

一週間でなれる！スパコンプログラマ

リポジトリ (kaityo256/sevendayshpc)

HTML 版

一括 PDF 版

はじめに

世の中にはスーパーコンピューター、略してスパコンというものがある。こういうすごそうな名前があるものの例にもれず、「スパコンとはなにか」という定義は曖昧である。人によって「何がスパコンか」の定義は違うと思うが、とりあえずここではCPUとメモリを積んだ「ノード」がたくさんあり、それらが高速なネットワークでつながっていて、大きなファイルシステムを持っているもの、と思っておけば良いと思う。

スパコンは名前に「スーパー」とついているだけに、「なんかすごそうなもの」「使うのが難しいもの」という印象を持つ人もいるだろう。しかし、スパコンを使うのに要求される技術そのものは非常に単純かつ簡単である。自分の経験として、プログラミングの素養がある学生であれば、詳しい人に一ヶ月もレクチャーを受ければ普通にスパコンにジョブを投げ始める。そのくらいスパコンを使うのは簡単なことである。しかし、スパコンは、「使いはじめる」のは簡単であるものの、「使い倒す」のはかなり難しい。経験的に、並列数が十進法で一桁増えるごとに、本質的に異なる難しさに直面する。例えば百並列で走ったコードが千並列で走らなかったり、千並列で走ったコードが一万並列でコケたりする。そのあたりの奥の深さは面白いものの、本稿では扱わない。

この記事では、近くにスパコンに詳しい人がいない人のために、「とりあえず7日間でスパコン使えるようになる」ことを目指す。より正確には、「7日間程度ければ、誰でもスパコンプログラマになれるそうだな」と思ってもらうことを目指す。

Day 0：なぜスパコンを使うのか

そもそも、なぜスパコンを使うのか？それは、そこにスパコンがあるからだ。この日本語で書かれた文章を読んでいるということは、あなたは高確率で日本人であろう。おめでとう。あなたは世界有数のスパコンを使うことができる。なぜなら日本はスパコン大国だからだ。Top500のサイトに行くと、世界の性能トップ500に入るスパコンの、国の内訳を見ることができる。2018年6月時点では、トップは中国の206サイト、二位がアメリカの124サイト、日本は36サイトで三位に入っている。最近の中国の躍進は目覚ましいのだが、そこはさておく。Top500に入るスパコン数は世界三位で、しかも何度も世界一位となるスパコンを保持している日本は、世界有数のスパコン大国と言ってよいだろう。

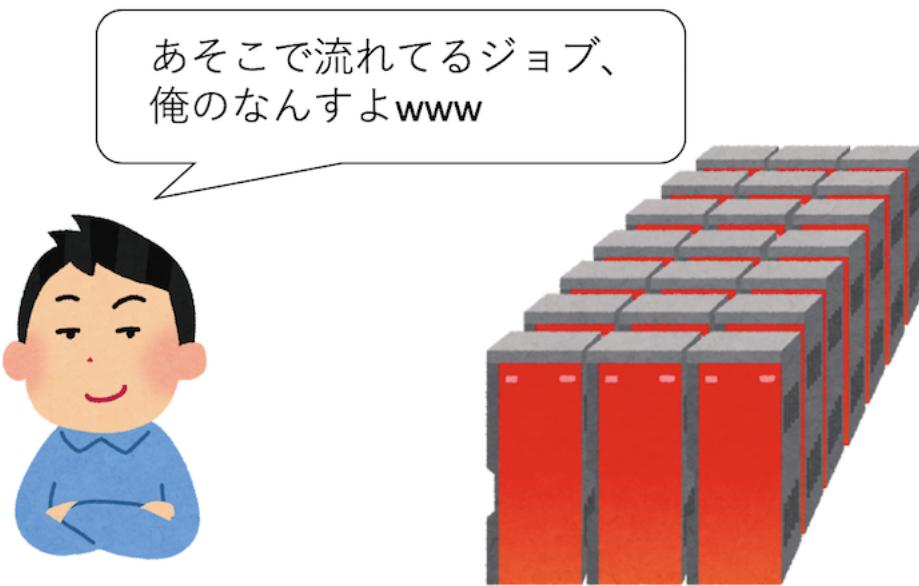
個人的な経験で、こんなことがあった。とある海外の方と共同研究をしていた時、共同研究者に「こんな計算をしてみたらどうだろう」と提案してみた。すると彼に「そういうことができたら面白いとは思うけど、計算が重すぎて無理だよ」と言われたので「いや、うちのスパコンでやったら一日仕事だ」と言ったらえらく驚いていた。これは、単に「日本は計算資源に恵まれている」という話にとどまらず、人の想像力の上限は、普段使っている計算資源の規模で決まるということを示唆する。これは極めて重要なと思う。

普通、スパコンを使うのは、「まずローカルなPCで計算をして、それで計算が苦しくなってから、次のステップアップとしてスパコンを検討する」といった順番となるだろう。その時も、まず「これくらいの規模のスパコンを使ったら、これくらいの計算ができる」という事前の検討をしてからスパコンの利用申請をするであろう。つまり「テーマ(目的)が先、スパコン(手段)が後」となる。それは全くもって正しいのであるが、私個人の意見としては、「やることが決まってなくても、どのくらいの計算が必要かわからなく

ても、まずスパコンを使ってしまう」方がいろいろ良いと思う。普段からローカルPCでしか計算していない人は、なにか研究テーマを検討する際に、スパコンが必要となるテーマを無意識に却下してしまう。逆に、普段からスパコンを使いなれないと、想像力の上限が引き上げられ、普通のPCでは計算できないような選択肢を検討できる。つまり「スパコン(手段)が先、テーマ(目的)は後」である。そもそもスパコンを使うのはさほど難しくないのだし、いろいろ検討する前に、ささっと使い始めてみよう。

注意

並列プログラミングに限らないことだが、なにかを始めようとすると、ちょっと先にそれを始めていた人がなんやかんや言ってくることだろう。「並列化効率ガード」とか、「そもそも実行効率が悪いコードを並列化するなんて云々」とか「最初から通信の最適化を考慮云々」とか、そういうことを言ってくる人が必ず湧くが、とりあえず二年くらいは無視してかまわない。なにはともあれスパコンを使えるようになること、チューニング不足の遅いコードであろうが並列化効率が悪かろうが、とりあえずそれなりのノード数で走るコードを書いて実行してみること、まずはそこを目指そう。それなりのノード数で走るコードが書ける、それだけで十分強力な武器になる。



スパコンが使えると楽しい

図 1: day1/fig/myjob.png

Day 1 : 環境構築

とりあえず手元のPCでMPIが使える環境を整え、簡単なMPIプログラミングを試してみる。

- MPIとは
- 余談：MPIは難しいか
- MPIのインストール
- はじめてのMPI
- ランク
- 標準出力について

- GDB による MPI プログラムのデバッグ

Day 2 : スパコンの使い方

スパコンを使うときに知っておきたいこと。ジョブの投げ方など。

- はじめに
- スパコンとは
- 余談 : BlueGene/L のメモリエラー
- スパコンのアカウントの取得方法
- ジョブの実行の仕組み
- ジョブスクリプトの書き方
- フェアシェア
- バックフィル
- チェーンジョブ
- ステージング
- 並列ファイルシステム

Day 3 : 自明並列

自明並列、通称「馬鹿パラ」のやり方について。

- 自明並列、またの名を馬鹿パラとは
- 自明並列の例 1: 円周率
- 自明並列テンプレート
- 自明並列の例 2: 多数のファイル処理
- 自明並列の例 3: 統計処理
- 並列化効率
- サンプル並列とパラメタ並列の違い

Day 4 : 領域分割による非自明並列

非自明並列の例として、一次元熱伝導方程式方程式を領域分割してみる。

- 非自明並列
- 一次元拡散方程式 (シリアル版)
- 一次元拡散方程式 (並列版)
- 余談 : Eager プロトコルと Rendezvous プロトコル

Day 5 : 二次元反応拡散方程式

本格的な MPI プログラムの例として、二次元反応拡散方程式を領域分割してみる。

- シリアル版
- 並列化ステップ 1: 通信の準備など
- 並列化ステップ 2: データの保存
- 並列化ステップ 2: のりしろの通信
- 並列化ステップ 3: 並列コードの実装
- 余談 : MPI の面倒くささ

Day 6 : ハイブリッド並列

プロセス並列とスレッド並列の併用によるハイブリッド並列化について。特にスレッド並列で気をつけたいことなど。

- ハイブリッド並列とは
- 仮想メモリと TLB
- 余談：TLB ミスについて
- NUMA
- OpenMP の例
- 性能評価
- 余談：ロックの話
- ハイブリッド並列の実例

Day 7 : SIMD 化

SIMD 化について。

- はじめに
- SIMD とは
- SIMD レジスタを触ってみる
- 余談：アセンブリ言語？アセンブラ言語？
- 簡単な SIMD 化の例
- 余談：x86 における浮動小数点演算の扱い
- もう少し実戦的な SIMD 化

おわりに

謝辞

この記事は、tanakamura さんの実践的低レベルプログラミングに影響されて書き始めたものです。 angel_p_57 さんに MPI におけるバッファリングについて教えていただきました。 fujita_d_h さんには BlueGene/L の L1 エラー訂正について議論していただきました。 n_IMRC さんには、行列積における TLB ミスの論文を教えていただきました。まだこの記事が書きかけだったときにたくさん星をつけてくださった皆様、Twitter などで感想を寄せてくださった皆様、ありがとうございます。みなさんのポジティブな反応がなければ書き続けられませんでした。

本稿を読んで「スパコンを使ってみよう」と思う人が一人でも増えたなら幸いです。

ライセンス

Copyright (C) 2018-2019 Hiroshi Watanabe

この文章と絵 (pptx ファイルを含む) はクリエイティブ・コモンズ 4.0 表示 (CC-BY 4.0) で提供する。

This article and pictures are licensed under a Creative Commons Attribution 4.0 International License.

本リポジトリに含まれるプログラムは、MIT ライセンスで提供する。

The source codes in this repository are licensed under the MIT License.

なお、HTML 版の作成に際し、CSS として github-markdown-css を利用しています。

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 コアを使いにいくことである。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。コンパイラー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。

オペレーティング・システム (OS)

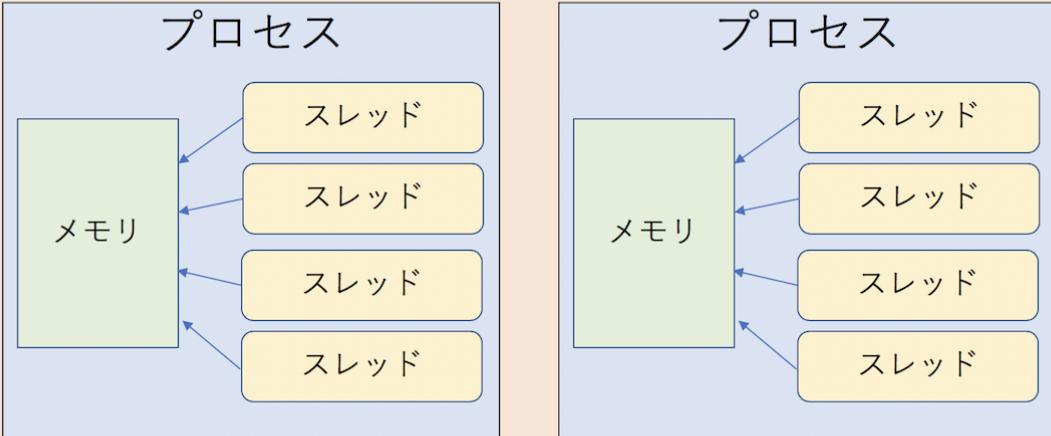


図 1: fig/process_thread.png

```
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しま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。

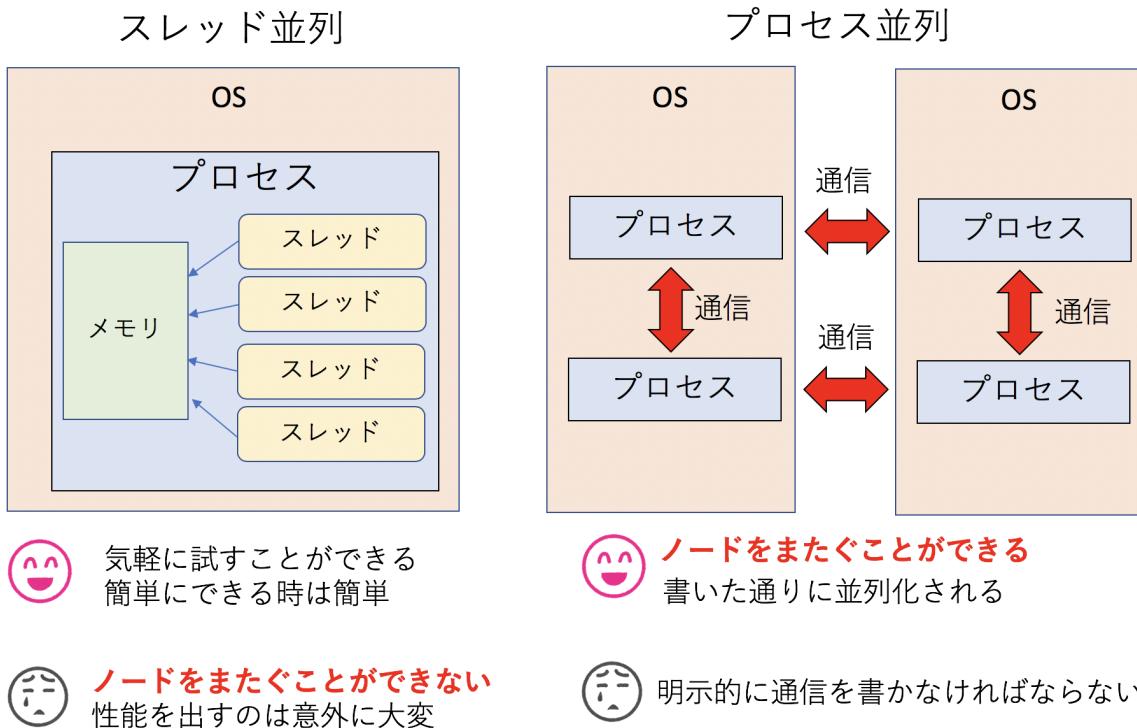


図 2: fig/comparison.png

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ー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、まあ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 <iostream>
#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++` はラップに過ぎず、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、

```

#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。まあ、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さ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 0x00000000000400a04 in main (argc=1, argv=0x7ffe6cf0d88) 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=0x7ffe6cf0d88) 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 0x00007fc21e9afbd 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=0x7ffe6cf0d88) 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。

Day 2 : スパコンの使い方

はじめに

スパコンを使うのに、必ずしもスパコンがどのように構成されているかを知る必要はない。しかし、せっかくスパコンを使うのだから、スパコンとは何かについて簡単に知っておいても良いであろう。ただし、こういう単語にありがちだが「何がスパコンか」は人によって大きく異なる。ここで紹介するのはあくまで「執筆者が思うスパコンの定義」の説明であり、他の人は他の定義があることを承知されたい。ここは、「読むとなにかができるようになる」というよりは、「スパコンを使ったことがない人が、将来スパコンを使うにあたって知っておくと良さうこと」を書いておく。特に手を動かすところはない。読み物として流して読んでいただければ良い。

スパコンとは

普通の PC は、CPU、メモリ、ネットワーク、ディスクなどから構成されている。スパコンも全く同様に、CPU、メモリ、ネットワーク、ディスクがある。それぞれちょっと高級品を使っているだけで、基本的には普通の PC と同じと思って良い。ただし、PC とはつなぎ方がちょっと異なる。スパコンは、CPU とメモリをまとめたものを「ノード」と呼ぶ。このノードをたくさん集めて高速なネットワークでつないだものがスパコン本体である。普通の PC では CPU の近くにディスクがあるが、最近のスパコンのノードはディスクレスの構成にすることが多い。そのかわり、大きなファイルシステムとネットワークでつなぐ。

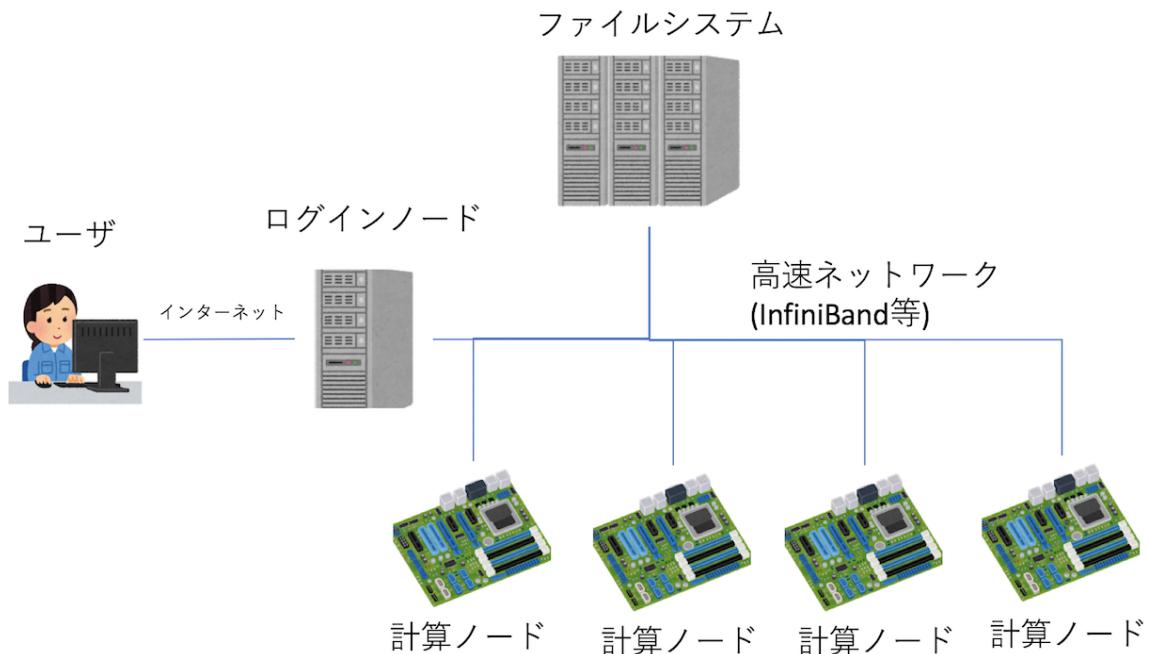


図 1: fig/kousei.png

一般的なスパコンの構成は上図のような感じになる。ノードは大きく分けて「ログインノード」と「計算ノード」の二種類がある。ユーザはインターネット経由でログインノードにログインし、そこで作業をする。実際の計算は計算ノードで行う。ログインノード、計算ノード、ファイルシステムは、高速なネットワークで接続されている。この高速ネットワークは長らく InfiniBand という規格がデファクトスタンダードとなっていたが、近年になって Intel の OmniPath の採用例も増えているようである。ファイルシステムに

