2021年5月31日（月）10:00-17:30
第14回スーパーコンピュータ「不老」利用型講習会 OpenMP（初級）
Zoomによるオンライン開催

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14回スーパーコンピュータ「不老」利用型講習会 OpenMP（初級）

名古屋大学 NAGOYA UNIVERSITY　ITC　不老

プログラムと時間の目安

- 9:30 Zoom接続開始

- 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
  - スーパーコンピュータ「不老」の使い方、ジョブの投入方法、実行確認
  - スーパーコンピュータ「不老」へのログイン

- 13:00 – 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

はじめに

- 講師について
  - 名前：大島　聡史（おおしま　さとし）
    - 情報基盤センター　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）」

サンプルプログラム（ソースコード）について

- 資料中で用いているソースコードは /home/center/a49979a/share/20210531.tgz にて公開しています
- cpでコピーして tar zxvf 20210531.tgz などで展開して使ってください
- 見た目の都合等から公開コードと資料中のコードが完全一致しているわけではない点に注意してください
  - 資料の1ページに収まるように省略していることなどがあります

- 講習会用のアカウントは本日中のみ有効です
- 必要なファイルはscpやsftpで手元にコピーしておいてください

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

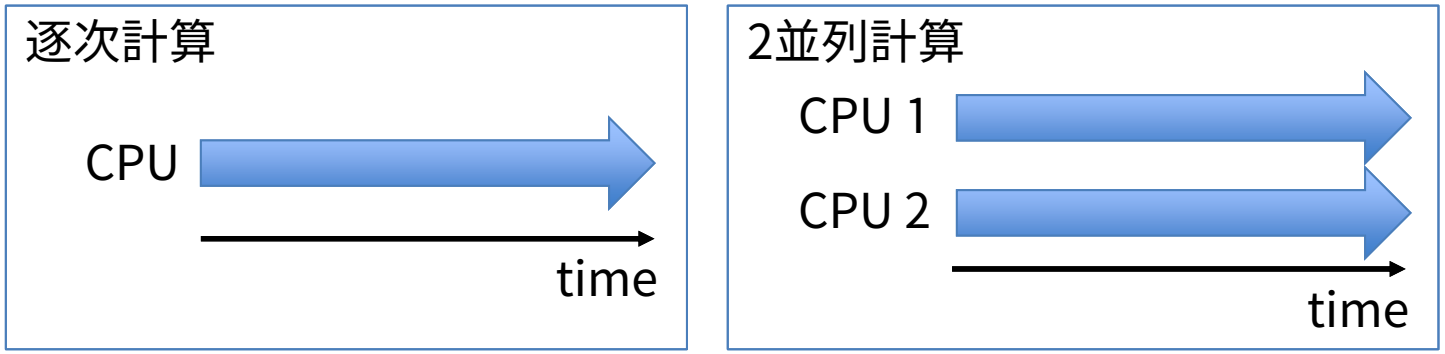
- 「計算科学のためのHPC技術1 」
  - 下司雅章 (編集), 片桐孝洋 , 中田真秀, 渡辺宙志, 山本有作, 吉井範行, Jaewoon Jung, 杉田 有治, 石村和也, 大石進一, 関根晃太, 森倉悠介, 黒田久泰，著
  - 大阪大学出版会、ISBN-10: 4872595866、ISBN-13: 978-4872595864、発売日：2017年4月3日
- 「並列プログラミング入門：サンプルプログラムで学ぶOpenMPとOpenACC」
  - 片桐 孝洋 著
  - 東大出版会、ISBN-10: 4130624563、ISBN-13: 978-4130624565、発売日：2015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

内容

- OpenMPを学ぶ前に
  - 並列計算の基礎
- OpenMPの基礎
  - OpenMPの仕様と使い方
  - 簡単なベクトル計算・行列計算の並列化
- OpenMPプログラムの最適化（高速化）
  - 一般的なOpenMPプログラムの最適化
  - スーパーコンピュータ「不老」向けのプログラム最適化
- まとめ
- 実習用サンプルソースコードの紹介

並列計算とは？

- 並列計算とは何か？並列プログラミング（並列化プログラミング）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



- 並列？　並行・平行？
  - 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
      - 1つのアプリケーションを高速に実行するために分割して同時実行するイメージ
  - 並行・平行
    - 何らかの処理を同時に行う、時分割多重のような疑似並列処理も含む
      - 直接関係のない複数の処理を同時に行うイメージ
      - 多数のプログラムの動作を調停するシステムソフトウェア（OSなど）の話をする場合はこちら
        - » 直接の関係がない処理同士だが、同じ資源を利用するために調整が必要

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、並列計算を行わなければ高い性能を得られない
    - PC用CPUもスマートフォン用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」並列計算向けのプロセッサ
  - HWの性能を十分に引き出すには
    並列計算が必須
    - 逐次計算では搭載された全ての計算コアを
      使い切れず、性能を活かしきれない

Surface Pro7の公式サイトの例。
プロセッサの選択肢が3つあり、コア数も明記されている。
「コア数＝性能」ではないが、重要な性能の目安ではある。
画像引用元：https://www.microsoft.com/ja-jp/surface/devices/surface-pro-7/tech-specs（2021.04.17の掲載内容）

最近のマルチコアCPUの性能の例

CPU名	コア数（スレッド数）	クロック周波数×同時実行命令数	理論演算性能（FP64）	理論メモリバンド幅	備考
SPARC64VIIIfx	8 (8)	2.0 GHz × 8	128 GFLOPS	64 GB/s	「京」コンピュータに搭載
SPARC64XIfx	32 (32)	2.2 GHz × 16	1.1 TFLOPS	480 GB/s	名大旧システムFX100に搭載
A64FX	48 (48)	2.2 GHz × 32	3.3 TFLOPS	1,024 GB/s	「富岳」、「不老」Type Iに搭載
Intel Xeon Gold 6230	20 (40)	3.00 – 3.70 GHz × 32	1.3 TFLOPS	140.75 GB/s	「不老」Type IIに搭載
IBM POWER9	22 (88)	3.07 GHz × 8	540 GFLOPS	170.7 GB/s	Summit（2018年11月時点の世界最速スパコン）に搭載
Intel Core i9 10900K	10 (20)	3.70 – 5.30 GHz × 16	592 GFLOPS	45.8 GB/s	最近のPC向けハイエンド
AMD Threadripper 3990X	64 (128)	2.90 – 4.30 GHz × 16	3.0 TFLOPS	102.4 GB/s	最近のPC向けハイエンド

- 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OpenMP

- 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

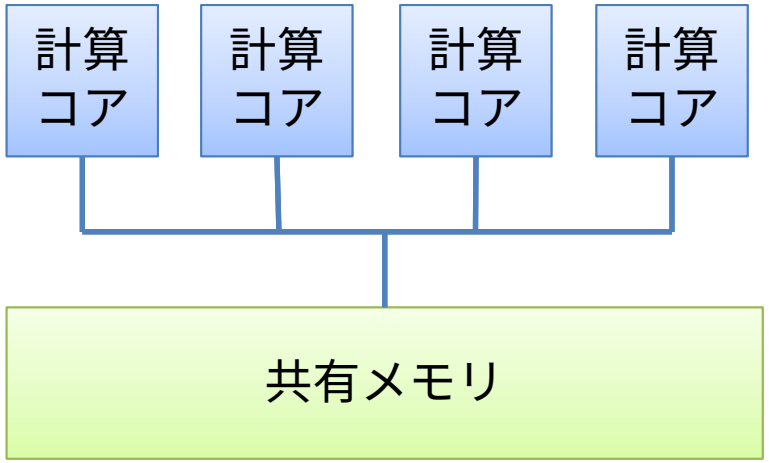
- OpenMPはスパコンのノード内並列計算にも活用されており、個人用のPCでも世界最大規模のスパコンでも大変重要なツール
  - ノード内並列化はOpenMP、ノード間並列化はMPI、などの使い分けがされる
  - 高性能計算（High Performance Computing, HPC）やその周辺分野ではとてもよく使われている基本的なツール、マルチコアCPUの性能をとても簡単に活用できる、高性能計算の研究を行わなくても知っていて損はない

OpenMPとは？

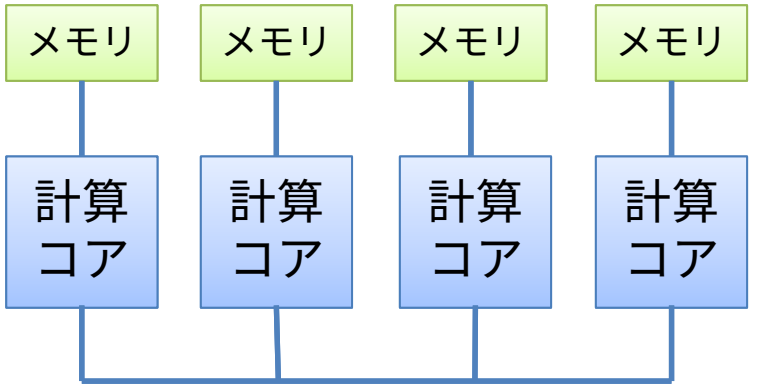
- 共有メモリ型並列計算機にて（主にループ計算を）簡単に並列実行することができるもの
- OpenMP自体はプログラミング言語ではなく、C/C++やFortranにコード（指示文）を追加して使う
  - 本講習会ではCとFortran90（自由形式）を想定して例示する
- スレッド並列化のための道具
  - スレッドとは？プロセスとは？を説明するのは今回の本題ではないため省略
    - 1ノード上で複数の処理を並列に実行するとき、その1つ1つの処理の流れをスレッドと呼んでいる、くらいの認識で良い
    - 単数の処理は、複数の処理の数が1である特殊な例
    - スレッド間ではメモリ（データ、変数・配列）を通信なしで共有できる（共有してしまう）
    - 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）
- 多くのコンパイラが対応しており、様々な計算機環境で利用可能

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）

並列計算環境を利用するための手段

- 並列計算環境が普及した今日では様々な「並列化のための道具」が使われている
  - バラバラだと提供側・利用側ともに不便なため共通化されることが多い
    - ある環境向けに書いたものが別の環境でも利用できる（互換性）
    - 性能まで互換性があるとは限らない点には注意が必要（性能可搬性）
      - ある環境では有効であった最適化が別の環境では性能低下要因になることも
      - 環境が変わってもその環境ごとに常に最大の性能が得られるようにするための研究も行われている→自動チューニング
- 自動化はできないのか？
  - 全く不可能なわけではない、ある程度はコンパイラ等が行ってくれる
  - 「コンパイラ様のご機嫌を伺う」コードを書く必要がある、コンパイラによって差が大きい、簡単なコードでないとうまくいかないことが多い
    - 単純な行列積などコードの形状と最適化のパターンが決まっているものは人が書くよりコンパイラやライブラリの方が得意
  - OpenMPは簡単・少量の記述で効果的な並列高速化を可能とする（良いところ取り、規格化により汎用性・可搬性も担保）

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];
      }
```

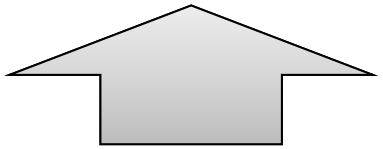
```
PE2   for(i=0; i<N; i++){
        D[i] = E[i] + F[i];
      }
```

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
```

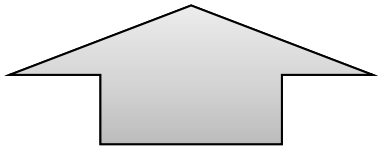
```
PE2   do i=1, N
        D(i) = E(i) + F(i)
      end do
```

