並列有限要素法による
三次元定常熱伝導解析プログラム
OpenMP＋ハイブリッド並列化

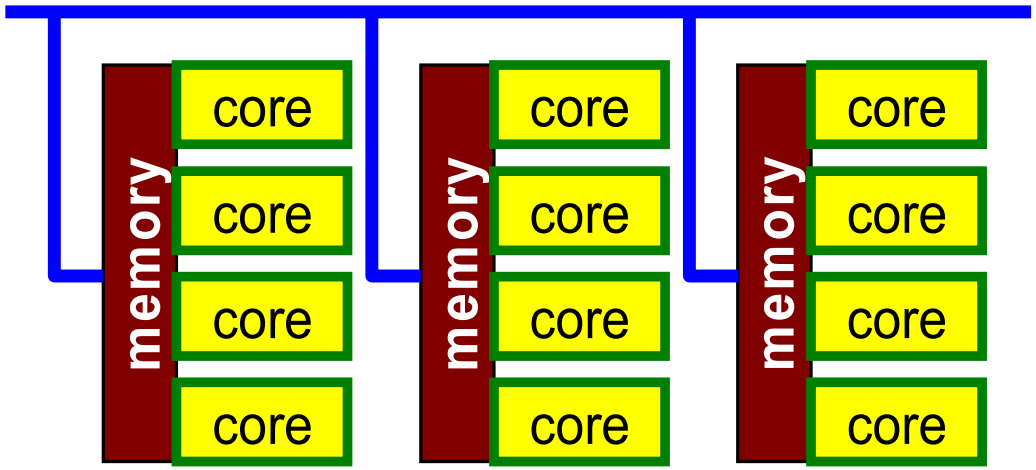
中島 研吾
東京大学情報基盤センター

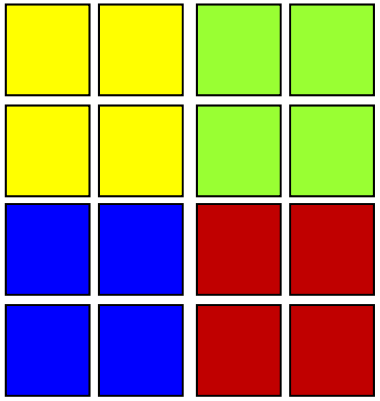
Hybrid並列プログラミング

- スレッド並列＋メッセージパッシング
  - OpenMP+ MPI
  - CUDA + MPI, OpenACC + MPI
- 個人的には自動並列化＋MPIのことを「ハイブリッド」とは呼んでほしくない
  - 自動並列化に頼るのは危険である
  - 東大センターでは現在自動並列化機能はコンパイラの要件にしていない（調達時に加点すらしない）
    - 利用者にももちろん推奨していない
- OpenMPがMPIより簡単ということはない
  - データ依存性のない計算であれば，機械的にOpenMP指示文を入れれば良い
  - NUMAになるとより複雑：First Touch Data Placement

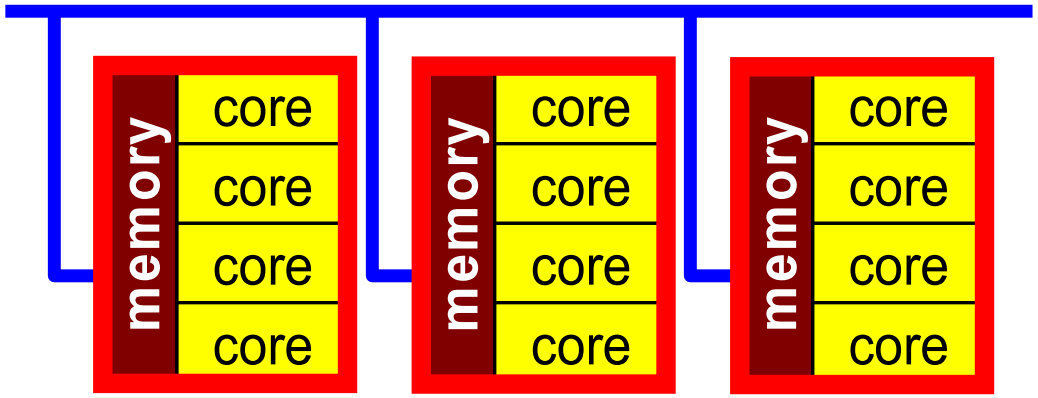
Flat MPI vs. Hybrid

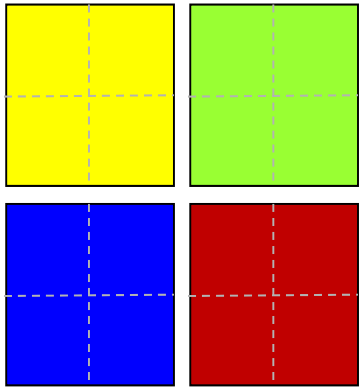
Flat-MPI：Each Core -> Independent

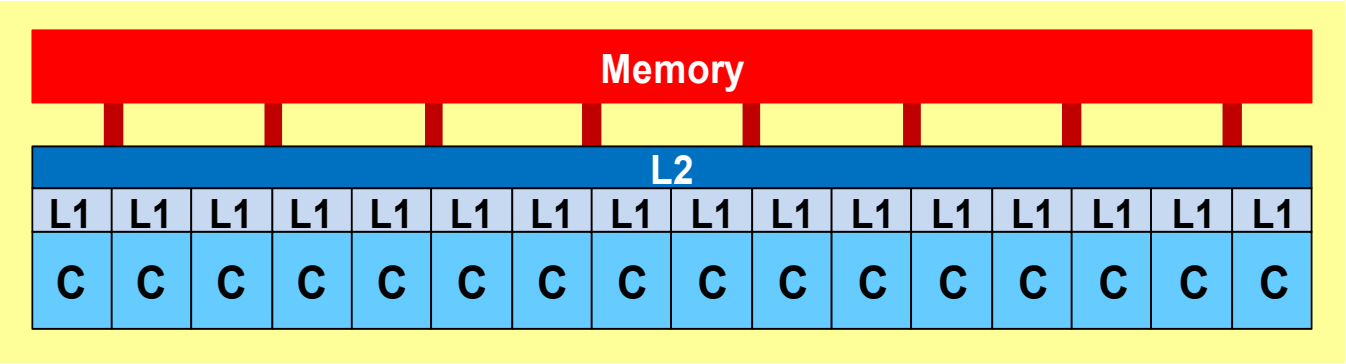




Hybrid：Hierarchal Structure







HB M x N

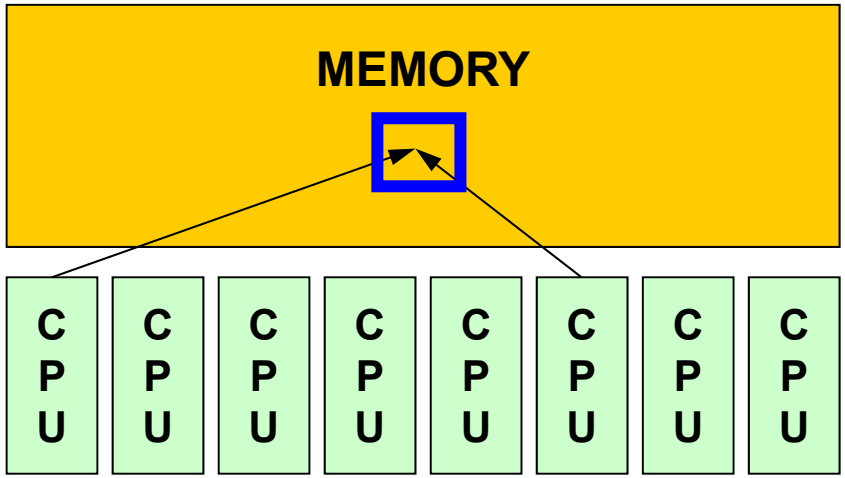
Number of OpenMP threads per a single MPI process

Number of MPI process per a single node

本編の背景

- マイクロプロセッサのマルチコア化，メニーコア化
  - 低消費電力，様々なプログラミングモデル
- OpenMP
  - 指示行（ディレクティヴ）を挿入するだけで手軽に「並列化」ができるため，広く使用されている
  - 様々な解説書
- データ依存性（data dependency）
  - メモリへの書き込みと参照が同時に発生
  - 並列化には，適切なデータの並べ替えを施す必要がある
  - このような対策はOpenMP向けの解説書でも詳しく取り上げられることは余りない：とても面倒くさい
  - **この部分は「マルチコアプログラミング」講習会で！**
- Hybrid 並列プログラミングモデル

共有メモリ型計算機



- SMP
  - Symmetric Multi Processors
  - 複数のCPU（コア）で同じメモリ空間を共有するアーキテクチュア

OpenMPとは

http://www.openmp.org

- 共有メモリ型並列計算機用のDirectiveの統一規格
  - この考え方が出てきたのは MPIやHPFに比べると遅く1996年であるという。
  - 現在 Ver.4.0
- 背景
  - CrayとSGIの合併
  - ASCI計画の開始
- 主な計算機ベンダーが集まって OpenMP ARBを結成し，1997年にはもう規格案ができていたそうである
  - SC98ではすでにOpenMPのチュートリアルがあったし，すでにSGI Origin2000でOpenMP-MPIハイブリッドのシミュレーションをやっている例もあった。

