# Day 3 : 自明並列

## 自明並列、またの名を馬鹿パラとは

例えば、100 個の画像データがあるが、それらを全部リサイズしたい、といったタスクを考える。それぞれのタスクには依存関係が全くないので、全部同時に実行してもなんの問題もない。したがって、100 並列で実行すれば 100 倍早くなる。このように、並列タスク間で依存関係や情報のやりとりが発生しない並列化のことを自明並列と呼ぶ。英語では、Trivial Parallelization(自明並列) とか、Embarrassingly parallel(馬鹿パラ) などと表現される。「馬鹿パラ」とは「馬鹿でもできる並列化」の略で (諸説あり)、その名の通り簡単に並列化できるため、文字通り馬鹿にされることも多いのだが、並列化効率が 100%であり、最も効率的に計算資源を利用していることになるため、その意義は大きい。なにはなくとも、まず馬鹿パラができないことには非自明並列もできないわけだし、馬鹿パラができるだけでも、できない人に比べて圧倒的な攻撃力を持つことになる。ここでは、まず馬鹿パラのやり方を見てみよう。

## 自明並列の例 1: 円周率

まず、自明並列でよく出てくる例として、サンプリング数を並列化で稼ぐ方法を見てみよう。とりあえず定番の、モンテカルロ法で円周率を計算してみる。

こんなコードを書いて、calc_pi.cpp という名前で保存してみよう。

```
#include <cstdio>
#include <random>
#include <algorithm>

const int TRIAL = 100000;

double calc_pi(const int seed) {
  std::mt19937 mt(seed);
  std::uniform_real_distribution<double> ud(0.0, 1.0);
  int n = 0;
  for (int i = 0; i < TRIAL; i++) {
    double x = ud(mt);
    double y = ud(mt);
    if (x * x + y * y < 1.0) n++;
  }
  return 4.0 * static_cast<double>(n) / static_cast<double>(TRIAL);
}

int main(void) {
  double pi = calc_pi(0);
  printf("%f\n", pi);
}
```

ここで、main 関数から呼ばれる関数 calc_pi(const int seed) が、わざとらしく乱数の種だけ受け取る形になっていることに注意。

普通にコンパイル、実行してみる。

```
$ g++ calc_pi.cpp
$ ./a.out
3.145000
```

100000 回の試行の結果として、円周率の推定値「3.145000」が得られた。これを並列化してみよう。並列化手順は簡単である。

1. mpi.h をインクルードする
2. main 関数の最初と最後に MPI_Init と、MPI_Finalize をつける。ただし MPI_Init が引数に argc と argv を要求するので、main 関数の引数をちゃんと書く。
3. MPI_Comm_rank 関数で、ランク番号を得る。
4. ランク番号を乱数の種に使う
5. そのまま calc_pi を呼ぶ。

以上の修正をしたコードを calc_pi_mpi.cpp という名前で作成する。

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

const int TRIAL = 100000;

double calc_pi(const int seed) {
  std::mt19937 mt(seed);
  std::uniform_real_distribution<double> ud(0.0, 1.0);
  int n = 0;
  for (int i = 0; i < TRIAL; i++) {
    double x = ud(mt);
    double y = ud(mt);
    if (x * x + y * y < 1.0) n++;
  }
  return 4.0 * static_cast<double>(n) / static_cast<double>(TRIAL);
}

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  double pi = calc_pi(rank);
  printf("%d: %f\n", rank, pi);
  MPI_Finalize();
}
```

ついでに、円周率の推定値を表示するときに自分のランク番号も表示するようにしてみた。実行結果はこの通り。

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

```
0: 3.145000
1: 3.142160
3: 3.144200
2: 3.146720

$ mpirun -np 4 --oversubscribe ./a.out
0: 3.145000
2: 3.146720
3: 3.144200
1: 3.142160
```

ただし、--oversubscribe は、論理コア以上の並列実行を許可するオプションである。この実行結果から、

1. 実行するたびに出力順序が異なる
2. しかし、同じランクに対応する推定値は変わらない

ことがわかる。

ちゃんと並列計算されているか、time コマンドで調べてみよう。

