付章　スカイライン法アルゴリズムの並列化

付.1 はじめに

本章では、SPACE Ver. 3.40 以上で用いているスカイライン法の並列化について述べる。同法は係数行列が実対称でしかもバンド状である線形方程式を効率よく解法する方法のひとつである。ここでは、最初にスカイライン法の数値計算アルゴリズムについて解説し、次にどのように並列化するかについて述べる。ここで用いる並列化手法はスレッド並列と呼ばれ、マルチコアの演算装置を有するコンピュータに適している。並列化には Frotran コードと OpenMP[8] を使用する。

付.2 スカイライン法による LDL^{T}分解と方程式の解法

線形の方程式は次式で表される。

$$[G]\{u\}=\{p\} \qquad \cdots\cdots\cdots(付.1)$$

係数行列$[G]$は実対称行列を仮定すると次のように一意に分解可能である。

$$[G]=[L][D][L]^T \qquad \cdots\cdots\cdots(付.2)$$

ここで、$[L]$は下三角行列であり、対角項は全て１である。また、$[D]$は対角行列である。上式は、アルゴリズムを述べる都合で次のように表される。

$$[G]=[L][U] \qquad \cdots\cdots\cdots(付.3)$$

上の行列$[U]$は上三角行列であり、下式で示される。

$$[U]=[D][L]^T \qquad \cdots\cdots\cdots(付.4)$$

式(付. 3)は、行列の成分で示すと次のように表される。

$$g_{ij}=\sum_{k=1}^{i} l_{ik}u_{kj}=\sum_{k=1}^{i-1} l_{ik}u_{kj}+u_{ij} \quad (i\le j) \qquad \cdots\cdots(付.5)$$

$$g_{ij}=\sum_{k=1}^{j} l_{ik}u_{kj}=\sum_{k=1}^{j-1} l_{ik}u_{kj}+l_{ij}u_{jj} \quad (i\ge j) \qquad \cdots\cdots(付.6)$$

式(付. 6)を少し変更すると、下三角行列の成分l_{ij}が次のように求められる。

$$l_{ij}=\frac{1}{u_{jj}}\left(g_{ij}-\sum_{k=1}^{j-1} l_{ik}u_{kj}\right) \quad (i>j) \qquad \cdots\cdots(付.7)$$

次に、式(付.5)で i=j とすると、

$$u_{ii} = g_{ii} - \sum_{k=1}^{i-1} l_{ik} u_{ki} \qquad \cdots\cdots\cdots(付.8)$$

となる。ここで、式(付.4)も行列の成分で表すと次式となる。

$$u_{ij} = d_{ii} l_{ji} \qquad \cdots\cdots\cdots(付.9)$$

ここでは、d_{ii} は対角行列の対角項を示し、l_{ji} は$[L]^T$ が転置行列であることを利用している。上式を式(付.7)と式(付.8)に代入すると、スカイライン法アルゴリズムの基礎式が以下のよう得られる。

$$l_{ij} = \frac{1}{d_{jj}}(g_{ij} - \sum_{k=1}^{j-1} l_{ik} d_{kk} l_{jk}) \quad (i > j) \qquad \cdots\cdots\cdots(付.10)$$

$$d_{ii} = g_{ii} - \sum_{k=1}^{i-1} l_{ik}^2 d_{kk} \qquad \cdots\cdots\cdots(付.11)$$

ここでは、l_{jj} が 1 であることを用いている。

次に、スカイライン法アルゴリズムがどのように動作するかについてみていこう。まず、1 行目では、l_{11} は下三角行列の定義より 1 であり、d_{11} は式(付.11)より $d_{11} = g_{11}$ となる。第 2 行目では式(付.10)より l_{21} は、

$$l_{21} = \frac{1}{d_{11}}(g_{21}) \qquad \cdots\cdots\cdots(付.12)$$

となり、l_{22} は 1 である。また、d_{22} は式(付.11)より、

$$d_{22} = g_{22} - l_{21}^2 d_{11} \qquad \cdots\cdots\cdots(付.13)$$

となり、既に、l_{21} と d_{11} は求められていることから容易に計算される。さらに、第 3 行目では、下三角行列の成分は、

$$\left.\begin{aligned} l_{31} &= \frac{1}{d_{11}}(g_{31}) \\ l_{32} &= \frac{1}{d_{22}}(g_{32} - l_{31} d_{11} l_{21}) \\ l_{33} &= 1 \end{aligned}\right\} \cdots\cdots\cdots(付.14)$$

として得られ、d_{33} は、

$$d_{33} = g_{33} - l_{31}^2 d_{11} - l_{32}^2 d_{22} \qquad \cdots\cdots\cdots(付.15)$$

