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

中 島 研 吾

東京大学情報基盤センター

1. はじめに

本連載講座も今回で 5 回目を迎える。第 2 回～第 4 回では，様々な高性能化のための技術について紹介した〔1～3〕。本稿は，Hybrid 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。対象とするのは，有限要素法，有限体積法，差分法など，疎行列を係数行列とし，間接参照を伴うアプリケーションである。

T2K オープンスパコンのようなマルチコアクラスタにおける並列プログラミングモデルとして，Flat MPI（または Pure MPI）と Hybrid の 2 種類のモデルが考えられる（図 1）。Flat MPI は全コアを独立に扱い，各コア間にメッセージパッシング（MPI 等）を適用する。Hybrid ではメッセージパッシングは各共有メモリユニット（ノードあるいはソケット）単位で適用し，ノード／ソケット内では OpenMP 等のスレッドによる並列化を実施するものである。

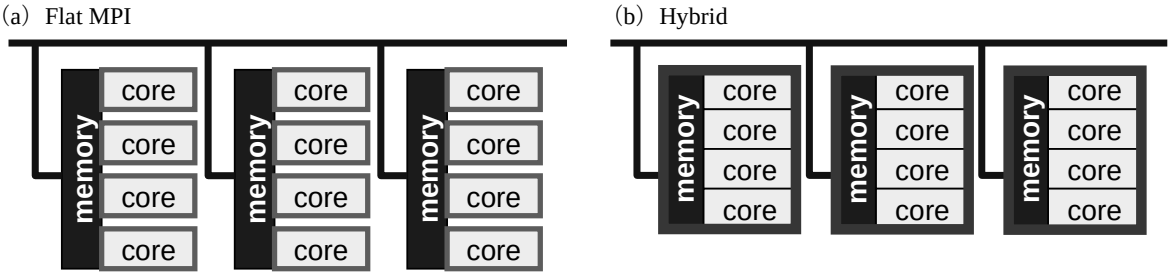


図 1　Flat MPI，Hybrid 並列プログラミングモデル

両プログラミングモデルの優劣については，ここ 10 年ほど様々な比較が試みられている。文献〔4〕では，「地球シミュレータ」，「IBM SP3」，「Hitachi SR8000」等に対して，有限要素法向け ICCG ソルバーの Flat MPI と Hybrid 並列プログラミングモデルの優劣を比較している。ハードウェア・コンパイラ・アプリケーションの特性，問題サイズに依存するものの，全体として，通信がボトルネックになる場合は Hybrid，メモリ性能が支配的な場合は Flat MPI が有利であり，特にコア数，ノード数が増加すると Hybrid の方が有効となる。しかしながら，ある程度の粒度が保たれていれば，数千コア程度の範囲までであれば，両者の性能はあまり変わらず，Hybrid ではノード内の並列化に手間もかかるため余り浸透していない。ところが近年 1 万コアを超えるシステムが続々と登場し，プロセッサのマルチコア化，メニーコア化が進んだことも相俟って，再び Hybrid 並列プログラミングモデルが注目されている。2008 年 11 月 8 に開催された SC08（International Conference for High Performance Computing, Networking, Storage and Analysis）[1]では，Hybrid 並列プログラミングモデルに関連した発表，チュートリアルなどが目を引いた。

Hybrid 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

[1] http://sc08.supercomputing.org/

法などのデータ依存性のある演算では ordering によって並列性を抽出する必要がある〔4〕。加えて，これまでの連載でも述べたように，T2K オープンスパコン（東大）は図 2 に示すような NUMA（Non-Uniform Memory Access）アーキテクチュアによっており，この特性を考慮したプログラミング，データ配置が必要となる。

本稿では 2007 年度に本「スーパーコンピューティングニュース」に連載された「OpenMP によるプログラミング入門（I～III）」〔5～7〕に基づき，T2K オープンスパコン（東大）1 ノード（4 ソケット，16 コア）（図 2）を使用して，有限体積法によるアプリケーションの NUMA アーキテクチュアにおける最適化を考慮した OpenMP 並列化について解説する。なお，本稿の内容は 2008 年 12 月 3 日，4 日に日本応用数理学会「2008 年秋の学校：科学技術計算のためのマルチコアプログラミング入門」（共催：東京大学情報基盤センター）[2]で実施した内容のダイジェスト版である。興味を持たれた読者は，「秋の学校」のホームページを参照されたい。より詳細な説明資料，サンプルプログラムが入手可能である。本稿では FORTRAN で記述されたプログラムによる説明を実施するが，上記「秋の学校」サイトからは C 言語版のプログラムも（一部）入手可能である。規則的な差分格子体系を対象にしているが，有限要素法，不規則な非構造格子にも適用可能な内容である。本稿第 7 章は有限要素法アプリケーションを扱っているが，ほぼ同じ手法を使用している。

T2K オープンスパコン（東大）のノードの構成についてはこれまでの連載でも何回か説明されている。詳細については「利用の手引」〔8〕を参照されたいが，図 2 に示すように 4 つのコアをもつ AMD Quad-core Opteron（2.3 GHz，コアあたりピーク性能：9.2 GFLOPS）が 4 ソケット，すなわち 16 コアで 1 ノードを構成している。ノードあたりのピーク性能は 147.2GFLOPS である。ソケットあたりのメモリ容量は 8GB，ノードあたり 32GB である（一部ノードでは 128GB）。

本稿の最後（7 章）では T2K オープンスパコン（東大）における Flat MPI と Hybrid 並列プログラミングの比較例〔9〕について紹介する。

なお，OpenMP の文法等については文献〔10〕，〔11〕を参照されたい。

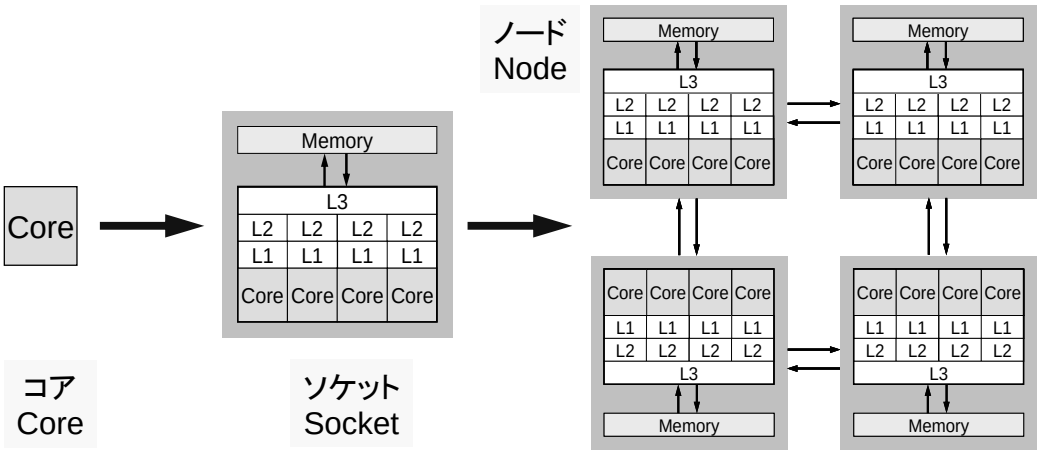


図2　T2Kオープンスパコン（東大）の各ノードの構成〔8〕

[2] http://nkl.cc.u-tokyo.ac.jp/seminars/0812-JSIAM/

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 3 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：

