OpenMP によるプログラミング入門（Ⅲ）

中 島 研 吾

東京大学大学院理学系研究科地球惑星科学専攻

1. はじめに

前回は，有限体積法によって離散化された三次元領域で，ポアソン方程式をICCG法によって解くプログラム「ICCG-L1」に，MC法（multicolor ordering），CM法（Cuthill-McKee ordering），RCM法（Reverse Cuthill-McKee ordering）を施したプログラム「ICCG-L2」を紹介した。互いに独立な要素を抽出して，並び替えることによって，ICCG法の修正不完全コレスキー分解，前進後退代入などの処理において，効率的にデータ依存性を除去した計算の実行が可能となった。

連載の最終回となる今回は，「ICCG-L2」にOpenMPを導入した「ICCG-L3」作成のための手順を示し，情報基盤センターのHitachi SR11000/J2における並列計算実行例を示す。特にICCG法計算部分の並列化に注目して説明する。前回紹介した「ICCG-L2」の段階で，準備はほとんど完了しており，各ループにOpenMPのディレクティヴを挿入するだけでも「並列化」は可能である。

```
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

リスト1に示す前処理付きCG法に代表される，前処理付きKrylov部分空間型非定常反復解法の主要な計算プロセスのうち：

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

については，リスト2に示すように，OpenMPのディレクティヴを入れるだけで並列化が可能である。また，「ICCG-L2」では，前回紹介したMC法，CM法，RCM法によって，同じ「色」，「レベル」に属する要素が依存関係を持たないように，各要素が色（レベル）づけ，並び替えられている（図2参照）。したがって，ICCG法の前処理に関連した不完全修正コレスキー分解，前進後退

代入のようなプロセスにおいて，同じ「色」,「レベル」に属する要素に対しては，データ依存性が生じないため，並列処理が可能である。すなわち，リスト3，リスト4に示すように，前回「ICCG-L2」で示したプログラムに直接ディレクティヴを挿入すれば「並列化」は可能である。

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}

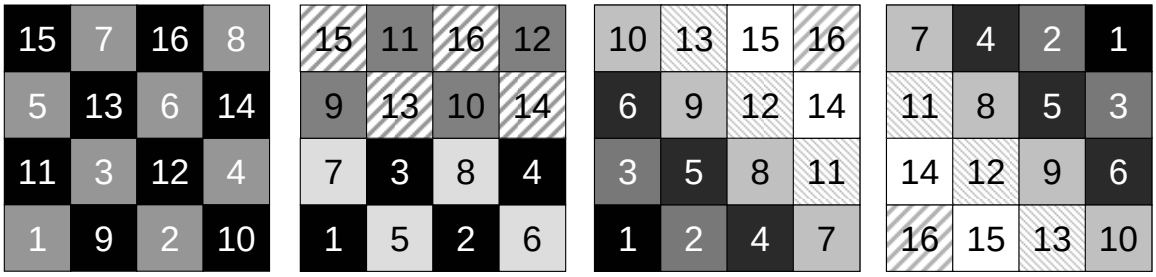
```
do i= 1, N
  VAL= D(i)*p(i)
  do j= 1, INL(i)
    VAL= VAL + AL(j,i)*p(IAL(j,i))
  enddo
  do j= 1, INU(i)
    VAL= VAL + AU(j,i)*p(IAU(j,i))
  enddo
  q(i)= VAL
enddo
```

```
  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 2　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



(a) MC 法（2 色：Red-Black）　(b) MC 法（4 色）　(c) CM 法（7 レベル）　(d) RCM 法（7 レベル）

図 1　各種手法による ordering

```
      do icol= 1, NCOLORtot
!$omp parallel do private (i, VAL, j)
        do i= COLORindex(icol-1)+1, COLORindex(icol)
          VAL= D(i)
          do j= 1, INL(i)
            VAL= VAL - (AL(j,i)**2) * DD(IAL(j,i))
          enddo
          DD(i)= 1.d0/VAL
        enddo
!$omp end parallel do
      Enddo
```

リスト 3　不完全修正コレスキー分解の OpenMP による並列化

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

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

リスト 4　前進後退代入プロセスの OpenMP による並列化

今回は，並列処理のプロセスをより深く理解するために，プログラム内部でスレッド数を調節可能なように「ICCG-L2」を書き換えて，OpenMP を適用してみよう。

OpenMP を使用して効率的な並列計算を実施するためには，各スレッドの負荷ができるだけ均等になることが望ましい。今回紹介する方法はその条件を陽的に満たすことが可能である。

2. プログラムのダウンロード，コンパイル，実行

今回使用するプログラム，関連ファイルは下記ホームページよりダウンロードできる：

http://www-solid.eps.s.u-tokyo.ac.jp/~nakajima/tutorial/sr11k/

本連載用のディレクトリ（<$tutorial-top>）へ，関連 TAR ファイル（Lesson-3-C.tar, Lesson-3-F.tar）をコピー，解凍すると，以下のディレクトリが得られる：

```
<$tutorial-top>/Lesson-3/src ⇒ <$L3-src>
<$tutorial-top>/Lesson-3/run ⇒ <$L3-run>
```

以下のように，<$L3-run>/L3-sol，という実行形式が生成されていることを確認する：

```
$> cd <$L3-src>
$> make
$> ls ../run/L3-sol
```

ここで生成される<$L3-run>/L3-sol は，「ICCG-L2」をプログラム内部でスレッド数を調節可能なように書き換えて，OpenMP を適用した「ICCG-L3」の実行形式である。MPI の時間測定関数（MPI_WTIME）を使用するために MPI（mpif90，mpicc）を使ってコンパイルしている（<L3-src>/Makefile 参照）。コンパイルオプションは以下を使用している。詳しくは文

献〔1〕を参照されたい：

- FORTRAN　　-Oss -omp -64 –looptiling
- C　　-Os +Op -omp -64 -looptiling

計算実施にあたっては，要素データ「mesh.dat」と制御データ「INPUT.DAT」が必要である。要素生成のためには，「ICCG-L1」，「ICCG-L2」のときと同様に，<$L3-run>において予めコンパイル，リンクして要素生成プログラム「mg」を作成しておく。

```
$> cd <$L3-run>
$> f90 –O mg.f –o mg  または cc –O mg.c –o mg
```

要素生成のためには：

```
$> cd <$L3-run>
$> ./mg
```

とすると入力待ちの状態になるので，例えば NX=20，NX=20，NX=20 を入力して，要素数 8,000 の「mesh.dat」が生成される。

計算実行のためには，制御データ「INPUT.DAT」，シェルスクリプトを書き換えて，バッチ処理を実施する。<$L3-run>にある制御データ「INPUT.DAT」の内容は自由形式で表 1 のようになっている：