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
```

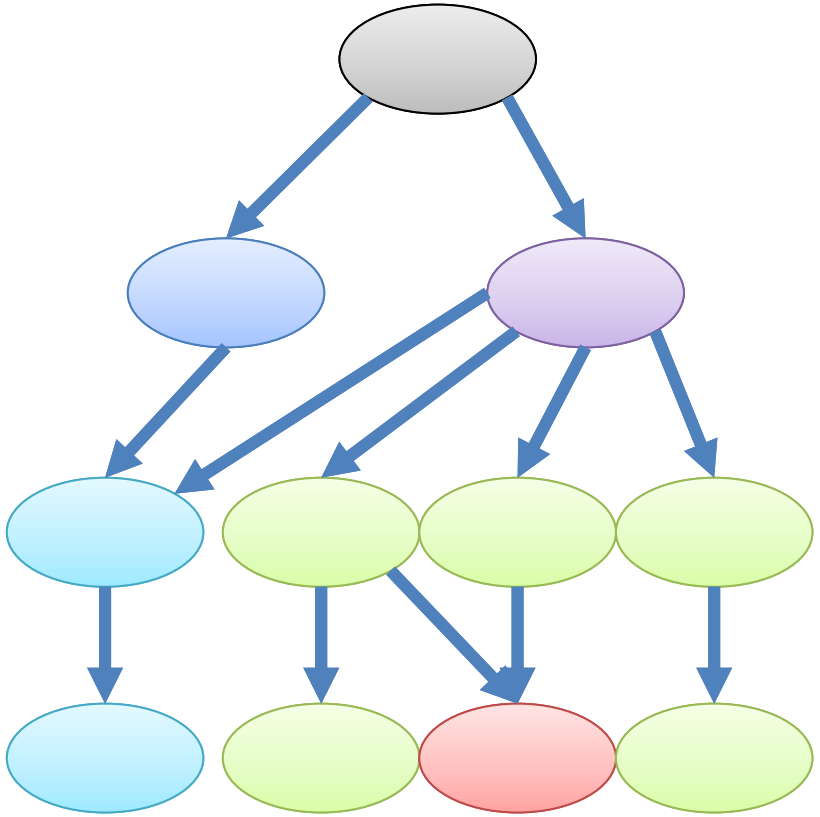
```
PE2   do i=N/2+1, N
        A(i) = B(i) + C(i)
        D(i) = E(i) + F(i)
      end do
```



- 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



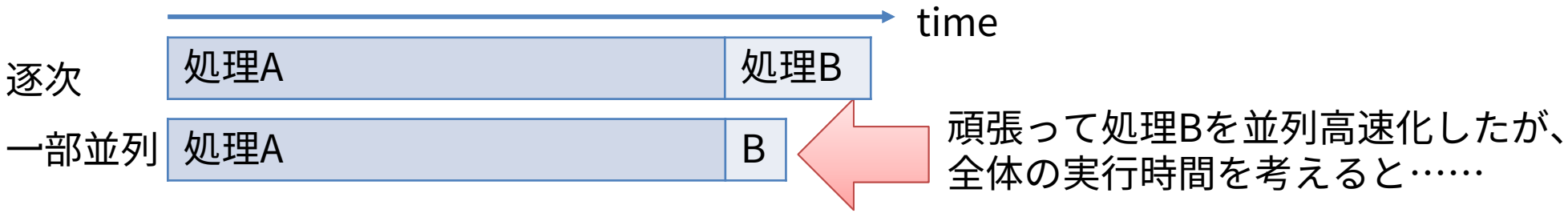
並列計算の限界

- 並列化できないプログラムも少なくない
  - 計算順序に制約がある場合は困難
    - 工夫により可能となることもある

```
for(i=1; i<N; i++){
 A[i] = A[i-1] + B[i];
 C[i] = A[i] + D[i];
}
```

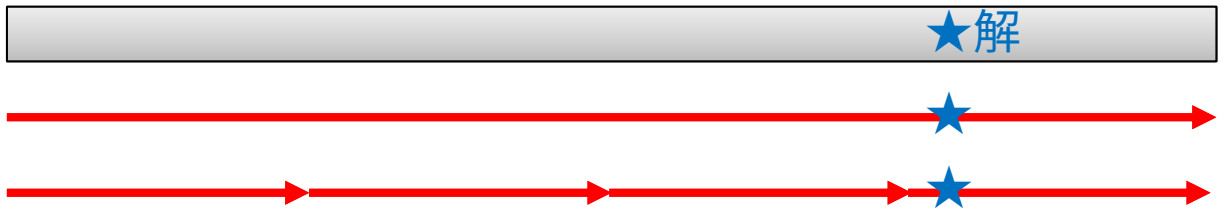
例：ループイタレーション間にも、
同じループイタレーション内にも、
依存関係がある ⇒ 並列化が困難

- （プログラム高速化全般に言えることだが）
  速くした部分しか速くならない



頑張って処理Bを並列高速化したが、
全体の実行時間を考えると……

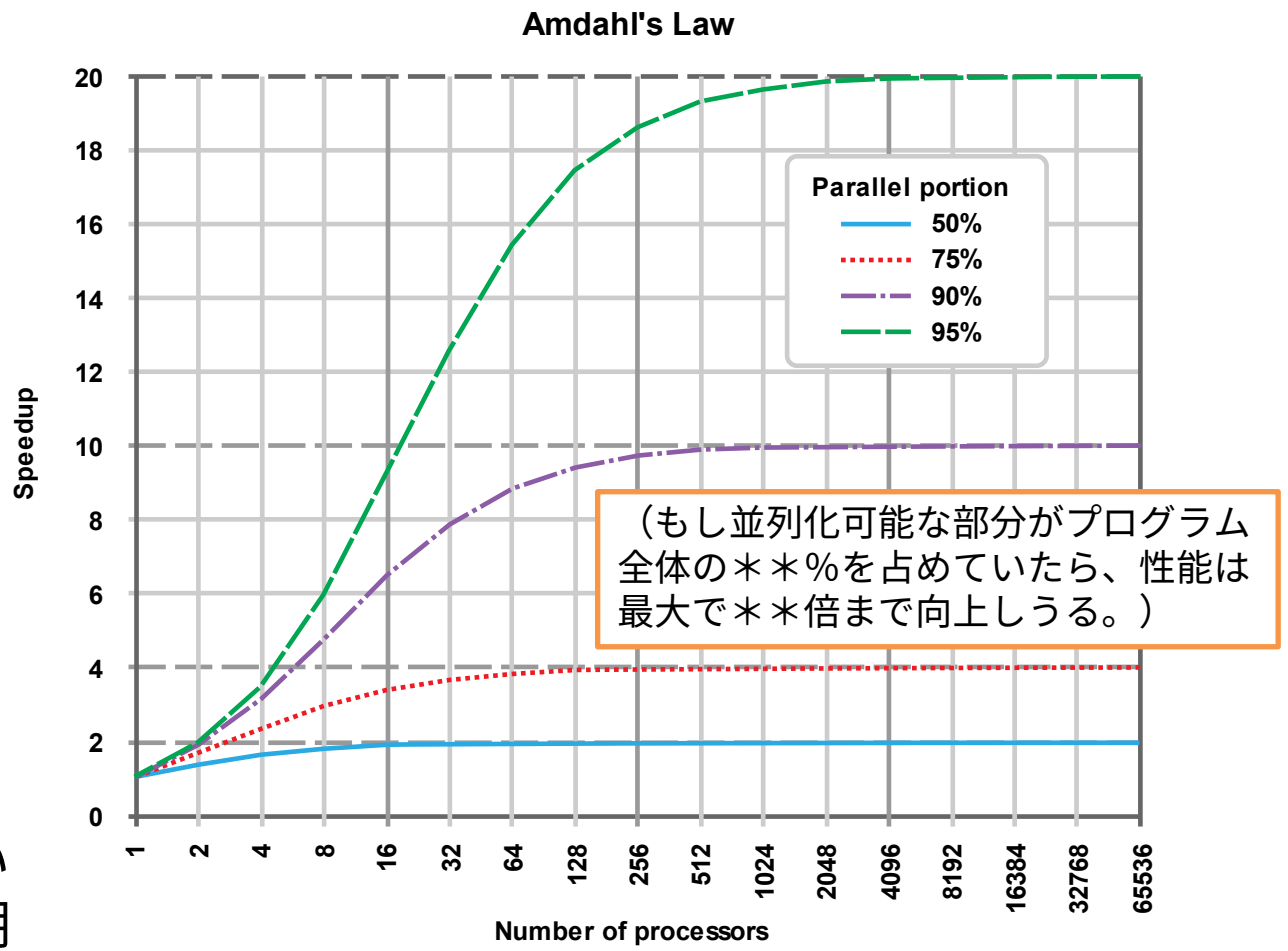
  - 探索問題などでは分割数（並列度）よりずっと速くなることもある
    - 探索対象
    - 逐次探索
    - 並列探索



性能向上率と並列化の限界

- アムダールの法則
  - 並列化できる範囲の割合と、
    並列化により得られる性能向上の関係

- そもそも対象問題（プログラム）に
  十分な並列度がなければならない
- 並列化できない部分が大きいと
  いくら並列化しても時間が短くならない
- 並列化に伴い通信などのオーバーヘッドが
  増える可能性がある
  - 逐次実行では不要であった通信が増える
  - 共有メモリモデルであるOpenMPに通信はないが、スレッドの生成・破棄やスレッド間の同期は必要でありオーバーヘッドとなる



※グラフはWikipedia「アムダールの法則」から引用

内容

- OpenMPを学ぶ前に
  - 並列計算の基礎
- OpenMPの基礎
  - OpenMPの仕様と使い方
  - 簡単なベクトル計算・行列計算の並列化
- OpenMPプログラムの最適化（高速化）
  - 一般的なOpenMPプログラムの最適化
  - スーパーコンピュータ「不老」向けのプログラム最適化
- まとめ
- 実習用サンプルソースコードの紹介

OpenMPを用いた並列化プログラミングの簡単な例

- 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

  - これらの関数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

cond1.c

```
#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

cond1.f90

```
!$ 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 ……

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

do/parallel do など一部の指示文はendを省略可能
（閉じなくても範囲が自明）

- 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）

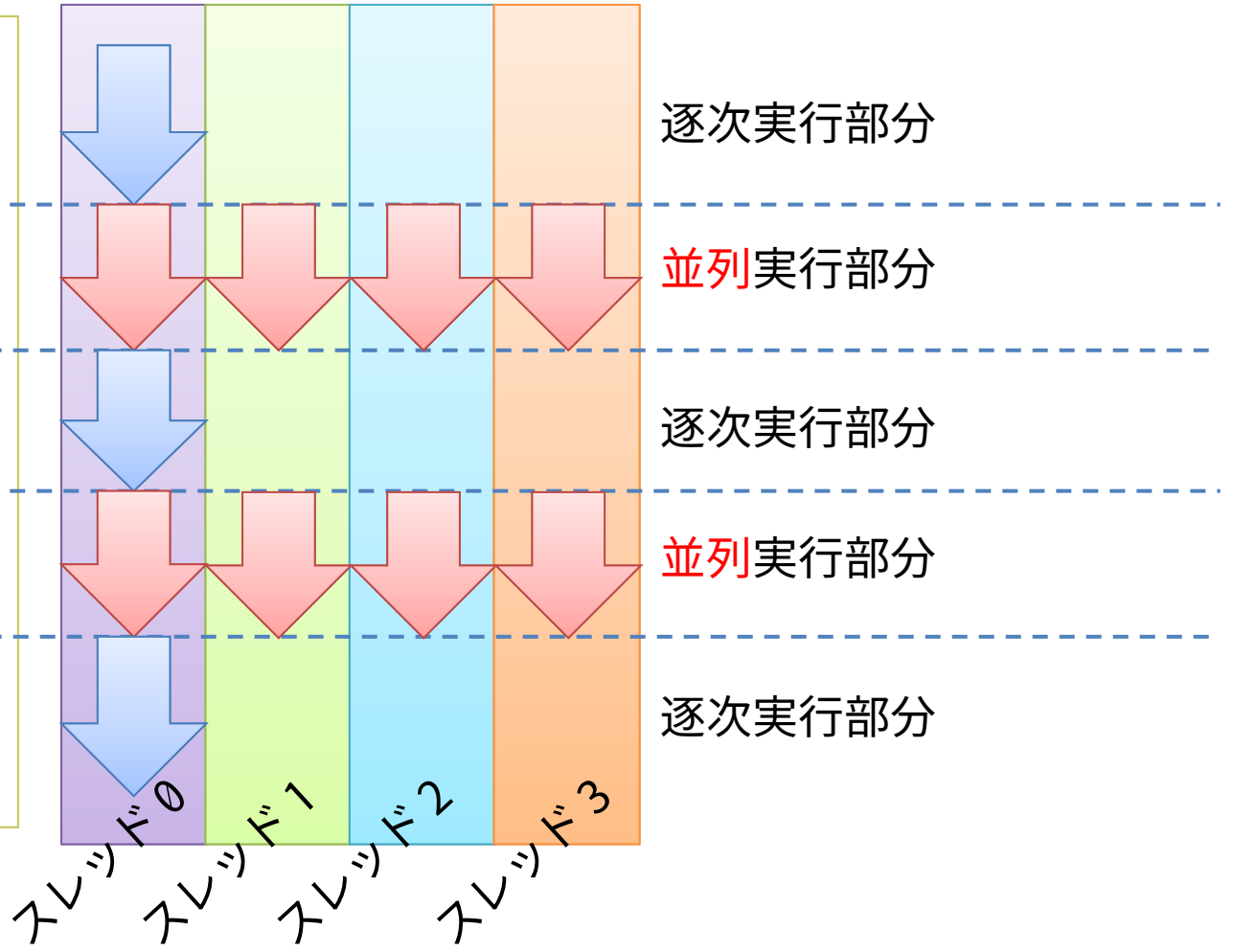
```
#pragma omp hoge ¥
  foo bar
```

