MPI による並列プログラミング

1. MPI について

MPI (Message Passing Interface) は分散メモリ型並列計算機における，メッセージパッシングをパラダイムとする並列プログラミングのための標準的な API (Application Programming Interface) を提供するライブラリである．かつては個々の並列計算機が独自のメッセージパッシングライブラリを持っていたが，1990 年代前半に PVM (Parallel Virtual Machine)，そしてその後 MPI へと移行し，現在は MPI がほぼ標準となり，ほとんどの分散メモリ型並列計算機に実装されている．

MPI を用いることにより，可搬性 (portability) の高い並列プログラミングが可能となる．また，本来は並列計算機用であったが，その後ワークステーションまたはパーソナルコンピュータを用いたクラスタシステムが普及するにあたり，それらを並列計算機システムとして扱うことを可能にする最も有力な手法として重要視されている．

2. メッセージパッシングに基づく並列プログラム

分散メモリ型並列計算機（クラスタシステムもこれに含めることとする）では，各ノード（プロセッサ）上でそれぞれプロセスを走らせ，それらがメッセージ交換を行なうことにより協調的な並列処理を進めるというプログラミングパラダイムが一般的である（分散メモリ上のメッセージパッシングを用いて，ソフトウェア的に共有メモリ環境を実現する software DSM (Distributed Shared Memory) の研究も行なわれている）．ここでは，各プロセスは独立なアドレス空間を持つため，変数を共有した並列プログラミングは行なえない．プロセス群は処理を進めるために，適宜，各々のデータを他のプロセスに送信したり，他のプロセスから必要なデータを受信したりして処理を進める．

例えば，あるプロセスが持つ配列データの個々の要素に対する処理が，要素間でのデータ依存性を持たない場合，それを複数のプロセスに分配し，それぞれが部分処理を行なった後でもう一度元のプロセスに計算結果を戻すことにより並列処理が実現できる．例として，以下のループ処理を考える．

```
double a[10000], b[10000];
int i;

for(i = 0; i < 10000; i++)
b[i] = func(a[i]);
```

ここで，関数 `func()` 内の処理は引数 `a[i]` のみに依存し，10000 回のループ処理における各々の反復でのデータ依存関係はないものとする．これを 10 個のプロセスによって並列処理する場合，以下のように行なえる．

```
/* process-0 */
double a[10000], b[10000];
int j;
```

```
for(j = 0; j < 10; j++)
   a[j*1000] から a[(j+1)*1000-1] までを process-(j+1) に送信;
for(j = 0; j < 10; j++)
   process-(j+1) からのデータを b[j*1000] から b[(j+1)*1000-1] までに格納;

/* process-j (j=1,2,...,10) */
double aa[1000], bb[1000];
int i;

process-0 から受信したデータを aa[0] から aa[999] までに格納;
for(i = 0; i < 1000; i++)
   bb[i] = func(aa[i]);
aa[0] から aa[999] までを process-0 に送信;
```

このように主となる 1 つのプロセスが，他の worker となるプロセスに仕事を分配する手法を master-slave 型と呼ぶ．この他にもいろいろなスタイルがあるが，master-slave は最も典型的な並列処理のスタイルの一つである．

このように，並列処理に参加する複数のプロセスは，互いにデータを交換し合いながら全体的には 1 つの協調処理を進めていく．

3. SPMD モデル

MPI では並列プロセスの一つ一つは，UNIX における通常のプロセスとほとんど同じである．実際に，ワークステーションクラスタ等では本当の UNIX のプロセスを並列プロセスとして扱う．ここで，各並列プロセスの内容をどうプログラムするかであるが，MPI では SPMD (Single Program Multiple Data) と呼ばれる概念に基づきプログラムを記述する．

SPMD モデルでは，1 つの並列処理のためにはプログラムは 1 つしか存在しない．その仕事のための複数の並列プロセスは全て，この 1 つのプログラムを実行する．そして，そのプログラムの中で「自分は何番目のプロセスとして動いているか」ということを基準として，適宜処理を変更しながら (例えば if 文等で) 全体として無矛盾になるようにプログラムを記述する．MPI 環境下でプログラムが実行されると，指定された数のプロセスが生成され，並列計算機 (またはクラスタ) 環境のプロセッサに分配される．プログラムの起動時に，コマンドライン引数としてプログラムのファイル名を 1 つ指定する．

