OpenMPによるマルチコア・メニィコア並列プログラミング入門 C言語編

第Ⅱ部：OpenMP

中島研吾

東京大学情報基盤センター

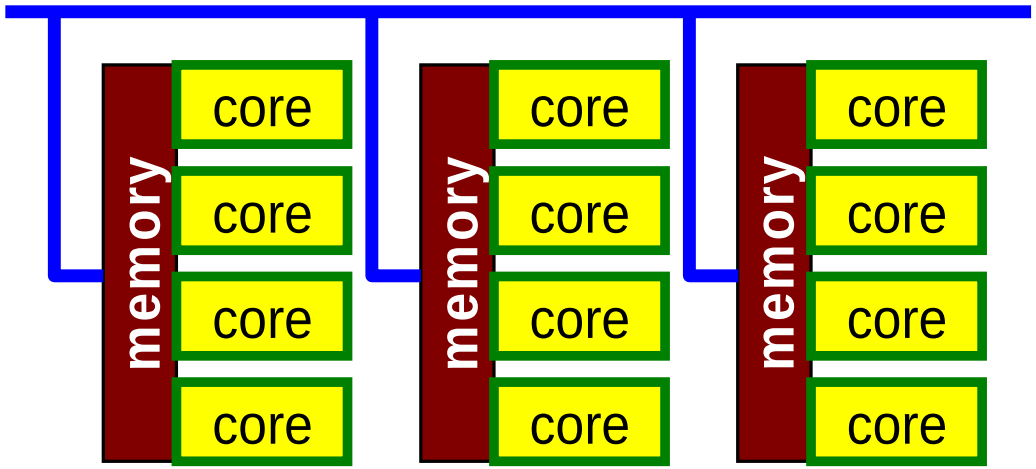
- OpenMP
- Login to OBCX
- Parallel Version of the Code by OpenMP
- STREAM
- Data Dependency

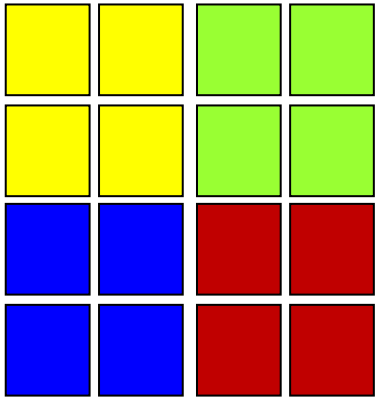
Hybrid並列プログラミング

- スレッド並列＋メッセージパッシング
  - OpenMP+ MPI
  - CUDA + MPI, OpenACC + MPI
- 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

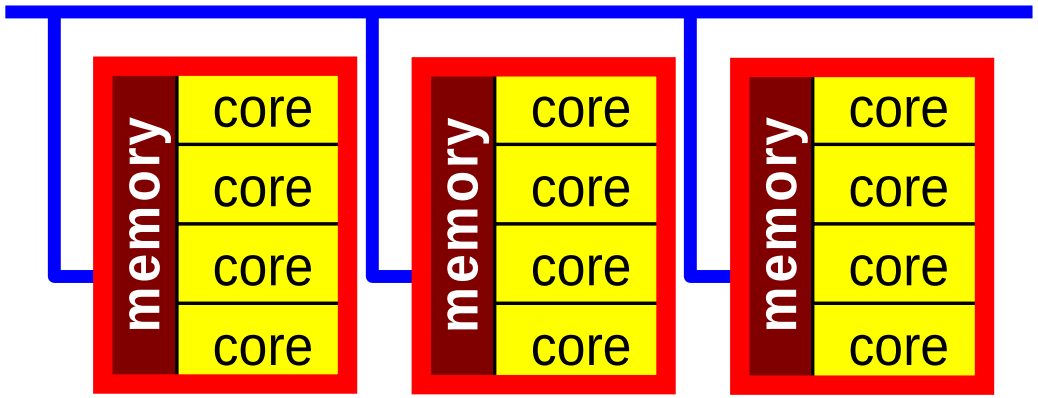
Flat MPI vs. Hybrid

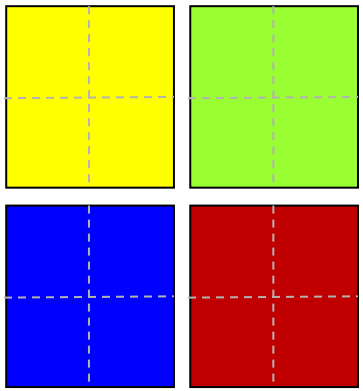
Flat-MPI：Each Core -> Independent



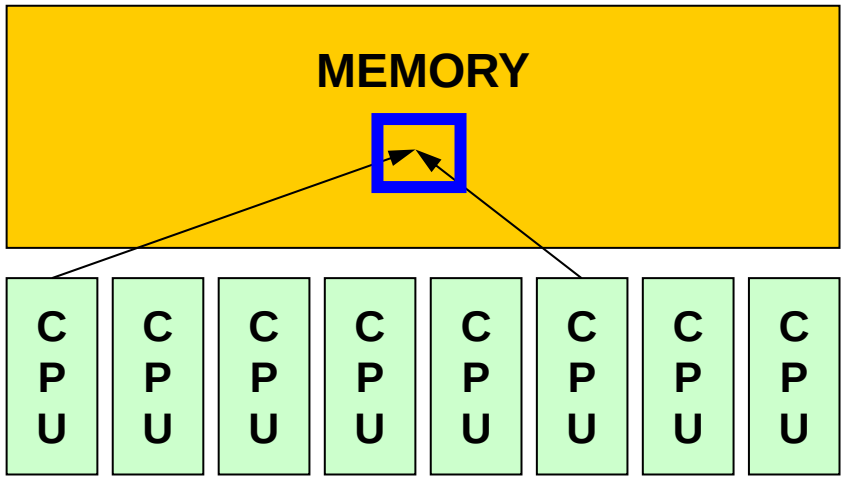


Hybrid：Hierarchal Structure





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（1/2）

http://www.openmp.org

- 共有メモリ型並列計算機用のDirectiveの統一規格
  - MPIやHPFに比べると遅く1996年頃から活動開始
  - 現在 Ver.4.X
- 背景
  - CrayとSGIの合併（1996）
  - ASCI計画の開始（1995）
    - Accelerated Strategic Computing Initiative (ASCI) -> Advanced Simulation and Computing Program (ASC)
    - ASCI：核実験のシミュレーションによる代替
      - 計算機開発，シミュレーションソフトウェア
    - SMPクラスタにフォーカス
      - ASCI Red (Intel)，Blue Pacific/White/Purple/BlueGene (IBM)，Blue Mountain（SGI）
    - SMPクラスタ向けの並列プログラミングの共通API（Application Program Interface）が必要

OpenMPとは（2/2）

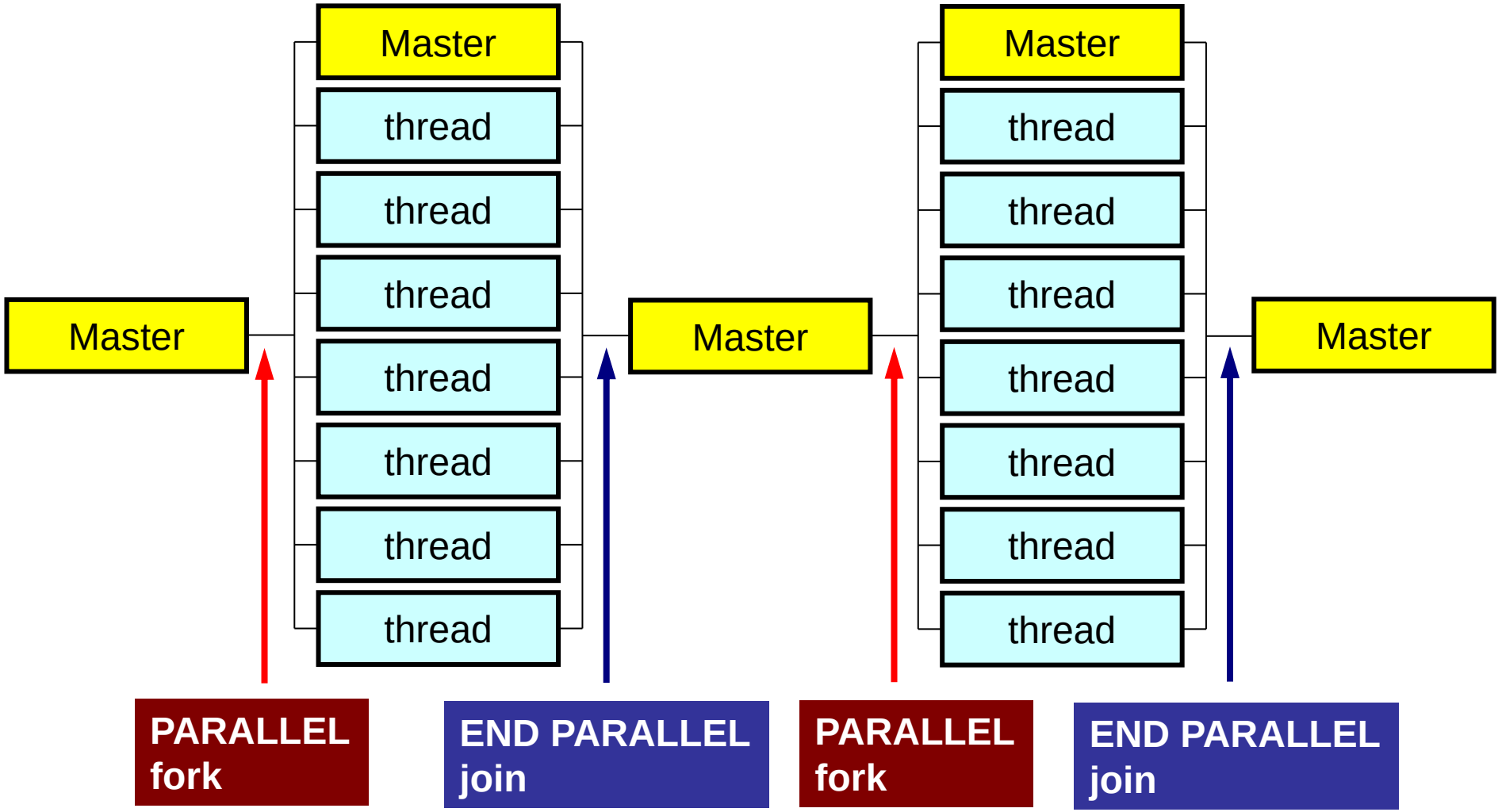
http://www.openmp.org

- 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に近づいている

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

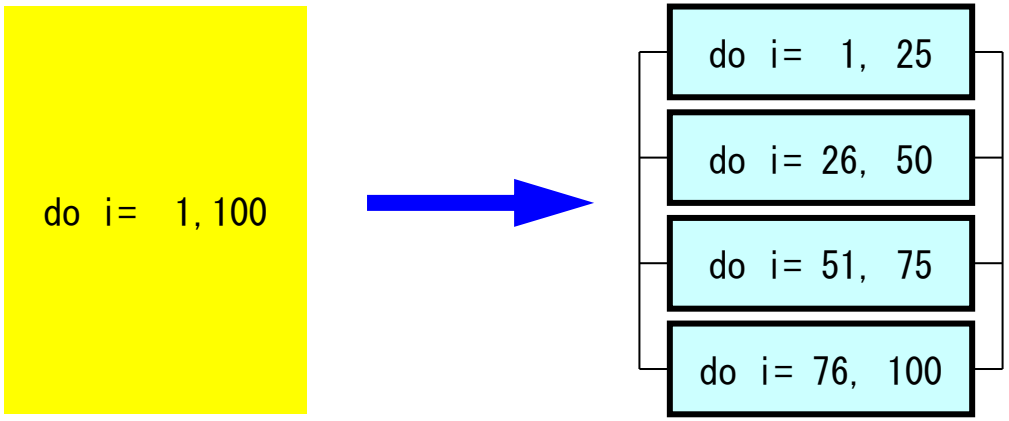
Fork-Join 型並列モデル



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

```
#pragma omp parallel for private (i)
 for (i=0; i<N; i++) {
      X[i] = 0.0;
      W[0][i] = 0.0;
      W[1][i] = 0.0;
      W[2][i] = 0.0;
 }
```

Dot Products

```
RHO = 0.0;
#pragma omp parallel for private (i)
 reduction (+:RHO)
 for(i=0; i<N; i++) {
     RHO += W[R][i] * W[Z][i];
 }
```

DAXPY

```
#pragma omp parallel for private (i)
 for (i=0; i<N; i++) {
    Y[i] = Y[i} + alphat*X[i];
 }
```

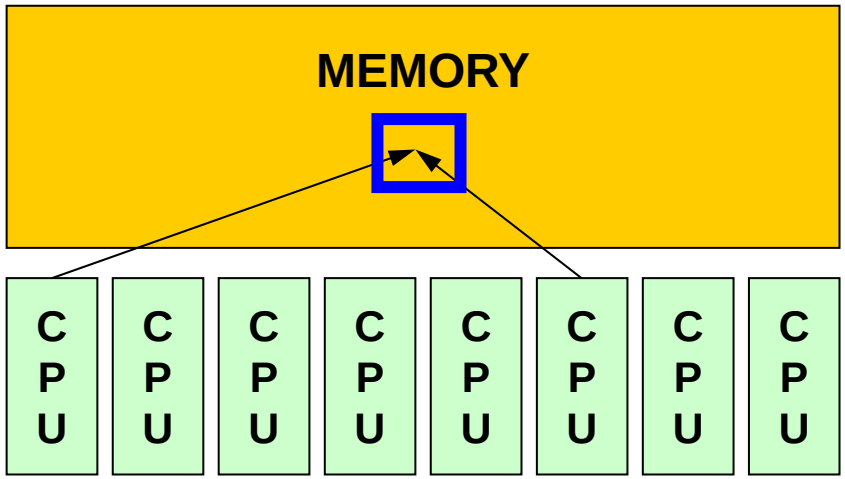
OpenMP/Direceives Matrix/Vector Products

```
#pragma omp parallel for private (i,VAL,j)
 for(i=0; i<N; i++) {
     VAL = D[i] * W[P][i];
     for(j=indexL[i]; j<indexL[i+1]; j++) {
         VAL += AL[j] * W[P][itemL[j]-1];
     }

     for(j=indexU[i]; j<indexU[i+1]; j++) {
         VAL += AU[j] * W[P][itemU[j]-1];
     }
     W[Q][i] = VAL;
 }
```