```
$ ./a.out
0: 3.145000
./a.out  0.04s user 0.01s system 57% cpu 0.086 total

$ time mpirun -np 4 --oversubscribe ./a.out
2: 3.146720
3: 3.144200
1: 3.142160
0: 3.145000
mpirun -np 4 --oversubscribe ./a.out  0.24s user 0.08s system 240% cpu 0.135 total
```

シリアル実行の場合は CPU の利用率が 57%だったのが、4 並列の場合には 240%と、100%を超えたのがわかるだろう。この例では実行が早く終わりすぎて分かりづらいが、TRIAL の値を大きくして実行に時間がかかるようにして、実行中に top してみると、ちゃんと並列実行されていることがわかる。

```
PID    COMMAND      %CPU TIME     #TH   #WQ  #PORT MEM    PURG   CMPRS  PGRP
45163  a.out        92.1 00:12.44 3/1   0    15    2612K  0B     0B     45163
45165  a.out        91.8 00:12.48 3/1   0    15    2620K  0B     0B     45165
45164  a.out        91.5 00:12.42 3/1   0    15    2608K  0B     0B     45164
45162  a.out        89.1 00:12.47 3/1   0    15    2620K  0B     0B     45162
```

4 並列実行したので、45162 から 45165 まで 4 つのプロセスが起動され、実行していることがわかる。このように、なにか統計平均を取りたい時、並列化によってサンプル数を稼ぐ並列化を**サンプル並列**と呼ぶ。

## 自明並列テンプレート

先程の並列プログラム calc_pi_mpi.cpp の main 関数はこうなっていた。

```
int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
```

```
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  double pi = calc_pi(rank);
  printf("%d: %f\n", rank, pi);
  MPI_Finalize();
}
```

実際のプログラムは `calc_pi(rank)` という関数だけで、これはランク (MPI プロセスの通し番号) を受け取って、その番号によって動作を変える関数である。したがって、自明並列プログラムのテンプレートはこんな感じになる。

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

void func(const int rank){
  // ここを埋める
}

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  func(rank);
  MPI_Finalize();
}
```

後は関数 `func` の中身を書き換えるだけで、なんでも並列化できる。ファイル処理でもレンダリングでも機械学習でもなんでも。「これくらい、MPI 使わなくてもスレッドでもできる」とか言う人がいるかもしれない。しかし、OpenMP や `std::thread` を使ったマルチスレッドプログラミングと、MPI を用いたマルチプロセスプログラミングには、「ノードをまたげるかまたげないか」という大きな違いが存在する。一般に**マルチスレッドプログラミングはノードをまたぐことができない**。したがって、一つのプログラムが専有できる計算資源は 1 ノードまでである。しかし、MPI を使った場合は何ノードでも使える。先程の円周率のコードは、あなたが望むなら数万ノードで実行することだってできる。つまり、このコードがかけた時点で、誰がなんと言おうとあなたはスパコンプログラマだ。「一週間でなれる！スパコンプログラマ」と題した本稿だが、三日目にしてもうスパコンプログラマになることができた。

## 自明並列の例 2: 多数のファイル処理

自明並列の例として、大量のファイル処理を考えよう。たとえば一つあたり 5 分で終わるファイルが 1000 個あったとする。普通にやれば 5000 分かかる。手元に 8 コアのマシンがあり、うまく並列化できたとしても 625 分。 10 時間以上かかってしまうこういう時、手元に MPI 並列ができるスパコンなりクラスタがあれば、あっという間に処理ができる。例えば 8 コアの CPU が 2 ソケット載ったノードが 10 ノード使えるとしよう。うまく並列化できれば 30 分ちょっとで終わってしまう。こういう「大量のファイル処理」は、スパコン使いでなくてもよく出てくるシチュエーションなので、自明並列で対応できるようにしたい。

さて、簡単のため、ファイルが連番で `file000.dat`…`file999.dat` となっているとしよう。これを N 並列する時には、とりあえず N で割ってあまりが自分のランク番号と同じやつを処理すればよい。このように、

異なる情報をバラバラに処理する並列化を**パラメタ並列**と呼ぶ。例えば 100 個のファイルを 16 並列で処理する場合にはこんな感じになるだろう。

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

void process_file(const int index, const int rank) {
  printf("Rank=%03d File=%03d\n", rank, index);
}

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
  int rank;
  const int procs = 16;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  const int max_file = 100;
  for (int i = rank; i < max_file; i += procs) {
    process_file(i, rank);
  }
  MPI_Finalize();
}
```

さて、ファイル数はともかく、プロセス数がハードコーディングされているのが気になる。 MPI のプログラムは、実行時にプロセス数を自由に指定することができる。実行するプロセス数を変えるたびにコンパイルし直すのは面倒だ。というわけで、実行時に総プロセス数を取得する関数 `MPI_Comm_size` が用意されている。使い方は `MPI_Comm_rank` と同じで、

