2019年12月04日（水）10:00-17:00
名古屋大学　2階演習室（201号室）

名古屋大学 情報基盤センター　大島聡史　（問い合わせ先 ohshima@cc.nagoya-u.ac.jp）

第17回
FX100システム利用型 OpenMP講習会（初級）

プログラムと時間の目安

- 9:30 – 10:00　受付

- 10:00 – 12:00　イントロダクション、端末設定など
  - 名古屋大学情報基盤センターの計算機および利用形態
  - FX100システムへのログイン
  - FX100システムへのジョブの投入方法、実行確認

- 13:30 – 17:00　並列プログラミングの基本とOpenMPの学習
  - OpenMPの基礎：仕様と使い方
  - 並列計算の考え方とOpenMPプログラムの最適化
  - 並列化演習

- 17:00 – 17:30　自由演習、スパコン利用相談会

講師について

- 名前：大島　聡史（おおしま　さとし）
  - 情報基盤研究センター　准教授
  - 出身：栃木県塩谷郡（那須と日光の間）
  - 主な経歴
    - 出身大学　電気通信大学
    - →　東京大学　情報基盤センター　助教（2009.09-2017.03）
    - →　九州大学　情報基盤研究開発センター　助教（2017.04-2019.06）
    - →　名古屋大学　情報基盤センター　准教授（2019.07-）

- スパコンの運用・調達に関わりながら最新の計算機環境の活用について研究
  - 「GPUコンピューティング（およびアプリケーションのGPU化）」
  - 「並列数値計算（行列計算、疎行列ソルバー、ライブラリ）」
  - 「プログラミング環境（言語やライブラリ）」

OpenMPを学ぶ前に：並列化とは？どのような処理が並列化できる？

- 並列計算とは何か？並列プログラミングとは何か？
  - 並列計算：何らかの計算処理を同時並行的に行うこと
    - 並列：同じ処理を同時に行う、ある処理を幾つかのサブ処理に分けて同時並行的に行う
    - 平行：何らかの処理を同時並行的に行う（処理同士の関連性はあまり気にしない）
  - 並列プログラミング（並列化プログラミング）：並列化・並列計算を行うためのプログラミング

- なぜ並列計算を行うのか？
  - 短時間で終わらせたい計算があるから、計算機（CPUやパソコンなど、計算を行うハードウェア）の持つ能力をフル活用したいから
  - 現代の計算機は並列計算により高い性能を達成するシステム、並列計算を行わなければ高い性能を得られない
    - PC用CPUもスマートフォン向けCPUもマルチコアCPUが主流
      - マルチコアCPU：コア（＝計算をするユニットのかたまり）を複数搭載したCPU
    - GPUも「超」並列計算志向のプロセッサ

➢ HWの性能を十分に引き出すには並列計算が必須

最近のマルチコアCPUの性能の例

CPU名	コア数（スレッド数）	演算性能	備考
SPARC64VIIIfx	8	128 GFLOPS	「京」コンピュータのCPU
SPARC64XIfx	32+2	1.1 TFLOPS	名大FX100のCPU
A64FX	48	2.7 TFLOPS超	ポスト「京」（富岳）のCPU
Intel Xeon Gold 6154	18 (36)	1.7 TFLOPS	九大ITO-AのCPU
IBM POWER9	22 (88)	540 GFLOPS	Summit（2018年11月時点の世界最速スパコン）のCPU
Intel Core i9 9900K	8 (16)	921 GFLOPS	最近のPC向けハイエンドCPU
Qualcomm Snapdragon 845	4 + 4	※複数種類のコア混載のため計算方法がよくわからないが、i9 9900Kよりは確実に低い	最近のスマホ向けハイエンドCPU

- 1FLOPS=1秒間に（倍精度）浮動小数点演算（加算・乗算）を1回行える性能
- 1K=1000, 1M=1000K, 1G=1000M, 1T=1000G, 1P=1000T, 1Exa=1000P
- 複数のコアを搭載するのが当たり前、コア数は徐々に増加してきている
- 実際のプログラムの性能はコア数や（理論）演算性能だけでは決まらない点には注意が必要

スーパーコンピュータ（スパコン）とは？

- そもそもスーパーコンピュータとは何か？
  - 一般的な計算機システムよりも大幅に高い性能をもつ計算機システム、ただし明確な定義はない
    - ある程度の基準になりそうなものはある：TOP500リストや外為法の規制対象など
  - 必要条件ではないが、現実的に多数の計算機を連結したシステム

- OpenMPはスパコンのノード内並列計算にも活用されており、個人用のPCでも世界最大規模のスパコンでも大変重要なツール
  - ノード内並列化はOpenMP、ノード間並列化はMPI、などの使い分けがされる

OpenMPの基礎：仕様と使い方

参考資料など

- 仕様
  - OpenMPのWebサイト https://www.openmp.org/
    - 各バージョンごとの仕様や、イベント、書籍の情報などが掲載されている（基本的に英語）
  - 1.0がリリースされたのが1997年、現在の最新仕様は5.0
  - 20年以上使われているため日英問わず多くの解説サイトが存在
    - 仕様は変更より追加が多く基本的な部分は一緒、古いページの情報でも使いものになる

（日本語で書かれた）参考書の例

- 「計算科学のためのHPC技術1 」
  - 下司雅章 (編集), 片桐孝洋 , 中田真秀, 渡辺宙志, 山本有作, 吉井範行, Jaewoon Jung, 杉田 有治, 石村和也, 大石進一, 関根晃太, 森倉悠介, 黒田久泰，著
  - 大阪大学出版会、ISBN-10: 4872595866、ISBN-13: 978-4872595864、発売日：2017年4月3日
- 「並列プログラミング入門：サンプルプログラムで学ぶOpenMPとOpenACC」
  - 片桐 孝洋 著
  - 東大出版会、ISBN-10: 4130624563、ISBN-13: 978-4130624565、発売日：2015年5月25日

OpenMPとは？

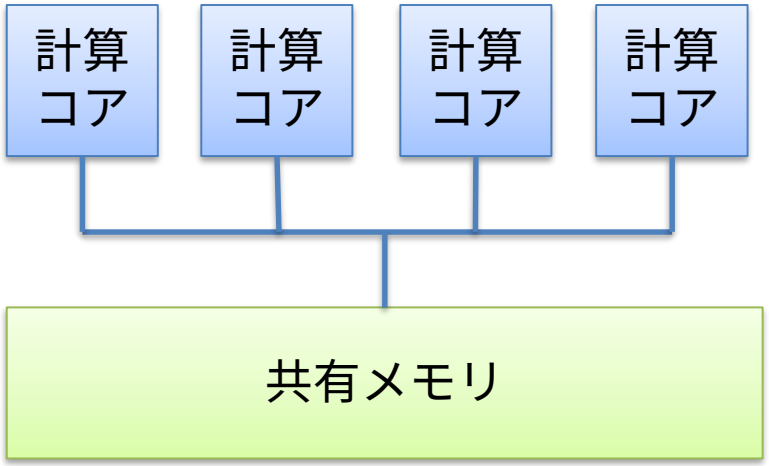
- 共有メモリ型並列計算機にて（主にループ計算を）簡単に並列実行することができるもの
- OpenMP自体はプログラミング言語ではなく、C/C++やFortranにコードを追加して使う
  - 本講習会ではCとFortran90（自由形式）を想定して例示する
- スレッド並列化のための道具
  - スレッドとは？プロセスとは？を説明するのは今回の本題ではないためスキップ
  - 1ノード上で複数の処理を並列に実行するとき、その1つ1つの処理の流れをスレッドと呼んでいる、くらいの認識で良い（単数の処理は、複数の処理の数が1である特殊な例）
    - 例：1スレッドで実行する、2スレッドで実行する、スレッドが同時に計算を行う、スレッド間で同期を取る
- 自動で並列化してくれるわけではない、並列実行できるかどうかの責任は利用者にある
  - コンパイラは指定されたとおりにプログラムを変型するだけ（苦言を呈してくれることもある）

- 共有メモリ型並列計算機？　主にループ計算を並列化？　簡単に並列実行？
  - もう少し細かく説明していく

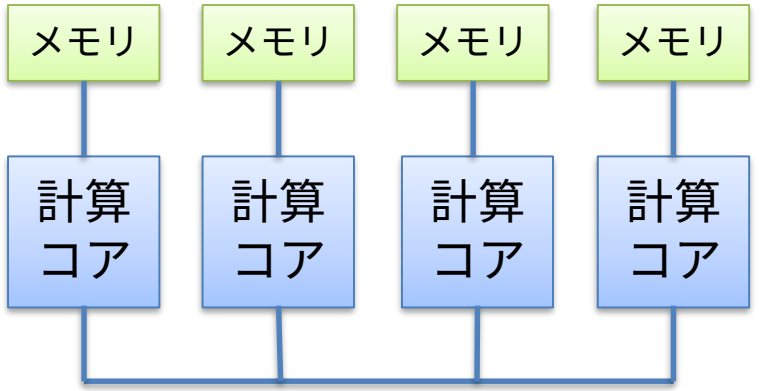
並列計算機の分類

- 並列計算機は共有メモリ型と分散メモリ型に大別される

- 共有メモリ型：メモリを共有している
  - 同じメモリ（データ）に各プロセッサが自由に直接アクセスできる、メモリアクセスへの競合（衝突）が起こる
  - 分散メモリ型として扱うこともできる



- 分散メモリ型：メモリを共有していない
  - 分散メモリモデル：各プロセッサが個別のメモリ（データ）を持つ、互いに直接アクセスできない、他のコアの持つメモリにアクセスするには通信が必要
  - 共有メモリ型として扱うためには、分散メモリを共有メモリとして扱う特別な仕組みが必要



- 大規模環境では混在した（階層的な）構成となる
- 物理的な構成とプログラミング手法は1対1の対応関係ではない
- ノードの概念とも1対1ではない（基本的にはノード内が共有、ノード間が分散ではある）

並列計算を行う方法（言語など）の例

- コンパイラによる自動並列化：SIMD並列化など一部の処理で有用
- pthread：スレッド並列処理のための関数群
  - pthread_createなどのスレッド操作関数を使う
  - 現在ではアプリケーションコードで直接利用することはあまりない
- OpenMP：スレッド並列処理、主にループ並列化向けの指示文規格
  - #pragma omp parallel for 、!$omp parallel do
  - 最近の規格ではタスク処理やGPU対応などを拡充
- OpenACC：GPU向けのOpenMPのようなもの
- CUDA：NVIDIA社のGPU向け、GPUを普及させた最大要因の一つ、NVIDIA GPUの能力をフル活用できる
- OpenCL：「汎用版CUDA」のようなもの、FPGAなどでも利用可能
- MPI：プロセス間の通信規格、特に複数ノード利用時に必須
  - MPI_Send, MPI_Recv, MPI_Gatherなどの通信関数を使う
- 必要に応じてこれらを組み合わせて用いる（OpenMP+MPIなど）