表 1　制御データ「INPUT.DAT」の内容

変数名	型	内　容
DX，DY，DZ	倍精度実数	各要素の 3 辺の長さ（ΔX，ΔY，ΔZ）
OMEGA	倍精度実数	不使用（適当な値を入れておく）
EPSICCG	倍精度実数	収束判定値
PEsmpTOT	整数	データ分割数
NCOLORtot	整数	Ordering 手法と色数 ≧2：MC 法（multicolor ordering），色数 =0：CM 法（Cuthill-Mckee ordering） =-1：RCM 法（Reverse Cuthill-Mckee ordering）

初期状態では DX=DY=DZ=1.0，EPSICCG=10^{-8} となっているが，以下この値をそのまま使用する。また，リスト 5 に示すようなシェルスクリプトのサンプル（<$L3-run>/go.sh）が用意されている。各パラメータの許容値は利用者のクラス，環境に依存するので，詳細は情報基盤センター（スーパーコンピューティング部門）ホームページ[1]を参照されたい。

[1] http://www.cc.u-tokyo.ac.jp/

```
#@$-r cube
#@$-q lecture4
#@$-N 1
#@$-e err
#@$-o 100-1000.lst
#@$-lM 55GB
#@$-lE 00:10:00
#@$-s /bin/sh
#@$

cd /batch/t12345/tutorial/Lesson-3/run

/usr/mpi/bin/mpirun ./L3-sol
exit
```

リスト５　「ICCG-L3」実行のためのシェルスクリプトの例

「ICCG-L2」では，色数を画面から入力していたが，「ICCG-L3」はバッチ処理のため，「INPUT.DAT」から値を読み込むようになっている。また「データ分割数（PEsmpTOT)」も「INPUT.DAT」で記入できるようになっている。「PEsmpTOT」は，基本的に OpenMP のスレッド数に相当し，実行時の利用可能コア数（情報基盤センターの Hitachi SR11000/J2 では 8 または 16 がデフォルト値）と同じかそれ以下であることを想定している。利用可能コア数（スレッド数）が 8 でも「PEsmpTOT=4」であれば，データが 4 分割されるため，8 スレッドのうち 4 スレッドは受け持つデータ量がゼロとなり，残りの 4 スレッドのみが計算を実施することになる。したがって，理想的な状況を考えると，PEsmpTOT=4 の場合は PEsmpTOT=8 の場合の 2 倍の計算時間を要することになる。

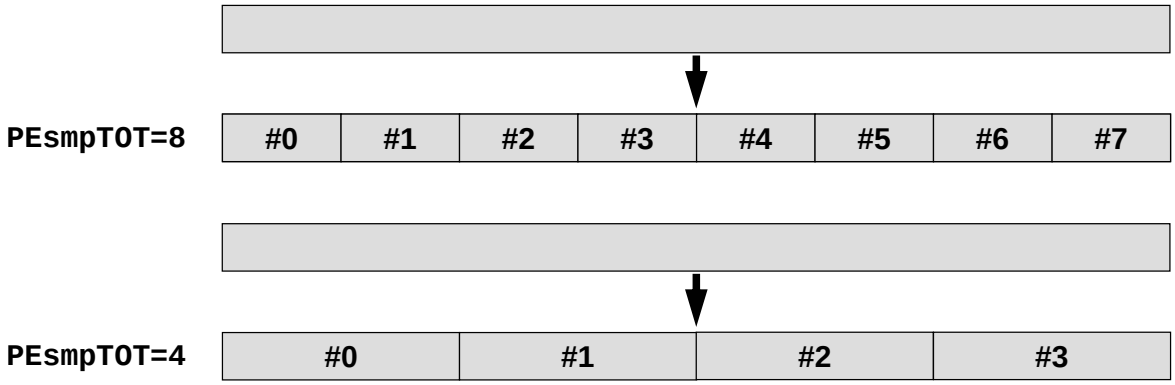


図 2　PEsmpTOT と各スレッドの受け持ちデータ量

これとは別に実行時の利用コア数（デフォルトでは 8 または 16）を変更する場合には，環境変数「OMP_NUM_THREADS」をシェルスクリプト内で陽に指定すれば良い。詳しくは前掲のホームページを参照されたい。**本稿では実行時利用コア数のデフォルト値を「8」とする。**

3. プログラムの処理内容

「ICCG-L3」では前処理手法としては,「ICCG-L1」における「METHOD=1」の場合，つまり，式（1）に示す，不完全修正コレスキー分解において，$\tilde{l}_{ij} = a_{ij}$ とした場合のみを考慮している。

図 3 にサブルーチン構成図を示す。サブルーチン構成は「ICCG-L2」の場合と同じである。ここでは特に「SOLVE_ICCG_mc」の「並列化」に主眼を置く。

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

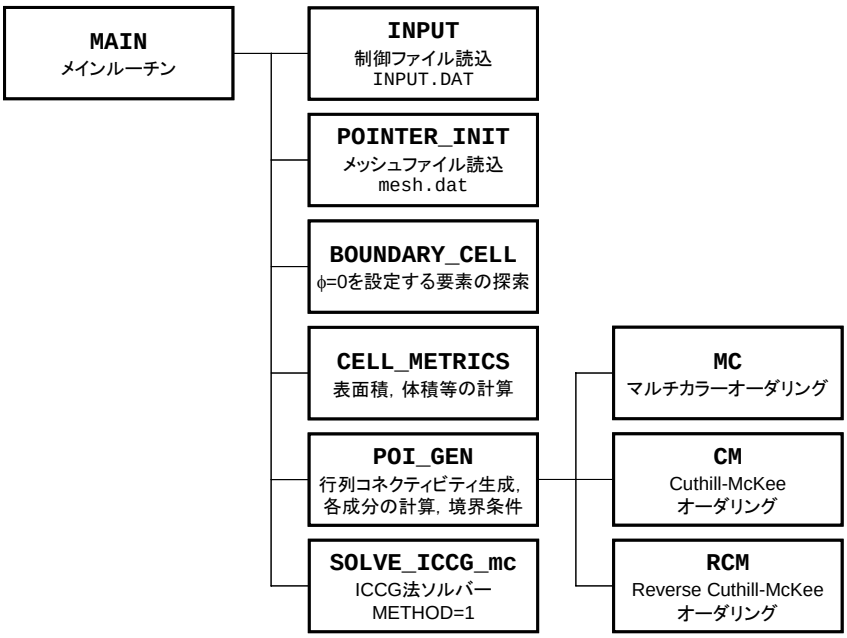


図 3　サブルーチン構成図（ICCG-L3）

表 2　　変数表

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
ICELTOT	整数	要素数（＝16）
NL	整数	各要素の下三角成分の最大数（=6 に設定してある）
NU	整数	各要素の上三角成分の最大数（=6 に設定してある）
INL(i)	整数	要素 *i* の下三角成分数（<NL），i=1~ICELTOT
INU(i)	整数	要素 *i* の上三角成分数（<NU），i=1~ICELTOT
IAL(j,i)	整数	要素 *i* の *j* 番目の下三角成分に対応する列番号，i=1~ICELTOT，j=1~INL(i)
IAU(j,i)	整数	要素 *i* の *j* 番目の上三角成分に対応する列番号，i=1~ICELTOT，j=1~INU(i)
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**	**データ分割数**
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**