については、Lustre というファイルシステムがデファクトスタンダードであるが、目的によっては GPFS(今は IBM Spectrum Scale に名前が変わったらしい)なども選択される。

さて、「ノード」という呼び方をしているが、これは本質的にはパソコンと同じものなので、スペコンとはパソコンを大量に並べて高速なネットワークで相互につないだもの、と言える。特に最近は CPU として x86 の採用が多いため、ますます「普通のパソコンを大量に並べたもの」という印象が強くなるだろう。しかし、「普通のパソコンを大量に並べたらスペコンになるか」というとそうではない。スペコンで最も重要なものの、それは「信頼性」である。

普通に PC を使っていたら、わりと PC の部品は壊れることを知っているであろう。ありとあらゆる箇所が壊れる可能性があるが、ファンやディスクなどの回るものなどは特に壊れやすい。また、メモリなどもよく壊れるものの代表である。CPU やマザー、ネットワークカードなども壊れる。スペコンは多くの部品から構成されるため、壊れるのが非常に稀であっても、全体としては無視できない確率で壊れてしまう。

例えば「京コンピュータ」が 10 ペタフロップスを達成した時には、計算の実行に 88128 ノードを使って 29 時間 28 分かかっている。これは約 260 万ノード時間であるから、この計算が 90% の確率で実行できるためには、各ノードが 3000 年くらいは壊れない「保証」が必要になる。逆に、もし各ノードが 1 年に一度くらい壊れるならば、およそ 6 分に一度どこかのノードが壊れることになり、とても使い物にならない。

大規模計算は大縄跳び



一人でもこけたら全体が失敗してしまう

図 2: fig/nawatobi.png

つまり、大規模計算とは大縄跳びのようなものであり、それなりのノード数でまとめて計算を行うためには、かなり高信頼なシステムを組む必要があることがわかるであろう。他にも様々な「スペコンらしさ」「スペコンならではの工夫」はあるのだが、なにはともあれ「高信頼であること」が非常に重要である。たまにゲーム機や格安チップを並べて「格安スペコン」を作った、というニュースが話題になるのだが、実はスペコンでは計算部分だけでなく「信頼性」と「ネットワーク」にかなりコストがかかっており、そこをケチると、当然ながら「信頼性」や「ネットワーク」が必要とされる計算ができないスペコンになる。

もちろん「やりたい計算」が「格安スパコン」でできるのであればそれを選択すれば良いが、ただピーク性能と値段を比較して「スパコンは高い」と即断しないようにしてほしい。星の王子さまも「いちばんたいせつなことは、目に見えない」って言つてることだしね…。

余談：BlueGene/L のメモリエラー

スパコンは部品が多いために故障が問題になると書いた。他にも、普通の PC ではあまり問題にならないことがスパコンでは問題になることがある。それは宇宙線である。

宇宙線とは、文字通り宇宙から飛んでくる放射線のことである。例えば国立科学博物館などに行った際には、霧箱展示を見て欲しい。普段意識していないが、宇宙線はわりとびゅんびゅん飛んでいる。これが半導体素子にぶつかることで誤動作を起こす。特に問題となるのはメモリで、宇宙線によりランダムにビット反転が起きてしまい、結果がおかしくなる。この問題を防ぐため、メモリには 1 ビットのエラーは訂正可能、2 ビットのエラーについては検出可能となるようなエラー訂正機能をつけるのが一般的である。

しかし、この機能がついていないスパコンがある。IBM の BlueGene/L である。BlueGene は、比較的非力なノードを多数結合させることで、全体として高い性能と省電力を両立させよう、という設計思想をもったスパコンで、例えば計算ノードの OS がマルチユーザをサポートしないなど、随所に思い切った割り切りが見られる。その中で特に驚くのが、CPU の L1 キャッシュにエラー訂正機能がないことである。BlueGene/L の L1 には、1 ビットのエラー検出機能のみがあり、訂正することができない。したがって、ビットエラーが起きるとそのままシステムがクラッシュする。

当時、BlueGene/L を導入したローレンス・リバモア国立研究所のマニュアルによると、フルノード (20 万コア) で計算すると平均的に 6 時間に一度程度、宇宙線による L1 のビットエラーで計算がクラッシュする、と試算されている。これに対してユーザは、3 つの手段を取ることができた。

- 諦める：フルノードで 6 時間程度なので、例えば 1 万コア使って計算しても、平均故障間隔は 120 時間程度になる。なので、そのまま計算して死んだら諦める、というのは現実的な選択であった。
- メモリ保護モードを使う：BlueGene/L には「write through モード」が用意されていた。これは L1 のビットエラーを L2 や主記憶を使ってソフトウェア的に保護するモードであったらしい。ユーザは何もしなくて良いが、このモードを選択すると 10% から 40% 程度の性能劣化があったようだ。
- 例外を受け取ってユーザ側でなんとかする：L1 がビットエラーの検出をした時、OS が例外を飛ばすモード。ユーザ側がなんとかする。

通常は「諦める」か、性能劣化が許容範囲内であれば write through モードを使っていたようだが、BlueGene/L のフルノードで長時間計算し、2007 年の Gordon Bell 賞を受賞したリバモアのチームは「例外を受け取ってユーザ側でなんとかする」方法を選択した。具体的には、メモリの一部にチェックポイントデータを保持しておき、例外を受け取ったら直近のチェックポイントからリストアするコードを書いたそうだ。論文によると、フルノードを三日間程度の計算中、数回例外を受け取ってリストアしたらしい。このチームのメンバーに話を聞いたことがあるが、プログラムのどの場所で例外が来ても大丈夫なように組むのが大変だったと言っていた記憶がある。

「システムが大きくなるとハードウェアだけで信頼性を担保するのは難しくなるため、一部をソフトウェアでなんとかしよう」というような話は昔から言われており、多くの研究があって実験的にシステムに組み込まれたものもあるのだが、そのような思想が実運用に供されたのは筆者の知る限り BlueGene/L を除いて他はない。

そもそもなぜ L1 にエラー訂正をつけなかったのか、つけられなかったのかはわからないのだが、後継機である BlueGene/P にはどうやら L1 にもエラー訂正がついたところを見ると、ユーザからは不評だったのかもしれない。BlueGene シリーズは HPC 業界ではかなり売れたようで、2007 年 6 月の Top500 リストのトッ

プ10に、BlueGeneが4つ入っている。BlueGeneは、第一世代のBlueGene/L、第二世代のBlueGene/P、第三世代のBlueGene/Qと開発が進められたが、そこでBlueGeneプロジェクトは終了した。

スパコンのアカウントの取得方法

スパコンを使うためには、スパコンのアカウントを手に入れなければならない。ある意味ここが最難関である。スパコンを使う技術そのものは非常に簡単に身につくが、スパコンのアカウントを手に入れる、というは技術に属すスキルではないからだ。

とりあえず、世の中には様々なスパコンがある。企業が開発のためにスパコンを導入していることもあるし、最近はクラウドもスパコンと呼んで良いような計算資源を用意している。将棋ソフト、Ponanzaの開発者の山本さんは機械学習のための計算資源が足りず、さくらインターネットにお願いして大規模な計算資源を借りたそうなのだが、そういう強いメンタルを持っていない人は、通常の手続きでスパコンのアカウントを取得しよう。

まず、スパコンといえば、大型計算機センター(大計センター)である。たとえば東京大学情報基盤センターのようなところは、申請すれば、かなり割安な金額で計算資源を借りることができる。また、様々な公募を行っており、共同研究という形で無料で計算資源を借りることもできるようである。

物性研究所は、テーマが「物性科学」に限られるが、申請が認められれば無料で計算資源を利用できる。ただし、利用資格は修士以上(原則として学部生不可)、申請資格は給料をもらっている研究者(ポスドク以上)なので注意。

「二位じゃだめなんでしょうか?」で話題となった「京コンピュータ」もテーマを一般公募しており、研究目的であるならば、得られた成果を公表することを前提に無料で利用できる。最大8万ノード、64万コアの計算資源が神戸で君を待って…いたのだが、2019年8月30日に運用を終了し、シャットダウンされた。しかし、「京」の次のスパコンの準備が進められており、順調にいけばまた大きな計算資源が稼働するであろう。

研究目的でスパコンを使うのであれば、利用者は修士以上、申請は研究者、というのが一般的であるが、たとえば「高校生だけどスパコンを使いたい」という人もいるであろう。そういう有望な若者のために、例えば阪大や東工大は「SuperCon」という、スパコン甲子園のようなイベントを行っている。このイベントは、競技プログラミングの一種であるが、問題を解くのに並列計算が必須となる規模が要求されるのが特徴である。

また、2014年には「高校生がスーパーコンピュータを使って5×5魔方陣の全解を求めるに成功」している。これは筑波大学の学際共同利用プログラムを利用したもので、当時高校一年生が教授と共同研究を行い、スパコンを使い倒した事例である。

率直に言って、日本はスパコン大国であるわりに、若い人(例えば高校生)が「そうだ、スパコン使おう」と気軽に使える状況はない(この状況はなんとかしたい)。しかし、門戸は狭いながらも高校生に開いているのは確かである。また、そもそも並列プログラミングができなければ、スパコンを使おう、という気持ちにはならないだろう。とりあえずさくっと並列プログラミングができるようになっておき(どうせ簡単なので)、チャンスがあればスパコンを使う、というスタンスでいればいいんじゃないでしょうか。

ジョブの実行の仕組み

ローカルPCは自分しか使わないので、好きな時に好きなプログラムを実行して良い。しかし、スパコンは大勢で共有する計算資源であり、各自が好き勝手にプログラムを実行したら大変なことになるため、なんらかの交通整理が必要となる。その交通整理を行うのが「ジョブスケジューラ」である。スパコンでは、

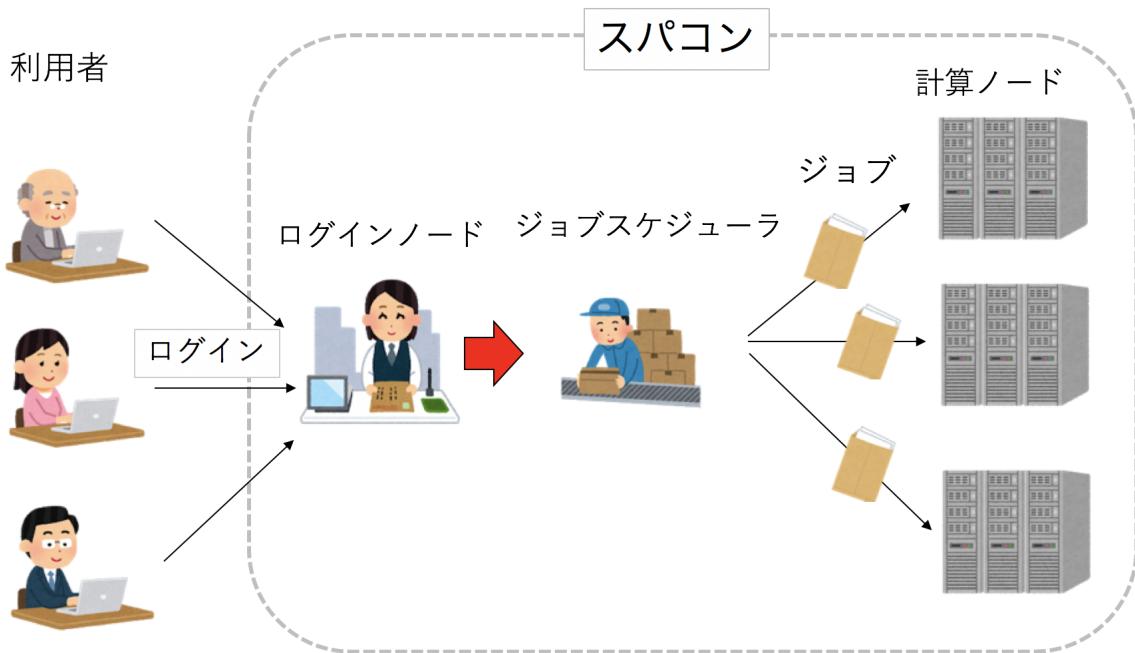


図 3: fig/supercomputer.png

プログラムは「ジョブ」という単位で実行される。ユーザはまず、「ジョブスクリプト」と呼ばれるシェルスクリプトを用意する。これは、自分のプログラムの実行手順を記した手紙のようなものである。次にユーザは、ログインノードからジョブスケジューラにジョブの実行を依頼する(封筒をポストに入れるイメージ)。こうしてジョブは実行待ちリストに入る。ジョブスケジューラは実行待ちのジョブのうち、これまでの利用実績や、要求ノード数、実行時間などを見て、次にどのジョブがどこで実行されるべきか決定する。

ジョブスクリプトの書き方

先に述べたように、スパコンにおいては、プログラムをコンパイルする場所と、実行する場所が異なる。ユーザは、実行ファイルの他に、ジョブスクリプトというファイルを用意し、それをジョブスケジューラに投げることでジョブの実行を依頼する。ジョブスクリプトにはジョブの要求資源と、ジョブの実行の仕方が書いてあり、ジョブスケジューラは要求資源をもとに、そのジョブをいつ、どこで実行するかを決める。そして実行の順番がきたら、ジョブスクリプトはシェルスクリプトとして実行され、それによってジョブが実行される。ちなみに待ち行列に入っているジョブが実行状態になることを「ディスパッチ」と呼ぶ。

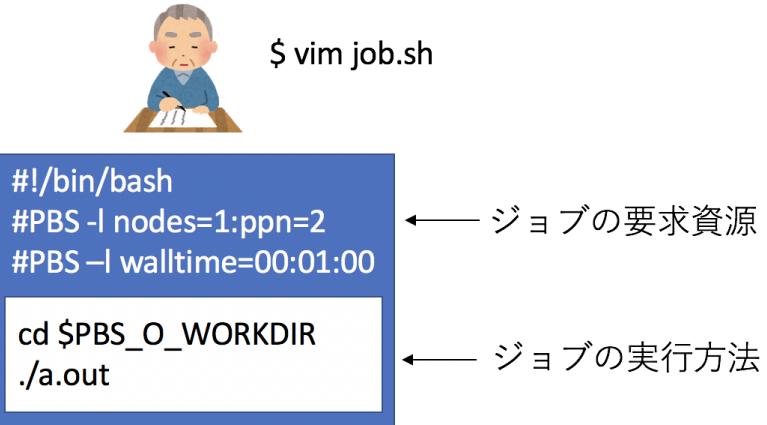
まず、ジョブスクリプトの冒頭に、特殊なコメントを使ってジョブの要求資源を書く。ジョブスクリプトの書き方は、どんなジョブスケジューラを使っているか、そのスパコンサイトがどういう運用をしているかによって異なるので、もしスパコンを使う場合は、スパコンサイトが用意しているであろうマニュアルを適宜参照されたい。しかし、基本的には、ジョブの要求ノード数や、実行時間を書けば良い。

例えば、ジョブスケジューラの一つ、PBSならば、#PBS の後に指示文を書く。2ノード、12時間の実行を要求するなら

```
#PBS -l nodes=2
#PBS -l walltime=24:00:00
```

といった具合である。その次に、ジョブの実行方法を書く。ここで注意して欲しいのは、このプログラムはコンパイル、ジョブ投入をした計算機とは異なる場所で実行されるということだ。したがって、カレン

ジョブスクリプトの用意



ジョブの投入

```
$ qsub job.sh
```



図 4: fig/job.png

トディレクトリや環境変数などは引き継がれない。特に、カレントディレクトリが変わることには注意したい。なので、

```
cd /home/path/to/dir  
./a.out
```

といった感じに絶対パスで書いても良いが、普通は「ジョブ投入時のカレントディレクトリ」が特別な環境変数に入っているのでそれを使う。PBSならば PBS_O_WORKDIR にカレントディレクトリが入っているので、

```
cd $PBS_O_WORKDIR  
./a.out
```

と書くと良いだろう。他にも実行に必要な環境変数などがあれば、それも指定しよう。こうしてできたジョブスクリプトを、例えば go.sh という名前で保存し、

```
qsub go.sh
```

などと、ジョブをサブミットするコマンドに渡してやればジョブ投入完了である。ジョブの状態を見るコマンド (PBSなら qstat) で、ジョブが実行待ちに入ったことを確認しよう。

スペコンによっては、搭載メモリ量が多いノード (Fat ノードなどと呼ばれる) や、GPGPU が搭載されているノードなど、異なる種類のノードから構成されている場合がある。その場合はノードの種類ごとに「キュー」と呼ばれる実行単位が設定されているため、適切なキューを選んで実行すれば良い。

フェアシェア

実行待ちになったジョブは、ジョブスケジューラによって実行予定の場所と時刻が決定される。基本的には、前のジョブが終わった時に、待ち行列の先頭にあるジョブが実行されるのだが、他にも様々な要因がある。その中で重要なのは「フェアシェア」という概念である。

