Day 1: 環境構築

スパコン上で実行されるプログラムは並列プログラムである。したがって「スパコンを使う」ということは、狭義には「並列化されたプログラムを実行する」ということを意味する。したがって、誰かが作った並列プログラムをスパコン上で実行すれば、スパコンは使えることになる。それはそれで OK なのだが、本稿のタイトルは「一週間でなれる！スパコンプログラマ」であるから、スパコン上で動くコードを開発できるようになることを目的とする。それはすなわち、「並列プログラミングをする」ということである。「並列プログラミング」という字面を見ると「難しそう」という印象を持つかもしれない。しかし、(世の中の多くの「一見難しそうなもの」がそうであるように) 並列プログラミングはさほど難しくない。「一週間でなれる！スパコンプログラマ」の初日は、まずローカルに並列プログラミング環境を構築し、並列プログラミングに触れてみるところからはじめてみよう。

MPI とは

一口に「並列化」といっても、様々な種類がありえる。一般に使われている並列プログラミングモデルは、「データ並列」「共有メモリ並列」「分散メモリ並列」の三種類であろう。以後、プロセスやスレッドといった単語についてかなりいい加減な言葉遣いをするため、ちゃんと学びたい人はちゃんとした書籍を参考にされたい。特に Windows と Linux のプロセスの違いとか言い出すと話が長くなるので、ここでは説明しない。また、データ並列についてはとりあえずおいておく。

「共有メモリ並列」とは、並列単位がメモリを共有する並列化方法である。通常は並列単位としてスレッドを用いるので、ここでは「スレッド並列」と呼ぶ。逆に「分散メモリ並列」とは、並列単位がメモリを共有しない並列化方法である。通常は並列単位としてプロセスを用いるので、ここでは「プロセス並列」と呼ぶ。また、「プロセス並列」と「スレッド並列」を両方行う「ハイブリッド並列」という並列化もある。

まずはプロセスとスレッドの違いについて見てみよう。プロセスとは、OS から見た資源の管理単位である。プロセスは OS から様々な権限を与えられるが、最も重要なことは「OS から独自のメモリ空間を割り当てられる」ことである。異なるプロセスは異なるメモリ空間を持っており、適切な権限がなければ他のプロセスのメモリを参照できない (そうしないとセキュリティ的に問題がある)。

スレッドとは CPU の利用単位である。通常、一つの CPU コアを利用できるのは一度に一つのスレッドだけである (SMT などはさておく)。各プロセスは最低一つのスレッドを持っており、プログラムの実行とは、スレッドが CPU コアを使いにいくとである。図解するとこんな感じになる。

「スレッド並列」では、一つのプロセスの中に複数のスレッドを立ち上げ、各スレッドが複数の CPU コアを使うことで性能向上をはかる。複数のスレッドが一つのメモリを共有するため、「共有メモリ並列」となる。例えば OpenMP でディレクティブを入れたり、`std::thread` などを使って明示的にスレッドを起動、制御することで並列化を行う。同じプロセス内のスレッドはメモリを共有するため、お互いに通信をする必要はないが、同時に同じところを書き換えたりしないようにプログラマの責任で排他制御を行う必要がある。コンパイラによっては自動並列化機能を持っているが、それで実現されるのはこのスレッド並列である。

「プロセス並列」とは、複数のプロセスを立ち上げ、それぞれのプロセスに属すスレッドが CPU コアを使いにいくことで性能向上をはかる。プロセス並列は MPI(Message Passing Interface) というライブラリを使って行う。それぞれのプロセスは独自のメモリ空間を持っており、基本的にはお互いから見えないため、意味のある並列化を行うためには、必要に応じて通信を行わなければならない。

