コンピュータシステムII（5）

情報基盤センター
天野　浩文

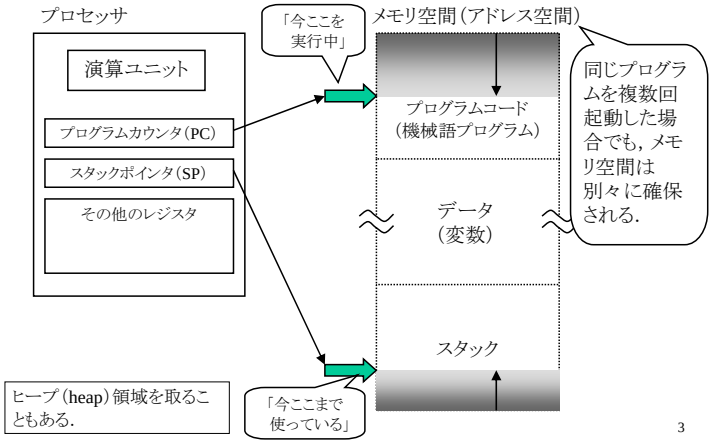
1

前回のおさらい（1）

- **プロセス**（process）
  - プログラムが計算機上で動作しているときの「実体」
    - それぞれに，メモリ中に読み込まれた機械語プログラムコード，メモリ上のデータとスタックなどの固有の資源が付随する．
- **スレッド**（thread）
  - プロセスよりもさらに小さな単位
    - 1つのプロセスは単一または複数のスレッドからなる．
      - 各スレッドには，固有のプログラムカウンタ，スタックとスタックポインタ，その他のレジスタ（の値）が付随する．
      - 同一のプロセス内のスレッドは，そのプロセスに割り当てられたメモリ空間を共有する．

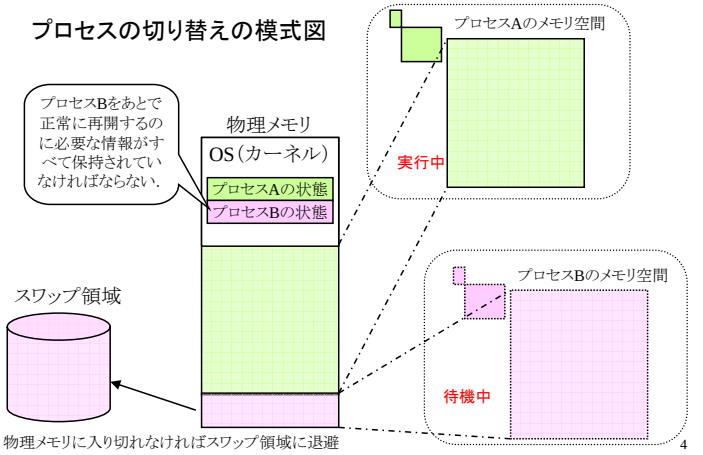
2

前回のおさらい（2）



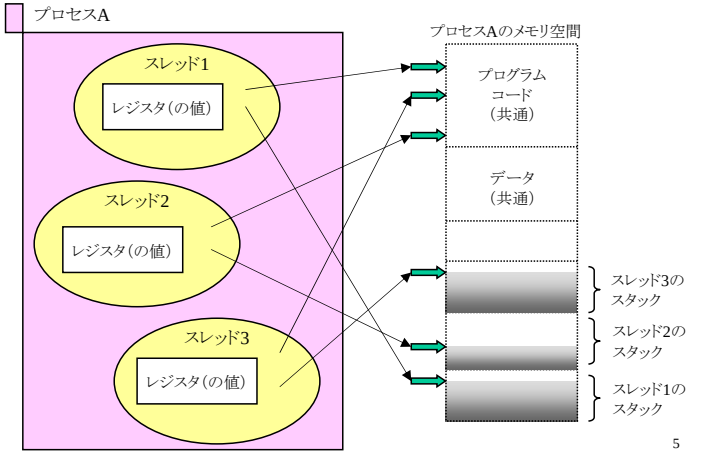
3

前回のおさらい（3）



4

前回のおさらい（4）



5

前回のおさらい（5）

- スレッドの例
  単一プロセスの内部で:
  - クライアントからのリクエストに応答するスレッドを5個起動して，同時に5つまでのリクエストに対応できるようにする.
  - i=1 から 1000 までのループを，100ずつ10個のスレッドに分割して実行する.
- ただし:
  - どんなプロセスも複数のスレッドに分割して動作できるわけではない.
    - 同一プロセス内のスレッドが相互に悪影響を及ぼさないように注意深くプログラミングしなければならない.

