OpenMP＋ハイブリッド並列化

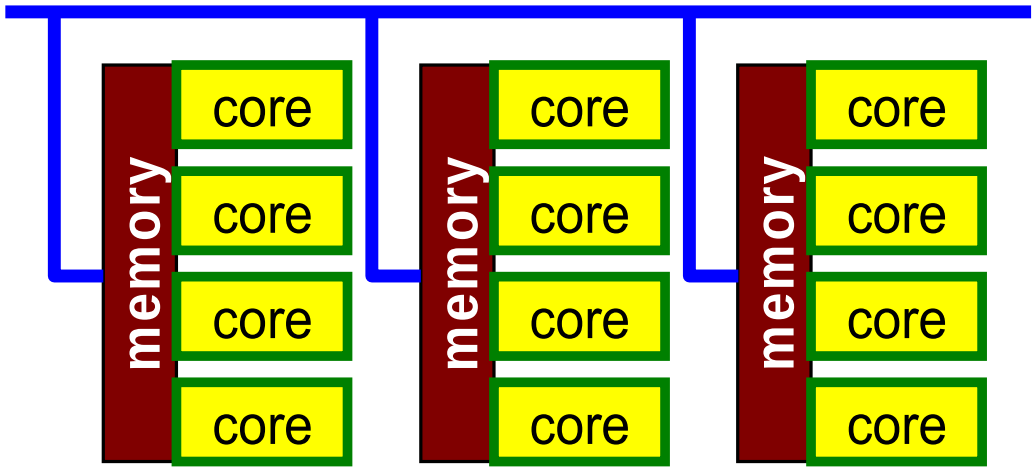
中島研吾
東京大学情報基盤センター

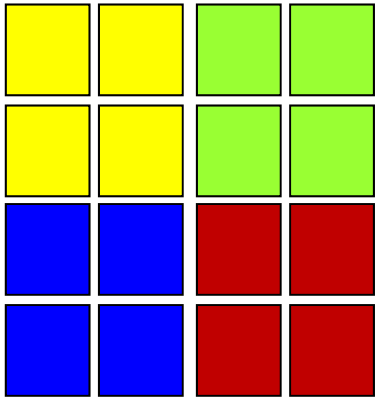
Hybrid並列プログラミング

- 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

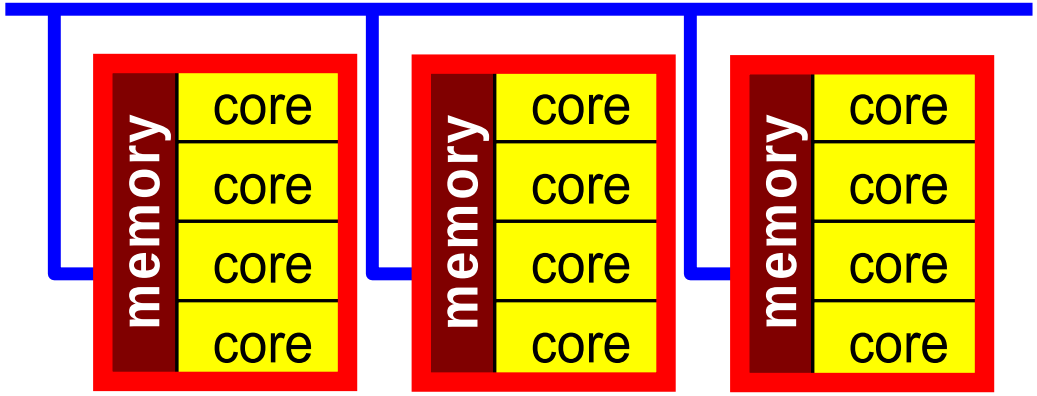
Flat MPI vs. Hybrid

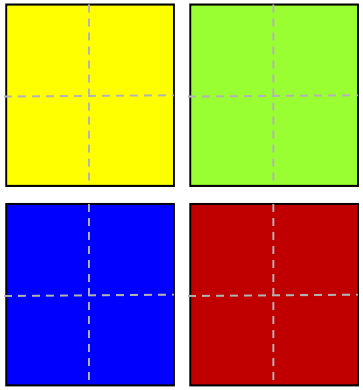
Flat-MPI：Each Core -> Independent

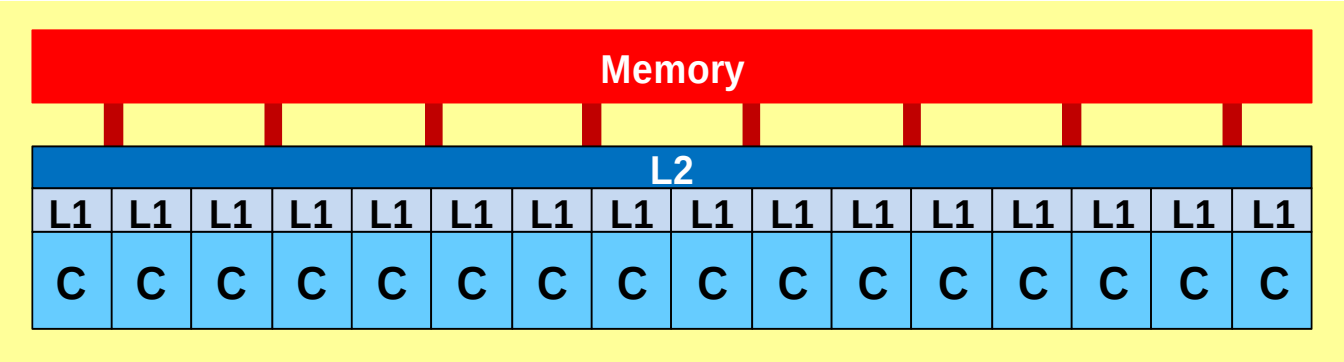




Hybrid：Hierarchal Structure







HB M x N

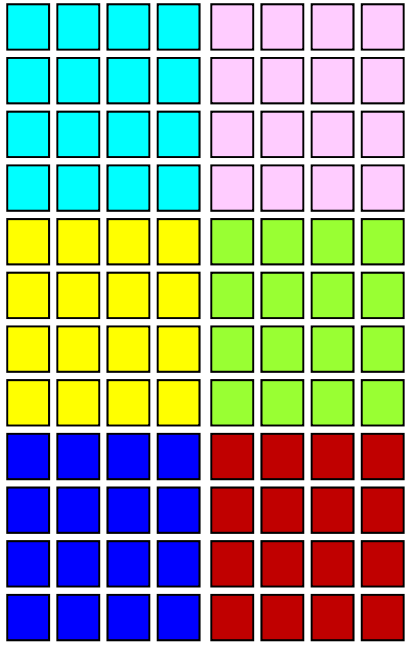
Number of OpenMP threads per a single MPI process