- ノード内CPU内スレッド並列化
- 共有メモリモデル

- GPU内並列化
- デバイス内では共有メモリモデル、外部とのやりとりは分散メモリモデル

- ノード間並列化
- 分散メモリモデル

並列計算の基本的なイメージ：ループ並列化（C版）

- 元となる逐次計算
  - 単純な繰り返しループ計算

```
for(i=0; i<N; i++){
  A[i] = B[i] + C[i];
  D[i] = E[i] + F[i];
}
```

- ループ内の処理を分割し、同時に計算

PE1
```
for(i=0; i<N; i++){
  A[i] = B[i] + C[i];
}
```

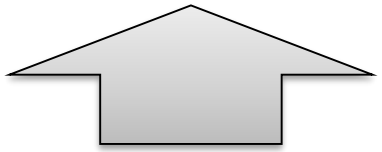
PE2
```
for(i=0; i<N; i++){
  D[i] = E[i] + F[i];
}
```

PE: Processor Element、CPUやGPUなど計算をする「何か」

- ループそのものを分割し、同時に計算

PE1
```
for(i=0; i<N/2; i++){
  A[i] = B[i] + C[i];
  D[i] = E[i] + F[i];
}
```

PE2
```
for(i=N/2; i<N; i++){
  A[i] = B[i] + C[i];
  D[i] = E[i] + F[i];
}
```



- OpenMPはこちらのイメージ

並列計算の基本的なイメージ：ループ並列化（Fortran版）

- 元となる逐次計算
  - 単純な繰り返しループ計算

```
do i=1, N
  A(i) = B(i) + C(i)
  D(i) = E(i) + F(i)
end do
```

- ループ内の処理を分割し、同時に計算

```
PE1   do i=1, N
        A(i) = B(i) + C(i)
      end do
```

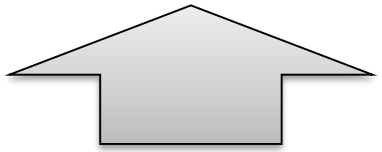
```
PE2   do i=1, N
        D(i) = E(i) + F(i)
      end do
```

PE: Processor Element、CPUやGPUなど計算をする「何か」

- ループそのものを分割し、同時に計算

```
PE1   do i=1, N/2
        A(i) = B(i) + C(i)
        D(i) = E(i) + F(i)
      end do
```

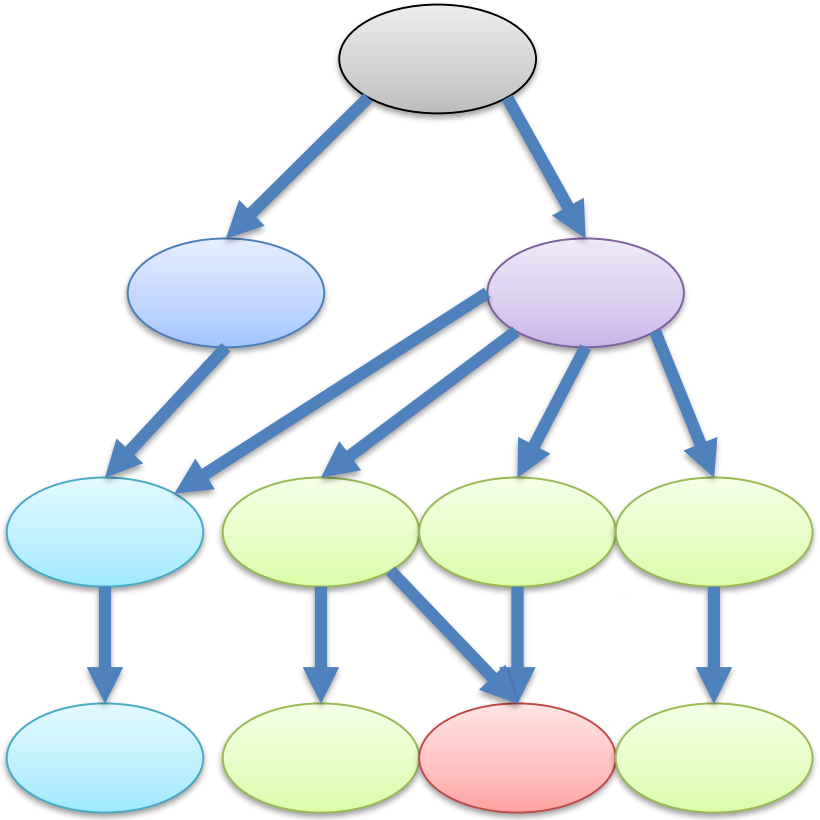
```
PE2   do i=N/2+1, N
        A(i) = B(i) + C(i)
        D(i) = E(i) + F(i)
      end do
```



- OpenMPはこちらのイメージ

並列計算の基本的なイメージ：タスク並列化

- ループによる並列化よりも大きい粒度の並列計算に適する
- 大規模なプログラム・複雑なプログラムの並列化に有用
- OpenMPにおいても近年サポートが活発
  - task構文
- 今回の講習会では扱わない



簡単な並列化の例

- 最も単純なパターンでは、並列化したい対象に一行（Fortranでは一組）の指示文を加えるだけで並列化が可能

hello0.c

```c
#include <stdio.h>

int main()
{
  printf("hello, world￥n");
#pragma omp parallel
  {
    printf("hello, parallel world￥n");
  }
  printf("bye￥n");

  return 0;
}
```

hello0.f90

```
program hello
  implicit none

  print *, "hello, world"

!$omp parallel
  print *, "hello, parallel world"
!$omp end parallel

  print *, "bye"

end program hello
```

OpenMP API

- 指示文以外にも色々な機能を提供する関数が用意されている
  - 主な（よく使われそうな）関数

関数名	機能
omp_get_thread_num()	自分のスレッド番号を取得
omp_get_max_threads()	並列リージョンで起動できるスレッド数を取得
omp_get_num_threads()	並列リージョン内で、現在実行中のスレッド数を取得
omp_get_wtime()	経過時間（秒）の取得
omp_set_num_threads(整数)	最大スレッド数の指定

  - これらを使う場合はライブラリヘッダファイルのインクルードが必要
    - C/C++：#include <omp.h>
    - Fortran：use omp_libまたはinclude "omp_lib.h"

指示文

- OpenMPは指示文によって並列化の制御を行う
- 指示文＝コンパイラに情報を与えるための特別なコメント
  - C/C++言語：　#pragma　で始まる行
  - Fortran：　!$　で始まる行
- 指示文は、OpenMP以外にもコンパイラ独自の様々な最適化機能の制御等に用いられる
  - キャッシュの制御
  - SIMD命令の制御
  - その他、共通言語仕様に含まれないCPU独自の機能の制御など
- 指示文は、対応していないコンパイラにとってはコメント行に見える
- →無視しても実行できる、対応しているコンパイラ向けとそうでないコンパイラ向けで別のソースコードにする必要がない
  - （性能を考えると、実行する計算機環境に合わせて別のコードを用意する必要は生じる）
  - APIを用いたコードが無視できない？　→条件付きコンパイル

条件付きコンパイル

- OpenMPに対応しているコンパイラと対応していないコンパイラのいずれでも問題なくコンパイルできるようなコードを書くにはどうすれば良いか？

C: _OPENMP定数の有無で分けるのが良い

```
#ifdef _OPENMP
  d1 = omp_get_wtime();
#endif

#pragma omp parallel
{
  ......
}

#ifdef _OPENMP
  d2 = omp_get_wtime();
  printf(" %fsec￥n", d2 - d1);
#endif
```

Fortran：!$ で始まる行はOpenMP有効時のみ有効となるため、OpenMPを使うときだけ動いて欲しい部分に挿入すると良い

```
!$ d1 = omp_get_wtime()

!$omp parallel
  ......
  ......
!omp end parallel

!$ d2 = omp_get_wtime()
!$ print *, d2 - d1, "sec"
```

指示文の対象範囲

- C/C++言語の場合：直後の構造化ブロック

```
#pragma omp ……
{
  ……
}

#pragma omp parallel for
for(i=0; i<N; i++){
  ……
}
```

※括弧で括らないと、
指示文直後の一行のみ
が対象となる

- Fortranの場合：endで閉じるまで

```
$!omp ……
  ……
  ……
$!omp end ……

$!omp parallel do
  ……
  ……
$!omp end parallel do
```

- 指示文を複数行に継続させることも可能
  - 指定する情報が多い際に有用
  - C/C++とFortranでは少し書き方が違う　（OpenMPの仕様というよりC/C++とFortranの言語仕様の差）

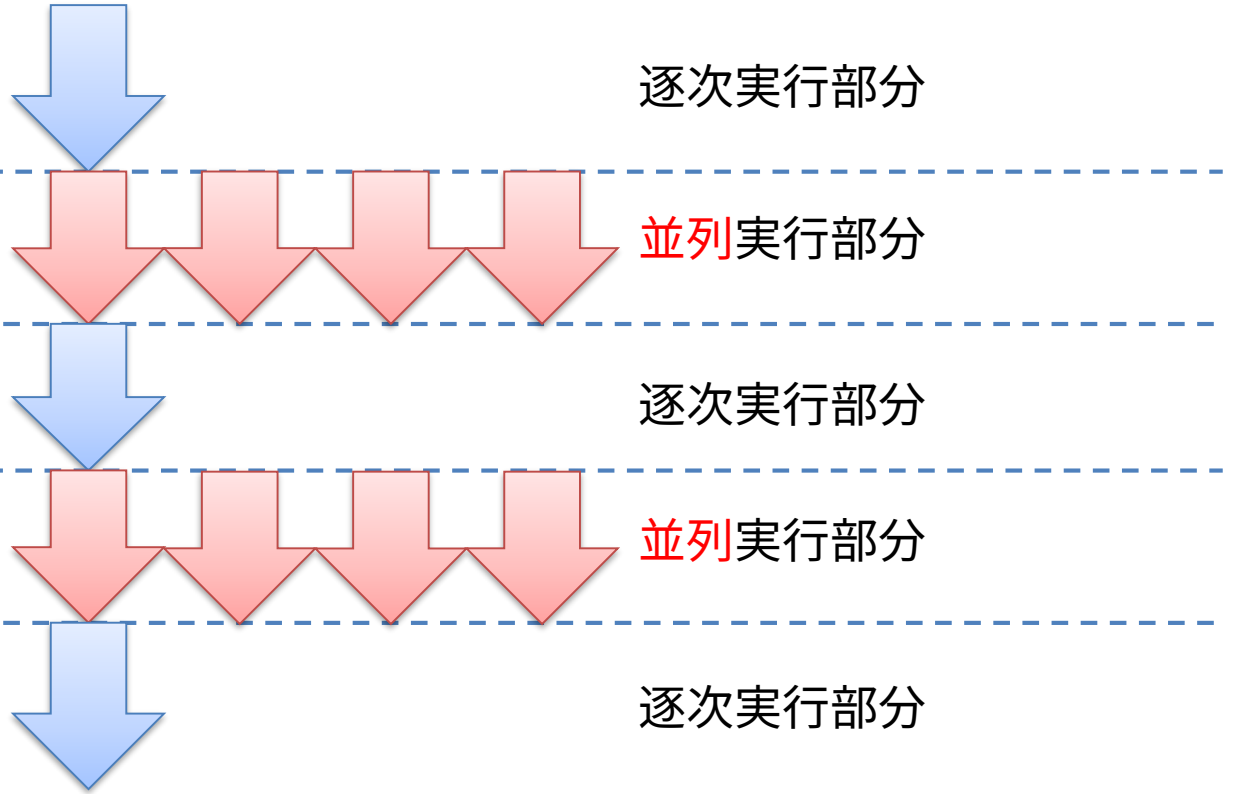
```
#pragma omp hoge ¥
  foo bar
```

```
$!omp hoge
$!omp+ foo bar
```