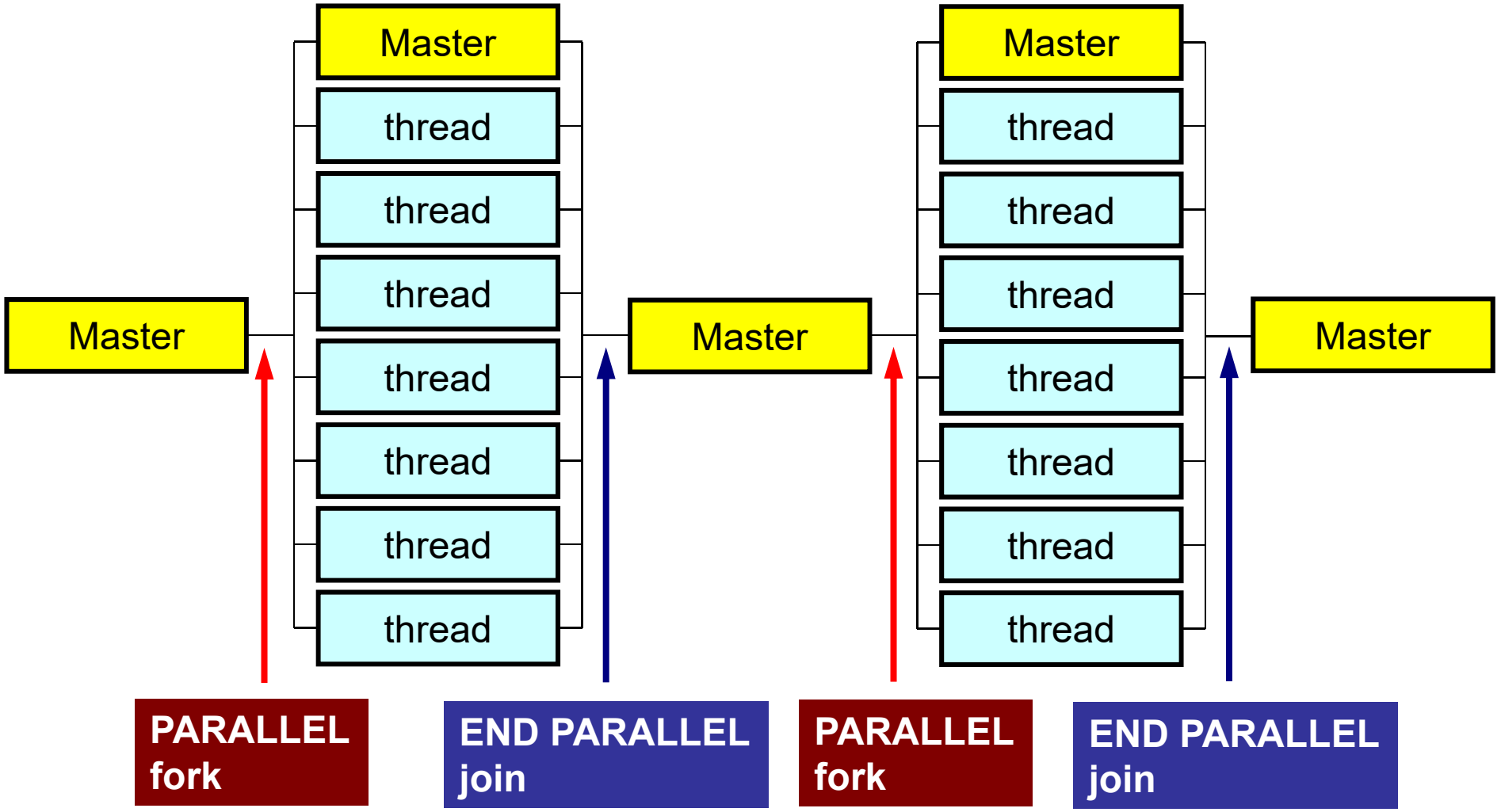
OpenMPとは（続き）

- OpenMPはFortan版とC/C++版の規格が全く別々に進められてきた。
  - Ver.2.5で言語間の仕様を統一
- Ver.4.0ではGPU，Intel-MIC等Co-Processor，Accelerator環境での動作も考慮
  - OpenACC

OpenMPの概要

- 基本的仕様
  - プログラムを並列に実行するための動作をユーザーが明示
  - OpenMP実行環境は，依存関係，衝突，デッドロック，競合条件，結果としてプログラムが誤った実行につながるような問題に関するチェックは要求されていない。
  - プログラムが正しく実行されるよう構成するのはユーザーの責任である。
- 実行モデル
  - fork-join型並列モデル
    - 当初はマスタスレッドと呼ばれる単一プログラムとして実行を開始し，「PARALLEL」，「END PARALLEL」ディレクティヴの対で並列構造を構成する。並列構造が現れるとマスタスレッドはスレッドのチームを生成し，そのチームのマスタとなる。
  - いわゆる「入れ子構造」も可能であるが，ここでは扱わない

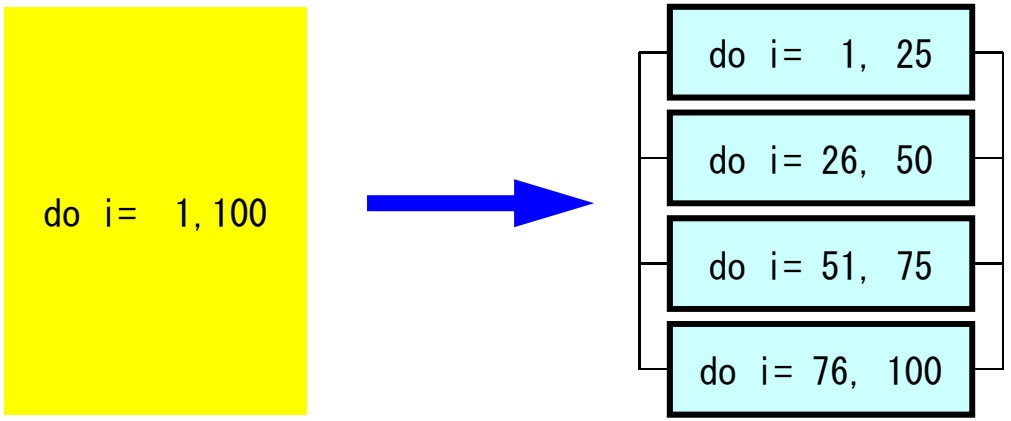
Fork-Join 型並列モデル



スレッド数

- 環境変数 **OMP_NUM_THREADS**
  - 値の変え方
    - bash(.bashrc)　　**export OMP_NUM_THREADS=8**
    - csh(.cshrc)　　**setenv OMP_NUM_THREADS 8**

- **たとえば，OMP_NUM_THREADS=4とすると，以下のように i=1~100のループが4分割され，同時に実行される。**



OpenMPに関連する情報

- OpenMP Architecture Review Board (ARB)
  - http://www.openmp.org
- 参考文献
  - Chandra, R. et al.「Parallel Programming in OpenMP」（Morgan Kaufmann）
  - Quinn, M.J.「Parallel Programming in C with MPI and OpenMP」（McGrawHill）
  - Mattson, T.G. et al.「Patterns for Parallel Programming」（Addison Wesley）
  - 牛島「OpenMPによる並列プログラミングと数値計算法」（丸善）
  - Chapman, B. et al.「Using OpenMP」（MIT Press）
- 富士通他による翻訳：（OpenMP 3.0）必携！
  - http://www.openmp.org/mp-documents/OpenMP30spec-ja.pdf

OpenMPに関する国際会議

- WOMPEI（International Workshop on OpenMP: Experiences and Implementations ）
  - 日本（1年半に一回）
- WOMPAT（アメリカ），EWOMP（欧州）

- 2005年からこれらが統合されて「IWOMP」となる，毎年開催。
  - International Workshop on OpenMP
  - http://www.nic.uoregon.edu/iwomp2005/
  - Eugene, Oregon, USA

OpenMPの特徴

- ディレクティヴ（指示行）の形で利用
  - 挿入直後のループが並列化される
  - コンパイラがサポートしていなければ，コメントとみなされる

OpenMP/Directives Array Operations

Simple Substitution

```
!$omp parallel do
      do i= 1, NP
        W(i,1)= 0.d0
        W(i,2)= 0.d0
      enddo
!$omp end parallel do
```

DAXPY

```
!$omp parallel do
      do i= 1, NP
        Y(i)= ALPHA*X(i) + Y(i)
      enddo
!$omp end parallel do
```

Dot Products

```
!$omp parallel do private(ip,iS,iE,i)
!$omp&             reduction(+:RHO)
      do ip= 1, PEsmpTOT
        iS= STACKmcG(ip-1) + 1
        iE= STACKmcG(ip  )
        do i= iS, iE
          RHO= RHO + W(i,R)*W(i,Z)
        enddo
      enddo
!$omp end parallel do
```

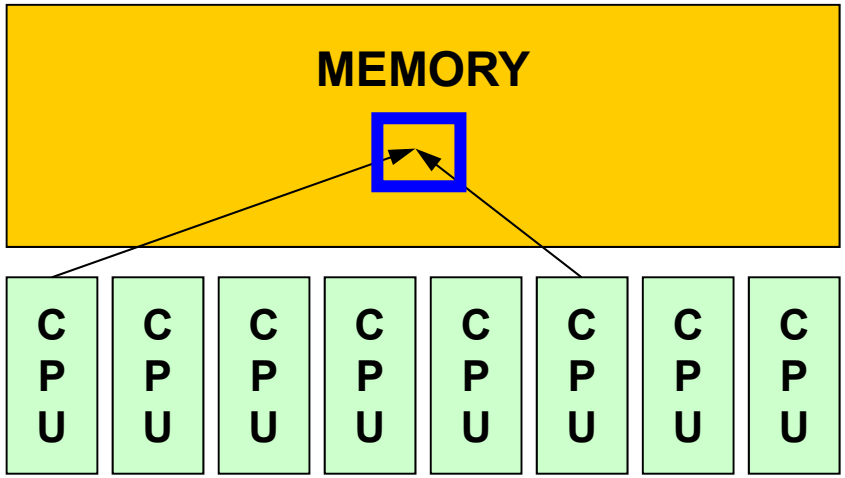
OpenMP/Direceives Matrix/Vector Products

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

OpenMPの特徴

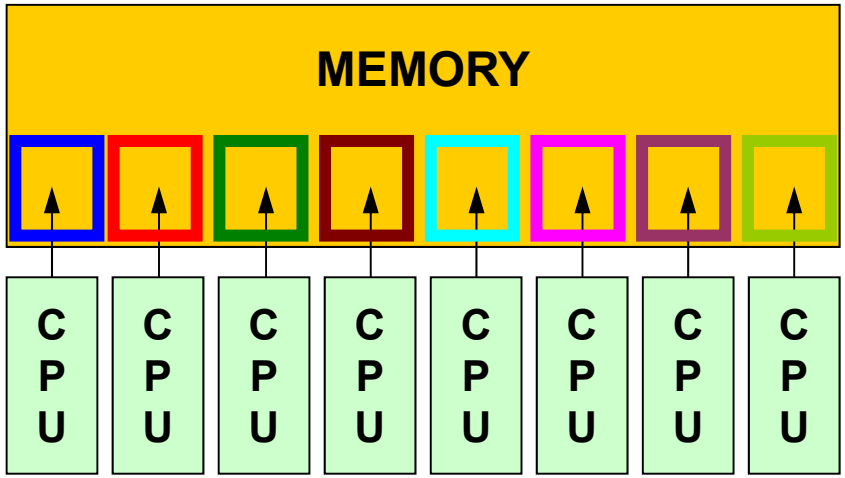
- ディレクティヴ（指示行）の形で利用
  - 挿入直後のループが並列化される
  - コンパイラがサポートしていなければ，コメントとみなされる
- 何も指定しなければ，何もしない
  - 「自動並列化」，「自動ベクトル化」とは異なる。
  - 下手なことをするとおかしな結果になる：ベクトル化と同じ
  - データ分散等（Ordering）は利用者の責任
- 共有メモリユニット内のプロセッサ数に応じて，「Thread」が立ち上がる
  - 「Thread」：MPIでいう「プロセス」に相当する。
  - 普通は「Thread数＝共有メモリユニット内プロセッサ数，コア数」であるが最近のアーキテクチャではHyper Threading（HT）がサポートされているものが多い（1コアで2-4スレッド）

メモリ競合



- 複雑な処理をしている場合，複数のスレッドがメモリ上の同じアドレスにあるデータを同時に更新する可能性がある。
  - 複数のCPUが配列の同じ成分を更新しようとする。
  - メモリを複数のコアで共有しているためこのようなことが起こりうる。
  - 場合によっては答えが変わる

メモリ競合（続き）



- 本演習で扱っている例は，そのようなことが生じないよう，各スレッドが同時に同じ成分を更新するようなことはないようにする。
  - これはユーザーの責任でやること，である。
- ただ多くのコア数（スレッド数）が増えるほど，メモリへの負担が増えて，処理速度は低下する。

OpenMPの特徴（続き）

- 基本は「!omp parallel do」～「!omp end parallel do」
- 変数について，グローバルな変数（shared）と，Thread内でローカルな「private」な変数に分けられる。
  - デフォルトは「shared」
  - 内積を求める場合は「reduction」を使う

```
!$omp parallel do private(ip,iS,iE,i)
!$omp&            reduction(+:RHO)
      do ip= 1, PEsmpTOT
        iS= STACKmcG(ip-1) + 1
        iE= STACKmcG(ip  )
        do i= iS, iE
          RHO= RHO + W(i,R)*W(i,Z)
        enddo
      enddo
!$omp end parallel do
```