となる。以上のように、下三角行列と対角行列の全ての成分は、第 1 行目から順次計算することによって、容易に求められることが分かる。

スカイライン法アルゴリズムについて、さらに理解を深めるために図を用いて説明しよう。式(付.10)は、図のように第 i 行と第 j に関連して計算が行われ、式中の積和は両行のバンド幅外では当然行われないこ

とになる。求めた下三角行列の成分は、元の係数行列と同じ配列位置に置き換えて保存されることになる。

特に、計算過程で第 i 行を次のように置くと、

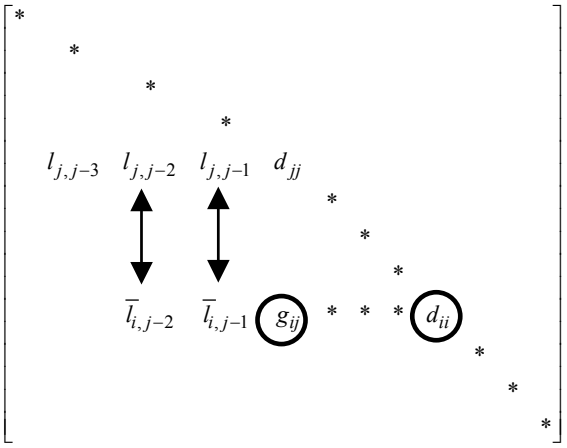
$$\bar{l}_{ij} = g_{ij} - \sum_{k=1}^{j-1} l_{ik} d_{kk} l_{jk} \quad (i > j) \quad \cdots\cdots\cdots(付.16)$$

式(付. 10)及び式(付. 11)の三重積の和から次のように単純な積和に置き直すことができる。

$$\bar{l}_{ij} = g_{ij} - \sum_{k=1}^{j-1} \bar{l}_{ik} l_{jk} \quad (i > j) \quad \cdots\cdots\cdots(付.17)$$

$$d_{ii} = g_{ii} - \sum_{k=1}^{i-1} \bar{l}_{ik} l_{ik} \quad \cdots\cdots\cdots(付.18)$$

$$l_{ij} = \frac{\bar{l}_{ij}}{d_{jj}} \quad \cdots\cdots\cdots(付.19)$$



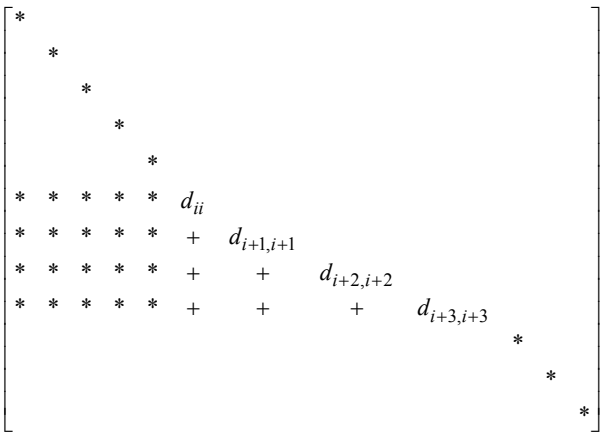
図付-1 スカイライン法のアルゴリズム

スカイライン法の分解アルゴリズムでは、式(付. 17)により第 i 行 j 列の下三角行列に関する計算過程の値 $\bar{l}_{ij}$ を求めて保存し、この操作を i-1 列まで全て実施する。その後、式(付. 18)を用いて対角行列の対角項を求める。最後に、式(付. 19)を計算して第 i 行の下三角行列成分を求めることになる。

付. 3 スカイライン法による LDL^T 分解の並列化

ここでは、前節で説明した LDL^T 分解アルゴリズムについて、並列的に処理する方法を述べる。スカイライン法で用いられている係数行列は、図の左より対角項まで各行のバンド内のデータが、1 行から n 行まで 1 次元で記憶されている。このような圧縮した記憶法は、メモリと演算処理の効率化をもたらす。特に、先のアルゴリズムの説明で分かるように、データへのアクセスが局所的になり、2 次キャシュを有する演算装置に有効である。

次に、このスカイライン法による LDL^T 分解アルゴリズムを、並列的に演算する方法を考えてみよう。式(付. 17)のアルゴリズムをそのまま採用すると、図付-2 に示すように数行を同時に行う方法が考えられる。例えば、図のように 4 行を同時に計算するためには、4 つのスレッドを生成し、並列的に各行の下三角行列の成分と対角行列の対角項を計算する。そこで問題となるのは、並列に実行される各スレッド間での演算で



図付-2 アルゴリズムの並列化

データに依存性が存在し、データ競合が起こることである。図付-2 の＋部分において式(付.17)の計算で上の行の結果が必要となり、また、対角項の計算（式(付.18)）で横の＋部分の結果が必要となる。一方、＊部分の下三角行列の成分における演算ではデータ依存がなく、並列処理が可能となる。