こんな状況を考えてみよう。計算ノードが 4 ノードあるクラスタを考えよう。最初に A さんが 1 ノードジョブを 10 個投げた。4 つのジョブが 4 ノードで実行され、6 つが待ち行列に入った。その後、B さんが

1ノードジョブを1つ投げたとする。この時、普通に FIFO でスケジューリングしてしまうと、Bさんのジョブは待ち行列の最後に入ってしまう。すでに Aさんは4ノードも占有しているのだから、次に走るべきジョブは Bさんのジョブのような気がしてくるであろう。そうするためには、Bさんのジョブが既に待ち行列に入っていたジョブを追い越さなければならない。

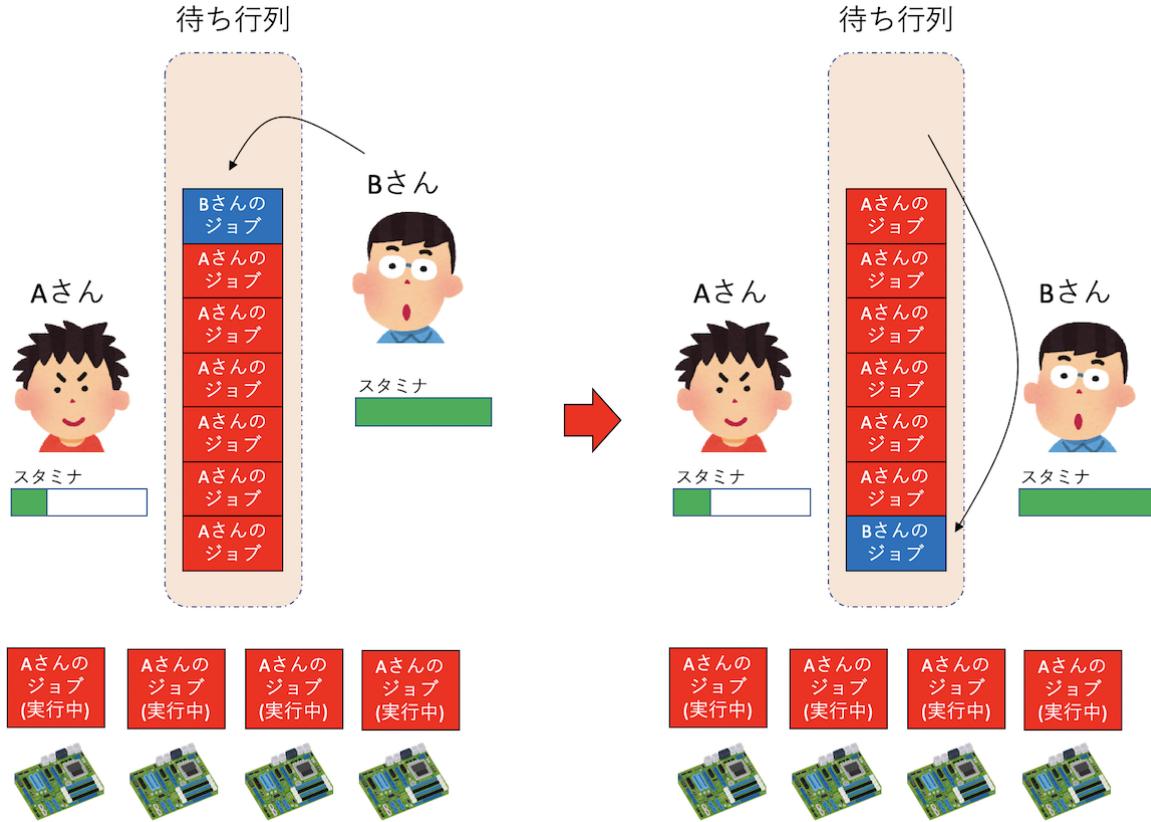


図 5: fig/fairshare.png

このようなスケジューリングを行うのがフェアシェアである。まず、ユーザごとに優先度を考える。ジョブを実行すればするほど優先度が下がり、しばらくジョブが実行されていなければ優先度が上がるようによこう。例えばソーシャルゲームのスタミナのようなものだと考えれば良い。ジョブスケジューラは、計算資源が空いたら、その計算資源で実行できるジョブのうち、もっとも優先度の高いジョブを選んで実行する。これにより、たくさんジョブを実行している人は優先度が下がり、あまりジョブを実行していない人が後からジョブを投げたら、そちらが優先されるようにジョブがスケジューリングされる。

では、Aさんの4つのジョブが走り始めた瞬間はどうだろう？まだほとんど計算が走っていないので、Aさんのスタミナは減っていない。もし「これまでに実際に走らせたジョブの実行時間」でスタミナを減らしてしまうと、現在4ノードジョブが走っているにもかかわらず、次に走る予定のジョブのも Aさんのジョブになってしまう。もちろん、時間がたって Aさんのスタミナが実際に減った後、再スケジューリングにより Bさんのジョブが優先されることになるが、これでは不便な気がしてくるであろう。そこで、ある種のジョブスケジューラでは、ジョブが走り始めた瞬間に、ノード数と「予定実行時間」の積によってスタミナを減らしてしまう。そうすると、スケジューラが描く「現在の予定」では、次に Bさんのジョブが走ることになる。もちろん、実際のジョブは「予定実行時間」に達する前に終了することもあるだろう。その場合は Aさんのスタミナをその分回復させ、再度スケジューリングする。

ジョブのスケジューリングは、「計算資源の利用効率を上げること」を重視するか、「最悪待ち時間を減らすこと」を重視するかで大きくポリシーが変わる。また、大きなジョブと小さなジョブが混在するキュー

を作ると隙間が空きやすく、またスケジューリングが難しくなるため、非効率的になることが多い。そこで、ネットワーク構成などもにらみつつ「これくらいの規模のジョブはこの計算資源にディスパッチする」と、ジョブのサイズに応じていくつかのキューに分けて運用することも行われている。

バックフィル

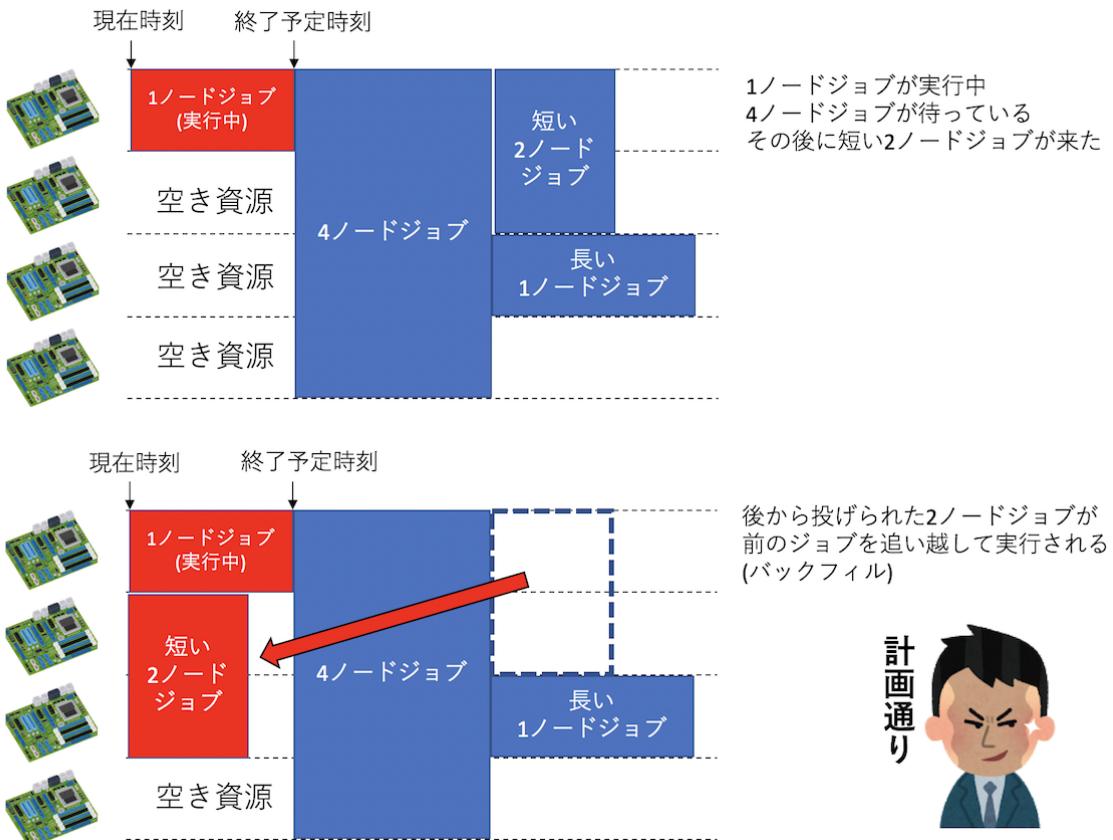


図 6: fig/backfill.png

今、4ノードの計算資源があったとして、1ノードから4ノードのジョブを実行できるとしよう。最初に1ノードのジョブが実行され、次に4ノードのジョブが投入された。4ノードを要求するジョブは、4ノードとも空かないと実行できないため待たれる。さて、その後、さらに1ノードジョブが投入された。もし、ジョブスケジューラが「待ち行列に入っているジョブのうち、次に実行可能なジョブを選ぶ」というアルゴリズムでジョブを選ぶと、いま3ノード空いており、4ノードジョブと1ノードジョブがあるので、1ノードジョブを選んで実行してしまう。すると、1ノードジョブと4ノードジョブが混ざって投入されるような場合、1ノードジョブのみが実行され、4ノードジョブは永遠に実行できなくなってしまう。逆に、1ノードジョブが走っている状況で、次に4ノードジョブを実行させようとすると、先に走っている1ノードジョブが終わるまで、3ノードが無駄になってしまふ。こんな状況を改善するのが「バックフィル」である。

ジョブには「実行予定時間」が記載されている。したがって、ジョブスケジューラは、先に走っているジョブがいつ終わるか、その終了予定時刻を知っている。したがって、その終了予定時刻前に終わるジョブであれば、4ノードジョブの実行前に実行させて良い。このようなジョブのディスパッチを「バックフィル」と呼ぶ。

普通、スペコンにはキューごとにデフォルトで「最長実行時間」が決められており、ジョブに要求時間の記載がなければ最長実行時間を要求したとみなされるのが普通である。ユーザは自分のジョブの実行時間

が「最長実行時間内に終わるかどうか」だけを気にして、ジョブの要求時間を記載しない(=最長実行時間 を要求する)ことが多い。しかし、ちょっと考えてみればわかるが、すべてのジョブが同じ実行時間 を要求すると、バックフィルされない。逆に、もし短い時間で終わることがわかっているジョブに、ちゃんとその時間を要求して投入すれば、バックフィルによってジョブの隙間に入りやすくなり、ジョブが実行されやすくなる。

スパコン運用担当としての経験から、ジョブスケジューリングを全く気にしないでジョブを投げるユーザと、ジョブスケジューリングを気にして、最適なジョブ投入戦略を練るユーザの両極端に分かれる。まあ、あまりそういうのを気にするのもアレなのだが、3時間とかで終わるジョブなのに、実行時間を指定せずに24時間 を要求し、その結果なかなか実行されずに待たされて「すぐ終わるジョブなのに長く待たされる！」と文句を言うのもどうかと思うので、少しはジョブスケジューリングを気にもしても良いかもしれない。

チーンジョブ

既に述べたように、ジョブの予定実行時間を短くしたほうがバックフィルによってジョブが実行されやすくなる。したがって、最大24時間の実行が許されるキューであっても、4時間ジョブ6つに分けて走らせれば、早く結果が得られる可能性が高い。この場合、前のジョブが終了したらチェックポイントファイルを吐いて、次のジョブはそれを読み込んで続きを実行することになる。したがって、前のジョブが終了していないにもかかわらず、次のジョブが走り走り始めるとジョブが失敗してしまう。また、6つのジョブのうちの最初の方のジョブが失敗した場合、残りのジョブは実行してほしくない。このような状況に対応するため、多くのジョブスケジューラでは「チーンジョブ」もしくは「ジョブチェーン(Job Chaining)」という仕組みを用意している。これは、ジョブ同士に依存関係を指定する方法であり、例えば「このジョブが終了しなければこのジョブは実行してはならない」「前のジョブが失敗したら次のジョブは実行しない」といった指定が可能である。チーンジョブの指定の仕方はジョブスケジューラシステムに依存するが、一般的にはジョブを投入した際に振られるJOB IDを使い、次のジョブを「このJOB IDを持つジョブが正常終了したら実行する」という条件付きでジョブを投入することで、ジョブの依存関係を指定する。例えばPBSであれば以下のようにする。

ターミナルから直接依存関係を指定する場合。

```
qsub go.sh -W depend=afterok:123456
```

ジョブスクリプトに記載する場合。

```
# PBS -W depend=afterok:123456
```

ここで `afterok` は、「正常終了した場合」を意味する。バックフィルとチーンジョブを有効活用するだけでなく、ジョブを実行した後に「あ、もうちょっと長く実行したい」と思うこともあるので、スパコンに投入するプログラムはなるべくチェックポイントリストアにさせておきたい。

ステージング

スパコンが、大きく分けて「ログインノード」「計算ノード」「ファイルシステム」から構成され、それらが高速なネットワークで結合されていることは既に述べた。ファイルシステムはログインノードにマウントされており、そこで作業をするのだが、計算ノードからファイルシステムが見えるかどうかはスパコンの構成による。ログインノードからもすべての計算ノードからファイルシステムが見えるようになっている場合、「グローバルファイルシステム」などと呼ばれる。計算ノードからグローバルファイルシステムにファイルの読み書きをする場合、ネットワークを経由するため、ネットワークの構成によっては渋滞が起きることがある。特に大規模なジョブが多数のノードから一気にファイルの読み書きをすると、通信路がサチって性能がでなくなったり、他のジョブと競合して性能劣化してしまったりする。そうすると、ファイルの

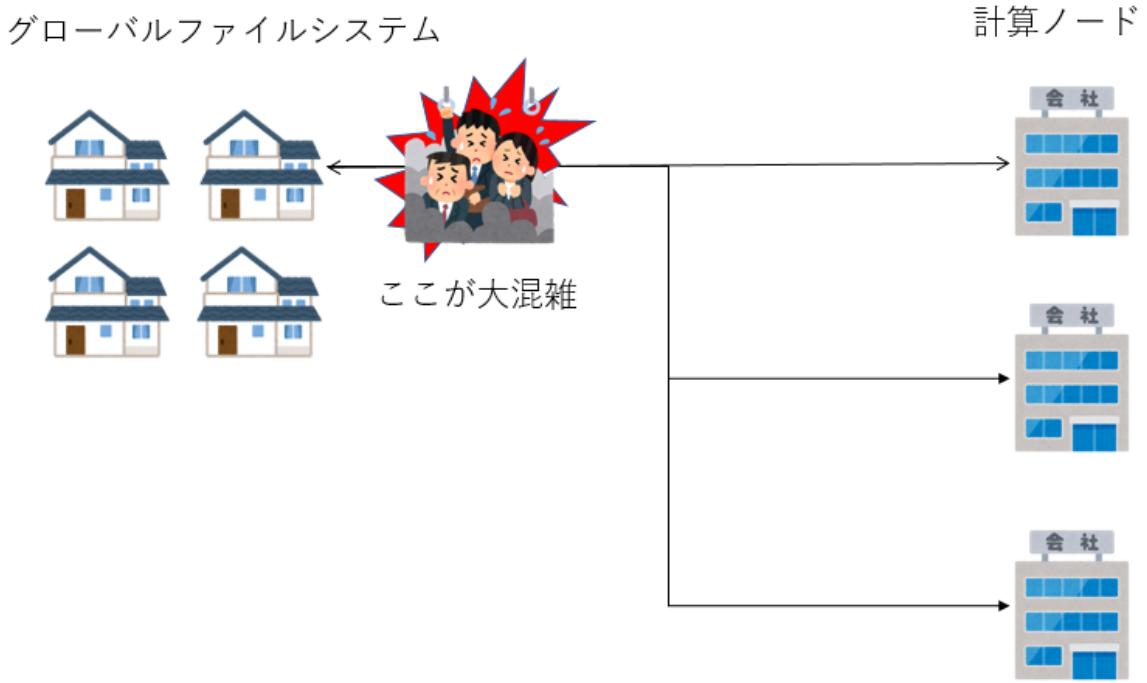


図 7: fig/crowded.png

読み書きがボトルネックになってしまい、シミュレーションなどが遅くなってしまう。こういう状況を防ぐため、計算ノードに小容量だが高速なローカルファイルシステムをつける場合がある。ローカルファイルシステムは、各計算ノードからしか見えない。もちろんログインノードからも見えない。各計算ノードが独占して利用できるため、高速に読み書きできる。ローカルファイルシステムとして SSD を採用する例もあるようだ。

さて、各計算ノードにローカルファイルシステムが用意されているが、それは計算ノードからしか見えない（計算ノードにしかマウントされていない）。例えば、大量のファイルを処理したいとしよう。そのためには、各プロセスが処理すべきファイルを、そのプロセスが実行されるノードに接続されたローカルファイルシステムにコピーする必要がある。また、処理が終わったファイルを、ジョブ終了時にローカルファイルシステムからグローバルファイルシステムに持ってこなければならない。しかし、ジョブが実行されるまで、ジョブがどの計算ノードで実行されるかわからない。そこで、これもジョブスケジューラが面倒を見る。