W(:,:), R, Z, PEsmpTOT などはグローバル変数

FORTRANとC

```
use omp_lib
...
!$omp parallel do shared(n,x,y) private(i)
      do i= 1, n
        x(i)= x(i) + y(i)
      enddo
!$ omp end parallel do
```

```
#include <omp.h>
...
{
  #pragma omp parallel for default(none) shared(n,x,y) private(i)

    for (i=0; i<n; i++)
        x[i] += y[i];
}
```

本講義における方針

- OpenMPは多様な機能を持っているが，それらの全てを逐一教えることはしない。
  - 講演者も全てを把握，理解しているわけではない。

- 数値解析に必要な最低限の機能のみ学習する。
  - 具体的には，講義で扱っているICCG法によるポアソン方程式ソルバーを動かすために必要な機能のみについて学習する
  - それ以外の機能については，自習，質問のこと(全てに答えられるとは限らない)。
- MPIと同じ

最初にやること

- use omp_lib　　FORTRAN
- #include <omp.h>　　C

- 様々な環境変数，インタフェースの定義（OpenMP3.0以降でサポート）

OpenMPディレクィヴ（FORTRAN）

```
sentinel directive_name [clause[[,] clause]…]
```

- 大文字小文字は区別されない。
- sentinel
  - 接頭辞
  - FORTRANでは「!$OMP」,「C$OMP」,「*$OMP」, 但し自由ソース形式では「!$OMP」のみ。
  - 継続行にはFORTRANと同じルールが適用される。以下はいずれも「!$OMP PARALLEL DO SHARED(A,B,C)」

```
!$OMP PARALLEL DO
!$OMP+SHARED (A,B,C)
```

```
!$OMP PARALLEL DO &
!$OMP SHARED (A,B,C)
```

OpenMPディレクィヴ（C）

```
#pragma omp directive_name [clause[[,] clause]…]
```

- 継続行は「＼」
- 小文字を使用（変数名以外）

```
#pragma omp parallel for shared (a,b,c)
```

PARALLEL DO

```
!$OMP PARALLEL DO[clause[[,] clause] … ]
  (do_loop)
!$OMP END PARALLEL DO
```

```
#pragma parallel for [clause[[,] clause] … ]
  (for_loop)
```

- 多重スレッドによって実行される領域を定義し，DOループの並列化を実施する。
- 並び項目（clause）：よく利用するもの
  - PRIVATE（list）
  - SHARED（list）
  - DEFAULT（PRIVATE|SHARED|NONE）
  - REDUCTION（{operation|intrinsic}: list）

REDUCTION

```
REDUCTION ({operator|instinsic}: list)
reduction ({operator|instinsic}: list)
```

- 「MPI_REDUCE」のようなものと思えばよい
- Operator
  - ＋, *, -, .AND., .OR., .EQV., .NEQV.
- Intrinsic
  - MAX, MIN, IAND, IOR, IEQR

実例A1：簡単なループ

```
!$OMP PARALLEL DO
      do i= 1, N
        B(i)= (A(i) + B(i)) * 0.50
      enddo
!$OMP END PARALLEL DO
```

- ループの繰り返し変数（ここでは「i」）はデフォルトでprivateなので，明示的に宣言は不要。
- 「END PARALLEL DO」は省略可能。
  - C言語ではそもそも存在しない

実例A2：REDUCTION

```
!$OMP PARALLEL DO PRIVATE (i,Alocal,Blocal) REDUCTION(+:A,B)
      do i= 1, N
        Alocal= dfloat(i+1)
        Blocal= dfloat(i+2)
        A= A + Alocal
        B= B + Blocal
      enddo
!$OMP END PARALLEL DO
```

- 「END PARALLEL DO」は省略可能。

OpenMP使用時に呼び出すことのできる関数群

関数名	内容
int omp_get_num_threads (void)	スレッド総数
int omp_get_thread_num (void)	自スレッドのID
double omp_get_wtime (void)	MPI_Wtimeと同じ
void omp_set_num_threads (int num_threads) call omp_set_num_threads (num_threads)	スレッド数設定

OpenMPを適用するには？(内積)

```
VAL= 0.d0
do i= 1, N
  VAL= VAL + W(i,R) * W(i,Z)
enddo
```

OpenMPを適用するには？(内積)

```
      VAL= 0.d0
      do i= 1, N
        VAL= VAL + W(i,R) * W(i,Z)
      enddo
```

```
      VAL= 0.d0
!$OMP PARALLEL DO PRIVATE(i) REDUCTION(+:VAL)
      do i= 1, N
        VAL= VAL + W(i,R) * W(i,Z)
      enddo
!$OMP END PARALLEL DO
```

OpenMPディレクティヴの挿入
これでも並列計算は可能

OpenMPを適用するには？(内積)

```
      VAL= 0.d0
      do i= 1, N
        VAL= VAL + W(i,R) * W(i,Z)
      enddo
```

```
      VAL= 0.d0
!$OMP PARALLEL DO PRIVATE(i) REDUCTION(+:VAL)
      do i= 1, N
        VAL= VAL + W(i,R) * W(i,Z)
      enddo
!$OMP END PARALLEL DO
```

OpenMPディレクティヴの挿入
これでも並列計算は可能

```
      VAL= 0.d0
!$OMP PARALLEL DO PRIVATE(ip,i) REDUCTION(+:VAL)
      do ip= 1, PEsmpTOT
        do i= index(ip-1)+1, index(ip)
          VAL= VAL + W(i,R) * W(i,Z)
        enddo
      enddo
!$OMP END PARALLEL DO
```

多重ループの導入
PEsmpTOT：スレッド数
あらかじめ「INDEX(:)」を用意しておく
より確実に並列計算実施
（別に効率がよくなるわけでは無い）

OpenMPを適用するには？(内積)

```
      VAL= 0.d0
      do i= 1, N
        VAL= VAL + W(i,R) * W(i,Z)
      enddo
```

```
      VAL= 0.d0
!$OMP PARALLEL DO PRIVATE(i) REDUCTION(+:VAL)
      do i= 1, N
        VAL= VAL + W(i,R) * W(i,Z)
      enddo
!$OMP END PARALLEL DO
```

OpenMPディレクティヴの挿入
これでも並列計算は可能

```
      VAL= 0.d0
!$OMP PARALLEL DO PRIVATE(ip,i) REDUCTION(+:VAL)
      do ip= 1, PEsmpTOT
        do i= index(ip-1)+1, index(ip)
          VAL= VAL + W(i,R) * W(i,Z)
        enddo
      enddo
!$OMP END PARALLEL DO
```

多重ループの導入
PEsmpTOT：スレッド数
あらかじめ「INDEX(:)」を用意しておく
より確実に並列計算実施

PEsmpTOT個のスレッドが立ち上がり，
並列に実行

OpenMPを適用するには？(内積)

```
      VAL= 0.d0
!$OMP PARALLEL DO PRIVATE(ip,i) REDUCTION(+:VAL)
      do ip= 1, PEsmpTOT
        do i= index(ip-1)+1, index(ip)
          VAL= VAL + W(i,R) * W(i,Z)
        enddo
      enddo
!$OMP END PARALLEL DO
```

多重ループの導入
PEsmpTOT：スレッド数
あらかじめ「INDEX(:)」を用意しておく
より確実に並列計算実施

PEsmpTOT個のスレッドが立ち上がり，
並列に実行

各要素が計算されるスレッドを
指定できる

GPUには良くないらしい

例えば，N=100，PEsmpTOT=4とすると：

INDEX(0)= 0
INDEX(1)= 25
INDEX(2)= 50
INDEX(3)= 75
INDEX(4)= 100

omp parallel (do)

- omp parallel-omp end parallelはそのたびにスレッドを生成，消滅させる：fork-join
- ループが連続するとこれがオーバーヘッドになることがある。
- omp parallel + omp do/omp for

```
!$omp parallel ...

!$omp do
   do i= 1, N
...
!$omp do
   do i= 1, N
...
!$omp end parallel     必須
```

```
#pragma omp parallel ...

#pragma omp for {

...
#pragma omp for {
```

課題

- CGソルバー（solver_CG，solver_SR）のOpenMPによるマルチスレッド化，Hybrid並列化
- 行列生成部（mat_ass_main，mat_ass_bc）のマルチスレッド化，Hybrid並列化

- 問題サイズを変更して計算を実施してみよ
- Hybridでノード内スレッド数を変化させてみよ
  - OMP_NUM_THREADS

FORTRAN(solver_CG)

```
!$omp parallel do private(i)
      do i= 1, N
         X(i)  = X (i)    + ALPHA * WW(i,P)
        WW(i,R)= WW(i,R) - ALPHA * WW(i,Q)
      enddo
```

```
      DNRM20= 0.d0
!$omp parallel do private(i) reduction (+:DNRM20)
      do i= 1, N
        DNRM20= DNRM20 + WW(i,R)**2
      enddo
```

```
!$omp parallel do private(j,k,i,WVAL)
      do j= 1, N
        WVAL= D(j)*WW(j,P)
        do k= index(j-1)+1, index(j)
           i= item(k)
          WVAL= WVAL + AMAT(k)*WW(i,P)
        enddo
        WW(j,Q)= WVAL
      enddo
```

C(solver_CG)

```
#pragma omp parallel for private (i)
    for(i=0;i<N;i++){
      X [i]    +=  ALPHA *WW[P][i];
      WW[R][i]+= -ALPHA *WW[Q][i];
    }

    DNRM20= 0.e0;
#pragma omp parallel for private (i) reduction (+:DNRM20)
    for(i=0;i<N;i++){
      DNRM20+=WW[R][i]*WW[R][i];
    }

#pragma omp parallel for private (j,i,k,WVAL)
    for( j=0;j<N;j++){
      WVAL= D[j] * WW[P][j];
      for(k=indexLU[j];k<indexLU[j+1];k++){
        i=itemLU[k];
        WVAL+= AMAT[k] * WW[P][i];
    }
    WW[Q][j]=WVAL;
```

solver_SR (send)

```
      do neib= 1, NEIBPETOT
        istart= EXPORT_INDEX(neib-1)
        inum  = EXPORT_INDEX(neib  ) - istart
!$omp parallel do private(k,ii)
        do k= istart+1, istart+inum
           ii   = EXPORT_ITEM(k)
           WS(k)= X(ii)
        enddo

        call MPI_Isend (WS(istart+1), inum, MPI_DOUBLE_PRECISION,        &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD, req1(neib),     &
     &                  ierr)
      enddo
```

```
 for( neib=1;neib<=NEIBPETOT;neib++){
    istart=EXPORT_INDEX[neib-1];
    inum  =EXPORT_INDEX[neib]-istart;
#pragma omp parallel for private (k,ii)
    for( k=istart;k<istart+inum;k++){
      ii= EXPORT_ITEM[k];
      WS[k]= X[ii-1];
    }
    MPI_Isend(&WS[istart],inum,MPI_DOUBLE,
              NEIBPE[neib-1],0,MPI_COMM_WORLD,&req1[neib-1]);
  }
```

pfem3dのスレッド並列化

- CG法
  - ほぼOpenMPの指示文（directive）を入れるだけで済む
  - 前処理がILU系になるとそう簡単ではない
- 行列生成部（mat_ass_main, mat_ass_bc）
  - 複数要素から同時に同じ節点に足し込むことを回避する必要がある
    - 計算結果が変わってしまう
    - 同時に書き込もうとして計算が止まる場合もある（環境依存）
  - 色分け（Coloring）
    - 色内に属する要素が同じ節点を同時に更新しないように色分けすれば，同じ色内の要素の処理は並列にできる
    - 現在の問題は規則正しい形状なので，8色に塗り分けられる（1節点を共有する要素数は最大8，要素内節点数8）
    - 色分け部分の計算時間が無視できない：並列化困難