変数名と内容を表 2 に示す。表 2 で太字で示した「PEsmpTOT」,「SMPindex」,「SMPindexG」の 3 種類の変数，配列が「ICCG-L3」で新たに加わっている。

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 4 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。「SMPindex」は，「ICCG-L2」と同様の「COLORindex」（表 2 参照）に基づき，「POI_GEN」の中でリスト 6 に示すように定義される。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。

リスト 7 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 8，リスト 9 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 3，リスト 4 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。

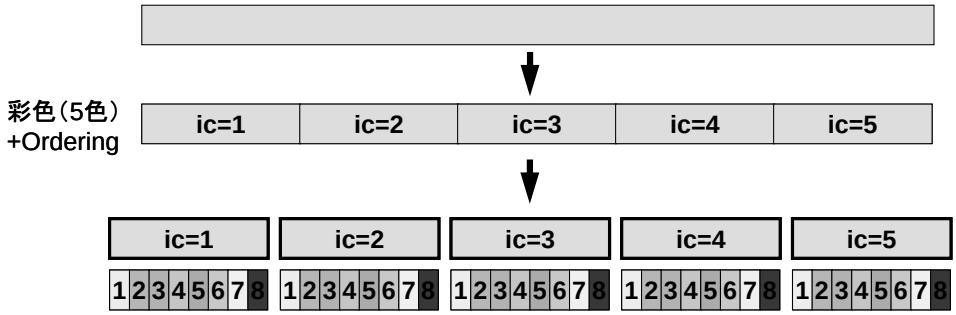


図 4　配列「SMPindex」の基本的な考え方

```
      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

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

リスト 6　配列「SMPindex」の定義

```
      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 7　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 8　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 9　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 5 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。「SMPindexG」は，「POI_GEN」の中でリスト 10 に示すように定義される。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 2 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。

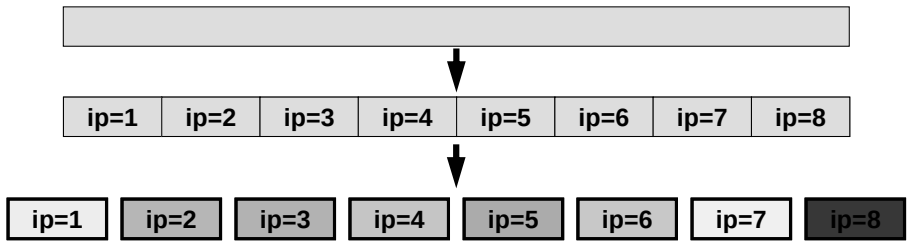


図 5　配列「SMPindexG」の基本的な考え方

```
      allocate (SMPindexG(0:PEsmpTOT))
      SMPindexG= 0
      nn= ICELTOT / PEsmpTOT
      nr= ICELTOT - nn*PEsmpTOT
      do ip= 1, PEsmpTOT
        SMPindexG(ip)= nn
        if (ip.le.nr) SMPindexG(ip)= nn + 1
      enddo

      do ip= 1, PEsmpTOT
        SMPindexG(ip)= SMPindexG(ip-1) + SMPindexG(ip)
      Enddo
```

リスト 10　配列「SMPindexG」の定義

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

リスト 15，リスト 16 は，それぞれ「ICCG-L2」，「ICCG-L3」におけるサブルーチン「solve_ICCG_mc」呼び出しのためのインタフェースである。引数に注目すると，「ICCG-L3」では「COLORindex」が使用されておらず，その代わりに表 2 に示したように「ICCG-L3」で新たに加わった変数（PEsmpTOT，SMPindex，SMPindexG）が使用されている。

```
      subroutine solve_ICCG_mc                                          &
     &         ( N, NL, NU, INL, IAL, INU, IAU, D, B, X,                &
     &           AL, AU, NCOLORtot, COLORindex, EPS, ITR, IER)
```

リスト 15　「ICCG-L2」におけるサブルーチン「solve_ICCG_mc」の呼び出し

```
      subroutine solve_ICCG_mc                                          &
     &         ( N, NL, NU, INL, IAL, INU, IAU, D, B, X,                &
     &           AL, AU, NCOLORtot, PEsmpTOT, SMPindex, SMPindexG,      &
     &           EPS, ITR, IER)
```

リスト 16　「ICCG-L3」におけるサブルーチン「solve_ICCG_mc」の呼び出し

ここまでは，主として，「solve_ICCG_mc」の並列化について述べて来たが，図 2 に示した各サブルーチンのうち「solve_ICCG_mc」に次いで計算量の多い「POI_GEN」についても，配列「SMPindexG」を使用すればリスト 17 に示すように OpenMP を使用した並列化が可能である。ここでは，後半のマトリクス成分の計算部分のループについてのみ並列化したが，前半部分（MC 法，CM 法，RCM 法などの適用前）の，コネクティビティ探索部についても同様に並列化は可能である。各自試みられると良い。

OpenMP を使用した並列化の場合，計算を正確に実行するためには，各変数が private か shared であるかを充分に注意する必要がある。特にリスト 17 に示したようなループ内の計算量が多い場合，様々な変数が使用されているため，特に注意が必要である。private と shared の属性を間違えると正しい解が得られない場合がある。

```
!C
!C +-----------------------------------+
!C | INTERIOR & NEUMANN BOUNDARY CELLs |
!C +-----------------------------------+
!C===

!$omp parallel do private (ip,icel,ic0,icN1,icN2,icN3,icN4,icN5,icN6)  &
!$omp&            private (VOL0,coef,j,ii,jj,kk)
      do ip  = 1, PEsmpTOT
      do icel= SMPindexG(ip-1)+1, SMPindexG(ip)
        ic0 = NEWtoOLD(icel)

        icN1= NEIBcell(ic0,1)
        icN2= NEIBcell(ic0,2)
        icN3= NEIBcell(ic0,3)
        icN4= NEIBcell(ic0,4)
        icN5= NEIBcell(ic0,5)
        icN6= NEIBcell(ic0,6)

        VOL0= VOLCEL  (ic0)

 （以下省略）
```