```
int procs;
MPI_Comm_size(MPI_COMM_WORLD, &procs)
```

とすれば、`procs` にプロセス数が入る。これを使うと、先程のコードはこんな感じにかける。

processfiles.cpp

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

void process_file(const int index, const int rank) {
  printf("Rank=%03d File=%03d\n", rank, index);
}

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
  int rank;
  int procs;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &procs);
  const int max_file = 100;
  for (int i = rank; i < max_file; i += procs) {
    process_file(i, rank);
```

```
  }
  MPI_Finalize();
}
```

あとは process_file の中身を適当に書けばファイル処理がノードをまたいで並列化される。ノードをまたがなくて良い、つまり共有メモリの範囲内で良いのなら、MPI でプログラムを書かなくても例えば makefile の並列処理機能を使って処理することもできる。しつこいが、MPI を使うメリットは並列プログラムがノードをまたぐことができることにある。

## 自明並列の例 3: 統計処理

最初に、馬鹿パラで円周率を求めた。 N 並列なら N 個の円周率の推定値が出てくるので、後でそれを統計処理すれば良い。しかし、せっかくなので統計処理も MPI でやってしまおう。各プロセスで円周率の推定値 $x_i$ が求まったとする。平均値は

$$\bar{x} = \frac{1}{N}\sum x_i$$

と求まる。また、不偏分散 $\sigma^2$ は

$$\sigma^2 = \frac{1}{n-1}\sum (x_i)^2 - \frac{n}{n-1}\bar{x}^2$$

である。以上から、円周率の推定値 $x_i$ の総和と、$x_i^2$ の総和が求められれば、期待値と標準偏差が求められる。

MPI における総和演算は MPI_Allreduce 関数で実行できる。

```
double pi =  calc_pi(rank);
double pi_sum = 0.0;
MPI_Allreduce(&pi, &pi_sum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

第一引数から、「和を取りたい変数」「和を受け取りたい変数」「変数の数」「変数の型」「やりたい演算」「コミュニケータ」の順番で指定する。ここでは一つの変数のみ総和演算を行っているが、配列を渡して一気に複数のデータについて総和を取ることもできる。また、総和だけでなく積や論理演算も実行できる。

円周率の推定値 pi と、その自乗 pi2 = pi*pi について総和を取り、定義通りに期待値と標準偏差を求めるコードが calc_pi_reduce.cpp である。

```
#include <cstdio>
#include <random>
#include <algorithm>
#include <cmath>
#include <mpi.h>

const int TRIAL = 100000;

double calc_pi(const int seed) {
  std::mt19937 mt(seed);
  std::uniform_real_distribution<double> ud(0.0, 1.0);
```

```
  int n = 0;
  for (int i = 0; i < TRIAL; i++) {
    double x = ud(mt);
    double y = ud(mt);
    if (x * x + y * y < 1.0) n++;
  }
  return 4.0 * static_cast<double>(n) / static_cast<double>(TRIAL);
}

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
  int rank;
  int procs;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &procs);
  double pi = calc_pi(rank);
  double pi2 = pi * pi;
  double pi_sum = 0.0;
  double pi2_sum = 0.0;
  printf("%f\n", pi);
  MPI_Allreduce(&pi, &pi_sum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  MPI_Allreduce(&pi2, &pi2_sum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  double pi_ave = pi_sum / procs;
  double pi_var = pi2_sum / (procs - 1) - pi_sum * pi_sum / procs / (procs - 1);
  double pi_stdev = sqrt(pi_var);
  MPI_Barrier(MPI_COMM_WORLD);
  if (rank == 0) {
    printf("pi = %f +- %f\n", pi_ave, pi_stdev);
  }
  MPI_Finalize();
}
```

最後に呼んでいる MPI_Barrier は、「ここで全プロセス待ち合わせをしなさい」という命令である。MPI_Allreduce は全プロセスで情報を共有するが、一番最後にランク 0 番が代表して統計情報を表示している。以下が実行例である。

```
$ mpic++ calc_pi_reduce.cpp
$ mpirun --oversubscribe -np 4  ./a.out
3.144200
3.142160
3.146720
3.145000
pi = 3.144520 +- 0.001892