さて、プロセス並列とスレッド並列では、一般的にはスレッド並列の方がとっつきやすいと言われている。以下のような単純なループがあったとする。

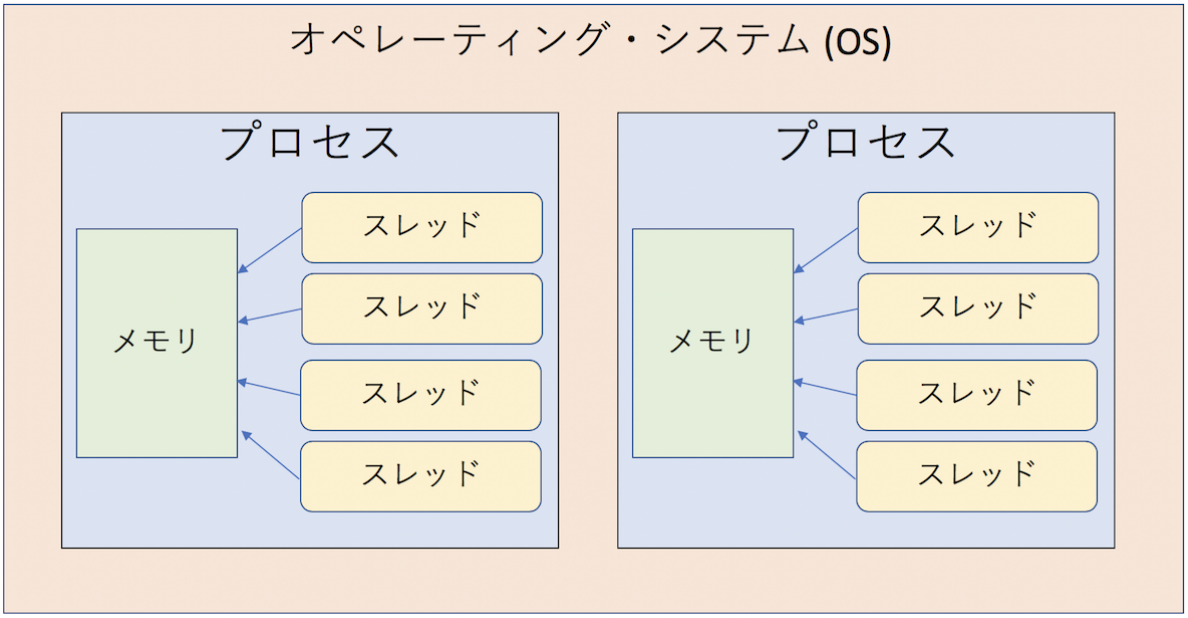

図 1: fig/process_thread.png

```
const int SIZE = 10000;
void func(double a[SIZE], double b[SIZE])
for (int i=0; i < SIZE; i++) {
  a[i] += b[i];
}
```

これを OpenMP でスレッド並列化したければ、以下のようなディレクティブを入れるだけで良い。

```
const int SIZE = 10000;
#pragma omp parallel for  // ← OpenMP による並列化の指示
void func(double a[SIZE], double b[SIZE])
for (int i=0; i < SIZE; i++) {
  a[i] += b[i];
}
```

同じようなことを MPI でやろうとするとわりとごちゃごちゃ書かなければいけない上に、下手な書き方をするとオーバーヘッドが大きくなって効率が悪くなるかもしれない。しかし、スレッド並列はプロセスの中の並列化であり、一つのプロセスは複数の OS にまたがって存在できないため、複数のノード (ノードの説明については後述) を同時に使うことができない。

逆にプロセス並列の場合は、各プロセスが独立したメモリを保持しているため、他のプロセスの保持するデータがほしければ通信を行う必要がある。この通信はユーザが関数呼び出しを使って明示的に行わなければならない。ただし、通信は別のハードにある別の OS 上で実行中の別のプロセスとも行うことができるため、複数のノードを同時に使うことができる。

上図に、簡単に「スレッド並列」と「プロセス並列」の違いをまとめた。本稿の目的は「スパコンプログラマ」になることであり、「スパコン」とは複数のノードを束ねたものであり、「スパコンプログラミング」とは複数のノードをまとめて使うことであるから、論理的必然としてスパコンプログラミングをするためにはプロセス並列が必要となる。というわけで、本稿では主に MPI を用いた分散メモリ並列を取り上げる。