```
!$omp hoge &
!$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

「不老」Type I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,openmp -Nfjomplib source.c`
- `frtpx -Kfast,openmp -Nfjomplib source.f90`

※-Kfast,openmpは-Kfast -Kopenmpと同じ

- 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によって並列実行数を指定可能
  - 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
- Type Iサブシステム向けの推奨コンパイル時オプション・実行時オプションについては終盤で改めて解説する

注意：ログインノードと計算ノードの違いについて

- Type Iサブシステムはログインノードと計算ノードが異なるアーキテクチャ
  - 計算ノード：CPUがA64FX、ARM系のCPU
  - ログインノード：CPUがXeon、x86系のCPU
- CPUが大きく異なるため、同じプログラム（実行可能ファイル）を実行できない
- fccpxやfrtpxで生成できるのは計算ノード向けの実行可能ファイル
- ちょっとしたテスト実行でも計算ノード上で実行する必要がある

- わざわざジョブとして実行するのがめんどくさい、といった場合は？
  - gccでコンパイルすればログインノードですぐに実行できる
    - 高い並列度・メモリを多く使う・実行時間が長いプログラムは他の利用者の邪魔になるため禁止
      - 環境変数OMP_NUM_THREADSを指定してスレッド数を減らすなどすること（後述）
  - インタラクティブジョブ実行すればコンパイルしてすぐに実行できる
    - fccpx/frtpxの代わりにfcc/frtを使う
    - 利用ポイントはどんどん消費されるため注意が必要

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

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

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

  return 0;
}
```

hello1.f90

```f90
program hello
  use omp_lib
  implicit none

  print *, "hello, world"

!$omp parallel
  print *, "hello, parallel world ", &
       omp_get_thread_num(), "/", &
       omp_get_num_threads()
!$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
  - /home/center/a49979a/share/20210531.tgz
- ファイルをコピーして展開しておく
  - cp /home/center/a49979a/share/20210531.tgz ./
  - tar zxvf ./20210531.tgz
  - cd 20210531
- 計算ノード向けにコンパイルする
  - fccpx -Kopenmp -Nfjomplib -o hello1 hello1.c
  - frtpx -Kopenmp -Nfjomplib -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

  - -Kfast オプションが有効だとコンパイラによる最適化が働き、簡単なプログラムだとプログラムの書き方の良し悪しよりも最適化が性能に大きく影響してしまうため、簡単なプログラムの性能比較時（本講習会のプログラムを試す際）には有効にしないほうが良い
  - さらに -O0 オプションを追加すれば最適化が全く行われなくなる

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<10; i++){
      c[i] = a[i] + b[i];
    }
  }
```

loop1.f90

```
!$omp parallel
!$omp do
  do i=1, 10
    c(i) = a(i) * b(i)
  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<10; i++){
    c[i] = a[i] + b[i];
  }
```

loop2.f90

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

```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

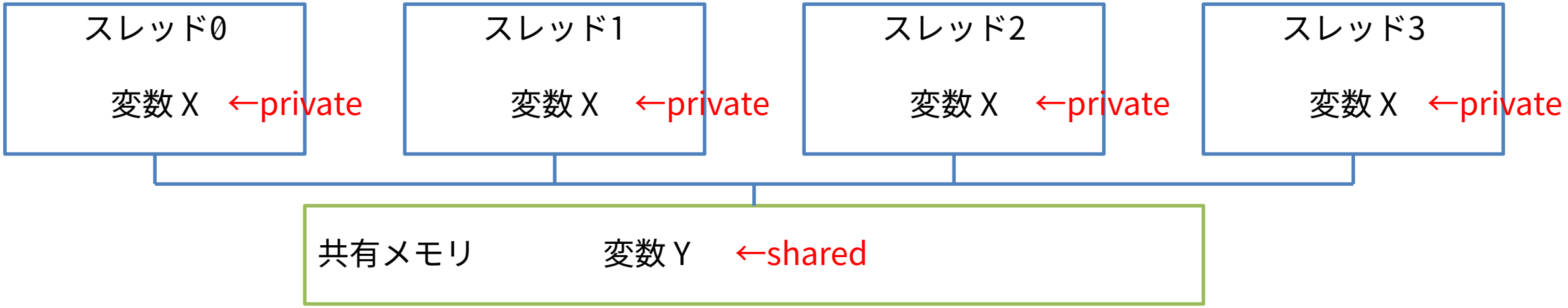
```
1  program private
2    use omp_lib
3    implicit none
4    integer :: i
5
6    print *, "hello, world"
7
8  !$omp parallel
9    i = omp_get_thread_num()
10   print *, "hello, parallel world ", i
11 !$omp end parallel
12
13   print *, "bye"
14 end program private
```

富士通Fortranコンパイラの場合は警告が出る

```
$ frtpx -O0 -g -Kopenmp -Nfjomplib -o private1f private1.f90
Fortran diagnostic messages: program name(private)
  jwd2890i-i  "private1.f90", line 9: SHARED変数'i'への定義がスレッド間で競合する可能性があります。
```

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];
  }
  for(i=0;i<10;i++)printf("c[%d] = %d¥n", i, c[i]);
  return 0;
}
```

tidがprivate化されるため、各スレッドは配列Cのうちスレッド ID番だけを確実に更新する

private2.f90

```f90
program private
  use omp_lib
  implicit none
  integer :: i, j, a(10), b(10), c(10)
!$omp parallel do
  do i=1, 10
    do j=1, 10
      c(i) = c(i) + a(j) + b(j)
    end do
  end do
!$omp end parallel do

  print *, a
  print *, b
  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、指定がなくてもprivate化される（問題なく並列実行される）

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は同時に指定可能：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などが使えていた時点で予想できることではある

func1.c

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

int func(int ret)
{
  return ret + 1;
}
```

```c
int main(){
#pragma omp parallel
  {
    int ret;
    int tid;
    tid = omp_get_thread_num();
    ret = func(tid);
    printf("%d: %d¥n", tid, ret);
  }
  return 0;
}
```

func1.f90

```
program hello
  use omp_lib
  implicit none
  integer ret, tid
!$omp parallel private(ret,tid)
    tid = omp_get_thread_num()
    ret = func(tid)
    print *, tid, ": ", ret
!$omp end parallel
```

```
contains
integer function func(ret)
  integer ret
  func = ret + 1
end function func

end program hello
```

- 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_wtime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、最適化オプションを付けない
  - 測定対象時間が短すぎたり外乱があると正確に測れない（誤差が大きくなる）点にも注意

実行時間の測定：補足

- プログラム全体の実行時間を測定するだけならtimeコマンドを使っても良い
  - time ./a.out

- コンソール上の処理でもバッチジョブ内でも使える
  - 実行例

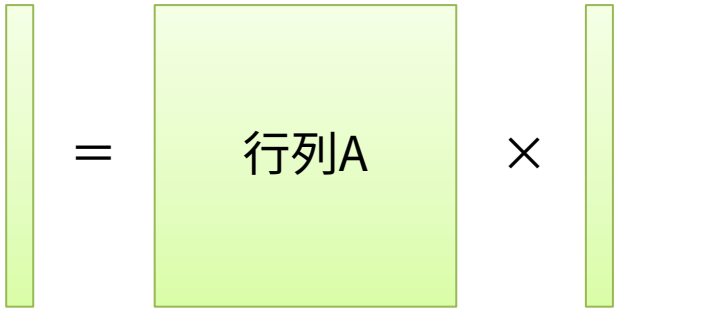
```
$ time fccpx -Kfast,openmp -Nfjomplib -o hello1 hello1.c

real    0m0.129s
user    0m0.055s    ←コンパイルにかかった時間
sys     0m0.050s
```

簡単なベクトル計算・行列計算の並列化を考える

◆ ベクトル同士・行列同士の和を並列計算してみる

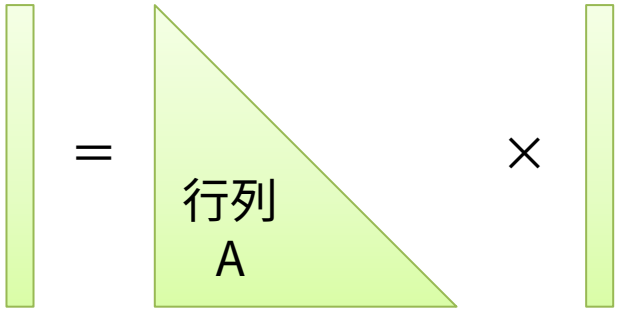
ベクトルCまたは行列C ＝ ベクトルAまたは行列A ＋ ベクトルBまたは行列B

◆ 行列ベクトル積を並列計算してみる



ベクトルC　　ベクトルB

◆ 三角行列ベクトル積を並列計算してみる



ベクトルC　　ベクトルB

実習：ベクトル同士の和、行列同士の和

◆ ベクトル同士・行列同士の和を並列計算してみる（vecadd1, matadd1）

ベクトルCまたは行列C	＝	ベクトルAまたは行列A	＋	ベクトルBまたは行列B

- 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）

実習手順

- まずはそのままコンパイルして実行
  - fccpx -Kopenmp -o vecadd1 vecadd1.c　または　frtpx -Kopenmp -o vecadd1 vecadd1.f90
  - ジョブスクリプトjob_vecadd1.shを作成（job_hello1.shを元に、実行ファイル名を変えるだけ）
  - pjsub job_vecadd1.sh　ジョブを実行
- 並列化して実行
  - cp vecadd1.c vecadd2.c　または　cp vecadd1.f vecadd2.f90　でソースコードを複製
  - cp job_vecadd1.sh job_vecadd2.sh　ジョブスクリプトも複製
  - ソースコード（vecadd2.c）を編集、指示文を挿入する
  - fccpx -Kopenmp -o vecadd2 vecadd2.c　または　frtpx -Kopenmp -o vecadd2 vecadd2.f90
  - ジョブスクリプト（job_vecadd2.sh）をvecadd2が実行されるように修正
  - pjsub job_vecadd2.sh　ジョブを実行
  - 動作を確認したら、さらに配列長を変えたりスレッド数を変えたりして試してみよう
- コンパイラによる最適化による影響が大きすぎるため、スレッド数と実行時間の関係を確認したいときはコンパイルオプション -Kfast をつけないこと

ベクトル加算

- ここまでの内容を理解していれば特に難しいところはないはず
  - 主計算ループに指示文を一行（一組）入れる、以外に思いつかない

vecadd1.c

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

int main(){
  const int N=10;
  int i;
  double a[N], b[N], c[N];
  double d1, d2;
  // 配列の初期化は省略
  printf("vecadd N=%d￥n", N);
  d1 = omp_get_wtime();
  for(i=0; i<N; i++){
    c[i] = a[i] + b[i];
  }
  d2 = omp_get_wtime();
  printf("%e msec￥n", d2-d1);
  for(i=0; i<N; i++) printf("%.2f￥n", c[i]);
  printf("￥n");
  return 0;
}
```

主計算ループ

vecadd1.f90

```f90
program vecadd
  use omp_lib
  implicit none
  integer, parameter :: N = 10
  integer :: i
  double precision :: a(N), b(N), c(N)
  double precision :: d1, d2

  ! 配列の初期化は省略

  print *, "vecadd N=", N
  d1 = omp_get_wtime()
  do i=1, N
     c(i) = a(i) + b(i)
  end do
  d2 = omp_get_wtime()

  print *, d2-d1, "msec"
  write(*,'(f6.2)') c
end program vecadd
```

主計算ループ

行列加算

- 行列加算も簡単に並列化できるが、主計算部が二重ループである点に注意が必要
  - まずは外側ループを並列化してみよう（特にC言語版は内側ループのループ変数の扱いに注意）

※privateの指定が必要

matadd1.c

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

int main(){
  const int N=10;
  int i, j;
  double a[N][N], b[N][N], c[N][N];
  double d1, d2;
  // 配列の初期化は省略
  printf("matadd N=%d￥n", N);
  d1 = omp_get_wtime();
  for(i=0; i<N; i++){
    for(j=0; j<N; j++){
      c[i][j] = a[i][j] + b[i][j];
    }
  }
  d2 = omp_get_wtime();
  // 出力も省略
  return 0;
}
```

主計算ループ

matadd1.f90

```f90
program matadd
  use omp_lib
  implicit none
  integer, parameter :: N = 10
  integer :: i, j
  double precision :: a(N,N), b(N,N), c(N,N)
  double precision :: d1, d2