OpenMPの特徴

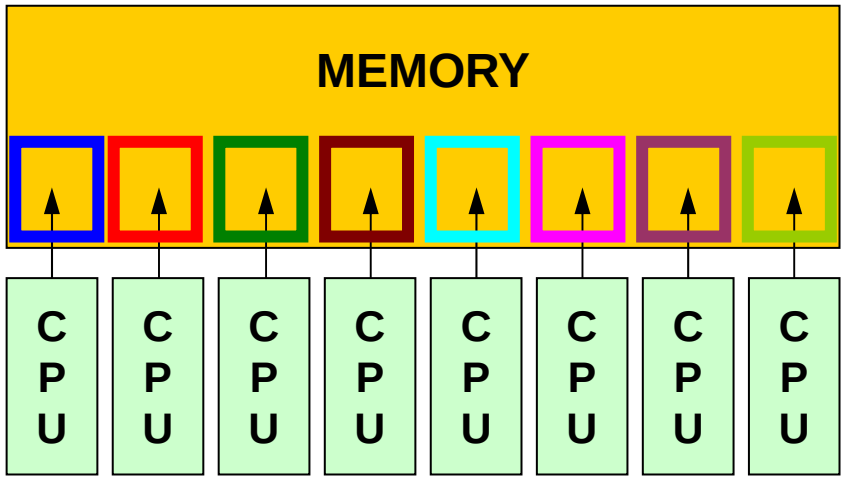
- ディレクティヴ（指示行）の形で利用
  - 挿入直後のループが並列化される
  - コンパイラがサポートしていなければ，コメントとみなされる
- 何も指定しなければ，何もしない
  - 「自動並列化」，「自動ベクトル化」とは異なる。
  - 下手なことをするとおかしな結果になる：ベクトル化と同じ
  - データ分散等（Ordering）は利用者の責任
- 共有メモリユニット内のプロセッサ数に応じて，「Thread」が立ち上がる
  - 「Thread」：MPIでいう「プロセス」に相当する。
  - 普通は「Thread数＝共有メモリユニット内プロセッサ数，コア数」であるが最近のアーキテクチ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



- 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（Memory Contention）（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。
- データ依存性（Data Dependency）
- 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（メモリ飽和）。

OpenMPの特徴（続き）

- 基本は「#pragma omp parallel for」
- 変数について，グローバル/sharedな変数と，Thread内でローカルな「private」な変数に分けられる。
  - デフォルトは「global/shared」
  - 内積を求める場合は「reduction」を使う

```
      VAL= 0.0;
#pragma omp parallel for private (i,ip)
 reduction(+:VAL)
    for(ip=0; ip<PEsmpTOT; ip++){
        for (i=INDEX[ip]; i<INDEX[ip+1]; i++){
           VAL= VAL + W[R][i] * W[Z][i];
        }
    }
```

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)。

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/for

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

```
#pragma omp 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

```
#pragma omp parllel for
 for(i=0; i<N; i++){
     B[i]= (A[i] + B[i]) * 0.50;
 }
```

- 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

```
#pragma omp parallel default(private) reduction(+:A,B)
 for(i=0; i<N; i++){
     err= work(Alocal, Blocal);
     A= A + Alocal;
     B= B + Blocal;
 }
```

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 for Dot Products

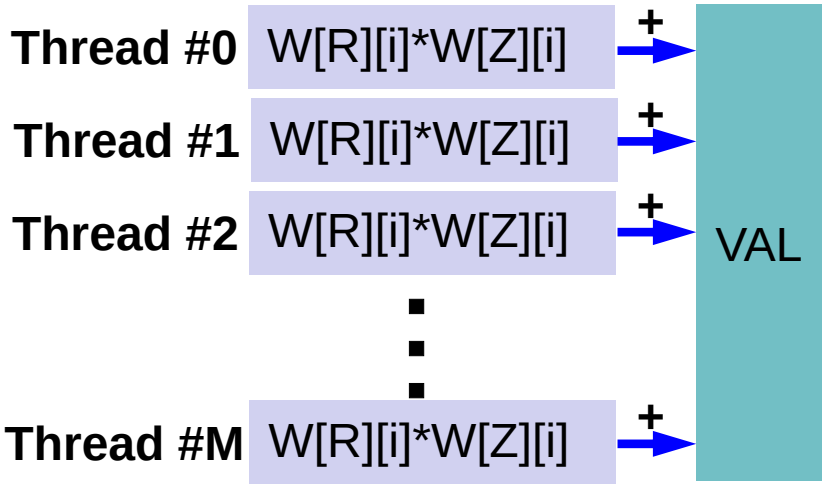
```
VAL= 0.0;
for(i=0; i<N; i++){
  VAL= VAL + W[R][i] * W[Z][i];
}
```

OpenMP for Dot Products

```
    VAL= 0.0;
    for(i=0; i<N; i++){
      VAL= VAL + W[R][i] * W[Z][i];
    }
```

```
    VAL= 0.0;
#pragma omp parallel for private (i) reduction(+:VAL)
    for(i=0; i<N; i++){
      VAL= VAL + W[R][i] * W[Z][i];
    }
```

Directives are just inserted.



OpenMP for Dot Products

```
      VAL= 0.0;
      for(i=0; i<N; i++){
        VAL= VAL + W[R][i] * W[Z][i];
      }
```

```
      VAL= 0.0;
#pragma omp parallel for private (i) reduction(+:VAL)
      for(i=0; i<N; i++){
        VAL= VAL + W[R][i] * W[Z][i];
      }
```

OpenMPディレクティヴの挿入
これでも並列計算は可能

```
       VAL= 0.0;
#pragma omp parallel for private (i,ip)
 reduction(+:VAL)
      for(ip=0; ip<PEsmpTOT; ip++){
          for (i=INDEX[ip]; i<INDEX[ip+1]; i++){
             VAL= VAL + W[R][i] * W[Z][i];
          }
      }
```

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 for Dot Products

```
      VAL= 0.0;
      for(i=0; i<N; i++){
        VAL= VAL + W[R][i] * W[Z][i];
      }
```

```
      VAL= 0.0;
#pragma omp parallel for private (i) reduction(+:VAL)
      for(i=0; i<N; i++){
        VAL= VAL + W[R][i] * W[Z][i];
      }
```

OpenMPディレクティヴの挿入
これでも並列計算は可能

```
       VAL= 0.0;
#pragma omp parallel for private (i,ip)
 reduction(+:VAL)
      for(ip=0; ip<PEsmpTOT; ip++){
          for (i=INDEX[ip]; i<INDEX[ip+1]; i++){
             VAL= VAL + W[R][i] * W[Z][i];
          }
      }
```

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 for Dot Products

```
 VAL= 0.0;
#pragma omp parallel for private (i,ip)
 reduction(+:VAL)
    for(ip=0; ip<PEsmpTOT; ip++){
        for (i=INDEX[ip]; i<INDEX[ip+1]; i++){
           VAL= VAL + W[R][i] * W[Z][i];
        }
    }
```

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

e.g.: N=100, PEsmpTOT=4

INDEX[0]= 0
INDEX[1]= 25
INDEX[2]= 50
INDEX[3]= 75
INDEX[4]= 100

Matrix-Vector Multiply

```
for(i=0; i<N; i++) {
    VAL = D[i] * W[P][i];
    for(j=indexL[i]; j<indexL[i+1]; j++) {
        VAL += AL[j] * W[P][itemL[j]-1];
    }

    for(j=indexU[i]; j<indexU[i+1]; j++) {
        VAL += AU[j] * W[P][itemU[j]-1];
    }
    W[Q][i] = VAL;
}
```

Matrix-Vector Multiply

```
#pragma omp parallel for private(ip,i,VAL,j)
 for (ip=0; ip<PEsmpTOT; ip++){
      for(i=SMPindexG[ip]; i<SMPindexG[ip+1]; i++){
          VAL = D[i] * W[P][i];
          for(j=indexL[i]; j<indexL[i+1]; j++) {
              VAL += AL[j] * W[P][itemL[j]-1];
          }

          for(j=indexU[i]; j<indexU[i+1]; j++) {
              VAL += AU[j] * W[P][itemU[j]-1];
          }
          W[Q][i] = VAL;
      }
 }
```

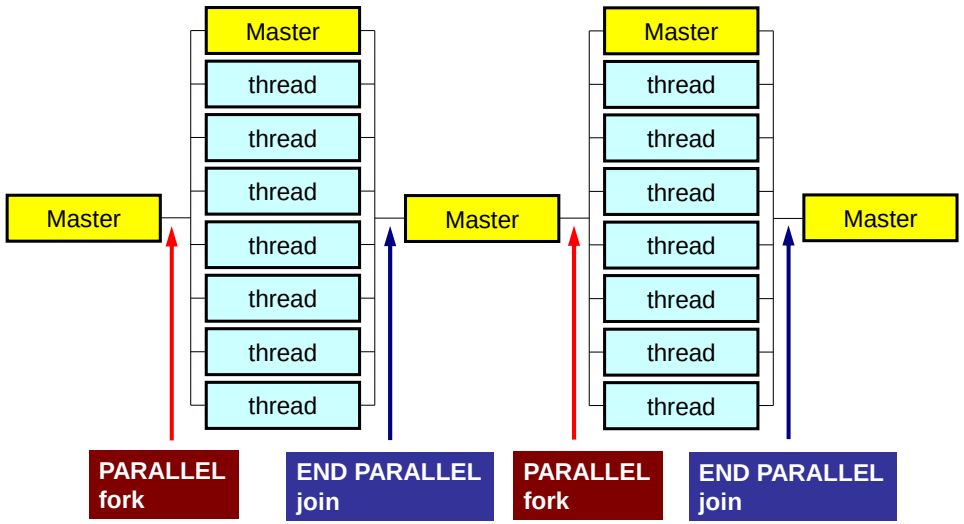
Matrix-Vector Multiply: Other Approach

This is rather better for GPU and (very) many-core architectures: simpler structure of loops

```
#pragma omp parallel for private(i,VAL,j)
 for(i=0; i<N; i++){
          VAL = D[i] * W[P][i];
          for(j=indexL[i]; j<indexL[i+1]; j++) {
              VAL += AL[j] * W[P][itemL[j]-1];
          }

          for(j=indexU[i]; j<indexU[i+1]; j++) {
              VAL += AU[j] * W[P][itemU[j]-1];
          }
          W[Q][i] = VAL;
 }
```

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オーバーヘッドになる。
- omp parallel + omp do/omp for




```
#pragma omp parallel ...

#pragma omp for {


...
#pragma omp for {
```

```
!$omp parallel ...

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

- OpenMP
- **Login to OBCX**
- Parallel Version of the Code by OpenMP
- STREAM
- Data Dependency

- OpenMP
- Login to OBCX
- **Parallel Version of the Code by OpenMP**
- STREAM
- Data Dependency

ここでの目標

- solve_PCG（対角スケーリング，点ヤコビ前処理付きCG法）の並列化
- “solver_PCG.f/c” (solve_PCG)
- OpenMP指示行を入れるだけ（Index使わない）でやってみる

前処理付共役勾配法

Preconditioned Conjugate Gradient Method（PCG）

```
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 p(i-1)
    endif
    q(i)= [A]p(i)
    αi  = ρi-1/p(i)q(i)
    x(i)= x(i-1) + αip(i)
    r(i)= r(i-1) – αiq(i)
    check convergence |r|
end
```

実際にやるべき計算は：

$$\{z\}=[M]^{-1}\{r\}$$

「近似逆行列」の計算が必要：

$$[M]^{-1}\approx[A]^{-1},\quad [M]\approx[A]$$

究極の前処理：本当の逆行列

$$[M]^{-1}=[A]^{-1},\quad [M]=[A]$$

対角スケーリング：簡単＝弱い

$$[M]^{-1}=[D]^{-1},\quad [M]=[D]$$

対角スケーリング，点ヤコビ前処理

- 前処理行列として，もとの行列の対角成分のみを取り出した行列を前処理行列 [M] とする。
  - 対角スケーリング，点ヤコビ（point-Jacobi）前処理

$$[M]=\begin{bmatrix} D_1 & 0 & \cdots & 0 & 0 \\ 0 & D_2 & & 0 & 0 \\ \cdots & & \cdots & & \cdots \\ 0 & 0 & & D_{N-1} & 0 \\ 0 & 0 & \cdots & 0 & D_N \end{bmatrix}$$

- **solve [M]z$^{(i-1)}$= r$^{(i-1)}$**という場合に逆行列を簡単に求めることができる。
- 簡単な問題では収束する。

Original: solve_PCG (1/3)

```
for(i=0; i<N; i++) {
    W[DD][i]= 1.e0/D[i];
}

...


for(L=0; L<(*ITR); L++) {

/*******************
 * {z} = [Minv]{r} *
 *******************/
    for(i=0; i<N; i++) {
        W[Z][i] = W[R][i]*W[DD][i];
    }


/*****************
 * RHO = {r}{z} *
 *****************/
    RHO = 0.0;
    for(i=0; i<N; i++) {
        RHO += W[R][i] * W[Z][i];
    }
```

対角成分の逆数(前処理用)その都度，除算をすると効率が悪いため，予め配列に格納。嘗ては除算と加減乗算は10:1と言われていたが最近はそれほどでもない。

Compute $r^{(0)}= b-[A]x^{(0)}$

for i= 1, 2, …

  solve $[M]z^{(i-1)}= r^{(i-1)}$

  $\rho_{i-1}= r^{(i-1)}\ z^{(i-1)}$

  if i=1

    $p^{(1)}= z^{(0)}$

  else

    $\beta_{i-1}= \rho_{i-1}/\rho_{i-2}$

    $p^{(i)}= z^{(i-1)} + \beta_{i-1}\ p^{(i-1)}$

  endif

  $q^{(i)}= [A]p^{(i)}$

  $\alpha_i = \rho_{i-1}/p^{(i)}q^{(i)}$

  $x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$

  $r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$

  check convergence $|r|$

end

Original: solve_PCG (2/3)

```
/*********************************
 * {p}  = {z} if      ITER=0    *
 * BETA = RHO / RHO1  otherwise *
 *********************************/

if(L == 0) {
   for(i=0; i<N; i++) {
       W[P][i] = W[Z][i];
   }
   } else {
   BETA = RHO / RHO1;
   for(i=0; i<N; i++) {
       W[P][i] = W[Z][i] + BETA * W[P][i];
   }
}

/*****************
 * {q} = [A]{p} *
 *****************/

for(i=0; i<N; i++) {
    VAL = D[i] * W[P][i];
    for(j=indexL[i]; j<indexL[i+1]; j++) {
        VAL += AL[j] * W[P][itemL[j]-1];
    }
    for(j=indexU[i]; j<indexU[i+1]; j++) {
        VAL += AU[j] * W[P][itemU[j]-1];
    }
    W[Q][i] = VAL;
}
```

Compute $r^{(0)}= b-[A]x^{(0)}$
for i= 1, 2, …
  solve $[M]z^{(i-1)}= r^{(i-1)}$
  $\rho_{i-1}= r^{(i-1)}\ z^{(i-1)}$
  if i=1
    $p^{(1)}= z^{(0)}$
  else
    $\beta_{i-1}= \rho_{i-1}/\rho_{i-2}$
    $p^{(i)}= z^{(i-1)} + \beta_{i-1}\ p^{(i-1)}$
  endif
  $q^{(i)}= [A]p^{(i)}$
  $\alpha_i = \rho_{i-1}/p^{(i)}q^{(i)}$
  $x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$
  $r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$
  check convergence $|r|$
end