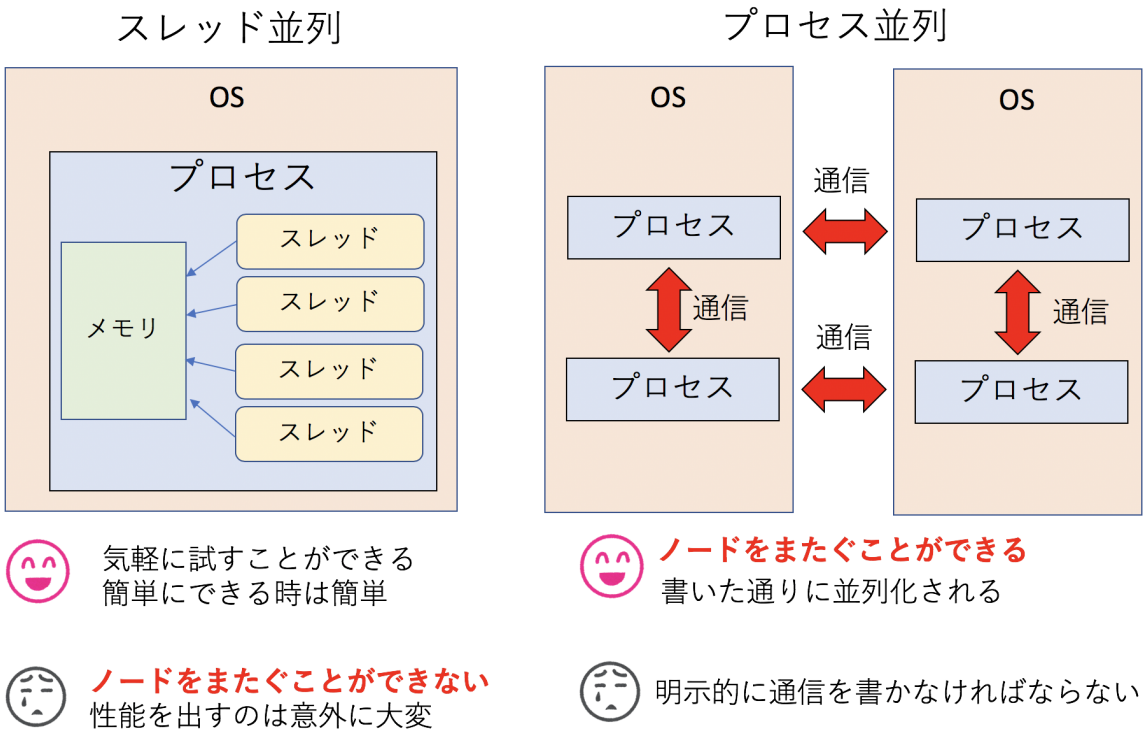

図 2: fig/comparison.png

余談：MPI は難しいか

たまに「スレッド並列は簡単」「MPI は難しい」という声を聞く。しかし、それなりの期間スパコンと関わって思うのは、「どちらかといえばスレッド並列の方が難しい」「MPI は面倒くさいが難しくない」というのが実感である。

確かに OpenMP なんかを使えば、一行いれるだけでスレッド並列化されて、簡単に数倍の性能が得られたりする。しかし、性能が出なかった時に「なぜ性能が出なかったのか」を調べるのはとても大変である。なぜなら OpenMP を使うと「どのように並列化されたか」が隠蔽されてしまうからだ。基本的にはコンパイラが吐くレポートを見て、どのように並列化されたかを「想像」しながら調査をする必要がある。また、同じメモリを複数のスレッドが同時に触りにくるため、「タイミングによってはバグる」という問題を起こす。**この手のマルチスレッドプログラミングのデバッグは一般に地獄である**。少なくとも私はやりたくない。

MPI はいろいろ書かなければならないことが多く、確かに面倒である。しかし、基本的には「書いた通り」に並列化される (ここ読んだプロが「そんなことない！」と怒っている顔が目に浮かぶが、まぁOpenMP と比較して、の話ですよ)。また、各プロセスのメモリは各プロセスだけのものである。通信するにしても、自分が用意したバッファに他のプロセスが書き込んでくるため、「いつ」「どこに」「誰が」「どのくらい」書き込んでくるかがわかる。これはデバッグするのに非常に重要な情報である。

そんなわけで、「どうせ並列化するなら、最初から MPI で書いてしまえばいいんじゃない？複数ノードを使えるようにもなるし」というのが私の見解である。 MPI で必要となる関数も、初期化 (`MPI_Init`) と後処理 (`MPI_Finalize`) を除けば、相互通信 (`MPI_Sendrecv`) と集団通信 (`MPI_Allreduce`) の二つだけ知っていればたいがいのことはなんとかなる。それ以上にややこしいことをやりたくなったら、その時にまた調べれば良い。