OpenMPの動作イメージ

- 指示文で囲まれた部分のみが複数スレッド（並列）で実行される
  - それ以外の部分は1スレッドのみ（逐次）で実行される

```
int main()
{
  ......
#pragma omp parallel
  {
    ......
  }
  ......
  ......
#pragma omp parallel
  {
    ......
  }
  ......
  return 0;
}
```



OpenMPプログラムのコンパイル方法と実行方法

- コンパイル方法：コンパイルオプションをつけるだけ
  - 富士通コンパイラ：-Kopenmp
  - インテルコンパイラ：-qopenmp
  - GNUコンパイラ：-fopenmp
  - PGIコンパイラ：-mp

  FX100における富士通社推奨コンパイルオプションを用いたコンパイルの例
  - `fccpx -Kfast -g -Ntl_trt -Xa -NRnotrap -Kopenmp source.c`
  - `frtpx -Kfast -g -Ntl_trt -X9 -NRnotrap -Kopenmp source.f90`

- 実行方法：そのまま実行するだけ
  - 並列度は環境変数によって制御可能、デフォルトでは論理スレッド数分だけ起動する
  - 環境変数OMP_NUM_THREADSによって数を指定可能
  - スレッドと計算コアの割り当てを細かく指定することもできる（後述）
    - 性能に大きく影響を与えることがある

実習：OpenMPプログラムのコンパイルと実行を試す

- hello1.cまたはhello1.f90を計算ノード向けにコンパイルし、ジョブとして実行する
  - さらに、OMP_NUM_THREADSを設定して並列度を変えて動作を比較してみる

hello1.c

```
#include <stdio.h>
#include <omp.h>

int main()
{
  printf("hello, world￥n");
#pragma omp parallel
  {
    printf("hello, parallel world %d￥n",
                omp_get_thread_num());
  }
  printf("bye￥n");

  return 0;
}
```

hello1.f90

```
program hello
  use omp_lib
  implicit none

  print *, "hello, world"

!$omp parallel
  print *, "hello, parallel world ", &
           omp_get_thread_num()
!$omp end parallel

  print *, "bye"

end program hello
```

実習：作業手順

- 実習用のファイルは以下の公開ディレクトリに置いてある
  - /center/a49979a/share/20191204.tgz
- ファイルをコピーして展開しておく
  - cp -r /center/a49979a/share/20191204.tgz
  - tar zxvf ./20191204.tgz
  - cd 20191204
- 計算ノード向けにコンパイルする
  - fccpx -Kopenmp -o hello1 hello1.c
  - frtpx -Kopenmp -o hello1 hello1.f90
- ジョブスクリプトを書いてジョブを投入する
  - その際にOMP_NUM_THREADSを変化させてみる

OpenMPプログラム向けのジョブスクリプトの書き方

```
#!/bin/bash
#PJM -L "rscgrp=fx-debug"          ← リソースグループ名：fx-debug
#PJM -L "node=1"                   ← 利用ノード数：1
#PJM -L "elapse=1:00"              ← 利用時間制限：1分
#PJM -S                            ← 統計情報を出力
#PJM -j                            ← 標準出力と標準エラー出力を統合

echo "######## ########"
export OMP_NUM_THREADS=1           ← 最大スレッド数を1にする
./hello1                           ← プログラムを実行

echo "######## ########"
export OMP_NUM_THREADS=2           ← 最大スレッド数を2にする
./hello1                           ← プログラムを実行

echo "######## ########"
export OMP_NUM_THREADS=4           ← 最大スレッド数を4にする
./hello1                           ← プログラムを実行
```

必須ではない

- 赤字部分は名大FX100での実行に固有の情報
- 黒字部分は一般的なbashスクリプト処理

実行結果の例

```
$ cat job_hello1.sh.o112881
######## ########
 hello, world
 hello, parallel world  0
 bye
######## ########
 hello, world
 hello, parallel world  0
 hello, parallel world  1
 bye
######## ########
 hello, world
 hello, parallel world  2
 hello, parallel world  0
 hello, parallel world  3
 hello, parallel world  1
 bye
```

※スレッドIDはC/C++でもFortranでも0から始まる

OMP_NUM_THREADS=1の場合、逐次実行

OMP_NUM_THREADS=2の場合、2並列実行

※各実行ごとの
hello, parallel world *
の出てくる順番は不定

OMP_NUM_THREADS=4の場合、4並列実行

※以下のような情報も出てくることがあるが、気にしなくて良い

```
jwe1050i-w The hardware barrier couldn't be used and continues processing using the software barrier.
taken to (standard) corrective action, execution continuing.
error summary (Fortran)
error number  error level  error count
  jwe1050i         w           1
total error count = 1
```

ループ並列化：for指示文・do指示文

- 対象ループをスレッドに割り当てて並列実行する

loop1.c

```
#pragma omp parallel
{
#pragma omp for
  for(i=0; i<N; i++){
    c[i] = a[i] + b[i];
  }
}
```

loop1.f90

```
!$omp parallel
!$omp do
  do i=1, N
    c(i) = a(i) + b(i)
  enddo
!$omp end do
!$omp end parallel
```

※!$omp end do は省略可能

- parallel と for、parallelとdoをまとめて指示することも可能

loop2.c

```
#pragma omp parallel for
  for(i=0; i<N; i++){
    c[i] = a[i] + b[i];
  }
```

loop2.f90

```
!$omp parallel do
  do i=1, N
    c(i) = a(i) + b(i)
  enddo
!$omp end parallel do
```

※!$omp end parallel do は省略可能

OpenMPにおける変数や配列の扱い

- 特に指定しない場合、全ての変数・配列が全スレッドで共有される

private1.c

```
#include <stdio.h>
#include <omp.h>

int main()
{
  int i;
  printf("hello, world￥n");
#pragma omp parallel
  {
    i = omp_get_thread_num();
    printf("hello, parallel world %d￥n", i);
  }
  printf("bye￥n");

  return 0;
```

一次変数に格納してから出力すると、スレッド間で上書きされてしまい正しい結果が得られないことがある
（タイミング次第）

private1.f90

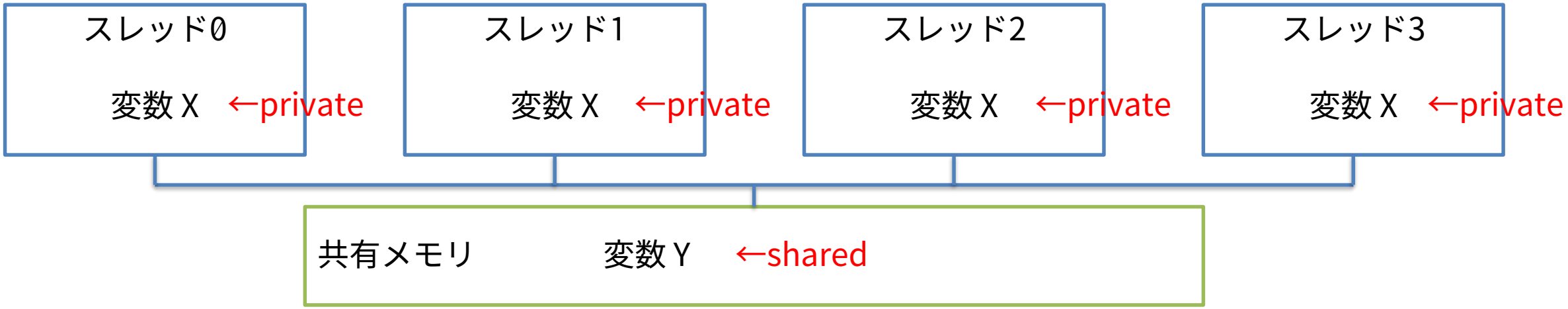
```
program private
  use omp_lib
  implicit none
  integer :: i

  print *, "hello, world"

!$omp parallel
  i = omp_get_thread_num()
  print *, "hello, parallel world ", i
!$omp end parallel

  print *, "bye"

end program private
```

富士通Fortranコンパイラの場合は警告が出る

```
$ frtpx -Kopenmp -o hello2 hello2.f90
Fortran diagnostic messages: program name(hello)
  jwd2890i-i  "hello2.f90", line 9: Definitions of shared variable 'i' will possibly conflict among threads.
```

変数・配列の属性

- shared：全スレッドで共有される
  - いずれかのスレッドが値を書き換えると、他のスレッドからも書き換わった値が見える
- private：各スレッドが独自に持つ、初期値は不定
  - いずれかのスレッドが値を書き換えても、他のスレッドから書き換わった値が見えない
- firstprivate：各スレッドが独自に持つ、初期値は並列実行部の前を引き継ぐ
  - privateの初期値引き継ぎ版
- sharedとprivateのイメージ



属性の指定方法

- 指示文に指示節を追加する

```
#pragma omp parallel private(a, b) shared(c, d)
{
  ……
```

```
!$omp parallel private(a, b) shared(c, d)
……
```

- デフォルトの属性を変更することも可能

```
#pragma omp parallel default(shared)

#pragma omp parallel default(none) private(a)
```

```
!$omp parallel default(shared)

!$omp parallel default(none) private(a)
```

  - noneを指定した場合は、並列化範囲内で使われる全ての変数の属性を明示せねばならない
    - バグを探す・取り除く際に便利なことがある
    - Fortranのみ、privateやfirstprivateもdefaultに指定が可能