Original: solve_PCG (3/3)

```
/*************************
 * ALPHA = RHO / {p}{q} *
 *************************/
C1 = 0.0;
for(i=0; i<N; i++) {
    C1 += W[P][i] * W[Q][i];
}
ALPHA = RHO / C1;

/*****************************
 * {x} = {x} + ALPHA * {p} *
 * {r} = {r} - ALPHA * {q} *
 *****************************/
for(i=0; i<N; i++) {
    X[i]    += ALPHA * W[P][i];
    W[R][i] -= ALPHA * W[Q][i];
}

DNRM2 = 0.0;
for(i=0; i<N; i++) {
    DNRM2 += W[R][i]*W[R][i];
}

ERR = sqrt(DNRM2/BNRM2);
if((L+1)%100 ==1) {
   fprintf(stderr, "%5d%16.6e¥n", L+1, ERR);
}
if(ERR < EPS) {
   *IER = 0; goto N900;
  } else {
   RHO1 = RHO;
  }
}
*IER = 1;
```

r= b-[A]x
DNRM2=|r|²
BNRM2=|b|²

ERR= |r|/|b|

Compute $r^{(0)}= b-[A]x^{(0)}$
for i= 1, 2, …
  solve $[M]z^{(i-1)}= r^{(i-1)}$
  $\rho_{i-1}= r^{(i-1)}\ z^{(i-1)}$
  if i=1
    $p^{(1)}= z^{(0)}$
  else
    $\beta_{i-1}= \rho_{i-1}/\rho_{i-2}$
    $p^{(i)}= z^{(i-1)} + \beta_{i-1}\ p^{(i-1)}$
  endif
  $q^{(i)}= [A]p^{(i)}$
  $\alpha_i = \rho_{i-1}/p^{(i)}q^{(i)}$
  $x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$
  $r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$
  check convergence |r|
end

solve_PCG (1/5)

parallel computing by OpenMP

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <math.h>
#include <omp.h>

#include "solver_PCG.h"

extern int
solve_PCG (int N, int NL, int NU, int *indexL, int *itemL, int *indexU, int *itemU,
                double *D, double *B, double *X, double *AL, double *AU,
                double EPS, int *ITR, int *IER, int *N2)
{
        double **W;
        double VAL, BNRM2, WVAL, SW, RHO, BETA, RHO1, C1, DNRM2, ALPHA, ERR;
        double Stime, Etime;
        int i, j, ic, ip, L, ip1, N3;
        int R = 0;
        int Z = 1;
        int Q = 1;
        int P = 2;
        int DD = 3;
```

solve_PCG (2/5)

```
#pragma omp parallel for private (i)
        for(i=0; i<N; i++) {
                X[i] = 0.0;
                W[1][i] = 0.0;
                W[2][i] = 0.0;
                W[3][i] = 0.0;
        }

#pragma omp parallel for private (i,VAL,j)
        for(i=0; i<N; i++) {
                VAL = D[i] * X[i];
                for(j=indexL[i]; j<indexL[i+1]; j++) {
                        VAL += AL[j] * X[itemL[j]-1];
                }
                for(j=indexU[i]; j<indexU[i+1]; j++) {
                        VAL += AU[j] * X[itemU[j]-1];
                }
                W[R][i] = B[i] - VAL;
        }

BNRM2 = 0.0;
#pragma omp parallel for private (i) reduction (+:BNRM2)
        for(i=0; i<N; i++) {
          BNRM2 += B[i]*B[i];
        }

#pragma omp parallel for private (i)
        for(i=0; i<N; i++) {
          W[DD][i]= 1.e0/D[i];
        }
```

Compute $r^{(0)}= b-[A]x^{(0)}$

for i= 1, 2, …
- solve $[M]z^{(i-1)}= r^{(i-1)}$
- $\rho_{i-1}= r^{(i-1)}\ z^{(i-1)}$
- if i=1
  - $p^{(1)}= z^{(0)}$
- else
  - $\beta_{i-1}= \rho_{i-1}/\rho_{i-2}$
  - $p^{(i)}= z^{(i-1)} + \beta_{i-1}\ p^{(i-1)}$
- endif
- $q^{(i)}= [A]p^{(i)}$
- $\alpha_i = \rho_{i-1}/p^{(i)}q^{(i)}$
- $x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$
- $r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$
- check convergence $|r|$

end

solve_PCG (3/5)

```
*ITR = N;

Stime = omp_get_wtime();

for(L=0; L<(*ITR); L++) {

#pragma omp parallel for private (i)
        for(i=0; i<N; i++) {
            W[Z][i] = W[R][i]*W[DD][i];
        }

RHO = 0.0;
#pragma omp parallel for private (i) reduction(+:RHO)
        for(i=0; i<N; i++) {
            RHO += W[R][i] * W[Z][i];
        }

if(L == 0) {
#pragma omp parallel for private (i)
        for(i=0; i<N; i++) {
            W[P][i] = W[Z][i];
          }
} else {
        BETA = RHO / RHO1;
#pragma omp parallel for private (i)
        for(i=0; i<N; i++) {
            W[P][i] = W[Z][i] + BETA * W[P][i];
        }
}
```

Compute $r^{(0)}= b-[A]x^{(0)}$

for i= 1, 2, …

  solve $[M]z^{(i-1)}= r^{(i-1)}$

  $\rho_{i-1}= r^{(i-1)}\ z^{(i-1)}$

  if i=1

    $p^{(1)}= z^{(0)}$

  else

    $\beta_{i-1}= \rho_{i-1}/\rho_{i-2}$

    $p^{(i)}= z^{(i-1)} + \beta_{i-1}\ p^{(i-1)}$

  endif

  $q^{(i)}= [A]p^{(i)}$

  $\alpha_i = \rho_{i-1}/p^{(i)}q^{(i)}$

  $x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$

  $r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$

  check convergence $|r|$

end

solve_PCG (4/5)

```
#pragma omp parallel for private (i,VAL,j)
        for(i=0; i<N; i++) {
            VAL = D[i] * W[P][i];
            for(j=indexL[i]; j<indexL[i+1]; j++) {
                VAL += AL[j] * W[P][itemL[j]-1];
            }
            for(j=indexU[i]; j<indexU[i+1]; j++) {
                VAL += AU[j] * W[P][itemU[j]-1];
            }
            W[Q][i] = VAL;
        }

C1 = 0.0;
#pragma omp parallel for private (i) reduction(+:C1)
        for(i=0; i<N; i++) {
            C1 += W[P][i] * W[Q][i];
        }
        ALPHA = RHO / C1;

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

DNRM2 = 0.0;
#pragma omp parallel for private (i) reduction(+:DNRM2)
        for(i=0; i<N; i++) {
            DNRM2 += W[R][i]*W[R][i];
        }

        ERR = sqrt(DNRM2/BNRM2);
...
```

Compute $r^{(0)}= b-[A]x^{(0)}$

for i= 1, 2, …

  solve $[M]z^{(i-1)}= r^{(i-1)}$

  $\rho_{i-1}= r^{(i-1)}\ z^{(i-1)}$

  if i=1

    $p^{(1)}= z^{(0)}$

  else

    $\beta_{i-1}= \rho_{i-1}/\rho_{i-2}$

    $p^{(i)}= z^{(i-1)} + \beta_{i-1}\ p^{(i-1)}$

  endif

  $q^{(i)}= [A]p^{(i)}$

  $\alpha_i = \rho_{i-1}/p^{(i)}q^{(i)}$

  $x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$

  $r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$

  check convergence $|r|$

end

solve_PCG (5/5)

```
        Stime = omp_get_wtime();

for(L=0; L<(*ITR); L++) {
...
        if(ERR < EPS) {
           *IER = 0;
           goto N900;
        } else {
           RHO1 = RHO;
        }
}
        *IER = 1;
N900:
        Etime = omp_get_wtime();

        fprintf(stderr, "%5d%16.6e¥n", L+1, ERR);
        fprintf(stderr, "%16.6e sec. (solver)¥n", Etime - Stime);

        *ITR = L;
        free(W);
        return 0;
}
```

Elapsed Time= Etime - Stime

ファイルコピー：OBCX

```
>$ cd /work/gt00/t00XYZ
>$ cp /work/gt00/z30088/omp/omp-c.tar .

>$ tar xvf omp-c.tar

>$ cd multicore
```

以下のディレクトリが出来ていることを確認

```
  omp stream

>$ cd omp/src20
>$ make
>$ cd ../run

<modify “INPUT.DAT”>
<modify “go1.sh”, “go2.sh”>

>$ pjsub go1.sh
>$ pjsub go2.sh
```

<$O-omp>/src20/Makefile

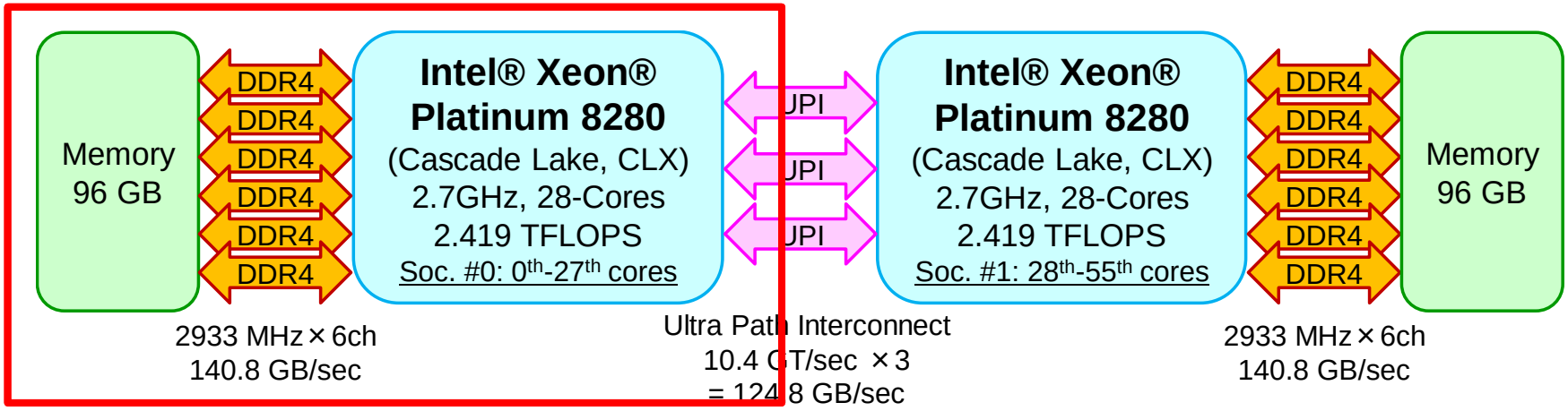
parallel computing by OpenMP

```
F90   = ifort
F90OPTFLAGS= -align array64byte -O3 -axCORE-AVX512 -qopenmp -ipo

F90FLAGS =$(F90OPTFLAGS)

.SUFFIXES:
.SUFFIXES: .o .f .f90 .c
#
.f90.o:; $(F90) -c $(F90FLAGS)  $(F90OPTFLAG) $<
.f.o:; $(F90) -c $(F90FLAGS)  $(F90OPTFLAG) $<
#
OBJS = ¥
solver_PCG.o rcm.o struct.o pcg.o ¥
boundary_cell.o cell_metrics.o ¥
input.o main.o poi_gen.o pointer_init.o outucd.o

TARGET = ../run/sol20

all: $(TARGET)
$(TARGET): $(OBJS)
        $(F90) $(F90FLAGS) -o $(TARGET) ¥
        $(OBJS) ¥
        $(F90FLAGS)
clean:
        rm -f *.o $(TARGET) *.mod *~ PI*
```

ジョブ実行

- 実行方法
  - 基本的にバッチジョブのみ
  - インタラクティヴの実行は「基本的に」できません
- 実行手順
  - ジョブスクリプトを書きます
  - ジョブを投入します
  - ジョブの状態を確認します
  - 結果を確認します
- その他
  - 実行時には1ノード（28x2=56コア）が占有されます
  - 他のユーザーのジョブに使われることはありません

Oakbridge-CX ノードのブロック図
1ノード：2CPU（ソケット）



28コア
Socket #0

28コア
Socket #1

基本的に1ソケット（1CPU）のみ使う

ジョブスクリプト(1/2)go1.sh

- **/work/gt00/t00XXX/multicore/omp/run/go1.sh**
- スケジューラへの指令 ＋ シェルスクリプト

```
#!/bin/sh
#PJM -N “test1”                    ジョブ名称（省略可）
#PJM -L rscgrp=tutorial            実行キュー名
#PJM -L node=1                     ノード数（原則＝1）
#PJM --omp thread=24               スレッド数（1-56，原則1-28）
#PJM -L elapse=00:15:00            実行時間
#PJM -g gt00                       グループ名（財布）
#PJM -j
#PJM -e err                        エラー出力ファイル
#PJM -o test1.lst                     標準出力ファイル

export KMP_AFFINITY=granularity=fine,compact
./sol20                                プログラム実行
```

```
export KMP_AFFINITY=granularity=fine,compact
各スレッドがSocket#0の0番から始まる各コアに順番に割り当てられる
```

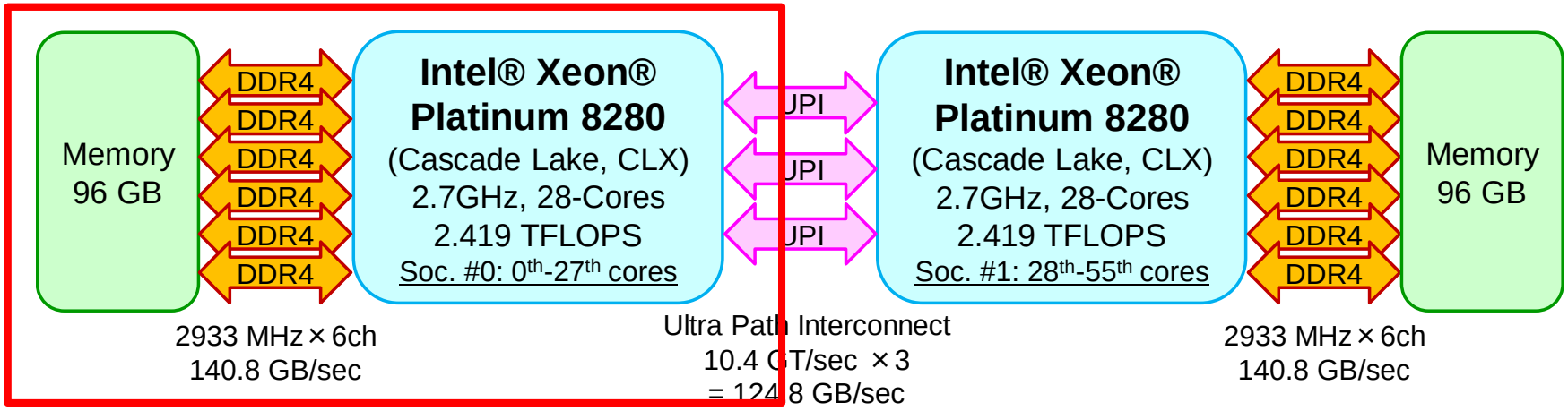
ジョブスクリプト(1/2)go2.sh

- **/work/gt00/t00XXX/multicore/omp/run/go2.sh**
- スケジューラへの指令 ＋ シェルスクリプト

```
#!/bin/sh
#PJM -N “test2”                  ジョブ名称（省略可）
#PJM -L rscgrp=tutorial          実行キュー名
#PJM -L node=1                   ノード数（原則＝1）
#PJM --omp thread=24             スレッド数（1-56，原則1-28）
#PJM -L elapse=00:15:00          実行時間
#PJM -g gt00                     グループ名（財布）
#PJM -j
#PJM -e err                      エラー出力ファイル
#PJM -o test2.lst                   標準出力ファイル



./sol20                          プログラム実行
```

各スレッドがSocket#0/#1の各コアにランダムに割り当てられる

Oakbridge-CX ノードのブロック図
1ノード：2CPU（ソケット）



28コア
Socket #0

28コア
Socket #1

go1.sh: ソケット#0のコアのみが使われる

```
export KMP_AFFINITY=granularity=fine,compact
```

go2.sh: ソケット#0と#1のコアがランダムに使われる。
go1.shよりも早い場合がある。

ジョブ投入