$ mpirun --oversubscribe -np 8  ./a.out
3.145000
3.142160
3.144200
```

```
3.144080
3.139560
3.146720
3.139320
3.136040
pi = 3.142135 +- 0.003565
```

4 並列では 4 つの推定値、8 並列では 8 つの推定値が出てきて、最後に平均と標準偏差が表示されている。各自、エクセルか Google Spreadsheet に値を突っ込んでみて、平均と標準偏差が正しく計算できていることを確かめられたい。ちなみに、並列数が多いほうが標準偏差が小さくなることが期待されるが、乱数の初期値の与え方が悪かったのか、データが偏ってしまった。そのあたりが気になる方は、適当に修正してほしい。

## 並列化効率

並列化した際、並列化しなかった場合に比べてどれくらい効果的に並列化されたかを知りたくなる。その効率を示すのが並列化効率である。いや、別にあなたが知りたくなくても、並列化していると「並列化効率は？」と聞いてくる人は絶対に出てくる。個人的には、最初は並列化効率とかあまり気にせずに楽しくやるのが良いと思うのだが、一応並列化効率についての知識もあったほうが良いだろう。

さて、並列化は、計算資源をたくさん突っ込むことで、同じタスクをより早く終わらせるか、より大きなタスクを実行することである。計算資源を増やした時に、どれだけ効率が良くなかったを示す指標を「スケーリング」という。N 倍の計算資源を突っ込んだ時に、同じタスクは理想的には 1/N の時間で終わってほしい。このようなスケーリングを「ストロングスケーリング」と呼ぶ。逆に、計算資源あたりのタスク (計算規模) を固定した場合、N 倍の計算資源を突っ込んでも、同じ時間で計算が終わってほしい。このようなスケーリングを「ウィークスケーリング」と呼ぶ。一般に、ストロングスケーリングの効率を上げる方がウィークスケーリングの効率を上げるより難しい。

まず、ストロングスケーリングの効率を定義しよう。並列化単位はプロセスでもスレッドでも良いが、とりあえずプロセス並列を考えることにする (スレッド並列でも全く同様に並列化効率を定義できる)。並列化前、つまり 1 プロセスで実行した時、あるタスクが $T_1$ の時間で終わったとしよう。同じサイズのタスクを、$N$ プロセスで計算して、計算時間が $T_N$ になったとする。この時、並列化効率 $\alpha$ は

$$\alpha = \frac{T_1}{NT_N}$$

で定義される。例えば、1 プロセスで 10 秒で終わるジョブが、10 プロセスで 1 秒で終われば並列化効率 1(つまり 100%)、10 プロセスで 2 秒で終わったら、並列化効率 0.5(つまり 50%) である。