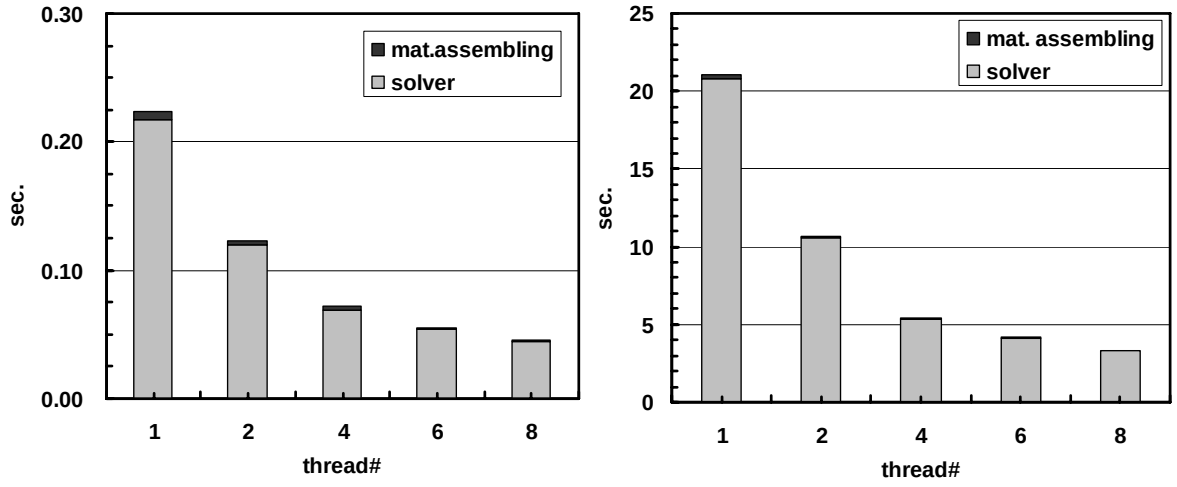
リスト 17　係数マトリクス成分計算ループの OpenMP による並列化

4. 計算結果

2.，3. で示したプログラム「ICCG-L3」を使用した計算結果を示す。以下に示す計算例ではノードあたり使用可能なコア数は 8 に固定してある。「INPUT.DAT」で定義する「PEsmpTOT」の値を 8 以下とすれば，図 2 に示すように有効なスレッド数と等しくなるものとする。以下，「PEsmpTOT」と「スレッド数」は等しいものとみなす。

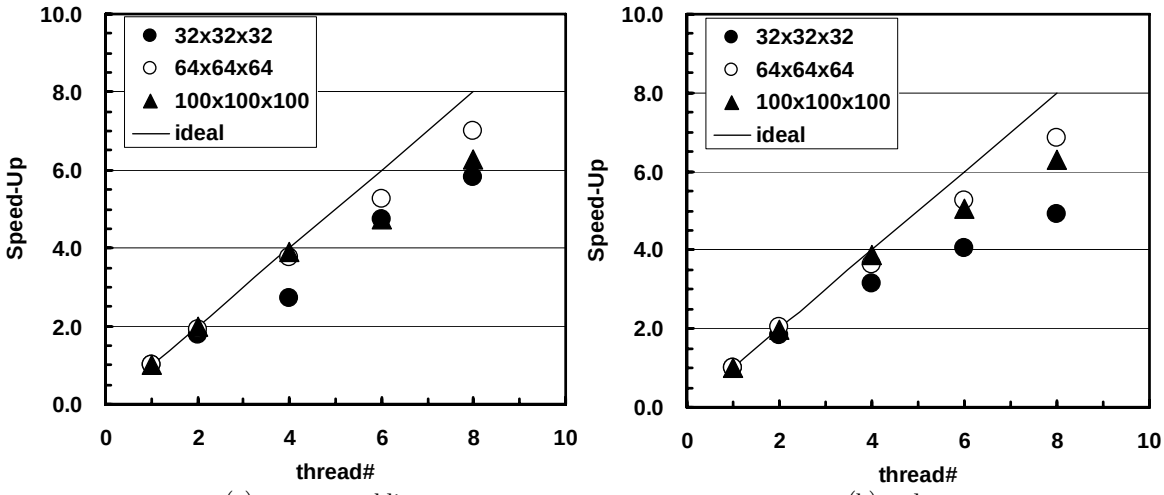
(1) スレッド数の効果

まず,問題規模を固定して,スレッド数(PEsmpTOT)を変化させた場合について示す。Ordering 手法としては CM 法を使用した。NX=NY=NZ=32, 64, 100 の場合, すなわち全要素数が 32,768, 262,144, 1,000,000 の場合について計算を実施した。図 6 は NX=NY=NZ=32 および 100 の場合の計算時間である。「solver」は「SOLVE_ICCG_mc」の,「mat. assembling」は「POI_GEN」のうちリスト 17 に示した係数マトリクス成分計算ループの計算時間である。



(a) NX=NY=NZ=32, 全要素数=32,768　　(b) NX=NY=NZ=100, 全要素数=1,000,000

図6　スレッド数と計算時間の関係



（a）mat. assembling　　（b）solver

図7　スレッド数を変化させた場合の性能向上比（1スレッドを基準）

図7はNX=NY=NZ=32，64，100の場合について，「solver」，「mat. assemble」の各部分のスレッド数増加による性能向上比を`PEsmpTOT=1`の場合の性能を1として計算時間から求めたものである。

スレッド数増加によって，性能は向上しており，理想値に近い性能向上比が得られている。特に，各要素において独立に計算が可能である「mat. assembling」の部分はこの傾向が顕著である。計算時間のほとんどを占めるICCGソルバー部分（solver）については，リスト8，9，12，13，14などに示したように，頻繁にOpenMPの同期を取る必要があり，オーバーヘッドの影響が顕著であるため，「mat. assembling」と比較して性能向上比が低い。しかしながら，ある程度問題規模が大きくなると，8スレッドを使用した場合の性能向上比は6.0を超える値となる（図7（b）参照）。

一般に問題規模が大きくなると，スレッド数を増加させた場合の性能向上比は理想値に近づくが，本問題の場合は必ずしもそうなっていない。原因については明らかでは無いが，メモリバンド幅，キャッシュ容量，同期オーバーヘッド等，様々な要因に起因しているものと考えられ，更なる検討が必要である。