6

前回のおさらい（6）

- プロセスとスレッドの比較
  - スレッドはプロセスよりもさらに小さく，単一のプロセスが複数のスレッドから構成されることがある.
  - 各プロセスは固有のメモリ空間を持つが，同一プロセス内のスレッドは，そのプロセスのメモリ空間を共有する.
    - 異なるプロセスがメモリアクセスによって相互に影響を及ぼすことはないが，異なるスレッドが共有されるデータへ不適切なアクセスを行うと相互に悪影響を及ぼすことがある.
    - 実行するプロセスを切り替えるのには，スレッドを切り替えるよりも時間がかかる.

7

前回のおさらい（7）

- プロセス・スレッドの並列処理・分散処理との親和性
  - 共有メモリ型並列計算機では
    - 同一プロセス内の複数のスレッドを別々のプロセッサ上で同時に実行可能
    - 並列処理用プログラムを設計する際に，各プロセッサに割り当てる処理の単位をスレッドにすると効率的に実行できることがある.
  - 分散メモリ型並列計算機・クラスタシステム・広域分散システムには共有メモリがない.
    - 単一のプロセスを複数のスレッドに分割しても，それらのスレッドを別のプロセッサ上で動作させることは難しい.
    - このようなシステムで並列処理・分散処理用プログラムを設計する際には，各プロセッサに割り当てる処理の単位をプロセスとせざるを得ない.

8

並列プログラム作成法

3.1.2節以降の議論に進む前に，並列プログラムの書き方について考える．

9

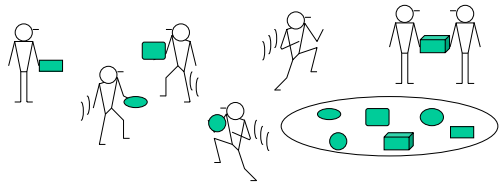
並列処理を行うには

- 独立した多数のプログラムを同時に動作させる場合を除き，多数の演算ノードを同時に用いて，何か一つの仕事をしようとするときに考えなければならないこと
  - その仕事を並列に行うにはどのようにやればよいか
    - 何と何を並列に行うのか
    - 並列に行っている作業の間で，通信や同期を行う必要はあるのか．あるとすれば，どのようなタイミングで行うのか
  - やり方がわかったとして，それをどのようにプログラムにすればよいか
    - 何を使って，どんなふうに書けばよいのか

10

並列処理のやり方（1）

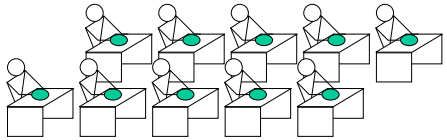
- コントロール並列プログラミング
  - 同時に実行できる多数の演算を，多数のコンピュータに分配して同時に処理させる
  - 「本当に同時に実行できる演算」の種類はあまりたくさんないことが多い



11

並列処理のやり方（2）

- データ並列プログラミング
  - 大量のデータを多数のコンピュータに分配して，それらに同じ演算を同時に適用する
  - データの量が増えれば増えるほど，同時に処理できる演算は増える



12

並列処理の記述法

- 並列処理のやり方がわかったとして，プログラムを書くためにどうしたらよいか．
  - ふつうの人は，並列プログラミングよりも逐次型プログラミングのほうが考えやすい．
  - 逐次型プログラミングよりも先に並列プログラミングを学ぶ人はほとんどいない．
- 並列プログラムの書き方は，多かれ少なかれ，逐次型プログラミングを拡張したものにならざるを得ない．