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

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

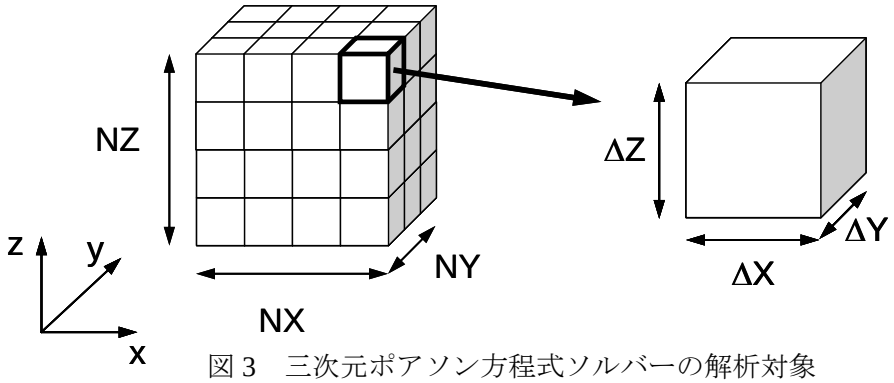


図 3　三次元ポアソン方程式ソルバーの解析対象

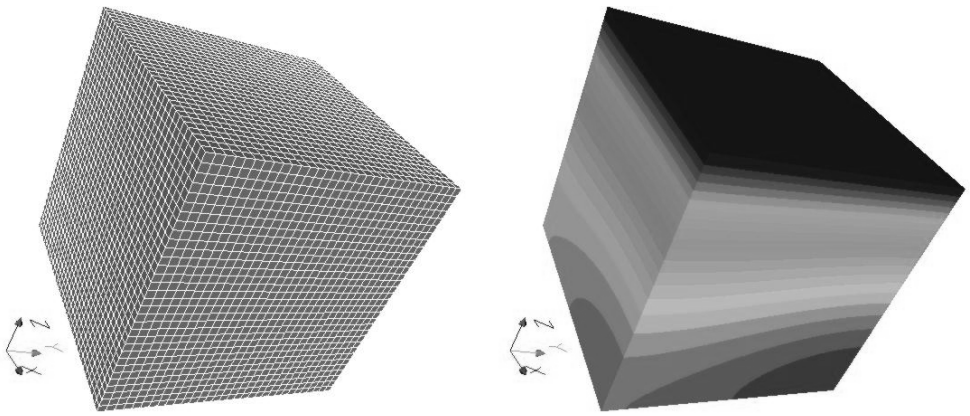
差分格子の各メッシュは直方体（辺長さはΔX，ΔY，ΔZ），X，Y，Z 各方向のメッシュ数は NX，NY，NZ

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 4 に示す〔5〕。



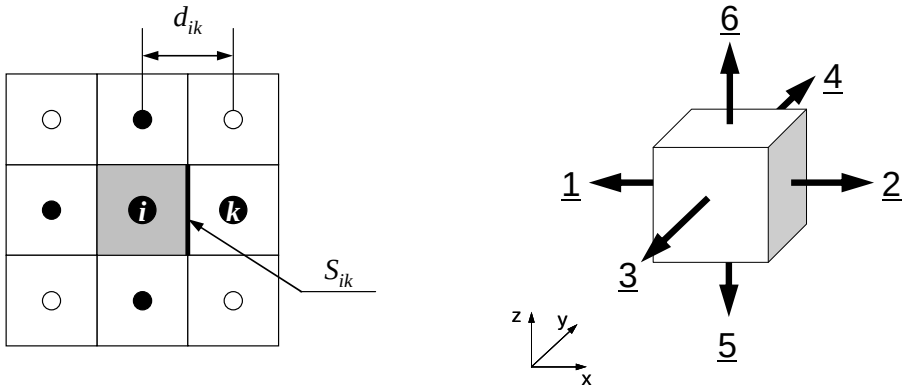
メッシュ分割図　　計算結果分布図（ϕ）

図 4　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 3 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 5（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 5（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 5　隣接メッシュとの関係

式（5）を図 3 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〔12〕。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〔12〕。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)
    ρi-1= {r}(i-1) {z}(i-1)
    if i=1
      {p}(1)={z}(0)
     else
      βi-1= ρi-1/ρi-2
      {p}(i)={z}(i-1) + βi-1{z}(i)
    endif
    {q}(i)=[A]{p}(i)
    αi = ρi-1/{p}(i){q}(i)
    {x}(i)={x}(i-1) + αi{p}(i)
    {r}(i)={r}(i-1) – αi{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：

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

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$\tilde{l}_{ij} = a_{ij} - \displaystyle\sum_{k=1}^{j-1} \tilde{l}_{ik} \cdot \tilde{d}_k \cdot \tilde{l}_{jk}$を計算している：

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

$\tilde{l}_{ij} = a_{ij} - \displaystyle\sum_{k=1}^{j-1} \tilde{l}_{ik} \cdot \tilde{d}_k \cdot \tilde{l}_{jk}$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，$\tilde{l}_{ij} = a_{ij} - \displaystyle\sum_{k=1}^{j-1} \tilde{l}_{ik} \cdot \tilde{d}_k \cdot \tilde{l}_{jk}$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。

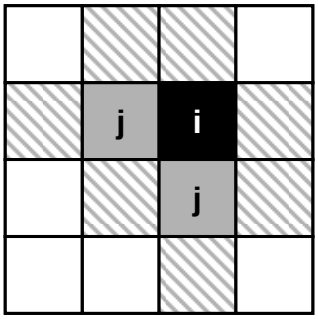


図 6　二次元格子における不完全（修正）コレスキー分解

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。

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$\left[M\right]\{z\} = \left[\tilde{L}\ \tilde{D}\ \tilde{L}^T\right]\{z\} = \{r\}$を解く（$\{z\} = \left[LDL^T\right]^{-1}\{r\}$）部分では：

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

のようにして，{*z*}を計算する。

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。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，非ゼロ成分のみを記憶している。各成分は表 1 のように記憶されている：

表 1　オリジナルプログラムの変数表

配列名	型	内 容
D(i)	倍精度実数	対角成分，i=1~N（N：全メッシュ数）
B(i)	倍精度実数	右辺ベクトル，i=1~N
INL(i)	整数	*i* の下三角成分数，i=1~N
INU(i)	整数	*i* の上三角成分数，i=1~N
IAL(j,i)	整数	*i* の *j* 番目の下三角成分に対応する列番号，i=1~N, j=1~INL(i)
IAU(j,i)	整数	*i* の *j* 番目の上三角成分に対応する列番号，i=1~N, j=1~INU(i)
AL(j,i)	倍精度実数	*i* の *j* 番目の下三角成分，i=1~N，j=1~INL(i)
AU(j,i)	倍精度実数	*i* の *j* 番目の上三角成分，i=1~N，j=1~INU(i)

CG 法に現れる[*A*]{*p*}={*q*}の計算は上記配列により，リスト 2 のように実施されている。

```
      do i= 1, N
        VAL= D(i)*p(i)
        do j= 1, INL(i)
          VAL= VAL + AL(j,i)*p(IAL(j,i))
        enddo
        do j= 1, INU(i)
          VAL= VAL + AU(j,i)*p(IAU(j,i))
        enddo
        q(i)= VAL
      enddo
```

リスト 2　行列ベクトル積[*A*]{*p*}={*q*}の計算

ここでは FORTRAN の場合を示したが，C 言語の場合は二重配列のメモリ上でのアクセスの順番が異なるため，添え字の参照の順番が逆になっている。

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）：

① ベクトル内積　ρ= {r}{z}

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

② ベクトルの実数倍の加減　{p}= {z}+β{p}

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

③ 行列ベクトル積　{q}= [A]{p}

```
do i= 1, N
  VAL= D(i)*p(i)
  do j= 1, INL(i)
    VAL= VAL + AL(j,i)*p(IAL(j,i))
  enddo
  do j= 1, INU(i)
    VAL= VAL + AU(j,i)*p(IAU(j,i))
  enddo
  q(i)= VAL
enddo
```

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

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

```
!$omp   parallel do
!$omp & private(i,j,VAL)
  do i= 1, N
    VAL= D(i)*p(i)
    do j= 1, INL(i)
      VAL= VAL + AL(j,i)*p(IAL(j,i))
    enddo
    do j= 1, INU(i)
      VAL= VAL + AU(j,i)*p(IAU(j,i))
    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

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 7（a）のような 16 メッシュから成る領域において，図 7（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 7（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）。

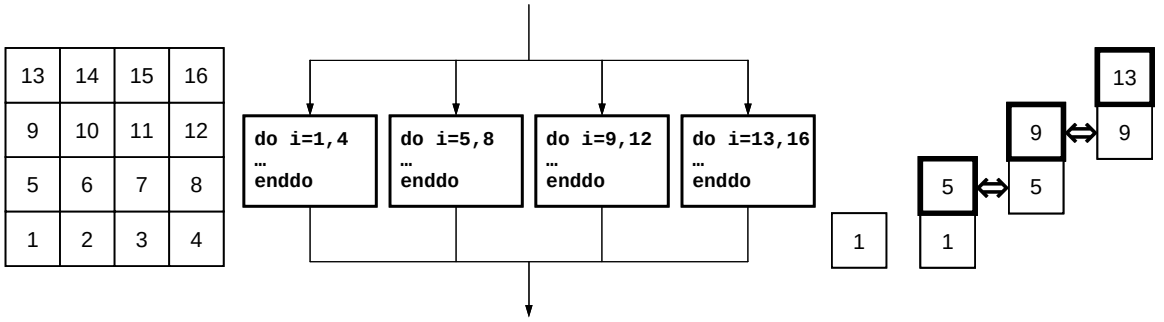
このようなデータ依存性を持つループを OpenMP で強制的に並列化すると，非常に長い計算時間を要したり，正しい計算結果を得られない場合がある。並列計算を実現するためには，こうしたデータ依存性の除去，すなわち右辺で参照される DD(k)や z(k)の内容が，DD(i)や z(i)を計算している間に「絶対に変わらない」ことを保証する必要がある。このような場合，要素間の接続に関する情報をもとに，要素を並び替える（reordering，oredering）ことによって，データ依存性を除去することが可能である〔6〕。次章では，「並び替え」の手法と実装について説明する。