続いて、式(付.18)で示される対角項の計算であるが、この積和部分で＋部分の演算が必要となり、結果、他のスレッドとのデータ依存が発生する。しかしながら、＊部分の積和ではデータ依存がないため、並列計算内で下三角行列成分を求めた直後に、この積和をとることができる。

以上のように、＊部分を並列処理で行い、＋部分を逐次で行うように処理を分割することで、全体の分解計算が可能となる。しかも、逐次処理で求める下三角行列の成分は、（スレッド数＊（スレッド数-1）/2 で与えられ、例えば、スレッド数が 4 の場合はわずかに 6 となり、その他の計算部分によって、大半は並列で処理されることになる。このように未知数が多い場合で、しかもマルチコアの PC では、ここで示した並列処理手法が非常に有効となる。

並列処理を効率よく実行するためには、ワーク領域が必要となる。ここでは、2 つのワーク領域を用意する。ひとつは、式(付.17)で求められる下三角行列の第 i 行成分計算過程 $\overline{l}_{ij}$ を記憶する配列である。この配列名を twork()とすると、その配列要素数は、スレッド数＊未知総数となる。他の一つは、式(付.18)の対角項を計算するために必要であり、積和を保存するための配列である。この配列名を swork()とすると、その配列要素数はスレッド数である。

スレッドの生成や、並列処理の実施には手続きが必要であり、かなりの時間を費やす。これはオーバーヘッドと呼ばれ、並列処理の効率化とトレードオフの関係にある。並列内の処理時間が短いと逆にオーバーヘッドによって、処理時間が長くなる場合がある。そこで、分解処理プログラムを作成する際、次の 2 点に注意すべきである。一つ目は未知総数がある閾値より小さいときは自動的に逐次処理となるようにすること。二つ目は、最初の数十行はバンド幅が短く並列処理の効果が期待できないことから、これもある閾値より小さい行は逐次処理を行い、その後並列処理に移ることができるようにすることである。特に、この 2 つの閾値は使用するコンピュータシステムに依存するため、変更が可能となるように設定すべきである。

付.4 スカイライン法による線形方程式の解法

前節では、線形方程式の係数行列を並列的に LDL^T 分解する手法について説明した。本節では、この分解された係数行列を有する方程式の解を並列的に求める手法について説明する。

係数行列が LDL^T 分解された線形方程式は以下のように表される。

$$[L][D][L]^T\{v\}=\{p\} \qquad \cdots\cdots\cdots(付.20)$$

ここで、

$$[D][L]^T\{v\}=\{x\}$$
$$[L]^T\{v\}=[D]^{-1}\{x\} \qquad \cdots\cdots\cdots(付.21)$$

とおくと、式(付.20)は、以下のようになる。

$$[L]\{x\}=\{p\} \qquad \cdots\cdots\cdots(付.22)$$

式(付.21)の $[D]^{-1}$ は対角行列の逆行列であるため、対角項の逆数をとれば与えられ、右辺項は容易に得られる。式を少し変更したことで、LDL^T 分解された線形方程式は式(付.22)と(付.21)を続けて解くことに帰着する。いずれも三角行列を係数行列に持つことから、容易にその解が得られることが分かる。一般に前者は前進代入法、後者は後退代入法によって解を求めると呼ばれる。

式(付.22)と式(付.21)は成分を用いて表すとi行は以下のようになる。

$$\sum_{k=1}^{i} l_{ik}x_k = p_i \qquad \cdots\cdots\cdots(付.23)$$

$$\sum_{k=n}^{i} l_{ki}v_k = \frac{x_i}{d_{ii}} \qquad \cdots\cdots\cdots(付.24)$$

上式で、$l_{ii}=1$ であることを考慮すると、次式のように変換される。

$$\sum_{k=1}^{i-1} l_{ik}x_k + x_i = p_i \qquad \cdots\cdots\cdots(付.25)$$

$$\sum_{k=n}^{i+1} l_{ik}v_k + v_i = \frac{x_i}{d_{ii}} \qquad \cdots\cdots\cdots(付.26)$$

さらに、移項すると、

$$x_i = p_i - \sum_{k=1}^{i-1} l_{ik}x_k \qquad \cdots\cdots\cdots(付.27)$$

$$v_i = \frac{x_i}{d_{ii}} - \sum_{k=n}^{i+1} l_{ki}v_k \qquad \cdots\cdots\cdots(付.28)$$

となる。上式が前進代入と後退代入の基礎式であり、前者はi=1からnに向かって逐次演算することで解が求まり、また後者はi=nから1に向

かって代入することで解が求められることになる。

次に、式(付.27)と(付.28)を用いて、前進代入と後退代入演算の並列化を行う。同式から分かるように係数行列の分解に比較して、並列化内の演算数が小さいため効率化が悪い。そのため、未知総数が小さい場合は、オーバーヘッドで消費される時間が並列による効率化を上回ってしまい、演算時間数が逐次処理より長くなることが予想される。