ファイルコピー on FX10

FORTRANユーザー

```
>$ cd ~/pFEM
>$ cp/home/S11502/nakajima/16Summer/F/fem3d1.tar .
>$ cp/home/S11502/nakajima/16Summer/F/fem3d2.tar .
```

Cユーザー

```
>$ cd ~/pFEM
>$ cp/home/S11502/nakajima/16Summer/C/fem3d1.tar .
>$ cp/home/S11502/nakajima/16Summer/C/fem3d2.tar .
```

展開

```
>$ tar xvf fem3d1.tar
>$ tar xvf fem3d2.tar
```

ディレクトリ確認

```
>$ cd pfem3d
>$ cd src1 ⇒    make ⇒ ls -l ../run/sol1
>$ cd ../src2 ⇒ make ⇒ ls -l ../run/sol2
>$ cd ../src3 ⇒ make ⇒ ls -l ../run/sol3
```

OpenMP版（Solverのみ）（F・C）

```
>$ cd ~/pfem/pfem3d/src1
>$ make
>$ cd ../run
>$ ls sol1
     sol1

>$ cd ../pmesh
並列メッシュ生成

>$ cd ../run
go1.shを編集

>$ pjsub go1.sh
```

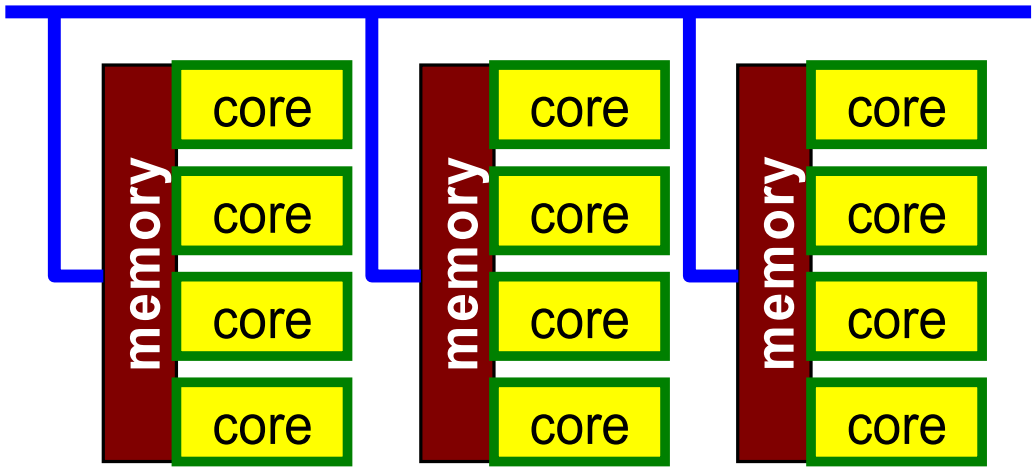
Makefile（Fortran, C）

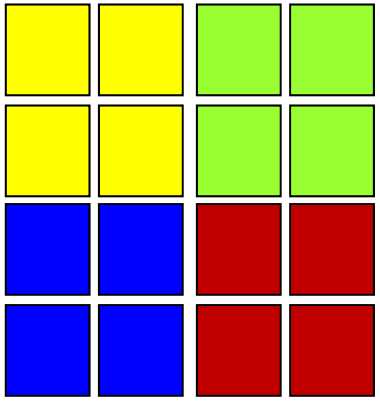
```
F90       =  mpifrtpx
F90LINKER = $(F90)
LIB_DIR   =
INC_DIR   =
OPTFLAGS  = -Kfast,openmp
FFLAGS = $(OPTFLAGS)
FLIBS =
F90LFLAGS=
#
TARGET = ../run/sol1
default: $(TARGET)
OBJS =¥
pfem_util.o ¥
...
pfem_finalize.o output_ucd.o

$(TARGET):  $(OBJS)
        $(F90LINKER) $(OPTFLAGS)
-o $(TARGET) $(OBJS) $(F90LFLAGS)
clean:
        /bin/rm -f *.o $(TARGET)
*~ *.mod
.f.o:
        $(F90) $(FFLAGS)
$(INC_DIR) -c  $*.f
.f90.o:
        $(F90) $(FFLAGS)
$(INC_DIR) -c  $*.f90
.SUFFIXES: .f90 .f
```

```
CC     = mpifccpx
LIB_DIR=
INC_DIR=
OPTFLAGS= -Kfast,openmp
LIBS =
LFLAGS=
#
TARGET = ../run/sol1
default: $(TARGET)
OBJS =¥
        test1.o¥
...
        util.o

$(TARGET):  $(OBJS)
        $(CC) $(OPTFLAGS) -o $@
$(OBJS) $(LFLAGS)
.c.o:
        $(CC) $(OPTFLAGS) -c
$*.c
clean:
        /bin/rm -f *.o $(TARGET)
*~ *.mod
```

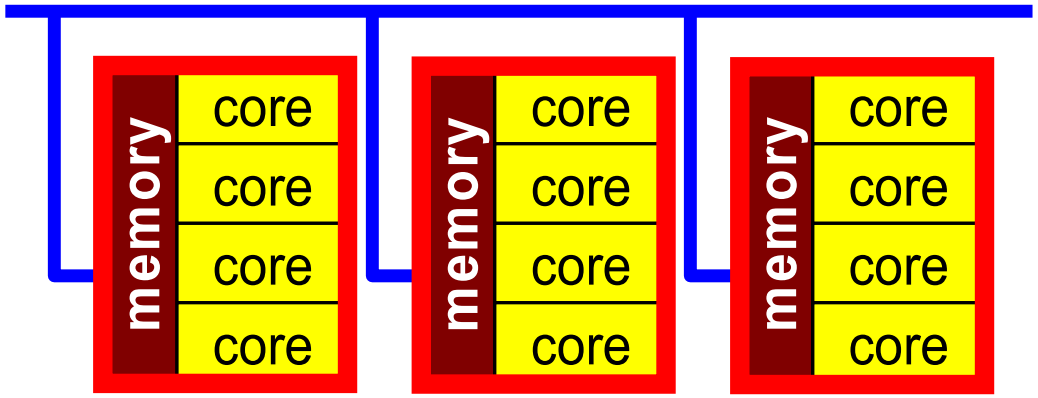
Flat MPI vs. Hybrid

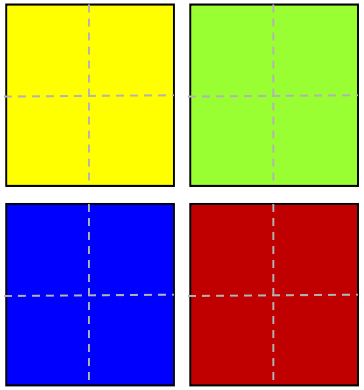
Flat-MPI：Each Core -> Independent

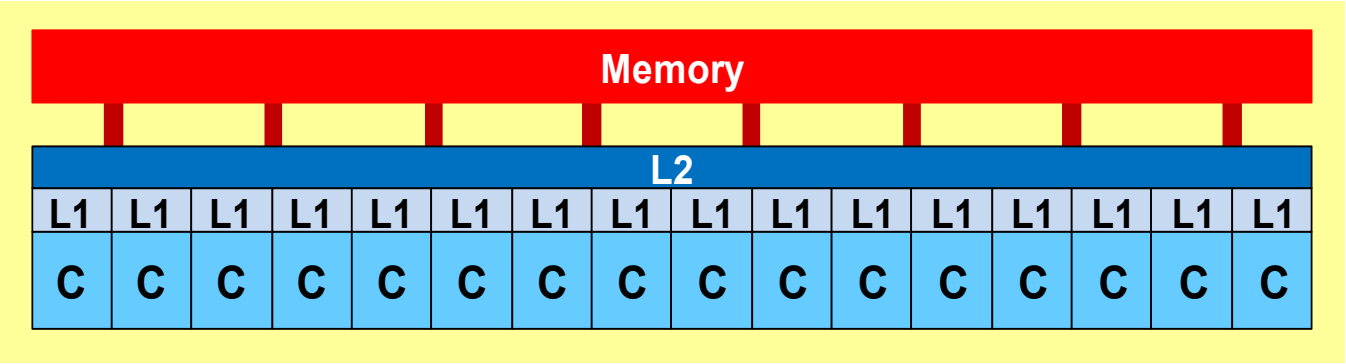




Hybrid：Hierarchal Structure







HB M x N

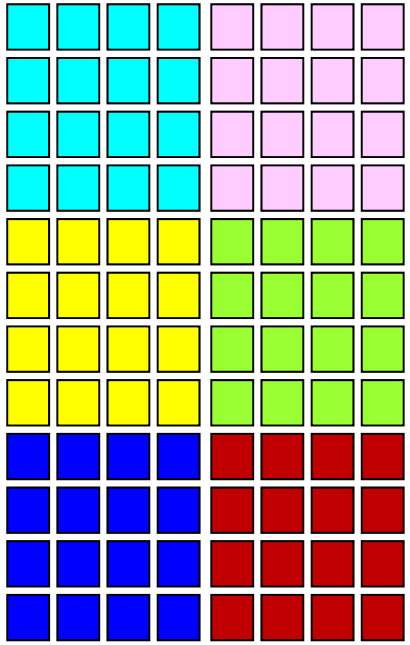
Number of OpenMP threads per a single MPI process

Number of MPI process per a single node

並列プログラミングモデルによって各プロセスの受け持つデータの量は変わる

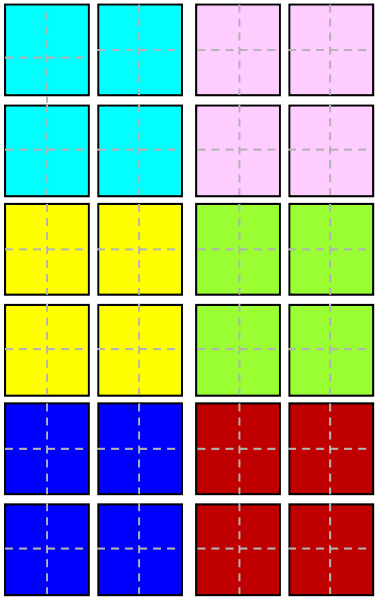
分散メッシュの数も各サイズも変わる

example: 6 nodes, 96 cores



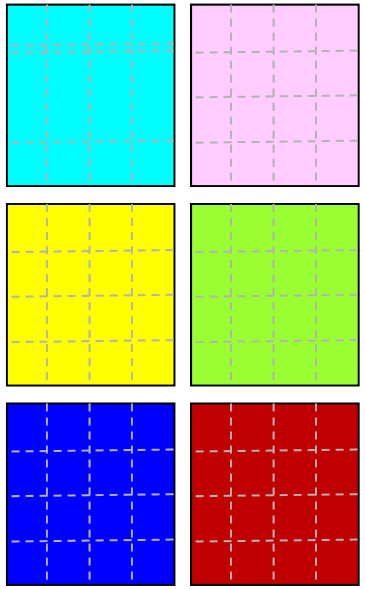
Flat MPI

```
128  192  64
  8   12   1
pcube
```