```
      do i= 1, N
        VAL= D(i)
        do j= 1, INL(i)
          VAL= VAL - (AL(j,i)**2) * DD(IAL(j,i))
        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, N
        z(i)= r(i)
      enddo
前進代入：[L̃]{y}={r}
      do i= 1, N
        WVAL= z(i)
        do j= 1, INL(i)
          WVAL= WVAL -  AL(j,i) * z(IAL(j,i))
        enddo
        z(i)= WVAL * DD(i)
      enddo
後退代入：[D̃ L̃ᵀ]{z}={y}
      do i= N, 1, -1
        SW  = 0.0d0
        do j= 1, INU(i)
          SW= SW + AU(j,i) * z(IAU(j,i))
        enddo
        z(i)= z(i) - DD(i)*SW
      enddo
```

リスト 5　前進後退代入



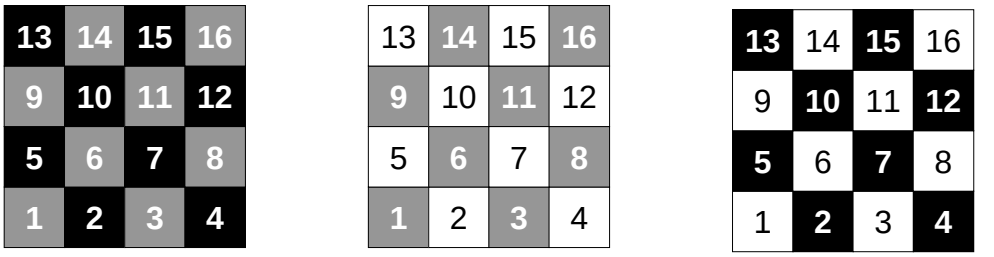
(a) 計算対象　　(b) 4 スレッドによる並列計算　　(c) 前進代入におけるデータ依存性

図 7　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) Red-Black Ordering, Multicolor Ordering

並列計算のためには，データ依存性の除去が必要であり，そのためには，図 7 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(IAL(j,i))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。対策として広く用いられているのは，「グラフ理論」における「彩色，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である〔6〕。図 7 に示した 16 要素の領域は図 8（a）に示すように「2 色」に「彩色」できる。このような彩色手法を，チェッカーボードにちなんで，「red-black coloring」と呼ぶ。「red」に彩色された要素｛1,3,6,8,9,11,14,16｝（図 8 (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」に彩色された残りの要素（図 8（c））についても同様である。



（a）red-black coloring　　（b）red に彩色された要素　　（b）black に彩色された要素

図 8　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。**従って，同じ「色」に属する要素に関する計算は並列に実施**

することができる。

```
      do i= 1, N
        if (COLOR(i).eq.'RED  ') then
          WVAL= z(i)
          do j= 1, INL(i)
            WVAL= WVAL -  AL(j,i) * z(IAL(j,i))
          enddo
          z(i)= WVAL * DD(i)
        endif
      enddo

      do i= 1, N
        if (COLOR(i).eq.'BLACK') then
          WVAL= z(i)
          do j= 1, INL(i)
            WVAL= WVAL -  AL(j,i) * z(IAL(j,i))
          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）

実際の計算では，図 9 に示すように要素を「色」の順に並び替える（ordernig，reordering）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 9 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，「彩色（coloring）」と「並び替え（ordering）」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 multicolor ordering（MC 法）が使用される。ここで示した red-black ordering 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 10 は 4 色による MC 法の適用例である。16/4=4 であるので，各色に 4 要素が割り当てられるものとして，互いに独立な要素を 4 つずつ，初期状態における若い番号の要素から順番に選び出している。MC 法の基本的なアルゴリズムを「アルゴリズム 1」に示す：

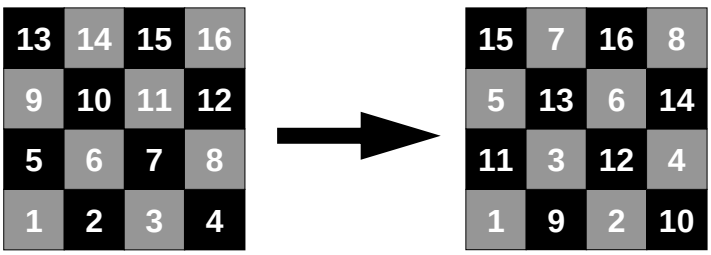


図 9　Red-Black Ordering の例

```
      do icol= 1, 2
        do i= COLORindex(icol-1)+1, COLORindex(icol)
          WVAL= z(i)
          do j= 1, INL(i)
            WVAL= WVAL -  AL(j,i) * z(IAL(j,i))
          enddo
          z(i)= WVAL * DD(i)
        enddo
      enddo
```

リスト 7　前進代入（red-black ordering の導入）

アルゴリズム 1：MC 法の基本的なアルゴリズム

① 「要素数÷色数」を「m」とする。
② 初期要素番号が若い順に互いに独立な要素を「m」個選び出して彩色し，次の色へ進む。
③ 指定した色数に達して，全ての要素が彩色されるまで②を繰り返す。
④ 色番号の若い順番に要素を再番号づけする（各色内では初期要素番号が若い順）。

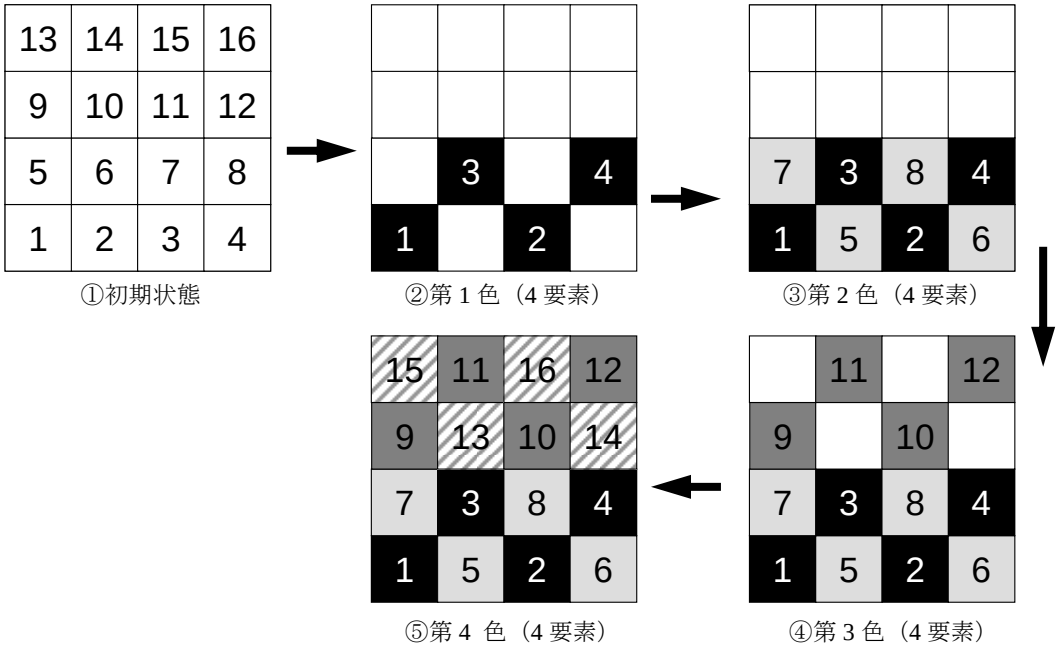


図 10　Multicolor Ordering の例（4 色）

実際は，要素数が色数で割り切れない場合など，様々な例外処理が必要となる。そのような場合の実装については〔6〕を参照されたい。本稿では色数＝2 の場合の MC 法のみを対象とするが，他の手法（CM 法，RCM 法等）については同じく〔6〕を参照されたい。

(2) OpenMP の実装

図 11 に，ordering を実装し，OpenMP を適用したプログラムのサブルーチン構成図を示す。ここでは特に並列ソルバー「SOLVE_ICCG_mc」の「並列化」に主眼を置く。