この2種の代入法は、LDL^T分解されたスカイライン行列の保存法によって、演算の局所性を考慮して代入方法を工夫すべきである。前進代入法の式(付.27)の積和はメモリアクセスに局所性があり、通常の方法で演算することができる。従って、並列処理方法として数行の前進代入を同時に実施することが考えられる。並列処理を行うスレッド間に、ここでもLDL^T分解と同様に、データ依存があり、並列処理を行う部分と逐次処理を行う部分とを分離して、積和演算を実施する必要がある。

次に、後退代入法であるが、式(付.28)から分かるように、積和部分に関する下三角行列へのメモリアクセスは飛び飛びに列を渡って行われている。行のバンド幅が長いと、キャシュ外にアクセスする可能性があり、演算時間が長くかかることがある。そこで、積和部分を次のように展開し、

$$v_i = \frac{x_i}{d_{ii}} - l_{ni}v_n - l_{n-1,i}v_{n-1} - l_{n-1,i}v_{n-2} - \cdots \qquad \cdots\cdots\cdots(付.29)$$

後退代入で解v_iがひとつ求められると、式(付.29)のv_iと該当する下三角行列i行と掛け算し、解ベクトルから引き算を行う。この処理をi=nから1までを後退代入を実施すると、自動的に解ベクトルが求められることになる。ここで求められたひとつのv_iと下三角行列i行ベクトルとの掛け算の処理では、連続した保存場所へのアクセスとなるため、メモリへのアクセス局所性があり、演算時間が短くなる。また、このデータv_iと下三角行列i行ベクトルとの掛け算ではデータの依存性がないため、この処理の並列化が可能である。この行のバンド幅を考慮して、ベクトル長をスレッドで分割し、上記の演算を並列に実施する。

以上で、分解された係数行列を有する方程式の解を並列的に求める手法について説明した。LDL^T分解法と異なり、並列化の効率は必ずしも良いとは言えず、特に未知数が小さいときはオーバーヘッドによって、演算時間が長くなることがある。そのため、このプログラムにも、未知総数によっては、全て逐次処理で行うためのオプションを用意すべきである。

付.5 並列スカイライン法の使用方法

本節では、実際に SPACE で使用されている並列スカイライン法に関する並列用のソースコード示す。最初に、OpenMP を使用して並列化した LDL^T 分解用サブルーチン decomp_sky_pa()の Fortran によるソースコードを示す。サブルーチン引数の仕様はコメント行に書かれている。注意事項を良く理解した上で使用されたい。