MPI のインストール

スパコンを使う前に、まずはローカル PC で MPI による並列プログラミングに慣れておこう。 MPI の開発環境としては、Mac OSX、もしくは Linux を強く推奨する (というか筆者は Windows での並列プログラミング環境の構築方法を知らない)。 Linux はなんでも良いが、とりあえず CentOS を想定する。本稿の読者なら、GCC くらいは既に利用可能な状況であろう。あとは MPI をインストールすれば並列プログラミング環境が整う。

Mac で Homebrew を使っているなら、

```
brew install openmpi
```

で一発、CentOS なら、

```
sudo yum install openmpi-devel
export PATH=$PATH:/usr/lib64/openmpi/bin/
```

でおしまいである。ここで

```
sudo yum install openmpi
```

とすると、開発環境がインストールされないので注意。

インストールがうまくいったかどうかは、MPI 用コンパイラ `mpic++`にパスが通っているかどうかで確認できる。実は、mpic++はインクルードパスやリンクの設定をユーザの代わりにやってくれるラッパーに過ぎず、実際のコンパイラは `clang++`、もしくは `g++`が使われる。

例えば Mac では

```
$ mpic++ --version
Apple LLVM version 10.0.0 (clang-1000.11.45.2)
Target: x86_64-apple-darwin17.7.0
Thread model: posix
InstalledDir: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin
```

と、clang++にパスが通っていることがわかる。 Linux の場合は g++である。

```
$ mpic++ --version
g++ (GCC) 4.8.5 20150623 (Red Hat 4.8.5-28)
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

したがって、インクルードパスやリンクの設定を明示的にするならば、`mpic++`を呼び出す必要はない。スパコンサイトによっては、環境変数で MPI のインクルードパスが設定されている場合もあるだろう。その場合は単に `g++`でも `icpc` でも、MPI を用いたコードがそのままコンパイルできる。ただし、リンクのために`-lmpi` の指定が (場合によっては`-lmpi_cxx` も) 必要なので注意。

はじめての MPI

環境構築ができたら、こんなコードを書いて、hello.cpp という名前で保存してみよう。

```
#include <cstdio>
#include <mpi.h>

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
  printf("Hello MPI World!\n");
  MPI_Finalize();
}
```

以下のようにしてコンパイル、実行してみる。

```
$ mpic++ hello.cpp
$ ./a.out
Hello MPI World!
```

せっかくなので並列実行する前に、mpic++を使わずにコンパイルできることを確認してみよう。 Mac の場合、g++で先程の hello.cpp をコンパイルしようとすると「mpi.h が見つからないよ」と怒られる。

```
$ g++ hello.cpp
hello.cpp:2:10: fatal error: mpi.h: No such file or directory
 #include <mpi.h>
          ^~~~~~~
compilation terminated.
```

なので、コンパイラにその場所を教えてあげればよい。ヘッダファイルの場所だけ教えても「ライブラリが見つからないよ」と怒られるので、それも一緒に教えてあげよう。

```
g++ hello.cpp -I/usr/local/opt/open-mpi/include -L/usr/local/opt/open-mpi/lib -lmpi  -lmpi_cxx
```

問題なくコンパイルできた。ちなみに筆者の手元の CentOS では、

```
g++ test.cpp -I/usr/include/openmpi-x86_64 -L/usr/lib64/openmpi/lib -lmpi -lmpi_cxx
```

でコンパイルできた。環境によってパスは異なるが、インクルードパスとライブラリパス、そしてライブラリ-lmpi(環境によっては-lmpi_cxx も) を指定すれば mpic++を使わなくてもコンパイルできる。「mpic++はラッパに過ぎず、ヘッダとライブラリを正しく指定すればどんなコンパイラでも使える」と知っていると、MPI 関連でトラブルが起きた時にたまに役に立つので覚えておきたい。

さて、並列実行してみよう。並列実行には mpirun の引数に実行プログラムと並列数を指定する。