ジョブスクリプトに、どのプロセスが、どんなファイルを必要とし、最後にどんなファイルをグローバルファイルシステムを持ってきて欲しいのかを記載する。ジョブスケジューラはそれを見て、グローバルファイルシステムとローカルファイルシステムの間のファイルのコピーを行う。このような作業を「ステージング」と呼ぶ。実行前にローカルにファイルをコピーするのを「ステージイン」、実行終了後にローカルからグローバルにファイルをとってくるのを「ステージアウト」と呼ぶ。ステージングはスパコンサイトによって様々な方式があるので、詳細はマニュアルを参照して欲しいが、とにかく「すべての計算ノードからグローバルファイルシステム見えるようにしてファイルを読み書きすると、途中の通信路で渋滞が起るので、計算ノードの近くにファイルを置いて、計算ノードとグローバルファイルシステムの間のやりとりはジョブの実行前と実行後の一回ずつだけにしましょう」というのがステージングである。例えるなら、会社員が一ヶ月遠くに出向することになって、通勤できなくもないがすごく大変なので、出向先の近くにマンスリーマンションを借りてしまえば、長時間の移動は最初と最後だけだよね、みたいな感じである。

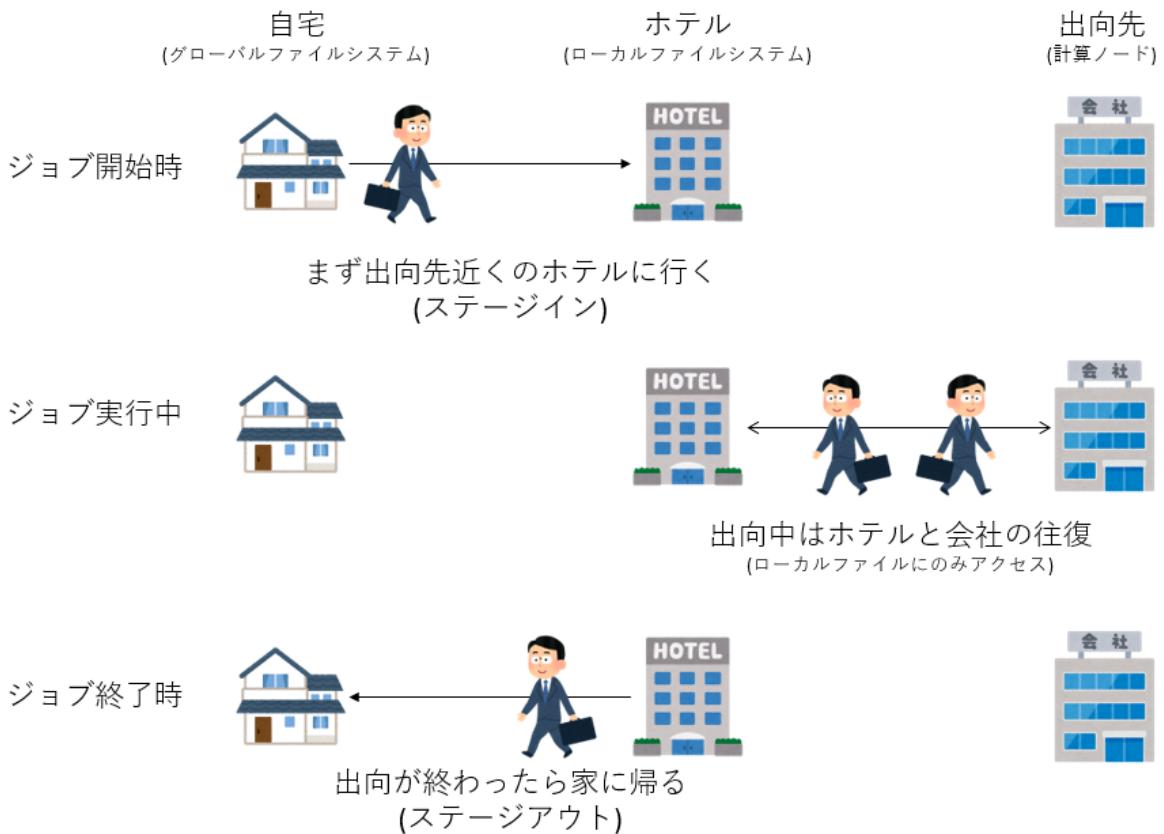


図 8: fig/staging.png

並列ファイルシステム

あなたが将来ものすごく大量のファイルを吐いたりすることがなければ、ファイルシステムについてあまり気を使うことはないだろう。しかし、ファイルシステムもスペコンを構成する重要な要素なので、それについてざっくりとでも知っておくことは有用であろう。

そもそもファイルシステムとはなんだろうか？我々が普段、クリックでファイルを開いたり、`cat` でファイルの中身を表示させたりしている時に、そのファイルがハードディスクのどこに、どのように保存されているか意識することはほとんどない。例えばハードディスクは「プラッタ」と呼ばれる円盤に情報が保存される仕組みだが、この「プラッタ」は「セクタ」と呼ばれる扇型の部分に分割されている。ファイルは、その「セクタ」単位で保存される。したがって、なにかファイルの情報がほしければ、まずそのファイルがどこのセクタに保存されているかを調べ、そのセクタに対して読み出しをかける必要がある。また、ファイルは複数のセクタに分かれて保存されている場合もあり、その場合は必要な回数だけセクタを読み出す必要がある。こういった作業を我々の代わりにやってくれるのがファイルシステムである。ファイルシステムは、主に二つの情報を扱う。一つはもちろんファイルのデータそのものである。もう一つは「どのファイルが、どの場所にあるか」という、いわば索引データである。この索引データのことを「メタデータ」と呼ぶ。

端末で作業している時、日常的に `ls` を叩くと思う。この時、「現在のディレクトリにはどんなファイルがありますか？」とファイルシステムに問い合わせていることになる。このアクションではファイルの情報は更新されないから、メタデータ情報のみが必要になる。スペコン向けに大容量のファイルシステムを組むと、大量のファイルが作成される。すると、「どのファイルがどこにあるか」の索引を調べるだけでも結構な労力となる。とあるスペコン向けファイルシステムでは、総ファイル数が増えてくるとメタデータの処理が目に見えて遅くなり、`ls` を叩いても帰ってくるのに数秒～10秒以上かかるようになっていた。`ls`

が重いと作業が極めてストレスフルになるため、メタデータの処理速度はユーザの幸せ度に直結する。

さて、スパコンは、複数のノードをネットワークでつないだものである。その複数のノードそれぞれに、別々のファイルシステムがつながっていたら、ジョブを投げるたびに「どのノードで実行されたか」を調べ、そこにファイルをコピーしなければならず不便である。したがって、複数のノードから共通して見えるファイルシステムが欲しくなる。この目的で昔から広く使われていたのが NFS というファイルシステムである。これは、ある PC にぶら下がっているファイルシステムを、別のファイルシステムからネットワーク越しに見えるようにしたもので、異なる OS や異なるファイルシステム間で接続できるように、その違いを吸収する層を持っている。「ファイルシステムを提供する側」を NFS サーバ、「ネットワーク越しにマウントする側」を NFS クライアントと呼ぶ。NFS クライアントからは、そのファイルはディレクトリの一つに見える。しかし、そのディレクトリにアクセスすると、ネットワーク越しに NFS サーバに問い合わせが飛び、ファイルの検索やファイルの作成、削除などができるシステムになっている。

この NFS、比較的小さなシステムを組むには便利なのだが、サーバにぶら下がるクライアントが増えてくると極めて遅くなる。複数のクライアントから同時に一つのサーバに問い合わせが飛ぶため、サーバも忙しくなるし、ネットワーク帯域を使い切ると、そこもボトルネックになる。近年のスパコンは千ノード、一万ノードといった構成を取るが、一つの NFS サーバに一万ノードをぶら下げるのは(おそらく)不可能である。また、昔は NFS サーバに多数のクライアントをぶら下げていっきにアクセスすると、NFS サーバごと落ちる、なんてことが頻繁にあった(筆者の設定が悪かったからかもしれないが……)。ちなみに並列向けの pNFS という規格もあるようなのだが、筆者はよく知らない。

とにかく、スパコン向けには、非常に多くのクライアントからの問い合わせを高速にさばいてくれる、スケーラブルな並列ファイルシステムが必要となる。現在、その目的で広く使われているのが Lustre ファイルシステムである。Lustre で特徴的なのは、メタデータを管理するサーバと、ファイルの実体を管理するサーバを分けて用意することである。これにより、多数のクライアントから大量の問い合わせが押し寄せても高速にさばくことができる。筆者の実感としては、Lustre は特にメタデータの問い合わせへの応答速度が早いよう思う。

簡単に Lustre の仕組みを見てみよう。まず、ファイルの索引情報であるメタデータを管理する「メタデータサーバ (Meta Data Server, MDS)」がある。メタデータの実体は、「メタデータターゲット (Meta Data Target, MDT)」と呼ばれる場所に格納されている。ファイルの実体を管理するのは「オブジェクトストレージサーバ (Object Storage Server, OSS)」であり、ファイルの実体そのものは「オブジェクトストレージターゲット (Object Storage Target, OST)」に保存されている。

ユーザが使うログインノードや、計算ノードは、Lustre のクライアントとして、Luster システムとネットワークでつながっている。ファイルの中身が欲しい時、例えば `cat test.txt` などとした時、Lustre クライアントは、まず MDS に「このファイルはどこにあるか?」と問い合わせる。すると MDS から、「どの OST のどこにある」という回答が帰ってくるため、それをもとに OST にファイルを要求、OST からファイルが届く、という手順を取る。

Lustre のメタデータ性能や、Read/Write 性能は、サーバの構成や、ネットワーク、ディスクのコントローラの性能などに依存する。また、Lustre には耐障害の保証は含まれていないため、MDT や OST で RAID を組むなどして対応する。このあたりを真面目に考えるといろいろ面倒なのだが、ここでは深入りしない。

最後に、手近に Lustre ファイルシステムがある人向けに、ちょっとだけ Lustre クライアントで遊んでみよう。まず、Lustre をマウントしたディレクトリに移動しよう。だいたいのスパコンでは、`/home` は Lustre で構成していると思われる所以、普通はホームディレクトリで良い。念の為 `df -T /home` で確認してみよう。手近なスパコンで試した結果はこんな感じになる。

```
$ df -T /home
ファイルシステム          タイプ   1K-ブロック   使用   使用可   使用% マウント位置

```

Lustreの仕組み

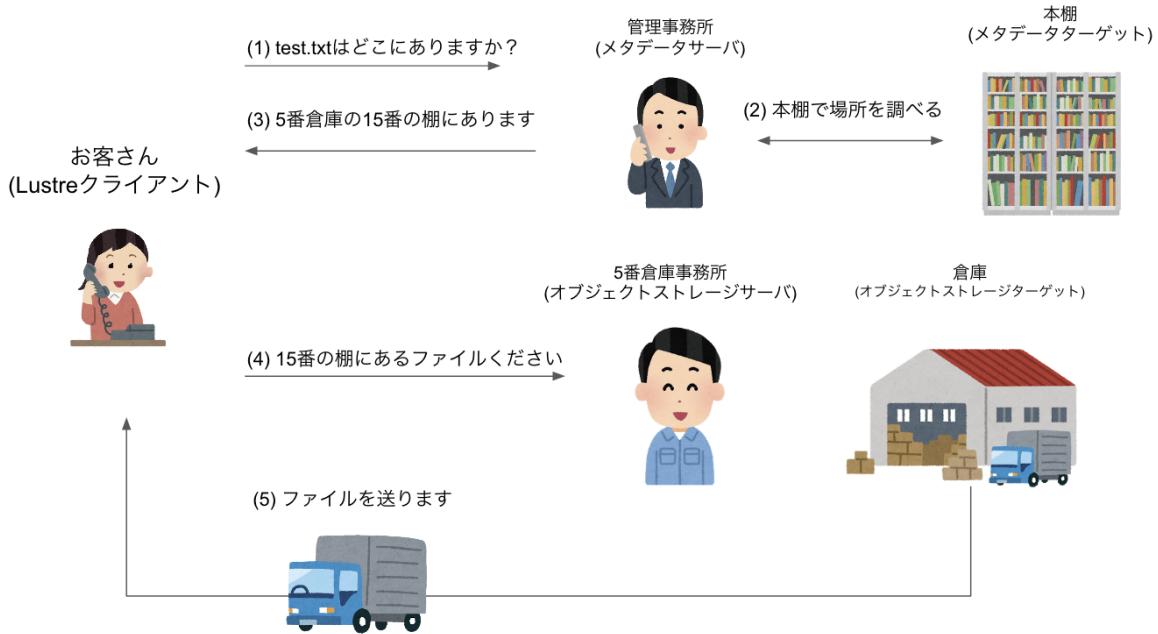


図 9: fig/lustre.png

```
path/to/mds:/home lustre XXXXXXXXXXXX YYYYYYYY ZZZZZZ PP% /home
```

「/home」のタイプが「lustre」になっていることがわかる。数字その他はなんなく伏せた。ここで、適当にディレクトリを掘る。例えば `temp` というディレクトリを作り、そこに移動しよう。

```
mkdir temp
cd temp
```

次に、適当にファイルを作る。例えば `test.txt` にしよう。

```
$ touch test.txt
$ ls
test.txt
```

この状態で、Lustre クライアントコマンド、`lfs` を叩いてみる。ここではファイル ID をとってみよう。

```
$ lfs path2fid test.txt
[0x20005cf46:0x1794e:0x0]
```

Lustre は、ファイルを「ファイル ID (File Identifier, FID)」で管理する。Lustre 2.x 系列では、FID は 128 ビットで表現されている。最初の 64 ビットが Sequence、次の 32 ビットが Object ID、最後の 32 ビットがバージョンである。ここでは `0x20005cf46` が Sequence、`0x1794e` が Object ID、`0x0` がバージョン番号である。

この FID はファイルに追記したり、リネームしても変わらない。

```
$ echo 1 >> test.txt
$ lfs path2fid test.txt
[0x20005cf46:0x1794e:0x0]

$ mv test.txt test2.txt
```

```
$ lfs path2fid test2.txt  
[0x20005cf46:0x1794e:0x0]
```

しかし、一度削除してから作り直すと FID は変わる。

```
$ rm test2.txt; touch test2.txt  
$ lfs path2fid test2.txt  
[0x20005cf46:0x1799c:0x0]
```

Lustre がこの FID をどう使ってファイルを管理しているか、なぜ Sequence を導入したかなどはここでは深入りしない。興味のある人は Lustre の公式ドキュメントなどを参照してほしい。