MPI プログラムでは，「その仕事にかかわっているプロセスが全部でいくつあるか」「自分はその中の何番目のプロセスであるか」といった情報を，プログラム中で参照できる．MPI では協調動作するプロセスの集合を communicator と呼び，その communicator の中での個々のプロセスの番号を rank と呼ぶ．従って，各プロセスはまず自分の rank を求め，それに従って処理内容を適宜変更しつつ処理を進める．プログラム実行時から暗黙に与えられる communicator として，WORLD と呼ばれるものがある．これは，起動された全ての並列プロセスを含む集合である．この他，プログラム実行中に適宜，任意のプロセスの集合によって communicator を新たに定義することも可能である．

4. メッセージパッシング

MPI プログラム中で各プロセスはメッセージを交換し合う．メッセージ交換には 2 つの種類がある．1 つ目は point-to-point 通信で，これは一対のプロセスの間で，片方が送信者 (sender)，もう片方が受信者 (receiver) として，それぞれ送受信のための関数を実行し，通信を行なうものである (さらに通信はバッファ型，同期式，非同期式等，いろいろなカテゴリに分類されるが，ここでは詳細は省略する)．2 つ目は collective 通信と呼ばれるもので，これは 2 つ以上の多数のプロセスが関係し，それら全てを含んだ協調的な通信が行なわれる．例えば，全てのプロセスが持つ部分データを一つのプロセスに集めたり，全プロセスのデータの総和を求めたりするような通信がこれに当たる．ここで注意すべきは「送信されたメッセージは，必ず誰かに受信されなければならない」ということである．つまり，point-to-point にしろ collective にしろ，全てのメッセージは送受信の関係がきちんと成り立ち，論理的に矛盾がないようにしなければならないということである．メッセージパッシングに基づく並列プログラミングはこの点が最も重要であり，またバグが入りやすい部分でもある．どのプロセスがどういうメッセージを出し，それが誰に届くべきであるかを常に把握しつつプログラミングする必要がある．

MPI における point-to-point のメッセージパッシングは，同じ communicator 内のプロセス間で相手を指定しながら行なわれる．この時，特定の相手を指定するのが一般的であるが，必要に応じて wild card を用いることも可能である．例えば「誰からでもいいからデータを受信する」というような記述が可能である．また，受信した後で結果的にそれが誰からの物であったかを調べることもできる．

また，MPI では point-to-point 送受信において，メッセージにタグ (tag) を付けることができる．tag は複数のメッセージを区別するための道具であり，例えば同じプロセス対の中でメッセージを送るにしても，異なる性質のデータには別の tag を付けて送り，プログラム上のミスを減らすことができる．MPI で送受信されるデータは全て型を持っている (int, float 等)．なお，各データは単なるバイトデータの配列としては扱わず，必ず数値データとして処理する．これは，endian 問題等，数値表現に関する互換性を維持するためである．また，送受信単位はそのデータ型の配列となっている．単一のデータの送受信する場合は，サイズ 1 の配列として扱う．

5. デッドロック

メッセージパッシング型のプログラムでは，デッドロック (deadlock) が生じやすい．これは，複数のプロセスが，相手の起こす何らかのイベントを互いに待ち合って，結果的に全体の処理が停止してしまう (つまり，「二進も三進もいかない状態」になる) ことを言う[*1]．デッドロックはプログラマの不注意や，プログラム全体の論理構造の見誤りによって容易に生じる．

最も簡単なデッドロックは，2 つのプロセスが，互いにまず相手からのデータを受信し，その後で相手に対しデータを送信する，というパターンである．最初の受信命令で，両者ともストップし永久に動かなくなってしまう．これと同様なことが，例えば三者間で生じたり，間接的に生じたりする例もよくある．この他に，システムのリソースが不足し，論理的には無矛盾に見えても，結果的にデッドロックしてしまうような場合もある．