```
$ mpirun -np 2 ./a.out
Hello MPI World!
Hello MPI World!
```

メッセージが二行表示された。プログラムの実行の際、こんなことが起きている。

1. mpirun が-np 2 を見て、プロセスを 2 つ立ち上げる。
2. MPI_Init が通信環境を初期化する
3. 各プロセスが、それぞれ Hello MPI World を実行する。
4. MPI_Finalize が通信環境を終了する。

複数のプロセスが立ち上がり、なにか処理をしているのだから、これは立派な並列プログラムである。しかし、このままでは、全てのプロセスが同じ処理しかできない。そこで、MPI は立ち上げたプロセスに**ランク (rank)** という通し番号を振り、それを使って並列処理をする。

ランク

MPI では、起動したプロセスに通し番号が振られる。その通し番号のことを**ランク (rank)** と呼ぶ。ランクの取得には `MPI_Comm_rank` 関数を使う。

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

これを実行すると変数 `rank` にランク番号が入る。 N 並列している場合、ランクは 0 から N-1 までである。試してみよう。

rank.cpp

```
#include <cstdio>
#include <mpi.h>

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Hello! My rank is %d\n", rank);
  MPI_Finalize();
}
```

実行するとこうなる。

```
$ mpic++ rank.cpp
$ mpirun -np 4 ./a.out
--------------------------------------------------------------------------
There are not enough slots available in the system to satisfy the 4 slots
that were requested by the application:
  ./a.out

Either request fewer slots for your application, or make more slots available
for use.
--------------------------------------------------------------------------
```

おっと、エラーが出た。このエラーは「予め定義されたスロット数よりプロセス数が多いよ」というもので、筆者の環境では Mac では出るが、Linux ではでない。このエラーが出た場合は `mpirun` に`--oversubscribe` オプションをつける。

```
$ mpirun --oversubscribe -np 4 ./a.out
Hello! My rank is 0
Hello! My rank is 2
Hello! My rank is 1
Hello! My rank is 3
```

無事にそれぞれのプロセスで異なるランク番号が表示された。

MPI プログラムでは、全く同じソースコードのレプリカが作成される。違いはこのランクだけである。したがって、プログラマはこのランク番号によって処理を変えることで、並列処理を書く。どんな書き方をしても自由である。例えば 4 並列実行する場合、

```cpp
#include <cstdio>
#include <mpi.h>

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    // Rank 0の処理
  } else if (rank == 1) {
    // Rank 1の処理
  } else if (rank == 2) {
    // Rank 2の処理
  } else if (rank == 3) {
    // Rank 3の処理
  }
  MPI_Finalize();
}
```

みたいな書き方をしても良い。この場合、ランク 0 から 3 まで全く関係のない仕事をさせることができる。まぁ、普通はこういう書き方をせずに、後で説明する「サンプル並列」「パラメタ並列」や、「領域分割」をすることが多いが、「そうすることが多い」というだけで、「そうしなければならない」ということではない。まとめると、

- MPI は、複数のプロセスを起動する
- それぞれのプロセスには、「ランク」という一意な通し番号が振られる。
- MPI プログラムは、ランクの値によって処理を変えることで、プロセスごとに異なる処理をする

こんな感じになる。これが MPI プログラムのすべてである。なお、複数のノードにまたがって MPI プロセスを立ち上げた場合でも、ランク番号は一意に振られる。例えば、ノードあたり 4 プロセス、10 ノードで実行した場合、全体で 40 プロセスが起動されるが、それぞれに 0 から 39 までのランク番号が振られる。その番号が、各ノードにどのように割当られるかは設定に依るので注意。

標準出力について

さて、端末で

```
mpirun -np 4 ./a.out
```

などとして MPI プログラムを実行したとする。この場合は 4 プロセス立ち上がり、それぞれに PID が与えられ、固有のメモリ空間を持つ。しかし、これらのプロセスは標準出力は共有している。したがって、「せーの」で標準出力に出力しようとしたら競合することになる。この時、例えば他のプロセスが出力している時に他のプロセスが書き込んだり、出力が混ざったりしないように、後ろで交通整理が行われる。そもそも画面になにかを表示する、というのはわりと奥が深いのだが、そのあたりの話は tanakamura さんの実践的低レベルプログラミングとかを読んでほしい。

さて、とにかく今は標準出力というリソースは一つしかないのに、4 つのプロセスがそこを使う。この時、「あるひとかたまりの処理については、一つのプロセスが独占して使う」ようにすることで競合が起きないようにする。「ひとかたまりの処理」とは、例えば「`printf` で出力を始めてから終わるまで」である。

例えば先程の rank.cpp の例では、