HB 4x4

```
128  192  64
  4    6   1
pcube
```



HB 16x1

```
128  192  64
  2    3   1
pcube
```

実行スクリプト（1/2） go0.sh
環境変数：OMP_NUM_THREADS

Flat MPI (go.sh)

```
#PJM -L "node=6"
#PJM -L "elapse=00:05:00"
#PJM -j
#PJM -L "rscgrp=school“
#PJM -o "test.lst"
#PJM --mpi "proc=96"

mpiexec ./sol

rm wk.*
```

Hybrid 16×1

```
#!/bin/sh
#PJM -L "node=6"
#PJM -L "elapse=00:05:00"
#PJM -j
#PJM -L "rscgrp=school“
#PJM -o "test.lst"
#PJM --mpi "proc=6"

export OMP_NUM_THREADS=16
mpiexec ./sol1

rm wk.*
```

実行スクリプト（2/2）go0.sh
環境変数：OMP_NUM_THREADS

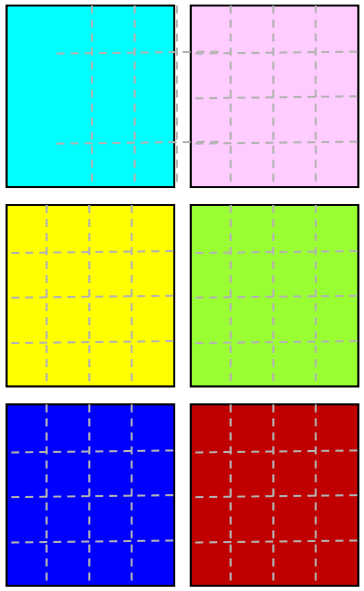
Hybrid 4×4

```
#!/bin/sh
#PJM -L "node=6"
#PJM -L "elapse=00:05:00"
#PJM -j
#PJM -L "rscgrp=school“
#PJM -o "test.lst"
#PJM --mpi "proc=24"

export OMP_NUM_THREADS=4
mpiexec ./sol1

rm wk.*
```

Hybrid 8×2

```
#!/bin/sh
#PJM -L "node=6"
#PJM -L "elapse=00:05:00"
#PJM -j
#PJM -L "rscgrp=school“
#PJM -o "test.lst"
#PJM --mpi “proc=12"

export OMP_NUM_THREADS=8
mpiexec ./sol1

rm wk.*
```

とりあえずHB 16×1でやってみよう



HB 16x1

```
128 192 64
  2   3  1
pcube
```

go0.sh

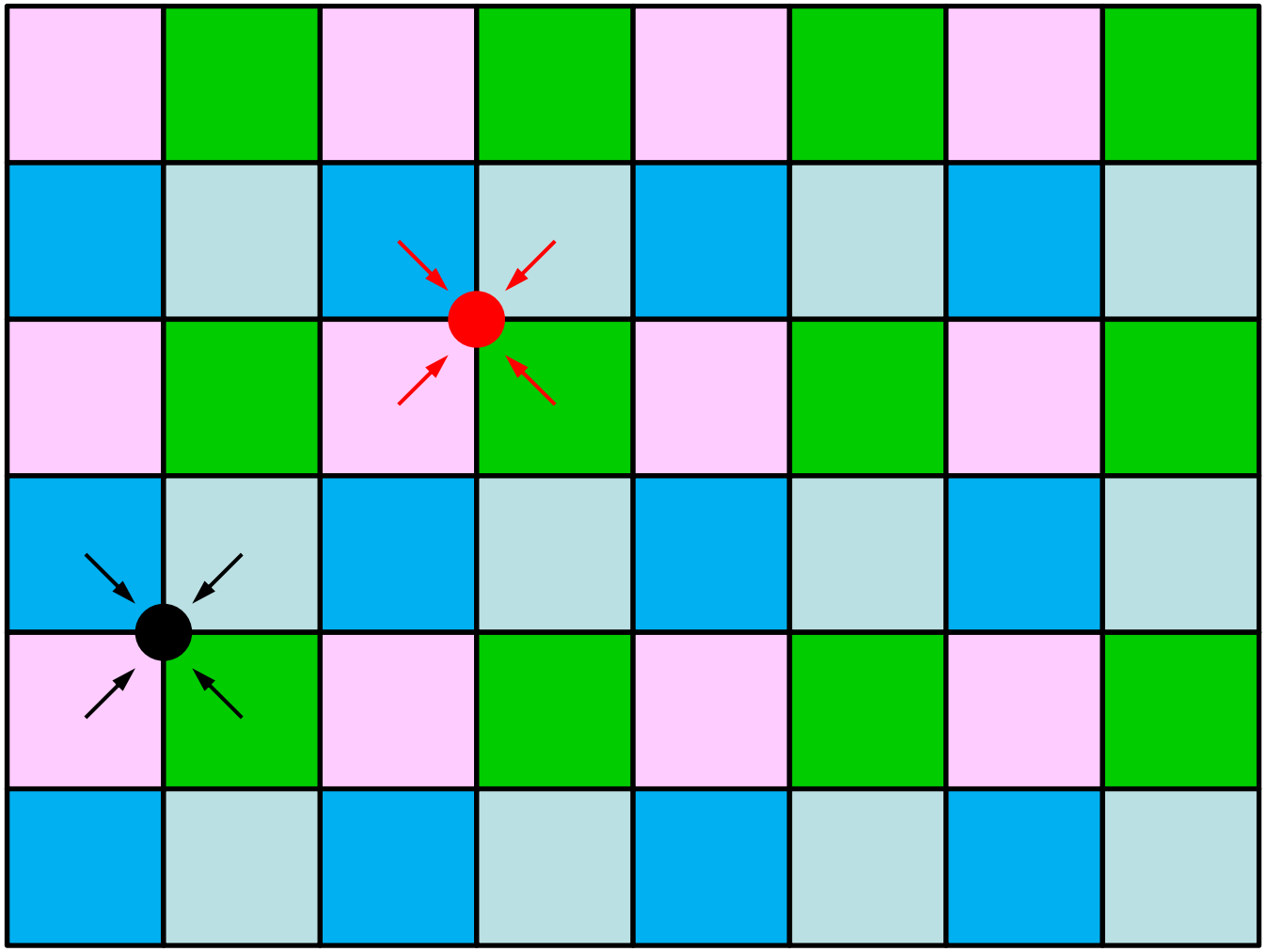
```
#!/bin/sh
#PJM -L "node=6"
#PJM -L "elapse=00:05:00"
#PJM -j
#PJM -L "rscgrp=school“
#PJM -o "test.lst"
#PJM --mpi "proc=6"

export OMP_NUM_THREADS=16
mpiexec ./sol1

rm wk.*
```

pfem3dのスレッド並列化

- CG法
  - ほぼOpenMPの指示文（directive）を入れるだけで済む
  - 前処理がILU系になるとそう簡単ではない
- 行列生成部（mat_ass_main, mat_ass_bc）
  - 複数要素から同時に同じ節点に足し込むことを回避する必要がある
    - 計算結果が変わってしまう
    - 同時に書き込もうとして計算が止まる場合もある（環境依存）
  - 色分け（Coloring）
    - 色内に属する要素が同じ節点を同時に更新しないように色分けすれば，同じ色内の要素の処理は並列にできる
    - 現在の問題は規則正しい形状なので，8色に塗り分けられる（1節点を共有する要素数は最大8，要素内節点数8）
    - 色分け部分の計算時間が無視できない：並列化困難

行列生成部スレッド並列化

同じ色の要素の処理は並列に実行可能



要素色分け(1/2)

```
allocate (ELMCOLORindex(0:NP))          各色に含まれる要素数（一次元圧縮配列）
allocate (ELMCOLORitem (ICELTOT))  色の順番に並び替えた要素番号
if (allocated (IWKX)) deallocate (IWKX)
allocate (IWKX(NP,3))

IWKX= 0
icou= 0
do icol= 1, NP
  do i= 1, NP
    IWKX(i,1)= 0
  enddo
  do icel= 1, ICELTOT
    if (IWKX(icel,2).eq.0) then
      in1= ICELNOD(icel,1)
      in2= ICELNOD(icel,2)
      in3= ICELNOD(icel,3)
      in4= ICELNOD(icel,4)
      in5= ICELNOD(icel,5)
      in6= ICELNOD(icel,6)
      in7= ICELNOD(icel,7)
      in8= ICELNOD(icel,8)

      ip1= IWKX(in1,1)
      ip2= IWKX(in2,1)
      ip3= IWKX(in3,1)
      ip4= IWKX(in4,1)
      ip5= IWKX(in5,1)
      ip6= IWKX(in6,1)
      ip7= IWKX(in7,1)
      ip8= IWKX(in8,1)
```

要素色分け(2/2)

```
            isum= ip1 + ip2 + ip3 + ip4 + ip5 + ip6 + ip7 + ip8
            if (isum.eq.0) then          要素各節点が同色内でアクセスされていない
              icou= icou + 1             カウンターを1つ増やす
              IWKX(icol,3)= icou         各色内に含まれる要素数の累積
              IWKX(icel,2)= icol
              ELMCOLORitem(icou)= icel   icou番目の要素をicelとする

              IWKX(in1,1)= 1             各節点は同色内でアクセス不可，Flag立てる
              IWKX(in2,1)= 1
              IWKX(in3,1)= 1
              IWKX(in4,1)= 1
              IWKX(in5,1)= 1
              IWKX(in6,1)= 1
              IWKX(in7,1)= 1
              IWKX(in8,1)= 1
              if (icou.eq.ICELTOT) goto 100  全要素が色づけされたら終了
            endif
          endif
        enddo
      enddo

100   continue
      ELMCOLORtot= icol                  色数
      IWKX(0          ,3)= 0
      IWKX(ELMCOLORtot,3)= ICELTOT

      do icol= 0, ELMCOLORtot
        ELMCOLORindex(icol)= IWKX(icol,3)
      enddo
```

スレッド並列化された マトリクス生成部

```
      do icol= 1, ELMCOLORtot
!$omp parallel do private (icel0,icel)                                    &
!$omp&            private (in1,in2,in3,in4,in5,in6,in7,in8)               &
!$omp&            private (nodLOCAL,ie,je,ip,jp,kk,iiS,iiE,k)             &
!$omp&            private (DETJ,PNX,PNY,PNZ,QVC,QV0,COEFij,coef,SHi)      &
!$omp&            private (PNXi,PNYi,PNZi,PNXj,PNYj,PNZj,ipn,jpn,kpn)     &
!$omp&            private (X1,X2,X3,X4,X5,X6,X7,X8)                       &
!$omp&            private (Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8)                       &
!$omp&            private (Z1,Z2,Z3,Z4,Z5,Z6,Z7,Z8,CONDO)
      do icel0= ELMCOLORindex(icol-1)+1, ELMCOLORindex(icol)
        icel= ELMCOLORitem(icel0)
        in1= ICELNOD(icel,1)
        in2= ICELNOD(icel,2)
        in3= ICELNOD(icel,3)
        in4= ICELNOD(icel,4)
        in5= ICELNOD(icel,5)
        in6= ICELNOD(icel,6)
        in7= ICELNOD(icel,7)
        in8= ICELNOD(icel,8)
...
```

OpenMP版（行列生成部並列化）
Fortranのみ

```
>$ cd ~/pFEM/pfem3d/src2
>$ make
>$ cd ../run
>$ ls sol2
     sol2

>$ cd ../pmesh
並列メッシュ生成

>$ cd ../run
go1.shを編集

>$ pjsub go1.sh
```