暗黙的にprivate属性になるケース

- C：並列実行部の中で宣言された局所変数
- Fortran：並列化されたループの中の逐次ループの制御変数

※C, Fortranともに配列の初期化は省略

private2.c

```c
#include <stdio.h>
#include <omp.h>

int main(){
  int i;
  int a[10], b[10], c[10];
#pragma omp parallel
  {
    int tid;
    tid = omp_get_thread_num();
    c[tid] = a[tid] + b[tid];
    printf("c[%d] = %d¥n", tid, c[tid]);
  }

  return 0;
}
```

tidがprivate化され、配列Cのうちスレッド番号までが更新される

private2.f90

```
program private
  use omp_lib
  implicit none
  integer :: i, j, a(10), b(10), c(10)

!$omp parallel do
  do i=1, 10
    do j=1, 10
      c(i) = c(j) + (a(j) + b(j))
    end do
  end do
!$omp end parallel do

  print *, c

end program private
```

jループは並列化されたループの中の逐次ループのため、jがprivate化される

ループ開始・終了時の変数（配列）の属性

- parallelではなくforやdoに指示節を加えることで、ループ前後の変数（配列）の属性を指定できる
  - shared：変数（配列）の値は、ループ開始時にループ前から引き継がれ、ループ終了時にループ後へと引き継がれる
  - private：変数（配列）の値は、ループ開始時もループ後も不定
    - 言語仕様として決まっていない、コンパイラによって振る舞いが異なる可能性がある
  - firstprivate：変数（配列）の値は、ループ開始時にループ前から各スレッドに引き継がれ、ループ後は不定となる
  - lastprivate：変数（配列）の値は、ループ開始時は不定だが、ループ終了時には最後のイタレーションの結果がループ後に引き継がれる
    - firstprivateとlastprivateは同時に利用可能：ループ開始時にループ前から各スレッドに引き継がれ、ループ終了後には最後のイタレーションの結果がループ後に引き継がれる

関数の呼び出しとスレッドセーフ

- OpenMPによって並列化された部分から関数の呼び出しをしても良い
  - omp_get_thread_numが使えていた時点で予想できることではある

```c
#include <stdio.h>
#include <omp.h>

int main(){
  int ret;
#pragma omp parallel
  {
    ret = func(omp_get_thread_num());
  }
```

```fortran
program func
  use omp_lib
  implicit none
  integer :: ret

!$omp parallel do
  ret = func(omp_get_thread_num())
!$omp end parallel do
```

func：スレッドIDを渡すと、それに対応した何らかの処理をしてくれる関数、と仮定

- 呼び出された先の処理がスレッド並列実行しても問題ないように書かれていないと、予期せぬ問題を引き起こすことがある
  - グローバル変数を使っていたり、通信やファイルI/Oなど外部のリソースを使っている場合は特に注意が必要
  - スレッド並列実行しても問題ないものを「スレッドセーフ」と呼ぶ

実行時間の測定

- omp_get_time関数を使えば簡単に実行時間を測定できる
  - スレッド並列化されている部分かどうかに関係なく利用可能

```
d1 = omp_get_wtime();
// 測定したい対象
d2 = omp_get_wtime();
d = d2 - d1; // 経過時間（秒）が得られる
```

```
d1 = omp_get_wtime()
! 測定したい対象
d2 = omp_get_wtime()
d = d2 - d1 ! 経過時間（秒）が得られる
```

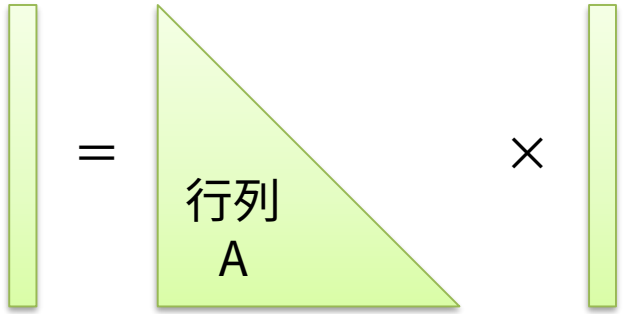
- OpenMPプログラムの実行時間を測定する際には、測定範囲に気を付ける必要がある
  - スレッド並列化範囲全体の実行時間を測定したいのか？
  - スレッド並列化における各スレッドの実行時間を測定したいのか？
    - 測定結果を配列やprivateな変数に格納しないと上書きされてしまう点にも注意
  - プログラム最適化（高速化）をする場合、最終的には全体の実行時間を短くするのが目標だとしても、多くの場合はスレッドごとの実行時間を確認し調査する必要がある
  - 単純なプログラムの場合などはコンパイラによる最適化の効果が強すぎて並列化による差が目立たないこともある　→　問題サイズを大きくする、コンパイル時に -O0 オプションを付ける
  - 測定対象時間が短すぎると正確に測れない（誤差が大きくなる）点にも注意

実習：簡単なベクトル計算・行列計算の並列化と実行時間測定

◆ ベクトル同士・行列同士の和を並列計算してみる（vecadd, matadd）

◆ 行列ベクトル積を並列計算してみる（matvec1）

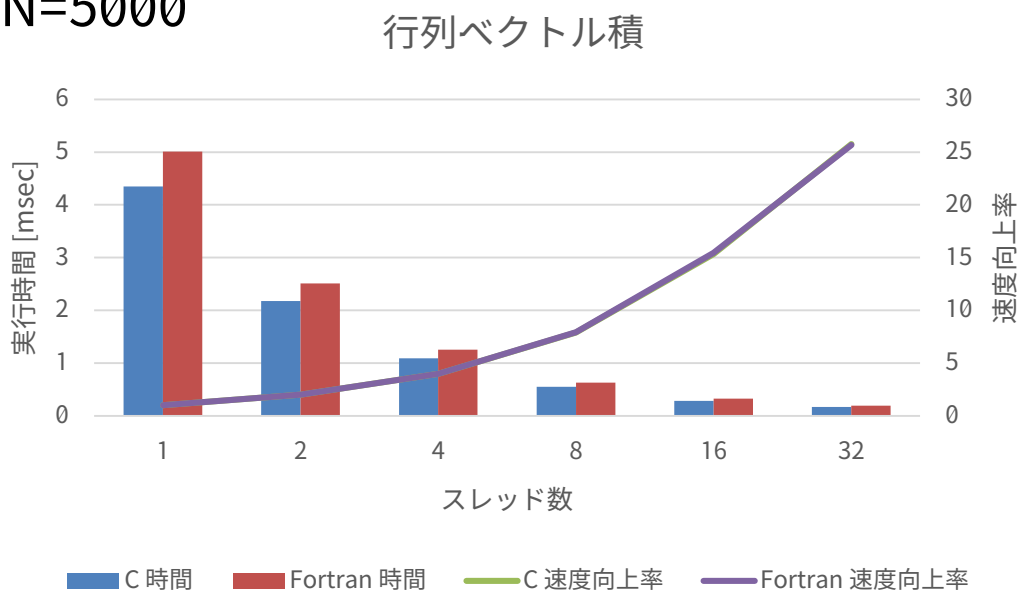
◆ 三角行列ベクトル積を並列計算してみる（matvec2）

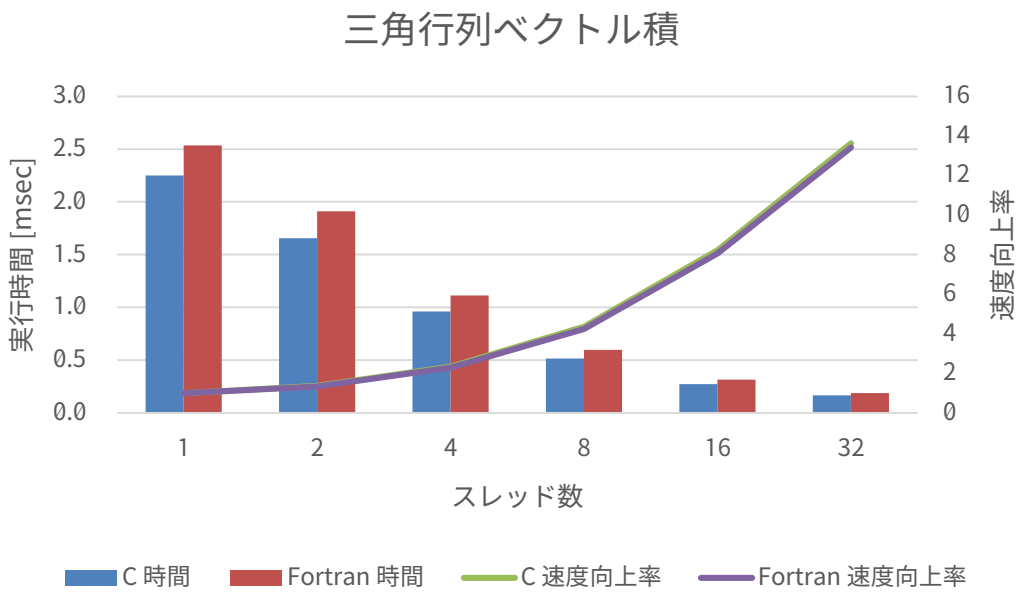


- 公開ディレクトリに逐次計算プログラムが提供されているため、OpenMP指示文を加えて並列化してみよう
  （ベクトル同士の和は要素ごとに、それ以外は（まずは）行列の行ごとに各スレッドに割り当てる）
- 行列サイズを大きくしたり、スレッド数を変えたりしてみよう
  （行列サイズを大きくする場合は、結果の出力は長くなりすぎるため省いた方が良い）

三角行列ベクトル積の並列化

- ベクトル和、行列和、行列ベクトル積のような単純なベクトルや行列の計算は、問題サイズ（ベクトルや行列の長さ）が十分大きければ、利用するスレッド数を増やすほど性能が向上する傾向にある
  - ※メモリ転送性能によって頭打ちになるまで性能が向上する
- 一方、三角行列ベクトル積は性能があまり向上しない
  - 例：N=5000





- 何故だろうか？

※問題サイズやコンパイルオプションにもよる

三角行列ベクトル積の並列化

- 特に指定しない限り、ループ並列化はループ長をスレッド数で均等に分割する

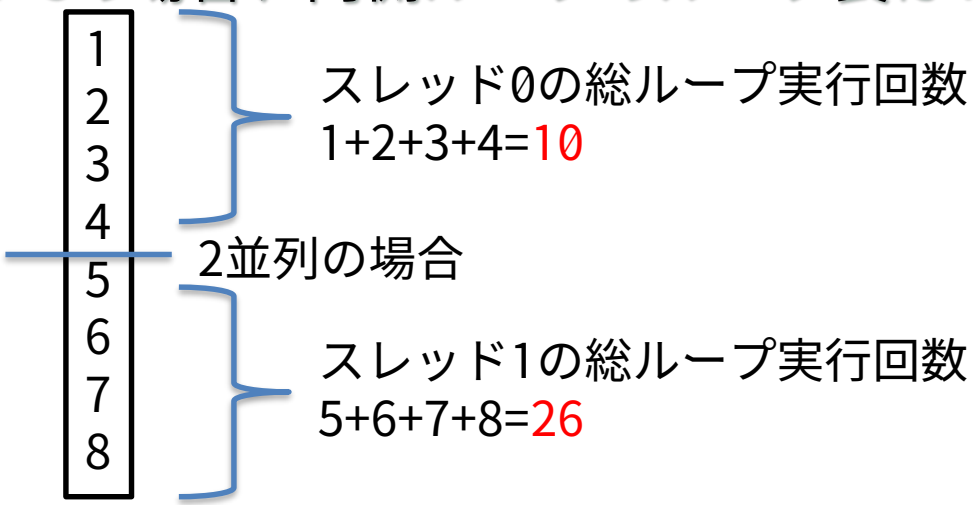
matvec2.cに指示文を追加したもの

```
#pragma omp parallel for private(j)
  for(i=0; i<N; i++){
    for(j=0; j<=i; j++){
      c[i] += a[i][j] * b[j];
    }
  }
```

matvec2.f90に指示文を追加したもの

```
!$omp parallel do
  do i=1, N
    do j=1, i
      c(i) = c(i) + a(j,i) * b(j)
    end do
  end do
```

