高性能コンピューティング特論 講義メモ(6)

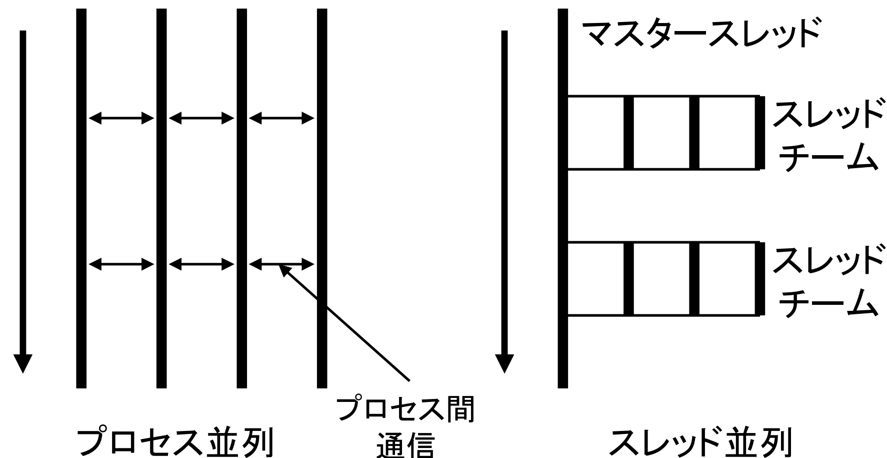
「並列プログラミング」

高橋大介
daisuke@cs.tsukuba.ac.jp
筑波大学大学院システム情報工学研究科
計算科学研究センター

講義内容

- OpenMP並列プログラミング
- MPI並列プログラミング

プロセス並列とスレッド並列



OpenMPとは

- OpenMPは，マルチスレッド並列プログラミングのためのAPI（Application Programming Interface）である．
- OpenMPは1997年に発表された業界標準規格であり，多くのハードウェアおよびソフトウェア・ベンダーが参加する非営利団体「OpenMP Architecture Review Board」によって管理されている．
- 最新のOpenMPのリリースは，2008年5月に発表されたOpenMP 3.0である．

OpenMPの特徴

- OpenMP APIはユーザ指示の並列化のみを対象にしている.
  - 指示文を挿入することにより並列化を行う.
- 指示文はOpenMPをサポートしないコンパイラでは，単にコメント行として無視される.
  - その場合，逐次計算プログラムとしての動作が保証される.
- OpenMPでは，逐次計算プログラムに対して指示文を挿入するという作業により，段階的に並列化を行うことが可能になる.

OpenMPを用いた並列プログラムの構成

```
#include <stdio.h>
int main(void)
{
…
#pragma omp parallel
   {
…      並列化される部分
   }
}
```


指示文の形式

- C/C++のOpenMP指示文は，pragmaプリプロセッサ指示文で指定する.

#pragma omp *directive-name* *[clause[[,] clause]*

- それぞれの指示文は，#pragma ompで始まる.
- 例
  - #pragma omp parallel { }
  - #pragma omp for

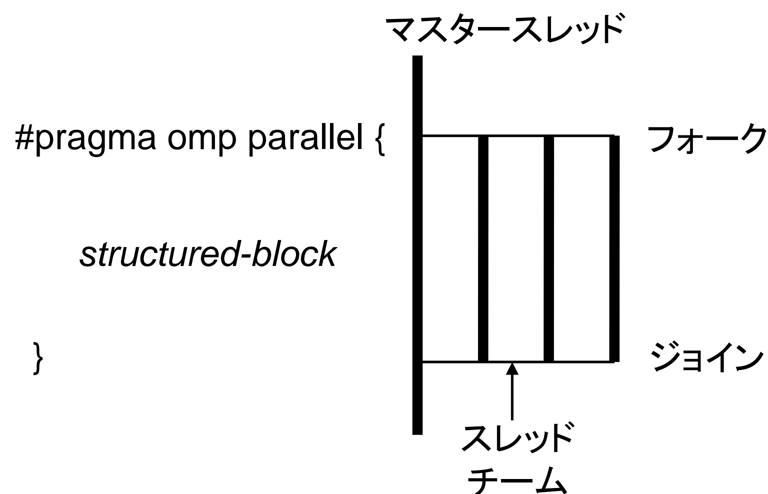
parallel構文

- parallel構文の文法は，以下の通り.