```
printf("Hello! My rank is %d\n", rank);
```

という命令があった。ここでは、まず出力すべき文字列、例えば Hello! My rank is 0 を作る。そして、せーので書き出す。イメージとしては

```
puts("Hello! My rank is 0");
puts("Hello! My rank is 1");
puts("Hello! My rank is 2");
puts("Hello! My rank is 3");
```

という 4 つの命令が「ランダムな順番で」に実行される。しかし、順番が入れ替わっても

```
puts("Hello! My rank is 0");
puts("Hello! My rank is 2");
puts("Hello! My rank is 1");
puts("Hello! My rank is 3");
```

とかになるだけで、さほど表示は乱れない。

さて、同様なプログラムを std::cout で書いてみよう。

こんな感じになると思う。

rank_stream.cpp

```
#include <iostream>
#include <mpi.h>

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  std::cout << "Hello! My rank is " << rank << std::endl;
  MPI_Finalize();
}
```

さて、この場合は、プロセスの切り替わりタイミング (標準出力の占有期限) が<<で切り替わる可能性がある。

イメージとしては、

```
std::cout << "Hello! My rank is ";
std::cout << "0";
std::cout << std::endl;
std::cout << "Hello! My rank is ";
std::cout << "1";
std::cout << std::endl;
std::cout << "Hello! My rank is ";
std::cout << "2";
std::cout << std::endl;
std::cout << "Hello! My rank is ";
```

```
std::cout << "3";
std::cout << std::endl;
```

という命令が「ランダムな順番で」に実行される可能性がある。

すると、例えば

```
$ mpirun -np 4 ./a.out
Hello! My rank isHello! My rank is
0
1
Hello! My rank is 3

Hello! My rank is 2
```

のように表示が乱れる可能性がある。これが乱れるかどうかは、システムのバッファリングがどうなっているかに依存し、システムのバッファリングがどうなっているかは、MPI の実装に依存する。

参考→ MPI における各プロセスの連携と標準出力のバッファリング

つまり、手元で表示が乱れなくても、別のマシンでは表示が乱れる可能性がある。なので、もし標準出力を使いたいなら、一度 `std::stringstream` にまとめてから一度に書き出すか、素直に `printf` を使うほうが良いと思う。

なお、MPI のデバッグ目的に標準出力を使うのは良いが、例えば結果の出力や計算の進捗の出力に使うのはあまりおすすめできない。そのあたりはジョブスケジューラを用いたジョブの実行のあたりで説明すると思う。

ちなみに、Open MPI には「各ランクごとの標準出力を、別々のファイルに吐き出す」というオプションがある。

参考：MPI プログラムのデバッグ

例えば、先程の例では、

```
mpirun --output-filename hoge -np 4 ./a.out
```

とすると、Linux では標準出力の代わりに `hoge.1.X` というファイルがプロセスの数だけ作成され、そこに保存される。中身はこんな感じ。

```
$ ls hoge.*
hoge.1.0  hoge.1.1  hoge.1.2  hoge.1.3

$ cat hoge.1.0
Hello! My rank is 0

$ cat hoge.1.1
Hello! My rank is 1
```

Mac で同様なことをすると、以下のようにディレクトリが掘られる。

```
$ mpiexec --output-filename hoge -np 4 --oversubscribe ./a.out
Hello! My rank is 0
Hello! My rank is 1
```