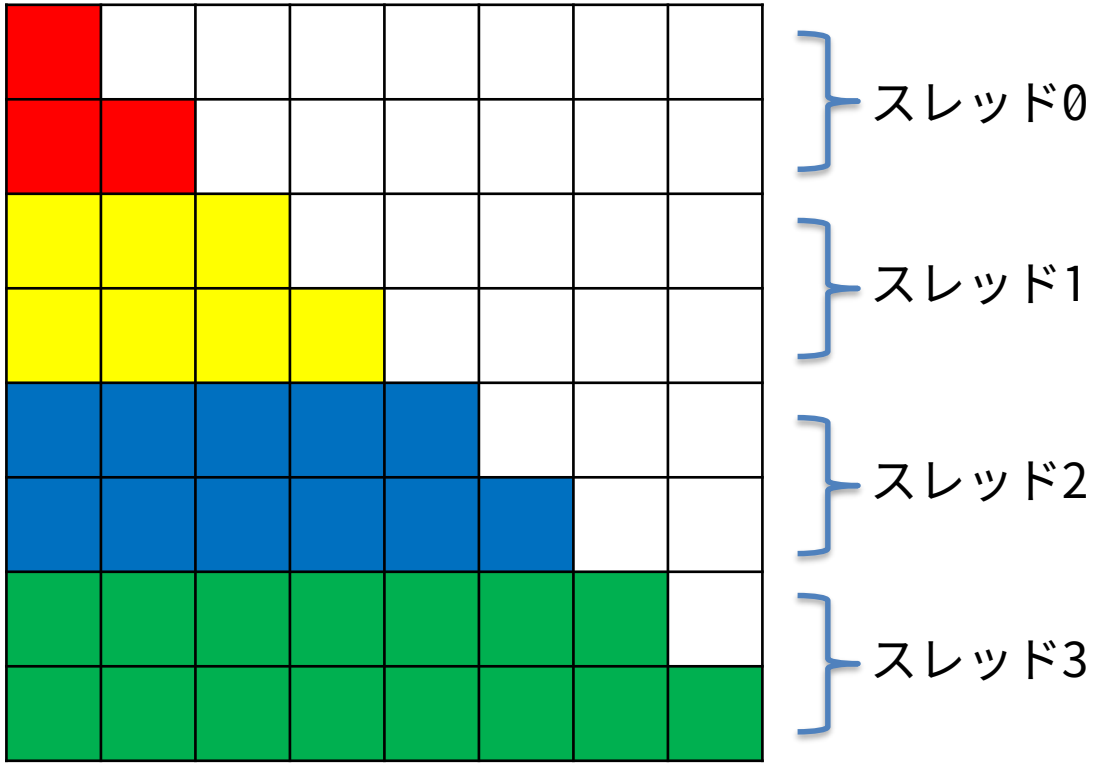
- N=8の場合、内側ループのループ長は1から8まで不均等



スレッドごとの計算量（≒実行時間）が均等ではない
→全体の実行時間は最も遅いスレッドに律速される

図で例示すると……（N=8で4スレッドの場合）

- デフォルト設定：ループ長をスレッド数で分割し、順番に割り当てる
  - for/doにschedule指示節を追加することで割り当て方法を変更することができる
    - この例はschedule(static)と指定した場合と等価

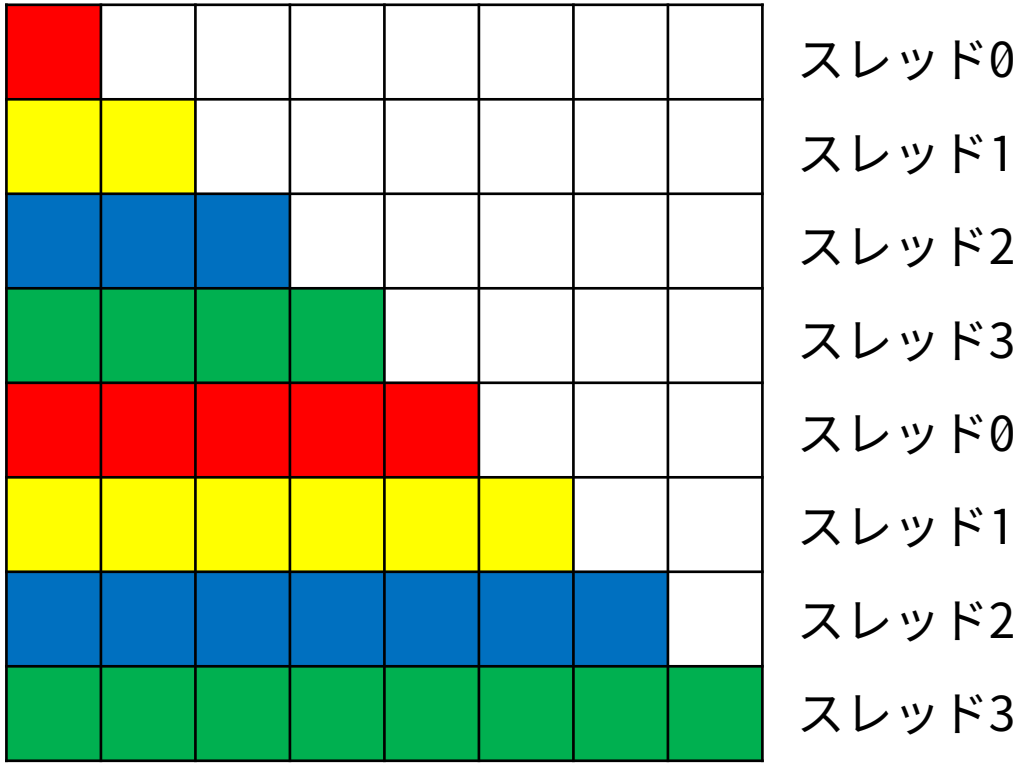



```
#pragma omp parallel for private(j) schedule(static)
  for(i=0; i<N; i++){
    for(j=0; j<=i; j++){
      c[i] += a[i][j] * b[j];
    }
  }
```

```
!$omp parallel do schedule(static)
  do i=1, N
    do j=1, i
      c(i) = c(i) + a(j,i) * b(j)
    end do
  end do
```

割り当て粒度（チャンクサイズ）の調整

- 第二引数で粒度を変更することができる
  - schedule(static, 1)　の例
  - 変数iについて、1刻みで分割してスレッドに割り当てる



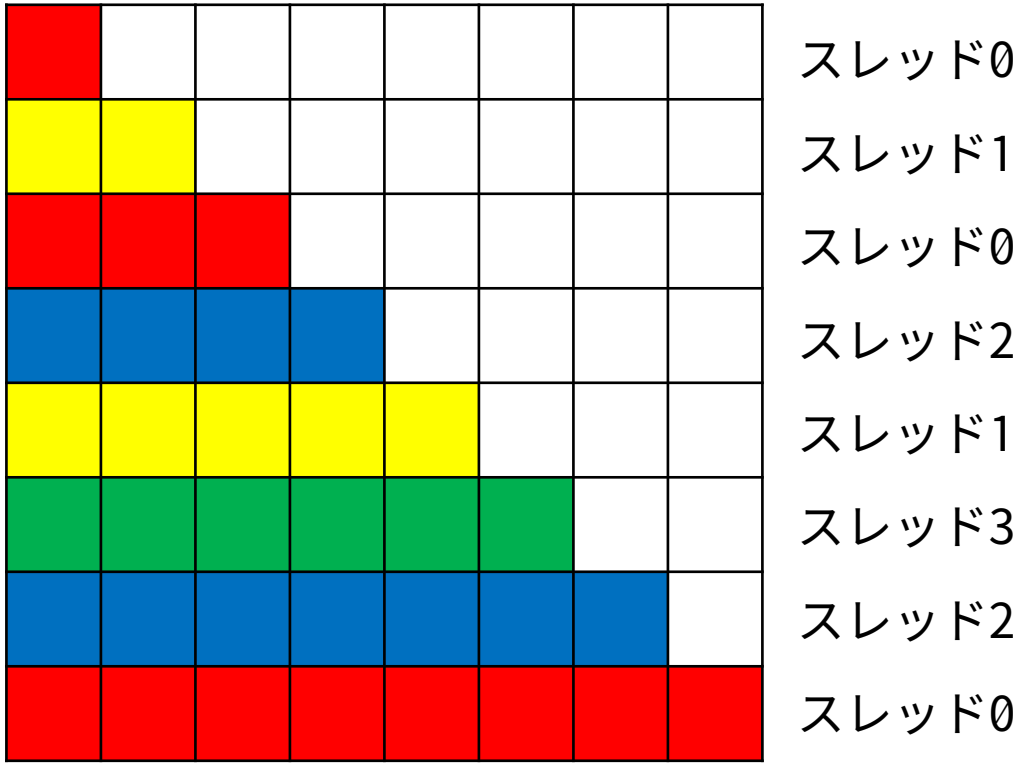

```
#pragma omp parallel for private(j) schedule(static,1)
  for(i=0; i<N; i++){
    for(j=0; j<=i; j++){
      c[i] += a[i][j] * b[j];
    }
  }
```

```
!$omp parallel do schedule(static,1)
  do i=1, N
    do j=1, i
      c(i) = c(i) + a(j,i) * b(j)
    end do
  end do
```

※割り当て粒度をだんだん小さくしていくguidedもあるが、省略

動的な割り当て

- dynamicを指定すると動的な割り当てになる（第二引数で粒度を指定、省略時は1）
  - schedule(dynamic, 1)　の例
  - デメリット：動的に割当を調整する分の手間が増える、キャッシュの引き継ぎができなくなる




```
#pragma omp parallel for private(j) schedule(dynamic)
  for(i=0; i<N; i++){
    for(j=0; j<=i; j++){
      c[i] += a[i][j] * b[j];
    }
  }
```

```
!$omp parallel do schedule(dynamic)
  do i=1, N
    do j=1, i
      c(i) = c(i) + a(j,i) * b(j)
    end do
  end do
```