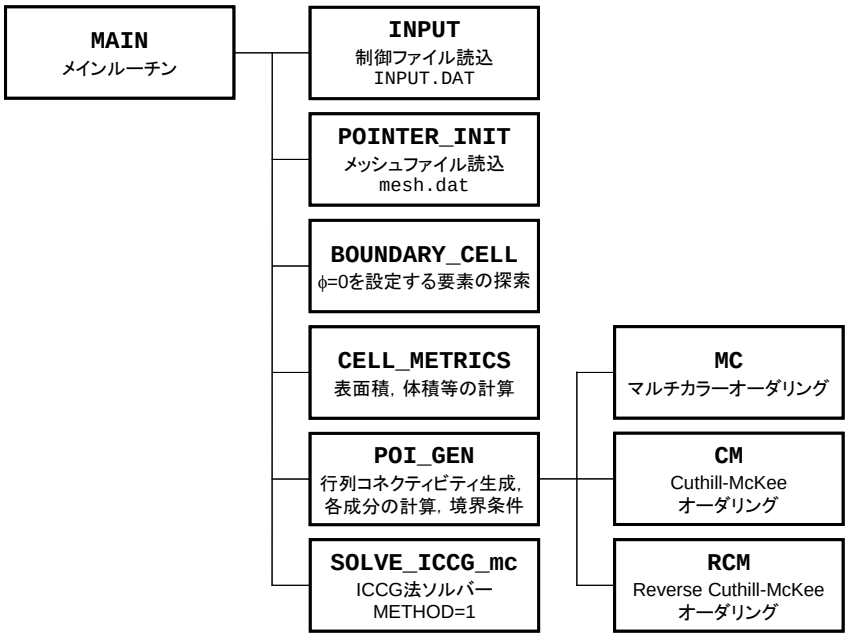


図 11　並列版プログラムのサブルーチン構成図

表 2　新たに追加された変数

変数，配列名	型	内 容
NCOLORtot	整数	入力時には Ordering 手法（≧2：MC，=0：CM，<0：RCM） 最終的には色数，レベル数が入る
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 12 の例では，データ依存性を持たないように 5 色に色づけ，ordering 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，ordering 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。

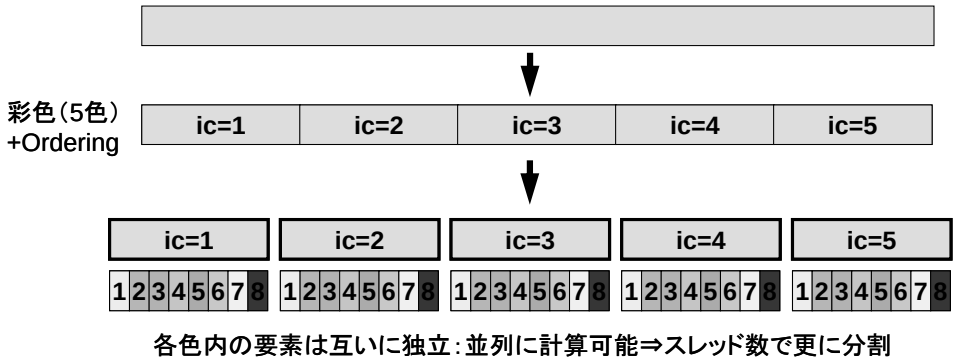


図 12　配列「SMPindex」の基本的な考え方

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

リスト 8　配列「SMPindex」の使用方法

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

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

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

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

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

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。図 13 の例では，全体データを 8 つのスレッドで並列に処理するためデータ分割（8 等分）を実施している。各スレッドへのデータ分割は，ordering 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。

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。

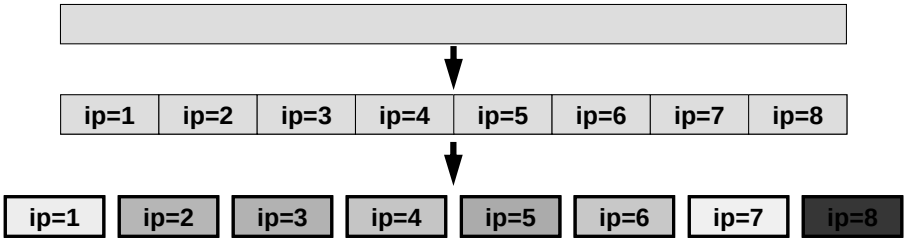


図 13　配列「SMPindexG」の基本的な考え方

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

```
!$omp parallel do private(ip,i,VAL,j)
      do ip= 1, PEsmpTOT
        do i = SMPindexG(ip-1)+1, SMPindexG(ip)
          VAL= D(i)*W(i,P)
          do j= 1, INL(i)
            VAL= VAL + AL(j,i)*W(IAL(j,i),P)
          enddo
          do j= 1, INU(i)
            VAL= VAL + AU(j,i)*W(IAU(j,i),P)
          enddo
          W(i,Q)= VAL
        enddo
      enddo
!$omp end parallel do
```

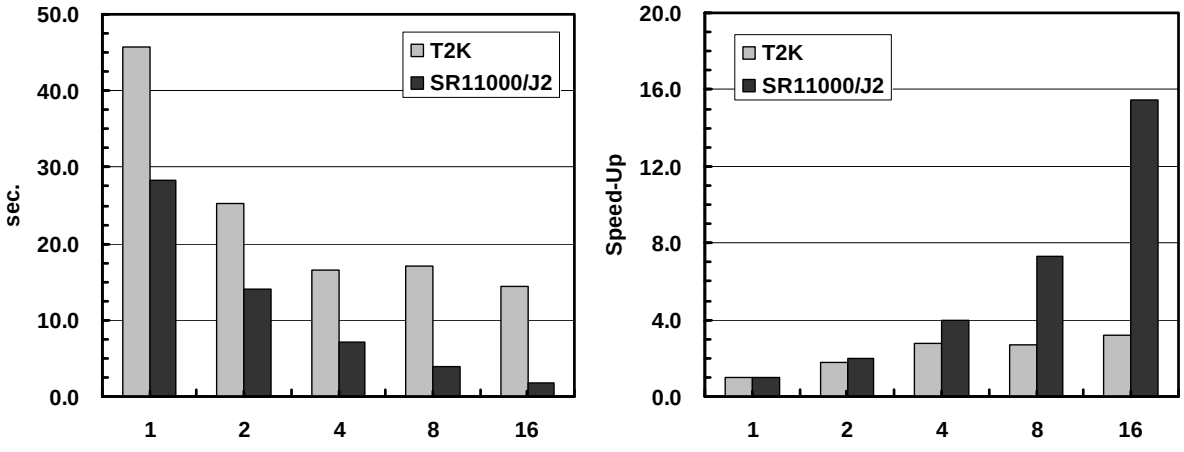
リスト 14　行列ベクトル積（{q}= [A]{p}）のループの OpenMP による並列化

6. 計算結果と最適化

(1) 初期計算結果

図 3 において，NX=NY=NZ=100（10^6 要素），MC 法（色数=2，つまり Red-Black）を適用した場合の計算結果を図 14 に示す。T2K オープンスパコン（東大）1 ノードにおいて OpenMP のスレッド数（すなわちコア数）を 1 から 16 まで変えて実行し，ICCG 法ソルバーの計算時間を，Hitachi SR11000/J2（東京大学情報基盤センター）[3]と比較したものである。

図 15 は，Hitachi SR11000/J2 のノードの構造である。表 3 はノード諸元を T2K オープンスパコン（東大）と比較して示したものである。2 つの IBM POWER5+コア（2.3 GHz，コアあたりピーク性能：9.2 GFLOPS）によって，POWER5+チップが構成されている。4 つのチップ，すなわち 8 つのコアからモジュール（Multi Chip Mudule：MCM）が構成され，2 つの MCM，すなわち 16 個のコアが 1 ノードを形成している〔13〕。T2K（東大）とはコアあたりピーク性能，コア数とも同じため，ノードあたりピーク性能はともに 147.2GFLOPS であるが，メモリバンド幅が 16 コア利用時に 5：1 程度であるため（表 3 参照），実効性能は利用コア数にもよるが 2 倍から 5 倍程度の差がある。図 14（a）の場合では，1 スレッドの場合で約 1.6 倍 SR11000/J2 の方が速い。SR11000/J2 は 16 スレッドでの性能向上比が 15.5 とほぼ完全にスケールしているのに対して，T2K（東大）は 4 スレッド以上では全く性能が向上していない（16 スレッドでの性能向上比が 3.17）。収束（$|r|<1.0^{-8}$）までの反復回数はいずれの場合も 333 である。