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る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 <iostream>
#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と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れば 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 σ^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」「コミュニケーション」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

```

$\pi = 3.142135 \pm 0.003565$

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いでも、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「ウェークスケーリング」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ェークスケーリングの効率を上げるより難しい。

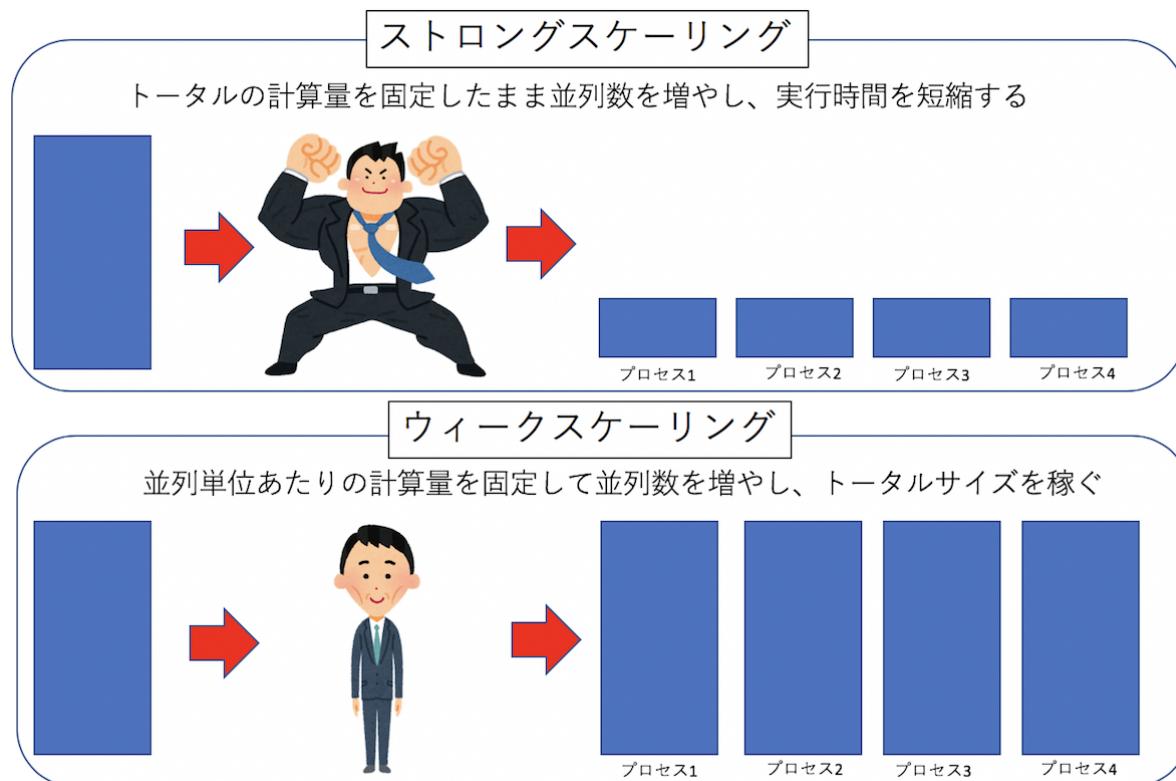


図 1: fig/scaling.png

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 α 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 α は

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

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。

まあ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。

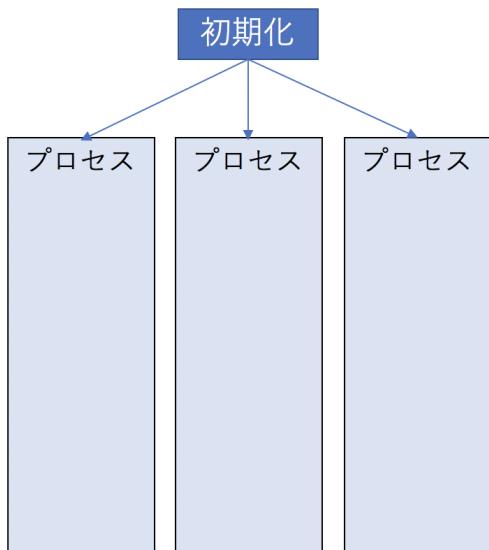
「サンプル並列」も「パラメタ並列」もウィークスケーリングに属し、それぞれのタスクの計算時間に大きなばらつきがあったりしなければ、かなり大規模な計算をしても理想的な並列化効率が出来る。ただし、「パラメタ並列」は、100 倍の計算資源を突っ込めば 100 倍のパラメータを処理できるのに対して、「サンプル並列」は 100 倍の計算資源を突っ込んでも、統計精度は 10 倍にしかならない。これは、サンプリングによる精度の向上が、サンプル数の平方根でしか増えないからだ。したがって、いかに並列化効率が 100% に近くても、サンプル平均に大規模な計算資源を突っ込むのはあまり効率がよくない。なにかの精度を一桁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

Day 4 : 領域分割による非自明並列

非自明並列

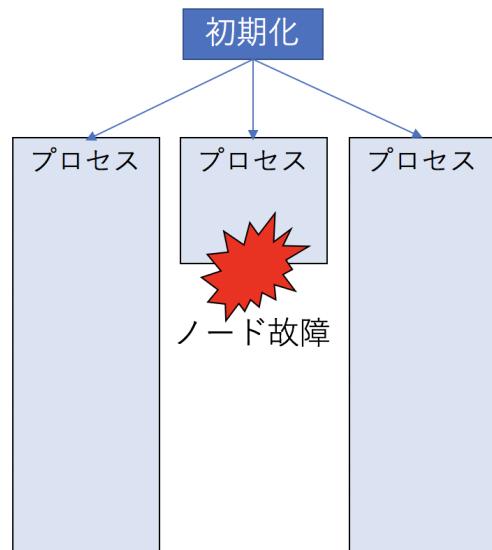
Day 3 では自明並列を扱ってきた。自明並列は別名「馬鹿パラ」と呼ばれ、馬鹿にされる傾向にあるのだが、並列化効率が高いため、「計算資源は」最も有効に使える計算方法である。さて、「スペコンはノードを束ねたもの」であり、「ノードとは本質的には PC と同じもの」であることは既に述べた。しかし「普通の PC を多数束ねたらスペコンになるか」というとそうではなく、スペコンとして動作をするためには「ネットワーク」と「信頼性」が重要なのであった。実は、馬鹿パラは「ネットワーク」と「信頼性」のどちらも必要としない。

馬鹿パラは通信がほぼ不要



初期化以外では通信しない

馬鹿パラは信頼性が不要



失敗した計算だけやり直せば良い

図 1: fig/bakapara.png

パラメタ並列の場合、一番最初に「どのパラメタをどのプロセスが担当すべきか」をばらまくのに通信したあとは通信不要である（計算が終わったら結果をファイルに吐いてしまえばよい）。したがって、各ノードが高速なネットワークで接続されている必要はなく、たとえばイーサネットなどでつないでしまって全く問題ない。また、大規模な非自明並列計算を実行するには高い信頼性が求められるが、馬鹿パラは信頼性も要求しない。計算途中でノードが壊れてしまっても、そのノードでしていた計算だけやり直せばよいだけのことである。つまり馬鹿パラとは最も計算資源は有効に使えるものの、「ネットワーク」と「信頼性」という、スペコンの重要な特性を全く使わない計算方法なのであった。なので、主に馬鹿パラで計算する場合には、「普通の PC を多数束ねた PC クラスタ」で全く構わない。

さて、馬鹿パラであろうとなんであろうと、スペコンを活用していることにはかわりないし、それで良い科学的成果が出るのならそれで良いのだが、せっかくスペコンを使うのなら、もう少し「スペコンらしさ」を活用してみたい。というわけで、「ネットワーク」と「信頼性」をどちらも要求する非自明並列 (**non-trivial parallel**) に挑戦してみよう。

馬鹿パラではほとんど通信が発生しなかったのに対して、非自明並列は頻繁に通信が必要とする。科学計算はなんらかの繰り返し計算（例えば時間発展）をすることが多いが、意味のある並列計算を行う場合、毎

ステップ通信が必要となる。この時、「計算に関わる全ノードと毎回通信が発生する」タイプと、「論理的に距離が近いノードのみと通信が必要となる」タイプにわかれる。

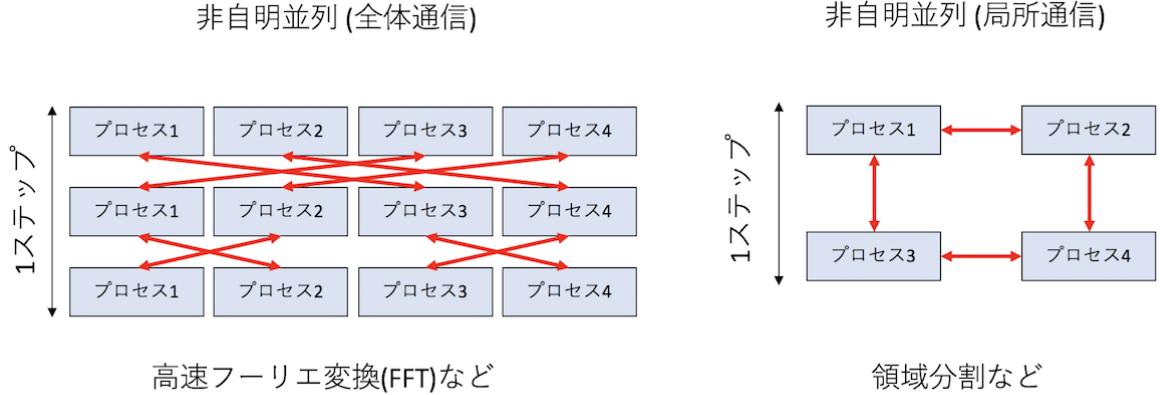


図 2: fig/nontrivial.png

毎回全ノードと計算が必要になるタイプは、典型的には高速フーリエ変換が必要となる場合である。例えば、「円周率を一兆桁計算する」といった計算において、多倍長計算が必要となり、その多倍長計算の実行にフーリエ変換が使われる。乱流の計算にもフーリエ変換が使われる。以上、地球シミュレータで大規模な乱流シミュレーションが行われたが、これは地球シミュレータの強力なネットワークがなければ実現が難しかった。こういう全ノード通信は、バタフライ型のアルゴリズムで実行されることが多いが、ここでは深入りしない。興味のある人は「並列 FFT」などでググってみて欲しい。

それに対して、「近いノードのみ」と通信が必要となるタイプは、典型的には領域分割による並列化にあらわれる。領域分割とは、「計算したい領域」をプロセス数で分割して、隣接する領域で必要となる情報を通信しながら計算を実行するような並列化である。うまくプロセスを配置しないと、論理的には近い領域が、物理的には遠いノードに配置されたりして効率が悪くなるので注意したい。

とりあえず図を見て、「全体通信の方が大変そうだな」と思うであろう。基本的には、計算量に対して通信量、通信頻度が低いほど並列化が楽になる(効率が出しやすい)。計算コストと通信コストの比を「粒度」と呼ぶこともある(後で説明するかもしれないし、しないかもしれない)。

とりあえずここでは非自明並列の例として、領域分割をやってみる。

一次元拡散方程式 (シリアル版)

領域分割による並列化の題材として、一次元拡散方程式を考えよう。これは熱を加えられたり冷やされたりした物質の温度はどう変化していくかを表す方程式である。時刻 t 、座標 x における温度を $T(x, t)$ とすると、熱伝導度を κ として、

$$\frac{\partial T}{\partial t} = \kappa \frac{\partial^2 T}{\partial x^2}$$

で表される。この方程式の定常状態は、時間微分がゼロとなる状態で与えられるから、

$$\kappa \frac{\partial^2 T}{\partial x^2} = 0$$

ある関数の二回微分がゼロなので、解は二次関数か一次関数で与えられることがわかりますね。ちなみに一般の時刻における解はフーリエ変換で求められる。理工系の大学生であれば二年生くらいまで習っているはずなので各自復習されたい。

さて、この偏微分方程式を数値的に解くために空間を L 分割して差分化しよう。時間については一次のオイラー法、空間については中央差分を取る。時間刻みを h 、空間刻みを 1、 n ステップ目における i 番目のサイトの温度を T_i^n とすると、同じ場所の次のステップの温度 T_i^{n+1} は、

$$T_i^{n+1} = T_i^n + h(T_{i+1}^n - 2T_i^n + T_{i-1}^n)$$

と表現される。例えば時間ステップ n の温度を `std::vector<double> lattice` で表すと、上記の式をそのままコードに落とすと

```
std::copy(lattice.begin(), lattice.end(), orig.begin());
for (int i = 1; i < L - 1; i++) {
    lattice[i] += h * (orig[i - 1] - 2.0 * orig[i] + orig[i + 1]);
}
```

と書ける。らくちんですね。これだと両端 (`lattice[0]` と `lattice[L-1]`) が更新されないので、周期境界条件を課しておく。以上を、1 ステップ時間を進める関数 `onestep` として実装しよう。

```
void onestep(std::vector<double> &lattice, const double h) {
    static std::vector<double> orig(L);
    std::copy(lattice.begin(), lattice.end(), orig.begin());
    for (int i = 1; i < L - 1; i++) {
        lattice[i] += h * (orig[i - 1] - 2.0 * orig[i] + orig[i + 1]);
    }
    // For Periodic Boundary
    lattice[0] += h * (orig[L - 1] - 2.0 * lattice[0] + orig[1]);
    lattice[L - 1] += h * (orig[L - 2] - 2.0 * lattice[L - 1] + orig[0]);
}
```

これで数値計算部分はおしまい。ついでに、系の状態をファイルにダンプする関数も書いておこう。

```
void dump(std::vector<double> &data) {
    static int index = 0;
    char filename[256];
    sprintf(filename, "data%03d.dat", index);
    std::cout << filename << std::endl;
    std::ofstream ofs(filename);
    for (int i = 0; i < data.size(); i++) {
        ofs << i << " " << data[i] << std::endl;
    }
    index++;
}
```

あとは適当な条件を与えれば時間発展させることができる。ここでは、「一様加熱」と「温度固定」の二通りを試してみよう。コードはこちら。

`thermal.cpp`

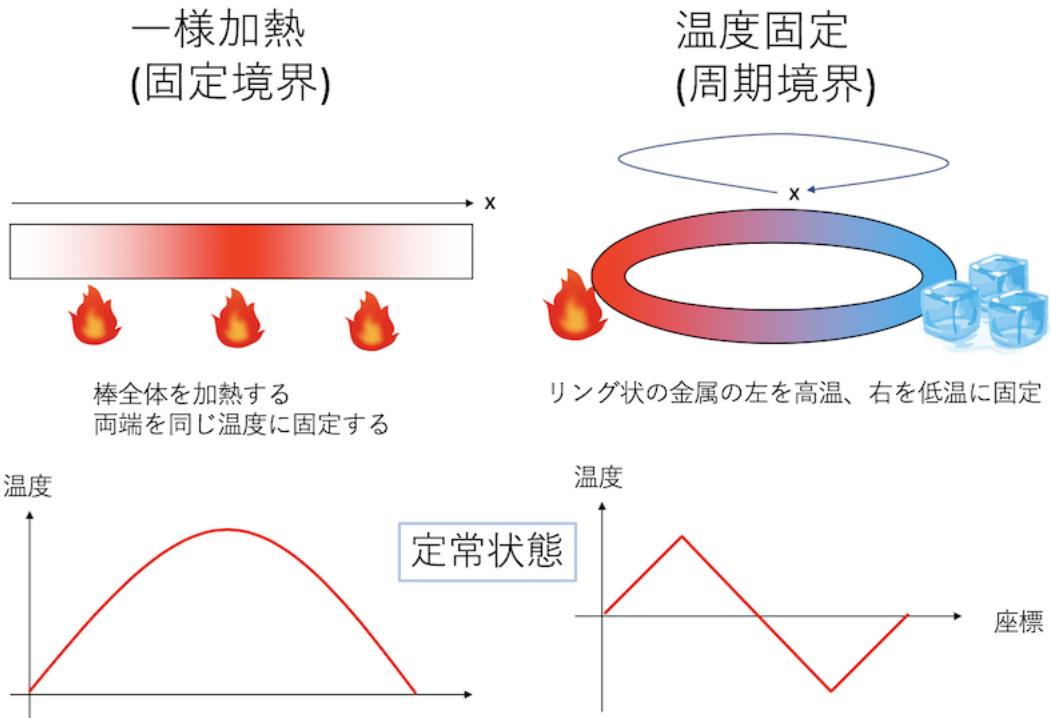


図 3: fig/thermal.png

一様加熱というのは、系のすべての場所を一様に加熱することである。単位時間あたりの加熱量を Q として、

```
for (auto &s : lattice) {
    s += Q * h;
}
```

とすれば良い。ただしこのままでは全体が熱くなってしまうので、棒の両端の温度を 0 に固定しよう。

```
lattice[0] = 0.0;
lattice[L - 1] = 0.0;
```

計算部分はこんな感じにかける。

```
void uniform_heating(std::vector<double> &lattice) {
    const double h = 0.2;
    const double Q = 1.0;
    for (int i = 0; i < STEP; i++) {
        onestep(lattice, h);
        for (auto &s : lattice) {
            s += Q * h;
        }
        lattice[0] = 0.0;
        lattice[L - 1] = 0.0;
        if ((i % DUMP) == 0) dump(lattice);
    }
}
```

何ステップかに一度、系の状態をファイルに吐いている。定常状態は、両端がゼロとなるような二次関数、

具体的には

$$T(x) = -x(x - L)$$

となる。二次関数で、両端がゼロとなること、これが熱伝導方程式の解になっていることを確認しよう。

計算結果はこんな感じになる。

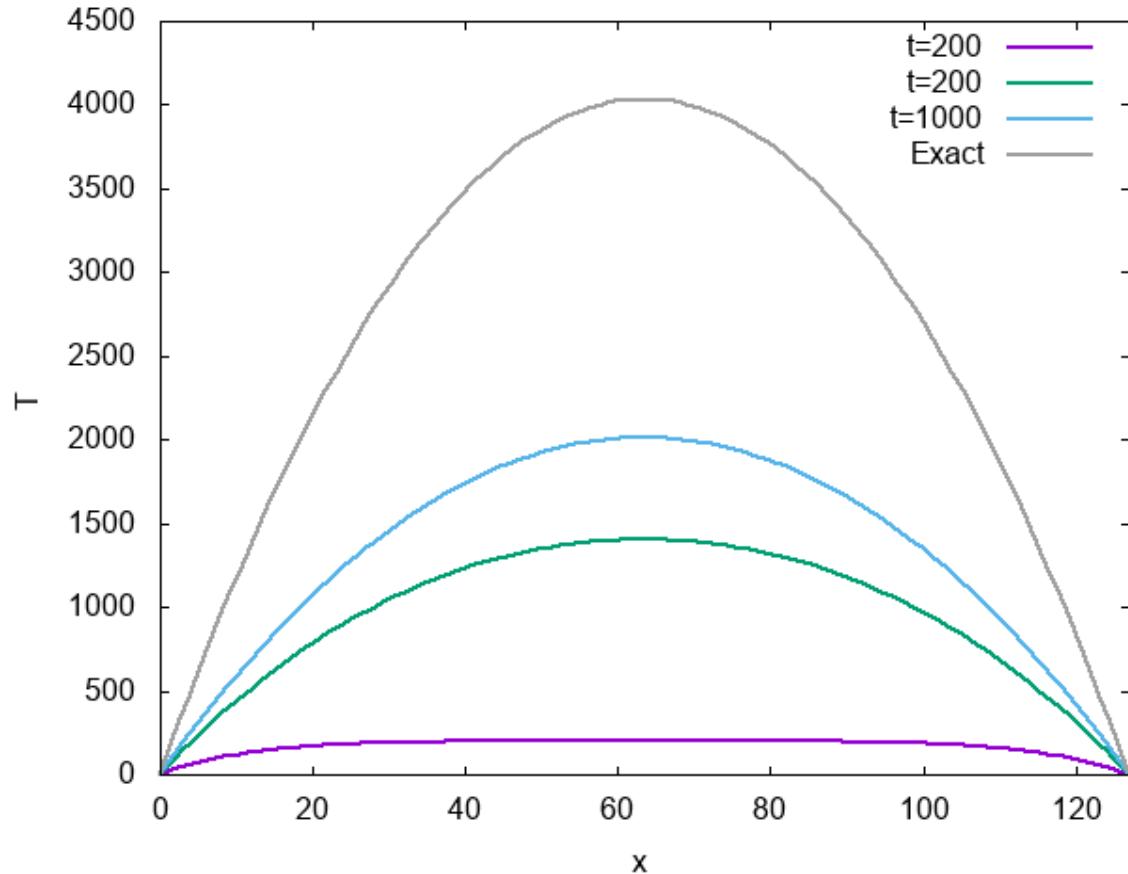


図 4: uniform.png

時間がたつにつれて温度が上がっていき、定常状態に近づいていくのがわかる。

この例では、周期境界条件がちゃんとできているか確認できないので、温度が境界をまたぐような条件、「温度固定」を試してみよう。リング状の金属の棒の、ある点を高温に、反対側を低温に固定する。すると、定常状態は、高温と低温を結ぶ直線になる。

計算部分はこんな感じに書けるだろう。