  ! 配列の初期化は省略

  write(*,fmt="(A,i,A)") "matadd N=", N," "
  d1 = omp_get_wtime()
  do i=1, N
     do j=1, N
        c(j,i) = a(j,i) + b(j,i)
     end do
  end do
  d2 = omp_get_wtime()
  ! 出力も省略
end program matadd
```

主計算ループ

実際にやってみましょう

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

matadd1.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];
  }
}
```

matadd1.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、スレッドの作成や破棄が外側と内側の両方で行われるため、オーバーヘッドとなる
  - 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

matadd1.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];
  }
}
```

matadd1.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
```

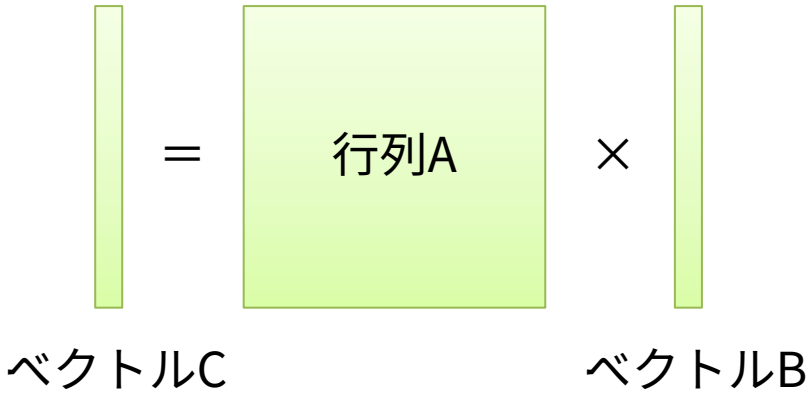
  - ループ長N*Nの大きな一つのループを見なして並列化してくれる
  - 多数の計算コアを持つ計算環境で全ての計算コアに十分な仕事を割りあてる際などに有効
  - コンパイラによって
    このようなループに変換
    されていると思えば良い　→

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

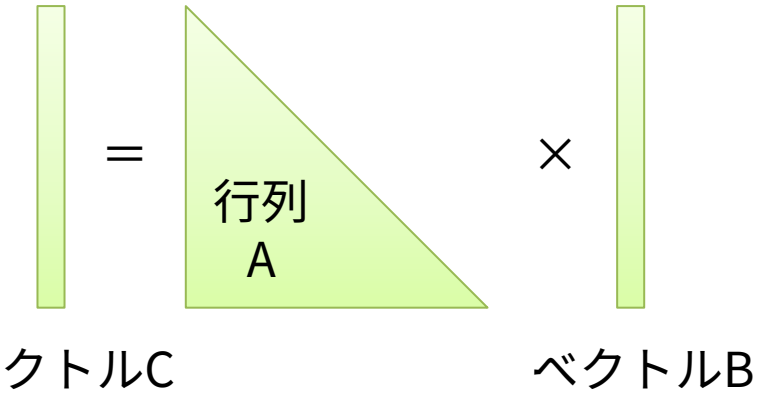
```
!$omp parallel do
do ij=1, N*N
  i = (ij-1) / N + 1
  j = mod(ij-1,N) + 1
  c(j,i) = a(j,i) + b(j,i)
end do
```

実習：行列ベクトル積、三角行列ベクトル積

◆ 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（trimatvec1）



実習手順

- ベクトル加算や行列加算と同様に、matvec1.c, matvec1.f90, trimatvec1.c, trimatvec1.f90の外側ループに指示文を追加してコンパイル・実行し、並列化の効果を確認してみよう
  - C言語版は変数のprivate化を忘れないように注意すること

実際にやってみましょう

行列ベクトル積の並列化

- 行列ベクトル積は二重ループ構造、行単位で並列に計算できることは明らか

matvec2.c

```
#pragma omp parallel for private(j)
  for(i=0; i<N; i++){          ←外側ループ：行単位の処理
    for(j=0; j<N; j++){        ←内側ループ：行内の処理
      c[i] += a[i][j] * b[j];
    }
  }
```

matvec2.f90

```
!$omp parallel do
  do i=1, N                    ←外側ループ：行単位の処理
     do j=1, N                 ←内側ループ：行内の処理
        c(i) = c(i) + a(j,i) * b(j)
     end do
  end do
```

- 内側ループは並列化できるだろうか？
  - 配列Cへの書き込みが競合するため不可能に見えるが……？

行列ベクトル積の並列化：内側ループの並列化を考える

- 何度も更新される配列cを一時変数に置き換えてから考えてみることにする

```
  for(i=0; i<N; i++){
    tmp = 0.0;
#pragma omp parallel for ?
    for(j=0; j<N; j++){
      tmp += a[i][j] * b[j];
    }
    c[i] = tmp;
  }
```

```
  do i=1, N
     tmp = 0.0
!$omp parallel do ?
     do j=1, N
        tmp = tmp + a(j,i) * b(j)
     end do
     c(i) = tmp
  end do
```

  - tmpへの書き込みが競合するため並列化できない気がする一方で、最終的な値の足しあわせ以外はスレッド並列化できそうな気もする

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・最小値の取得などの並列化が可能
  - 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
  - parallel節にもfor/do節にも適用可能

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

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

  - 配列に対するreductionは、昔はできなかったが、現在のOpenMPでは可能
    - エラーする場合は新しいコンパイラを使ってみよう
    - gcc 4.8.5で配列cをリダクションするコードをコンパイルした場合の例

```
matvec.c:28:26: エラー:  'c'  は  'reduction'  用の無効な型を持っています
 #pragma omp for reduction(+:c)
```

    - module load gcc/8.4.0 でgccを切り替えるとこのエラーは消える

リダクションを用いた行列ベクトル積の並列化

- 内側ループをreduction節で並列化

matvec4.c

```
 for(i=0; i<N; i++){
    tmp = 0.0;
#pragma omp parallel for reduction(+:tmp)
    for(j=0; j<N; j++){
      tmp += a[i][j] * b[j];
    }
    c[i] = tmp;
  }
```

matvec4.f90

```
 do i=1, N
     tmp = 0.0
!$omp parallel do reduction(+:tmp)
     do j=1, N
        tmp = tmp + a(j,i) * b(j)
     end do
     c(i) = tmp
  end do
```

- 両方のループを並列化

matvec5.c

```
#pragma omp parallel for private(tmp)
  for(i=0; i<N; i++){
    tmp = 0.0;
#pragma omp parallel for reduction(+:tmp)
    for(j=0; j<N; j++){
      tmp += a[i][j] * b[j];
    }
    c[i] = tmp;
  }
```

matvec5.f90

```
!$omp parallel do private(tmp)
  do i=1, N
     tmp = 0.0
!$omp parallel do reduction(+:tmp)
     do j=1, N
        tmp = tmp + a(j,i) * b(j)
     end do
     c(i) = tmp
  end do
```

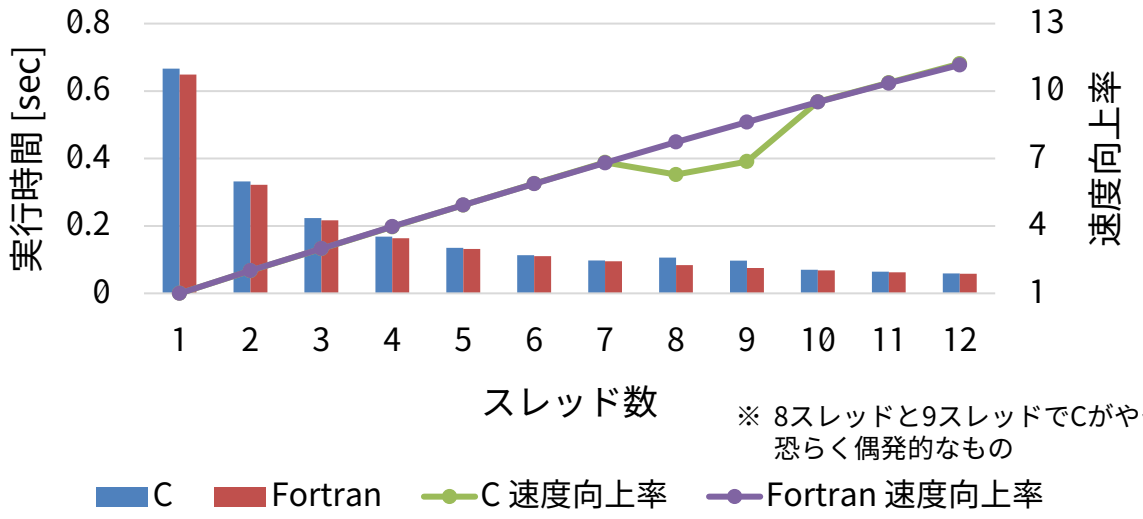
- 実行条件（環境、サイズ）によって性能の優劣（matvec4とmatvec5のどちらが高速か）は変わるかも知れない

三角行列ベクトル積の並列化と負荷の均等化

- 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
- 何故だろうか？
  - 例：N=5000

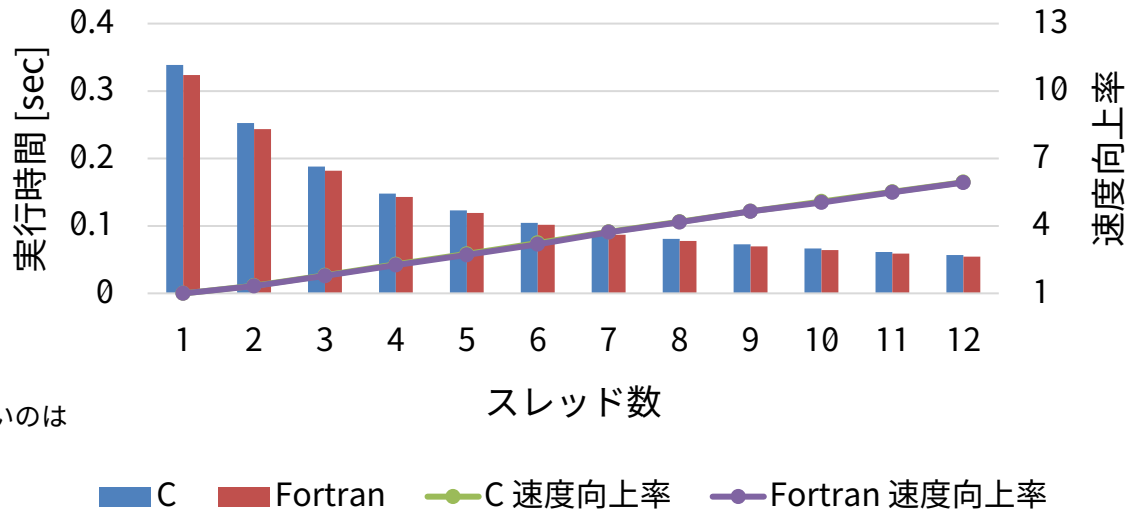
※問題サイズやコンパイルオプションにも影響は受ける

行列ベクトル積



※ 8スレッドと9スレッドでCがやや遅いのは恐らく偶発的なもの

三角行列ベクトル積



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

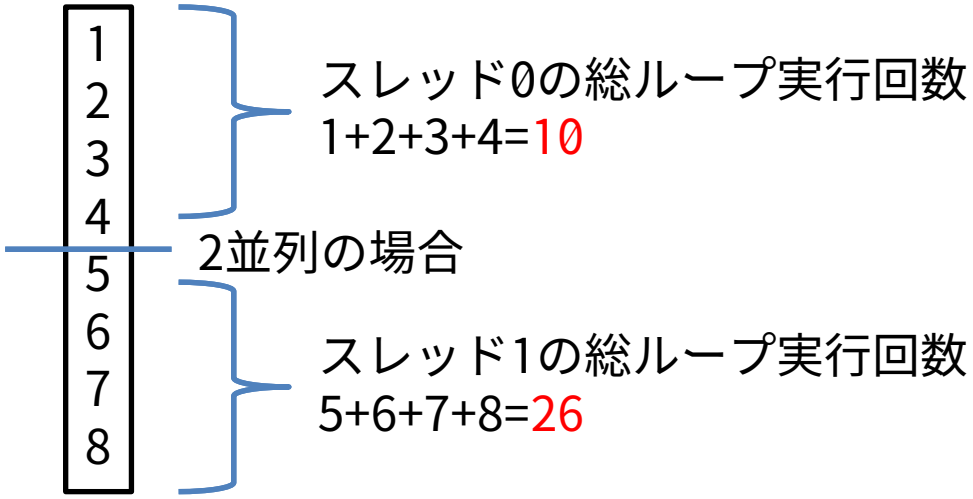
trimatvec2.c

```c
#pragma omp parallel for private(j,tmp)
  for(i=0; i<N; i++){
    tmp = 0.0;
    for(j=0; j<=i; j++){
      tmp += a[i][j] * b[j];
    }
    c[i] = tmp;
  }
```

trimatvec2.f90

```
!$omp parallel do private(tmp)
  do i=1, N
     tmp = 0.0
     do j=1, i
        tmp = tmp + a(j,i) * b(j)
     end do
     c(i) = tmp
  end do
```