```
>$ cd /work/gt00/t00XYZ
>$ cd multicore/omp/run
>$ pjsub go1.sh

>$ cat test1.lst
```

INPUT.DAT

```
128 128 128                          NX NY NZ
1.00e-0  1.00e-00 1.00e-00           DX/DY/DZ
1.0e-08                              EPSICCG
```

利用可能なキュー

- 以下の2種類のキューを利用可能
- 最大8ノードを使える
  - **lecture**
    - 8ノード（448コア），15分，アカウント有効期間中利用可能
    - 全教育ユーザーで共有
  - **tutorial**
    - 8ノード（448コア），15分，講義・演習実施時間帯
    - **lecture**よりは多くのジョブを投入可能（混み具合による）

様々なコマンド

• ジョブ実行	`pjsub SCRIPT NAME`
• ジョブ実行状況	`pjstat`
• ジョブ停止	`pjdel JOB ID`
• ジョブキューの状況	`pjstat --rsc`
• ジョブキューの状況(詳細)	`pjstat --rsc -x`
• 実行ジョブ情報	`pjstat -a`
• ジョブ実行履歴	`pjstat -H`
• ジョブ実行制限	`pjstat --limit`

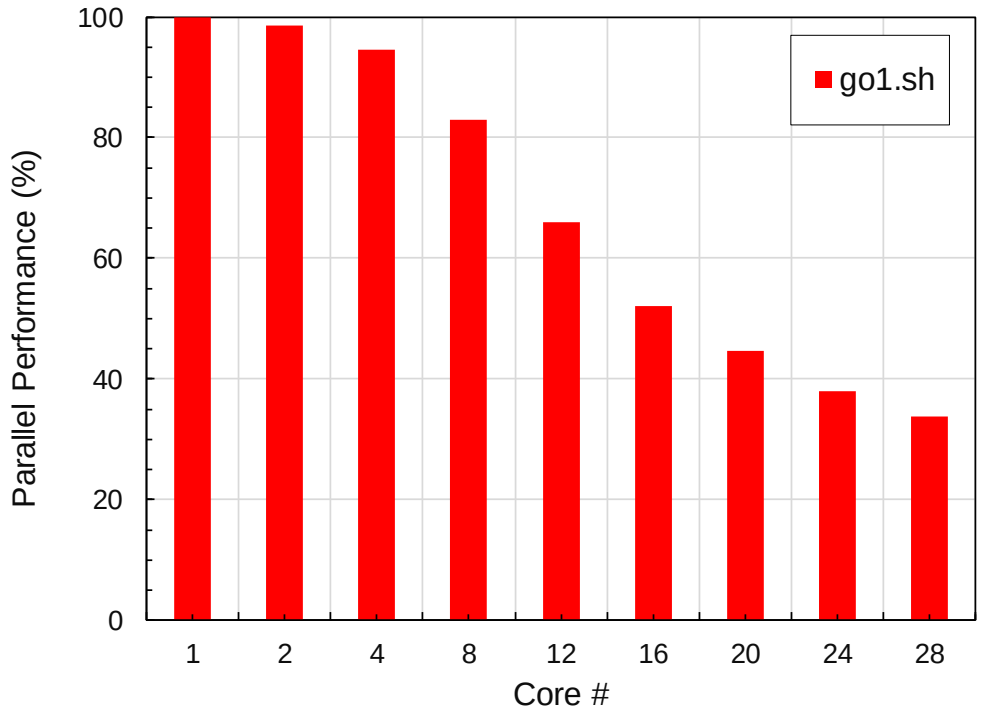
```
[t00XYZ@obcx04 run]$ pjsub go1.sh
[INFO] PJM 0000 pjsub Job 292019 submitted.

[t00XYZ@obcx04 run]$ pjsub go2.sh
[INFO] PJM 0000 pjsub Job 292020 submitted.

[t00XYZ@obcx04 run]$ pjstat
Oakbridge-CX scheduled stop time: 2020/04/24(Fri) 09:00:00 (Remain:  3days 23:09:15)

JOB_ID       JOB_NAME   STATUS  PROJECT  RSCGROUP       START_DATE        ELAPSE           TOKEN NODE
292019       test1      RUNNING gt00     lecture        04/20 09:50:42<   00:00:02             -    1
292020       test2      QUEUED  gt00     lecture        --/-- --:--:--    00:00:00             -    1

[t00XYZ@obcx04 run]$ pjstat
Oakbridge-CX scheduled stop time: 2020/04/24(Fri) 09:00:00 (Remain:  3days 23:09:12)

JOB_ID       JOB_NAME   STATUS  PROJECT  RSCGROUP       START_DATE        ELAPSE           TOKEN NODE
292019       test1      RUNNING gt00     lecture        04/20 09:50:42<   00:00:06             -    1
292020       test2      RUNNING gt00     lecture        04/20 09:50:46<   00:00:02             -    1

[t00XYZ@obcx04 run]$ pjdel 292020
[INFO] PJM 0100 pjdel Job 292020 canceled.

[t00XYZ@obcx04 run]$ pjstat
Oakbridge-CX scheduled stop time: 2020/04/24(Fri) 09:00:00 (Remain:  3days 23:09:04)

JOB_ID       JOB_NAME   STATUS  PROJECT  RSCGROUP       START_DATE        ELAPSE           TOKEN NODE
292019       test1      RUNNING gt00     lecture        04/20 09:50:42<   00:00:14             -    1

[t00XYZ@obcx04 run]$ pjstat
Oakbridge-CX scheduled stop time: 2020/04/24(Fri) 09:00:00 (Remain:  3days 23:07:14)

No unfinished job found.
```

```
[t00XYZ@obcx04 ~]$ pjstat --rsc
RSCGRP                  STATUS              NODE
lecture                 [ENABLE,START]        32
tutorial                [DISABLE,STOP]        64

[t00XYZ@obcx04 ~]$ pjstat --rsc -x
RSCGRP                  STATUS            MIN_NODE MAX_NODE MAX_ELAPSE REMAIN_ELAPSE MEM(GB) PROJECT
lecture                 [ENABLE,START]           1        8   00:15:00      00:15:00     168 gt00
tutorial                [DISABLE,STOP]           1        8   00:15:00      --:--:--     168 gt00

[t00XYZ@obcx04 ~]$ pjstat -a
Oakbridge-CX scheduled stop time: 2020/04/24(Fri) 09:00:00 (Remain:  3days 23:19:29)

JOB_ID       JOB_NAME   STATUS  PROJECT  RSCGROUP       START_DATE        ELAPSE           TOKEN NODE
284147       ********** RUNNING ******** -              04/19 12:58:20    --:--:--             -    -
284149       ********** RUNNING ******** -              04/19 11:50:18    --:--:--             -    -
284159       ********** RUNNING ******** -              04/19 19:16:10    --:--:--             -    -
289904       ********** RUNNING ******** small          04/18 19:59:41    37:40:50             -    2
289909       ********** RUNNING ******** small          04/19 01:02:58    32:37:33             -    2
(…)

[t00XYZ@obcx04 ~]$ pjstat -H
Oakbridge-CX scheduled stop time: 2020/04/24(Fri) 09:00:00 (Remain:  3days 23:19:16)

JOB_ID       JOB_NAME   STATUS  PROJECT  RSCGROUP       START_DATE        ELAPSE           TOKEN NODE
290914       test       END     gt00     lecture        04/18 12:46:24    00:01:44             -    1
290913       test       END     gt00     lecture        04/18 12:46:06    00:02:07             -    1
290915       test       END     gt00     lecture        04/18 12:49:26    00:00:59             -    1
(…)

[t00XYZ@obcx04 ~]$ pjstat --limit
PROJECT     ACCEPT       RUN  BULK_RUN      NODE
gt00        0/  80     0/  20    0/ 256     0/   -
```

PCG計算時間: Etime-Stime

NX=NY=NZ=128, go1.sh

実行時間:やや不安定:5回くらい測定して最速時間採用



Thread #	sec	Speed-up
1	27.201	1.000
2	13.796	1.972
4	7.185	3.786
8	4.099	6.637
12	3.435	7.918
16	3.260	8.343
20	3.048	8.925
24	2.982	9.123
28	2.877	9.456

Large
大

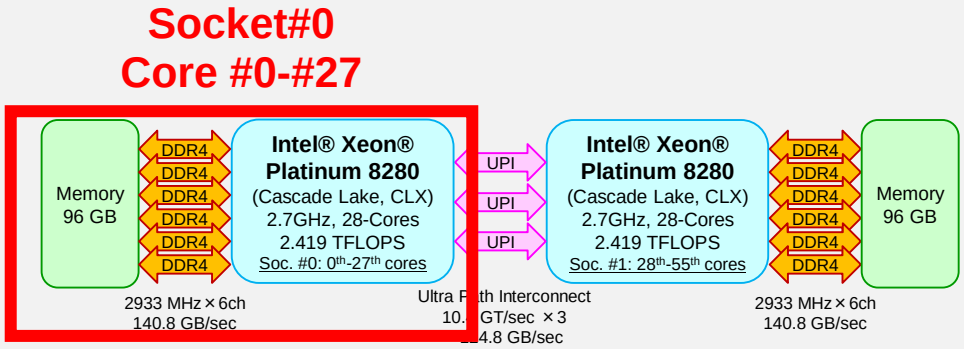
Granularity
粒度
Problem Size/Thread

Small
小

go1.sh

Only cores on a single socket used

```
#!/bin/sh
#PJM -N "test1"
#PJM -L rscgrp=tutorial
#PJM -L node=1
#PJM --omp thread=24                    1,2,4,8,12,16,20,24,28
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o test1.lst

export KMP_AFFINITY=granularity=fine,compact
./sol20
./sol20
./sol20
./sol20
./sol20
```

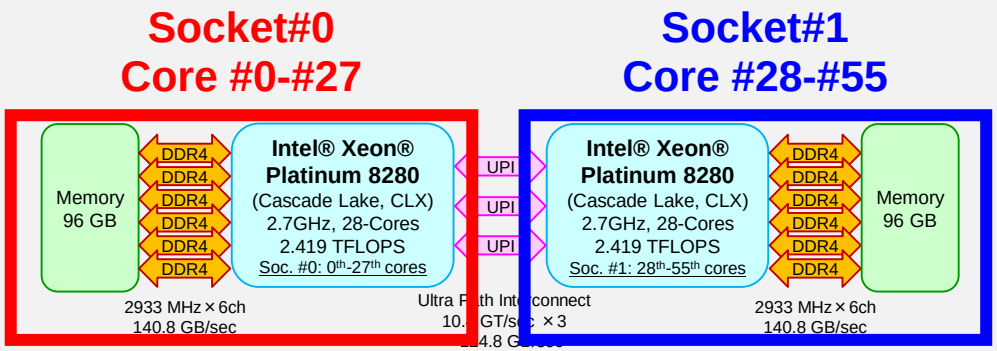


go2.sh

cores are randomly selected from 2 sockets

```
#!/bin/sh
#PJM -N “test1”
#PJM -L rscgrp=tutorial
#PJM -L node=1
#PJM --omp thread=24
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o test1.lst


./sol20
./sol20
./sol20
./sol20
./sol20
```

1,2,4,8,12,16,20,24,28



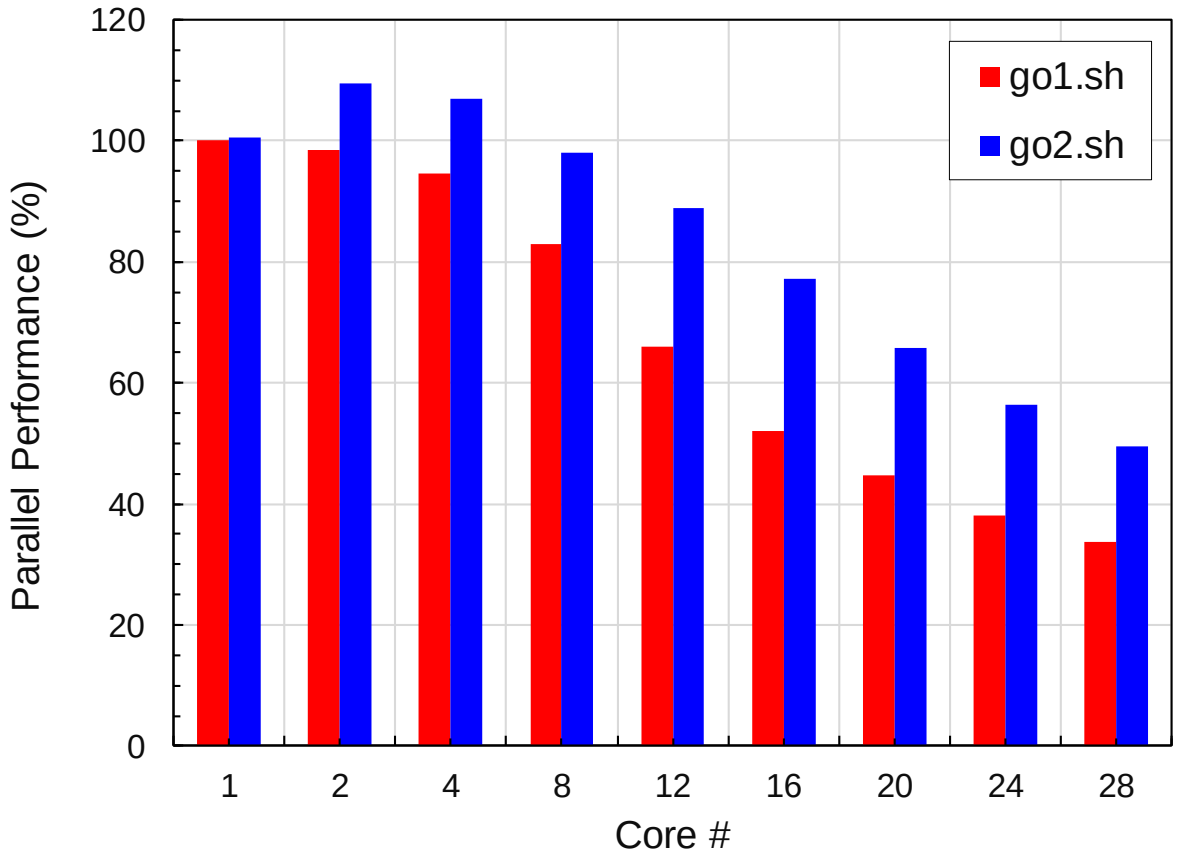
Results: Parallel Performance

NX=NY=NZ=128

Measurement: 5 times, best case

“go2.sh” is generally better

based on “go1.sh” with 1 thread

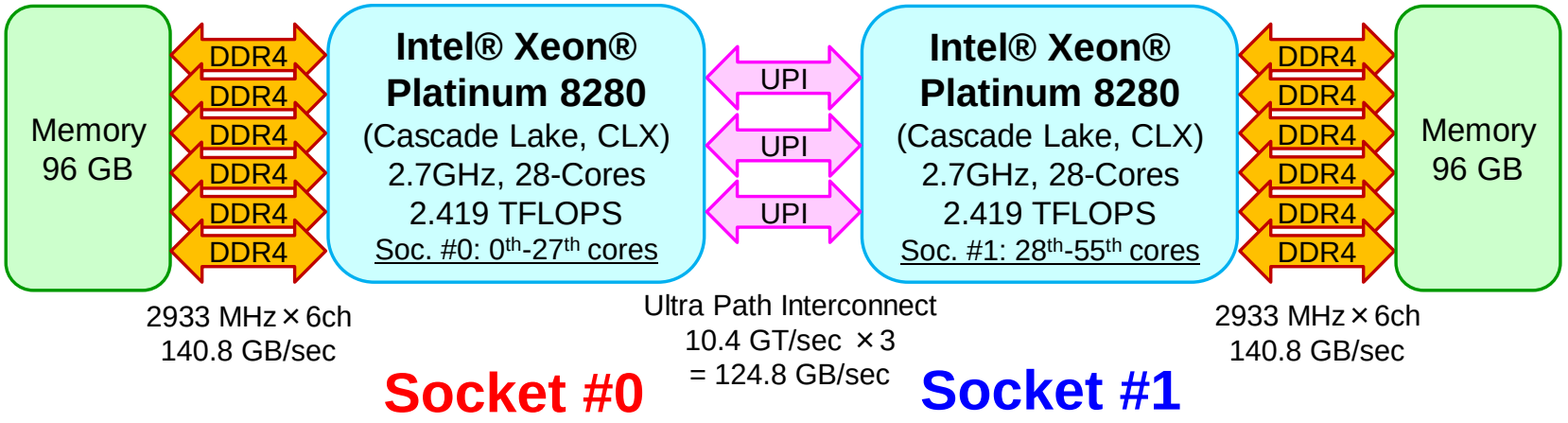


go1.sh

go2.sh

Oakbridge-CX (OBCX)

1-node: 2-CPU's/sockets

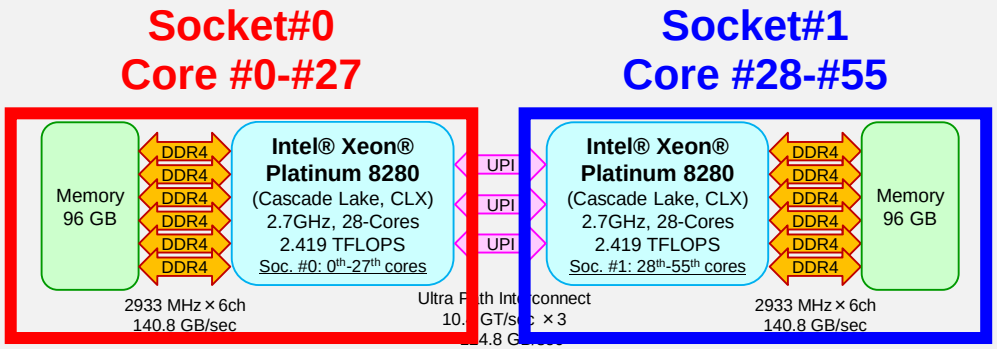


- Oakbridfe-CX
  - 各ノードは：2ソケット（CPU）：Intel Cascade Lake (CLX)
  - 各ソケットは28コア（合計56コア）
- 各ソケットの各コアは他のソケットのメモリにアクセス可能（遅い）: NUMA (Non-Uniform Memory Access)
- ローカルメモリを利用することがより効率的
  - 本講習会では基本的に1ソケットを使用
  - go2.shはソケット当たりコア数減少・効率的⇒遠隔メモリアクセス可能性

go3.sh