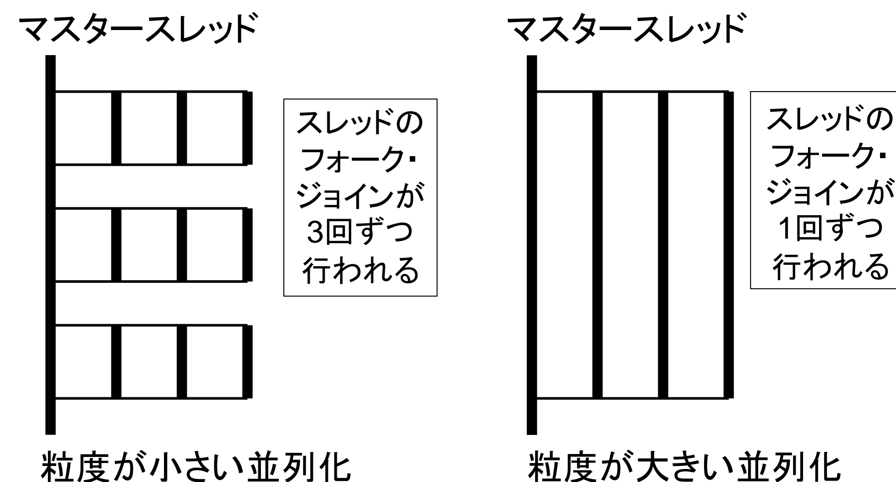
#pragma omp parallel *[clause[[,] clause] …]*
structured-block

- ここで，指示節(*clause*)は，以下のいずれかとなる.
  - if (*scalar-expression*)
  - num_threads (*integer-expression*)
  - default (shared | none)
  - private (*list*)
  - firstprivate (*list*)
  - shared (*list*)
  - copyin (*list*)
  - reduction (*operator: list*)

parallel構文とスレッドのフォーク･ジョイン



粒度が異なる並列化の例



ワークシェアリング構文

- ワークシェアリング構文は，その構文と関連付けられたリージョンの実行を，チーム内のスレッドに分配する．
- ワークシェアリング構文の例
  - ループ構文（#pragma omp for）
    - forループを各スレッドで分割
  - sections構文（#pragma omp sections）
    - 別々の処理を各スレッドが分担
  - single構文（#pragma omp single）
    - 1スレッドのみ実行

ループ構文

- ループ構文の文法は，以下の通り．

#pragma omp for *[clause[[,] clause] …] new-line*
for-loops

- ここで，指示節（*clause*）は，以下のいずれかとなる．
  - private (*list*)
  - firstprivate (*list*)
  - lastprivate (*list*)
  - reduction (*operator: list*)
  - schedule (*kind[, chunk_size]*)
  - collapse (*n*)
  - ordered
  - nowait

パラレルループ構文を用いたプログラムの例

```
#include <stdio.h>
#include <math.h>
double f( double );
double f( double a )
{
      return (4.0 / (1.0 + a*a));
}

int main( int argc, char *argv[])
{
      int n, i;
      double PI25DT = 3.141592653589793238462643;
      double pi, h, sum, x;

      scanf("&d", &n);
      h = 1.0 / (double) n;
      sum = 0.0;
#pragma omp parallel for private(x) reduction(+:sum)
      for (i = 1; i <=n; i++) {
         x = h * ((double) i – 0.5);
         sum += f(x);
      }
      pi = h * sum;
      printf(*pi is approximately %.16f, Error is %.16f¥n*, pi, fabs(pi – PI25DT));
      return 0;
}
```


sections構文

- sections構文の文法は，以下の通り．

```
#pragma omp sections  [clause[  [, ]  clause] …]  new-line
    {
    [  #pragma omp section  new-line]
           structured-block
    [  #pragma omp section  new-line
           structured-block  ]
     . . .
    }
```