（2）色数，レベル数の効果

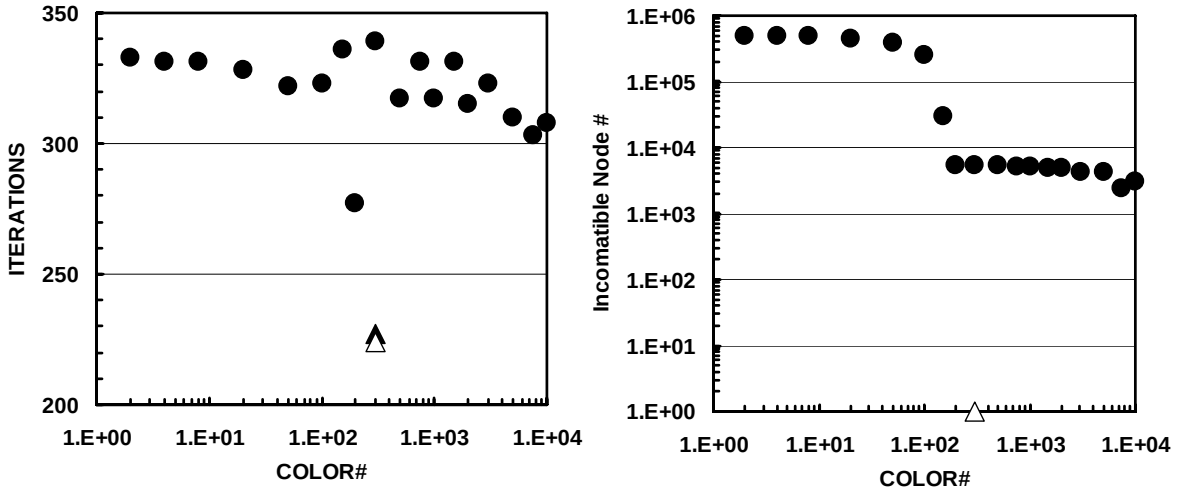
続いてNX=NY=NZ=100の場合の色数（レベル数）と反復回数の関係について検討する。前回はNX=NY=NZ=20の場合に「ICCG-L1」，「ICCG-L2」により同様の検討を実施した〔2〕。今回もMC法では色数を増加させるほど，反復回数が減少する傾向が見られた（図8（a）参照）。図8（b）は色数と「Incompatible Nodes（以下ICN）」〔3〕の関係であり，色数，ICN数，反復回数の間の相関が見られるが，図9に示したNX=NY=NZ=20の場合〔2〕と比較して，色数の反復回数減少に対する効果は顕著ではない。図8（a）に示すように，CM法，RCM法（レベル数=298）の場合の反復回数はMC法より少なく，反復回数は，それぞれ227回，224回である。

図10は色数（レベル数）と計算時間（`SOLVE_ICCG_mc`）の関係である。色数（レベル数）が増加すると1反復あたりの計算時間が増加する傾向が見られる（図10（a）参照）。特にMC法の場合，図10（b）に示すように，反復回数が減少傾向にも関わらず，全体としての計算時間は増加の傾向にある。これはリスト9に示した前進後退代入処理におけるOpenMPのオーバーヘッドによるものと考えられる〔4〕。前進後退代入処理はICCG法において全計算時間の50%程度を占めるプロセスである〔4〕。データ依存性を除去し，並列計算を適切に実施するためには，下記のよう

に色（レベル）が変わるときにOpenMPの同期をとる必要がある（リスト7参照）：

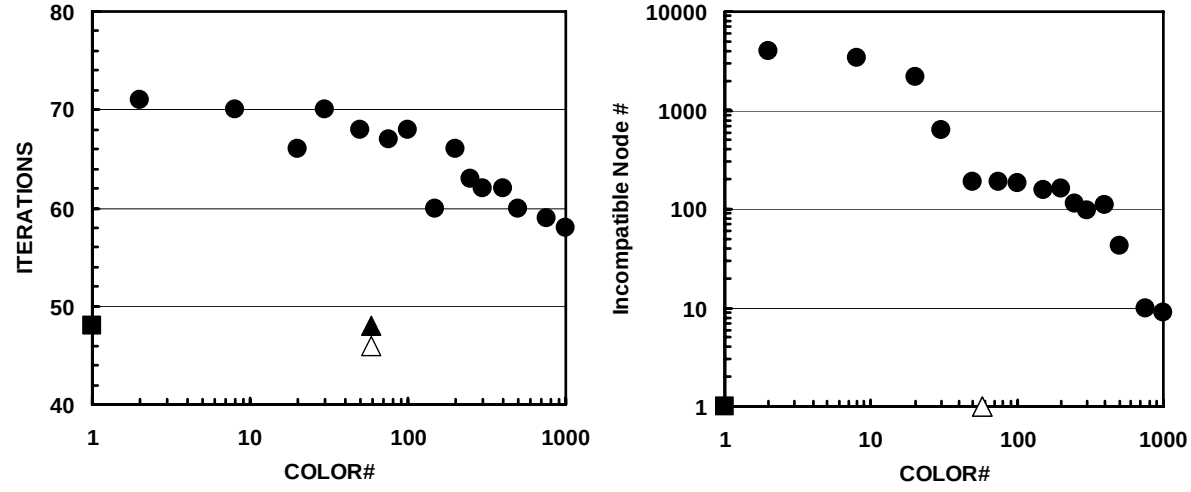
```
      do ic= 1, NCOLORtot
$!omp parallel do ...
        do ip= 1, PEsmpTOT
          (...)
        enddo
!$omp end parallel do
      enddo
```

色数（レベル数）が増加すると，この同期のオーバーヘッドの効果が大きくなり，計算性能が低下すると考えられている〔4〕。OpenMPによる並列化を実施する場合，色数（レベル数），反復回数，計算性能のトレードオフについては充分に留意する必要がある。本ケースの場合，CM法，RCM法はMC法の色数が少ない場合と比較して1反復あたりの計算性能は低いが，反復回数の少なさの効果が大きく，計算時間は短い。図10（a）でCM法，RCM法（レベル数=298）の計算性能が色数=300の場合のMC法と比較して性能が低いのは，CM法，RCM法では図1に示すように要素数の少ない「レベル」が存在することに起因する可能性がある〔4〕。



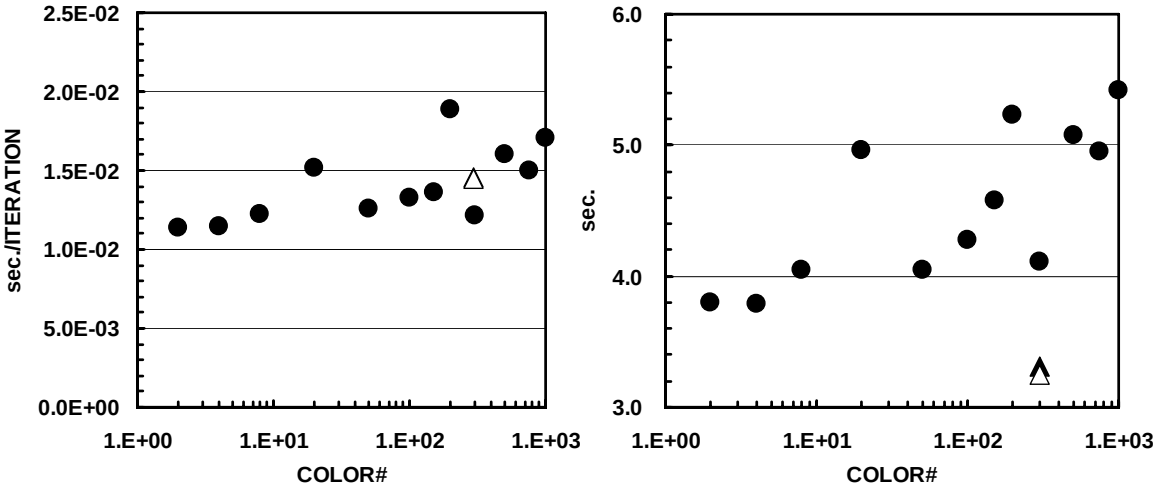
（a）ICCG法の反復回数　　（b）Incompatible Nodeの数

図8　色数，レベル数の計算結果への影響（NX=NY=NZ=100，10^6要素）
（●：ICCG-L3/MC，▲：ICCG-L3/CM，△：ICCG-L3/RCM）