13

並列に実行できるプログラムを得る手段の分類

- 並列に実行できる機械語コードを得る方法
  - **逐次型プログラムの自動並列化**
    - 逐次型プログラミング言語で記述されたプログラムを，コンパイラの力で（自動的に）並列化する．
  - **並列処理記述言語**
    - 逐次型プログラミング言語を並列処理記述用に機能拡張した言語で書かれたプログラムを，コンパイラが並列化する．
  - **逐次型プログラムの書き換え**
    - 逐次型プログラミング言語で記述されたプログラムを，（人間が）並列処理用に書き換え，それに従ってコンパイラがプログラムを並列化する．

14

逐次型プログラムの自動並列化

- 逐次型プログラムの自動並列化の利点
  - この方法では，プログラマがコントロール並列やデータ並列を意識する必要すらない．
  - 既存のプログラムがそのまま並列化できるので，（このような技術が本当に完成すれば）理想的
  - 並列計算機のアーキテクチャやOSに依存しないプログラム開発が可能

15

自動並列化の問題点

- 現在のコンパイラ技術では，自動的に並列化できる部分は限られている．
  - 逐次型プログラムを出発点としているのに，並列化したことによって実行結果が変わってしまうようでは元も子もない．
  - 「この部分は並列化しても正しい結果となる」ということが，機械的に判断できないことも多い．
    - このようなコンパイル技術の開発は，現在も重要な研究開発課題となっている．

16

並列処理記述言語

- 並列処理記述言語
  - 既存の逐次型言語を拡張して，並列処理の記述に適した言語を作る．
    例：HPF（High-Performance FORTRAN）
  - 全く新たな言語を作ることもあるが，あまり普及した例はない．
- 長所
  - ターゲットとなる並列計算機のアーキテクチャやOSの機能を活用した並列処理が行いやすい．
- 短所
  - 並列処理することを念頭においてプログラムを書かなければならない（逐次型プログラムよりは難しい）
  - 同じ言語を多数の並列計算機で利用できるようにするのは大変．

17

逐次型プログラムの書き換え

- 逐次処理用に書かれたプログラムを改造して，並列処理用プログラムにする．
  - プロセス間通信や同期のための特殊な文を書き入れる．
    例：MPI（Message Passing Interface）
    - 別途用意されたライブラリの関数を呼び出すことが多い．
    - 従来のコンパイラでコンパイル可能（ただし，もはや逐次型プログラムとしては動作しない）
    - 改造されたプログラムは，そのコピーを複数の演算ノードに配布して同時に実行できるようになる．
  - 並列化可能な（＝コンパイラに並列化して欲しい）部分を示す特殊な構文で，コンパイラに対する指示を与える．
    例：OpenMP
    - これもコンパイラによる並列化であるが，完全な自動並列化に比べると，並列化できる部分が多くなる可能性が高い．
    - 書き加えた部分が**コメント文**のみであれば，逐次型プログラムとしてもそのまま使える．

この分類は教科書p.114では取り上げられていないが，非常に重要である．

18

並列処理記述法の歴史的経緯（1）

- 初期の並列計算機では
  - 試作機として実験的に作られたものが多かった．
  - 自動並列化の技術もほとんどなかった．
  - 機種ごとに新たな並列処理記述言語が提案されることも多かった．
    - 作られたプログラムは，**機種が変わると動作しない**．
- 最近の並列計算機では
  - 多数のメーカがさまざまな機種を発表しており，いずれも実際の大規模科学技術計算に用いられることが多い．
    - 一度作られたプログラムは粗末にせず，いろいろな計算機で利用できるようにするべき．
  - しかし，自動並列化の技術は未完成のまま．
  - 性能は追求したいが，特定の機種でしか利用できない特殊な記述法ではいけない！

19

並列処理記述法の歴史的経緯（2）

- 同じ並列プログラムがどの並列計算機でも動作可能にするために
  - 各計算機メーカが参加する業界団体のようなところで，**国際標準**となる規格を作成する．
    - HPF
    - MPI
    - OpenMP
- その後…
  - 残念ながら，HPF はあまり普及しなかった．
  - 現在は，MPI と OpenMP が**事実上の業界標準**（de facto standard）となっている．
  - 逐次型言語の自動並列化技術は，現在もさかんに研究開発が進められている．
    - 人間による並列化には性能面で及ばないことが多いが，それでも各社の製品に搭載されている．