※あくまでイメージ、こう割り当てられるとは限らない

階層的な並列化

- 行列同士の和は全要素同時に計算できる→外側のループも内側のループも並列化が可能
  - OpenMPは階層的な並列化をサポートしており、parallel節の中でparallel節を使うことが可能

matadd.cに指示文を追加したもの

```
#pragam omp parallel for
for(i=0; i<N; i++){
#pragma omp parallel for
  for(j=0; j<N; j++){
    c[i][j] = a[i][j] + b[i][j];
  }
}
```

matadd.f90に指示文を追加したもの

```
!$omp parallel do
do i=1, N
!$omp parallel do
  do j=1, N
    c(j,i) = a(j,i) + b(j,i)
  end do
end do
```

  - ただし、スレッドを作成したり破棄したりという処理が外側と内側の両方で行われるため、大規模なプログラムかつ多数のスレッドでは性能が低下する可能性がある
  - 多重ループを並列化したいならcollapse節を使う方が良い（高い性能を得やすい）

ループcollapse

- 後続の複数のループをまとめて1つのループと見なして並列化する
  - collapse(n)　n個のループをまとめて1つと見なす

matadd.cに指示文を追加したもの

```
#pragam omp parallel for collapse(2)
for(i=0; i<N; i++){
  for(j=0; j<N; j++){
    c[i][j] = a[i][j] + b[i][j];
  }
}
```

matadd.f90に指示文を追加したもの

```
!$omp parallel do collapse(2)
do i=1, N
  do j=1, N
    c(j,i) = a(j,i) + b(j,i)
  end do
end do
```

  - ループ長N*Nの大きな一つのループを見なして並列化してくれる
  - 多数の計算コアを持つ計算環境で全ての計算コアに十分な仕事を割りふる際などに有効

値の足しあわせと並列処理

- （三角）行列ベクトル積の内側ループは並列化できるのか？

```
#pragma omp parallel for private(j)
  for(i=0; i<N; i++){
    for(j=0; j<N; j++){
      c[i] += a[i][j] * b[j];
    }
  }
```

```
!$omp parallel do
  do i=1, N
    do j=1, N
      c(i) = c(i) + a(j,i) * b(j)
    end do
  end do
```

  - 内側ループで配列の同じ要素に加算しているため並列化できない
  - ただし、足しあわせ処理自体が並列化できないわけではない
  - 以下のような一変数への足しあわせ処理は並列化が可能

```
#pragma omp parallel ... ?
for(i=0; i<N; i++){
  c += a[i] * b[i];
}
```

```
!$omp parallel … ?
do i=1, N
  c = c + a(i) * b(i)
end do
```

リダクション

- reduction指示節を用いることで、値の足しあわせや最大・最小値の取得などが可能
  - parallel節にもfor/do節にも適用可能

```
#pragma omp parallel for reduction(+:c)
for(i=0; i<N; i++){
  c += a[i] * b[i];
}
```

```
!$omp parallel do reduction(+:c)
do i=1, N
  c = c + a(i) * b(i)
end do
```

  - 書き方：reduction(処理内容:対象変数)
    - 処理内容には+, *, -, max, minなどが指定できる
  - 仮に行列ベクトル積の内側ループをreduction並列化するならこのようになる

```
#pragma omp parallel
  for(i=0; i<N; i++){   ←このループは分割実行されない→
#pragma omp for reduction(+:tmp)
    for(j=0; j<N; j++){    ←このループは分割実行される→
      tmp += a[i][j] * b[j];
    }
    c[i] = tmp
  }
```

```
!$omp parallel
  do i=1, N
!$omp do reduction(+:tmp)
    do j=1, N
      tmp = tmp + a(j,i) * b(j)
    end do
    c(i) = tmp
  end do
```

スレッド間の同期

- スレッドが同期を取らずに並列処理を行うと、計算結果の不整合などが生じることがある
- OpenMPでは様々なタイミングで自動的にスレッド間の同期が取られるが、利用者が制御する余地もある
- 暗黙的に同期が取られるタイミングの例
  - for指示節やdo指示節によるループ並列化の終了時
  - parallel指示節による並列実行範囲の終了時
  - sections指示節やsingle指示節（後述）などの対象範囲の終了時

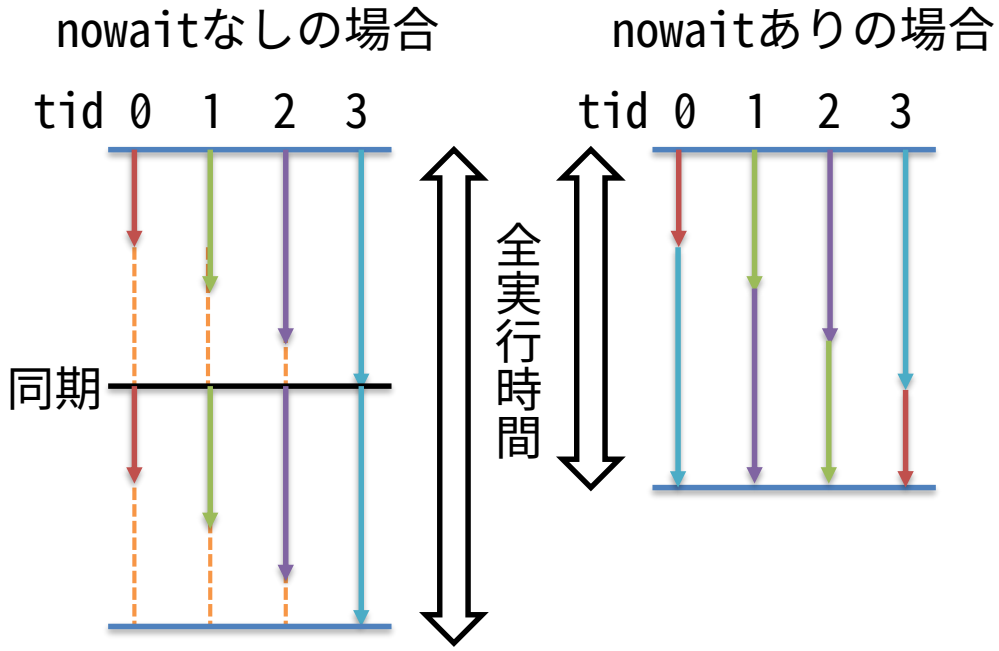
- 暗黙的な同期を排除することもできる
  - nowaitオプションをつける
- 明示的に同期を行うこともできる
  - barrier指示文を使う

nowaitの利用例

- 1つのparallel並列実行部分に複数のループ並列化が存在する場合に、途中の同期を省く
  - C/C++ではループ並列化のはじめに、Fortranではループ並列化の終わりに書く

```
#pragma omp parallel
#pragma omp for nowait
  for(i=0; i<N; i++){
    ......
  }
#pragma omp for
  for(i=0; i<N; i++){
    ......
  }
```

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



  - nowaitがない場合は全スレッドが前半ループを全て終えるまで全てのスレッドが待つ
  - nowaitがある場合は待たずに後半ループを開始する
  - スレッドの負荷にばらつきがある場合などに有効、schedule(dynamic)との併用も考えると良い

明示的な同期

- barrier指示文を使えば任意のタイミングでスレッド間の同期を取ることができる
  - 用途の例
    - parallel並列実行部の途中で同期を取りたい
    - デバッグのために動作順序を調整したい
  - 注意点
    - 全スレッドが到達できるかわからない処理フローの中で実行しないこと
      - 例：for/doループ並列化の中にbarrier同期が書かれていた場合、一部のスレッドは到達できないかも知れない　→　実行時エラーや計算結果の不整合を引き起こす
    - 使いすぎないこと
      - 使いすぎると性能が低下するため本当に必要な場所を見極めること

```
.......
#pragma omp barrier
.......
```

```
......
!$omp parallel barrier
......
```

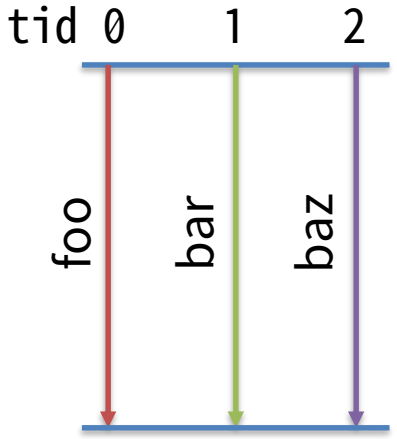
※当然だが、並列実行範囲の中で使う

ループ並列化以外の並列化機能

- sections/section指示節を用いた並列化
  - 連続する依存性のない処理を並列に実行する

```
#pragma omp sections
{
#pragma omp section
  foo();
#pragma omp section
  bar();
#pragma omp section
  baz();
}
```

```
!$omp sections
!$omp section
  foo()
!$omp section
  bar()
!$omp section
  baz()
!$omp end sections
```



実行順序を制限する仕組み（1／2）

- 並列実行範囲の中で並列実行に向かない処理などを行うための機能が用意されている

◆ single と master
- 1スレッドのみが処理を行うことを保証する

```
#pragma omp parallel
{
#pragma omp single
  {
    ......
  }
#pragma omp master
  {
    ......
  }
}
```

```
!$omp parallel
!$omp single
  ......
!$omp end single
!$omp master
  ......
!$omp end master
!$omp end parallel
```

- singleはいずれか1スレッドのみが実行
- masterはマススレッドのみが実行（特定の1スレッド（普通はID=0のスレッド）のみが実行）

◆ criticalとatomic
- 他のスレッドと競合せずに処理を行うことを保証する

```
#pragma omp parallel
{
#pragma omp critical
  {
    ......
  }
#pragma omp atomic
  {
    ......
  }
}
```

```
!$omp parallel
!$omp critical
  ......
!$omp end critical
!$omp atomic
  ......
!$omp end atomic
!$omp end parallel
```

- critical：対象範囲の処理を行えるのは一度に1スレッドのみ（同時に複数のスレッドが処理できない）
- atomic：他のスレッドに割り込まれずに処理をする（読み出して書き込むなど特定の一処理のみ、高速）

singleとmasterの例

single.c

```c
#include <stdio.h>
#include <omp.h>

int main()
{
#pragma omp parallel
  {
    printf("hello, parallel world %d¥n",
           omp_get_thread_num());
#pragma omp single
    printf("single thread %d¥n",
           omp_get_thread_num());
#pragma omp master
    printf("master thread %d¥n",
           omp_get_thread_num());
  }
  printf("bye¥n");

  return 0;
}
```

single.f90

```
program single
  use omp_lib
  implicit none

!$omp parallel
  print *, "hello, parallel world ", &
        omp_get_thread_num()
!$omp single
  print *, "single thread ", &
        omp_get_thread_num()
!$omp end single
!$omp master
  print *, "master thread ", &
        omp_get_thread_num()
!$omp end master
!$omp end parallel

  print *, "bye"
end program single
```