（a）ICCG法の反復回数　　（b）Incompatible Nodeの数

図9　色数，レベル数の計算結果への影響（NX=NY=NZ=20，8,000要素）〔2〕
（■：ICCG-L1，●：ICCG-L2/MC，▲：ICCG-L2/CM，△：ICCG-L2/RCM）



（a）1反復あたり計算時間　　（b）全反復の計算時間

図10　色数，レベル数の計算時間（SOLVE_ICCG_mc）への影響（NX=NY=NZ=100，10^6要素）
（●：ICCG-L3/MC，▲：ICCG-L3/CM，△：ICCG-L3/RCM）

5. 有限要素法による三次元弾性問題における計算結果

もう一つの計算例として，有限要素法による三次元弾性解析プログラムの Hitachi SR11000/J2 1 ノード（8 コア）での実行結果を紹介する。計算対象，境界条件を図 11 に示す。一辺長さ=1，ヤング率=1，ポアソン比=0.30 の立方体要素（一次補間関数）から構成されている。ここでは，X，Y，Z 方向に各 99 要素（すなわち 100 節点）の場合を計算しており，総節点数は 10^6，総未知数（自由度数）は 3×10^6 である。連立一次方程式の解法としては ICCG 法を使用しており，本連載で説明してきたのと同様の ordering によるデータ依存性の除去を行なって，OpenMP を使用した並列化を実施している。詳細は文献〔4〕を参照されたい。

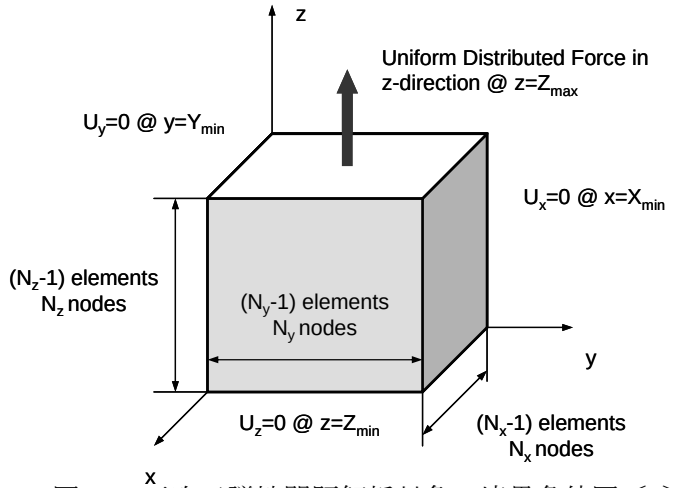
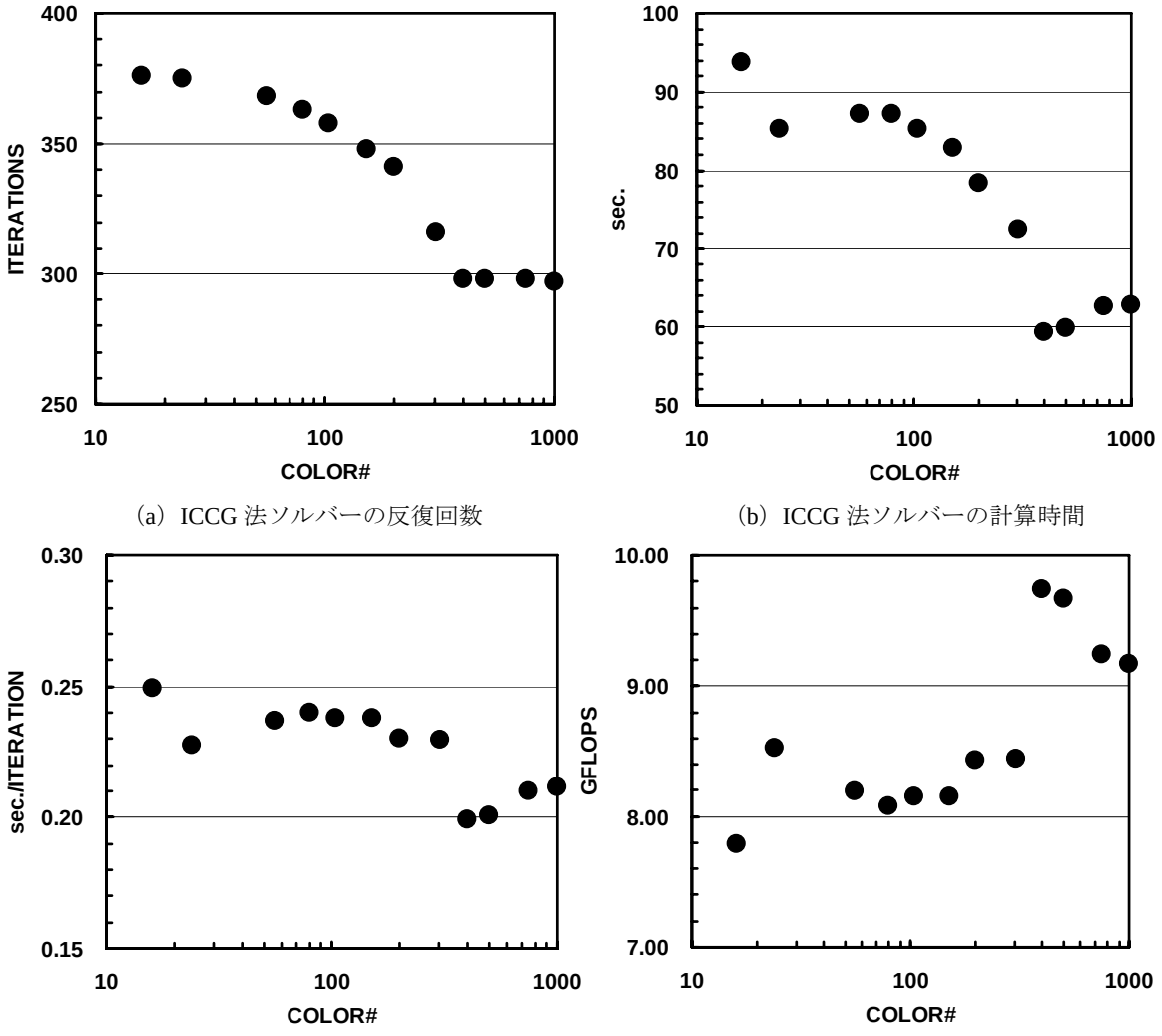