Number of MPI process per a single node

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

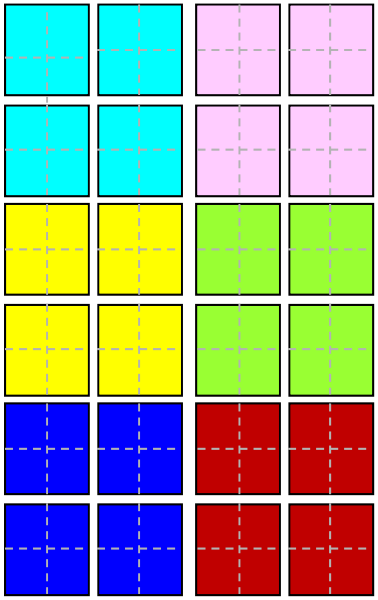
分散メッシュの数も各サイズも変わる

example: 6 nodes, 96 cores



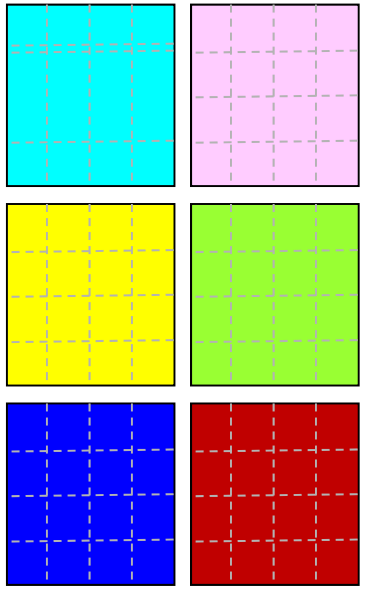
Flat MPI

```
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)
環境変数:OMP_NUM_THREADS

Flat MPI

```
#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 ./sol

rm wk.*
```

実行スクリプト(2/2)
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 ./sol

rm wk.*
```

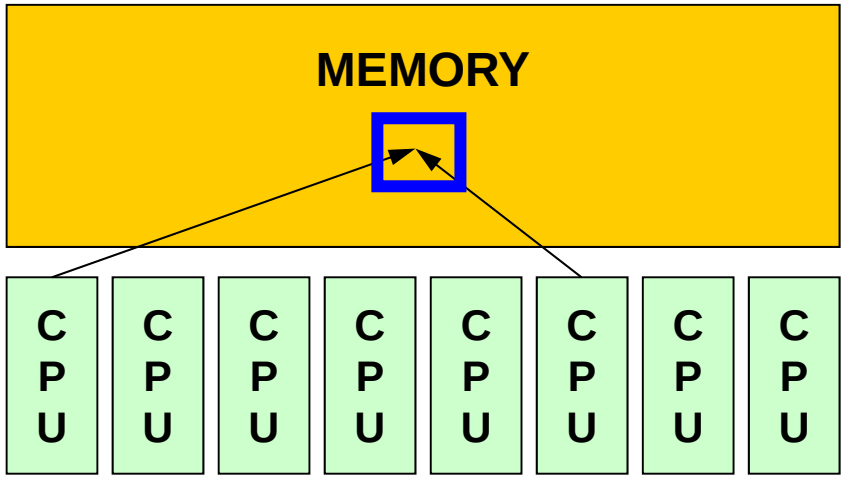
Hybrid 8×2

```
#!/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 ./sol

rm wk.*
```

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Spring School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。