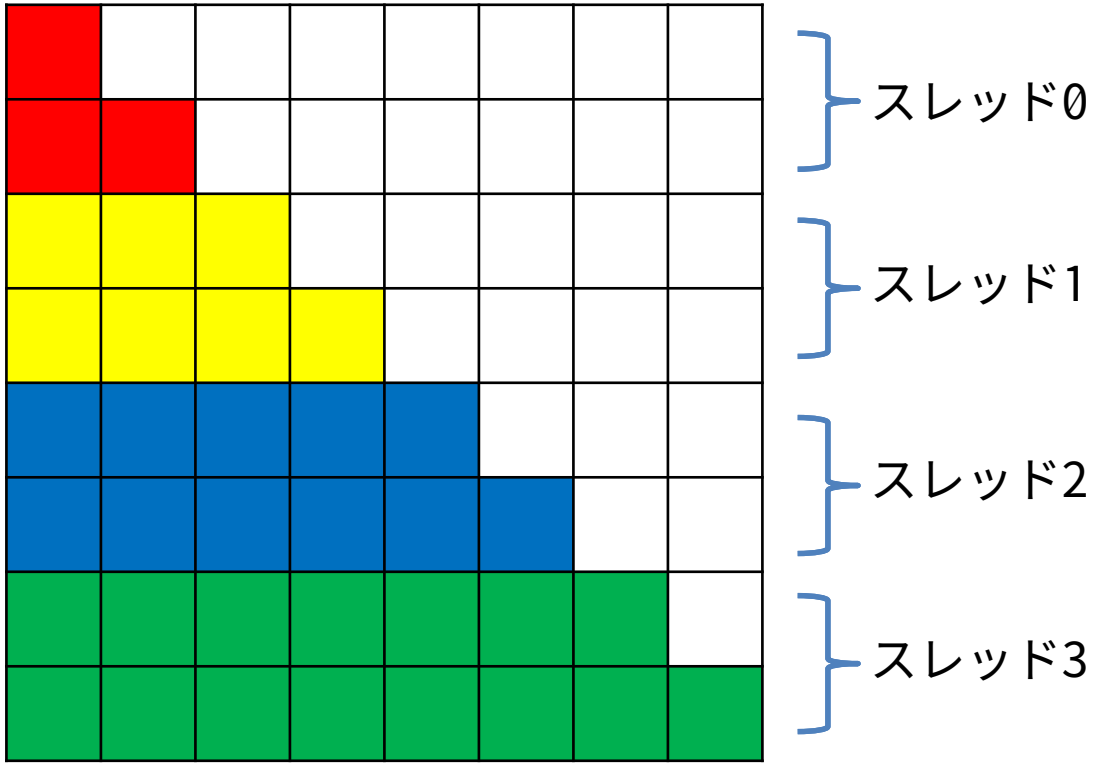
- N=8の場合、内側ループのループ長は1から8まで不均等



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
（足を引っ張られる）

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

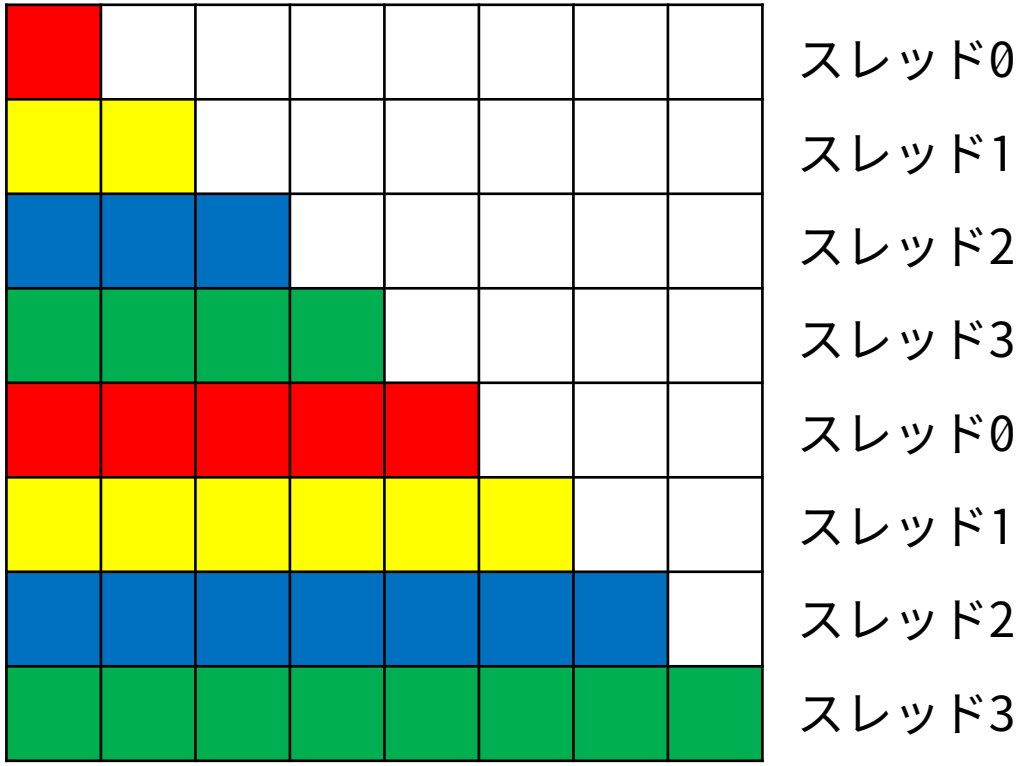



```
#pragma omp parallel for private(j,tmp) schedule(static)
  for(i=0; i<N; i++){
    tmp = 0.0;
    for(j=0; j<=i; j++){
      tmp += a[i][j] * b[j];
    }
    c[i] = tmp;
  }
```

```
!$omp parallel do private(tmp) schedule(static)
  do i=1, N
     tmp = 0.0
     do j=1, i
        tmp = tmp + a(j,i) * b(j)
     end do
     c(i) = tmp
  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



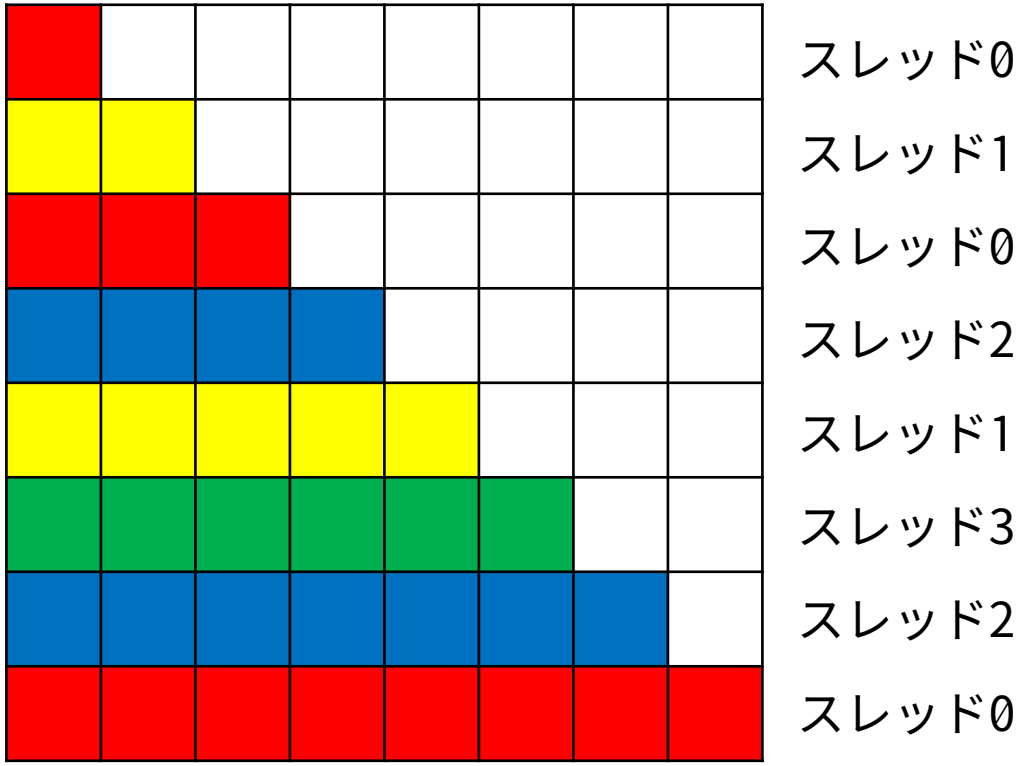

```
#pragma omp parallel for private(j,tmp) schedule(static,1)
  for(i=0; i<N; i++){
    tmp = 0.0;
    for(j=0; j<=i; j++){
      tmp += a[i][j] * b[j];
    }
    c[i] = tmp;
  }
```

```
!$omp parallel do private(tmp) schedule(static,1)
  do i=1, N
     tmp = 0.0
     do j=1, i
        tmp = tmp + a(j,i) * b(j)
     end do
     c(i) = tmp
  end do
```

動的な割り当て

※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

- 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



※あくまでイメージ、こう割り当てられるとは限らない

```
#pragma omp parallel for private(j,tmp) schedule(dynamic,1)
  for(i=0; i<N; i++){
    tmp = 0.0;
    for(j=0; j<=i; j++){
      tmp += a[i][j] * b[j];
    }
    c[i] = tmp;
  }
```

```
!$omp parallel do private(tmp) schedule(dynamic,1)
  do i=1, N
     tmp = 0.0
     do j=1, i
        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

- 明示的に同期を行うこともできる
  - barrier指示文を使う
- 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 barrier
......
```

※当然だが、並列実行範囲の中で使う

nowaitの利用例

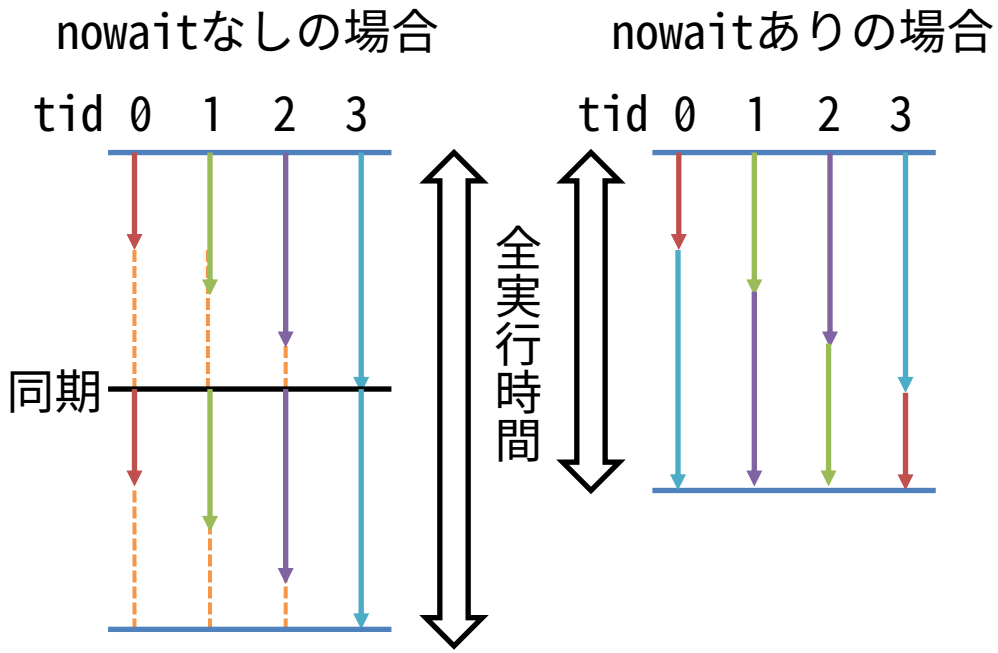
- 1つのparallel並列実行部分に複数のループ並列化が存在する場合に、途中の同期を省く
  - C/C++ではループ並列化のはじめに、Fortranではループ並列化の終わりに書く

nowait1.c

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

nowait1.f90

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



※あくまで書き方の例のため、サンプルコードのnowaitの有無では性能差は実感できない

- 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

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

single1.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;
}
```

single1.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

critical1.c

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

int main()
{
  int sum1=0, sum2=0, sum3=0;
#pragma omp parallel
  {
    sum1 += 1;
#pragma omp critical
    sum2 +=1;
#pragma omp atomic
    sum3 += 1;
  }
  printf("sum1 = %d¥n", sum1);
  printf("sum2 = %d¥n", sum2);
  printf("sum3 = %d¥n", sum3);
  return 0;
}
```

critical1.f90