- single部分は実行のたびに異なるIDが表示される可能性がある
- master部分は毎回0が表示される

criticalの例

critical.c

```c
#include <stdio.h>
#include <omp.h>

int main()
{
  int sum1=0, sum2=0;
#pragma omp parallel
  {
        sum1 += 1;
#pragma omp critical
        sum2 +=1;
  }
  printf("sum1 = %d¥n", sum1);
  printf("sum2 = %d¥n", sum2);

  return 0;
}
```

critical.f90

```
program critical
  use omp_lib
  implicit none
  integer :: sum1=0, sum2=0

!$omp parallel
  sum1 = sum1 + 1
!$omp critical
  sum2 = sum2 + 1
!$omp end critical
!$omp end parallel

  print *, "sum1 = ", sum1
  print *, "sum2 = ", sum2
end program critical
```

- sum1はスレッド数分だけ加算されたりされなかったりする
- sum2は必ずスレッド数分だけ加算される

実行順序を制限する仕組み（2／2）

◆ ordered

- 非並列実行時と同じ順序での実行を保証する
- for ordered / do orderedの中で使うと、その部分だけは順序が保証される

ordered.c

```
#pragma omp parallel
  {
#pragma omp for ordered
    for(i=0; i<10; i++){
      printf("parallel1 id %d, executed by %d￥n",
              i, omp_get_thread_num());
#pragma omp ordered
      printf("ordered   id %d, executed by %d￥n",
              i, omp_get_thread_num());
      printf("parallel2 id %d, executed by %d￥n",
              i, omp_get_thread_num());
    }
  }
```

ordered.f90

```
!$omp parallel
!$omp do ordered
  do i=1, 10
     print *, "parallel1 id", i, &
              "executed by ", omp_get_thread_num()
!$omp ordered
     print *, "ordered   id", i, &
              "executed by ", omp_get_thread_num()
!$omp end ordered
     print *, "parallel2 id", i, &
              "executed by ", omp_get_thread_num()
  end do
!$omp end parallel
```

- この例では、同じIDのparallel1→ordered→parallel2が保証されるのに加えて、ordered内の追い越しが起きないことも保証される
- （正直、あまり使いどころがわからない。デバッグ時には便利なこともある。）

実行例（10ループ、4スレッド実行）

- 再掲：同じIDのparallel1→ordered→parallel2が保証されるのに加えて、ordered内の追い越しが起きないことも保証される

```
parallel1 id          1 executed by          0
parallel1 id         10 executed by          3
parallel1 id          4 executed by          1
parallel1 id          7 executed by          2
ordered   id          1 executed by          0
parallel2 id          1 executed by          0
parallel1 id          2 executed by          0
ordered   id          2 executed by          0
parallel2 id          2 executed by          0
parallel1 id          3 executed by          0
ordered   id          3 executed by          0
parallel2 id          3 executed by          0
ordered   id          4 executed by          1
parallel2 id          4 executed by          1
parallel1 id          5 executed by          1
ordered   id          5 executed by          1
parallel2 id          5 executed by          1
parallel1 id          6 executed by          1
ordered   id          6 executed by          1
parallel2 id          6 executed by          1
ordered   id          7 executed by          2
parallel2 id          7 executed by          2
parallel1 id          8 executed by          2
ordered   id          8 executed by          2
parallel2 id          8 executed by          2
parallel1 id          9 executed by          2
ordered   id          9 executed by          2
parallel2 id          9 executed by          2
ordered   id         10 executed by          3
parallel2 id         10 executed by          3
```

Fortranのみ：workshareによる並列処理

- workshare指示文：配列要素の一斉操作を並列化してくれる、Fortranのみの便利機能
  - Fortranにのみ存在する「配列の全要素に対してまとめて処理を行う機能」の並列化と思えば良い

```
!$omp parallel do
do i=1, N
  c(i) = c(i) + a(i) + b(i)
end do
```

```
!$omp workshare
  c(:) = c(:) + a(:) + b(:)
!$omp workshare
```

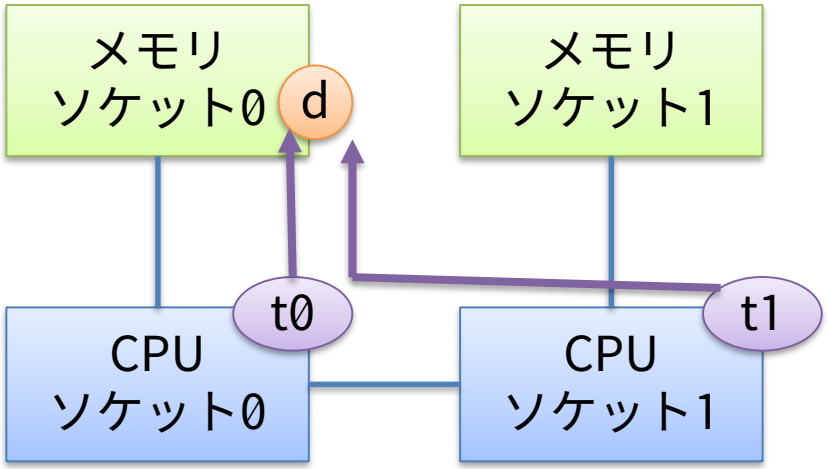
OpenMPプログラムの最適化（高速化）について

- 簡単に並列化を行えるOpenMPだが、最適化については考えられることが色々ある

- 対象ループ
  - どのループから並列化するべきか？キャッシュやメモリへの影響は？
    - プログラム全体の実行時間のうちの多くを占めるループを並列化すると影響が大
    - first touchの影響も考えたい（後述）
- 並列度
  - 十分な並列度がないループを並列化しても性能が上がらない
    - コア数の多いCPUで短いループを並列実行すると、全CPUに仕事を与えることができない
  - メモリアクセスが多いループを並列化してもあまり性能が上がらない
- コストのかかる処理を少なくする
  - barrier, critical, atomicなどは並列化を阻害するもの、少ない方が高速
  - nowaitも適切に使う

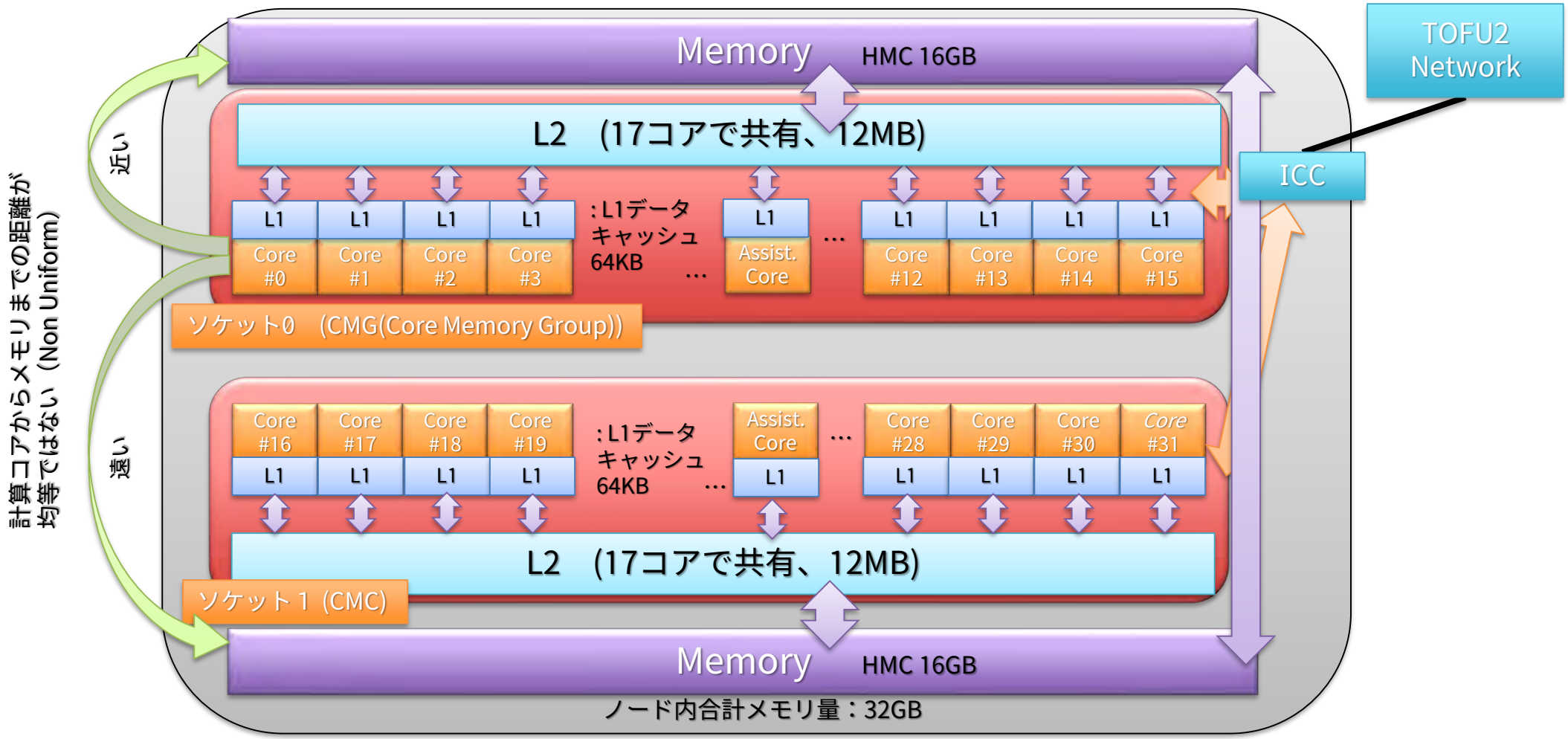
スレッドの配置

- 全CPUコアから全メモリへの距離が均等であれば考える余地はないが、CPUとメモリの距離が不均質な場合はスレッド・計算コア・メモリ（キャッシュ）の配置が性能に大きく影響することがある
  - （正確には、スレッド並列化されていないプログラムでも生じる問題）
- イメージ



- 全てのデータがメモリソケット0上に配置されているとき、CPUソケット0上のスレッドからのデータアクセスとCPUソケット1上のスレッドからのデータアクセスには性能差がある
- 同様に、全てのスレッドがCPUソケット0上に配置されているとき、メモリソケット0上に配置されているデータへのアクセスとメモリソケット1上に配置されているデータへのアクセスには性能差がある

SPARC64XIfxと計算コアの配置



- 1パッケージあたり2ソケットのNUMA (Non Uniform Memory Access)のため、スレッドとメモリの配置によって性能差が生じる　→どのような時に性能差がわかる？どうすれば良い性能を得やすい？