（a）計算時間　　（b）性能向上比

図 14　スレッド数と計算時間の関係（10^6 要素，ICCG 法ソルバー部分）

表 3　T2K（東大），Hitachi SR11000/J2 のノード諸元比較

	T2K（東大）	Hitachi SR11000/J2
L1（命令）キャッシュ	64 KB/core	32 KB/core
L1（データ）キャッシュ	64 KB/core	64 KB/core
L2 キャッシュ	512 KB/core	1,875 KB/chip（2 cores）
L3 キャッシュ	2,048 KB/socket（4 cores）	36,000 KB/chip（2 cores）
ピーク性能	147.2 GFLOPS/node	
実測メモリバンド幅[4]	102.4 GB/sec/node	19.6 GB/sec/node

[3] http://ｗｗｗ.cc.u-tokyo.ac.jp/
[4] STREAM ベンチマークによる実測値（http://www.streambench.org/）

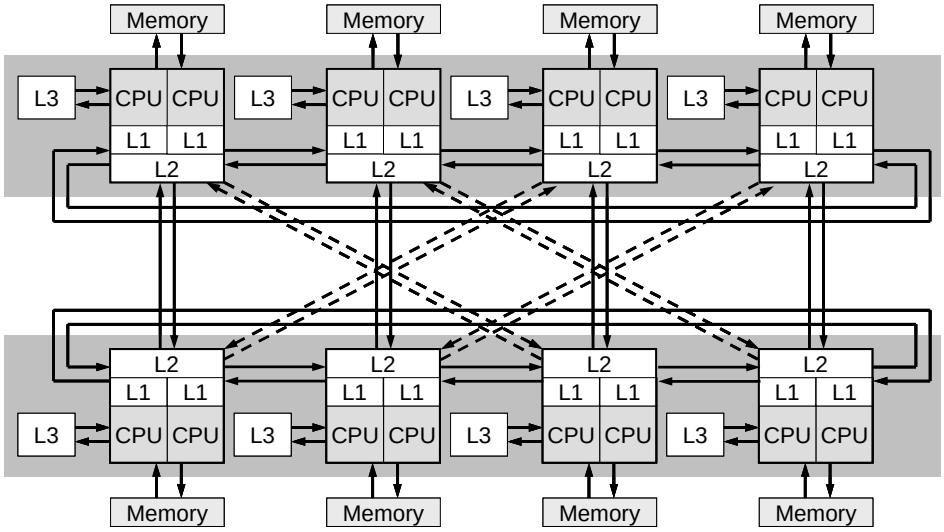


図15　Hitachi SR11000/J2のノード（プロセッサブック）のアーキテクチュア（〔13〕により作成）
点線はHitachi SR11000/J2特有のマルチコアモジュール（Multi Core Module，MCM）間の結線を示す

(2) 性能向上への道

NUMA（Non-Uniform Memory Access）アーキテクチュアでは，各コアができるだけローカルなメモリ上にあるデータをアクセスできるように，データ配置等に配慮したり，NUMA control などの実行時制御〔3，9〕を行う必要がある。図 15 に示すように，Hitachi SR11000/J2 も 2 つの POWER5+コア 2 個から構成される 1 chip がローカルなメモリを有する NUMA アーキテクチュアによっているが〔13〕，表 3 にも示したように，メモリ性能が高く，キャッシュ容量も多いため，遠隔のメモリ上のデータをアクセスしてもそれほど性能は低下しない。しかしながら，T2K オープンスパコン（東大）では NUMA アーキテクチュアの特性を考慮することが必要であると考えられる。

NUMA アーキテクチュアでは，ある変数を最初にアクセスしたコア（の属するソケット）のローカルメモリ上に，その変数の記憶領域が確保される（First Touch Rule〔14〕）ため，配列の初期化手順によって大幅な性能の向上が達成できる場合もある。本稿で扱ったような ordering を伴う場合はメモリへのアクセスパターンがランダムになるため，「First Touch Rule」に基づいて配列を初期化しただけでは不十分で，データの再配置が必要になる場合も考慮する必要があると考えられる。

本稿では以下に示すように，段階的に最適化を実施する。

- ケース 0：初期状態（図 14）
- ケース 1：NUMA control を適用した場合
- ケース 2：First Touch Rule を適用した場合
- ケース 3：ケース 2 に加えてデータの再配置を実施した場合

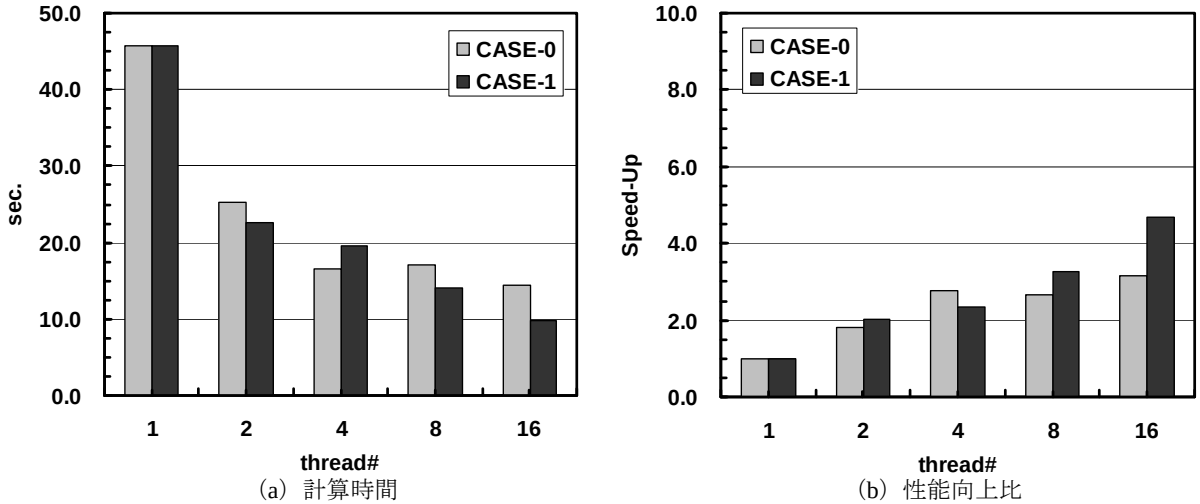
(3) ケース 1（NUMA control の適用）

前回連載〔3〕で紹介された，NUMA control（numactl）を使用して，使用するコア（またはソケット）とメモリを明示的に指定することによって，性能が向上する可能性がある。NUMA control には様々なオプションがあるが，本稿では，Hybrid 並列の場合に特に有効な方法である

以下の組み合わせ（NUMA Policy）を適用した。ここで，$SOCKET はソケット番号（0,1,2,3）である。

```
numactl --cpunodebind=$SOCKET --interleave=$SOCKET
```

計算結果を図 16 に示す。NUMA control を使用することによって若干ではあるが性能が向上していることがわかるが，16 スレッドでも性能向上比は 4.68 程度に留まっている。



（a）計算時間　　（b）性能向上比

図 16　スレッド数と計算時間の関係（10^6 要素，ICCG 法ソルバー部分）

(4) ケース 2 (First Touch Rule の適用)

既に述べたように, NUMA アーキテクチュアではある変数を最初にアクセスしたコアの属するソケットのローカルメモリ上に，その変数の記憶領域が確保されるので，配列の初期化手順によって大幅な性能の向上が達成できる可能性がある。

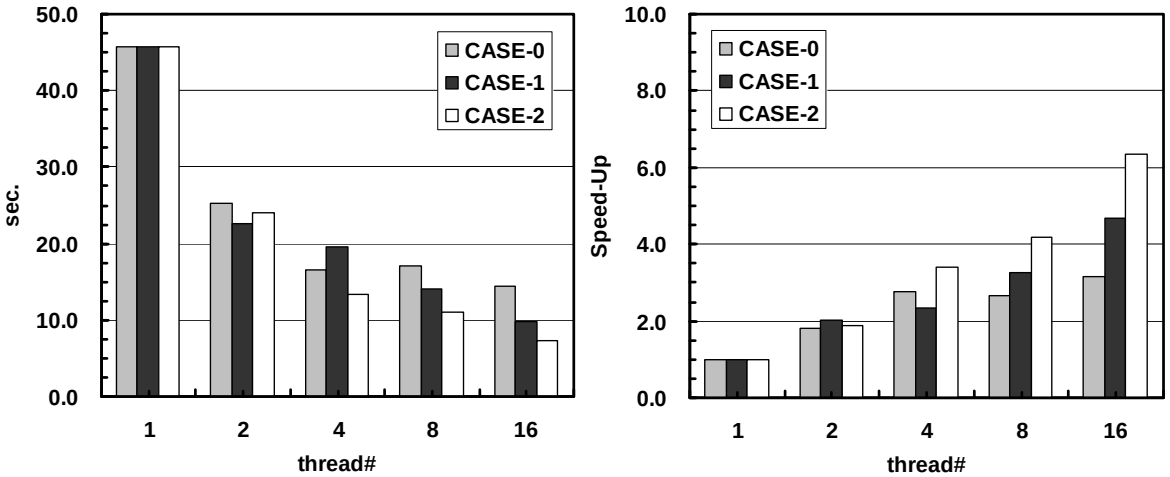
特に，各ベクトルの他，表 1 に示される，行列の係数に関わる配列（INL，INU，IAL，IAU，AL，AU，B）の初期化に関する配慮が必要である。現状の計算ではこれらの配列は以下のような手順で設定されている：