20

MPI

後述する OpenMP と並び，実用的な並列プログラムを書く方法の『業界標準』である．

21

MPI

- MPI（Message Passing Interface）
  - 並列に動作するプロセス間の通信や同期に必要な関数（手続き）をまとめたライブラリの仕様（規格）
    - 新しい言語ではない．
    - プログラマは，逐次型プログラムの中に，これらの関数の呼び出しを適切に埋め込む．
  - 国際的な業界団体で，各関数の呼び出し方と動作内容を規定した国際規格が定められている．
    http://www.mpi-forum.org/
    - 各社がこれに対応したライブラリを開発・販売している．
    - フリー版もある．
  - 現在のところ，C/C++とFortranで使うことができる．
    - それぞれの**言語の文法はまったく変更されていない**ので，既存のコンパイラでコンパイルできる．
    - ただし，コンパイルして生成される機械語プログラムは並列実行用のものになる．

22

MPIプログラムの概観（Cの場合）

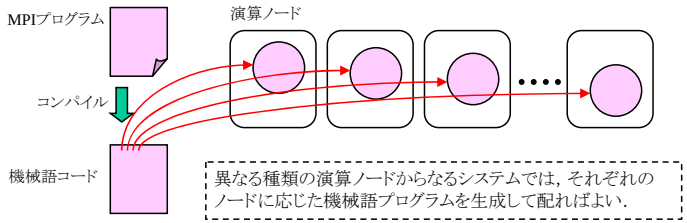
```
#include <stdio.h>
#include "mpi.h"
main(int argc, char* argv[]) {
  int my_rank;
  int p;
  ...

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  ...

  MPI_Finalize();
}
```

並列に実行される部分

23

MPIプログラムの実行形態

- SPMD（single program, multiple data）実行
  - 同じひとつのプログラムから作られた「分身」を，異なるデータに対して並列に実行する．
    - 演算ノードが同じハードウェア構成を持つ場合には，まったく同じ機械語プログラムのコピーを配ればよい．



24

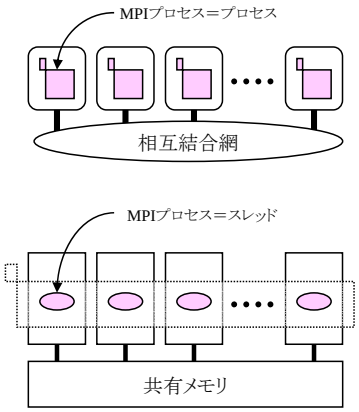
MPIプログラムにおける並列実行の主体

- MPIプロセス
  - 固有の機械語コードを実行する自律的な「プロセス」
    - OSでいうプロセスと同じものでなくともよい.
      - MPI規格では単に「プロセス」と呼んでいるが, この講義では混同を避けるために, あえて**MPIプロセス**と呼称する.
    - (前ページの絵ではMPIプロセスと演算ノード数が等しかったが)演算ノードよりも多いMPIプロセスを起動してもよい.
      - その場合には, いくつかのノードでは複数のMPIプロセスが動作する.
  - 各MPIプロセスには, その並列プログラムの中で固有の番号が付けられ, この番号によって互いを区別できる.
    - MPIでは, これを**MPIプロセスのランク**(rank)と呼ぶ.
  - 他のMPIプロセスのメモリ空間に直接アクセスすることはない.
    - 分散メモリ型並列計算機の上で動かすのが容易.
      - ただし, 共有メモリ型並列計算機の上で動かしてもよい.
    - データの交換が必要な場合は, MPIプロセス間のメッセージ通信を使用して記述する.

25

MPIプログラムの実装

- 実際の並列計算機上で動作させるには
  - 分散メモリ型並列計算機のとき
    - MPIプロセスと1対1に対応する通常のプロセスを各ノードに配置するのが普通.
    - MPIプロセス間のメッセージ通信は, そのままプロセス間通信に置き換える.
  - 共有メモリ型並列計算機のとき
    - MPIプロセスと1対1に対応するスレッドを各ノードに配置してもよい.
    - その場合は, MPIプロセス間のメッセージ通信は, 共有メモリを介したデータ交換でよい.