図 10　三次元弾性問題解析対象，境界条件図〔4〕

図 11 に色数と収束，計算性能の関係を示す。ここでは MC 法を適用しているが，色数増加とともに反復回数は減少しており，また計算時間（ICCG ソルバーの演算時間）も同様に色数とともに減少しているが，色数が 500 を超えると計算時間が若干増加している（図 11（a），（b））。図 11（c），（d）は ICCG ソルバーの 1 反復あたりの計算時間と GFLOPS 値である。色数の増加とともに，計算性能はむしろ向上しているが，色数が 500 を超えると若干の性能の低下が認め

られる。400 色での計算性能は 9.74 GFLOPS であり，これは 8 コアのピーク性能の 13.2%にあたり，非構造格子を使用した計算としては比較的高い性能である。

このように，色数が増加しても性能が低下しない原因として，三次元弾性問題を解いていることが考えられる。三次元弾性問題では 1 つの節点（要素の頂点）あたりに 3 方向に 3 つの自由度を有するため,3 自由度をまとめて処理することが行なわれる。係数行列で考えると 3×3=9 個の成分をまとめて処理することになる。したがって各ループ内における演算量がポアソン方程式の場合と比較して多くなるため，OpenMP の同期オーバーヘッドの影響を比較的受けにくくなるものと考えられる。リスト 18 は三次元弾性問題向けの ICCG ソルバーにおける前進代入ループの OpenMP による並列化例である〔4〕。リスト 9 のポアソン方程式の場合と比較すると，計算量の多さは明らかである。



（a）ICCG 法ソルバーの反復回数

（b）ICCG 法ソルバーの計算時間

（c）ICCG 法ソルバー1 反復あたりの計算時間

（d）ICCG 法ソルバーの GFLOPS 値

図 11　有限要素法による三次元弾性解析結果（図 10），ICCG 法，MC 法適用，OpenMP による並列化，Hitachi SR11000/J2　1 ノード（8 コア）使用：色数と計算時間，計算効率の関係

```
      do iv= 1, NVECT
!$omp parallel do private(iS,iE,i,SW1,SW2,SW3,isL,ieL,j,k,X1,X2,X3)
        do ip= 1,PEsmpTOT
          iS= STACKmc((iv-1)*PEsmpTOT + ip - 1) + 1
          iE= STACKmc((iv-1)*PEsmpTOT + ip    )
        do i= iS, iE
          SW1= WW(3*i-2,ZP)
          SW2= WW(3*i-1,ZP)
          SW3= WW(3*i  ,ZP)
          isL= INL(i-1)+1
          ieL= INL(i)
          do j= isL, ieL
              k= IAL(j)
             X1= WW(3*k-2,ZP)
             X2= WW(3*k-1,ZP)
             X3= WW(3*k  ,ZP)
            SW1= SW1 - AL(9*j-8)*X1 - AL(9*j-7)*X2 - AL(9*j-6)*X3
            SW2= SW2 - AL(9*j-5)*X1 - AL(9*j-4)*X2 - AL(9*j-3)*X3
            SW3= SW3 - AL(9*j-2)*X1 - AL(9*j-1)*X2 - AL(9*j  )*X3
          enddo
          X1= SW1
          X2= SW2
          X3= SW3
          X2= X2 - ALU(9*i-5)*X1
          X3= X3 - ALU(9*i-2)*X1 - ALU(9*i-1)*X2
          X3= ALU(9*i  )*  X3
          X2= ALU(9*i-4)*( X2 - ALU(9*i-3)*X3)
          X1= ALU(9*i-8)*( X1 - ALU(9*i-6)*X3 - ALU(9*i-7)*X2)
          WW(3*i-2,ZP)= X1
          WW(3*i-1,ZP)= X2
          WW(3*i  ,ZP)= X3
        enddo
        enddo
!$omp end parallel do
      Enddo
```

リスト 18　三次元弾性解析における前進代入（[M]{z}={r}）のループの OpenMP による並列化

6. まとめ，問題点

（1）ICCG 法ソルバーの並列化について

本連載では，情報基盤センターの Hitachi SR11000 の利用者を対象として，「ICCG 法による有限体積法向けポアソン方程式ソルバーの OpenMP による並列化」の内容を解説した。本連載の目的は，OpenMP の個々のディレクティヴや文法的な事項について解説することではなく，OpenMP を適用するために必要なプログラミングやデータ構造の考え方を伝えることである。

ICCG法においては，メモリへの書き込みと参照が同時に起こるような「データ依存性（data dependency)」が生じる。これを回避して，OpenMPによる並列化を実施するためには，適切なデータの並べ替えを施す必要があるが，このような場合の対策はOpenMP向けの解説書でも詳しく取り上げられることはこれまで余り無かった。本連載では，「並び替え（ordering)」のための手法として，MC法（multicolor ordering)，CM法（Cuthill-McKee ordering)，RCM法（Reverse Cuthill-McKee ordering）を紹介し，Hitachi SR11000/J2上でOpenMPを使用して並列化し，高い性能を得られることを示した。

色数（レベル数）を増加させることによって，反復法（ICCG法）の収束までの反復回数は減少する。しかし，OpenMPの同期オーバーヘッドのため，計算性能は低下し，結果的に反復回数が少なくなっているのに計算時間が増加するような例は本連載（今回）でも紹介したように実問題ではしばしば起こりうる。扱う問題，境界条件，問題サイズや使用計算機によって，最適解は

様々である。データ依存性を含むループに対してOpenMPによる並列化を実施する場合には，色数，反復回数と計算時間のトレードオフを常に念頭に置く必要がある。

また，色数，レベル数が増加すると収束のための反復回数は減少する傾向にあるが，ローカルに見ると必ずしもそうではなく，本連載で説明した「ICN」だけで全てが説明できるわけではない。色数と収束性の関係については様々な研究がなされており，この分野に興味のある読者は例えば文献〔5〕などを参考にされると良い。

本連載では，主として反復法ソルバー（ICCG法），係数マトリクス成分計算部分の並列化に主眼を置いた。同様の方法で係数マトリクスコネクティビティについては並列化が可能であるが，問題は，MC法，CM法，RCM法などの「並び替え（ordering）」の部分である。MC法などのordering手法の並列化についてもここ10年ほど研究が進められている。この分野の研究に興味を持たれた読者は文献〔6〕，〔7〕などを参考にされると良い。

（2）マルチコアプロセッサでの稼動

本連載で紹介したプログラムはマルチコアプロセッサを搭載した PC でも動作可能である。Intel，PGI などのコンパイラでも動作は確認済である。ただし「ICCG-L3」を稼動させるには MPI をインストールしておくことが必要である。しかし，MPI は時間計測に使用しているだけなので，MPI の部分をコメントアウトしてリコンパイル，実行しても構わない。

問題は性能で，図 7 に示したような性能向上は市販の PC では達成できない。図 12 はクァドコアの Intel Xeon Clovertown（1.66GHz）を 2 台（すなわち 8 コア）使用して，「ICCG-L3」を実行した計算例である。コンパイラは Intel FORTRAN 9.1, コンパイルオプションとしては「`-fast -openmp`」を使用している。