① メッシュデータより「INL，INU，IAL，IAU」を求める
② Ordering によって「INL，INU，IAL，IAU」が入れ替わる
③ 新しい番号付けに従って「AL，AU，D，B」の実数情報を計算する

「INL，INU，IAL，IAU」については，「古い」番号の状態で First Touch が行われているため，Reordering の後の「新しい」番号の状態で First Touch をやり直す必要がある。本稿では，リスト 15 に示すように，もとの配列を「INLorg，INUorg，IALorg，IAUorg」として定義しておき，②の Ordering のプロセスが終了した時点で，新しい番号付けに従って「INL，INU，IAL，IAU」に値を代入している。リスト 15 の代入手順は, リスト 9, 10 に示した計算手順と一致しており，計算に使用されるコアの属するソケットのローカルメモリに各記憶領域が確保される。図 17 に示す計算結果では，ケース 1 に比べて更に性能が向上しており，16 スレッドでの性能向上比は 6.34 である。

```
      do ic= 1, NCOLORtot
!$omp parallel do private (ip,icel,k)
        do ip= 1, PEsmpTOT
          do icel= SMPindex((ic-1)*PEsmpTOT+ip-1)+1,
     &             SMPindex((ic-1)*PEsmpTOT+ip)
            D  (icel)= 0.d0
            PHI(icel)= 0.d0
            B  (icel)= 0.d0
            INL(icel)= INLorg(icel)
            INU(icel)= INUorg(icel)
            do k= 1, NL
              IAL(k,icel)= IALorg(k,icel)
               AL(k,icel)= 0.d0
            enddo
            do k= 1, NU
              IAU(k,icel)= IAUorg(k,icel)
               AU(k,icel)= 0.d0
            enddo
          enddo
        enddo
!$omp end parallel do
      enddo
```

リスト 15　配列の初期化：計算で使用するソケットに属するソケットのローカルメモリ上に記憶領域が確保される



（a）計算時間　　（b）性能向上比

図 17　スレッド数と計算時間の関係（10^6 要素，ICCG 法ソルバー部分）

(5) ケース 3（First Touch Rule＋データ再配置の適用）

前項で述べたように，First Touch Rule を適用することにより，16 スレッドの性能は初期状態（ケース 0）の 2 倍になったが，それでも並列化効率は 50%未満である。現状の手法では図 12 に示すように：

① 同一の色（またはレベル）に属する要素は独立であり，並列に計算可能
② 「色」の順番に番号付け
③ 色内の要素を各スレッドに振り分ける

という方式を採用しているが，同じスレッド（すなわち同じコア）に属する要素は連続の番号では無いため，効率が低下している可能性がある。そこでケース 3 では，同じスレッドで処理するデータをなるべく連続に配置するように更に並び替え，効率の向上を図ることとした。番

号の付け替えによって要素番号の大小関係は変わる可能性があるが，上三角，下三角の関係は変えず，元の計算と反復回数が変わらないようにする。従って自分より要素番号が大きいのにIAL（下三角成分）に含まれているような場合もありうる。

図 18 はこのよう手法を実現するための配列「SMPindex_new」の考え方であり，リスト 16 は「SMPindex⇒SMPindex_new」の並べ替えの実装である。

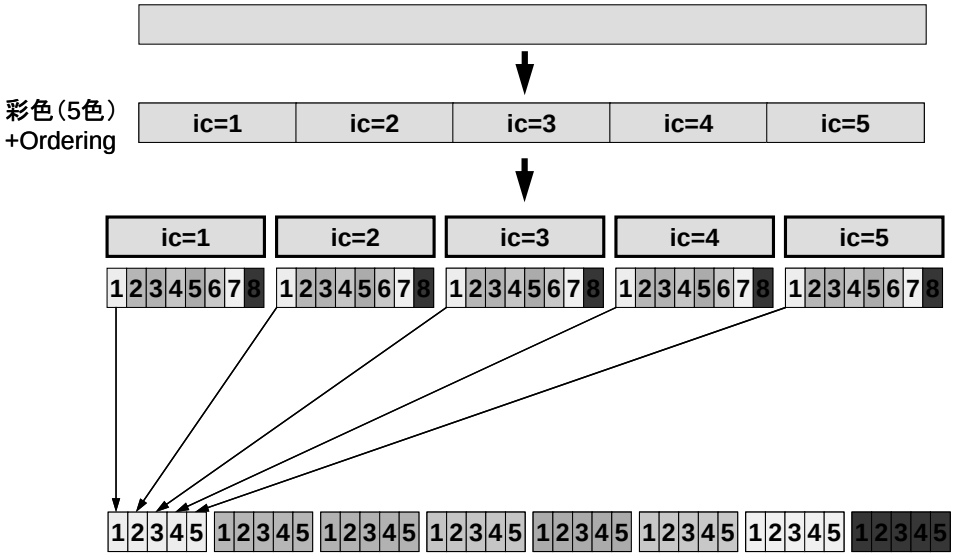


図 18　配列「SMPindex_new」の考え方：各スレッド上の要素は連続した番号付け

```
      allocate (SMPindex(0:PEsmpTOT*NCOLORtot))
      SMPindex= 0
      do ic= 1, NCOLORtot
        nn1= COLORindex(ic) - COLORindex(ic-1)
        num= nn1 / PEsmpTOT
        nr = nn1 - PEsmpTOT*num
        do ip= 1, PEsmpTOT
          if (ip.le.nr) then
            SMPindex((ic-1)*PEsmpTOT+ip)= num + 1
           else
            SMPindex((ic-1)*PEsmpTOT+ip)= num
          endif
        enddo
      enddo

      allocate (SMPindex_new(0:PEsmpTOT*NCOLORtot))
      SMPindex_new(0)= 0
      do ic= 1, NCOLORtot
        do ip= 1, PEsmpTOT
          j1= (ic-1)*PEsmpTOT + ip
          j0= j1 - 1
          SMPindex_new((ip-1)*NCOLORtot+ic)= SMPindex(j1)
          SMPindex(j1)= SMPindex(j0) + SMPindex(j1)
        enddo
      enddo

      do ip= 1, PEsmpTOT
        do ic= 1, NCOLORtot
          j1= (ip-1)*NCOLORtot + ic
          j0= j1 - 1
          SMPindex_new(j1)= SMPindex_new(j0) + SMPindex_new(j1)
        enddo
      enddo
```

リスト 16　「SMPindex⇒SMPindex_new」の変換

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

NEW
      do ic= 1, NCOLORtot
!$omp parallel do private(ip,ip1,i,WVAL,j)
        do ip= 1, PEsmpTOT
          ip1= (ip-1)*NCOLORtot + ic
        do i= SMPindex_new(ip1-1)+1, SMPindex_new(ip1)
          WVAL= W(i,Z)
          do j= 1, INL(i)
            WVAL= WVAL -  AL(j,i) * W(IAL(j,i),Z)
          enddo
          W(i,Z)= WVAL * W(i,DD)
        enddo
        enddo
!$omp end parallel do
      enddo
```

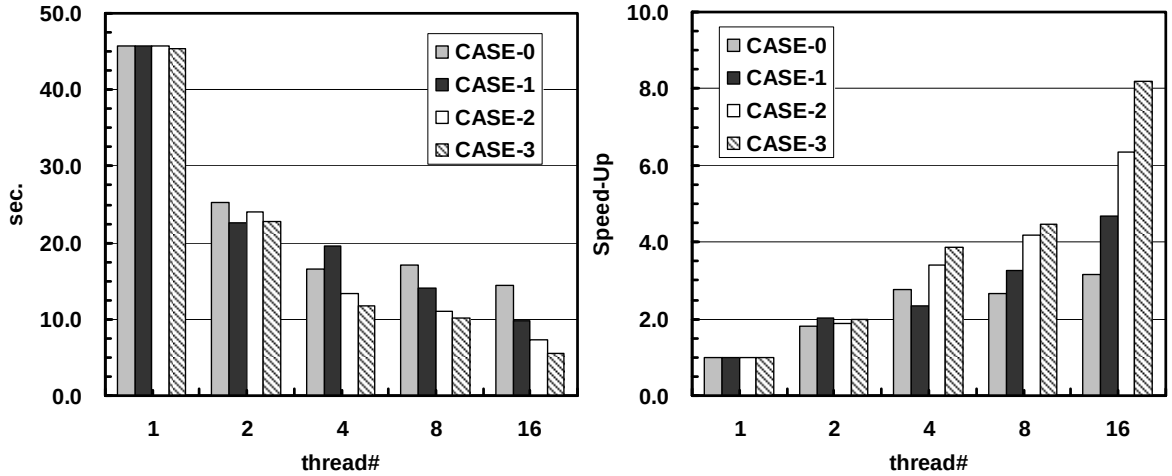
リスト 17　前進代入（[M]{z}={r}）のループの OpenMP による並列化（新旧の比較）