- ここで，指示節（*clause*）は，以下のいずれかとなる．
  - private (*list*)
  - firstprivate (*list*)
  - lastprivate (*list*)
  - reduction (*operator: list*)
  - nowait

single構文

- single構文の文法は，以下の通り．

```
#pragma omp single  [clause[  [, ]  clause] …]  new-line
   structured-block
```

- ここで，指示節（*clause*）は，以下のいずれかとなる．
  - private (*list*)
  - firstprivate (*list*)
  - copyprivate (*list*)
  - nowait

複合パラレル・ワークシェアリング構文

- 複合パラレル・ワークシェアリング構文は，parallel構文のすぐ内側にネストされたワークシェアリング構文を指定するためのショートカットである．
- これらの指示文の文法は，1つのワークシェアリング構文だけから成るparallel構文を明示的に指定するのと同じ．
- 複合ワークシェアリング構文の例
  - パラレルループ構文（#pragma omp parallel for）
    - Forループを各スレッドで分割
  - parallel sections構文（#pragma omp parallel sections）
    - 別々の処理を各スレッドが分担

マスター・同期構文（1/2）

- master構文（#pragma omp master）
  - チームのマスタースレッドによって実行される構造化ブロックを指定する.
- critical構文（#pragma omp critical）
  - 一度に一つのスレッドだけが関連した構造化ブロックを実行するように制限する.
- barrier構文（#pragma omp barrier）
  - この構文が現れたポイントに明示的なバリアを指定する.

マスター・同期構文（2/2）

- atomic構文（#pragma omp atomic）
  - 特定の記憶域が，複数のスレッドによって同時に書き込みされる可能性を排除し，アトミックに更新されることを保証する.
- ordered構文（#pragma omp ordered）
  - ループリージョン中の構造化ブロックが，ループ繰り返しの順序で実行されることを指定する.

データ共有属性

- parallelリージョン内では，演算に用いられる変数が，
  - 共有変数（shared variable）
    - プログラムで1つの領域
    - どのスレッドからでも参照，更新が可能
  - プライベート変数（private variable）
    - スレッド毎に独立した領域
    - 各スレッドからのみ参照，更新が可能

  のどちらかであるかを明確に認識しておく必要がある.
- デフォルトでは基本的に共有変数（shared variable）となっている.

プライベート変数が必要な例

```
#pragma omp parallel for private(t)
for (i = 0; i < n; i++) {
      t = i + 1;
      a[i] = t + n;
}
printf(“%d¥n”, t);
```

forループの制御変数 i はデフォルトでプライベート変数

⇨ t の値は不定

- 変数 t をプライベート変数にしない場合，a[i] に入る値がタイミングによって変わってしまう.
- forループを出た後は，プライベート変数 t の値は不定になることに注意する.

reduction指示節

- reduction指示節の文法は，以下の通り．

reduction (*operator : list*)

- 右の表は，有効な演算子（operator）の例

Operator（演算子）	初期化値
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0

MPI並列プログラミング

- MPI（Message Passing Interface）は，並列プログラミングの規格として最も広く使われている．
- MPIは新しいプログラミング言語ではなく，CまたはFortranから呼び出す通信ライブラリである．

並列プログラミングのモデル

- 並列プログラミングのモデルとしては，大きく分けて，
  - SPMD（Single Program Multiple Data）
  - MPMD（Multiple Program Multiple Data）

  がある．
- SPMDモデルでは，同一のプログラムが各ノードで実行される．
- MPMDモデルでは，異なるプログラムが各ノードで実行される（マスター・ワーカー方式など）．

MPIを用いた並列プログラムの構成

```
#include “mpi.h”
#include <stdio.h>
#define N 1000

int main( int argc, char *argv[])
{
  int myid, nprocs, sendbuf[N], recvbuf[N];
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
…
  MPI_Send(sendbuf, N, MPI_INTEGER, (myid + 1) % nprocs, MPI_COMM_WORLD);
  MPI_Recv(recvbuf, N, MPI_INTEGER, (myid + 1) % nprocs, 0,
           MPI_COMM_WORLD, &status);
…
  MPI_Finalize();
  return 0;
}
```

MPI並列プログラミングの概要

(1) 最初に #include “mpi.h” を書く.
(2) MPI_Init()関数を呼んで，MPIの実行環境の初期化を行う.
(3) MPI_Comm_size()関数を呼んで，プロセス数を知る.
(4) MPI_Comm_rank()関数を呼んで，自分のプロセス番号を知る.
(5) MPI_Send()，MPI_Recv()などの関数を用いて通信を行う.
(6) MPI_Finalize()関数を呼んで，MPIの実行環境を終了する.