26

その他の基本的な約束事(Cの場合)

- MPIプログラムは, ヘッダファイル **mpi.h** をインクルードする.
  - MPI固有の定数・変数の宣言や, MPI関数のプロトタイプ宣言などが含まれている.
- 関数呼び出しの順序
  - **MPI_Init()** は, 並列実行環境を整えるため, 他のすべてのMPI関数の呼び出しよりも前に書かなければならない.
  - **MPI_Finalize()** は, 並列実行環境を解放するため, 他のすべてのMPI関数の呼び出しよりも後に書かなければならない.
- 同じプログラムから生成されて同時に動いているMPIプロセスのグループには, **MPI_COMM_WORLD**という名前が付けられている.
- MPIプロセスの総数とランク
  - 各MPIプロセスが同じプログラムから生成されて同時に動いているMPIプロセスの総数を知るためには, 関数**MPI_Comm_size()** を使用する.
  - 各MPIプロセスが自分のランクを知るには, 関数**MPI_Comm_rank()** を使用する.

27

メッセージ通信(1)

- 通信の基本
  - 1対1通信
- メッセージの送信:**MPI_Send()**
  - 呼び出しには以下のような引数を取る.
    - 送るべきメッセージ
    - メッセージに含まれるデータの個数
    - メッセージに含まれるデータの型
    - 宛先のMPIプロセスのランク
    - 複数のメッセージを送る際に受信側がそれらを区別できるようにするためのタグ
    - 通信に参加するMPIプロセスのグループ

28

メッセージ通信（2）

- メッセージの受信:**MPI_Recv()**
  - 引数
    - 受信したメッセージを受け付けるための変数
    - メッセージに含まれるデータの個数
      - 多めに指定してもよい.
    - メッセージに含まれるデータの型
    - 送信元のMPIプロセスのランク
    - 複数のメッセージが来るときにそれらを区別するためのタグ
    - 通信に参加するMPIプロセスのグループ
    - 実際に受信されたメッセージのステータス情報