ウィークスケーリングでは、「プロセスあたり」のタスクを固定する。したがって、並列数を増やす時、解くべきタスクのサイズも比例して大きくする。例えばタスクのサイズを 10 倍にして、並列数も 10 倍にした場合、理想的には計算時間が変わらないで欲しい。なので、そういう場合に並列化効率が 1 となるように定義する。あるタスクを 1 プロセスで実行した時、あるタスクが $T_1$ の時間で終わったとしよう。その $N$ 倍のサイズのタスクを、$N$ プロセスで計算した時に、計算時間が $T_N$ となったとする。この時、並列化効率 $\alpha$ は

$$\alpha = \frac{T_1}{T_N}$$

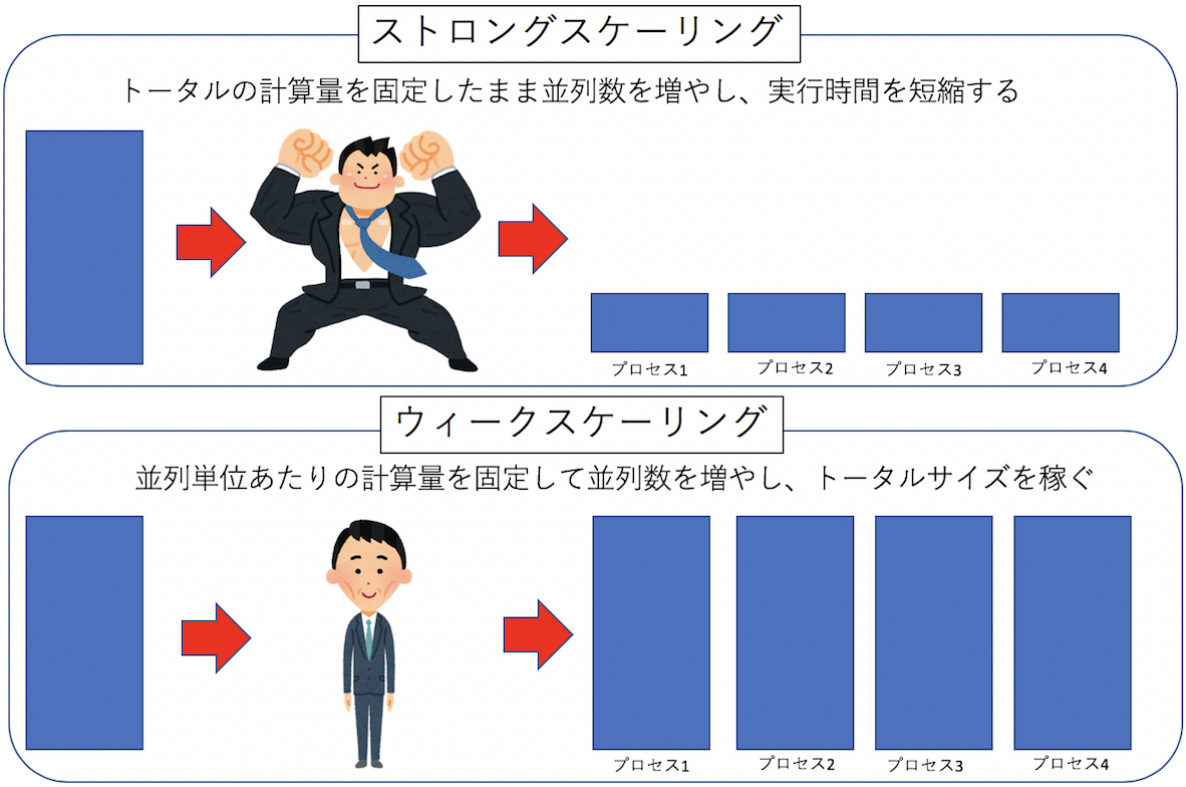


図 1: fig/scaling.png

で定義される。例えば、1 プロセスで 12 秒で終わったタスクがあったとして、10 倍のタスクを 10 プロセスで計算した場合に 16 秒かかったとすると、並列化効率は $12/16 = 0.75$、つまり 75%となる。

まぁ要するに並列化した後の実行時間 $T_N$ が小さいほど嬉しい (＝並列化効率が高い) のであるから、$T_N$ が分母にある、とおぼえておけば間違いない。ストロングスケーリングにはファクター $N$ がつき、ウィークスケーリングにはファクターがつかないが、これも理想的な状況を考えればどっちがどっちだかすぐ思い出せるであろう。

## サンプル並列とパラメタ並列の違い

Day3 では、馬鹿パラとして「サンプル並列」と「パラメタ並列」を紹介した。「サンプル並列」とは、計算資源を統計平均を稼ぐのに利用する並列化で、「パラメタ並列」とは、計算資源を異なるパラメータを処理するのに利用する並列化である。ここでは「パラメタ並列」としてファイル処理の例をあげたが、例えばなにかの温度依存性を調べたい時、温度をパラメータとして、温度 10 点を 10 並列する、みたいな計算が典型的なパラメタ並列である。

「サンプル並列」も「パラメタ並列」もウィークスケーリングに属し、それぞれのタスクの計算時間に大きなばらつきがあったりしなければ、かなり大規模な計算をしても理想的な並列化効率が出せる。ただし、「パラメタ並列」は、100 倍の計算資源を突っ込めば 100 倍のパラメータを処理できるのに対して、「サンプル並列」は 100 倍の計算資源を突っ込んでも、統計精度は 10 倍にしかならない。これは、サンプリングによる精度の向上が、サンプル数の平方根でしか増えないからだ。したがって、いかに並列化効率が 100%に近くても、サンプル平均に大規模な計算資源を突っ込むのはあまり効率がよくない。なにかの精度を一桁上げたいからといって 100 倍の計算資源を突っ込む前に、もう少し効率的な方法がないか調べた方が良い。

また、パラメタ並列でも、複数のパラメタを組み合わせると、調べるべき点数が非常に増える。この時、それらすべてを、詳細にパラメタ並列で調べるのはあまり効率がよくない。ただ、とりあえず雑に「あたり」をつけるのにパラメタ並列は向いているので、それでだいたい調べるべき場所を確認した後、なにか別の方法でそこを詳細に調べるのが良いであろう。