列化である. リスト 3 の場合と比べて，ループが 1 レベル深い構造になっている.

```
!$omp parallel do ...
      do ip= 1, PEsmpTOT
        do i= SMPindexG(ip-1)+1, SMPindexG(ip)
          (...)
        enddo
      enddo
!$omp end parallel do
```

リスト 11　配列「SMPindexG」の使用方法

```
      C1= 0.d0
!$omp parallel do private(ip,i) reduction(+:C1)
      do ip= 1, PEsmpTOT
        do i = SMPindexG(ip-1)+1, SMPindexG(ip)
          C1= C1 + W(i,P)*W(i,Q)
        enddo
      enddo
!$omp end parallel do

      ALPHA= RHO / C1
```

リスト 12　ベクトルの内積（ρ= {r}{z}）のループの OpenMP による並列化

```
!$omp parallel do private(ip,i)
      do ip= 1, PEsmpTOT
        do i = SMPindexG(ip-1)+1, SMPindexG(ip)
          W(i,P)= W(i,Z) + BETA*W(i,P)
        enddo
      enddo
!$omp end parallel do
```

リスト 13　ベクトルの実数倍の加減（{p}= {z}+β{p}）のループの OpenMP による並列化

```
!$omp parallel do private(ip,i,VAL,j)
      do ip= 1, PEsmpTOT
        do i = SMPindexG(ip-1)+1, SMPindexG(ip)
          VAL= D(i)*W(i,P)
          do k= indexL(i-1)+1, indexL(i)
            VAL= VAL + AL(k)*W(itemL(k),P)
          enddo
          do k= indexU(i-1)+1, indexU(i)
            VAL= VAL + AU(k)*W(itemU(k),P)
          enddo
          W(i,Q)= VAL
        enddo
      enddo
!$omp end parallel do
```

リスト 14　行列ベクトル積（{q}= [A]{p}）のループの OpenMP による並列化

（第 2 回（7 月号）へ続く）

## 参 考 文 献


[1] Nakajima, K.: Flat MPI vs. Hybrid: Evaluation of Parallel Programming Models for Preconditioned Iterative Solvers on “T2K Open Supercomputer”, IEEE Proceedings of the 38th International Conference on Parallel Processing (ICPP-09), pp.73-80 (2009)

[2] Nakajima, K.: OpenMP/MPI Hybrid Parallel Multigrid Method on Fujitsu FX10 Supercomputer System, IEEE Proceedings of 2012 International Conference on Cluster Computing Workshops, IEEE Digital Library: 10.1109/ClusterW.2012.35; p.199-206 (2012)

[3] 中島研吾：OpenMPによるプログラミング入門（I），スーパーコンピューティングニュース（東京大学情報基盤センター） 9-5（2007）
http://www.cc.u-tokyo.ac.jp/publication/news/VOL9/No5/200709OpenMP.pdf

[4] 中島研吾：OpenMPによるプログラミング入門（II），スーパーコンピューティングニュース（東京大学情報基盤センター） 9-6（2007）
http://www.cc.u-tokyo.ac.jp/publication/news/VOL9/No6/200711OpenMP.pdf

[5] 中島研吾：OpenMPによるプログラミング入門（III），スーパーコンピューティングニュース（東京大学情報基盤センター）10-1（2008）
http://www.cc.u-tokyo.ac.jp/publication/news/VOL10/No1/200801OpenMP.pdf

[6] 中島研吾：T2K オープンスパコン（東大）チューニング連載講座（その 5）OpenMP による並列化のテクニック：Hybrid 並列化に向けて，スーパーコンピューティングニュース（東京大学情報基盤センター）11-1（2009）
http://www.cc.u-tokyo.ac.jp/support/press/news/VOL11/No1/200901tuning.pdf

[7] Chandra, R.他：Parallel Programming in OpenMP， Morgan Kaufmann Publishers（2001）

[8] 牛島省：OpenMP による並列プログラミングと数値計算法，丸善（2006）

[9] Saad, Y.: Iterative Methods for Sparse Linear Systems Second Edition, SIAM (2003)

[10] Mattson, T.G., Sanders, B.A., Massingill, B.L.: Patterns for Parallel Programming, Software Patterns Series (SPS), Addison-Wesley (2005)