```f90
program critical
  use omp_lib
  implicit none
  integer :: sum1=0, sum2=0, sum3=0

!$omp parallel
  sum1 = sum1 + 1
!$omp critical
  sum2 = sum2 + 1
!$omp end critical
!$omp atomic
  sum3 = sum3 + 1
!$omp end atomic
!$omp end parallel

  print *, "sum1 = ", sum1
  print *, "sum2 = ", sum2
  print *, "sum3 = ", sum3
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とsum3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

ordered1.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());
    }
  }
```

ordered1.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
```

※doのオプションなのでorderedのendは不要

- 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が、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）

- 同一IDのparallel1→ordered→parallel2が保証される

```
$ cat job_ordered.sh.97186.out | grep "id 0"
parallel1 id 0, executed by 0
ordered   id 0, executed by 0
parallel2 id 0, executed by 0
```

- ordered内の追い越しが起きない

```
$ cat job_ordered.sh.97186.out | grep "^o"
ordered   id 0, executed by 0
ordered   id 1, executed by 0
ordered   id 2, executed by 0
ordered   id 3, executed by 1
ordered   id 4, executed by 1
ordered   id 5, executed by 1
ordered   id 6, executed by 2
ordered   id 7, executed by 2
ordered   id 8, executed by 3
ordered   id 9, executed by 3
```

```
parallel1 id 3, executed by 1
parallel1 id 8, executed by 3
parallel1 id 6, executed by 2
parallel1 id 0, executed by 0
ordered   id 0, executed by 0
parallel2 id 0, executed by 0
parallel1 id 1, executed by 0
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
ordered   id 3, executed by 1
parallel2 id 3, executed by 1
parallel1 id 4, executed by 1
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
ordered   id 6, executed by 2
parallel2 id 6, executed by 2
parallel1 id 7, executed by 2
ordered   id 7, executed by 2
parallel2 id 7, executed by 2
ordered   id 8, executed by 3
parallel2 id 8, executed by 3
parallel1 id 9, executed by 3
ordered   id 9, executed by 3
parallel2 id 9, 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（C言語には存在しない）「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

workshare1.f90

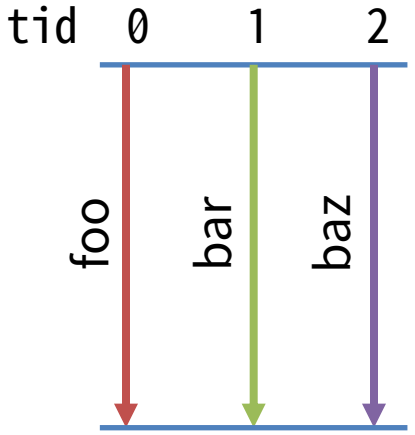
```
!$omp workshare
  c(:) = a(:) + b(:)
!$omp end workshare
```

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



その他、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

内容

- OpenMPを学ぶ前に
  - 並列計算の基礎
- OpenMPの基礎
  - OpenMPの仕様と使い方
  - 簡単なベクトル計算・行列計算の並列化
- OpenMPプログラムの最適化（高速化）
  - 一般的なOpenMPプログラムの最適化
  - スーパーコンピュータ「不老」向けのプログラム最適化
    - Type Iサブシステム向けの最適化
    - その他の一般的な環境向けの最適化
- まとめ
- 実習用サンプルソースコードの紹介

OpenMPプログラムの最適化（高速化）について

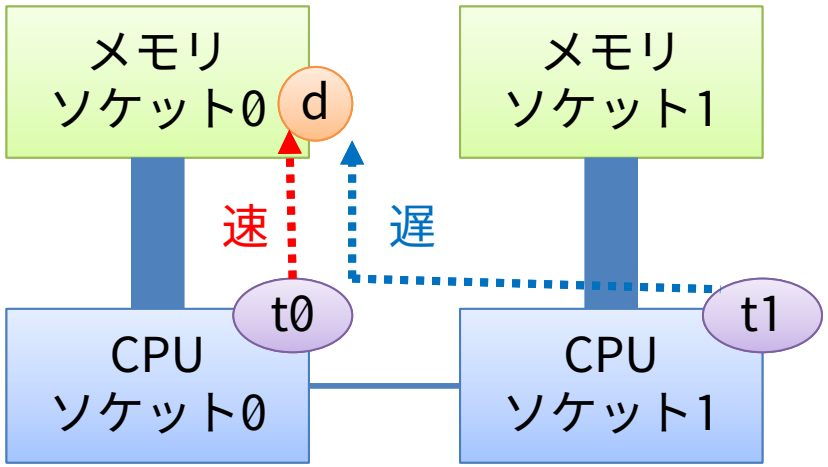
- 簡単に並列化を行えるOpenMPだが、最適化については考えられることが色々ある

- 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
  - 並列化すればするほど性能が上がるわけではない
- 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、便利ではあるが少ない方が高速
  - nowaitも適切に使う

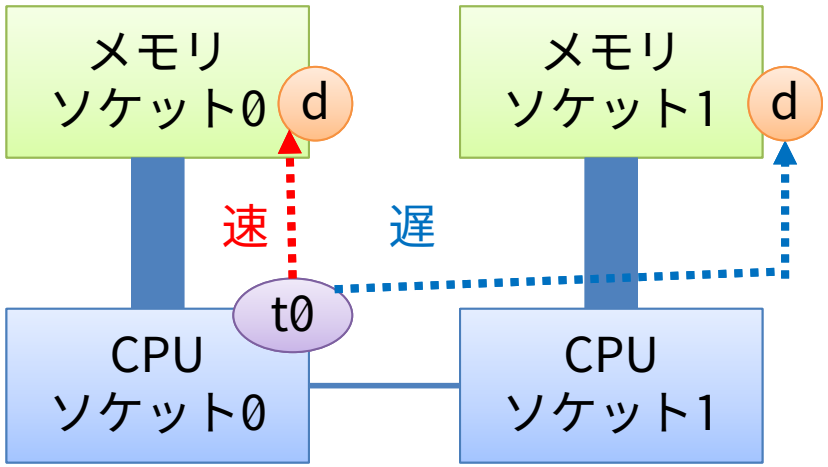
スレッドの配置が性能に影響を与える

- 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
  - 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



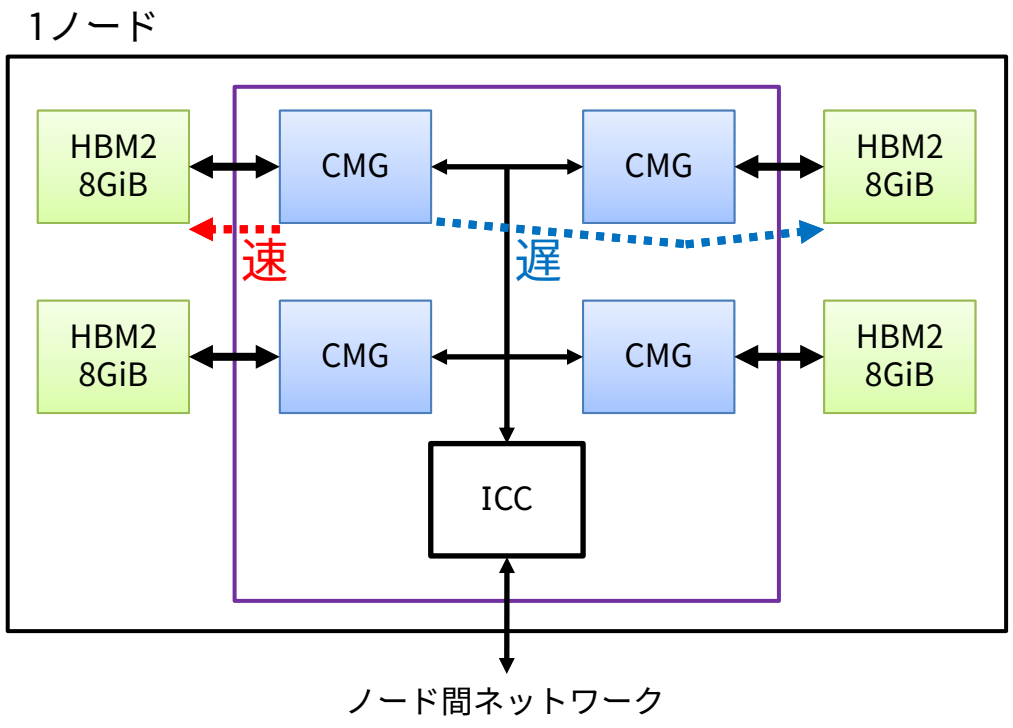
- 全てのデータがメモリソケット0上に配置されているとき、
  CPUソケット0上のスレッドからのデータアクセスと
  CPUソケット1上のスレッドからのデータアクセスには
  性能差がある

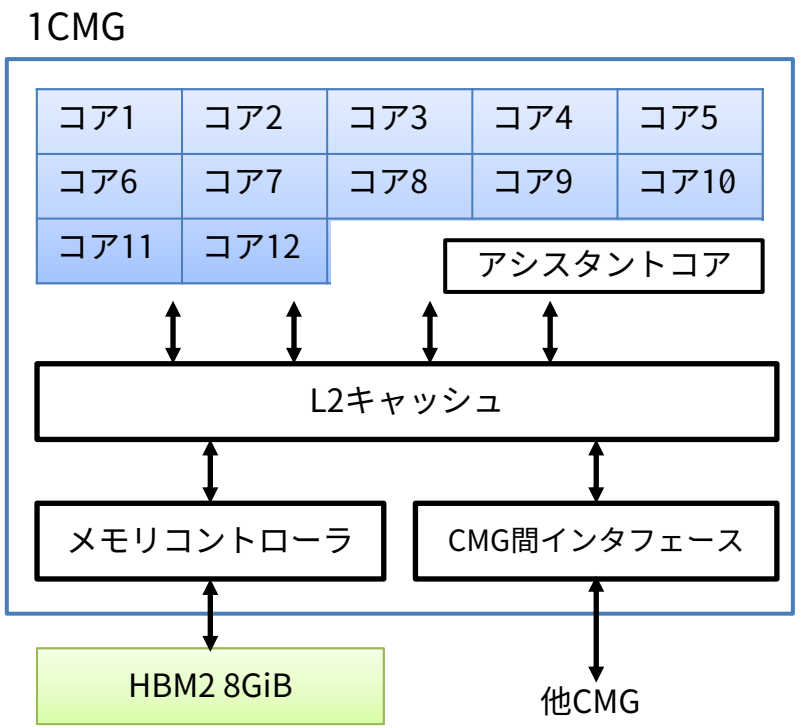


- スレッドがCPUソケット0上に配置されているとき、
  メモリソケット0上に配置されているデータへのアクセスと
  メモリソケット1上に配置されているデータへのアクセスには
  性能差がある

A64FXの構成

- Type Iサブシステム（FX1000）は1ノードあたり1CPUのシステムであるが、1CPUが4つのCMGによって構成されており、メモリは各CMGに分散して接続されている
- CMGをまたいだメモリアクセスも可能だが、あるCMG上のコアから異なるCMGに接続されたメモリにアクセスしようとすると、同一CMG上のメモリにアクセスするよりも時間がかかる





A64FXの高速なメモリの性能を有効に活用する方法

- 1：CMGをまたいだメモリアクセスが発生しにくいようなプログラムにする
  - OpenMPだけで並列化を行うならこの方法になる
  - 難点：プログラムのメモリアクセスパターンによっては対処不能

- 2：各CMGごとにプロセスを実行し、各プロセスはCMGに直結したメモリのみを使うようにする
  - 基本的にはOpenMP + MPIによるハイブリッド並列化プログラムを考えることになる
  - 難点：48コアOpenMP並列実行を諦めるため、OpenMPプログラム1つだけでCPUの性能をフルに活用することはできない

スレッドやメモリの配置の指定方法

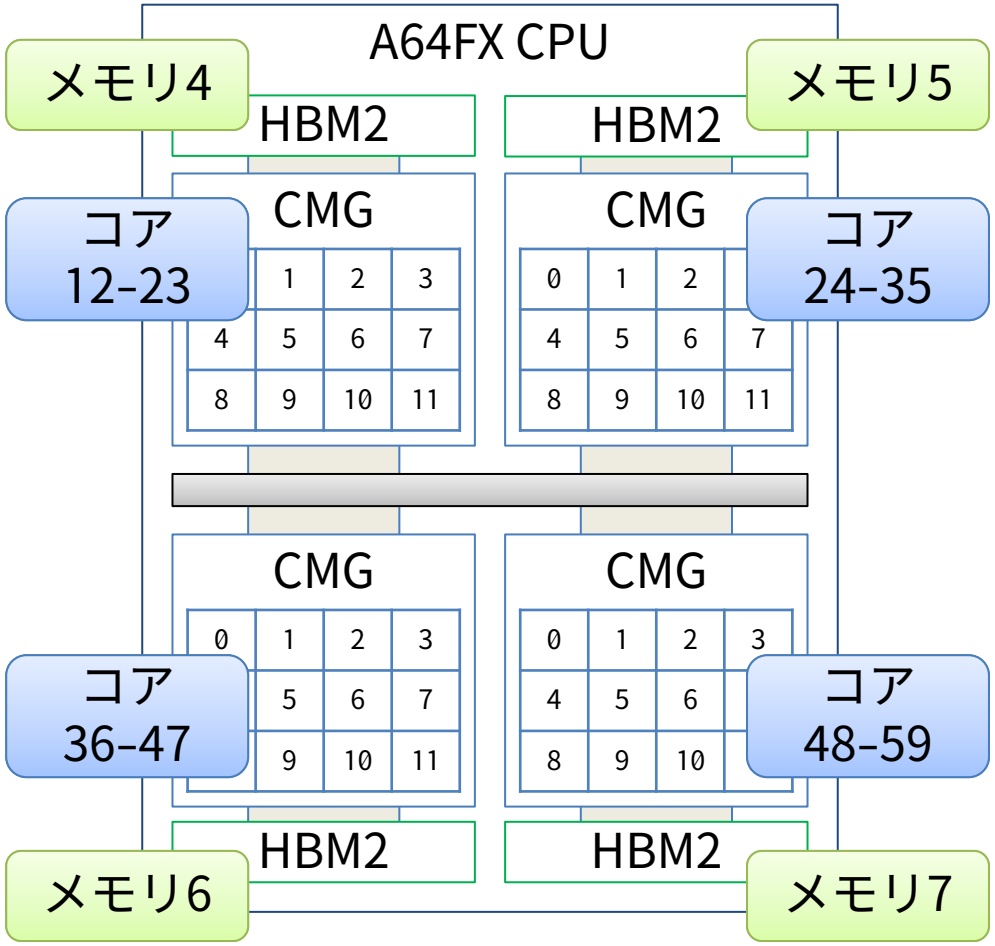
- 環境変数を指定することで配置の調整ができる、具体的な指定方法はコンパイラに依存
  - 富士通コンパイラ：FLIB_CPU_AFFINITY
  - Intelコンパイラ： KMP_AFFINITY
  - GNUコンパイラ：GOMP_CPU_AFFINITY
  - PGIコンパイラ：MP_BLIST　　など

- numactlコマンドでCPUやメモリの割り当てを細かく指定することができる
  - 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
  - Intel CPU環境でよく使うのは-Nや-l
    - -N : 対象プログラムをどのCPUソケットで実行するかを指定するオプション
    - -l : --localalloc, CPUと近いメモリに割り当てる

A64FX向けのスレッドやメモリの配置の指定方法（numactlの活用）

- 「プログラミングガイド プロセッサ編」に掲載されているとおり、numactlでコアの指定（-C）とメモリの指定（-m）が可能

CMG	コア番号	メモリ番号
CMG0	12-23	4
CMG1	24-35	5
CMG2	36-47	6
CMG3	48-59	7



- 1つのCMGだけを使ったOpenMP実行の例
  - CMG0に対応するコア番号とメモリ番号を指定