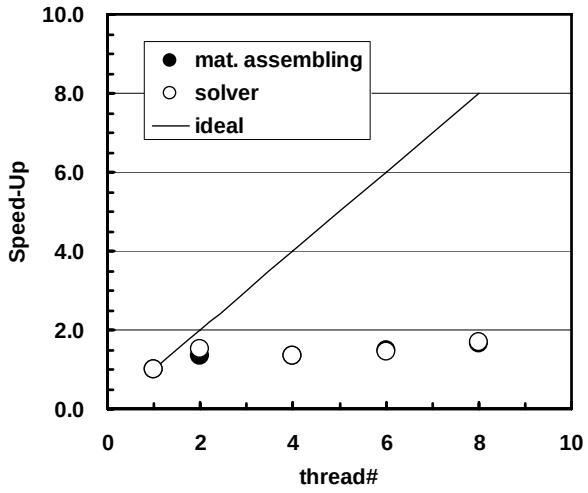


図 12　「ICCG-L3」の計算結果：スレッド数を変化させた場合の性能向上比（1 スレッドを基準）（Intel Xeon Clovertown（1.66GHz）×2，Intel FORTRAN 9.1，NX=NY=NZ=100，要素数=10^6）

上図からも明らかなように，2 コアのときに 1.5 倍程度の性能になってからは，コア数（スレッド数）を増やしてもほとんど性能は向上しない。このような現象はメモリの性能（バンド幅，レイテンシ），キャッシュの性能・容量の相違に起因している。

良く知られているように昨今の共有メモリ型マルチプロセッサコンピュータシステムの代表的アーキテクチュアとして NUMA（Non-Uniform Memory Access）が採用されている。このよ

うな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，データ配置等を配慮する必要がある。図 13 に示すように，Hitachi SR11000 で使用されている，IBM Power 5，IBM Power 5+も NUMA アーキテクチュアによっているが〔8〕，メモリ性能，キャッシュ性能・容量が高い（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。

NUMA ではある変数を最初にアクセスしたコア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」〔9〕）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場合もある。本連載で扱ったような 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，キャッシュフレンドリーでなく，NUMA アーキテクチュアにも向いていないため，「First Touch Rule」に基づいて配列を初期化しても性能はほとんど向上しない。

アルゴリズムも含めて，マルチコアプロセッサを搭載した PC での性能の向上は今後の大きな技術的課題である。

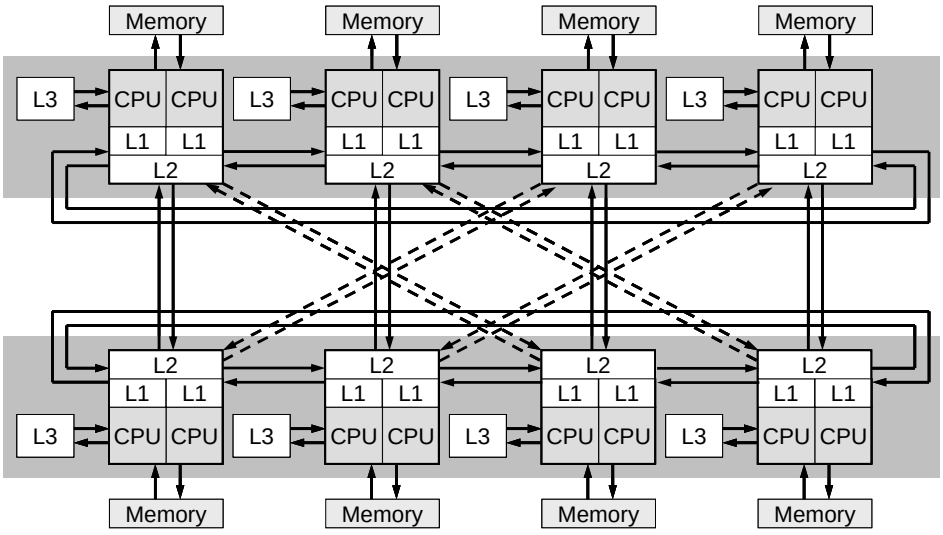


図12　Hitachi SR11000/J1とIBM p5-595のノード（プロセッサブック）のアーキテクチュア，点線はHitachi SR11000/J1特有のマルチコアモジュール（Multi Core Module，MCM）間の結線を示す（16コア／ノードの場合）（〔8〕により作成）

謝 辞

本連載の執筆にあたっては多くの皆様にご助力とご指導を頂いた。この場を借りて深く御礼を申し上げたい。特に本連載の執筆を勧めてくださった片桐孝洋准教授をはじめ，東京大学情報基盤センターの皆様には大変お世話になった。深甚なる謝意を表するものである。

計算機科学を専門としない学生に対して科学技術シミュレーションのための並列プログラミング技術を体系的に教える，という意欲的な試みに挑戦できたことは貴重な体験であった。このような機会を与えてくださった山形俊男教授，松浦充宏教授を始め，東京大学大学院理学系研究科地球惑星科学専攻の皆様にも深く感謝したい。

また，C 言語版のプログラムの作成に協力いただいた江連真一氏（エム・アール・アイ リサーチアソシエイツ）にも厚く御礼申し上げる。

最後に，「並列計算プログラミング」，「先端計算機演習Ⅰ・Ⅱ」を受講した学生諸君には，貴重な意見を率直に述べてくれたことを深く感謝したい。

参 考 文 献

[1] （株）日立製作所（2005）ベクトル並列型スーパーコンピューター SR11000　チューニングガイド，東京大学情報基盤センター

[2] 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, 7-26

[3] Doi, S. and Washio, T. (1999) Using Multicolor Ordering with Many Colors to Strike a Better Balance between Parallelism and Convergence, RIKEN Symposium on Linear Algebra and its Applications, 19-26

[4] Nakajima, K. (2005) Three-Level Hybrid vs. Flat MPI on the Earth Simulator: Parallel Iterative Solvers for Finite-Element Method, Applied Numerical Mathematics 54, 237-255

[5] 岩下武史，島崎眞昭（2006）ランダムスパース係数行列に対する不完全コレスキー分解前処理におけるオーダリングの評価指標，情報処理学会論文誌：コンピューティングシステム 47-SIG3（ACS13），40-48

[6] Gebremedhin, A., Manne, F. (2000) Scalable Parallel Graph Coloring Algorithms, Concurrency: Practice and Expereince 12, 1131-1146

[7] Gebremedhin, A., Guerrin-Lassous, I., Gustedt, J. and Telle, J.A. (2003) Graph Coloring on Coarse Grained Multicomputers, Discrete Applied Mathematics 131-1, 179-198

[8] 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

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

（問い合わせ先）

中島 研吾（nakajima@eps.s.u-tokyo.ac.jp）

（本連載のホームページ）

http://www-solid.eps.s.u-tokyo.ac.jp/~nakajima/tutorial/sr11k/