このソースコードの中で、!$　で始まる文が OpenMP のコードである。OpenMP は、記述仕様が標準化されており、多くのコンパイラーに標準で仕様可能となっている。文法などについては文献 8 を参照されたい。

```
C-----------------------------------------------------------------------
C     ●  SUBROUTINE /decomp_sky_pa
C     -------------------------------------------------------------
C     ●  並列スカイライン法（ＬＤLt分解）OMP版
C-----------------------------------------------------------------------
      SUBROUTINE decomp_sky_pa(ntdil,ntdis,n_unknown,nsum_d,gskym,
     1                 gskym_d,nwork,twork,swork,iexit,i_exnwork,n_sld)
      IMPLICIT  REAL*8(A-H,O-Z)
!$ use omp_lib     ! Open MP のmoduleの使用宣言
      INTEGER*4 i_nb,nsum_d(0:ntdis)
      DIMENSION gskym(ntdil),gskym_d(ntdis)
          dimension nwork(ntdis),twork(*),swork(*)
C-----------------------------------------------------------------------
C     -   スカイライン法によるＬＤLt分解
C-----------------------------------------------------------------------
C     -   ntdil         : 全体剛性行列の次元での配列次数(in)
C     -   ntdis         : nworkワーク、及び対角行列対角項の配列次数(in)
C     -   n_unknown     : 未知数(in)
C     -   nsum_d        : スカイライン法による未知数格納用指標(対角項位置番号) (in)
C                         gskym()各行対角項の配列番号  nsum_d(0)=0;nsum_d(1)=1;nsum_d(i);;
C     -   gskym         : 全体剛性マトリックス(in)  LDLt分解後下三角行列Ｄ(Out)
C                         次元圧縮型、各行バンド端より対角項まで、i=1からｎまで
C     -   i_exnwork     : nworkを計算するか否かのパラメタ(in) 0：内部で計算  ：計算せず
C-----------------------------------------------------------------------
C     -   n_sld         : 並列用スレッド数(in)
C     -   gskym_d       : LDLt分解後のＤ行列対角項(out)
C     -   nwork         : ワークエリア各行のバンド幅（先頭列番号）(Out) ntdis次の領域を確保すること
C     -   twork         : ワークエリア(n_sld*n_unknown)(Out)  n_sld*n_unknown次の領域を確保すること
C     -   swork         : ワークエリア(n_sld)(Out)  n_sld次の領域を確保すること
C     -   iexit         : 計算コード(0:正常終了1:偽特異行列)(Out)
C-----------------------------------------------------------------------
C     -   n_x1          : 並列開始閾値(各個人でコンパイル前に設定)
C     -   n_x2          : 並列可能閾値(各個人でコンパイル前に設定)
C     -   n_xx          : 逐次計算最大数（内部で計算）
C     -   n_yy          : 並列計算総数（内部で計算）
c-----------------------------------------------------------------------
C     ------------内部計算チューニング用パラメタの設定
c-----------------------------------------------------------------------
      iexit=0     ! 処理コード  :正常終了  :特異行列
      n_x1=10     ! 並列開始未知番号（以下にはしないこと。常にまでは逐次で行っている）
      n_x2=100    ! この値以下の未知数は全て逐次処理で分解する
```

```
c      n_sld=4   ! スレッドは主プログラムで定義
c-----------------------------------------------------------------------
c          write(76,'(20e12.5)')(gskym(nsum_d(l)),i=1,n_unknown)
c          write(76,'(a)') ' decomp ',n_unknown
c          do i=1,n_unknown
c          write(76,'(2i10,e12.4)')i,nsum_d(l),gskym(nsum_d(l))
c          enddo
c-----------------------------------------------------------------------
C     ---------- ワークエリア各行のバンド幅（先頭列番号）の計算処理
c-----------------------------------------------------------------------
      if(i_exnwork.eq.0) then
      do i=1,n_unknown
      nwork(i)=i-(nsum_d(i)-nsum_d(i-1))+1     ! i行目のバンド先頭列番号
      enddo
      i_exnwork=1                              ! 一回上の表を作成した後、次回からは設定しないようにするパラメタ
      endif
c-----------------------------------------------------------------------
C     ----------i=1の場合の分解処理
c-----------------------------------------------------------------------
      if(gskym(1).eq.0.0d0) goto 99
      gskym_d(1)=1.0D0/gskym(1)
c-----------------------------------------------------------------------
C     ----------i=2の場合の分解処理
c-----------------------------------------------------------------------
      i_nb=nwork(2)
      s=0.0d0
      ii=nsum_d(1)-i_nb+1
      if(i_nb.eq.2) then
       if(gskym(nsum_d(2)).eq.0.0d0) goto 99
       gskym_d(2)=1.0d0/gskym(nsum_d(2))
      else
       twork(1)=gskym(ii+1)
       gskym(ii+1)=gskym_d(1)*twork(1)