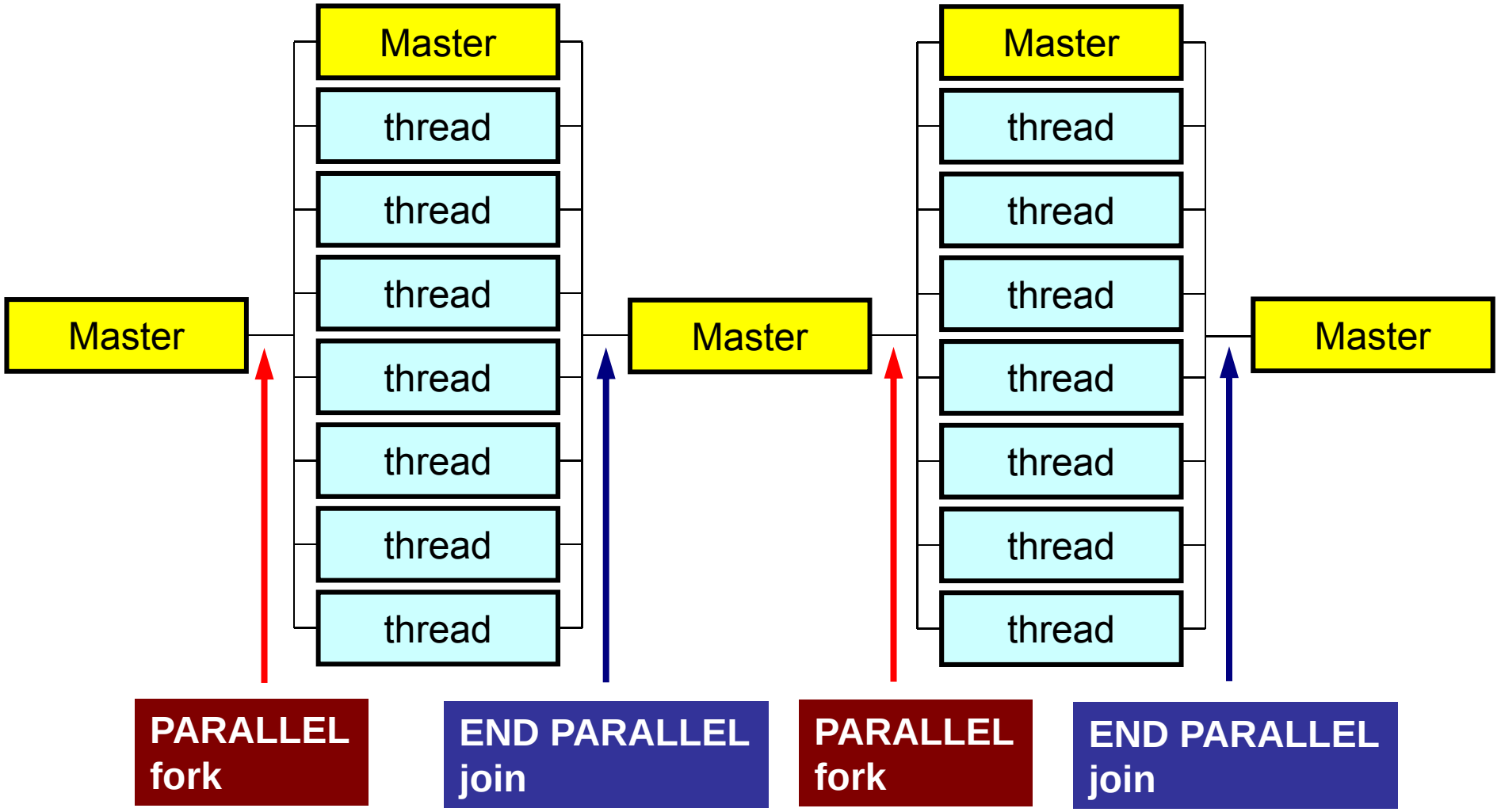
OpenMPとは（続き）

- 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

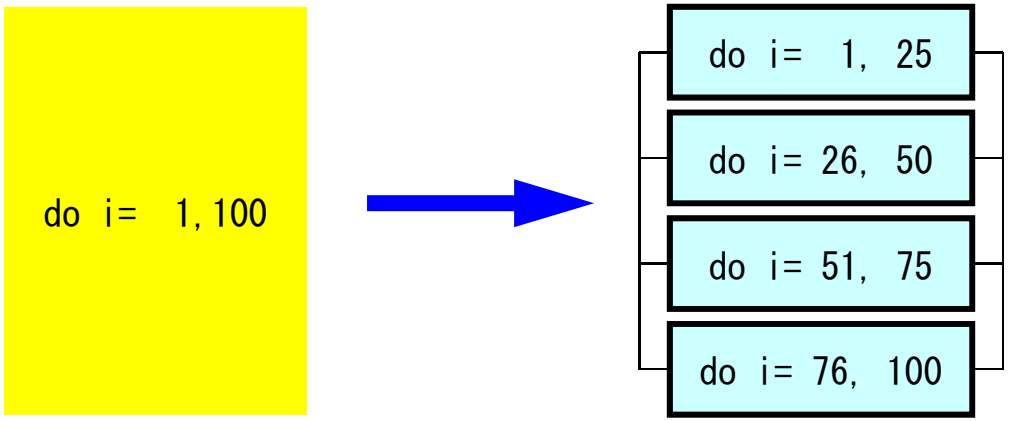
Fork-Join 型並列モデル



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）最新!
- 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
```

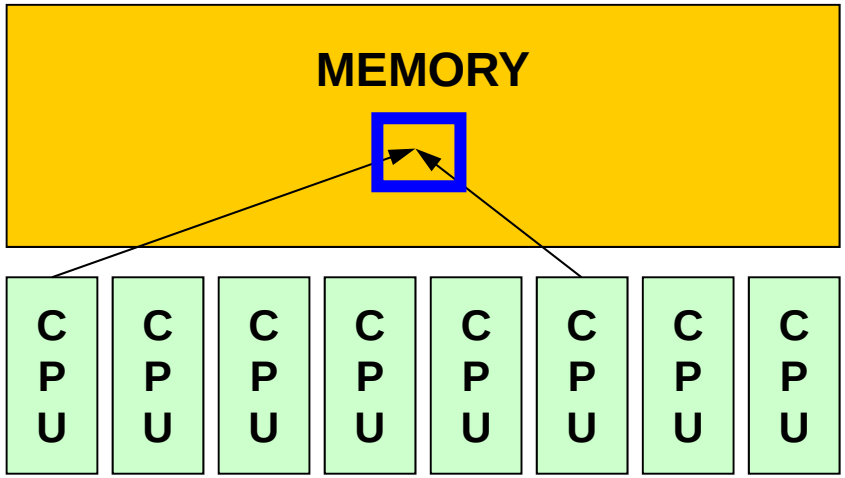
OpenMP/Direceives Matrix/Vector Products

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

OpenMPの特徴

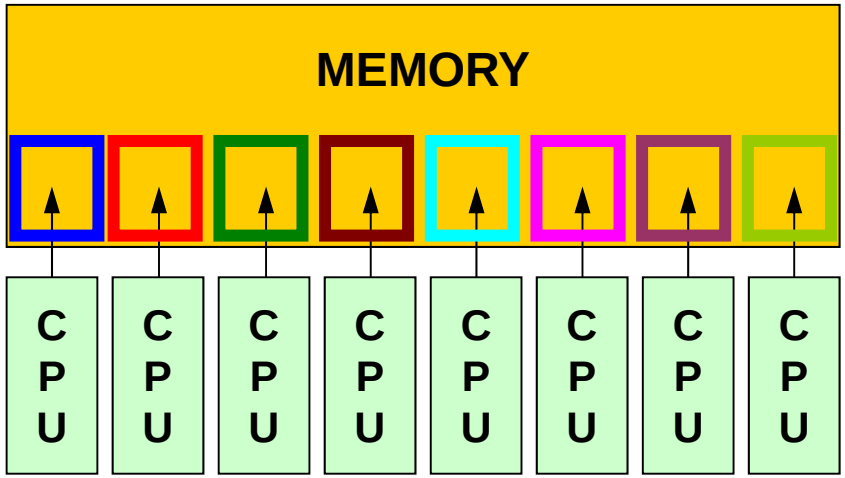
- ディレクティヴ（指示行）の形で利用
  - 挿入直後のループが並列化される
  - コンパイラがサポートしていなければ，コメントとみなされる
- 何も指定しなければ，何もしない
  - 「自動並列化」，「自動ベクトル化」とは異なる。
  - 下手なことをするとおかしな結果になる：ベクトル化と同じ
  - データ分散等（Ordering）は利用者の責任
- 共有メモリユニット内のプロセッサ数に応じて，「Thread」が立ち上がる
  - 「Thread」：MPIでいう「プロセス」に相当する。
  - 普通は「Thread数＝共有メモリユニット内プロセッサ数，コア数」であるが最近のアーキテクチ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

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

ファイルコピー
FX10

```
>$ cd ~/pFEM

>$ cp /home/ss/aics60/2014Summer/F/omp.tar .
>$ cp /home/ss/aics60/2014Summer/C/omp.tar .