```
#PJM -L node=1
export OMP_NUM_THREADS=12
numactl -C 12-23 -m 4 ./a.out
```

ファーストタッチ最適化（First Touch, FT）

※ 1CMGしか使わない場合は
効果がありません

- 「最近アクセスしたデータ」はキャッシュに記録され、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破綻しやすいので注意

```
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
```

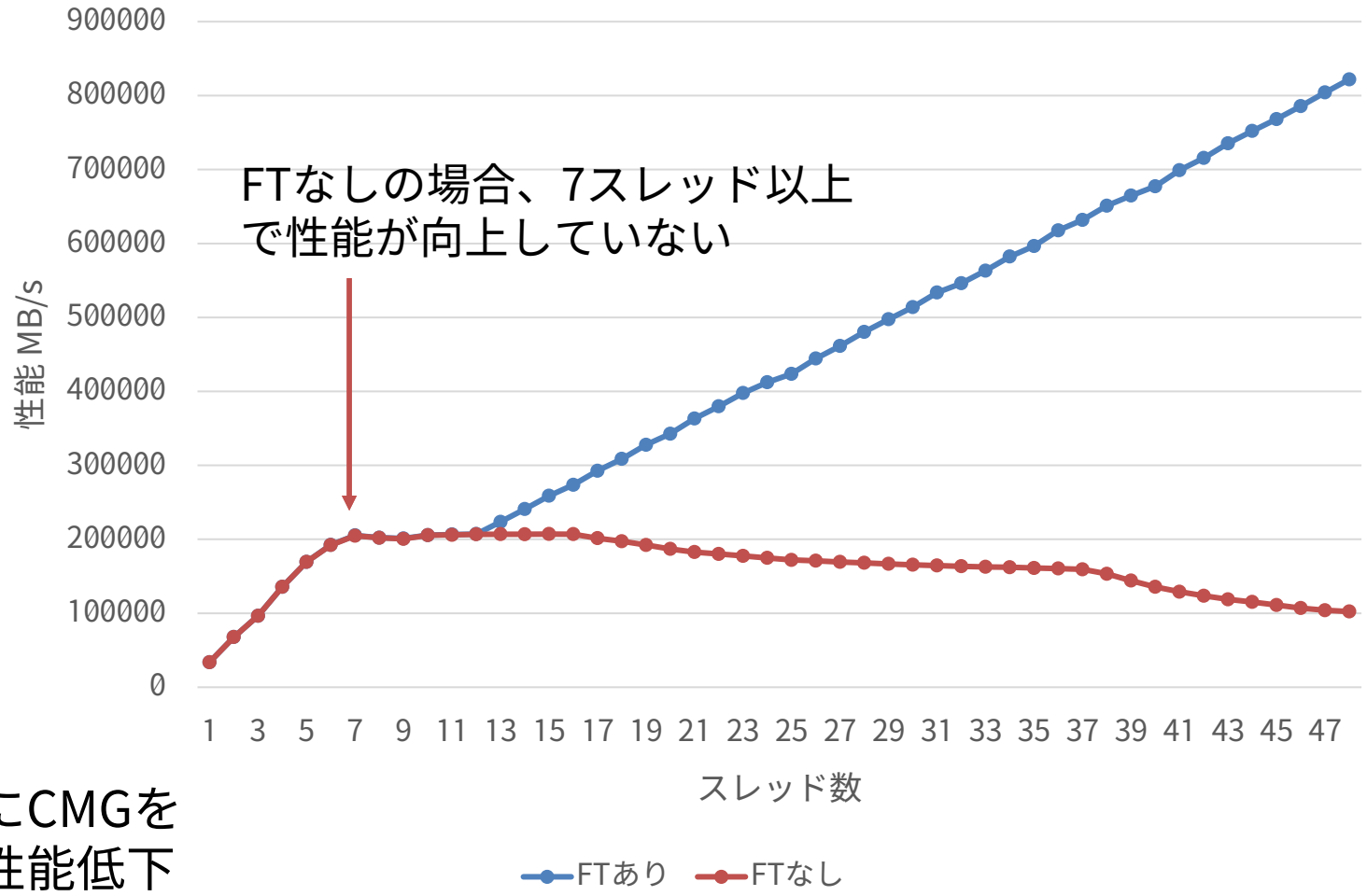
OpenMP並列化していない場合はarrayに対するキャッシュは全てマスタースレッドの近くに配置されるが、
OpenMP並列化すると各スレッドに配置される　→　並列化ループ部のメモリアクセス性能が向上する

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ベンチマークの性能を比較してみた
- STREAM Triadの結果を比較

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

- 本来はFTありのコードだが、なし版を作り比較してみた
- 結果
  - FTありの場合：各スレッドの配列へのアクセスが各CMGに限定されたため、高い性能が得られた
  - FTなしの場合：配列の全データのキャッシュが1つのCMGに配置されてしまったため、多スレッド時にCMGをまたいだメモリアクセスが頻発して性能低下



コンパイル時・リンク時・実行時の推奨オプションなど

- 富士通コンパイラ利用時に必要・考慮すべきオプション等をいくつか紹介する
- OpenMPを利用するために指定が必要なオプション：-Kopenmp
  - 実アプリで指定するべき主なオプションは -Kfast,openmp,parallel -Nfjomplib など
    - 主要な最適化が適用される、OpenMPや自動並列化が適用される、高速なOpenMP実装を使用
- -Ntrad / -Nclang
  - 推奨オプションではないが、このオプションによって富士通コンパイラの挙動（モード）が変わる。デフォルトは-Ntrad。
  - -Ntrad：伝統的な富士通コンパイラをベースとした挙動。自動並列化などが強力。
  - -Nclang：LLVM/clangベース。tradよりも新しい言語仕様への対応などが進んでいる。
- -Nfjomplib
  - -Ntrad時のみ利用可能、デフォルトで高速なハードウェアバリアが有効な富士通独自のOpenMP実装が使われる。
  - -Nclangでハードウェアバリアを使いたい場合はFLIB_BARRIER環境変数をHARDに設定する。幾つか制限があるので注意。詳細はHPCポータルにて公開している「プログラミングガイド プログラミング共通編」を参照。

その他の重要な設定

- プログラム実行時のOMP_STACKSIZE環境変数
  - 各スレッドのスタックサイズを指定する環境変数。大きな行列を扱う際などに指定が必要。
    例：export OMP_STACKSIZE=8M
- コンパイル時の-Kzfillオプションと
  プログラム実行時の環境変数XOS_MMM_L_PAGING_POLICY=demand:demand:demand
  - キャッシュやメモリの処理に関する挙動を指定するもの。STREAMベンチマークではこれらを指定しないと十分な性能が得られない。効果があるかどうかはプログラムの内容次第。

- 性能に影響のあるコンパイラオプションや実行時オプションは大量にありプログラムごとに向き不向きがある。最適なパラメタを確実に選ぶのは極めて難しいと言わざるを得ない。

プロファイラの活用について

- 高速なプログラムを作るにはプログラムの性能をしっかり解析することが重要
- OpenMPプログラム（に限った話ではないが）の解析にはプロファイラが有効
  - 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への対応は難しい
    - 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

- Type Iサブシステムでは「基本プロファイラ」「詳細プロファイラ」「CPU性能解析レポート」が利用できる
  - 概要について紹介とデモを行う
  - 詳細な使い方はHPCポータルにて公開されている「プロファイラ 使用手引書」を参照

基本プロファイラ

- サンプリングにより解析するプロファイラ
  - 1秒以下の短いプログラムは計測不可能
- 基本的な使い方手順
  - ソースコードに専用の関数呼び出しを追加して測定範囲を指定する（全範囲の測定であれば不要）
    - fipp_start / fipp_stop 関数
  - コンパイルする
    - プロファイラを有効化・無効化するオプションがあるが、デフォルトで有効
  - fippコマンドを介してプログラムを実行する
    - 例：`fipp -C -d ./outdir -Icall ./a.out`
  - プロファイル結果を出力する
    - 計算ノードではfippコマンド、ログインノードではfipppxコマンド
    - 例：`fipppx -A -Ibalance,cpupa -d ./outdir`

詳細プロファイラ

- 基本的な使い方手順
  - ソースコードに専用の関数呼び出しを追加して測定範囲を指定する
    - fapp_start / fapp_stop 関数
  - コンパイルする
    - 基本プロファイラ利用時と同様（プロファイラを有効化・無効化するオプションがあるが、デフォルトで有効）
  - fappコマンドを介してプログラムを実行する
    - 例：`fapp -C -d ./outdir -Icpupa -Hevent=statics ./a.out`
  - プロファイル結果を出力する
    - 計算ノードではfappコマンド、ログインノードではfapppxコマンド
    - 例：`fapppx -A -ttext -o out.txt -d ./outdir`

CPU性能解析レポート（1/2）

- プログラムを複数回実行して解析情報を収集、最大17回
- 基本的な使い方手順
  - ソースコードに専用の関数呼び出しを追加して測定範囲を指定する
    - fapp_start / fapp_stop 関数
  - コンパイルする
    - 基本プロファイラ利用時と同様（プロファイラを有効化・無効化するオプションがあるが、デフォルトで有効）
  - fappコマンドを介してプログラムを必要回数実行する
    - 例
      - `fapp -C -d ./rep1 -Hevent=pa1 ./a.out`
      - `fapp -C -d ./rep2 -Hevent=pa2 ./a.out`
      - ……

番号を増やしながら最大17回実行する。
bashスクリプトなどを覚えておくと便利。