29

MPIプログラムの例

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char* argv[]) {
  ...
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &num_proc);
  if (my_rank != 0) {
    sprintf(message, "Hello from %d¥n", my_rank);
    MPI_Send(message, strlen(message)+1, MPI_CHAR, 0,
             tag, MPI_COMM_WORLD);
  }
  else {
    for (source=1; source<num_proc; source++) {
      MPI_Recv(message, 100, MPI_CHAR, source, tag,
               MPI_COMM_WORLD, &status);
      printf("%s¥n", message);
  }
  MPI_Finalize();
}
```

ランク0以外のプロセスから送られたメッセージを，ランク0のプロセスが画面に表示する.

その他のメッセージ通信（1）

- 1対 *N* 通信の例:**MPI_Bcast()**
  - 全MPIプロセスが同じ関数を呼び出す.
    - このような関数は，集団型（collective）通信関数と呼ばれる.
    - **MPI_Bcast()** を含め，集団型通信関数は，そのMPIプログラムの全MPIプロセスが処理の終了を待ち合わせる同期ポイントとなる.
  - 引数
    - 送受信されるメッセージ
    - メッセージに含まれるデータの個数
    - メッセージに含まれるデータの型
    - 送信元のMPIプロセス（**root**）のランク
    - 通信に参加するMPIプロセスのグループ
  - 動作
    - **root** のメッセージが，他のすべてのMPIプロセスに送られる.

31

その他のメッセージ通信（2）

- *N* 対1通信の例:**MPI_Reduce()**
  - これも集団型通信関数の一種で，**リダクション**（reduction）と呼ばれる.
    - 全MPIプロセスが持っているデータをどれか一つのMPIプロセスに集めてくると同時に，引数で指定された演算を行って1つの値に集約する.
  - 引数
    - 各MPIプロセスから集めてくるべきデータ
    - **root** プロセスで集約結果を受け付ける変数
    - 各MPIプロセスから集めてくるデータの型
    - 集約に用いる演算子（種類と名前は **mpi.h** で定義されている）

    など.

ここまでで紹介した以外にも，MPIには多数の通信関数が用意されている.

32

OpenMP

MPI と並ぶもうひとつの『業界標準』である. ただし, MPIとは異なり, 分散メモリ型並列計算機上で効率的に実行することはできない.

33

OpenMP（1）

- OpenMPが登場した背景
  - 分散メモリ型並列計算機用のプログラムを人が書くのはかなり難しい.
    - 他の演算ノードのデータを変数として直接アクセスできないため, データ交換はすべてメッセージでやりとりする.
      - メッセージの送信元・宛先の制御
      - 送信メッセージの型や数と受信側の準備の整合性の確保
      - 送受信のタイミングの制御

      などなど.
    - 逐次型プログラムには存在しないメッセージ通信を書き加えると, 元の逐次型プログラムとはかなり異なる動作をするようになる.
  - 共有メモリ上のデータを変数として直接アクセスできれば, メッセージ通信は不要になるので, プログラムを書くのはもう少し簡単になる.
  - そうかと言って, 逐次型プログラムを自動並列化するのは, 共有メモリ型でもまだまだ難しい.

34

OpenMP（2）

- そこで...
  - 分散メモリ型並列計算機上で実行することはひとまずあきらめて, 共有メモリ型並列計算機を対象とする.
    - **スレッドによる並列処理**だけを考える.
      - このために, 並列性の抽出法がある程度限定されることになっても, それはしかたがないものとあきらめる.
  - 並列化すべき箇所は, プログラマが明示的に指定する.
    - ただし, 書き加える部分はできるだけ少なくてすむように.

⇩

- 逐次型プログラムの中で, 通常のコンパイラが無視するような箇所に, 並列化のための指示文を書いておくことにしよう.
- 特定の言語の文法を拡張するというよりも, いろいろな言語で共通に使えるような, コンパイラへの並列化指示の与え方を中心に考えよう.

35

OpenMP（3）

- OpenMP
  - 一種の「プログラミングスタイル」とでも呼ぶべきもの
  - その規格では, 以下のようなことを定めている.
    - コンパイラに並列化の箇所および方法を指示するための**特殊なコメント文・指示文の文法**
    - コンパイルされたプログラムが実行されるときの環境を決定したりその情報を取得したりするしくみ
    - 並列実行を補助するための特殊なサブルーチンや関数
  - 国際的な業界団体 OpenMP Architechture Review Board で上記の3つに関する国際規格が定められている.
    `http://www.openmp.org/`

36

OpenMP（4）

- OpenMPで並列化できるプログラミング言語
  - 現在のところ，C/C++とFortran用のOpenMP規格が定められている．
    - それらの**言語の文法はほとんど変更されていない**が，与えられた指示に従って並列化できるように**コンパイラを改造する必要がある**．
    - 並列実行補助用の特殊な関数やサブルーチンを使っていなければ，並列化の機能を持たないコンパイラでも，逐次型プログラムとしてコンパイルできる．

37

OpenMPプログラムの一例（Cの場合）

```
#include <stdio.h>
#include <omp.h>
main(){
  int i;
  double z[100], x[100], a, b;

  /* x, a, b の初期化など */

  #pragma omp parallel for

  for (i = 0; i < 100; i++) {
    z[i] = a*x[i] + b;
  }

  /* z の出力など */
}
```

「この直後の **for** 文を複数のスレッドで並列に実行せよ」という指示

100回の繰り返しを，環境変数 **OMP_NUM_THREADS** で指定された数のスレッドで分担して並列実行する．

- Cでは，**pragma**文（コンパイラ指示文）で直後の文の並列化指示を書く．
- これらの指示を「解釈」できないコンパイラは，それを無視する．

38

OpenMPプログラムの一例（Fortranの場合）

```
program example
integer i
double precision z(100), x(100), a, b;