MPIの関数

- MPIでは100以上の関数が定義されている．大きく分けて以下の関数がある.
  - 1対1通信関数
  - 派生データ型とMPI_Pack/Unpack
  - 集団通信関数
  - グループ，コンテクスト，コミュニケータ
  - プロセストポロジ
  - 環境管理
- よほど特殊な並列化を行わない限り，20個程度の関数を知っていれば十分
  - その中でも特によく使用するのは10個程度

コミュニケータ

- コミュニケータ（communicator）はお互いにメッセージを送れるプロセスの集団である.
- よほど特殊な並列化を行わない限り，MPI_COMM_WORLD（全プロセスを含む初期コミュニケータ）を使えば十分.
- 必要であれば，別のコミュニケータを作成することも可能.

定義済みMPIデータ型

MPIデータ型	Cデータ型
MPI_CHAR	signed char
MPI_SHORT	signed short
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

1対1通信

- 1対1通信関数の例
  - ブロッキング通信（MPI_Send, MPI_Recv）
    - 一度呼び出すと，送受信が正常に完了するまで次の処理に進めない．
  - 非ブロッキング通信（MPI_Isend, MPI_Irecv, MPI_Wait）
    - 通信と演算をオーバーラップさせることが可能
  - 双方向通信（MPI_Sendrecv）
    - 安全に（デッドロックを起こさずに）双方向通信を行える．

1対1通信関数（1/2）

- int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
  - 送信関数
    - buf: 送信データバッファ
    - count: 送信データの個数
    - datatype: データタイプ
    - dest: メッセージの送信先を指定
    - tag: メッセージタグ
    - comm: 通信を行うグループの指定

1対1通信関数（2/2）

- int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
  - 受信関数
    - buf: 受信データバッファ
    - count: 受信データの個数
    - datatype: データタイプ
    - source: メッセージの送信元のプロセス番号を指定
    - tag: メッセージタグ
    - comm: 通信を行うグループの指定
    - status: 構造体MPI_Statusで受信状況を返す

メッセージタグ

- メッセージの種類を示す整数
  - メッセージの種類によって適切にタグを付けると便利．
  - コミュニケータ，メッセージタグ，送受信プロセス番号でsendとrecvの対応を決定する．
- 任意の送信元（MPI_ANY_SOURCE）や，任意のタグ（MPI_ANY_TAG）を指定することもある．

集団通信

- コミュニケータの中のすべてのプロセスを含む通信パターンを集団通信（collective communication）と呼ぶ.
- 集団通信は通常，二つ以上のプロセスを含む.
- 集団通信関数の例
  - ブロードキャスト（MPI_Bcast）
  - リダクション（MPI_Reduce，MPI_Allreduce）
  - ギャザ（MPI_Gateher，MPI_Allgather）
  - スキャッタ（MPI_Scatter，MPI_Allscatter）
  - 全対全通信（MPI_Alltoall）