C     ------------------------------------
       ss=gskym(nsum_d(2))-gskym(ii+1)*twork(1)
       if(ss.eq.0.d0) goto 99
        gskym_d(2)=1.0d0/ss
        gskym(nsum_d(2))=1.0D0/gskym_d(2)
       endif
c-----------------------------------------------------------------------
C     ---------- 並列計算戦略
c-----------------------------------------------------------------------
       if(n_unknown .le.  n_x2)then
       n_xx=  n_unknown
       n_yy=  0
       else
       n_yy= (n_unknown - n_x1)/n_sld
       n_xx=  n_unknown - n_yy*n_sld
       endif
c-----------------------------------------------------------------------
C     ---------- 逐次計算処理　i=3からi=n_xxまでの分解処理（n_xxは逐次処理限界値）
c-----------------------------------------------------------------------
      if(n_unknown.gt.2) then
      if(n_xx.ge.3) then
```

```
      do 20 i=3,n_xx
       i_nb=nwork(i)
      if(i_nb.eq.i) then
       if(gskym(nsum_d(i)).eq.0.0d0) goto 99
       gskym_d(i)=1.0d0/gskym(nsum_d(i))
      else
       jj=nsum_d(i)-i
       iii=2
       i_1=i-1
       iii=max(2,i_nb)
       do j=iii,i_1
        lgth=MAX(i_nb,nwork(j))
        if(lgth.lt.j) then
         ii=nsum_d(j)-j
         s=0.d0
         do  k=lgth,j-1
         s=s+gskym(ii+k)*gskym(jj+k)
         enddo
         gskym(jj+j)=gskym(jj+j)-s
        endif
      enddo
C     ------------------------------------
       s=0.d0
       ii=nsum_d(i-1)-i_nb+1
       do j=i_nb,i_1
        twork(j)=gskym(ii+j)
        gskym(ii+j)=gskym_d(j)*twork(j)
        s=s+gskym(ii+j)*twork(j)
       enddo
C     ------------------------------------
       ss=gskym(nsum_d(i))-s
C     ------------------------
       if(ss.eq.0.d0) goto 99
        gskym_d(i)=1.0d0/ss
      endif
   20 continue
      endif
c-------------------------------------------n_yy=0の場合は並列処理なし
       if(n_yy.ne.0 ) then
c-----------------------------------------------------------------------
C     ------------並列処理開始n_yyは並列処理総数
c-----------------------------------------------------------------------
!$OMP parallel  num_threads(n_sld)
      do 40 ix=1,n_yy
!$OMP single
      jxx=(ix-1)*n_sld + n_xx       ! 最初並列計算を行う未知番号の一つ前の番号
!$OMP end single
c-----------------------------------------------------------------------
C     ------------doループ並列処理スレッド数で並列処理を行う
c-----------------------------------------------------------------------
!$OMP do  private(i,jx,i_nb,jj,iii,j,ii,s,k,lgth,jj_sld,sxx)
      do jx=1,n_sld                 ! スレッド回数分の並列処理
       i=jxx+jx                     ! 未知番号
       i_nb=nwork(i)                ! i行バンド幅の列番号
```

```
        jj=nsum_d(i)-i              ! i行目の次元剛性行列先頭番地
        jj_sld=(jx-1)*n_unknown     ! スレッドワーク領域最初の番号
        swork(jx)=gskym(nsum_d(i))  ! 対角項を計算するために必要となる総和を入れるワーク領域に剛性対角項をセット
c        iii=i_nb                    ! i行の先頭列番号をセット
c-------------------------------------------------------行方向に下三角行列の成分計算
C                                          jxx:スレッド間でデータ依存がない項まで
        if(jxx.ge.i_nb) then
          do j=i_nb,jxx
           lgth=MAX(i_nb,nwork(j))
           s=0.d0
          if(lgth.lt.j) then
           ii=nsum_d(j)-j
c-----------------------------------------------------積和項
           do  k=lgth,j-1
            s=s+gskym(ii+k)*twork(jj_sld+k)
           enddo
           endif
c-----------------------------------------------------i行j列の下三角行列の成分セット
           s=gskym(jj+j)-s
           twork(jj_sld+j)=s
           gskym(jj+j)=s*gskym_d(j)
           swork(jx)=swork(jx)-gskym(jj+j)*s
          enddo
        endif
       enddo
!$OMP end do
c-------------------------------------------------------------------------
C     ----------逐次処理
C-------------------------------
C     ---------- スレッド番目では、既に非対角項は計算され、swork(1)に
C      積和がとられている。そのため、直接対角行列対角項を計算することができる。
c-------------------------------------------------------------------------