```
#!/bin/sh
#PJM -N “test2”
#PJM -L rscgrp=lecture5
#PJM -L node=1
#PJM --omp thread=24
#PJM -L elapse=00:15:00
#PJM -g gt55
#PJM -j
#PJM -e err
#PJM -o test3.lst

./sol20
```

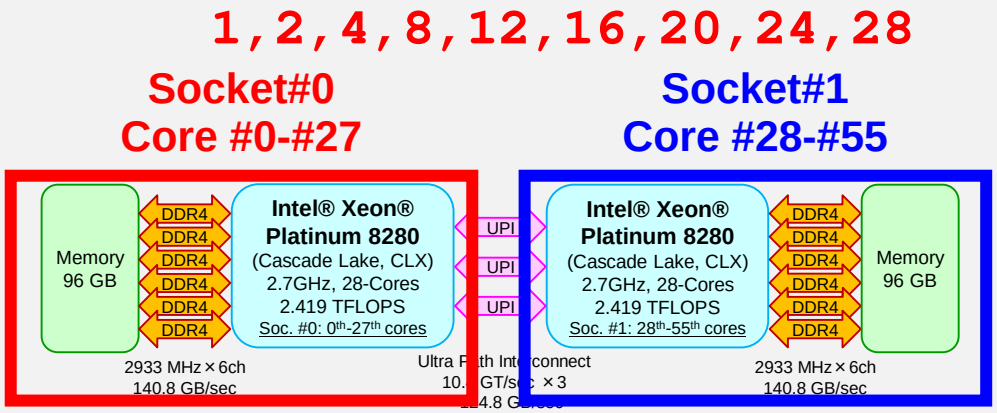
1,2,4,8,12,16,20,24,28




```
export KMP_AFFINITY=granularity=fine,compact
./sol20
export KMP_AFFINITY=granularity=fine,balanced
./sol20
export KMP_AFFINITY=granularity=fine,scatter
./sol20
```

go4.sh: “verbose” provides topology mapping (core-thread)

```
#!/bin/sh
#PJM -N “test2”
#PJM -L rscgrp=lecture5
#PJM -L node=1
#PJM --omp thread=24
#PJM -L elapse=00:15:00
#PJM -g gt55
#PJM -j
#PJM -e err
#PJM -o test3.lst
```

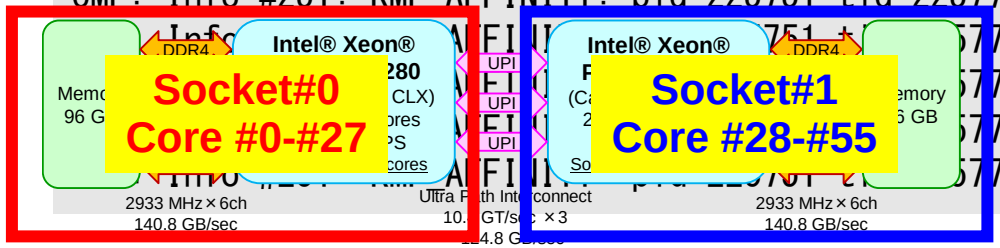



```
export KMP_AFFINITY=verbose
./sol20
export KMP_AFFINITY=granularity=fine,compact,verbose
./sol20
export KMP_AFFINITY=granularity=fine,balanced,verbose
./sol20
export KMP_AFFINITY=granularity=fine,scatter,verbose
./sol20
```

Results(none): 128^3, 24 threads

2.044 sec.

```
OMP: Info #251: KMP_AFFINITY: pid 225751 tid 225751 thread 0 bound to OS proc set 0-55
OMP: Info #251: KMP_AFFINITY: pid 225751 tid 225753 thread 2 bound to OS proc set 0-55
OMP: Info #251: KMP_AFFINITY: pid 225751 tid 225752 thread 1 bound to OS proc set 0-55
OMP: Info #251: KMP_AFFINITY: pid 225751 tid 225754 thread 3 bound to OS proc set 0-55
OMP: Info #251: KMP_AFFINITY: pid 225751 tid 225755 thread 4 bound to OS proc set 0-55
OMP: Info #251: KMP_AFFINITY: pid 225751 tid 225756 thread 5 bound to OS proc set 0-55
OMP: Info #251: KMP_AFFINITY: pid 225751 tid 225757 thread 6 bound to OS proc set 0-55
OMP: Info #251: KMP_AFFINITY: pid 225751 tid 225758 thread 7 bound to OS proc set 0-55
OMP: Info #251: KMP_AFFINITY: pid 225751 tid 225759 thread 8 bound to OS proc set 0-55
OMP: Info #251: KMP_AFFINITY: pid 225751 tid 225760 thread 9 bound to OS proc set 0-55
OMP: Info #251: KMP_AFFINITY: pid 225751 tid 225761 thread 10 bound to OS proc set 0-55
OMP: Info #251: KMP_AFFINITY: pid 225751 tid 225762 thread 11 bound to OS proc set 0-55
OMP: Info #251: KMP_AFFINITY: pid 225751 tid 225763 thread 12 bound to OS proc set 0-55
OMP: Info #251: KMP_AFFINITY: pid 225751 tid 225764 thread 13 bound to OS proc set 0-55
OMP: Info #251: KMP_AFFINITY: pid 225751 tid 225765 thread 14 bound to OS proc set 0-55
OMP: Info #251: KMP_AFFINITY: pid 225751 tid 225766 thread 15 bound to OS proc set 0-55
OMP: Info #251: KMP_AFFINITY: pid 225751 tid 225767 thread 16 bound to OS proc set 0-55
OMP: Info #251: KMP_AFFINITY: pid 225751 tid 225768 thread 17 bound to OS proc set 0-55
OMP: Info #251: KMP_AFFINITY: pid 225751 tid 225769 thread 18 bound to OS proc set 0-55
OMP: Info #251: KMP_AFFINITY: pid 225751 tid 225770 thread 19 bound to OS proc set 0-55
OMP: Info #251: KMP_AFFINITY: pid 225751 tid 225771 thread 20 bound to OS proc set 0-55
OMP: Info #251: KMP_AFFINITY: pid 225751 tid 225772 thread 21 bound to OS proc set 0-55
OMP: Info #251: KMP_AFFINITY: pid 225751 tid 225774 thread 23 bound to OS proc set 0-55
OMP: Info #251: KMP_AFFINITY: pid 225751 tid 225773 thread 22 bound to OS proc set 0-55
```

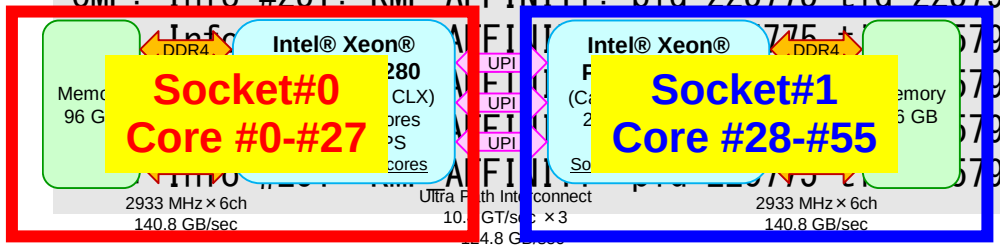


Thread ID

Core ID

Results(compact): 128^3, 24 threads 2.978 sec., All threads on Soc.#0

```
OMP: Info #251: KMP_AFFINITY: pid 225775 tid 225775 thread 0 bound to OS proc set 0
OMP: Info #251: KMP_AFFINITY: pid 225775 tid 225776 thread 1 bound to OS proc set 1
OMP: Info #251: KMP_AFFINITY: pid 225775 tid 225777 thread 2 bound to OS proc set 2
OMP: Info #251: KMP_AFFINITY: pid 225775 tid 225778 thread 3 bound to OS proc set 3
OMP: Info #251: KMP_AFFINITY: pid 225775 tid 225780 thread 5 bound to OS proc set 5
OMP: Info #251: KMP_AFFINITY: pid 225775 tid 225779 thread 4 bound to OS proc set 4
OMP: Info #251: KMP_AFFINITY: pid 225775 tid 225781 thread 6 bound to OS proc set 6
OMP: Info #251: KMP_AFFINITY: pid 225775 tid 225782 thread 7 bound to OS proc set 7
OMP: Info #251: KMP_AFFINITY: pid 225775 tid 225783 thread 8 bound to OS proc set 8
OMP: Info #251: KMP_AFFINITY: pid 225775 tid 225784 thread 9 bound to OS proc set 9
OMP: Info #251: KMP_AFFINITY: pid 225775 tid 225785 thread 10 bound to OS proc set 10
OMP: Info #251: KMP_AFFINITY: pid 225775 tid 225786 thread 11 bound to OS proc set 11
OMP: Info #251: KMP_AFFINITY: pid 225775 tid 225787 thread 12 bound to OS proc set 12
OMP: Info #251: KMP_AFFINITY: pid 225775 tid 225788 thread 13 bound to OS proc set 13
OMP: Info #251: KMP_AFFINITY: pid 225775 tid 225789 thread 14 bound to OS proc set 14
OMP: Info #251: KMP_AFFINITY: pid 225775 tid 225790 thread 15 bound to OS proc set 15
OMP: Info #251: KMP_AFFINITY: pid 225775 tid 225791 thread 16 bound to OS proc set 16
OMP: Info #251: KMP_AFFINITY: pid 225775 tid 225792 thread 17 bound to OS proc set 17
OMP: Info #251: KMP_AFFINITY: pid 225775 tid 225793 thread 18 bound to OS proc set 18
OMP: Info #251: KMP_AFFINITY: pid 225775 tid 225795 thread 20 bound to OS proc set 20
                                                  thread 19 bound to OS proc set 19
                                                  thread 21 bound to OS proc set 21
                                                  thread 22 bound to OS proc set 22
                                                  thread 23 bound to OS proc set 23
```



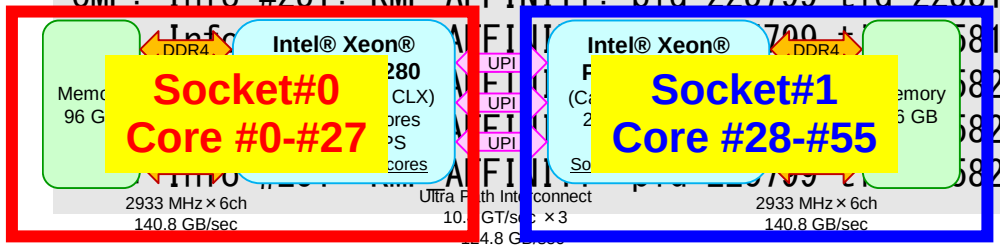
Thread ID

Core ID

Results(balanced): 128^3, 24 threads

2.007 sec., 0-11:Soc.#0, 12-23:Soc.#1

```
OMP: Info #251: KMP_AFFINITY: pid 225799 tid 225799 thread 0 bound to OS proc set 0
OMP: Info #251: KMP_AFFINITY: pid 225799 tid 225800 thread 1 bound to OS proc set 1
OMP: Info #251: KMP_AFFINITY: pid 225799 tid 225801 thread 2 bound to OS proc set 2
OMP: Info #251: KMP_AFFINITY: pid 225799 tid 225802 thread 3 bound to OS proc set 3
OMP: Info #251: KMP_AFFINITY: pid 225799 tid 225803 thread 4 bound to OS proc set 4
OMP: Info #251: KMP_AFFINITY: pid 225799 tid 225804 thread 5 bound to OS proc set 5
OMP: Info #251: KMP_AFFINITY: pid 225799 tid 225805 thread 6 bound to OS proc set 6
OMP: Info #251: KMP_AFFINITY: pid 225799 tid 225806 thread 7 bound to OS proc set 7
OMP: Info #251: KMP_AFFINITY: pid 225799 tid 225807 thread 8 bound to OS proc set 8
OMP: Info #251: KMP_AFFINITY: pid 225799 tid 225808 thread 9 bound to OS proc set 9
OMP: Info #251: KMP_AFFINITY: pid 225799 tid 225809 thread 10 bound to OS proc set 10
OMP: Info #251: KMP_AFFINITY: pid 225799 tid 225810 thread 11 bound to OS proc set 11
OMP: Info #251: KMP_AFFINITY: pid 225799 tid 225811 thread 12 bound to OS proc set 28
OMP: Info #251: KMP_AFFINITY: pid 225799 tid 225812 thread 13 bound to OS proc set 29
OMP: Info #251: KMP_AFFINITY: pid 225799 tid 225813 thread 14 bound to OS proc set 30
OMP: Info #251: KMP_AFFINITY: pid 225799 tid 225814 thread 15 bound to OS proc set 31
OMP: Info #251: KMP_AFFINITY: pid 225799 tid 225815 thread 16 bound to OS proc set 32
OMP: Info #251: KMP_AFFINITY: pid 225799 tid 225816 thread 17 bound to OS proc set 33
OMP: Info #251: KMP_AFFINITY: pid 225799 tid 225817 thread 18 bound to OS proc set 34
OMP: Info #251: KMP_AFFINITY: pid 225799 tid 225818 thread 19 bound to OS proc set 35
OMP: Info #251: KMP_AFFINITY: pid 225799 tid 225819 thread 20 bound to OS proc set 36
OMP: Info #251: KMP_AFFINITY: pid 225799 tid 225820 thread 21 bound to OS proc set 37
OMP: Info #251: KMP_AFFINITY: pid 225799 tid 225821 thread 22 bound to OS proc set 38
OMP: Info #251: KMP_AFFINITY: pid 225799 tid 225822 thread 23 bound to OS proc set 39
```



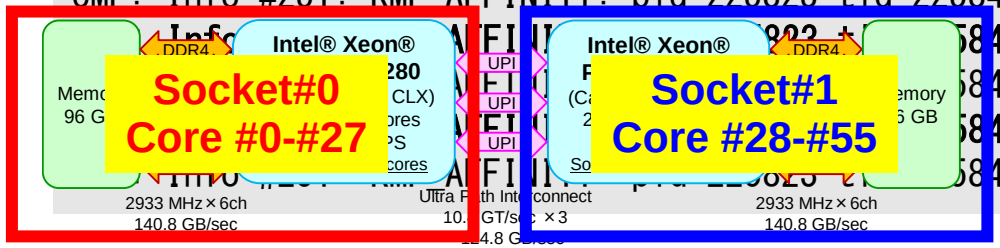
Thread ID

Core ID

Results(scatter): 128^3, 24 threads

2.175 sec.