```
void fixed_temperature(std::vector<double> &lattice) {
    const double h = 0.01;
    const double Q = 1.0;
    for (int i = 0; i < STEP; i++) {
        onestep(lattice, h);
        lattice[L / 4] = Q;
        lattice[3 * L / 4] = -Q;
```

```

    if ((i % DUMP) == 0) dump(lattice);
}
}

```

計算結果はこんな感じ。

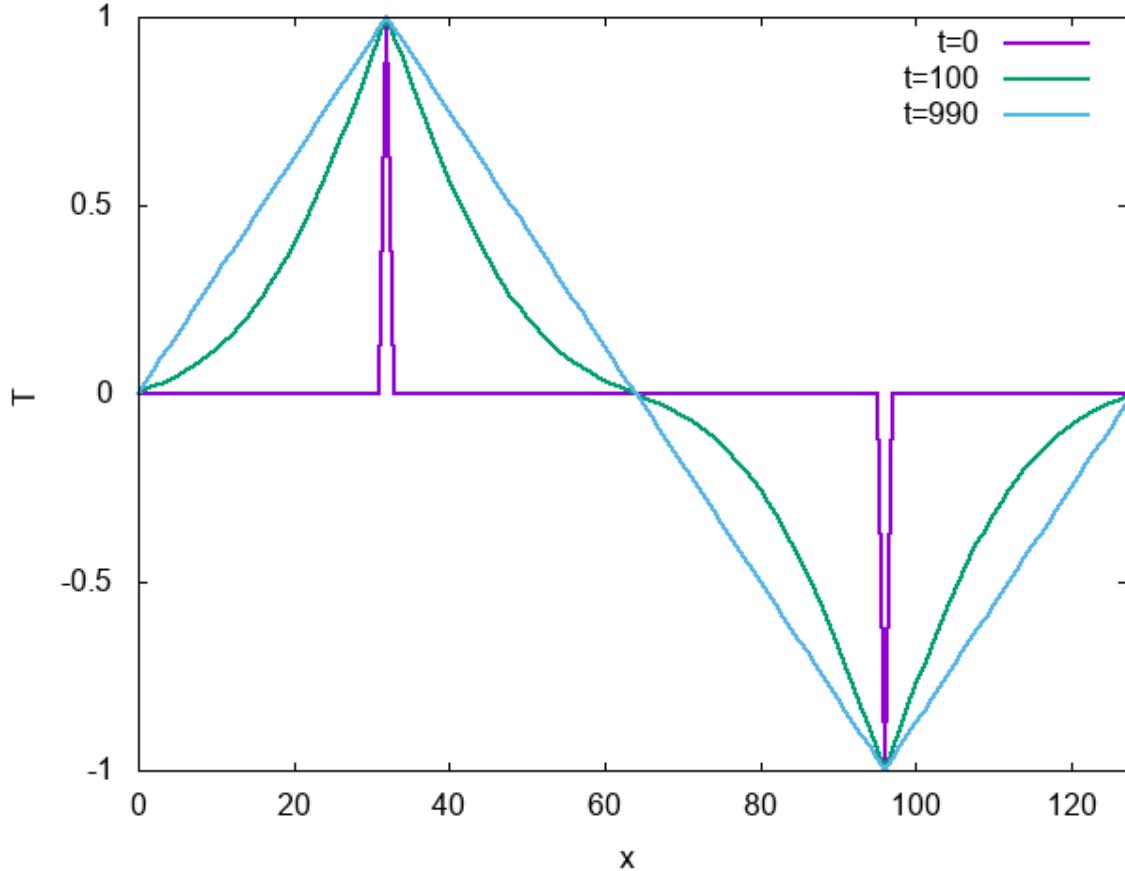


図 5: fixed.png

時間がたつにつれて、定常状態である直線になる。ちなみに、定常状態で温度勾配が直線になる現象はフーリエの法則という名前がついている。あのフーリエ変換のフーリエさんである。もともとフーリエは熱伝導の問題を解くためにフーリエ級数を編み出したのであった。

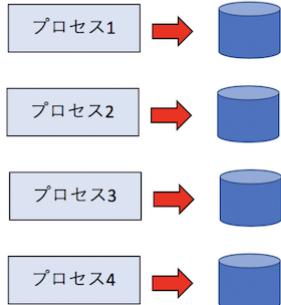
一次元拡散方程式（並列版）

さて、一次元拡散方程式のシミュレーションコードがかけたところで、これを並列化しよう。並列化の方法としては領域分割を採用する。要するに空間をプロセスの数で分割して、各プロセスは自分の担当する領域を、必要に応じて隣のプロセスから情報をもらいながら更新すればよい。ただし、隣の領域の情報を参照する必要があるので、その部分を「のりしろ」として保持し、そこを通信することになる。

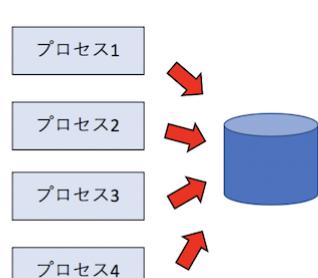
並列化で考えなければいけないことの一つに「ファイル出力をどうするか」というものがある。これまでプロセスが一つしかなかったので、そいつがファイルを吐けばよかったのだが、プロセス並列をしていると、別々のプロセスがそれぞれ系の状態を分割して保持している。どうにかしてこれをファイルに吐かないといけない。並列計算をする前に、まずは領域分割をして、各プロセスが別々に保持している状態をどうやってファイルに吐くか考えてみよう。いろいろ方法はあるだろうが、とりあえず「全プロセス勝手に吐くく」「一つのファイルに追記」「一度まとめてから吐く」の三通りの方法が考えられる。

並列プログラムのファイルの吐き方

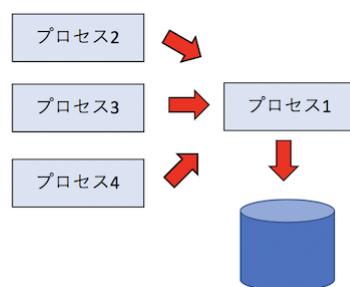
1. 全プロセス勝手に吐く



2. 一つのファイルに追記



3. 一度まとめてから吐く



コーディングが楽



ファイル数が減る



ファイル出力がまとめて
一回なので早い



ステップ数×プロセス数の
大量のファイルが出力される



プロセス数が多いと遅い



メモリを消費する

図 6: fig/parafile.png

- 「全プロセス勝手に吐く」これは各プロセスが毎ステップ勝手にファイルを吐く方法。例えば t ステップ目に i 番目のプロセスが `file_t_i.dat` みたいな形式で吐く。コーディングは楽だが、毎ステップ、プロセスの数だけ出力されるので大量のファイルができる。また、解析のためには各プロセスが吐いたファイルをまとめないといけないのでファイル管理が面倒。
- 「一つのファイルに追記」毎ステップ、ファイルをひとつだけ作成し、プロセスがシリアルに次々と追記していく方法。出力されるファイルはシリアル実行の時と同じなので解析は楽だが、「追記」をするためにプロセスが順番待ちをする。数千プロセスでやったら死ぬほど遅かった。
- 「一度まとめてから吐く」いちど、ルートプロセス（ランク 0 番）に通信でデータを集めてしまつてから、ルートプロセスが責任を持って一気に吐く。数千プロセスでも速度面で問題なくファイル出力できたが、全プロセスが保持する状態を一度一つのノードに集めるため、数万プロセス実行時にメモリ不足で落ちた。

とりあえずメモリに問題なければ「3. 一度まとめてから吐く」が楽なので、今回はこれを採用しよう。メモリが厳しかったり、数万プロセスの計算とかする時にはなにか工夫してくださいまし。

さて、「一度まとめてから吐く」ためには、「各プロセスにバラバラにあるデータを、どこかのプロセスに一括して持ってくる」必要があるのだが、MPI にはそのものずばり `MPI_Gather` という関数がある。使い方は以下のサンプルを見たほうが早いと思う。

`gather.cpp`

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

const int L = 8;
```

```

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    int rank, procs;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &procs);
    const int mysize = L / procs;
    // ローカルなデータ (自分の rank 番号で初期化)
    std::vector<int> local(mysize, rank);
    // 受け取り用のグローバルデータ
    std::vector<int> global(L);
    // 通信 (ランク 0 番に集める)
    MPI_Gather(local.data(), mysize, MPI_INT, global.data(), mysize, MPI_INT, 0, MPI_COMM_WORLD);

    // ランク 0 番が代表して表示
    if (rank == 0) {
        for (int i = 0; i < L; i++) {
            printf("%d", global[i]);
        }
        printf("\n");
    }
    MPI_Finalize();
}

```

これは、長さ $L=8$ のデータを、それぞれのプロセスが $mysize = L/procs$ 個ずつ持っている、という状況を模している。それぞれのプロセスが保持するデータは `local` に格納されている。これらはそれぞれ自分のランク番号で初期化されている。これを全部ランク 0 番に集め、`global` で受け取って表示する、というコードである。

実行結果はこんな感じ。

```

$ mpic++ gather.cpp
$ mpirun -np 1 ./a.out
00000000

$ mpirun -np 2 ./a.out
00001111

$ mpirun -np 4 ./a.out
00112233

$ mpirun -np 8 ./a.out
01234567

```

1分割から8分割まで試してみた。これができれば、一次元熱伝導方程式の並列化は難しくないだろう。全データをまとめた後は、そのデータをシリアル版のファイルダンプに渡せば良いので、こんな関数を書けば良い。

```
void dump_mpi(std::vector<double> &local, int rank, int procs) {
```

```

static std::vector<double> global(L);
MPI_Gather(&(local[1]), L / procs, MPI_DOUBLE, global.data(), L / procs, MPI_DOUBLE, 0, MPI_COMM_WORLD);
if (rank == 0) {
    dump(global);
}
}

```

各プロセスは `local` という `std::vector` にデータを保持しているが、両端に「のりしろ」があるので、そこだけ除いたデータをまとめて `global` という `std::vector` に受け取り、ランク 0 番が代表してシリアル番のダンプ関数 `dump` を呼んでいる。

ファイル出力の目処がついたところで、並列化を考えよう。差分方程式なので、両端にそれぞれ 1 サイト分の「のりしろ」を用意して、そこだけ隣と通信すれば良い。

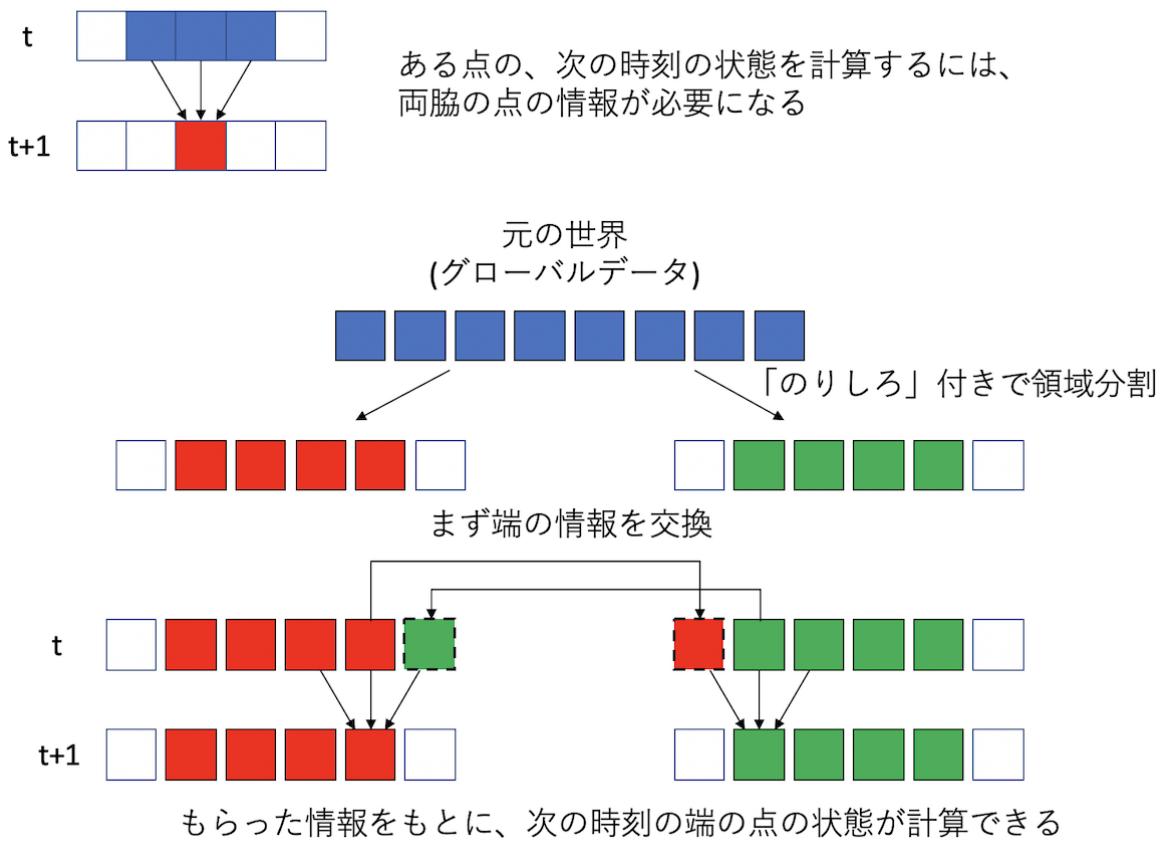


図 7: fig/margin.png

計算をする「前」に、両脇の領域を管轄するプロセスから端の情報を「のりしろ」にコピーして、その後は普通に計算すれば良い。端の情報を「のりしろ」にコピーするのには一対一通信を用いる。MPI の基本的な一対一通信関数として、`MPI_Send` による送信と `MPI_Recv` による受信が用意されているが、それらをペアで使うより、送受信を一度にやる `MPI_Sendrecv` を使った方が良い。`MPI_Send` と `MPI_Recv` を使うとデッドロックの可能性がある上に、一般には `MPI_Sendrecv` の方が性能が良いためだ。というわけで、並列化した計算コードはこんな感じになる。

```

void onestep(std::vector<double> &lattice, double h, int rank, int procs) {
    const int size = lattice.size();
    static std::vector<double> orig(size);

```

```

std::copy(lattice.begin(), lattice.end(), orig.begin());
// ここから通信のためのコード
const int left = (rank - 1 + procs) % procs; // 左のランク番号
const int right = (rank + 1) % procs; // 右のランク番号
MPI_Status st;
// 右端を右に送って、左端を左から受け取る
MPI_Sendrecv(&(lattice[size - 2]), 1, MPI_DOUBLE, right, 0, &(orig[0]), 1, MPI_DOUBLE, left, 0, MPI_DOUBLE);
// 左端を左に送って、右端を右から受け取る
MPI_Sendrecv(&(lattice[1]), 1, MPI_DOUBLE, left, 0, &(orig[size - 1]), 1, MPI_DOUBLE, right, 0, MPI_DOUBLE);

//あとはシリアル版と同じ
for (int i = 1; i < size - 1; i++) {
    lattice[i] += h * (orig[i - 1] - 2.0 * orig[i] + orig[i + 1]);
}
}

```

コードのコメント通りで、難しいことはないと思う。MPI_Sendrecv で「データを送るプロセス」と「データを受け取るプロセス」が違うことに注意。クリスマスパーティのプレゼント交換の時のように、みんなで輪になって、右の人にプレゼントを渡し、左の人からプレゼントを受け取る、みたいなイメージである。もちろん「右に渡して右から受け取る」という通信方式でも良いが、「右に渡して左から受け取る」方がコードが楽だし、筆者の経験ではそっちの方が早かった。

計算部分ができたので、あとは条件を追加すれば物理的なシミュレーションができる。まずは一様加熱。

```

void uniform_heating(std::vector<double> &lattice, int rank, int procs) {
    const double h = 0.2;
    const double Q = 1.0;
    for (int i = 0; i < STEP; i++) {
        onestep(lattice, h, rank, procs);
        for (auto &s : lattice) {
            s += Q * h;
        }
        if (rank == 0) {
            lattice[1] = 0.0;
        }
        if (rank == procs - 1) {
            lattice[lattice.size() - 2] = 0.0;
        }
        if ((i % DUMP) == 0) dump_mpi(lattice, rank, procs);
    }
}

```

シリアル版とほぼ同じだが、「両端の温度を固定」する時に、左端はランク 0 番が、右端は procs-1 版が担当しているので、そこだけ if 文が入る。あとは dump を dump_mpi に変えるだけ。

次に、温度の固定条件。

```

void fixed_temperature(std::vector<double> &lattice, int rank, int procs) {
    const double h = 0.01;
    const double Q = 1.0;

```

```

const int s = L / procs;
for (int i = 0; i < STEP; i++) {
    onestep(lattice, h, rank, procs);
    if (rank == (L / 4 / s)) {
        lattice[L / 4 - rank * s + 1] = Q;
    }
    if (rank == (3 * L / 4 / s)) {
        lattice[3 * L / 4 - rank * s + 1] = -Q;
    }
    if ((i % DUMP) == 0) dump_mpi(lattice, rank, procs);
}
}

```

これも一様加熱と同じで、「温度を固定している場所がどのプロセスが担当するどの場所か」を調べる必要があるが、それを考へるのはさほど難しくないだろう。

そんなわけで完成した並列コードがこちら。

thermal_mpi.cpp

せっかく並列化したので、高速化したかどうか調べてみよう。一様加熱の計算をさせてみる。

まずはシリアル版の速度。

```

$ clang++ -O3 -std=c++11 thermal.cpp
$ time ./a.out
data000.dat
data001.dat
(snip)
data099.dat
./a.out 0.05s user 0.12s system 94% cpu 0.187 total

```

次、並列版。

```

$ time mpirun -np 2 --oversubscribe ./a.out
data000.dat
data001.dat
(snip)
data099.dat
mpirun -np 2 --oversubscribe ./a.out 0.42s user 0.16s system 176% cpu 0.330 total