!$OMP single
       i=jxx+1              !  未知番号
          if(swork(1).eq.0.d0)then
            iexit=1
                write(76,'(a,i4,a,e16.5)') ' Near singular :',i
c              return  ! 偽特異行列
           else
            gskym_d(i)=1.0D0/swork(1)
           endif
c-------------------------------------------------------------------------
C     ---------- スレッド番目からn_sldまでの処理
c-------------------------------------------------------------------------
       do jx=2,n_sld
        i=jxx+jx               !  未知番号
        i_nb=nwork(i)
c-------------------------------------------------------------------------
C     ---------- i行とj行が練成しているかチェック、練成の場合は以下の処理
C             スレッドの間でデータ依存が存在するため、逐次処理で分解を行う
C             逐次処理で分解する成分数は：n_sld*(n_sld-1)/2 であり、例えば、
C             ４スレッドの場合は、となり、非常に少なく、計算時間も短い。
c-------------------------------------------------------------------------
        if(i.ne.i_nb )then
```

```
            jj=nsum_d(i)-i
            jj_sld=(jx-1)*n_unknown
            iii=MAX(jxx+1,i_nb)
           do j=iii,i-1
              lgth=MAX(i_nb,nwork(j))
              s=0.d0
            if(lgth.lt.j) then
              ii=nsum_d(j)-j
c-------------------------------------------------------積和項
             do  k=lgth,j-1
              s=s+gskym(ii+k)*twork(jj_sld+k)
             enddo
            endif
c-----------------------------------------------------
            s=gskym(jj+j)-s
            twork(jj_sld+j)=s
            gskym(jj+j)=s*gskym_d(j)
            swork(jx)=swork(jx)-gskym(jj+j)*s
           enddo
c-----------------------------------------------------------------
C     ------------- 対角行列対角項のチェック　０の場合は特異行列となる
c-----------------------------------------------------------------
           if(swork(jx).eq.0.d0)then
             iexit=1
                  write(76,'(a,i4,a,e16.5)') ' Near singular: ',i
c              return  ! 偽特異行列
            else
             gskym_d(i)=1.0D0/swork(jx)
            endif
c-----------------------------------------------------------------
C     ------------- i行とj行が練成していない場合の処理
c-----------------------------------------------------------------
         else
            if(gskym(nsum_d(i)).eq.0.d0)then
              iexit=1
                   write(76,'(a,i4,a,e16.5)') ' Near singular:',i
c             return  ! 偽特異行列
            else
             gskym_d(i)=1.0D0/gskym(nsum_d(i))
            endif
         endif
         enddo
!$OMP end single
  40   continue
!$OMP end parallel
c-----------------------------------------------------------------
C     -------------並列処理の終了
c-----------------------------------------------------------------
         if(iexit.eq.1) goto 99
         endif
         endif
c-----------------------------------------------------------------
C     -------------負固有値の数（不安定次数）を数える
c-----------------------------------------------------------------
```

```
      istable=0
      if(n_unknown.lt.n_x2) then
          do i=1,n_unknown
          if(gskym_d(i).le.0.d0) istable=istable+1
          enddo
      else
!$OMP parallel  num_threads(n_sld) firstprivate(istable)
!$OMP do  private(i) reduction(+:istable)
          do i=1,n_unknown
          if(gskym_d(i).le.0.d0) istable=istable+1
          enddo
!$OMP end do
!$OMP end parallel
      endif
C     ------------------------------------------
          write(76,'(a,i4,a,e16.5)') ' Unstable number:',istable
c         write(76,'(10e16.5)') (gskym_d(i),i=1,n_unknown)
      return
C     ------------------------------------------------------------
C     ------------------------------------------  特異行列メッセージ
C     ------------------------------------------------------------
   99 continue
      iexit=1
          write(76,'(a,i4,a,e16.5)') ' Near singular:',i
      return
      end
```