>$ tar xvf omp.tar

>$ cd omp
```

実例：FORTRAN，C共通

```
>$ cd ~/pFEM/omp

>$ frtpx –Kfast,openmp test.f
>$ fccpx –Kfast,openmp test.c

>$ pjsub go.sh
```

実行

go.sh

```
#!/bin/sh
#PJM -L "node=1"
#PJM -L "elapse=00:10:00"
#PJM -L "rscgrp=school"
#PJM -j
#PJM -o "t0-08.lst"

export OMP_NUM_THREADS=8        スレッド数を変えられる

./a.out < INPUT.DAT
```

INPUT.DAT

```
N  nopt

N:    問題サイズ（ベクトル長）
nopt: First-touchの有無（0:無し，1:有り）
```

test.fの内容

- DAXPY
  - ベクトルとその定数倍の加算
- DOT
  - 内積
- OpenMPディレクティブ挿入の効果

test.f(1/3):初期化

```
      use omp_lib
      implicit REAL*8 (A-H,O-Z)
      real(kind=8), dimension(:), allocatable :: X, Y
      real(kind=8), dimension(:), allocatable :: Z1, Z2
      real(kind=8), dimension(:), allocatable :: Z3, Z4, Z5
      integer, dimension(0:2) :: INDEX
!C
!C +------+
!C | INIT |
!C +------+
!C===
      write (*,*) 'N, nopt ?'
      read  (*,*)  N, nopt

      allocate (X(N), Y(N), Z1(N), Z2(N), Z3(N), Z4(N), Z5(N))
      if (nopt.eq.0) then
        X = 1.d0
        Y = 1.d0
        Z1= 0.d0
        Z2= 0.d0
        Z3= 0.d0
        Z4= 0.d0
        Z5= 0.d0
       else
!$omp parallel do private (i)
        do i= 1, N
          X (i)= 0.d0
          Y (i)= 0.d0
          Z1(i)= 0.d0
          Z2(i)= 0.d0
          Z3(i)= 0.d0
          Z4(i)= 0.d0
          Z5(i)= 0.d0
        enddo
!$omp end parallel do
      endif

      ALPHA= 1.d0
!C===
```

問題サイズ，オプション指定

nopt＝0　First Touchなし
nopt≠0　First Touchあり

test.f(1/3):初期化

```
      use omp_lib
      implicit REAL*8 (A-H,O-Z)
      real(kind=8), dimension(:), allocatable :: X, Y
      real(kind=8), dimension(:), allocatable :: Z1, Z2
      real(kind=8), dimension(:), allocatable :: Z3, Z4, Z5
      integer, dimension(0:2) :: INDEX
!C
!C +------+
!C | INIT |
!C +------+
!C===
      write (*,*) 'N, nopt ?'
      read  (*,*)  N, nopt

      allocate (X(N), Y(N), Z1(N), Z2(N), Z3(N), Z4(N), Z5(N))
      if (nopt.eq.0) then
        X = 1.d0
        Y = 1.d0
        Z1= 0.d0
        Z2= 0.d0
        Z3= 0.d0
        Z4= 0.d0
        Z5= 0.d0
       else
!$omp parallel do private (i)
        do i= 1, N
          X (i)= 0.d0
          Y (i)= 0.d0
          Z1(i)= 0.d0
          Z2(i)= 0.d0
          Z3(i)= 0.d0
          Z4(i)= 0.d0
          Z5(i)= 0.d0
        enddo
!$omp end parallel do
      endif

      ALPHA= 1.d0
!C===
```

nopt＝0 First Touchなし
並列化せずに初期化

test.f(1/3):初期化

```
      use omp_lib
      implicit REAL*8 (A-H,O-Z)
      real(kind=8), dimension(:), allocatable :: X, Y
      real(kind=8), dimension(:), allocatable :: Z1, Z2
      real(kind=8), dimension(:), allocatable :: Z3, Z4, Z5
      integer, dimension(0:2) :: INDEX
!C
!C +------+
!C | INIT |
!C +------+
!C===
      write (*,*) 'N, nopt ?'
      read  (*,*)  N, nopt

      allocate (X(N), Y(N), Z1(N), Z2(N), Z3(N), Z4(N), Z5(N))
      if (nopt.eq.0) then
        X = 1.d0
        Y = 1.d0
        Z1= 0.d0
        Z2= 0.d0
        Z3= 0.d0
        Z4= 0.d0
        Z5= 0.d0
       else
!$omp parallel do private (i)
        do i= 1, N
          X (i)= 0.d0
          Y (i)= 0.d0
          Z1(i)= 0.d0
          Z2(i)= 0.d0
          Z3(i)= 0.d0
          Z4(i)= 0.d0
          Z5(i)= 0.d0
        enddo
!$omp end parallel do
      endif

      ALPHA= 1.d0
!C===
```

nopt≠0　First Touchあり

計算をするときと同じように
OpenMPを使って並列化

これで計算をするコアのローカル
メモリにデータが保存される

test.f(2/3):DAXPY

```
!C
!C +-------+
!C | DAXPY |
!C +-------+
!C===
      S2time= omp_get_wtime()
!$omp parallel do private (i)
        do i= 1, N
          Z1(i)= ALPHA*X(i) + Y(i)
          Z2(i)= ALPHA*X(i) + Y(i)
          Z3(i)= ALPHA*X(i) + Y(i)
          Z4(i)= ALPHA*X(i) + Y(i)
          Z5(i)= ALPHA*X(i) + Y(i)
        enddo
!$omp end parallel do
      E2time= omp_get_wtime()