```
ORIGINAL
!$omp parallel do private(ip,i,VAL,j)
      do ip= 1, PEsmpTOT
        do i = SMPindexG(ip-1)+1, SMPindexG(ip)
          VAL= D(i)*W(i,P)
          do j= 1, INL(i)
            VAL= VAL + AL(j,i)*W(IAL(j,i),P)
          enddo
          do j= 1, INU(i)
            VAL= VAL + AU(j,i)*W(IAU(j,i),P)
          enddo
          W(i,Q)= VAL
        enddo
      enddo
!$omp end parallel do

NEW
!$omp parallel do private(ip,i,VAL,j)
      do ip= 1, PEsmpTOT
        do i= SMPindex_new((ip-1)*NCOLORtot)+1, SMPindex_new(ip*NCOLORtot)
          VAL= D(i)*W(i,P)
          do j= 1, INL(i)
            VAL= VAL + AL(j,i)*W(IAL(j,i),P)
          enddo
          do j= 1, INU(i)
            VAL= VAL + AU(j,i)*W(IAU(j,i),P)
          enddo
          W(i,Q)= VAL
        enddo
      enddo
!$omp end parallel do
```

リスト 18　行列ベクトル積（{q}=[A]{p}）のループの OpenMP による並列化（新旧の比較）

リスト 17，18 はそれぞれ，前進代入，行列ベクトル積を「SMPindex_new」を使って実装した場合の適用例である。配列「SMPindexG」は連続な番号付けで無いため，リスト 18 に示したように，行列ベクトル積においても「SMPindex_new」を使わなければならない。図 19 がこのデータ再配置と First Touch Rule を併用した場合の結果である。4 スレッドまではほぼ理想に近い性能向上が見られる。4 スレッドから 8 スレッドでの性能向上は小さいものの，16 スレッドでの性能向上比は約 8.18 である。16 スレッドでの計算時間は 5.55 秒であるが， First Touch を適用しないと 8.08 秒となりケース 2 と比べても遅くなる。

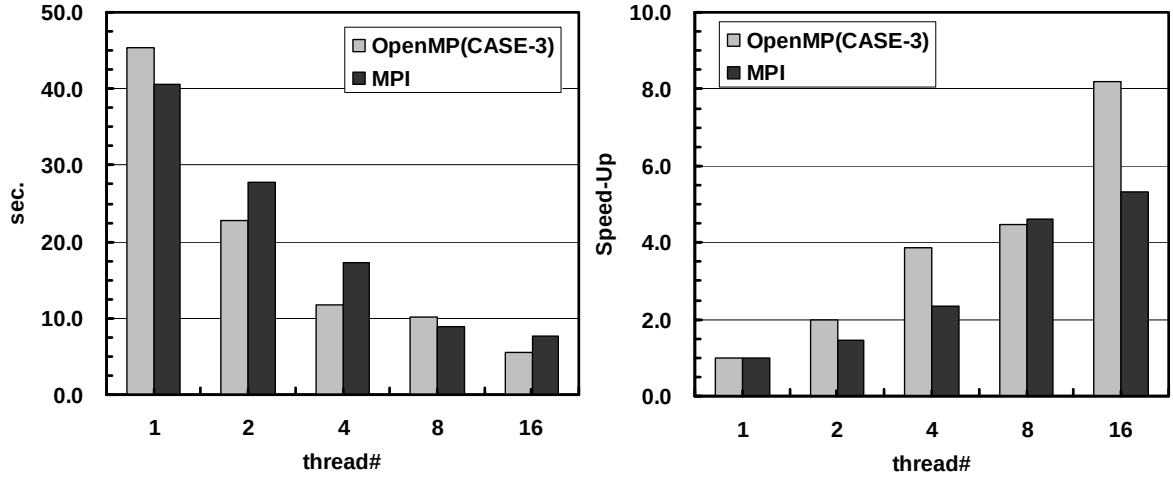


（a）計算時間　　（b）性能向上比

図 19　スレッド数と計算時間の関係（10^6 要素，ICCG 法ソルバー部分）

理想値と比べると大きな隔たりがあるものの，初期状態（ケース 0）と比較すると，大幅な性能向上が見られた。リスト 17，18 からもわかるように右辺の値は必ずしも同じソケット上にあるとは限らないため，そのような変数をアクセスする場合の性能低下は避けられない。

図 20 は OpenMP（ケース 3）と MPI を適用した場合の比較である。MPI の場合は ordering を適用しないオリジナルのコードに対して，ブロック・ヤコビ型の局所前処理〔9〕を適用しているため，領域数が増加すると反復回数が増加する（表 4 参照）。16 コア利用時の性能は OpenMP の方がむしろ良い（OpenMP：5.55 秒，MPI：7.64 秒）



（a）計算時間　　（b）性能向上比

図 20　スレッド数，プロセス数と計算時間の関係（10^6 要素，ICCG 法ソルバー部分）

表 4　収束までの反復回数

スレッド数，プロセス数	OpenMP	MPI
1	333	227
2	333	293
4	333	276
8	333	280
16	333	317

7. Hybrid 並列プログラミングモデルの例

T2K オープンスパコン（東大）における Hybrid 並列プログラミングモデルの適用例として，筆者による研究例〔9〕を一部紹介する。本研究は，図 21 に示すような不均質場（弾性係数の分布が 10^6 のオーダー）における三次元弾性問題を並列有限要素法，前処理付反復法によって解くものであり，反復法としては GPBi-CG 法（Generalized Product-type methods based on Bi-Conjugate Gradient）〔15〕，前処理手法は SGS（Symmetric Gauss-Seidel）〔9〕，領域分割手法としては，HID（Hierarchical Interface Decomposition）〔16〕を適用している。

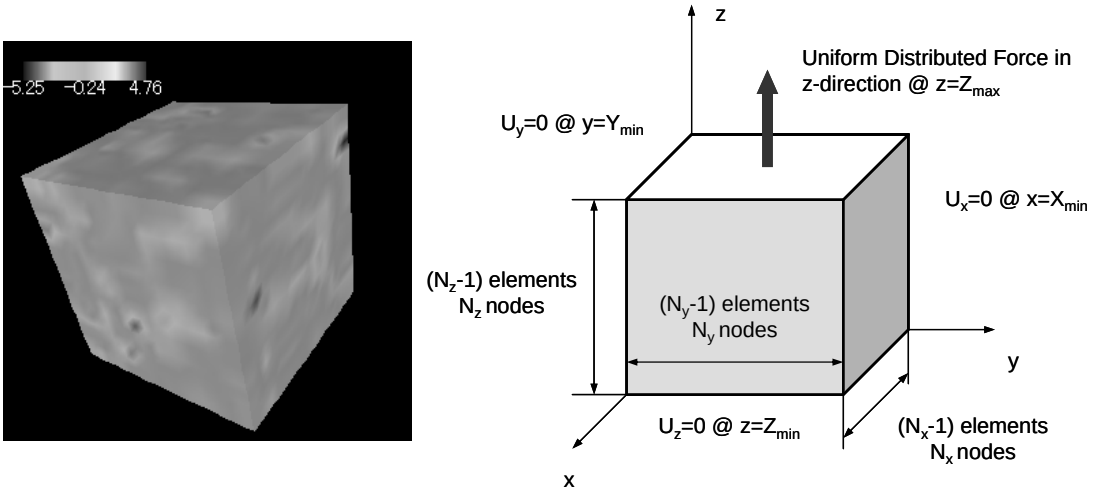


図 21　不均質場における三次元弾性問題

2,097,152 要素，6,440,067 自由度の問題を T2K オープンスパコン（東大）の 2 ノードから 32 ノード（すなわち 32 コアから 512 コア）を使用して計算を実施している。以下に示す 3 つの並列プログラミングモデルの性能を比較している。Hybrid における各プロセス内では CM-RCM（Cyclic Multicoloring + Reverse Cuthill-Mckee）〔9〕による ordering を実施している。

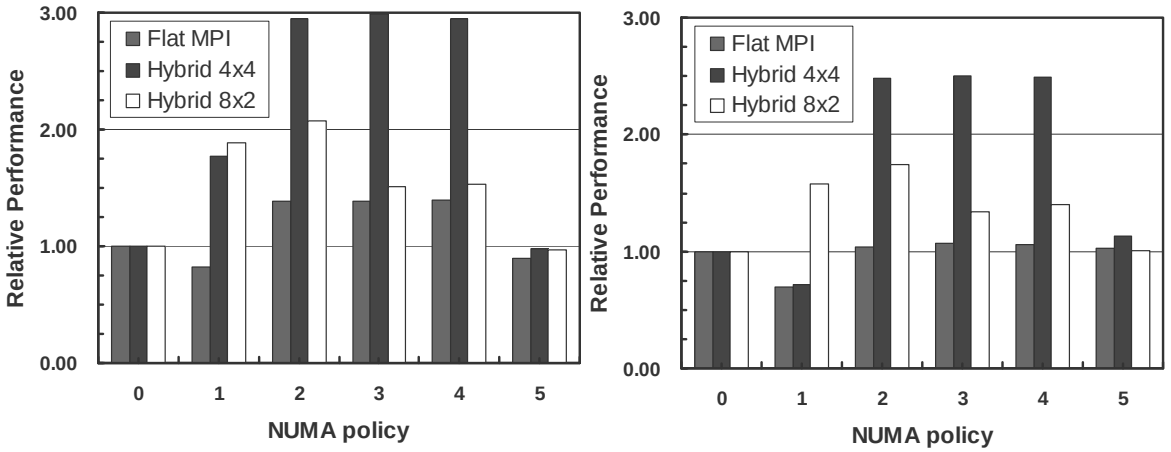
- Flat MPI
- Hybrid 4×4：図 2 の各ソケットに MPI プロセス，MPI プロセス内に 4 スレッド
- Hybrid 8×2：図 2 の 2 ソケットあたり 1MPI プロセス，MPI プロセス内に 8 スレッド

また，表 4 に示すような NUMA Policy を適用しその効果についても検討している。

表 4　適用した NUMA Policy

Policy ID	Command line switches
0	no command line switches
1	`--cpunodebind=$SOCKET --interleave=all`
2	`--cpunodebind=$SOCKET --interleave=$SOCKET`
3	`--cpunodebind=$SOCKET --membind=$SOCKET`
4	`--cpunodebind=$SOCKET --localalloc`
5	`--localalloc`

図 22 は，8 ノード，32 ノードのそれぞれの場合について，各 NUMA Policy の効果について，計算時間に各並列プログラミングモデルの「Policy 0」の場合の計算時間で無次元化した相対効率である。従ってこの相対効率が 1.0 より大きいと「Policy 0」の場合よりも効率が良いということになる。総じて，NUMA Policy の効果は，特にノード数が少ない（従ってノードあたりの粒度が大きい）場合に特に顕著である。また，Hybrid では特に効果が大きい。図 23 は各プログラミングモデルの最良ケースの場合について，Flat MPI の計算時間で無次元化した相対効率である。Hybrid 4×4 は Flat MPI とほぼ同じ性能であることがわかる。