次は、解を求めるサブルーチン solve_sky_pa()であり、引数の仕様は同じくコメント行に書かれている。逐次処理の閾値は現在 n_x2=5000としているが、この値を変更する場合は注意されたい。あまり小さい値を設定すると、並列処理が逐次処理より演算効率の低下を招くことになる。

```
c-----------------------------------------------------------------------
c     ●  SUBROUTINE /solve_sky_pa(ok)
c      ------------------------------
c     ●  並列スカイライン法代入計算OMP版
c-----------------------------------------------------------------------
      subroutine solve_sky_pa(ntdil,ntdis,n_unknown,nsum_d,gskym,
     *                  gskym_d,nwork,twork,swork,pload,disp,n_sld)
C     ------------------------------------------------------------
!$ use omp_lib     ! Open MP のmoduleの使用宣言
      implicit real*8(A-H,O-Z)
      integer*4 i_nb,nsum_d(0:ntdis)
      dimension gskym(ntdil),gskym_d(ntdis),
     1          pload(ntdis),disp(ntdis)
          dimension nwork(ntdis),twork(*),swork(n_sld)
c-----------------------------------------------------------------------
C     -   スカイライン法による代入計算。
```

```
C     -   ntdil        : 全体剛性行列の定義数(in)
C     -   ntdis        : 変位ベクトル、の定義数(in)
C     -   n_unknown    : 未知変位数(in)
C     -   nsum_d       : スカイライン法による未知数格納用指標(対角項位置番号) (in)
C     -   gskym        : 全体剛性マトリックス(in)
C     -   gskym_d      : 全体剛性マトリックスの対角項
C     -   nwork        : 各行のバンド幅 (先頭列番号)
C     -   twork        : ワークエリア(n_sld*n_unknown)
C     -   swork        : ワークエリア(n_sld)
C     -   iexit        : 計算コード(0:正常終了1:偽特異行列)
C     -   pload        : 荷重ベクトル (変更されない)
C     -   disp         : 変位変位ベクトル
C     -   n_x1         : 並列開始閾値(各個人で設定)
C     -   n_x2         : 並列可能閾値(各個人で設定)
C     -   n_x3         : 後退代入並列可能閾値(各個人で設定)
c-----------------------------------------------------------------------
C     ------------
c-----------------------------------------------------------------------
      n_x1=10
      n_x2=5000
c      n_x3=n_sld*5
      n_x3=2000
      if(n_unknown .le. n_x2)then
c-----------------------------------------------------------------------
C     ------------ 逐次処理を実施する
c-----------------------------------------------------------------------
c-----------------------------------------------------------------------
c     ●前進代入法
c-----------------------------------------------------------------------
      twork(1)=pload(1)
      do i=2,n_unknown
      i_nb=nwork(i)
      ii=nsum_d(i-1)-i_nb+1
      s=pload(i)
      do j=i_nb,i-1
      s=s-gskym(ii+j)*twork(j)
      enddo
c      pload(i)=S
      twork(i)=S
      enddo
c-----------------------------------------------------------------------
c     ●後退代入法
c-----------------------------------------------------------------------
      do i=1,n_unknown
      disp(i)=twork(i)*gskym_d(i)
      enddo
      do i=n_unknown,2,-1
      i_nb=nwork(i)
      if(i_nb.ne.i)then
      ii=nsum_d(i-1)-i_nb+1
      S=disp(i)
       do j=i_nb,i-1
       disp(j)=disp(j)-gskym(ii+j)*S
       enddo
```

```
      endif
      enddo
      return
c-----------------------------------------------------------------------
C     ---------- 逐次処理終了
c-----------------------------------------------------------------------
      else
c-----------------------------------------------------------------------
C     ---------- 並列計算戦略
c-----------------------------------------------------------------------
       n_yy= (n_unknown - n_x1)/n_sld
       n_xx=  n_unknown - n_yy*n_sld
       endif
c-----------------------------------------------------------------------
C     ---------- 並列処理開始
c-----------------------------------------------------------------------
c-----------------------------------------------------------------------
c     ●前進代入法
c-----------------------------------------------------------------------
      twork(1)=pload(1)
      do i=2,n_xx
      i_nb=nwork(i)
      ii=nsum_d(i-1)-i_nb+1
      s=pload(i)
      do j=i_nb,i-1
      s=s-gskym(ii+j)*twork(j)
      enddo
c      pload(i)=S
      twork(i)=S
      enddo
c-----------------------------------------------------------------------
C     ----------並列処理　前進代入
c-----------------------------------------------------------------------
      if(n_yy.ne.0 ) then
!$OMP parallel  num_threads(n_sld)
      do ix=1,n_yy
!$OMP single
      jxx=(ix-1)*n_sld + n_xx
!$OMP end single
c-----------------------------------------------------------------------
C     ----------並列処理
c-----------------------------------------------------------------------
!$OMP do  private(i,jx,j,s,ii,i_nb,iii)
      do jx=1,n_sld
      i=jxx+jx      ! 未知番号
      i_nb=nwork(i)
      ii=nsum_d(i-1)-i_nb+1
      s=pload(i)
      iii=jxx
      if(iii.ge.i_nb)then
      do j=i_nb,iii
      s=s-gskym(ii+j)*twork(j)
      enddo
      endif
```

```
      swork(jx)=s
      enddo
!$OMP end do
!$OMP single
      i=jxx+1      ! 未知番号
c      pload(i)=swork(1)
      twork(i)=swork(1)
      do jx=2,n_sld
      i=jxx+jx      ! 未知番号
      i_nb=nwork(i)
      ii=nsum_d(i-1)-i_nb+1
      s=0.d0
      iii=jxx+1
      s=0.d0
      if(iii.lt.i_nb) iii=i_nb
      do j=iii,i-1
      s=s-gskym(ii+j)*twork(j)
      enddo
      twork(i)= S+swork(jx)
      enddo
!$OMP end single
      enddo
!$OMP end parallel
c-----------------------------------------------------------------------
C     ----------前進代入終了
c-----------------------------------------------------------------------
      endif
c-----------------------------------------------------------------------
c     ●後退代入法
c-----------------------------------------------------------------------
!$OMP parallel  num_threads(n_sld)
!$OMP do
      do i=1,n_unknown
      disp(i)=twork(i)*gskym_d(i)
      enddo
!$OMP end do
!$OMP end parallel
c-----------------------------------------------------------------------
C     ----------並列処理　後退代入
c-----------------------------------------------------------------------
      if( n_unknown.lt.nx_3)then
       do i=n_unknown,2,-1
      i_nb=nwork(i)
      if(i_nb.ne.i)then
       ii=nsum_d(i-1)-i_nb+1
       S=disp(i)
       do j=i_nb,i-1
       disp(j)=disp(j)-gskym(ii+j)*S
       enddo
       endif
c-------------------------------------
      enddo
c-----------------------------------並列処理
      else
```

```
      do i=n_unknown,2,-1
      i_nb=nwork(i)
      if(i_nb.ne.i)then
       ii=nsum_d(i-1)-i_nb+1
       S=disp(i)
!$OMP parallel  num_threads(n_sld)
!$OMP do
       do j=i_nb,i-1
       disp(j)=disp(j)-gskym(ii+j)*S
       enddo
!$OMP end do
!$OMP end parallel
       endif
c-------------------------------------
      enddo
      endif
c-------------------------------------------------
      return
      end
C-------------------------------------------------------
```

付.6 まとめ

本章では、スカイライン行列を用いた連立方程式を、LDU 分解法で数値解析する手法について解説した。特に、骨組の応力解析プログラムでは、連立方程式の解法に要するコストが大きい。そこで、このスカイライン法の数値計算アルゴリズムに少し変更を加え、並列化する手法を学んだ。この方法は、マルチコアの演算装置を有するコンピュータに適しており、大いに役立つ手法となる。