      write (*,'(/a)')           '# DAXPY'
      write (*,'( a, 1pe16.6)') '  omp-1 ', E2time - S2time
!C===
```

test.f(3/3):内積

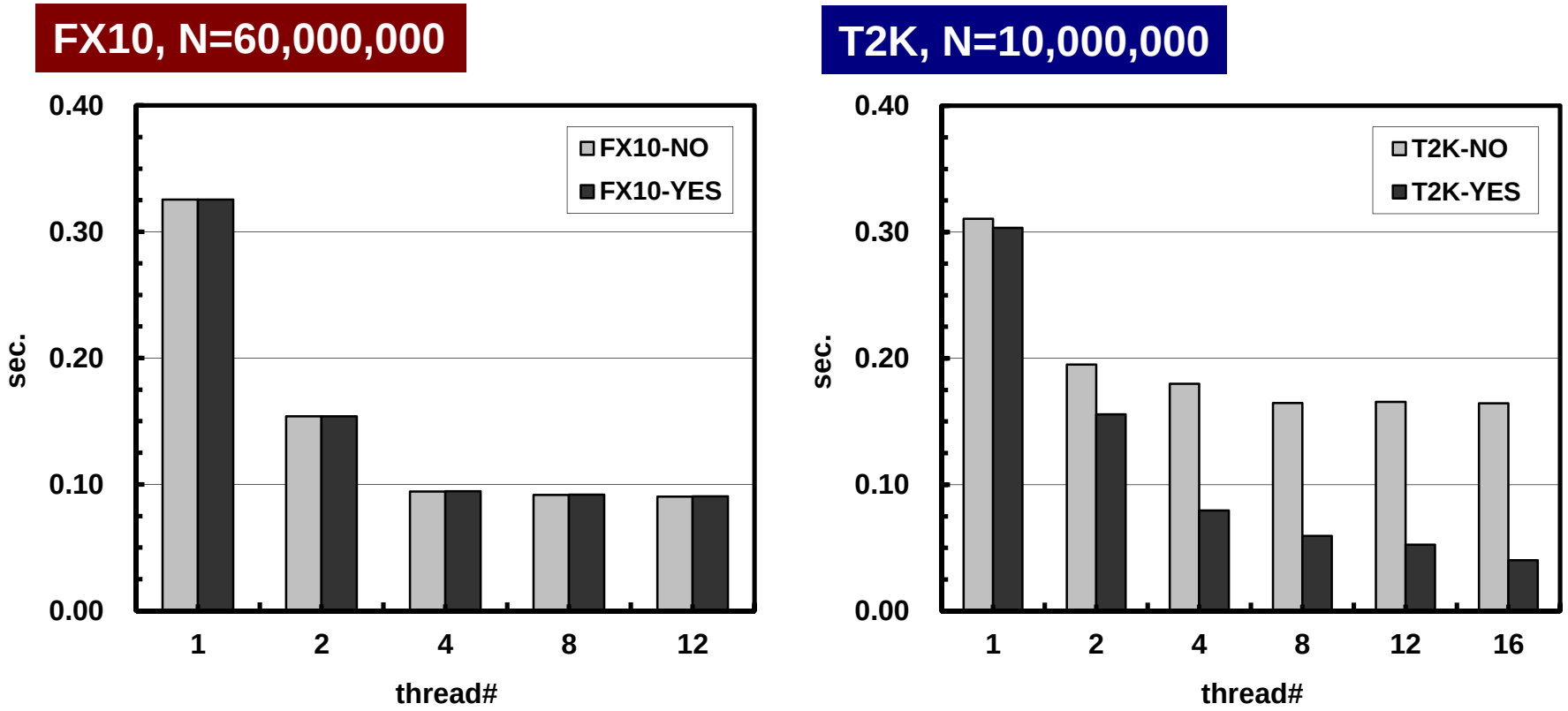
```
!C
!C +-----+
!C | DOT |
!C +-----+
!C===
      V1= 0.d0
      V2= 0.d0
      V3= 0.d0
      V4= 0.d0
      V5= 0.d0
      S2time= omp_get_wtime()
!$omp parallel do private(i) reduction (+:V1,V2,V3,V4,V5)
        do i= 1, N
          V1= V1 + X(i)*(Y(i)+1.d0)
          V2= V2 + X(i)*(Y(i)+2.d0)
          V3= V3 + X(i)*(Y(i)+3.d0)
          V4= V4 + X(i)*(Y(i)+4.d0)
          V5= V5 + X(i)*(Y(i)+5.d0)
        enddo
!$omp end parallel do
      E2time= omp_get_wtime()

      write (*,'(/a)')           '# DOT'
      write (*,'( a, 1pe16.6)') '  omp-1 ', E2time - S2time
!C===
      stop
      end
```

test.c：ダミーの#pragmaが必要