```
OMP: Info #251: KMP_AFFINITY: pid 225823 tid 225823 thread 0 bound to OS proc set 0
OMP: Info #251: KMP_AFFINITY: pid 225823 tid 225824 thread 1 bound to OS proc set 28
OMP: Info #251: KMP_AFFINITY: pid 225823 tid 225825 thread 2 bound to OS proc set 1
OMP: Info #251: KMP_AFFINITY: pid 225823 tid 225826 thread 3 bound to OS proc set 29
OMP: Info #251: KMP_AFFINITY: pid 225823 tid 225827 thread 4 bound to OS proc set 2
OMP: Info #251: KMP_AFFINITY: pid 225823 tid 225828 thread 5 bound to OS proc set 30
OMP: Info #251: KMP_AFFINITY: pid 225823 tid 225829 thread 6 bound to OS proc set 3
OMP: Info #251: KMP_AFFINITY: pid 225823 tid 225830 thread 7 bound to OS proc set 31
OMP: Info #251: KMP_AFFINITY: pid 225823 tid 225831 thread 8 bound to OS proc set 4
OMP: Info #251: KMP_AFFINITY: pid 225823 tid 225832 thread 9 bound to OS proc set 32
OMP: Info #251: KMP_AFFINITY: pid 225823 tid 225833 thread 10 bound to OS proc set 5
OMP: Info #251: KMP_AFFINITY: pid 225823 tid 225834 thread 11 bound to OS proc set 33
OMP: Info #251: KMP_AFFINITY: pid 225823 tid 225835 thread 12 bound to OS proc set 6
OMP: Info #251: KMP_AFFINITY: pid 225823 tid 225836 thread 13 bound to OS proc set 34
OMP: Info #251: KMP_AFFINITY: pid 225823 tid 225837 thread 14 bound to OS proc set 7
OMP: Info #251: KMP_AFFINITY: pid 225823 tid 225838 thread 15 bound to OS proc set 35
OMP: Info #251: KMP_AFFINITY: pid 225823 tid 225839 thread 16 bound to OS proc set 8
OMP: Info #251: KMP_AFFINITY: pid 225823 tid 225840 thread 17 bound to OS proc set 36
OMP: Info #251: KMP_AFFINITY: pid 225823 tid 225841 thread 18 bound to OS proc set 9
OMP: Info #251: KMP_AFFINITY: pid 225823 tid 225842 thread 19 bound to OS proc set 37
OMP: Info #251: KMP_AFFINITY: pid 225823 tid 225843 thread 20 bound to OS proc set 10
OMP: Info #251: KMP_AFFINITY: pid 225823 tid 225844 thread 21 bound to OS proc set 38
OMP: Info #251: KMP_AFFINITY: pid 225823 tid 225845 thread 22 bound to OS proc set 11
OMP: Info #251: KMP_AFFINITY: pid 225823 tid 225846 thread 23 bound to OS proc set 39
```



Thread ID

Core ID

Exercises

- Effect of problem size (NX, NY, NZ)
- Effect of Thread # (omp_thread: 1-28)

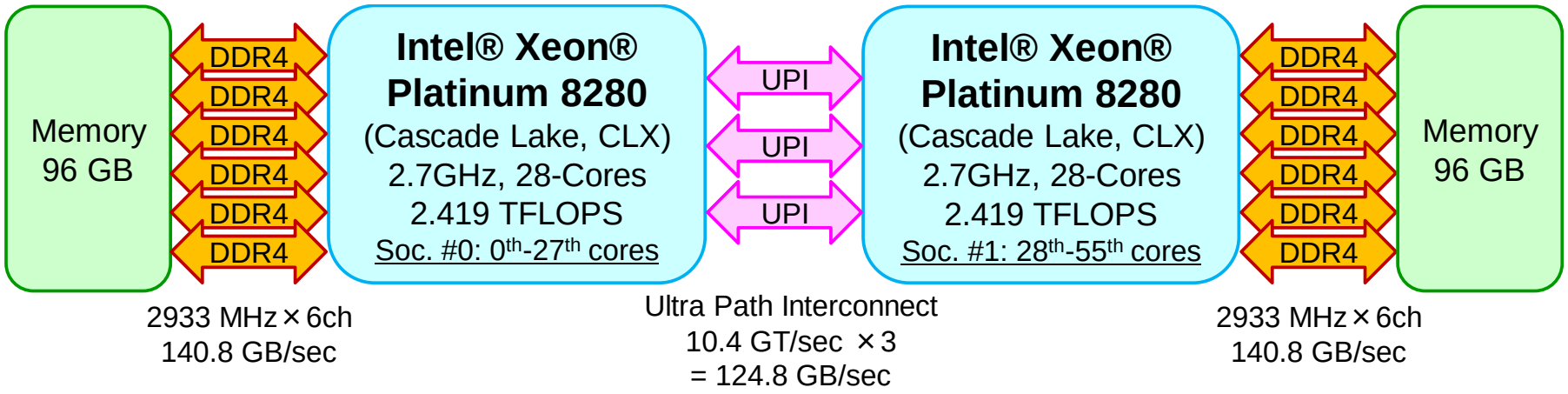
- OpenMP
- Login to OBCX
- Parallel Version of the Code by OpenMP
- **STREAM**
- Data Dependency

何故28倍にならないか?

- 28スレッドがメモリにアクセスすると，1スレッドの場合と比較して，スレッド当り（コア当り）メモリ性能は低下
  - 飽和
- 疎行列はmemory-boundなためその傾向がより顕著
  - 疎行列計算の高速化：研究途上の課題
- 問題規模が比較的小さい

- 何故go2.shが良いか？

Category	Capacity	X-Way Set Associative	Cache Line
L1$Data	32 KB/core	8-Way	64B
L1$Instruction	32 KB/core	8-Way	64B
L2	1.00 MB/core	16-Way	64B
L3	38.5 MB/socket	11-Way	64B



疎行列・密行列

```
do i= 1, N
  Y(i)= D(i)*X(i)
  do k= index(i-1)+1, index(i)
    kk= item(k)
    Y(i)= Y(i) + AMAT(k)*X(kk)
  enddo
enddo
```

```
do j= 1, N
  Y(j)= 0.d0
  do i= 1, N
    Y(j)= Y(j) + A(i,j)*X(i)
  enddo
enddo
```

- “X” in RHS
  - 密行列：連続アクセス，キャッシュ有効利用
  - 疎行列：連続性は保証されず，キャッシュを有効に活用できず
    - より「memory-bound」

GeoFEM Benchmark

ICCG法の性能（固体力学向け）

	SR11K/J2	SR16K/M1	T2K	FX10	京
Core #/Node	16	32	16	16	8
Peak Performance (GFLOPS)	147.2	980.5	147.2	236.5	128.0
STREAM Triad (GB/s)	101.0	264.2	20.0	64.7	43.3
B/F	0.686	0.269	0.136	0.274	0.338
GeoFEM (GFLOPS)	19.0	72.7	4.69	16.0	11.0
% to Peak	12.9	7.41	3.18	6.77	8.59
LLC/core (MB)	18.0	4.00	2.00	0.75	0.75

疎行列ソルバー：Memory-Bound

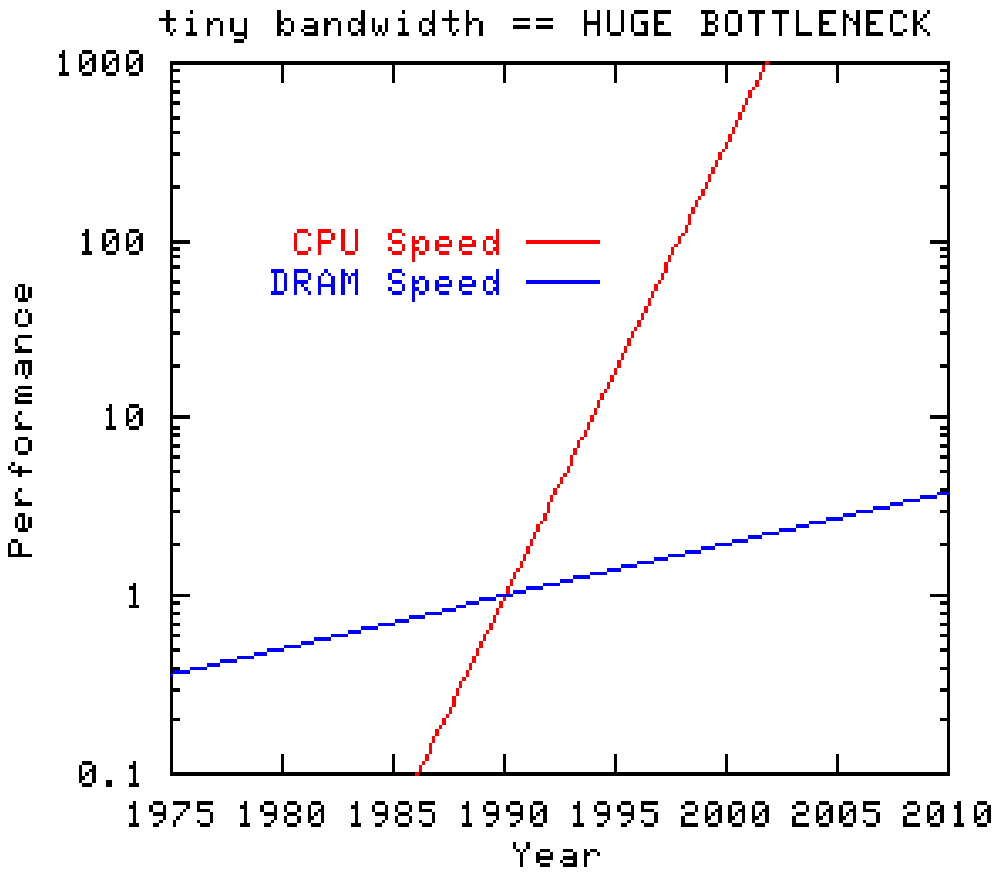
STREAM benchmark

http://www.cs.virginia.edu/stream/

- メモリバンド幅を測定するベンチマーク
  - Copy: c(i)= a(i)
  - Scale: c(i)= s*b(i)
  - Add: c(i)= a(i) + b(i)
  - Triad: c(i)= a(i) + s*b(i)

```
-------------------------------------------------
 Double precision appears to have 16 digits of accuracy
 Assuming 8 bytes per DOUBLE PRECISION word
-------------------------------------------------
 Number of processors =           16
 Array size =    2000000
 Offset     =          0
 The total memory requirement is     732.4 MB (
45.8MB/task)
 You are running each test  10 times
 --
 The *best* time for each test is used
 *EXCLUDING* the first and last iterations
 -----------------------------------------------------
 -----------------------------------------------------
Function     Rate (MB/s)  Avg time    Min time  Max time
Copy:      18334.1898      0.0280      0.0279      0.0280
Scale:     18035.1690      0.0284      0.0284      0.0285
Add:       18649.4455      0.0412      0.0412      0.0413
Triad:     19603.8455      0.0394      0.0392      0.0398
```

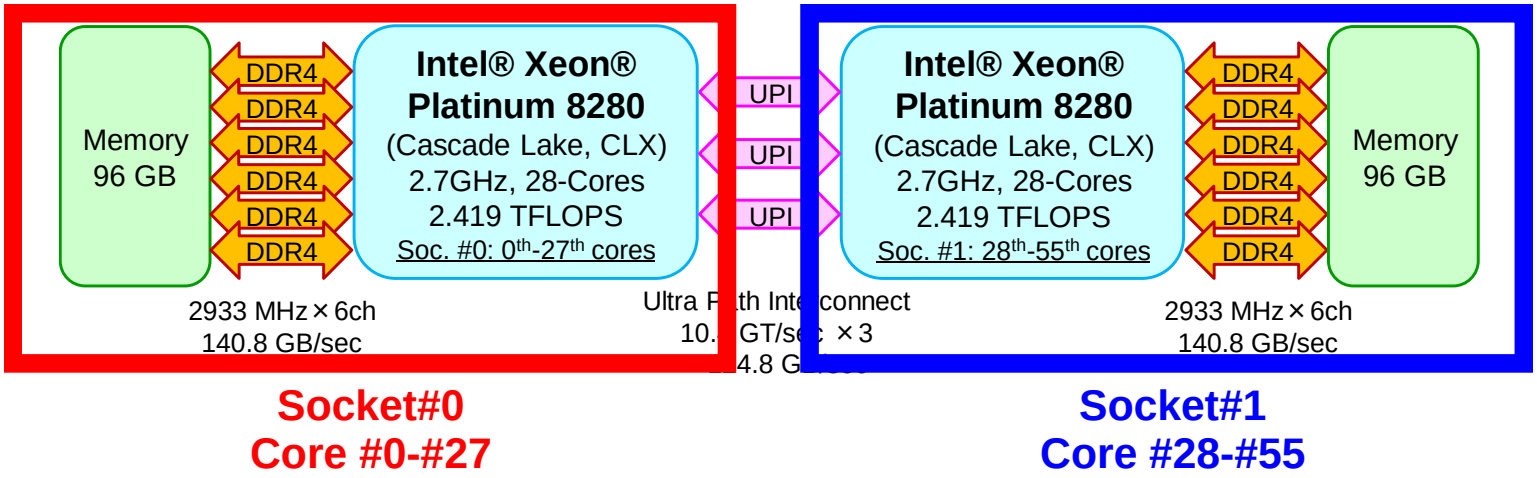
マイクロプロセッサの動向
CPU性能，メモリバンド幅のギャップ



http://www.cs.virginia.edu/stream/

実行：MPIバージョン

```
>$ cd /work/gt00/t00XYZ
>$ cd multicore/stream
>$ mpiifort -align array64byte -O3 -axCORE-AVX512 stream.f -o stream