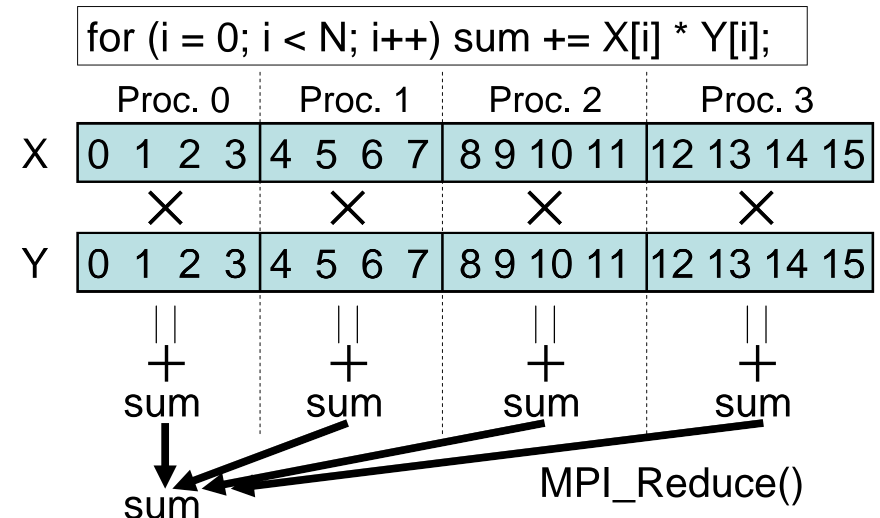
集団通信関数（1/2）

- int MPI_Barrier(MPI_Comm comm)
  - バリア同期を取る
    - comm: 通信を行うグループの指定
- int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
  - メッセージをrootで示されるプロセスからグループ全体に向けて送信する.
    - buf: 送受信データバッファ
    - count: 送受信データの個数
    - datatype: データタイプ
    - root: 他のプロセスに対してメッセージを送信するプロセスのプロセス番号
    - comm: 通信を行うグループの指定

集団通信関数（2/2）

- int MPI_Reduce(void *buf, int count, MPI_Datatype datatype, MPI_Op operator, int dest, MPI_Comm comm)
  - リダクション演算を行い，destで示されるプロセスに向けて送信する.
    - buf: 送受信データバッファ
    - count: 送受信データの個数
    - datatype: データタイプ
    - operator: オペレーションの指定
    - dest:メッセージの送信先を指定
    - comm: 通信を行うグループの指定

MPI_Reduceを用いた内積の並列化



コミュニケータ関数

- int MPI_Comm_size(MPI_Comm comm, int *size)
  - 通信を行うグループのサイズを決める.
    - comm: 通信を行うグループの指定
    - size: グループ内のプロセスの数を受け取る.
- int MPI_Comm_rank(MPI_Comm comm, int *rank)
  - 通信を行うグループのプロセスにプロセス番号を与える.
    - comm: 通信を行うグループの指定
    - rank: commの中でのプロセス番号を受け取る

環境管理関数

- int MPI_Init(int *argc, char **argv)
  - MPIの実行環境の初期化を行う.
    - argc: コマンド行の引数の数
    - argv: コマンド行の引数
- int MPI_Finalize(void)
  - MPIの実行環境を終了する.
- double MPI_Wtime(void)
  - 呼び出しプロセスの経過時間を返す

数値積分により円周率を求めるプログラム

```
#include “mpi.h”
#include <stdio.h>
#include <math.h>
double f( double );
double f( double a )
{
        return (4.0 / (1.0 + a*a));
}

int main( int argc, char *argv[])
{
        int done = 0, n, myid, numprocs, i;
        double PI25DT = 3.141592653589793238462643;
        double mypi, pi, h, sum, x;
        double startwtime = 0.0, endwtime;
        int namelen;
        char processor_name[MPI_MAX_PROCESSOR_NAME];

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD, &myid);
        MPI_Get_processor_name(processor_name, &namelen);

        if (myid == 0) scanf(“&d”, &n);
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        h = 1.0 / (double) n;
        sum = 0.0;
        for (i = myid + 1; i <=n; i += numprocs) {
           x = h * ((double) i – 0.5);
           sum += f(x);
        }
        mypi = h * sum;
        MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

        if (myid == 0) {
                printf(*pi is approximately %.16f, Error is %.16f¥n*, pi, fabs(pi – PI25DT));
        }
        MPI_Finalize();
        return 0;
}
```


数値積分により円周率を求める プログラムの一部

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
h = 1.0 / (double) n;
sum = 0.0;
for (i = myid + 1; i <=n; i += numprocs) {
   x = h * ((double) i – 0.5);
   sum += f(x);
}
mypi = h * sum;
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE,
                  MPI_SUM, 0, MPI_COMM_WORLD);
```