```
#pragma omp parallel
  {
}

        if (nopt==0) {for(i=0; i<N; i++) {
                X[i]  = 1.0;
                Y[i]  = 1.0;
                Z1[i] = 0.0;
                Z2[i] = 0.0;
                Z3[i] = 0.0;
                Z4[i] = 0.0;
                Z5[i] = 0.0;}
        }else{
#pragma omp parallel for private(i)
                for(i=0; i<N; i++) {
                        X[i]  = 1.0;
                        Y[i]  = 1.0;
                        Z1[i] = 0.0;
                        Z2[i] = 0.0;
                        Z3[i] = 0.0;
                        Z4[i] = 0.0;
                        Z5[i] = 0.0;
                }
        }
```

OpenMP のスレッド（マスタスレッド以外のスレッド）は、プログラム中で最初に現れた parallel 構文の実行時に生成される。このため、最初に現れた parallel 構文のみスレッド生成のための大きなコストがかかる。Fortranでは自動挿入，Cでは手動挿入要する。京，FX10に特有の現象。

DAXPY：First Touchの効果



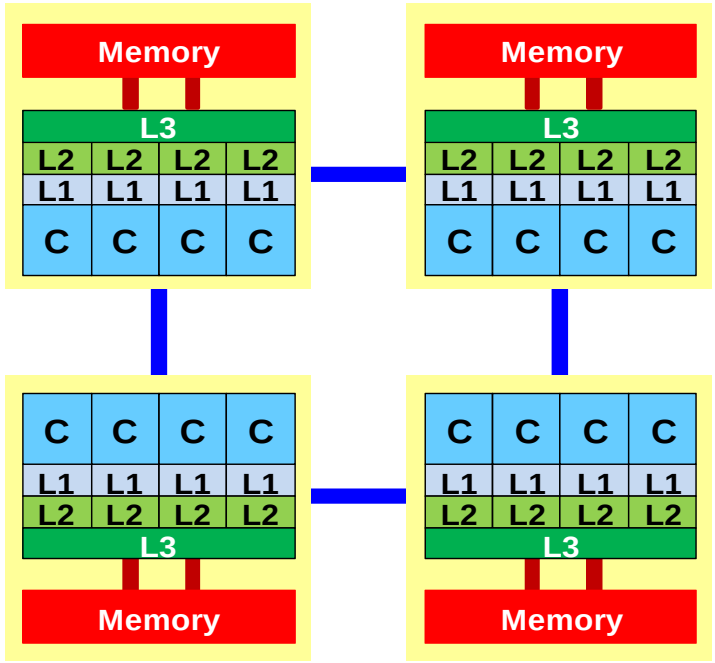
- T2K：First Touchの有無の効果が大
- コア数を増やしても性能が上がらない
  - オーバーヘッド，メモリ競合

First Touch Rule

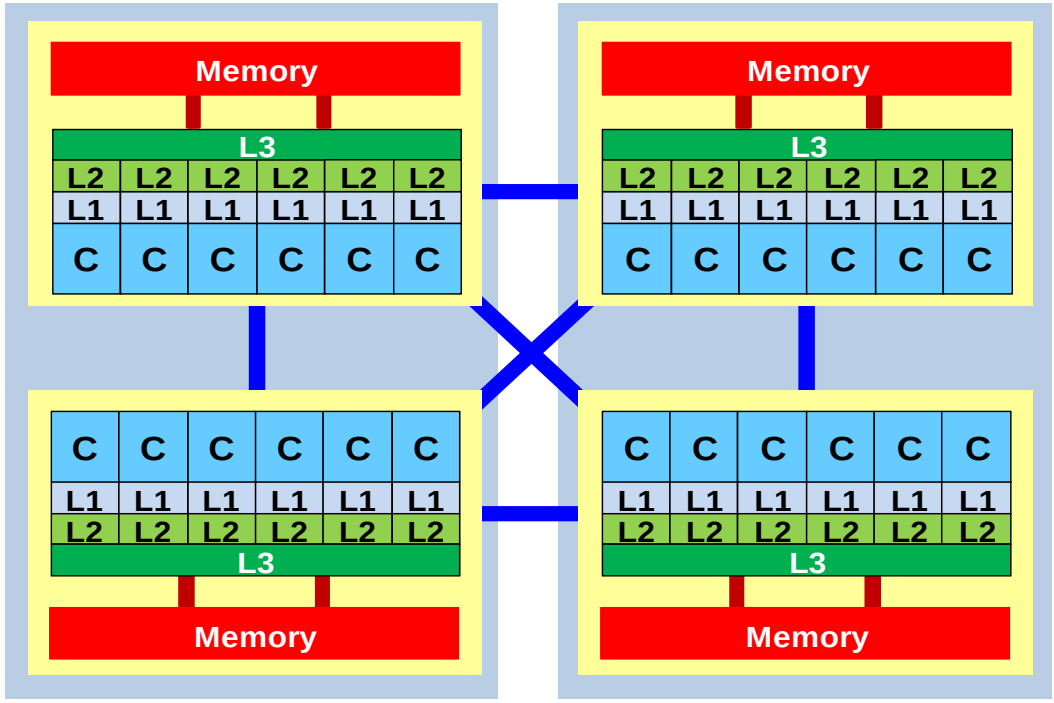
- NUMA（Non-Uniform Access）アーキテクチュアでは，「最初にそのバッファにアクセスしたプロセッサ」のメモリ上にバッファの内容がアサインされる。

- 初期化等の工夫が必要
  - Hitachi SR シリーズ，IBM SP，地球シミュレータ等では問題にはならない
  - T2K東大等では結構効く（効いていた）
    - 京，FX10では効かない
    - ローカルなメモリ上のデータをアクセスするような工夫が必要

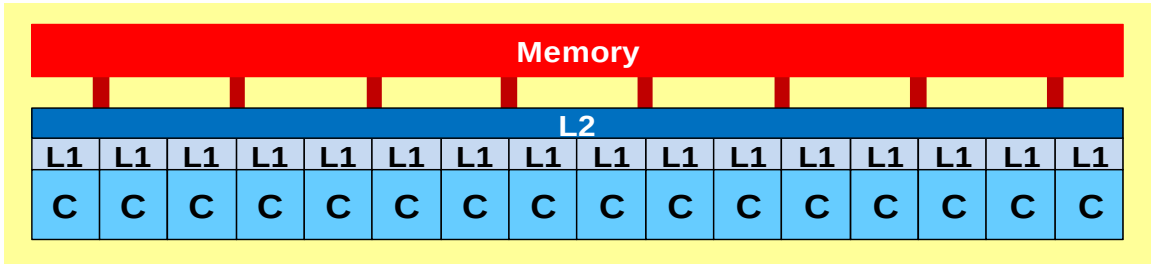
T2K/Tokyo



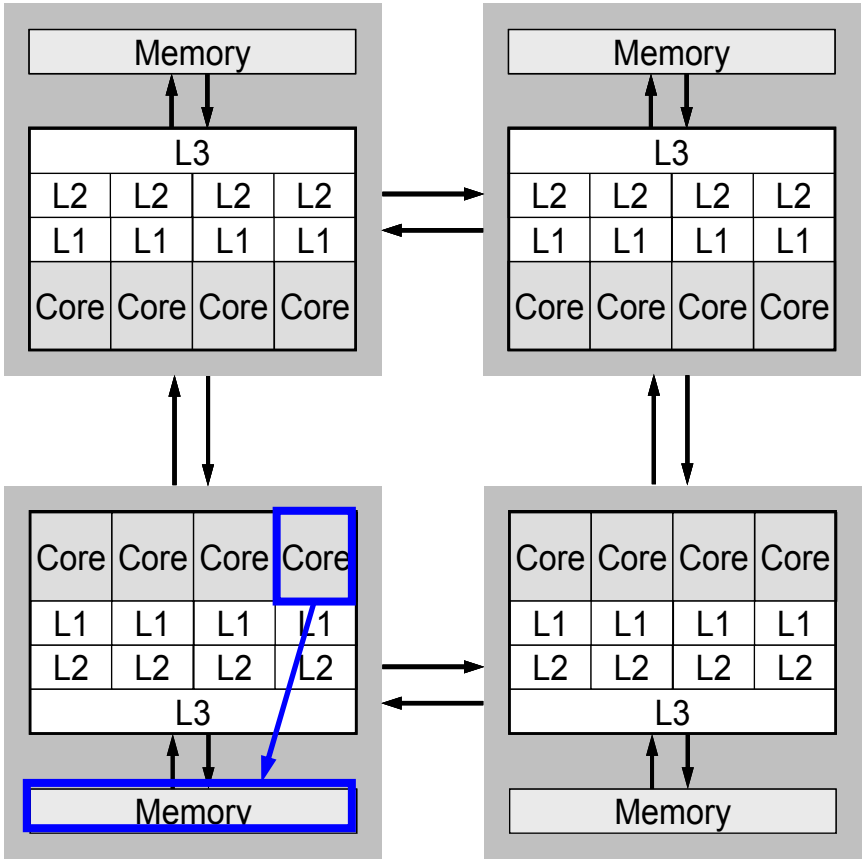
Cray XE6 (Hopper)