[*1] 実は本当に始末におえないのは，「一見，処理が進んでいるように見えるが意味のないメッセージが循環するような状態に陥いり，結果として処理が進まない」というライブロック (livelock) の方である．が，ここでは割愛する．

並列プログラムの初心者は大抵，このデッドロック状態を招いてしまう．プログラムの設計を十分に吟味し，初期段階での誤りがないように注意すべきである．

6. MPI の実行環境

MPI にはいくつかの実装がある．クラスタ等を対象としたフリーなもので最も有名なのは MPICH, MVAPICH 及び OpenMPI である．本演習では OpenMPI を用いる．実際に MPI を用いるには以下のような環境設定が必要となる．

Pegasus でコンパイラを使えるようにするには，`module` コマンドを用いてモジュールをロードする必要がある．以下のコマンドを実行すると，使用可能なモジュール一覧が表示される．

```
% module avail
```

例えば，GCC と OpenMPI を使いたい場合は，

```
% module load openmpi/5.0.7/gcc11.4.0-cuda12.8.1
```

と実行することで，`mpicc` コマンドが使えるようになる．

次に，hostfile というファイルを用意する場合があるが，このファイルはオプショナルであり，通常の場合，システムに任せておけばよく，省略可能である（Pegasus では自分で hostfile を用意する必要はない）．このファイルは，クラスタシステム上のどのマシンを MPI 環境として取り込むかを指定するファイルである．このファイル中に，1 行当たり 1 つずつ，マシンの名前を記述する．1 つのマシンが複数行に書かれていてもエラーではない．実際に並列プロセスが生成される時，ホストファイル中の各マシンに順番にラウンドロビン方式でプロセスが割り当てられる．

実際のプログラムのコンパイルは，MPI 用の C コンパイラコマンドである `mpicc` を用いる．ただし，OpenMP 等と組み合わせて用いる場合（ハイブリッドプログラミング）は，OpenMP 用のコンパイラに適宜ライブラリを追加する形でコンパイルする．

オブジェクトファイルの実行には mpirun というコマンドを用いる．

`% mpirun -np` **プロセス数 オブジェクトファイル** ［**引数**］

プロセス数には，そのプログラムをいくつの並列プロセスで実行するかを指定する．

7. MPI の主な関数

MPI の関数は非常に多数あるが，ここでは代表的なものとその機能を紹介する．

(1) 初期設定に関するもの

- `int MPI_Init(int *argc, char **argv[])`
  MPI の初期化を行ない，引数のうち MPI 設定に関係するものを処理する．

- `int MPI_Comm_size(MPI_Comm comm, int *size)`
  communicator 中のプロセス数を求める．

- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
  communicator 中の自分の rank を求める.

- `int MPI_Get_processor_name(char name[], int *length)`
  プロセスが実行されているホスト名を得る.

- `int MPI_Finalize()`
  そのプロセスの MPI としての実行を終了する.

(2) point-to-point 通信に関するもの

- `int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  データを送信する.

- `int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
  データを受信する.

(3) collective 通信に関するもの

- `int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`
  root の持つデータを全プロセスに放送する.

- `int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
  全プロセスに分散しているデータをまとめ，root プロセスに 1 つの配列として渡す.

- `int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
  root プロセス中の配列データを全プロセスに分配する.

- `int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`
  全プロセスが持つ部分データを集め，その全体を全プロセスに戻す.

- `int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`
  全プロセスが持つ部分データを，それぞれ一部ずつが他のプロセスに渡るように分配する（全員に同じデータが行かない）.

- `int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`
  リダクション (reduction) 通信を実行し，結果を root に返す.

- `int MPI_Allreduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`
  リダクション通信を実行し，結果を全プロセスに返す.

- `int MPI_Scan(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`
  リダクション通信に似ているが，各プロセスに返されるのは，自分の rank より低い rank のプロセスだけからなる集合でのリダクション結果である.

(4) その他

- `int MPI_Barrier(MPI_Comm comm)`
  全てのプロセスがこの命令を実行し終わるのを待つ.

- `double MPI_Wtime()`
  wall clock の値を求める.

データの型には，以下のものがある.

MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE

また，リダクション演算には以下のものがある.

MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR, MPI_MAXLOC, MPI_MINLOC