計算例(1/2)@東大

512×384×256= 50,331,648 節点
12ノード，192コア
64^3=262,144節点／コア

```
512  384  256
ndx  ndy  ndz
pcube
```

	ndx,ndy,ndz (#MPI proc.)	Iter's	sec.	
Flat MPI	8 6 4 (192)	1240	73.9	
HB 1×16	8 6 4 (192)	1240	73.6	-Kopenmpでコンパイル，OMP_NUM_THREADS=1
HB 2×8	4 6 4 (96)	1240	78.8	
HB 4×4	4 3 4 (48)	1240	80.3	
HB 8×2	4 3 2 (24)	1240	81.1	
HB 16×1	2 3 2 (12)	1240	81.9	

計算例(2/2)@東大

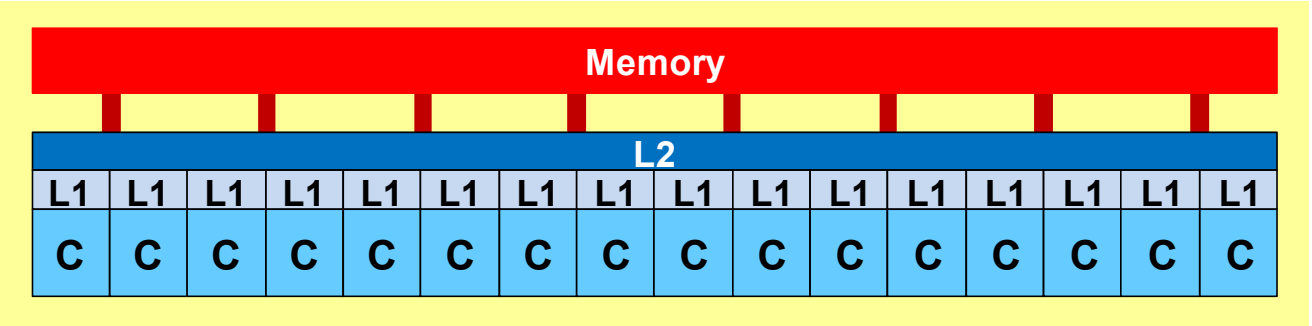
50,331,648 節点，12ノード，192コア
12 MPIプロセス，スレッド数変化
64^3=262,144節点／コア

```
512  384  256
  2    3    2
pcube
```

OMP_NUM_THREADS	sec.	Speed-Up
1	1056.2	1.00
2	592.5	1.78
4	289.8	3.64
8	148.1	7.13
12	103.6	10.19
16	81.9	12.90
Flat MPI, 1 proc./node	1082.4	-

Flat MPI vs. Hybrid

- アプリケーション，問題サイズ，HW特性に依存
- 疎行列ソルバーの場合，計算ノード数が少なければFlat MPIの方が概して性能がよい
  - メモリへの負担，メモリ競合
- ノード数が増えるとHybridの性能が比較的良くなる
  - MPIプロセス数が小さい
- Intel Xeon Phi/MICのようなメニィコア環境（1ノード240スレッド）ではFlat MPIは現実的ではない
  - MPI：一定のメモリを消費



HB 16×1，1ノード

ソルバー計算時間（反復回数）

```
NX   NX   NX
 1    1    1
pcube
```

	NX=128 2,097,152 nodes	NX=129 2,146,689 nodes
初期解	14.7 (392)	6.28 (396)

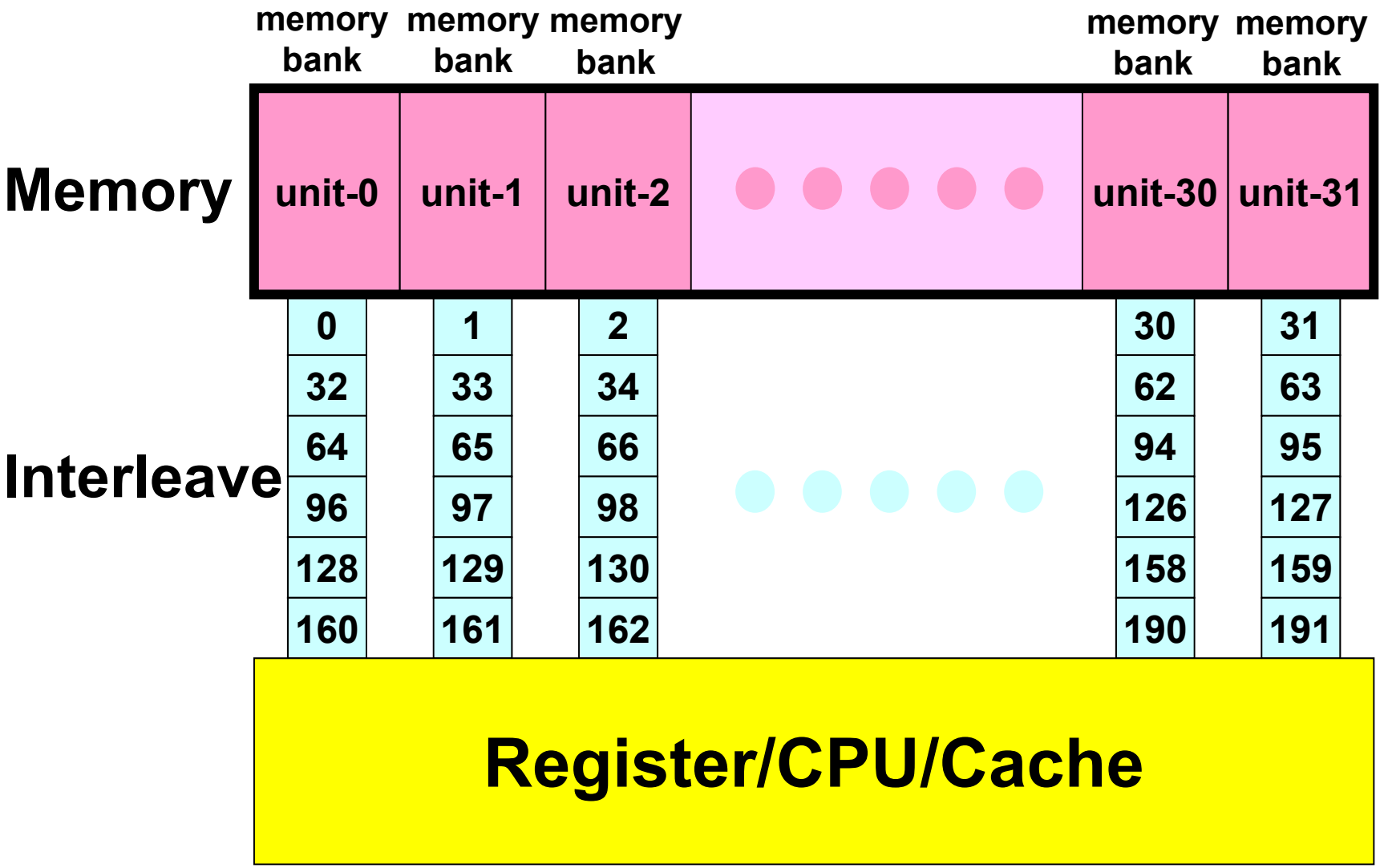
- このような差異が生じる要因
  - バンクコンフリクト
  - キャッシュスラッシング

Memory Interleaving と Bank Conflict

- メモリインターリーブ（memory interleaving）
  - メモリのデータ転送を高速化する技術の一つ。複数のメモリバンクに同時並行で読み書きを行なうことにより高速化を行なう手法。
- メモリーバンク，バンク（memory bank）
  - メモリコントローラがメモリを管理するときの単位となる，一定の容量を持ったメモリの集合。
    - 通常は 2^n個の独立したモジュール
  - 同一のメモリバンクに対しては，同時には1つの読み出しまたは書き込みしかできないため，連続して同じグループのデータをアクセスすると性能が低下。
  - 例えば，一つの配列を32要素飛び(又は32の倍数要素飛び)にアクセスすると，バンク競合（コンフリクト）が発生する。これを防ぐためには，配列宣言を奇数になるように変更するか，ループを入れ換えて，配列のアクセスが連続するように変更する。

Bank Conflict

例：32とびにアクセスすると1つのバンクしか使えない



Bank Conflictの回避

×

```
REAL*8 A(32,10000)

k= N
do i= 1, 10000
   A(k,i)= 0.d0
enddo
```

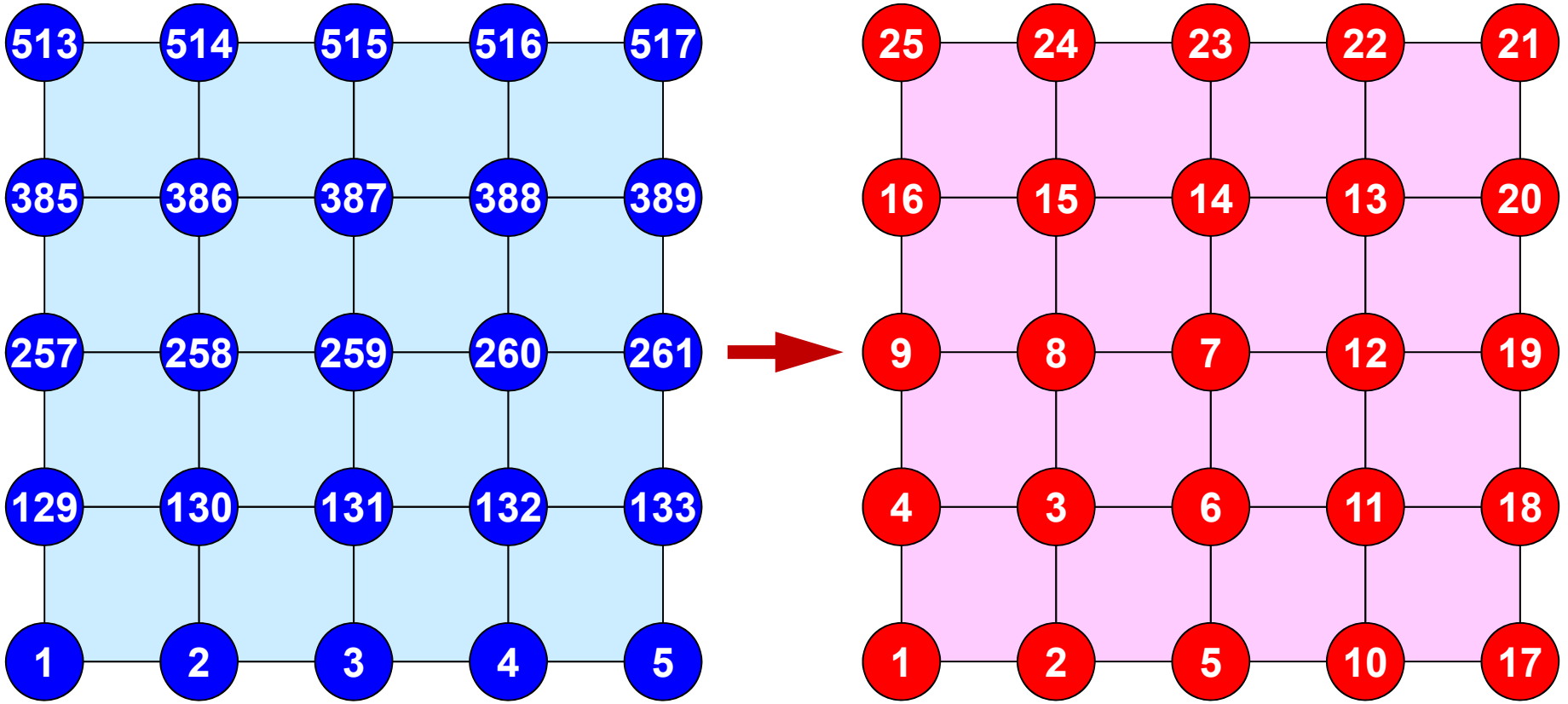
○

```
REAL*8 A(33,10000)

k= N
do i= 1, 10000
   A(k,i)= 0.d0
enddo
```