! x, a, b の初期化など

!$omp parallel do

do i = 1, 100
    z(i) = a*x(i) + b;
end do

end program example
```

「この直後の **do** 文を複数のスレッドで並列に実行せよ」という指示

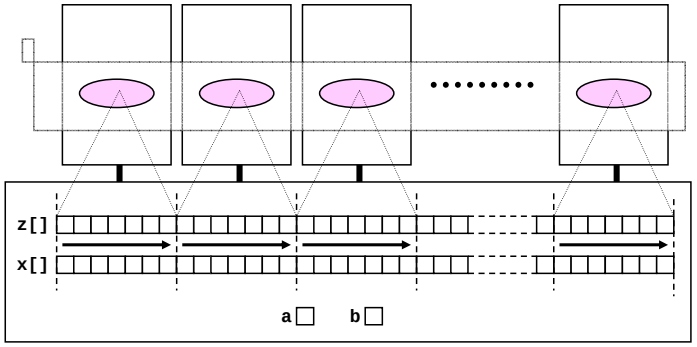
100回の繰り返しを，環境変数 **OMP_NUM_THREADS** で指定された数のスレッドで分担して並列実行する．

- Fortranでは，コメント文を使って並列化を指定する．
  - **!$omp parallel**と**!$omp parallel end**で範囲を指定することもある．
- これらの指示を「解釈」できないコンパイラは，それを無視する．

39

単純な並列化の例

- 前ページのプログラムをOpenMP対応のコンパイラでコンパイルすると，100個の要素を持つ配列を複数のスレッドで分担してループを処理するような並列プログラムが生成される．



MPIの場合との違い

- MPIでループを分割して並列化する場合には...
  - 各MPIプロセスの担当する繰り返しのどこからどこまでを担当するのかをプログラム中で計算しなければならない.
    - たとえば, 逐次型のときの配列の大きさを変更しないならば

```
for (i = my_rank*array_size/num_proc;
     i < (my_rank+1)*array_size/num_proc; i++) ...
```

    - 繰り返し回数(配列のサイズ)がMPIプロセス数で割り切れない場合の処置などもプログラマの責任

41

その他の基本的な約束事

- OpenMPで並列化できるためには, ループの処理に入る直前で, ループ全体の繰り返し回数が計算できなければならない.
  - 終了値が繰り返しの途中で変更されるようなループは並列化できない.
  - 終了値に達する前に強制終了して次の処理に進むようなループも並列化できない.
- 実行中のスレッドの総数を取得する関数など, OpenMP固有の関数を用いる場合には, ヘッダファイル **omp.h** をインクルードする.

42

ループの並列化の限界

- ループ内で実行される文の種類によっては, 並列化すると正しい結果(逐次型プログラムと同じ実行結果)が得られないこともある.
  - プログラムを変更すれば正しく並列化可能なループの例

```
for (i = 0; i < 99; i++) {
  a[i] = a[i] + a[i+1];
}
```

  - 正しく並列化できないループの例

```
for (i = 1; i < 100; i++) {
  a[i] = a[i] + a[i-1];
}
```

1つ前の繰り返しで書き換えられた **a[i-1]** の値を使って **a[i]** を書き換えるので, この順序を守らない限り, 正しい結果にならない.

43

並列化できるようにするためのプログラム変更の例

- 前ページの例では, あらかじめ **a[]** を別な配列 **b[]** にコピーして, それを使って計算するようにすればよい.

```
for (i = 0; i < 99; i++) {
  a[i] = a[i] + a[i+1];
}
```

代入文で書き換えられる前の **a[i+1]** を使っているので, コピーした値を使うように書き換えても結果は同じ.

```
for (i = 0; i < 100; i++) {
  b[i] = a[i];
}
for (i = 0; i < 99; i++) {
  a[i] = a[i] + b[i+1];
}
```

```
#pragma omp parallel for
for (i = 0; i < 100; i++) {
  b[i] = a[i];
}
#pragma omp parallel for
for (i = 0; i < 99; i++) {
  a[i] = a[i] + b[i+1];
}
```

入れ子ループの場合

- 入れ子になったループを並列化する場合，外側のループを分割してスレッドに割り振るのが効率的

```
for (i = 0; i < 99; i++) {
  y[i] = 0;
  for (j = 0; j < 99; j++) {
    y[i] = y[i] + a[i][j]*x[j];
  }
}
```

```
#pragma omp parallel for private(j)
for (i = 0; i < 99; i++) {
  y[i] = 0;
  for (j = 0; j < 99; j++) {
    y[i] = y[i] + a[i][j]*x[j];
  }
}
```

ただし，内側のループのカウンタ **j** がスレッドごとに別の変数になるようにしておかないと，正しい結果が得られない．

45

まとめ

- コントロール並列プログラミングとデータ並列プログラミング
- 並列に実行できる機械語コードを得る方法
  - 逐次型プログラムの自動並列化
  - 並列処理記述言語
  - 逐次型プログラムの書き換え
    - MPI
    - OpenMP

46