```
for i in `seq 17`
do
  fapp -C -d ./rep$i -Hevent=pa$i ./a.out
done
```

CPU性能解析レポート（2/2）

- 基本的な使い方手順（つづき）
  - プロファイル結果を出力する
    - 計算ノードではfappコマンド、ログインノードではfapppxコマンド
    - 例
      - `fapppx -A -d ./rep1 -Icpupa,nompi -tcsv -o pa1.csv`
      - `fapppx -A -d ./rep2 -Icpupa,nompi -tcsv -o pa2.csv`　　番号を増やしながら最大17回実行する
      - `.....`
  - レポートを作成する
    - HPCポータルまたは/opt/FJSVxtclanga/tcsds-1.2.31/misc/cpupaにあるcpu_pa_report.xlsmと出力したcsvファイルを手元のPCの同じ場所に置き、xlsmファイルを開く
    - Excelマクロにより解析されて結果が表示される

内容

- OpenMPを学ぶ前に
  - 並列計算の基礎
- OpenMPの基礎
  - OpenMPの仕様と使い方
  - 簡単なベクトル計算・行列計算の並列化
- OpenMPプログラムの最適化（高速化）
  - 一般的なOpenMPプログラムの最適化
  - スーパーコンピュータ「不老」向けのプログラム最適化
- まとめ
- 実習用サンプルソースコードの紹介

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るので、指示文を挿入して並列化してみましょう
    - 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？
  - 手持ちのPCなどでgccやgfortranを使って実験する場合は、コンパイル時に-fopenmpを付けることでOpenMPが有効化されます

内容

- OpenMPを学ぶ前に
  - 並列計算の基礎
- OpenMPの基礎
  - OpenMPの仕様と使い方
  - 簡単なベクトル計算・行列計算の並列化
- OpenMPプログラムの最適化（高速化）
  - 一般的なOpenMPプログラムの最適化
  - スーパーコンピュータ「不老」向けのプログラム最適化
- まとめ
- 実習用サンプルソースコードの紹介

実習用サンプルソースコード：行列積

- 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
  - 行列積はコンパイラにより最適化されてしまいやすいコードであるため、最適化オプションを調整して（-O0をつけて）最適化させない方がOpenMPの効果がわかりやすい

  - 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

コードの説明

```
int main(int argc, char **argv){
  int i, j, k;
  int n;
  double **a=NULL, **b=NULL, **c=NULL;
  int out;

  out = 1;
  if(argc==1){
    n = 10;
  }else{
    n = atoi(argv[1]);
    if(argc==3){
      out = 0;
    }
  }
  printf("n = %d￥n", n);

  a = (double**)malloc(sizeof(double*)*n);
  for(i=0;i<n;i++)a[i] = (double*)malloc(sizeof(double)*n);
  b = (double**)malloc(sizeof(double*)*n);
  for(i=0;i<n;i++)b[i] = (double*)malloc(sizeof(double)*n);
  c = (double**)malloc(sizeof(double*)*n);
  for(i=0;i<n;i++)c[i] = (double*)malloc(sizeof(double)*n);
  for(i=0; i<n; i++){
    for(j=0; j<n; j++){
      a[i][j] = (double)(i+1) + (double)(j+1)*0.01;
      b[i][j] = (double)(i+1) + (double)(j+1)*0.01;
      c[i][j] = 0.0;
    }
  }
```

引数がない場合は問題サイズnを10にする

第一引数を問題サイズとして用いる
第二引数が与えられた場合は計算結果の出力をやめる（問題サイズが大きい場合に大量出力されるのを防ぐための措置）

配列の確保と値の初期化

```
  for(i=0; i<n; i++){
    for(j=0; j<n; j++){
      for(k=0; k<n; k++){
        c[i][j] += a[i][k] * b[k][j];
      }
    }
  }

  if(out==1){
    printf("result:￥n");
    for(i=0; i<n; i++){
      for(j=0; j<n; j++){
        printf(" %.2f", c[i][j]);
      }
      printf("￥n");
    }
    printf("￥n");
  }

  for(i=0; i<n; i++)free(a[i]); free(a);
  for(i=0; i<n; i++)free(b[i]); free(b);
  for(i=0; i<n; i++)free(c[i]); free(c);

  return 0;
}
```

主計算部（単純な三重ループ）
ここを並列化する

計算結果の出力

配列の解放

コードの説明

```
program main
  use omp_lib
  implicit none
  integer i, j, k, n
  double precision, allocatable :: a(:,:), b(:,:), c(:,:)
  integer iargc, nargc
  external iargc
  character*100 tmpstr
  integer out

  out = 1

  nargc=iargc()
  if(nargc.eq.0)then
     n = 10
  else
     call getarg(1,tmpstr)
     read(tmpstr,*) n
     if(nargc.gt.1)then
        out = 0
     endif
  endif
  write(*,*)"n = ",n

  allocate(a(n,n))
  allocate(b(n,n))
  allocate(c(n,n))
```

引数がない場合は問題サイズnを10にする

第一引数を問題サイズとして用いる
第二引数が与えられた場合は計算結果の出力をやめる（問題サイズが大きい場合に大量出力されるのを防ぐための措置）

配列の確保

```
  do i=1, n
     do j=1, n
        a(j,i) = dble(i) + dble(j)/100.0d0
        b(j,i) = dble(i) + dble(j)/100.0d0
        c(j,i) = 0.0d0
     enddo
  enddo
  do i=1, n
     do j=1, n
        do k=1, n
           c(j,i) = c(j,i) + a(k,i) * b(j,k)
        enddo
     enddo
  enddo
  if(out.eq.1)then
     write(*,'(A)')"result:"
     do i=1,n
        do j=1,n
           write(*,'(1H F6.2)',advance="NO")c(j,i)
        enddo
        write(*,'(A)')""
     enddo
     write(*,*)""
  endif
  deallocate(a)
  deallocate(b)
  deallocate(c)
9999 stop
end program
```

配列の値の初期化

主計算部（単純な三重ループ）
ここを並列化する

計算結果の出力

配列の解放

実習用サンプルソースコード：CG法

- 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

今回用いているCG法の計算手順

Compute $r^{(0)} = b-[A]x^{(0)}$	• 初期値、x(0)は適当な値（今回は0ベクトル）
for i= 1, 2, …	• 収束するまで繰り返す
solve $[M]z^{(i-1)} = r^{(i-1)}$	• 前処理、今回はrを対角要素で割るだけ
$\rho_{i-1} = r^{(i-1)}\ z^{(i-1)}$	• リダクション
if i=1	
$p^{(1)} = z^{(0)}$	• コピー
else	
$\beta_{i-1} = \rho_{i-1} / \rho_{i-2}$	• リダクション
$p^{(i)} = z^{(i-1)} + \beta_{i-1} z^{(i)}$	• ベクトル積和
endif	
$q^{(i)} = [A]p^{(i)}$	• 行列ベクトル積＝一番時間がかかる処理
$\alpha_i = \rho_{i-1} / p^{(i)} q^{(i)}$	• リダクション
$x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$	• ベクトル積和
$r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$	• ベクトル積和
check convergence $\|r\|$	• 収束判定（中身はリダクションと平方根）
end	

※上付き文字は反復回数に対応

- Ax=b：Aは行列、xとbはベクトル
- z,r,p,qはベクトル
- α・β・ρはスカラー（ベクトルのリダクション結果）

主要計算部の構造

- 行列やベクトルに対する単純な計算ばかりで構成されている
  - ★：行列要素同士のコピーや四則演算
  - ★：集約演算（dot product, reduction）
- Fortran版も構造は同様

すぐに収束してしまう問題設定のため、
ここに収束妨害用のコードを加えてある

```
for(iter=1; iter<=maxiter; iter++){
  printf("iter %d ", iter);
  // {z} = [Minv]{r}
  for(i=0;i<N;i++){    z[i] = dd[i]*r[i];  }  ★
  // {rho} = {r}{z}     ※対角要素分の1だけのベクトルddを用意済
  rho = 0.0;
  for(i=0;i<N;i++){    rho += r[i]*z[i];  }  ★
  // {p} = {z} if iter=1
  // beta = rho/rho1 otherwise
  if(iter==1){
    for(i=0;i<N;i++){      p[i] = z[i];    }  ★
  }else{
    beta = rho/rho1;
    for(i=0;i<N;i++){      p[i] = z[i] + beta*p[i];    }  ★
  }
  // {q} = [A]{p}
  for(i=0;i<N;i++){          ★行列ベクトル積
    q[i] = 0.0;                行ごとの計算を並列に行える
    for(j=0;j<N;j++){
      q[i] += A[i*N+j]*p[j];
    }
  }
  // alpha = rho / {p}{q}
  pq = 0.0;
  for(i=0;i<N;i++){    pq += p[i]*q[i];  }  ★
  alpha = rho / pq;
  // {x} = {x} + alpha*{p}
  // {r} = {r} - alpha*{q}
  for(i=0;i<N;i++){
    x[i] += + alpha*p[i];  ★
    r[i] += - alpha*q[i];
  }
  // check converged
  dnrm = 0.0;
  for(i=0;i<N;i++){    dnrm += r[i]*r[i];  }  ★
  resid = sqrt(dnrm/bnrm);
  if(resid <= cond){break;}
  if(iter == maxiter){break;}
  rho1 = rho;
}
```

初期データ

行列A（サイズN×N）

ベクトルx（サイズN）

全て0.0

ベクトルxx（サイズN）

ランダム

ベクトルb（サイズN）

- A, x, xxを初期化し、b = A×xxを計算しておいたうえで、cg法でAx = bを計算しxを求めるというプログラムにしてある。
- これなら前処理のない単純なCG法でも簡単に収束する。
- むしろ簡単過ぎてすぐに収束して終了してしまうため、意図的に値を破壊して収束しないようにしてある。（行列サイズNと同じ回数分だけ計算して収束しないまま終わる。）

実行例

```
A:
 9.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
 1.000000 9.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
 1.000000 1.000000 9.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
 1.000000 1.000000 1.000000 9.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
 1.000000 1.000000 1.000000 1.000000 9.000000 1.000000 1.000000 1.000000 1.000000 1.000000
 1.000000 1.000000 1.000000 1.000000 1.000000 9.000000 1.000000 1.000000 1.000000 1.000000
 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 9.000000 1.000000 1.000000 1.000000
 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 9.000000 1.000000 1.000000
 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 9.000000 1.000000
 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 9.000000
x:
 8.400000
 8.700000
 7.800000
 1.600000
 9.400000
 3.600000
 8.700000
 9.300000
 5.000000
 2.200000
b:
 131.900000
 134.300000
 127.100000
 77.500000
 139.900000
 93.500000
 134.300000
 139.100000
 104.700000
 82.300000
iter 1  1.089293e-01 1.000000e-16
iter 2  2.331098e-16 1.000000e-16
iter 3  5.450800e-18 1.000000e-16
time: 0.000010 sec , 0.000003 sec/iter
result(x):
 8.400000
 8.700000
 7.800000
 1.600000
 9.400000
 3.600000
 8.700000
 9.300000
 5.000000
 2.200000
1 8.400000 8.400000: 3.552714e-15
2 8.700000 8.700000: 5.329071e-15
3 7.800000 7.800000: -8.881784e-16
4 1.600000 1.600000: -8.881784e-16
5 9.400000 9.400000: 0.000000e+00
6 3.600000 3.600000: -1.776357e-15
7 8.700000 8.700000: 0.000000e+00
8 9.300000 9.300000: 0.000000e+00
9 5.000000 5.000000: -8.881784e-16
10 2.200000 2.200000: -1.776357e-15
```

←既知の行列A

←ベクトルX（求める答え）

←既知のベクトルb

←反復計算の履歴、反復計算終了時に所要時間も表示

←計算結果ベクトルX

←計算結果ベクトルXと最初に設定した
ベクトルXの比較

- 第一引数は問題サイズN
- 第二引数を0にすると初期データや計算結果を出力しなくなる
- 第三引数に0以外を指定すると収束しても計算を続行する（すぐに終わってしまうのを防ぐもう1つの策）