- 2^nを避ける（32だけがダメというわけではない）
- NX=128の場合は2^nとならざるを得ない：規則正しい形状
  - 非対角成分の2^nおきに参照することになってしまう

Bank Conflictの回避

- リオーダリング（並び替え）
- CM法（Cuthill-Mckee）の採用
  - Hyperplane，Hyperlineとも呼ばれる
  - キャッシュにも当たりやすくなる



CM法（Cuthill-McKee），RCM法（Reverse Cuthill-McKee）

- fill-inを減らす
- マトリクスのバンド幅を減らす，プロフィルを減らす
- 並列性

  - β_i: i 行における非ゼロ成分の列番号の最大値を k とするとき，$\beta_i = k - i$
  - バンド幅: β_iの最大値
  - プロフィル: β_iの和
  - バンド幅，プロフィル，Fill-inともに少ない方が都合が良い（計算量，前処理に使う場合の収束性）

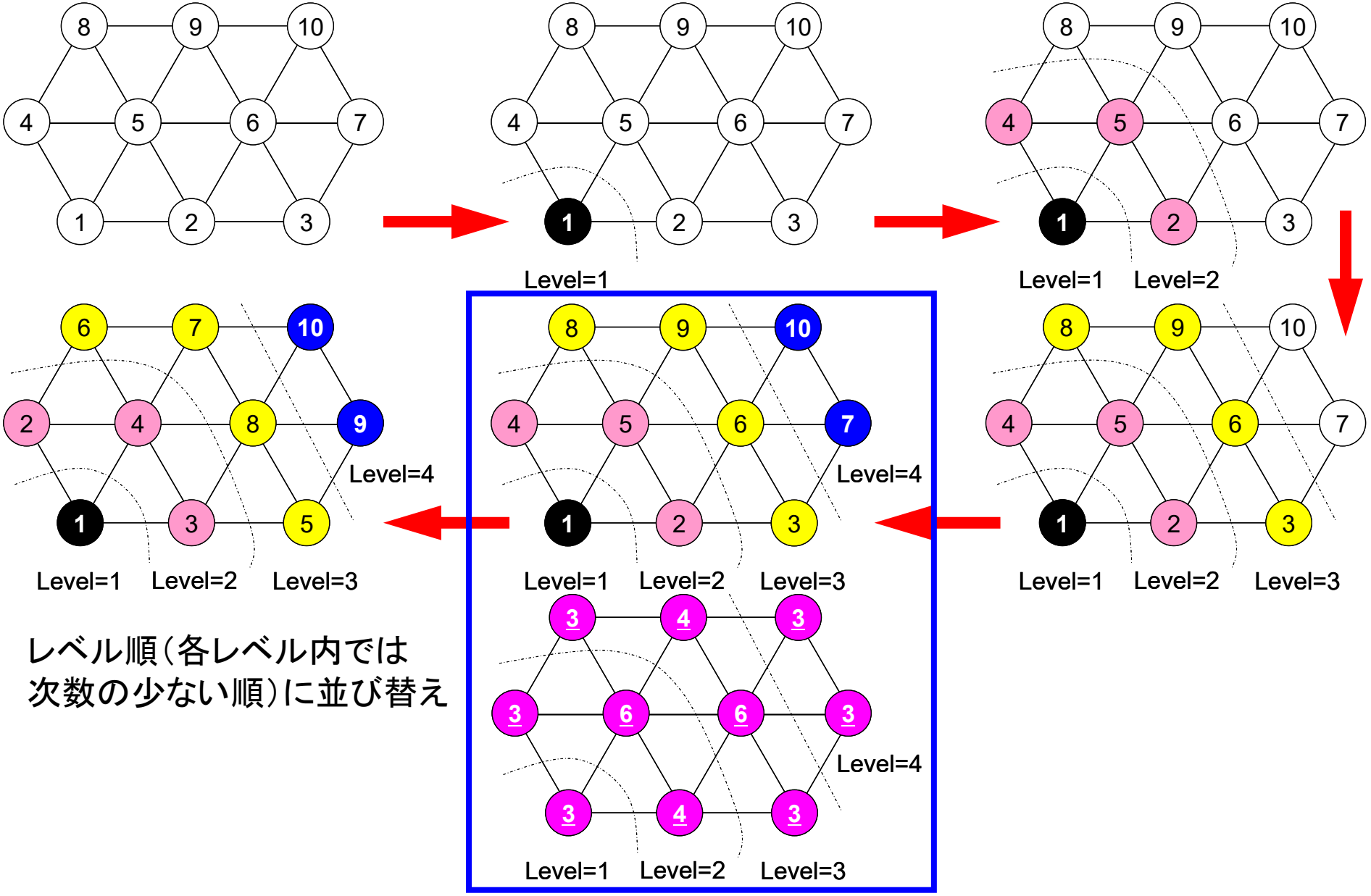
CM法の基本的なアルゴリズム

① 各点に隣接する点の数を「次数（degree）」，最小次数の点を「レベル=1」の点とする。

② 「レベル=k-1」に属する点に隣接する点を「レベル=k」とする。全ての点にレベルづけがされるまで「k」を1つずつ増やして繰り返す。

③ 全ての点がレベルづけされたら，レベルの順番に再番号づけする（各レベル内では「次数」の番号が少ない順）。

Cuthill-Mckee Ordering（CM法）の例



RCM（Reverse CM）

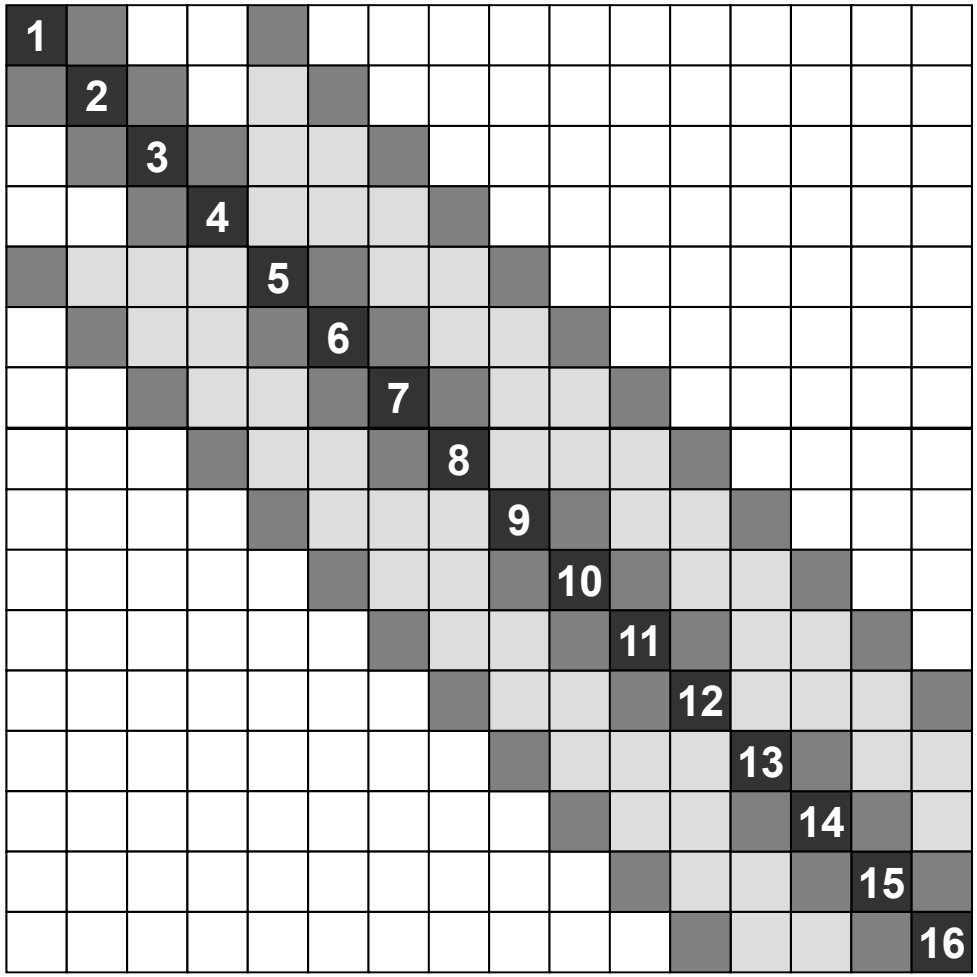
- まずCMの手順を実行
  - 次数（degree）の計算
  - 「レベル（k（k≧2））」の要素の選択
  - 繰り返し，再番号付け