```

```

$ time mpirun -np 4 --oversubscribe ./a.out
data000.dat
data001.dat
(snip)
data099.dat
mpirun -np 4 --oversubscribe ./a.out 1.73s user 0.88s system 234% cpu 1.116 total

```

```

$ time mpirun -np 8 --oversubscribe ./a.out
data000.dat
data001.dat

```

```
(snip)  
data099.dat  
mpirun -np 8 --oversubscribe ./a.out 3.28s user 2.89s system 311% cpu 1.980 total
```

うん、無事に並列数を上げるほど遅くなった。

YA・TTA・NE☆

まあ、サイズが小さいし、一次元だから計算もとても軽いので、通信のために余計なことをする分遅くなることは実は予想できていた。しかし、領域分割の基本的なテクニックはこのコードにすべて含まれているし、これができれば原理的には差分法で陽解法なコードは全部並列化できてしまうので、応用範囲は広い。

余談：Eager プロトコルと Rendezvous プロトコル

先程、`MPI_Send` と `MPI_Recv` を使うとデッドロックの可能性があると書いた。例えばプロセス 0 番が 1 番に Send 要求と Recv 要求をして、やはりプロセス 1 番が 0 番に Send 要求と Recv 要求をしたとする。プロセス 0 番は、1 番からまず Send 要求の返答が返ってこないと次に進めない。しかし、プロセス 1 番も 0 番からの Send 要求の返答が来ないと次に進めず、Recv によって Send 要求に答えられない。したがってこのままではデッドロックしそうに思えるが、実際には小さいデータサイズの送受信ではデッドロックしない。これは、サイズが小さいデータのやりとりには「Eager (イーガー) プロトコル」という方式が使われるからだ。データサイズが大きい時には「Rendezvous (ランデブー) プロトコル」という方式が使われる。まず、Rendezvous プロトコルは、送信側から「これくらいのデータを送って良いですか？」と聞く。受信側は、そのデータを受け取るバッファが用意できることを確認してから「いいですよ」と答える。こうして「ハンドシェイク」した後にデータの送受信が始まる。この方式だと、相手からの返事がないと次に進めないのでデッドロックの可能性がある。また、受信側がデータを保存するバッファを用意できることがわかっているのであれば、いちいち送信側が受信側の応答を待つのは時間のロスになる。

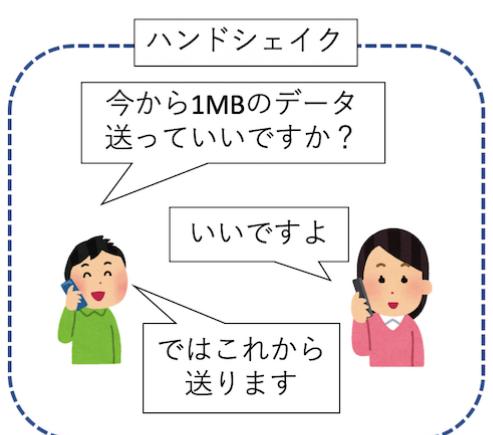
そこで Eager プロトコルでは、「あらかじめこれくらいのバッファは必ず受信側で用意している」と仮定して、送信側はそこに送りつけてしまう。よく、荷物の受け取りなどで、小さな荷物であれば再配達にせず、ドアポストやどこか決められた場所に置いてしまったりする、アレである。受信側は、自分のタイミングでそのバッファから必要な場所にデータをコピーする。送信側は相手の返事を待たずに `MPI_Send` を終了して次に処理が進んでしまうため、先程の例でもデッドロックしない。

Eager プロトコルと Rendezvous プロトコルは、送受信のサイズによって切り替わり、その切替サイズはシステムによって異なる。`MPI_Send` と `MPI_Recv` を使ったデッドロックの可能性のあるコードは、あるサイトでは正常に動作するのに、別のサイトではデッドロックしたり、同じサイトでも小さな系では正しく動作するのに、大きくするとデッドロックしたりと、分かりづらいバグを埋め込むことになる。この問題は `MPI_Sendrecv` を使うことで回避できるので、なるべくそちらを使っていきたい。

なお、`MPI_Send` や `MPI_Recv` は「ブロッキング通信」と呼ばれるため、「相手が答えるまでは次に進まない(処理がブロックされる)」という誤解がたまに見受けられるが、このように Eager プロトコルを採用することで、相手の返答の有無にかかわらず処理は次に進む。ブロッキング通信が保証するのは `MPI_Recv` の処理が終わった時には受信バッファにデータが入っていることである。ノンブロッキング通信の場合には、通信の終了処理である `MPI_Wait` による通信の完了待ちが終了するまでは通信の結果が保証されない。このあたりの事情については、例えば以下の講義資料を参照されたい。

並列プログラミングの基本 (堀 敦史)

Rendezvousプロトコル



Eagerプロトコル

決められた場所にデータを置く



受信側のタイミングで受け取りに行く



図 8: fig/r_and_e.png

Day 5 :二次元反応拡散方程式

Day 4 で一次元拡散方程式を領域分割により並列化した。後はこの応用で相互作用距離が短いモデルはなんでも領域分割できるのだが、二次元、三次元だと、一次元よりちょっと面倒くさい。後、熱伝導方程式は、「最終的になにかに落ち着く」方程式なので、シミュレーションしてあまりおもしろいものではない。そこで、二次元で、差分法で簡単に解けて、かつ結果がそこそこ面白い題材として反応拡散方程式 (reaction-diffusion system) を取り上げる。反応拡散方程式とは、拡散方程式に力学系がくついたような系で、様々なパターンを作る。例えば「reaction-diffusion system」でイメージ検索してみて欲しい。生物の模様なんかがこの方程式系で説明されたりする。

世の中には様々な反応拡散方程式があるのだが、ここでは Gray-Scott モデルと呼ばれる、以下の方程式系を考えよう。

$$\frac{\partial u}{\partial t} = D_u \Delta u + u^2 v - (F + k)u$$

$$\frac{\partial v}{\partial t} = D_v \Delta v - u^2 v + F(1 - v)$$

これは U と V という化学物質の化学反応を模した方程式である。 U が活性化因子、 V が抑制因子と呼ばれる。 U と V の濃度を u 、 v とすると、 V の濃いところでは U が生成されないことがわかる。 D_u や D_v は拡散係数であり、 $D_v/D_u = 2$ にとる。つまり、 V の方が拡散しやすい物質となる。この方程式を計算することにしよう。

ちなみに、世界で広く使われている表記と U と V が逆のようである。プログラム全部書き終わってから気がついたので、申し訳ないがそのままにする。

シリアル版

まず、ある点におけるラプラシアンを返す関数 `laplacian` を用意しよう。中央差分で表現すると、上下左右の点との平均との差で表現すれば良いので、こう書ける。

```
double laplacian(int ix, int iy, vd &s) {
    double ts = 0.0;
    ts += s[ix - 1 + iy * L];
    ts += s[ix + 1 + iy * L];
    ts += s[ix + (iy - 1) * L];
    ts += s[ix + (iy + 1) * L];
    ts -= 4.0 * s[ix + iy * L];
    return ts;
}
```

また、 u と v の力学系の部分を計算する関数も作っておこう。

```
double calcU(double tu, double tv) {
    return tu * tu * tv - (F + k) * tu;
}

double calcV(double tu, double tv) {
    return -tu * tu * tv + F * (1.0 - tv);
}
```

さて、差分を計算する際、 $t + 1$ ステップ目の計算に t ステップの物理量を使う。もしここで t の値をどんどん更新してしまうと、ある場所の物理量を計算する時に t の値と $t + 1$ の値が混ざっておかしなことになる。実は、一次元拡散方程式ではそれを防ぐため、一度 t の時の値を別の領域にコピーして、それを使って $t + 1$ の値を計算するようにしていた(要するに手抜きである)。しかし、二次元でこれをやるとさすがにコピーのオーバーヘッドが大きい。そこで、同じ物理量を表す配列を二本ずつ用意して、奇数時刻と偶数時刻で使い分けることにしよう。具体的には u に対して u_2 という配列も用意しておく。いま偶数時刻だとすると u_2 から u を、奇数時刻なら u から u_2 を計算する。

というわけで、1ステップ時間発展を行う関数 `calc` はこう書ける。

```
void calc(vd &u, vd &v, vd &u2, vd &v2) {
    for (int iy = 1; iy < L - 1; iy++) {
        for (int ix = 1; ix < L - 1; ix++) {
            double du = 0;
            double dv = 0;
            const int i = ix + iy * L;
            du = Du * laplacian(ix, iy, u);
            dv = Dv * laplacian(ix, iy, v);
            du += calcU(u[i], v[i]);
            dv += calcV(u[i], v[i]);
            u2[i] = u[i] + du * dt;
            v2[i] = v[i] + dv * dt;
        }
    }
}
```

Gray-Scott 系は、最初に「種」を置いておくと、そこから模様が広がっていく系である。なので最初に種を置いておこう。

```
void init(vd &u, vd &v) {
    int d = 3;
    for (int i = L / 2 - d; i < L / 2 + d; i++) {
        for (int j = L / 2 - d; j < L / 2 + d; j++) {
            u[j + i * L] = 0.7;
        }
    }
    d = 6;
    for (int i = L / 2 - d; i < L / 2 + d; i++) {
        for (int j = L / 2 - d; j < L / 2 + d; j++) {
            v[j + i * L] = 0.9;
        }
    }
}
```

系の中央の u と v に、それぞれ 6×6 の領域、 12×12 の初期値を種として置くコードである。

以上を元に、時間発展を行う `main` 関数はこんな感じになる。

```
int main() {
    const int V = L * L;
```

```

vd u(V, 0.0), v(V, 0.0);
vd u2(V, 0.0), v2(V, 0.0);
init(u, v);
for (int i = 0; i < TOTAL_STEP; i++) {
    if (i & 1) {
        calc(u2, v2, u, v);
    } else {
        calc(u, v, u2, v2);
    }
    if (i % INTERVAL == 0) save_as_dat(u);
}
}

```

先程述べたように、偶数時刻と奇数時刻で二つの配列を使い分けているのに注意。

`save_as_dat` は、呼ばれるたびに配列を連番のファイル名で保存する関数である。

全体のコードはこんな感じになる。

gs.cpp

コンパイル、実行してみよう。

```

$ g++ -O3 gs.cpp
$ time ./a.out
conf000.dat
conf001.dat
conf002.dat
(snip)
conf097.dat
conf098.dat
conf099.dat
./a.out 1.61s user 0.03s system 96% cpu 1.697 total

```

出てきたデータ (*.dat) は、倍精度実数が $L \times L$ 個入っている。これを Ruby で読み込んで PNG 形式で吐くスクリプトを作っておこう。

image.rb

```

require "cairo"
require "pathname"

def convert(datfile)
    puts datfile
    buf = File.binread(datfile).unpack("d*")
    l = Math.sqrt(buf.size).to_i
    m = 4
    size = l * m

    surface = Cairo::ImageSurface.new(Cairo::FORMAT_RGB24, size, size)
    context = Cairo::Context.new(surface)

```

```

context.set_source_rgb(1, 1, 1)
context.rectangle(0, 0, size, size)
context.fill

l.times do |x|
  l.times do |y|
    u = buf[x + y * l]
    context.set_source_rgb(0, u, 0)
    context.rectangle(x * m, y * m, m, m)
    context.fill
  end
end
pngfile = Pathname(datfile).sub_ext(".png").to_s
surface.write_to_png(pngfile)
end

`ls *.dat`.split(/\n/).each do |f|
  convert(f)
end

```

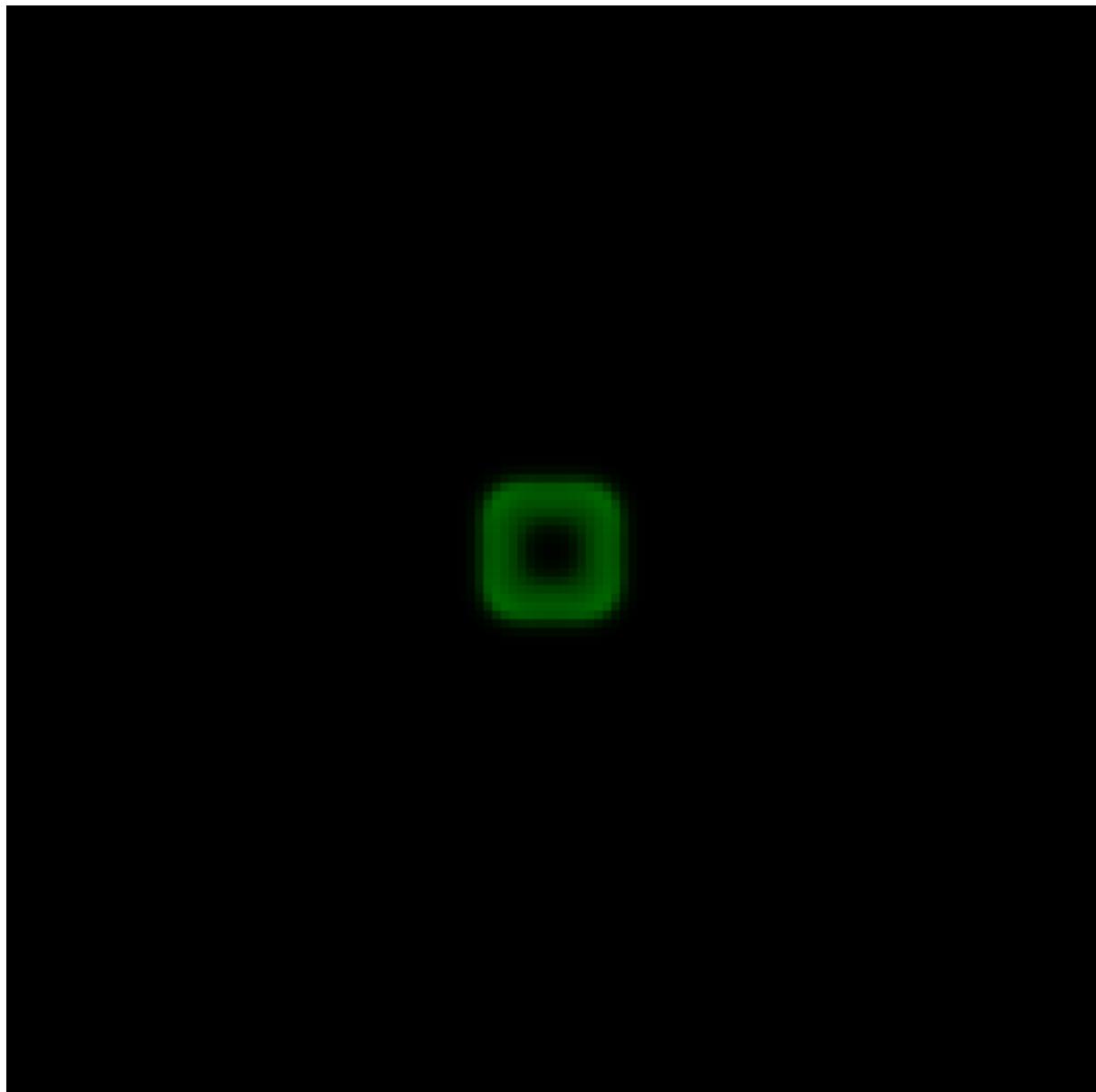
これで一括で処理する。

```

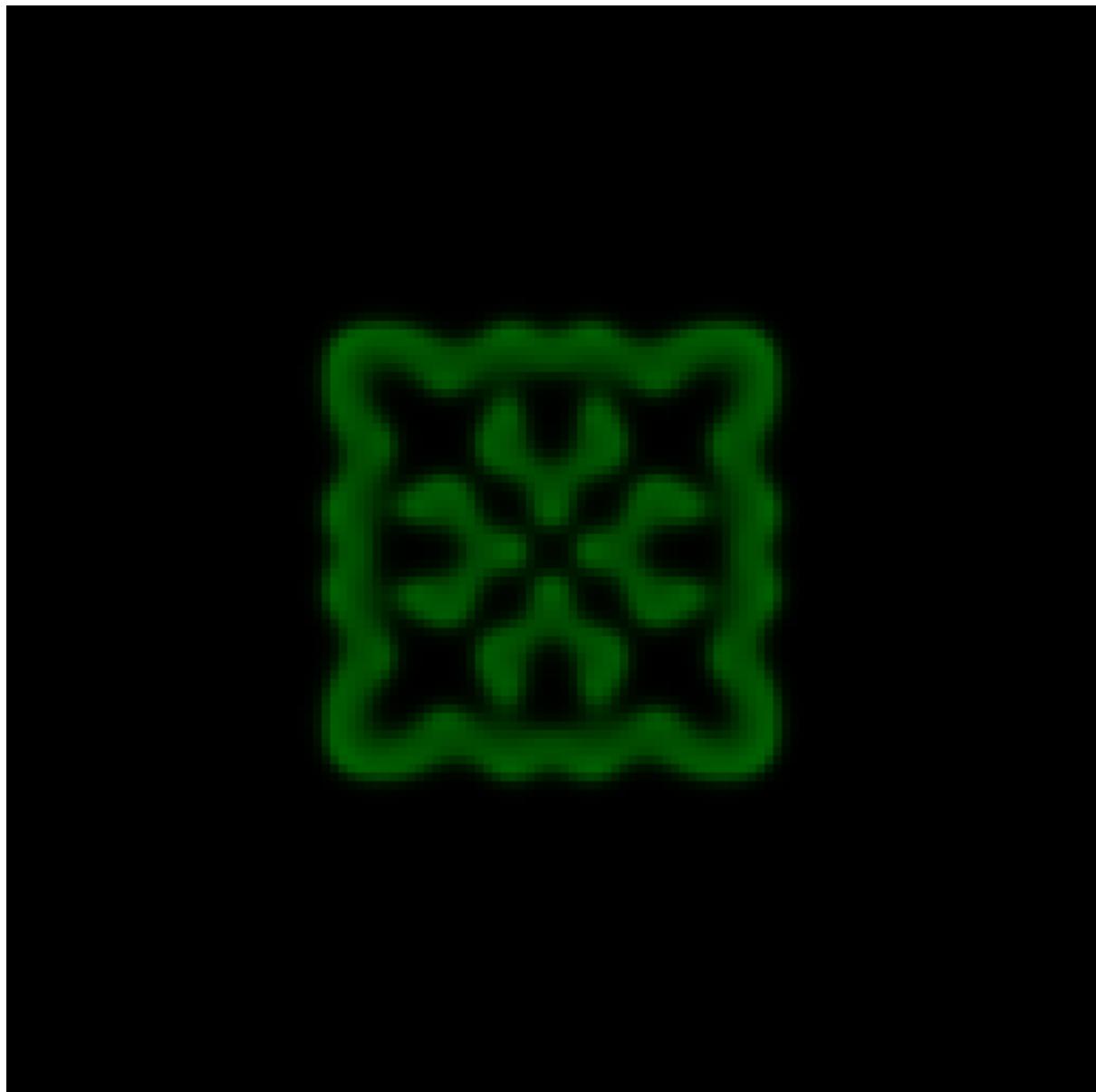
$ ruby image.rb
conf000.dat
conf001.dat
conf002.dat
(snip)
conf097.dat
conf098.dat
conf099.dat

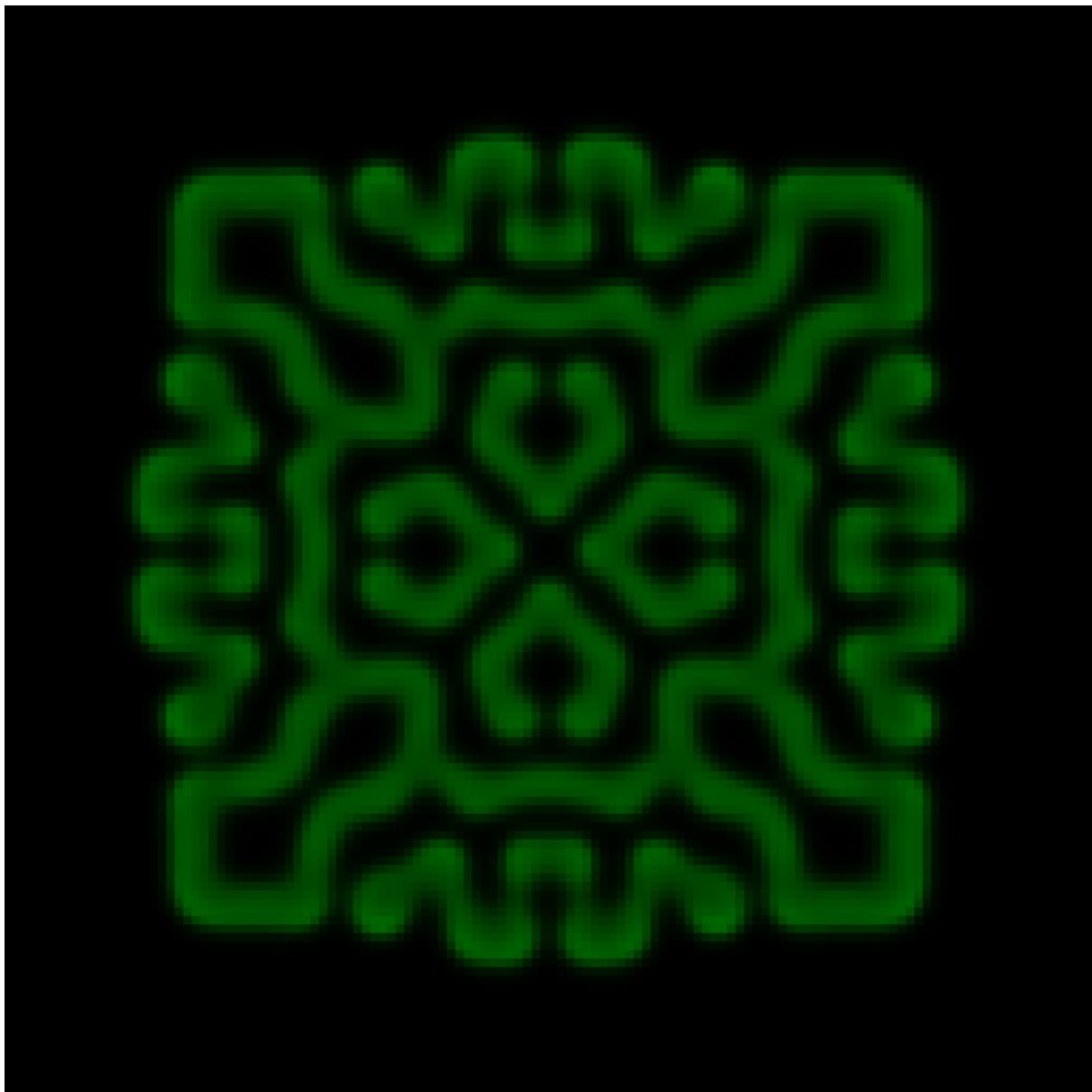
```