>$ pjsub XXX.sh
```

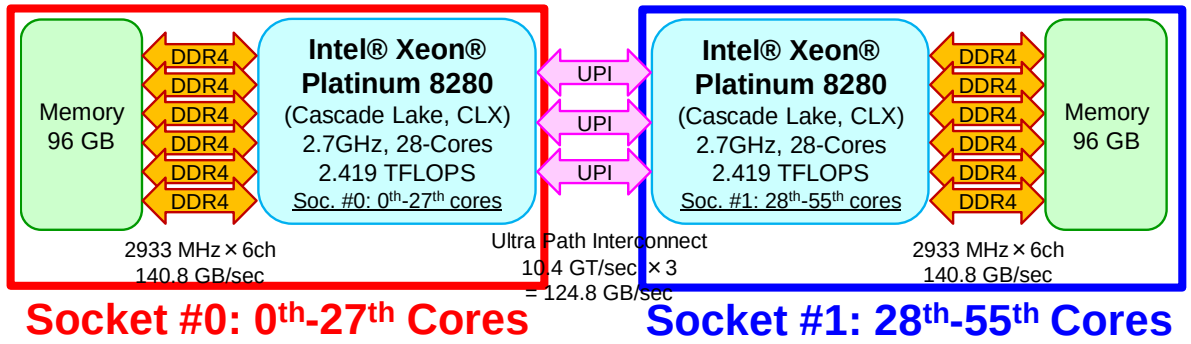


s01.sh: Use 1 core

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=1
#PJM --mpi proc=1
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o s01.lst

mpiexec.hydra -n ${PJM_MPI_PROC} ./stream

export I_MPI_PIN_PROCESSOR_LIST=0
mpiexec.hydra -n ${PJM_MPI_PROC} ./stream
```

Cores are randomly selected ■ **go2.sh random**

Core are specified ■ **go1.sh compact**

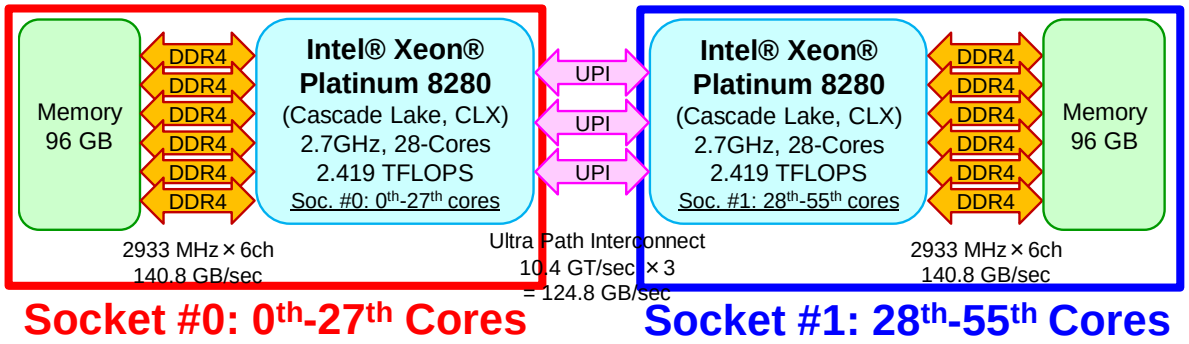


s16.sh: Use 16 cores

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=1
#PJM --mpi proc=16
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o s16.lst

mpiexec.hydra -n ${PJM_MPI_PROC} ./stream

export I_MPI_PIN_PROCESSOR_LIST=0-15
mpiexec.hydra -n ${PJM_MPI_PROC} ./stream
```

Cores are randomly selected ■ **go2.sh random**

Core are specified ■ **go1.sh compact**

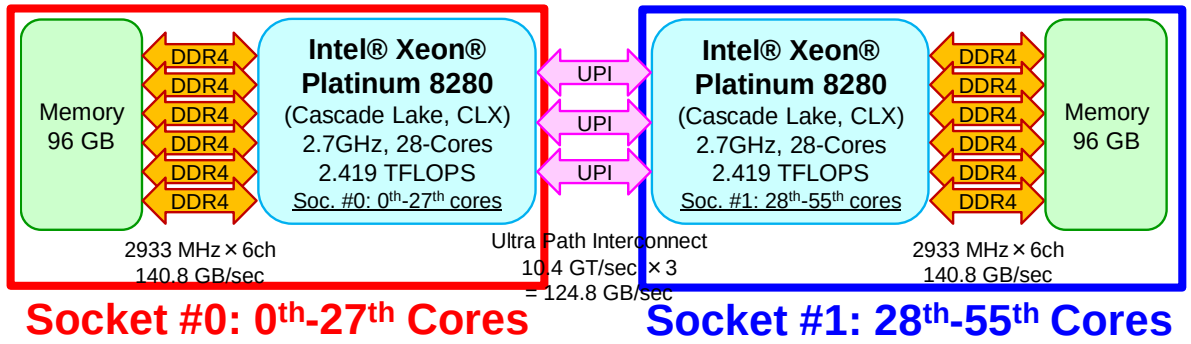


s32.sh: Use 32 cores

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=1
#PJM --mpi proc=32
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o s32.lst

mpiexec.hydra -n ${PJM_MPI_PROC} ./stream

export I_MPI_PIN_PROCESSOR_LIST=0-15,28-43
mpiexec.hydra -n ${PJM_MPI_PROC} ./stream
```

Cores are randomly selected ■ **go2.sh random**

Core are specified ■ **go1.sh compact**

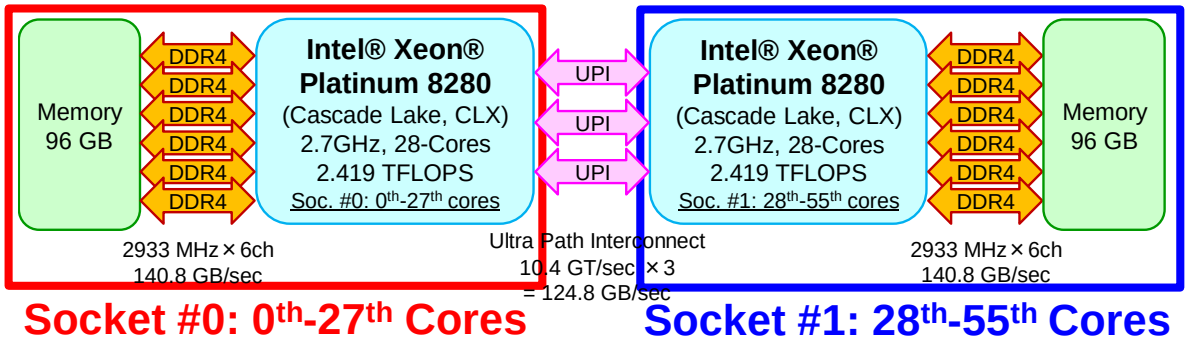


s48.sh: Use 48 cores

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=1
#PJM --mpi proc=48
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o s48.lst

mpiexec.hydra -n ${PJM_MPI_PROC} ./stream

export I_MPI_PIN_PROCESSOR_LIST=0-23,28-51
mpiexec.hydra -n ${PJM_MPI_PROC} ./stream
```

Cores are randomly selected ■ **go2.sh random**

Core are specified ■ **go1.sh compact**

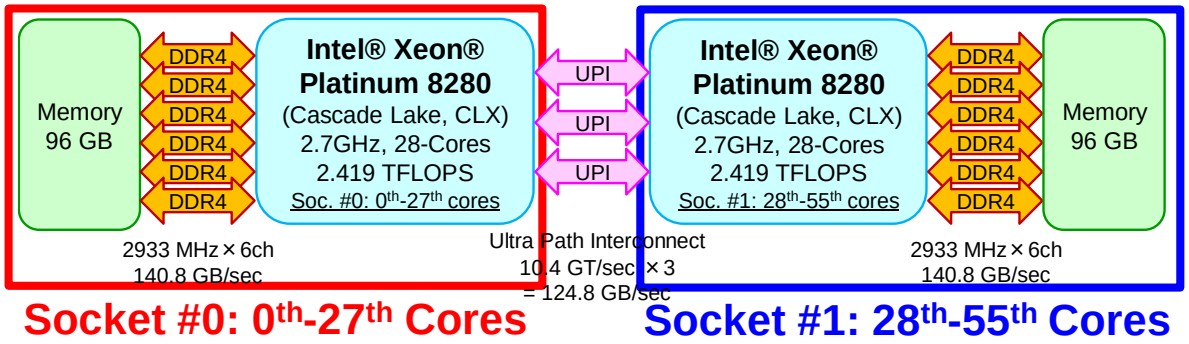


s56.sh: Use 56 cores

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=1
#PJM --mpi proc=56
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o s56.lst

mpiexec.hydra -n ${PJM_MPI_PROC} ./stream

export I_MPI_PIN_PROCESSOR_LIST=0-55
mpiexec.hydra -n ${PJM_MPI_PROC} ./stream
```

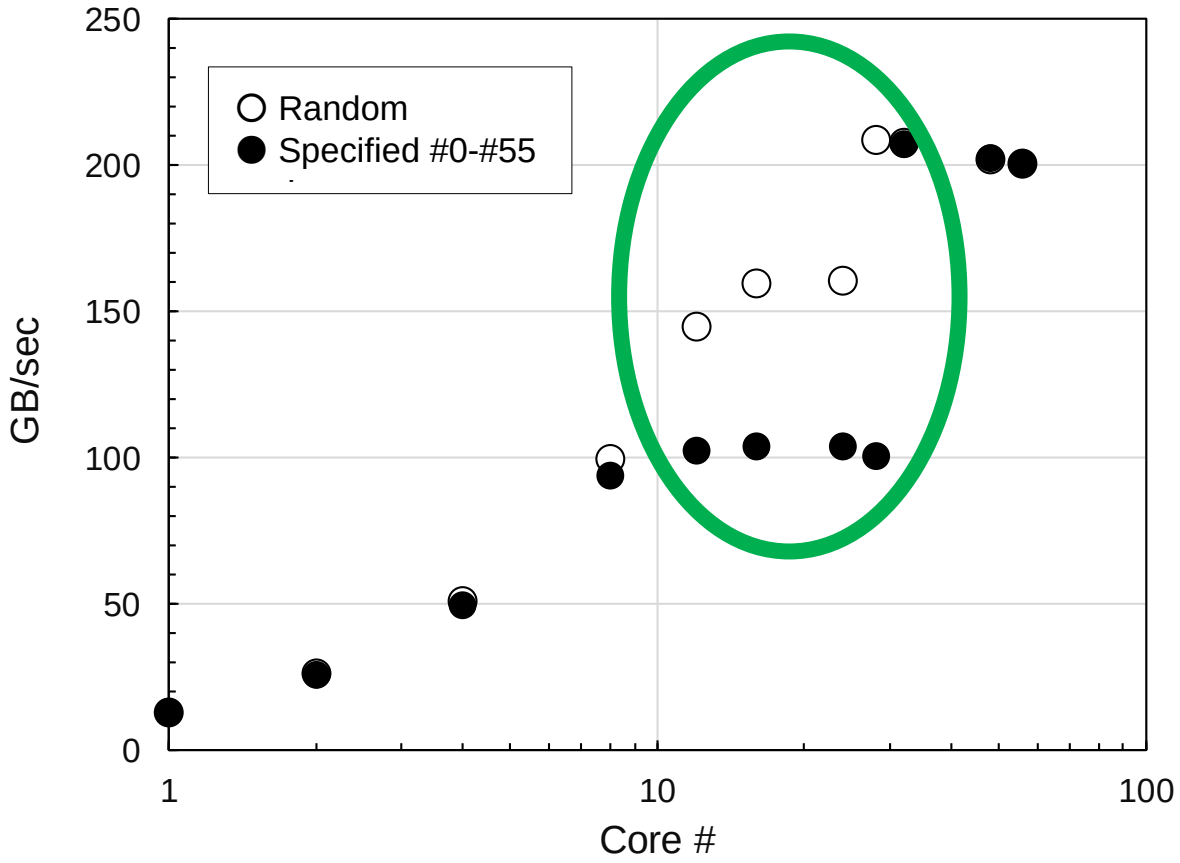
Cores are randomly selected ■ go2.sh random

Core are specified ■ go1.sh compact



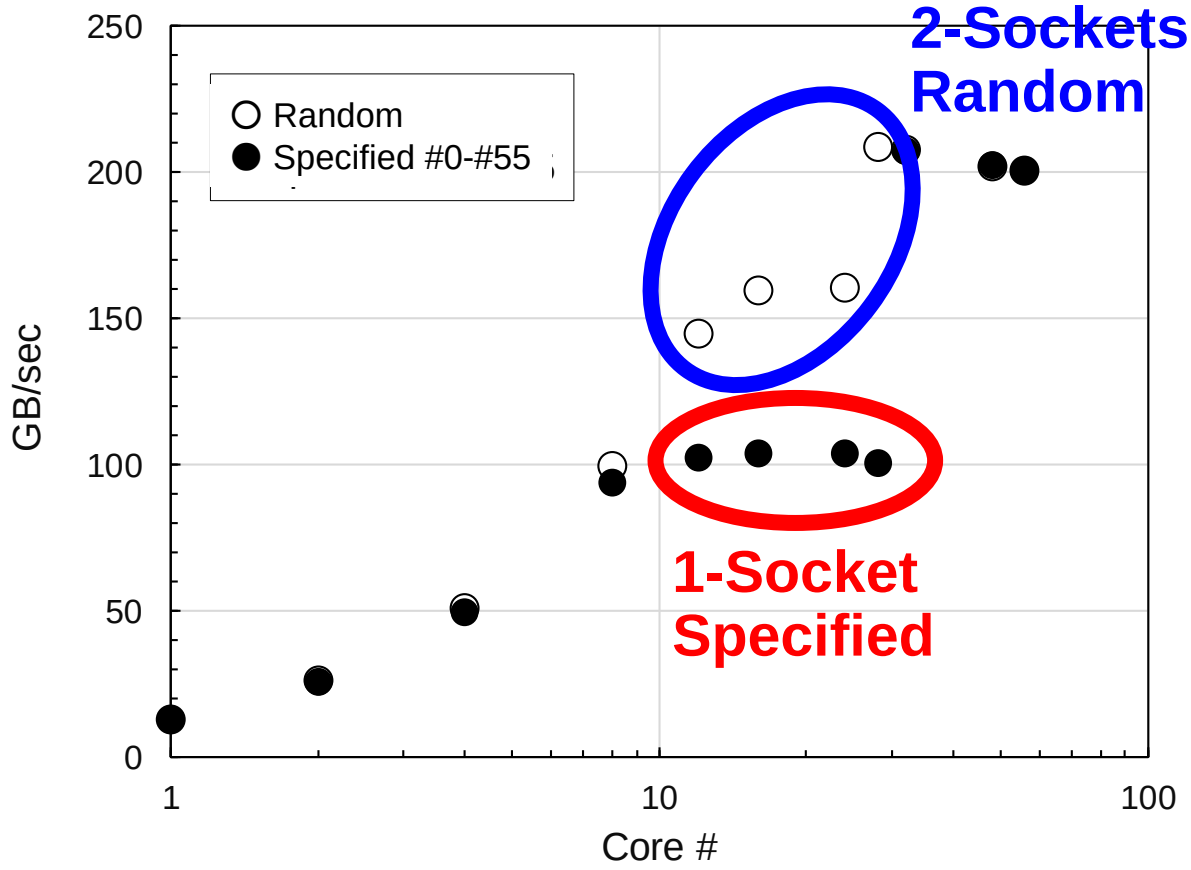
Triad on a Single Node of OBCX Peak is 281.57 GB/sec.