- 再々番号付け
  - CMの番号付けを更に逆順にふり直す
  - Fill-inがCMの場合より少なくなる

初期状態

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

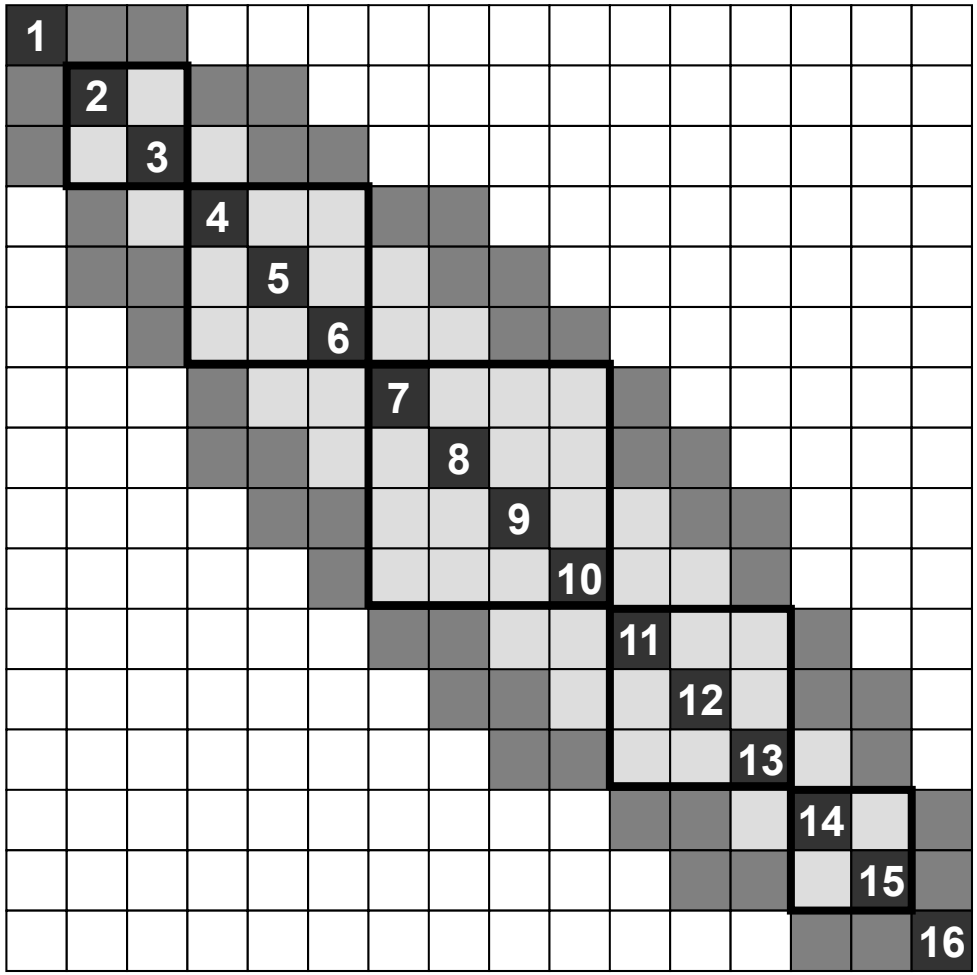
バンド幅　　4
プロフィル　51
Fill-in　　54



■非ゼロ成分，■Fill-in

CMオーダリング

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



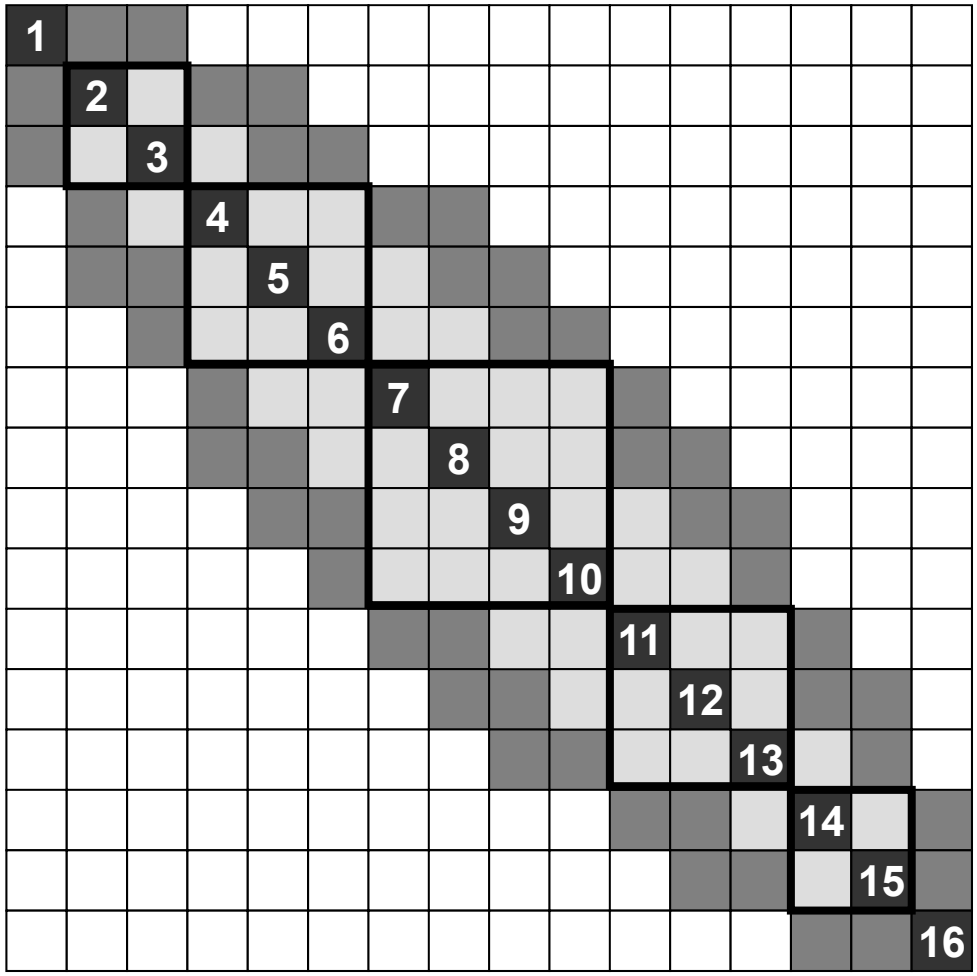
バンド幅　4
プロフィル　46
Fill-in　44

■非ゼロ成分，■Fill-in

RCMオーダリング

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

バンド幅　　4
プロフィル　46
Fill-in　　44



■非ゼロ成分，■Fill-in

CM法の実装(1/3)

```
!C
!C +-------+
!C | INIT. | 次数最小節点探索，新IDを1とする
!C +-------+
!C===
      allocate (IW(NP))
      IW = 0

       INmin= NP
      NODmin= 0

      do i= 1, N
        if (INLU(i).lt.INmin) then
          INmin = INLU(i)
          NODmin= i
        endif
      enddo
 200  continue


      if (NODmin.eq.0) NODmin= 1

      IW(NODmin)= 1

      NEWtoOLD(1     )= NODmin
      OLDtoNEW(NODmin)= 1

      icol= 1
!C===
```

NODmin:　次数最小の節点ID（旧ID）

OLDtoNEW(旧ID)=新ID
NEWtoOLD(新ID)=旧ID

IW(旧ID)=CM法のレベル

```
!C
!C +---------------+
!C | CM-redordering |
!C +---------------+
!C===
      icouG= 1
      do icol= 2, N
        do i= 1, N
          if (IW(i).eq.icol-1) then
            do k= 1, INLU(i)
              in= IALU(i,k)
              if (IW(in).eq.0) then
                IW(in)= icol
                icouG= icouG + 1
              endif
            enddo
          endif
        enddo
        if (icouG.eq.N) exit
      enddo

      NCOLORtot= icol
      icoug    = 0
      do ic= 1, NCOLORtot
        do i= 1, N
          if (IW(i).eq.ic) then
            icoug= icoug + 1
            NEWtoOLD(icoug)= i
            OLDtoNEW(i    )= icoug
          endif
        enddo
      enddo
!C===
```

CM法の実装(2/3)

```
!C
!C +-------+
!C | INIT. |
!C +-------+
!C===
      allocate (IW(NP))
      IW = 0

       INmin= NP
      NODmin= 0

      do i= 1, N
        if (INLU(i).lt.INmin) then
          INmin = INLU(i)
          NODmin= i
        endif
      enddo
 200  continue


      if (NODmin.eq.0) NODmin= 1

      IW(NODmin)= 1

      NEWtoOLD(1     )= NODmin
      OLDtoNEW(NODmin)= 1

      icol= 1
!C===
```

NODmin:　次数最小の節点ID（旧ID）

OLDtoNEW(旧ID)=新ID
NEWtoOLD(新ID)=旧ID

IW(旧ID)=CM法のレベル

```
!C
!C +---------------+
!C | CM-redordering |
!C +---------------+
!C===
      icouG= 1                  レベル付けされた節点数
      do icol= 2, N
        do i= 1, N
          if (IW(i).eq.icol-1) then
            do k= 1, INLU(i)
              in= IALU(i,k)
              if (IW(in).eq.0) then
                IW(in)= icol
                icouG= icouG + 1
              endif
            enddo
          endif
        enddo
        if (icouG.eq.N) exit
      enddo

      NCOLORtot= icol
      icoug    = 0
      do ic= 1, NCOLORtot
        do i= 1, N
          if (IW(i).eq.ic) then
            icoug= icoug + 1
            NEWtoOLD(icoug)= i
            OLDtoNEW(i    )= icoug
          endif
        enddo
      enddo
!C===
```

CM法の実装(3/3)

```
!C
!C +-------+
!C | INIT. |
!C +-------+
!C===
      allocate (IW(NP))
      IW = 0

       INmin= NP
      NODmin= 0

      do i= 1, N
        if (INLU(i).lt.INmin) then
          INmin = INLU(i)
          NODmin= i
        endif
      enddo
 200  continue


      if (NODmin.eq.0) NODmin= 1

      IW(NODmin)= 1

      NEWtoOLD(1     )= NODmin
      OLDtoNEW(NODmin)= 1

      icol= 1
!C===
```

NODmin:　次数最小の節点ID（旧ID）

OLDtoNEW(旧ID)=新ID
NEWtoOLD(新ID)=旧ID

IW(旧ID)=CM法のレベル

```
!C
!C +---------------+
!C | CM-redordering |
!C +---------------+
!C===
      icouG= 1                レベル付けされた節点数
      do icol= 2, N
        do i= 1, N
          if (IW(i).eq.icol-1) then
            do k= 1, INLU(i)
              in= IALU(i,k)
              if (IW(in).eq.0) then
                IW(in)= icol
                icouG= icouG + 1
              endif
            enddo
          endif
        enddo
        if (icouG.eq.N) exit
      enddo

      NCOLORtot= icol         レベル順に再番号付け
      icoug    = 0            次数の大小は考慮せず
      do ic= 1, NCOLORtot
        do i= 1, N
          if (IW(i).eq.ic) then
            icoug= icoug + 1
            NEWtoOLD(icoug)= i
            OLDtoNEW(i    )= icoug
          endif
        enddo
      enddo
!C===
```

HB 16×1，1ノード

ソルバー計算時間（反復回数）

```
NX   NX   NX
 1    1    1
pcube
```

	NX=128 2,097,152 nodes	NX=129 2,146,689 nodes
初期解	14.7 (392)	6.28 (396)
CM法適用	6.33 (392)	6.26 (396)

- だいぶ改善されたが，それでもN=128^3の場合がN=129^3より明らかに遅い

キャッシュスラッシング（thrashing）

- FX10：32KBのL1Dキャッシュ⇒コア毎，2-way
  - n-way set associative（n群連想記憶式）
  - キャッシュ全体がn個のバンクに分割
  - 各バンクがキャッシュラインに分割される
    - キャッシュライン数，ラインサイズ（FX10の場合128byte）は2のべき乗
- 実はこの「2-way」が曲者
  - N（総節点数（内点数））が2のべき乗の場合，共役勾配法の下記の箇所で，WW(i,P)， WW(i,Q)， WW(i,R)の3キャッシュラインが競合し，性能が著しく低下⇒キャッシュスラッシング
    - R=1， P=2， Q=3
    - X(i)は影響受けず

```
!$omp parallel do private(i)
      do i= 1, N
         X(i)  = X (i)   + ALPHA * WW(i,P)
        WW(i,R)= WW(i,R) - ALPHA * WW(i,Q)
      enddo
```

回避法

- ループを分割すると，同一ループ内で競合するライン数が2以内に抑えられる(方策1)

```
!$omp parallel do private(i)
      do i= 1, N
         X(i)  = X (i)   + ALPHA * WW(i,P)
      enddo

!$omp parallel do private(i)
      do i= 1, N
        WW(i,R)= WW(i,R) - ALPHA * WW(i,Q)
      enddo
```

- Nが2のべき乗のときにNに適当な数(例えば64，128)を加えて配列のサイズが2のべき乗にならないようにして，競合を回避(方策2)
  - WS， WR， Xにはこの操作は不要

```
N2=128
allocate (WW(NP+N2,4),WR(NP),WS(NP))
```

HB 16×1，1ノード

ソルバー計算時間（反復回数）

```
NX   NX   NX
 1    1    1
pcube
```

	NX=128 2,097,152 nodes	NX=129 2,146,689 nodes
初期解	14.7 (392)	6.28 (396)
CM法適用	6.33 (392)	6.26 (396)
+方策2	6.08 (392)	6.24 (396)

- これで大体計算量と整合した計算時間になる
- そもそもキャッシュが2-wayなのが問題
- FX100（後継機種）では4-wayになっている（世の中の普通のプロセッサは4-way，8-way）

性能改良版：CM+スラッシング回避 Fortranのみ

```
>$ cd ~/pFEM/pfem3d/src3
>$ make
>$ cd ../run
>$ ls sol3
     sol3

>$ cd ../pmesh
並列メッシュ生成

>$ cd ../run
go1.shを編集

>$ pjsub go1.sh
```