Fujitsu FX10 (Oakleaf-FX)

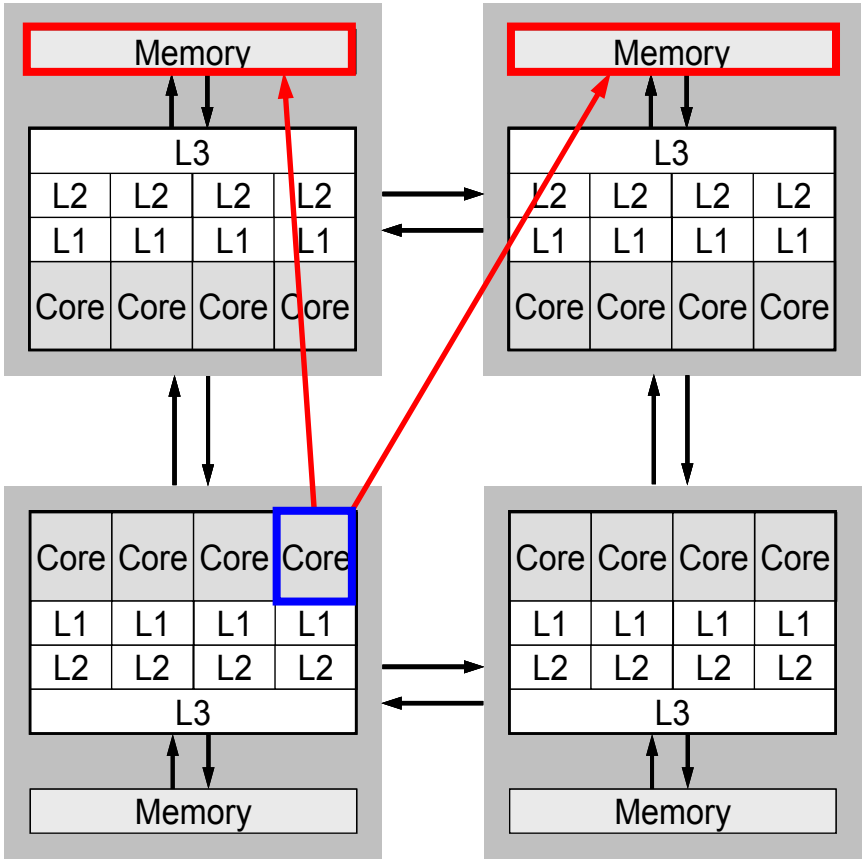


NUMAアーキテクチャ



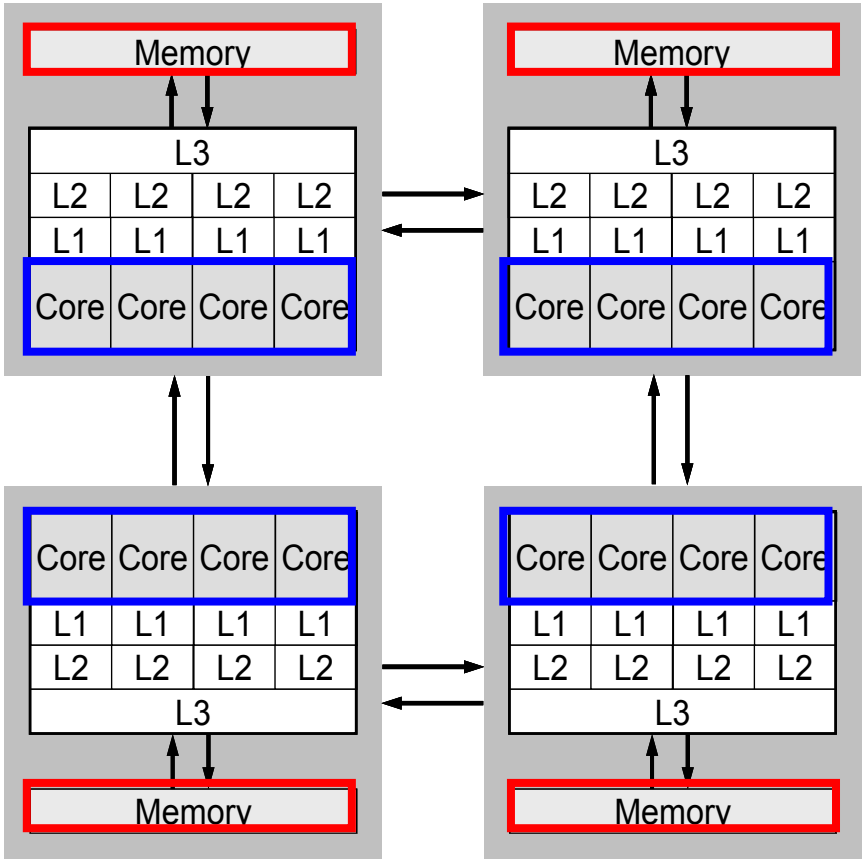
- コアで扱うデータはなるべくローカルなメモリ(コアの属するソケットのメモリ)上にあると効率が良い。

NUMAアーキテクチャ



- 異なるソケットにある場合はアクセスに時間がかかる。

NUMAアーキテクチャ



- First-touchによって，できるだけローカルなメモリ上にデータを持ってくる。
- NUMAアーキテクチャでは，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（ページファイル）が確保される。
  - 配列の初期化手順によって大幅な性能向上が期待できる。

First Touch Data Placement

"Patterns for Parallel Programming" Mattson, T.G. et al.

To reduce memory traffic in the system, it is important to keep the data close to the PEs that will work with the data (e.g. NUMA control).

On NUMA computers, this corresponds to making sure the pages of memory are allocated and "owned" by the PEs that will be working with the data contained in the page.

The most common NUMA page-placement algorithm is the "first touch" algorithm, in which the PE first referencing a region of memory will have the page holding that memory assigned to it.

A very common technique in OpenMP program is to initialize data in parallel using the same loop schedule as will be used later in the computations.

First Touch Data Placement

配列のメモリ・ページ:
最初にtouchしたコアのローカルメモリ上に確保

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

First Touch Data Placement

配列のメモリ・ページ:
最初にtouchしたコアのローカルメモリ上に確保
計算と同じ順番で初期化