```
Hello! My rank is 2
Hello! My rank is 3

$ tree hoge
hoge
└── 1
    ├── rank.0
    │   ├── stderr
    │   └── stdout
    ├── rank.1
    │   ├── stderr
    │   └── stdout
    ├── rank.2
    │   ├── stderr
    │   └── stdout
    └── rank.3
        ├── stderr
        └── stdout
```

標準出力にも出力した上で、各ディレクトリに、各プロセスの標準出力と標準エラー出力が保存される。覚えておくと便利な時があるかもしれない。

GDB による MPI プログラムのデバッグ

本稿の読者の中には普段から gdb を使ってプログラムのデバッグを行っている人がいるだろう。並列プログラムのデバッグは一般に極めて面倒だが、とりあえず gdb を使った MPI プログラムのデバッグ方法を知っていると将来何かの役に立つかもしれない。あと、これは筆者だけかもしれないが、ソース読んだりするより、gdb 経由でプログラムの振る舞いを解析したほうがなんかいろいろ理解しやすかったりする。というわけで、gdb で MPI プロセスにアタッチする方法を紹介する。普段から gdb を使っていなければこの節は読み飛ばしてかまわない。

gdb でデバッグできるのは一度に一つのプロセスのみである。しかし、MPI プログラムは複数のプロセスを起動する。したがって、

- 起動されたすべてのプロセスについて gdb をアタッチする
- 特定のプロセス一つだけに gdb をアタッチする

の二通りの方法が考えられる。ここでは後者の方法を採用する。なお、両方の方法が Open MPI の FAQ: Debugging applications in parallel に記載されているので興味のある方は参照されたい。

gdb は、プロセス ID を使って起動中のプロセスにアタッチする機能がある。そこで、まず MPI プログラムを実行し、その後で gdb で特定のプロセスにアタッチする。しかし、gdb でアタッチするまで、MPI プログラムには特定の場所で待っていてほしい。というわけで、

- 故意に無限ループに陥るコードを書いておく
- MPI プログラムを実行する
- gdb で特定のプロセスにアタッチする
- gdb で変数をいじって無限ループを脱出させる
- あとは好きなようにデバッグする

という方針でいく。なお、なぜか Mac OS では MPI プロセスへの gdb のアタッチがうまくいかなかったので、以下は CentOS で実行している。

こんなコードを書く。

gdb_mpi.cpp

```
#include <cstdio>
#include <sys/types.h>
#include <unistd.h>
#include <mpi.h>

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Rank %d: PID %d\n", rank, getpid());
  fflush(stdout);
  int i = 0;
  int sum = 0;
  while (i == rank) {
    sleep(1);
  }
  MPI_Allreduce(&rank, &sum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
  printf("%d\n", sum);
  MPI_Finalize();
}
```

MPI_Allreduce はまだ説明していないが、全プロセスで変数の総和を取る関数である。このコードは、自分の PID を出力してから、ランク 0 番のプロセスだけ無限ループに陥る。このコードを-g つきでコンパイルし、とりあえず 4 プロセスで実行してみよう。

```
$ mpic++ -g gdb_mpi.cpp
$ mpirun -np 4 ./a.out
Rank 2: PID 3646
Rank 0: PID 3644
Rank 1: PID 3645
Rank 3: PID 3647
```

4 プロセス起動して、そこでランク 0 番だけ無限ループしているので、他のプロセスが待ちの状態になる。この状態でランク 0 番にアタッチしよう。もう一枚端末を開いて gdb を起動、ランク 0 の PID(実行の度に異なるが、今回は 3644) にアタッチする。

```
$ gdb
(gdb) attach 3644
Attaching to process 3644
Reading symbols from /path/to/a.out...done.
(snip)
(gdb)
```

この状態で、バックトレースを表示してみる。

```
(gdb) bt
#0  0x00007fc229e2156d in nanosleep () from /lib64/libc.so.6
#1  0x00007fc229e21404 in sleep () from /lib64/libc.so.6
#2  0x0000000000400a04 in main (argc=1, argv=0x7ffe6cfd0d88) at gdb_mpi.cpp:15
```

sleep 状態にあるので、main 関数から sleep が、sleep から nanosleep が呼ばれていることがわかる。ここから main に戻ろう。 finish を二回入力する。

```
(gdb) finish
Run till exit from #0  0x00007fc229e2156d in nanosleep () from /lib64/libc.so.6
0x00007fc229e21404 in sleep () from /lib64/libc.so.6
(gdb) finish
Run till exit from #0  0x00007fc229e21404 in sleep () from /lib64/libc.so.6
main (argc=1, argv=0x7ffe6cfd0d88) at gdb_mpi.cpp:14
14    while (i == rank) {
```

main 関数まで戻ってきた。この後、各ランク番号 rank の総和を、変数 sum に入力するので、sum にウォッチポイントを設定しよう。