メモリ・キャッシュは「Random」の方がより効果的に利用されている（12-28コアの場合）

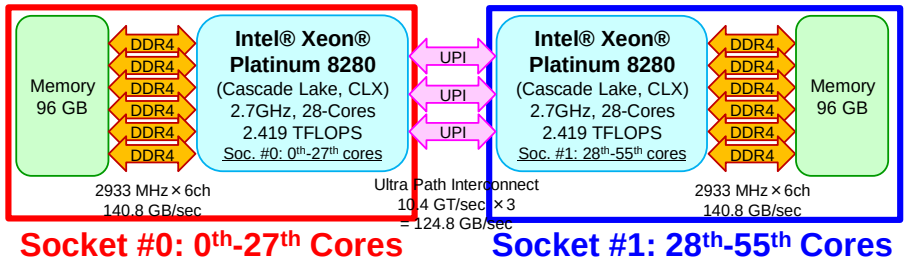


Triad on a Single Node of OBCX Peak is 281.57 GB/sec.

メモリ・キャッシュは「Random」の方がより効果的に利用されている（12-28コアの場合）



Triad on a Single Node of OBCX



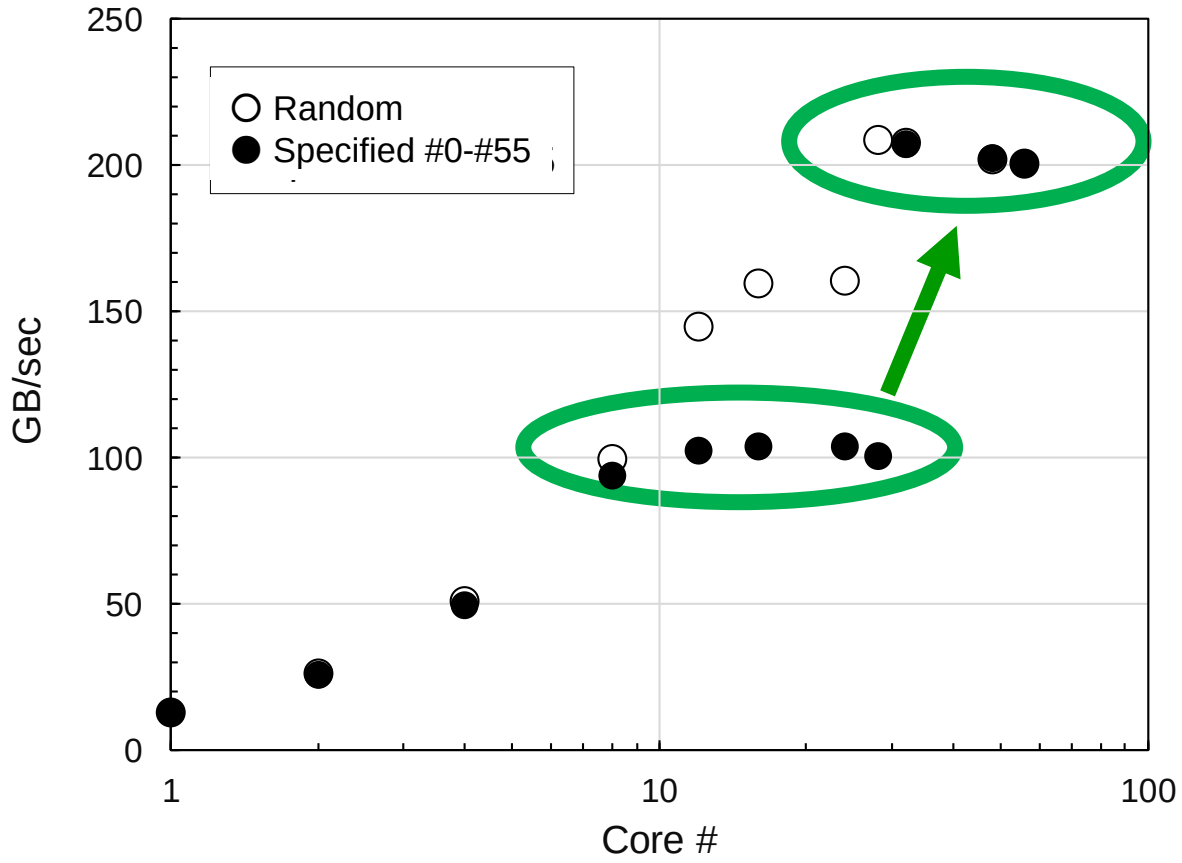
Socket #	Core #	Random	Specified
1	1	1.000	1.000
	2	1.998	1.963
	4	3.912	3.809
	6	5.792	5.682
	8	7.615	7.177
	12	11.089	7.844
	16	12.214	7.968
	24	12.278	7.945
	28	15.962	7.705
2	32	15.901	15.862
	48	15.452	15.497
	56	15.370	15.358

1コアに対する性能向上（Speed-Up）

- ソケット当たりメモリチャンネル数=6
  - 各ソケットに6枚のメモリを装着できる
  - 1ソケット6コア（6プロセス）まではほぼ比例
- 12-28 cores
  - **Random**：2ソケット利用
  - **Specified/Compact**：1ソケット

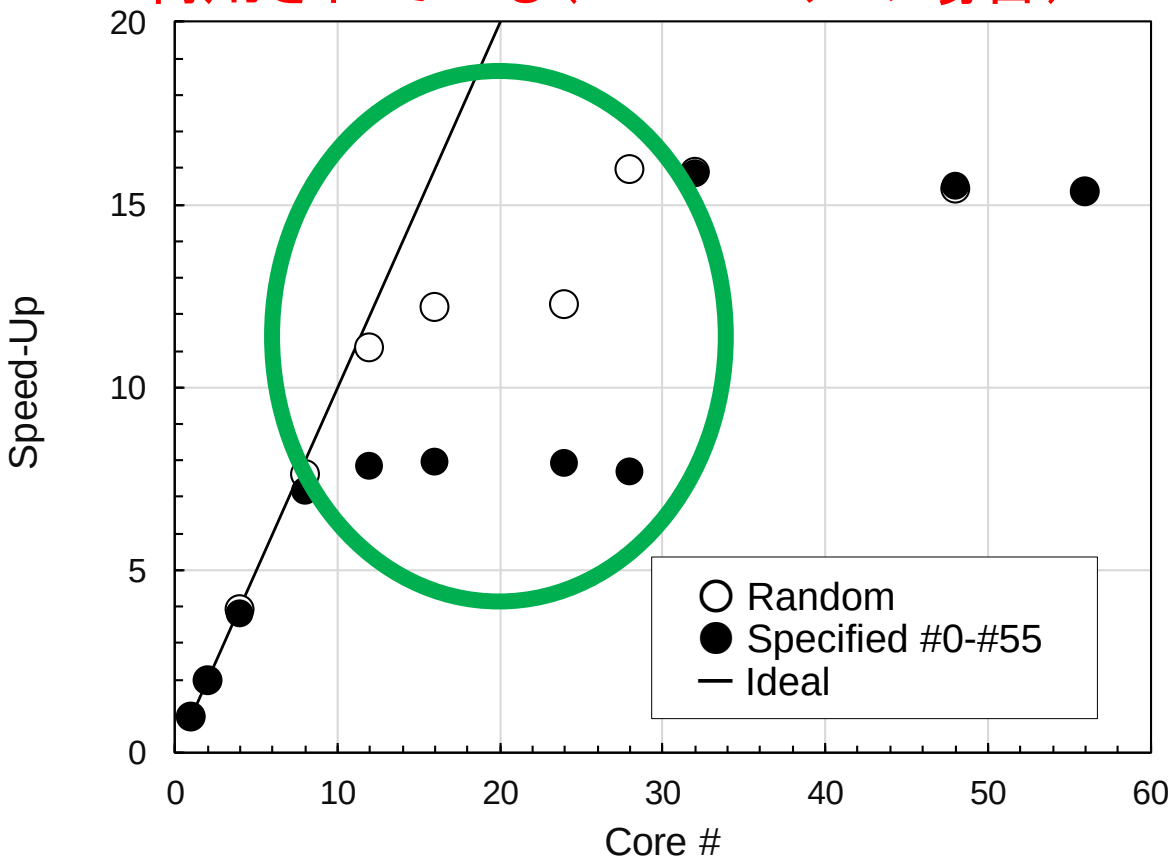
Triad on a Single Node of OBCX Peak is 281.57 GB/sec.

●：メモリバンド幅は8-28コアでほぼ一定（飽和）
32-56コアでは倍になる



Triad on a Single Node of OBCX Peak is 281.57 GB/sec.

メモリ・キャッシュは「Random」の方がより効果的に利用されている（12-28コアの場合）



Exercises

- Running the code
- Try various number of processes (1-56)
- OpenMP-version and Single PE version are available
  - Fortran, C
  - Web-site of STREAM
  - http://www.cs.virginia.edu/stream/

- OpenMP
- Login to OBCX
- Parallel Version of the Code by OpenMP
- STREAM
- **Data Dependency**

ICCG法の並列化

- 内積：OK
- DAXPY：OK
- 行列ベクトル積：OK
- 前処理

前処理はどうするか?
対角スケーリングなら簡単:でも遅い

```
  for (i=0; i<N; i++){
    W[Z][i]= W[R][i]*W[DD][i];
  }
```

```
#omp pragma parallel for private(i)
  for (i=0; i<N; i++){
    W[Z][i]= W[R][i]*W[DD][i];
  }
```

```
#omp pragma parallel for private(i,ip)
  for (ip=0; ip<PEsmpTOT; ip++){
  for (i=INDEX[ip]; i<INDEX[ip+1]; i++)
    W[Z][i]= W[R][i]*W[DD][i];
  }
  }
```

```
64*64*64
METHOD= 1
1     6.543963E+00
101   1.748392E-05
146   9.731945E-09

real    0m14.662s
```

```
METHOD= 3
1     6.299987E+00
101   1.298539E+00
201   2.725948E-02
301   3.664216E-05
401   2.146428E-08
413   9.621688E-09

real    0m19.660s
```

前処理はどうするか?

不完全修正コレスキー分解

```
for(i=0; i<N; i++) {
  VAL = D[i];
  for(j=indexL[i]; j<indexL[i+1]; j++) {
    VAL -= AL[j]*AL[j]*W[DD][itemL[j]-1];
  }
  W[DD][i] = 1.0 / VAL;
}
```

前進代入

```
for(i=0; i<N; i++) {
  WVAL = W[Z][i];
  for(j=indexL[i]; j<indexL[i+1]; j++) {
    WVAL -= AL[j] * W[Z][itemL[j]-1];
   }
   W[Z][i] = WVAL * W[DD][i];
}
```

データ依存性:メモリの読み込みと書き出しが同時に発生し，並列化困難

不完全修正コレスキー分解

```
for(i=0; i<N; i++) {
  VAL = D[i];
  for(j=indexL[i]; j<indexL[i+1]; j++) {
    VAL -= AL[j]*AL[j]*W[DD][itemL[j]-1];
  }
  W[DD][i]= 1.0 / VAL;
}
```

前進代入

```
for(i=0; i<N; i++) {
  WVAL = W[Z][i];
  for(j=indexL[i]; j<indexL[i+1]; j++) {
    WVAL -= AL[j] * W[Z][itemL[j]-1];
   }
   W[Z][i]= WVAL * W[DD][i];
}
```

前進代入
4スレッドによる並列化を試みる

“icel” starts at 0

12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

“itemL” starts at 1

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

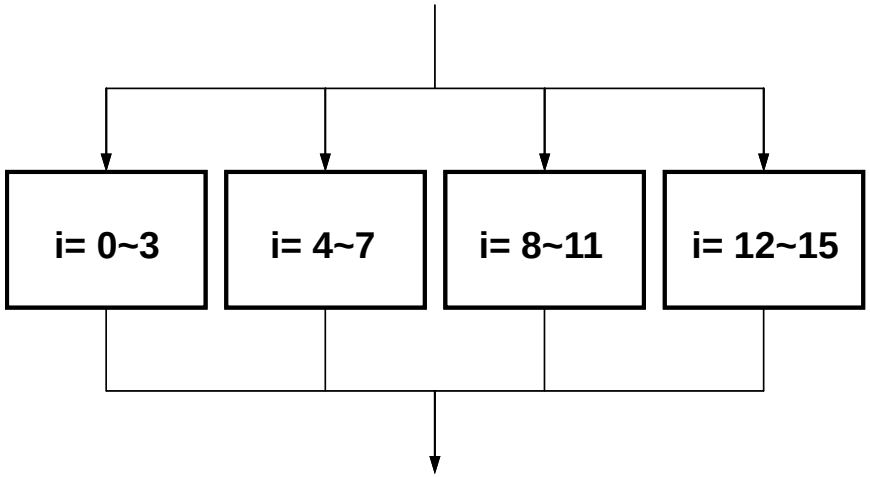
```
for(i=0; i<N; i++) {
  VAL = D[i];
  for(j=indexL[i]; j<indexL[i+1]; j++) {
    VAL -= AL[j]*AL[j]*W[DD][itemL[j]-1];
  }
  W[DD][i]= 1.0 / VAL;
}
```

前進代入
4スレッドによる並列化を試みる

12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

```
#pragma omp parallel private (ip,i,k,VAL)
for(ip=1; ip<4; ip++) {
for(i=INDEX[ip]; i<INDEX[ip+1];i++){
  VAL = D[i];
  for(j=indexL[i]; j<indexL[i+1]; j++) {
    VAL -= AL[j]*AL[j]*W[DD][itemL[j]-1];
  }
  W[DD][i]= 1.0 / VAL;
}
}
```

```
INDEX[0]= 0
INDEX[1]= 4
INDEX[2]= 8
INDEX[3]=12
INDEX[4]=16
```



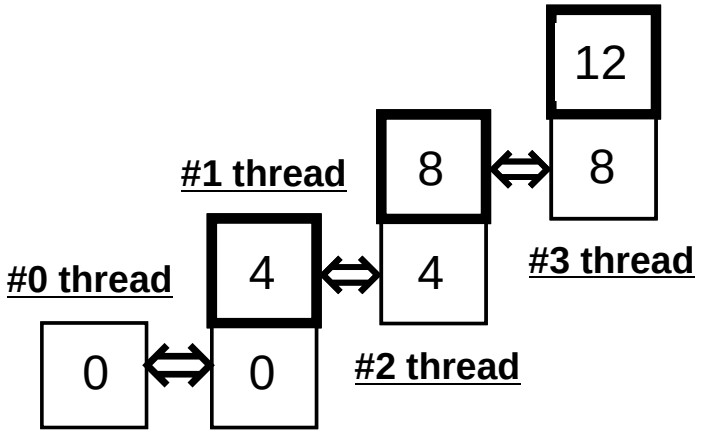
このような4スレッドが同時に
実施される・・・

データ依存性：メモリへの書き出し，読み込みが同時に発生

12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

```
#pragma omp parallel private (ip,i,k,VAL)
for(ip=1; ip<4; ip++) {
for(i=INDEX[ip]; i<INDEX[ip+1];i++){
  VAL = D[i];
  for(j=indexL[i]; j<indexL[i+1]; j++) {
    VAL -= AL[j]*AL[j]*W[DD][itemL[j]-1];
  }
  W[DD][i]= 1.0 / VAL;
}
}
```

```
INDEX[0]= 0
INDEX[1]= 4
INDEX[2]= 8
INDEX[3]=12
INDEX[4]=16
```



⇔の部分に
データ依存性発生

ICCG法の並列化

- 内積：OK
- DAXPY：OK
- 行列ベクトル積：OK
- 前処理：なんとかしなければならない
  - 単純にOpenMPなどの指示行（directive）を挿入しただけでは「並列化」できない。