```
!$omp parallel do private(ip,i,VAL,k)
      do i= 1, N
        D(i)  = 0.d0
        W(i,P)= 0.d0
        do k= index(i-1)+1, index(i)
          AMAT(k)= 0.d0
          item(k)= 0.d0
        enddo
      enddo
```

First Touch Data Placement

```
!$omp parallel do private(ip,i,VAL,k)
      do i= 1, N
        VAL= D(i)*W(i,P)
        do k= index(i-1)+1, index(i)
          VAL= VAL + AMAT(k)*W(item(k),P)
        enddo
      enddo
```

- 青字：ローカルメモリに載ることが保証される変数
- 赤字：ローカルメモリに載ることが保証されない変数（右辺のp）

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
```

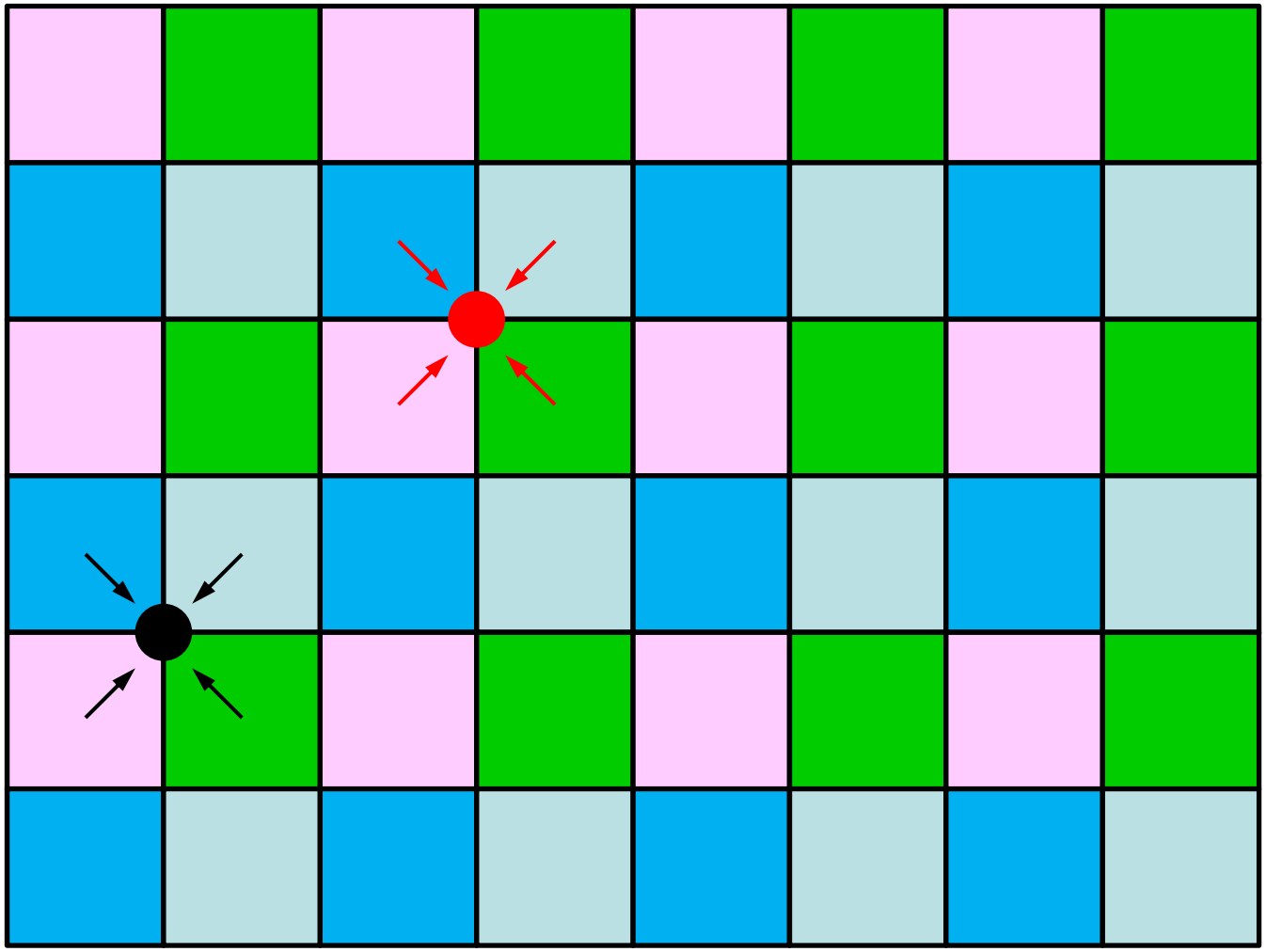
```
 for( neib=1;neib<=NEIBPETOT;neib++){
    istart=EXPORT_INDEX[neib-1];
    inum  =EXPORT_INDEX[neib]-istart;
#pragma omp parallel for private (k,ii)
    for( k=istart;k<istart+inum;k++){
      ii= EXPORT_ITEM[k];
      WS[k]= X[ii-1];
    }
    MPI_Isend(&WS[istart],inum,MPI_DOUBLE,
              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（Spring Schoolでやります）
- 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

      write (*,'(a,2i8)') '### Number of Element Colors',
     &                    my_rank, ELMCOLORtot

      deallocate (IWKX)
```

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)
...
```

計算例(1/2)@東大

512×382×256= 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

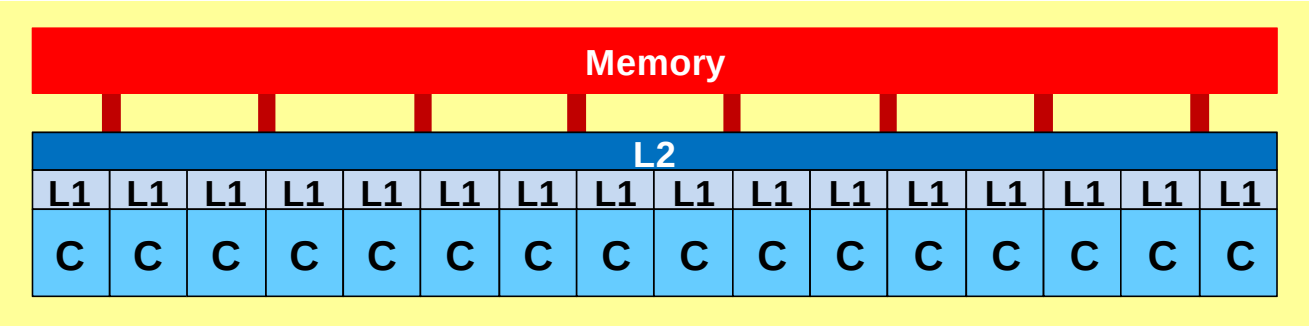
50,331,648 節点，12ノード，192コア
12 MPIプロセス，スレッド数変化
64^3=262,144節点／コア

```
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



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 {
```