```
(gdb) watch sum
Hardware watchpoint 1: sum
```

現在は変数 i の値が 0 で、このままでは無限ループするので、変数の値を書き換えてから続行 (continue) してやる。

```
(gdb) set var i = 1
(gdb) c
Continuing.
Hardware watchpoint 1: sum

Old value = 0
New value = 1
0x00007fc229eaa676 in __memcpy_ssse3 () from /lib64/libc.so.6
```

ウォッチポイントにひっかかった。この状態でバックトレースを表示してみよう。

```
(gdb) bt
#0  0x00007fc229eaa676 in __memcpy_ssse3 () from /lib64/libc.so.6
#1  0x00007fc229820185 in opal_convertor_unpack ()
   from /opt/openmpi-2.1.1_gcc-4.8.5/lib/libopen-pal.so.20
#2  0x00007fc21e9afbdf in mca_pml_ob1_recv_frag_callback_match ()
   from /opt/openmpi-2.1.1_gcc-4.8.5/lib/openmpi/mca_pml_ob1.so
#3  0x00007fc21edca942 in mca_btl_vader_poll_handle_frag ()
   from /opt/openmpi-2.1.1_gcc-4.8.5/lib/openmpi/mca_btl_vader.so
#4  0x00007fc21edcaba7 in mca_btl_vader_component_progress ()
   from /opt/openmpi-2.1.1_gcc-4.8.5/lib/openmpi/mca_btl_vader.so
#5  0x00007fc229810b6c in opal_progress ()
   from /opt/openmpi-2.1.1_gcc-4.8.5/lib/libopen-pal.so.20
#6  0x00007fc22ac244b5 in ompi_request_default_wait_all ()
   from /opt/openmpi-2.1.1_gcc-4.8.5/lib/libmpi.so.20
#7  0x00007fc22ac68955 in ompi_coll_base_allreduce_intra_recursivedoubling ()
```

```
   from /opt/openmpi-2.1.1_gcc-4.8.5/lib/libmpi.so.20
#8  0x00007fc22ac34633 in PMPI_Allreduce ()
   from /opt/openmpi-2.1.1_gcc-4.8.5/lib/libmpi.so.20
#9  0x0000000000400a2c in main (argc=1, argv=0x7ffe6cfd0d88) at gdb_mpi.cpp:17
```

ごちゃごちゃっと関数呼び出しが連なってくる。 MPI は規格であり、様々な実装があるが、今表示されているのは Open MPI の実装である。内部で `ompi_coll_base_allreduce_intra_recursivedoubling` とか、それっぽい関数が呼ばれていることがわかるであろう。興味のある人は、OpenMPI のソースをダウンロードして、上記と突き合わせてみると楽しいかもしれない。

さて、続行してみよう。二回 continue するとプログラムが終了する。

```
(gdb) c
Continuing.
Hardware watchpoint 1: sum

Old value = 1
New value = 6
0x00007fc229eaa676 in __memcpy_ssse3 () from /lib64/libc.so.6
(gdb) c
Continuing.
[Thread 0x7fc227481700 (LWP 3648) exited]
[Thread 0x7fc226c80700 (LWP 3649) exited]

Watchpoint 1 deleted because the program has left the block in
which its expression is valid.
0x00007fc229d7e445 in __libc_start_main () from /lib64/libc.so.6
```

`mpirun` を実行していた端末も、以下のような表示をして終了するはずである。

```
$ mpic++ -g gdb_mpi.cpp
$ mpirun -np 4 ./a.out
Rank 2: PID 3646
Rank 0: PID 3644
Rank 1: PID 3645
Rank 3: PID 3647
6
6
6
6
```

ここでは gdb で MPI プロセスにアタッチするやり方だけを説明した。ここを読むような人は gdb を使いこなしているであろうから、アタッチの仕方さえわかれば、後は好きなようにデバッグできるであろう。ただし、私の経験では、並列プログラミングにおいて gdb を使ったデバッグは最終手段であり、できるだけ細かくきちんとテストを書いて、そもそもバグが入らないようにしていくことが望ましい。