（a）8 ノード（128 コア）　　（b）32 ノード（512 コア）

図 22　NUMA Policy の効果（「Policy 0」の計算効率を 1 としている）

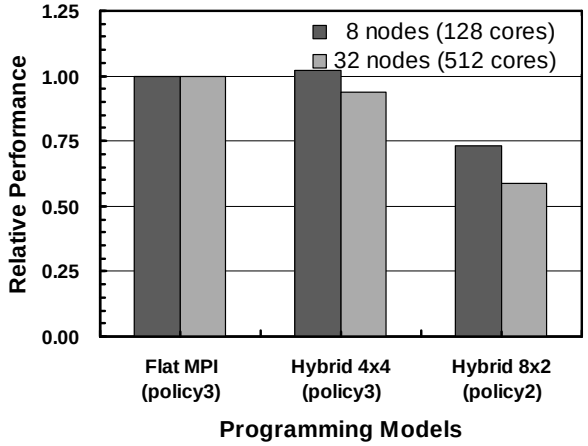


図 23　並列プログラミングモデルの性能比較（Flat MPI の計算効率を 1 としている）

本計算結果における最適化は 6. で述べたケース 1 までの最適化に留まっており，今後ケース 2，ケース 3 に相当する最適化を実施していく必要がある。特に，今後のコア数の増加を考慮すると，Hybrid 8×2，Hybrid 16×1（本研究では実施していない）などの効率向上が重要な課題である。

8. おわりに

有限要素法，有限体積法などの疎行列を係数行列とし，間接参照を伴うアプリケーションについて，OpenMP，Hybrid 並列プログラミングモデルを適用した場合の T2K オープンスパコン（東大）上での並列化の事例について説明した。

NUMA control の適用は有効であるが，First Touch Rule の適用，データの再配置を更に実施することによって，OpenMP，Hybrid において更に効率を向上させることができる。Ordering のプロセスを考慮すると，元のプログラムの変更には多大な労力を必要とする。「次世代」，「次次世代」のスーパーコンピュータでの稼動を目指した，アプリケーションプログラム開発の生産性を向上させるためには本稿で述べたような機能を持つライブラリを T2K オープンスパコン（東大）上で整備することが重要である。

参 考 文 献

[1] 片桐孝洋（2008）T2K オープンスパコン（東大）チューニング連載講座，高性能プログラミング（I）入門編，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）10-4
http://www.cc.u-tokyo.ac.jp:16080/publication/news/VOL10/No4/200807tuning.pdf

[2] 黒田久泰（2008）T2K オープンスパコン（東大）チューニング連載講座，高性能プログラミング（II）上級編，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）10-5
http://www.cc.u-tokyo.ac.jp/publication/news/VOL10/No5/200809tuning.pdf

[3] 吉廣 保（2008）T2K オープンスパコン（東大）チューニング連載講座，実アプリケーションの最適化のテクニック，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）10-6
http://www.cc.u-tokyo.ac.jp/publication/news/VOL10/No6/200811tuning.pdf

[4] Nakajima, K.（2007）The Impact of Parallel Programming Models on the Linear Algebra Performance for Finite Element Simulations, Lecture Notes in Computer Science 4395, 334-348

[5] 中島研吾（2007）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
http://www.cc.u-tokyo.ac.jp/publication/news/VOL9/No5/200709OpenMP.pdf

[6] 中島研吾（2007）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
http://www.cc.u-tokyo.ac.jp/publication/news/VOL9/No6/200711OpenMP.pdf

[7] 中島研吾（2008）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
http://www.cc.u-tokyo.ac.jp/publication/news/VOL10/No1/200801OpenMP.pdf

[8] HA8000 クラスタシステム　利用の手引（東京大学情報基盤センター）
http://www.cc.u-tokyo.ac.jp/ha8000/ha8000-tebiki.pdf

[9] Nakajima, K. （2008）Parallel Multistage Preconditioners by Hierarchical Interface Decomposition on “T2K Open Super Computer　(Todai Combined Cluster)” with Hybrid Parallel Programming Models, Proceedings of the 2008 IEEE International Conference on Cluster Computing (Cluster 2008), 298-303

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

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

[12] Dongarra, J.J. et al.（1994）Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, SIAM（邦訳:長谷川里美, 長谷川秀彦, 藤野清次(1996), 反復法 Templates, 朝倉書店）

[13] 青木秀貴，中村友洋，助川直伸，齋藤拡二，深川正一，中川八穂子，五百木伸洋（2005）スーパーテクニカルサーバーSR11000 モデルJ1のノードアーキテクチュアと性能評価, 情報処理学会論文誌：コンピューティングシステム　45-SIG12（ACS11），27-36

[14] Mattson, T.G., Sanders, B.A. and Massingill, B.L.（2005） Patterns for Parallel Programming, Addison Wesley

[15] Zhang, S.L.（1997）GPBi-CG: Generalized Product-type methods based on Bi-CG for solving nonsymmetric linear systems. SIAM Journal of Scientific Computing 18, 537-551

[16] Henon, P., Saad, Y.（2007）A Parallel Multistage ILU Factorization based on a Hierarchical Graph Decomposition. SIAM Journal for Scientific Computing 28, 2266-2293