スレッドやメモリの配置の指定方法

- 環境変数を指定することで配置の調整ができる、具体的な指定方法はコンパイラに依存
  - 富士通コンパイラ：FLIB_CPU_AFFINITY
  - Intelコンパイラ： KMP_AFFINITY
  - GNUコンパイラ：GOMP_CPU_AFFINITY
  - PGIコンパイラ：MP_BLIST　　など

- 富士通コンパイラによるスレッドの配置指定例
  - スレッド番号0から順番に、FLIB_CPU_AFFINITYで指定したCPUコアに割り当てる
    - export FLIB_CPU_AFFINITY="0,2,1,3"　　0,2,1,3,0,2,1,3,……
    - export FLIB_CPU_AFFINITY="0-3"　　0,1,2,3,0,1,2,3,……
    - export FLIB_CPU_AFFINITY="0-7:2"　　0,2,4,6,0,2,4,6,……（：の後の値は増分）
  - FLIB_CPUBINDによる指定も可能（これを指定するとFLIB_CPU_AFFINITYは無視される）
    - export FLIB_CPUBIND=chip_pack　1つのCPUチップ（ソケット）に集中的に配置
    - export FLIB_CPUBIND=unpack　多くのCPUチップ（ソケット）に分散して配置

numactl

- CPUやメモリの割り当てを細かく指定することができるプログラム
- 一般的なLinux環境では大変有用だが、FX100では利用不可能

- 使い方：`numactl [options] ./a.out [args]`
  - numactlコマンドに配置オプションと対象プログラムを与えると、配置オプションに対応したコアやメモリの割り当てで対象プログラムを実行してくれる
- 主な配置オプション
  - `--cpunodebind=0,1`　プロセスやスレッドをどのCPUに割り当てるか
  - `--membind=0,1`　メモリをどのCPUに割り当てるか
  - `--interleave=all`　メモリをノード内のメモリ全体にまたがって割り当てる

- 環境変数による割り当てを行ってもnumactlによる割り当てを行っても良い
  - ジョブスクリプトの書きやすさやなどの都合で、MPI+OpenMPハイブリッド並列化を行う場合はnumactlを使うことが多いか

ファーストタッチ最適化（first touch）

- 「最近アクセスしたデータ」はキャッシュに配置され、次回のアクセスが高速化される
- 変数や配列に対するキャッシュは、そのメモリに初めて触れたCPUコアの近くに置かれる
- →最初に変数や配列を初期化する際に、メインループ内でのアクセスと同様のパターンで初期化を行うと、メインループを計算する際にキャッシュに当たりやすくなり高性能
- dynamicスケジューリングの場合などは意味がないので注意

```
for(i=0;i<N;i++){
  array[i] = 0.0;
}

#pragma omp parallel for
for(i=0;i<N;i++){
  array[i] = ……;
}
```

```
#pragma omp parallel for
for(i=0;i<N;i++){
  array[i] = 0.0;
}
```

```
do i=1, N
  arrray(i) = 0.0d0
end do


!$omp parallel do
do i=1, N
  arrray(i) = ……
end do
```

```
!$omp parallel do
do i=1, N
  arrray(i) = 0.0d0
end do
```

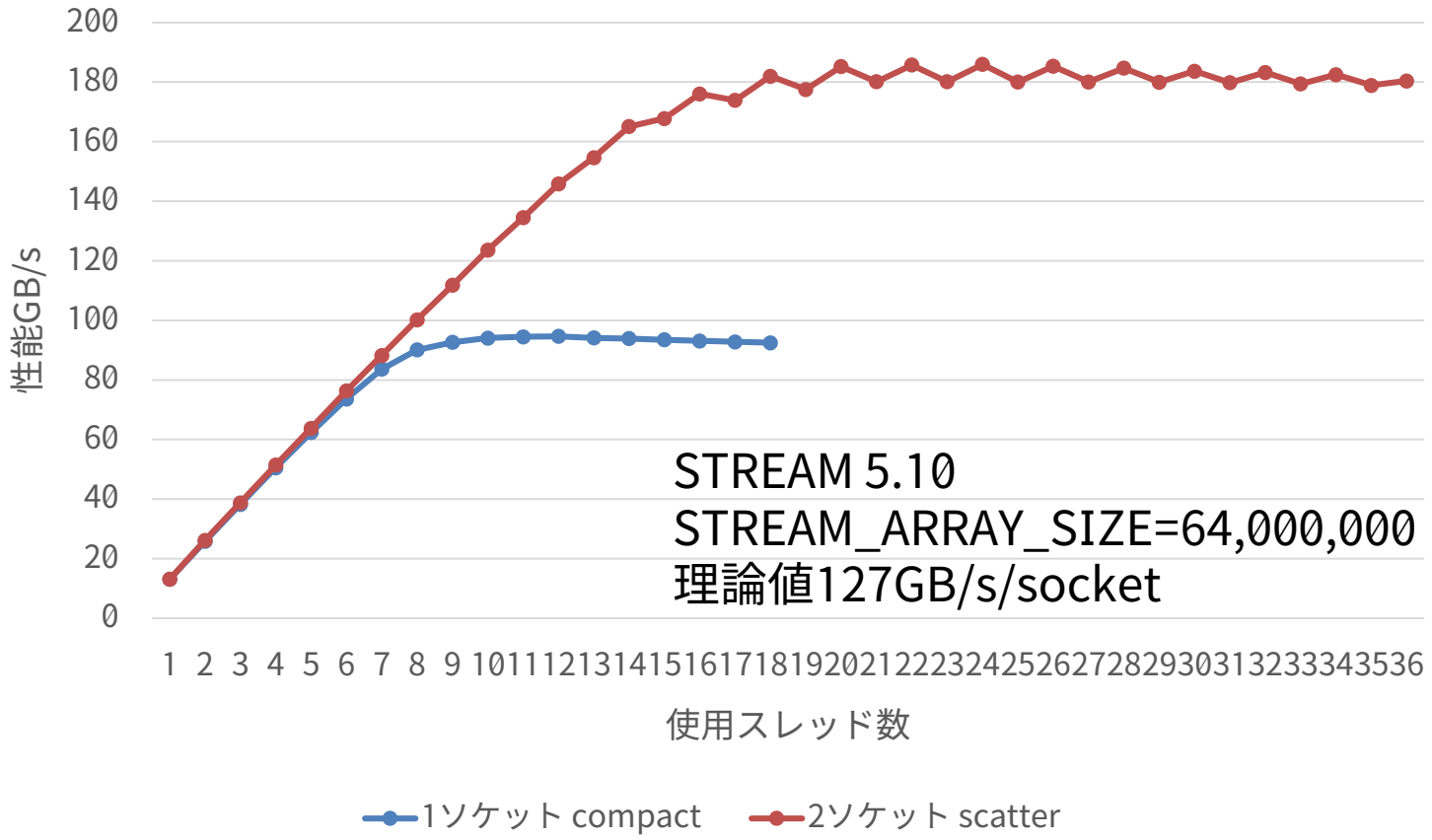
OpenMP並列化していない場合はarrayに対するキャッシュは全てマスタースレッドの近くに配置されるが、
OpenMP並列化すると各スレッドに配置される　→　並列化ループ部のメモリアクセス性能が向上する

スレッド割り当て方法とメモリアクセス性能の関係の例

- メモリ転送性能を測定するSTREAMベンチマークの例
  - 九大ITOで測定した例（2ソケットXeonでわかりやすい結果が得られる環境だったため）
  - STREAM Triadの結果を比較

```
for(j=0; j<STREAM_ARRAY_SIZE; j++){
  a[j] = b[j] + scalar * c[j];
}
```

  - 1ソケット compact affinityでは最大95GB/s
  - 2ソケットscatter affinityでは最大187GB/s
  - メモリが各ソケットに別々につながっているため2ソケットにまたがってスレッドを配置することで最大性能が得られる



プロファイラの活用

- OpenMPプログラムの解析にもプロファイラは有効
  - 例えば手動でOpenMPプログラムの挙動を詳しく調べたい場合、スレッドごとの実行時間を測定して書き出すような処理を並列実行部分ごとに記述せねばならず、手間がかかる
  - プロファイラを使えば、スレッドごとの実行時間のばらつきなども一目で分かる
  - ただし、実行するたびに実行時間が変わるようなプログラムは根本的に難しい
    - プロファイルを取るたびに変わってしまう
  - 基本的に、プロファイラを使ってプログラムを実行すると実行時間が延びるため、測定したい部分のみを抽出したプログラムを用意したり、測定範囲の指定をしっかり行うことも重要
  - 基本的な使い方は「FX100システム利用型 講習会 共通資料」のとおり

今回は扱わなかった指示文・機能の例

- flush指示文
  - メモリ書き込みの保証をするもの、適切なバリア同期（barrier）の利用をすればほぼ不要
- タスク処理・GPU対応など新しいOpenMPの機能
  - task：タスク処理を行うもの、動的なジョブの生成（生産者消費者問題）や再帰処理の並列化などで有用
  - target：GPUに処理を行わせる際に使う

まとめ

- OpenMPの仕様や使い方について紹介した
  - わずか数行の指示文でプログラムの並列化ができる、とても便利なもの
    - 記述量が少ない
    - ループ1つからなど段階的な並列化も行いやすい
  - HW/SW提供ベンダーを問わず現在のマルチコアCPUでは一般的に利用可能なノード内並列化の手法であるため、今後のスパコンでも活躍してくれるはずである
  - ハードウェアの特徴と対象プログラムの中身とOpenMPの適切な使い方を知ることで、最大限の性能を得ることができるだろう

- 残りの時間は演習時間とします
  - 単純な行列積とCG法の逐次コードを提供していますので、指示文を挿入して並列化してみよう
    - どのループにどのように指示文を入れれば良いだろうか？問題サイズが変わっても同じ指示文で良いだろうか？

実習：行列積の並列化

- 単純な三重ループによる行列積のソースコードを提供している
  - サンプルコード：matmul.c, matmul.f90
  - いずれかのループに並列化指示文を追加すれば並列実行できる
  - 実験例
    - スレッド数を変えてみたり、並列化するループを変更してみたり、一次変数を使ってみたりして性能を比較してみよう
    - 問題サイズを変えてみて（大きくしてみて）性能を比べてみよう

実習：CG法の並列化

- 前処理のない単純なCG法のソースコードを提供している
  - サンプルコード：cg.c, cg.f90
  - 並列可能なループがたくさんある（反復ループ内の各行列・ベクトル計算ループがいずれも並列化可能）
  - 特に行列ベクトル積計算の並列化が性能に大きく影響するため並列化の効果が大きい

  - 実験例
    - 並列化するループを変更して実行時間を比べてみよう
    - first touchが効くか調査してみよう