するとこんな感じの画像が得られる。



88





並列化ステップ 1: 通信の準備など

さて、さっそく反応拡散方程式を二次元領域分割により並列化していくわけだが、並列化で重要なのは、いきなり本番コードで通信アルゴリズムを試さないということである。まずは、今やろうとしている通信と同じアルゴリズムだけを抜き出したコードを書き、ちゃんと想定通りに通信できていることを確かめる。実際のデータは倍精度実数だが、とりあえず整数データでいろいろためそう。

まず、通信関連のコードを書く前に、領域分割により、全体をどうやって分割するか、各プロセスはどこを担当するかといった基本的なセットアップを確認しよう。

いま、 $L \times L$ のグリッドがあるとしよう。これを `procs` プロセスで分割する。この時、なるべく「のりしろ」が小さくなるように分割したい。例えば 4 プロセスなら 2×2 に、24 プロセスなら 6×4 という具合である。このためには、与えられたプロセス数を、なるべく似たような数字になるように因数分解してやらないといけない。MPI にはそのための関数、`MPI_Dims_create` が用意されている。使い方は、二次元分割なら、`procs` にプロセス数が入っているとして、

```

int d2[2] = {};
MPI_Dims_create(procs, 2, d2);

```

のように呼ぶと、d2[0] と d2[1] に分割数が入ってくる。三次元分割をしたければ、

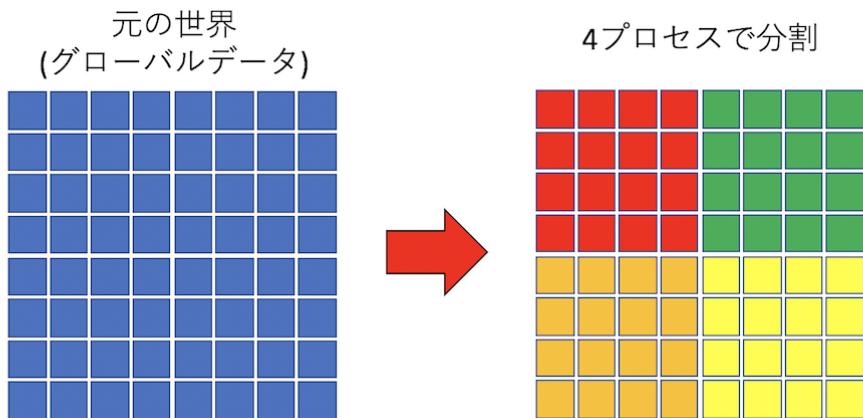
```

int d3[3] = {};
MPI_Dims_create(procs, 3, d3);

```

などと、分割数 3 を指定し、3 要素の配列を食わせてやれば良い。ただし、OpenMPI の MPI_Dims_create は若干動作が怪しいので注意すること。例えば 9 プロセスを二次元分割したら 3x3 になってほしいが、9x1 を返してくる。Intel MPI や SGI MPT はちゃんと 3x3 を返してくるので、このあたりは実装依存のようだ。気になる場合は自分で因数分解コードを書いて欲しい。

さて、procs プロセスを、GX*GY と分割することにしよう。すると、各プロセスは、横が L/GX、縦が L/GY 個のサイトを保持すれば良い。例えば 8x8 の系を 4 プロセスで並列化する際、一つのプロセスが担当するのは 4x4 となるが、上下左右に 1 列余分に必要になるので、合わせて 6x6 のデータを保持することになる。



各プロセスは「のりしろ」部分込みの領域を保持する

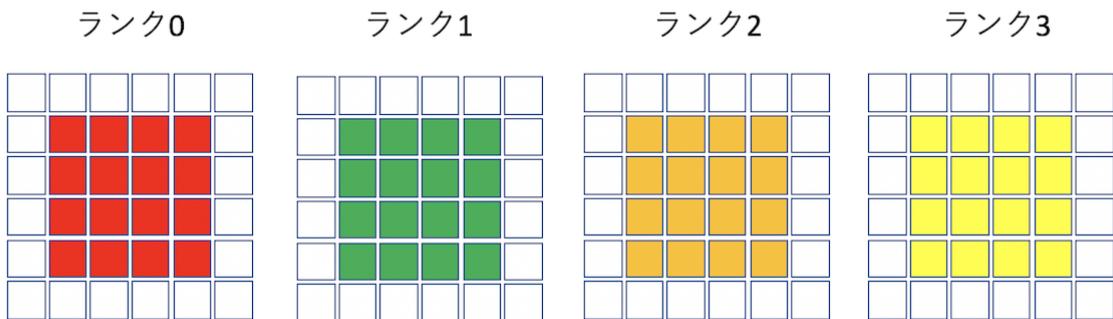


図 1: fig/margin.png

また、各プロセスは自分がどの場所を担当しているかも知っておきたいし、担当する領域のサイズも保持しておきたい。これらに加えてランクや総プロセス数といった並列化情報を、MPIinfo という構造体にまとめて突っ込んで置こう。とりあえず必要な情報はこんな感じだろうか。

```

struct MPIinfo {
    int rank; //ランク番号

```

```

int procs; //総プロセス数
int GX, GY; // プロセスをどう分割したか (GX*GY=procs)
int local_grid_x, local_grid_y; // 自分が担当する位置
int local_size_x, local_size_y; // 自分が担当する領域のサイズ (のりしろ含まず)
};

MPIinfo の各値をセットする関数、setup_info を作っておこう。こんな感じかな。

```

```

void setup_info(MPIinfo &mi) {
    int rank = 0;
    int procs = 0;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &procs);
    int d2[2] = {};
    MPI_Dims_create(procs, 2, d2);
    mi.rank = rank;
    mi.procs = procs;
    mi.GX = d2[0];
    mi.GY = d2[1];
    mi.local_grid_x = rank % mi.GX;
    mi.local_grid_y = rank / mi.GX;
    mi.local_size_x = L / mi.GX;
    mi.local_size_y = L / mi.GY;
}

```

自分が保持するデータを `std::vector<int> local_data` として宣言しよう。のりしろの部分も考慮するところな感じになる。

```

MPIinfo mi;
setup_info(mi);
std::vector<int> local_data((mi.local_size_x + 2) * (mi.local_size_y + 2), 0);

```

あの通信がうまくいっているか確認するため、ローカルデータに「のりしろ」以外に通し番号を降っておこう。例えば $L=8$ で、 $procs = 4$ の場合に、各プロセスにこういうデータを保持させたい。

```

rank = 0
000 000 000 000 000 000
000 000 001 002 003 000
000 004 005 006 007 000
000 008 009 010 011 000
000 012 013 014 015 000
000 000 000 000 000 000

```

```

rank = 1
000 000 000 000 000 000
000 016 017 018 019 000
000 020 021 022 023 000
000 024 025 026 027 000
000 028 029 030 031 000
000 000 000 000 000 000

```

```

rank = 2
000 000 000 000 000 000
000 032 033 034 035 000
000 036 037 038 039 000
000 040 041 042 043 000
000 044 045 046 047 000
000 000 000 000 000 000

```

```

rank = 3
000 000 000 000 000 000
000 048 049 050 051 000
000 052 053 054 055 000
000 056 057 058 059 000
000 060 061 062 063 000
000 000 000 000 000 000

```

このような初期化をする関数 `init` を用意する。

```

void init(std::vector<int> &local_data, MPIinfo &mi) {
    const int offset = mi.local_size_x * mi.local_size_y * mi.rank;
    for (int iy = 0; iy < mi.local_size_y; iy++) {
        for (int ix = 0; ix < mi.local_size_x; ix++) {
            int index = (ix + 1) + (iy + 1) * (mi.local_size_x + 2);
            int value = ix + iy * mi.local_size_x + offset;
            local_data[index] = value;
        }
    }
}

```

自分が担当する領域の左上に来る番号を `offset` として計算し、そこから通し番号を降っているだけである。このローカルなデータをダンプする関数も作っておく。

```

void dump_local_sub(std::vector<int> &local_data, MPIinfo &mi) {
    printf("rank = %d\n", mi.rank);
    for (int iy = 0; iy < mi.local_size_y + 2; iy++) {
        for (int ix = 0; ix < mi.local_size_x + 2; ix++) {
            unsigned int index = ix + iy * (mi.local_size_x + 2);
            printf("%03d ", local_data[index]);
        }
        printf("\n");
    }
    printf("\n");
}

```

`dump_local_sub` に自分が保持する `std::vector` を渡せば表示されるのだが、複数のプロセスから一気に標準出力に吐くと表示が乱れる可能性がある。各プロセスからファイルに吐いてしまっても良いが、こういう時は、プロセスの数だけループをまわし、ループカウンタが自分のランク番号と同じになった時に書き込む、というコードが便利である。全プロセスが順番待ちをするので速度は遅いが、主にデバッグに使

うので問題ない。こんな感じである。

```
void dump_local(std::vector<int> &local_data, MPIinfo &mi) {
    for (int i = 0; i < mi.procs; i++) {
        MPI_Barrier(MPI_COMM_WORLD);
        if (i == mi.rank) {
            dump_local_sub(local_data, mi);
        }
    }
}
```

毎回バリア同期が必要なことに注意。この、

```
for (int i = 0; i < procs; i++) {
    MPI_Barrier(MPI_COMM_WORLD);
    if (i == rank) {
        do_something();
    }
}
```

というイディオムは、MPIで頻出するので覚えておくと良いかもしれない。4プロセス実行し、dump_localを呼ぶと、先程の「のりしろ付きのローカルデータ」がダンプされる。

並列化ステップ2: データの保存

計算を実行するにあたり、必要な通信は、

- 時間発展のための「のりしろ」の通信
- 計算の途中経過のデータの保存のための集団通信

の二種類である。一次元分割の時と同様に、まずは後者、データの保存のための通信を考えよう。

時間発展した結果を保存したいので、各プロセスが保持するデータを集約したい。各プロセスが保持するデータをローカルデータ、系全体のデータをグローバルデータと呼ぶことにする。「のりしろ」は計算の時には必要だが、データの保存の時には不要だ。なので、各プロセスはまず、ローカルデータから「のりしろ」を除いたデータを用意し、それをMPI_Gatherを使ってルートプロセスに集める。

今、各プロセスがこんな感じにデータを持っていたとする。

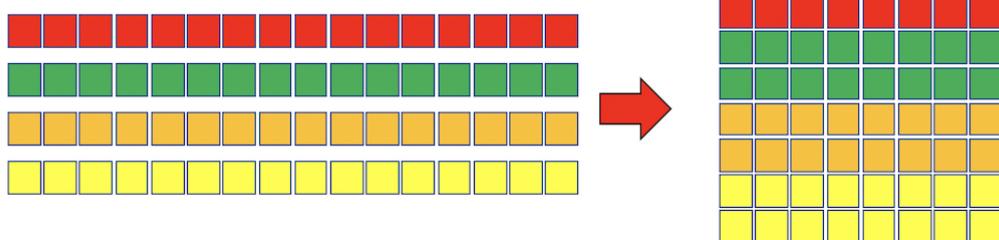
```
rank = 0
000 000 000 000 000 000
000 000 001 002 003 000
000 004 005 006 007 000
000 008 009 010 011 000
000 012 013 014 015 000
000 000 000 000 000 000
```

```
rank = 1
000 000 000 000 000 000
000 016 017 018 019 000
000 020 021 022 023 000
```

1. 送信バッファの作成



2. MPI_Gatherで集める



3. 受け取ったデータを並び替える

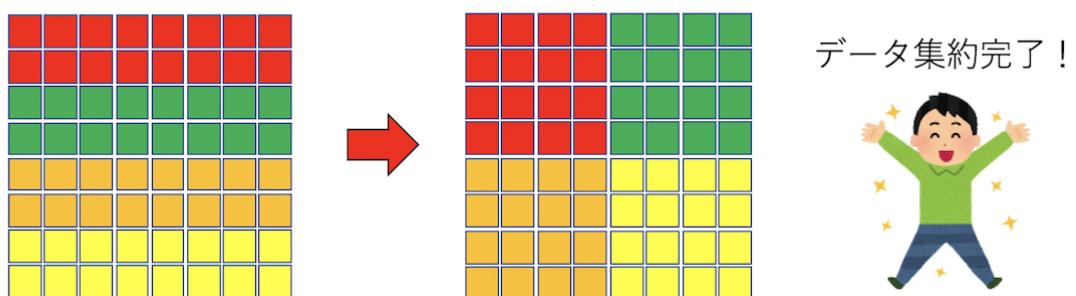


図 2: fig/gather.png

```
000 024 025 026 027 000  
000 028 029 030 031 000  
000 000 000 000 000 000
```

```
rank = 2  
000 000 000 000 000 000  
000 032 033 034 035 000  
000 036 037 038 039 000  
000 040 041 042 043 000  
000 044 045 046 047 000  
000 000 000 000 000 000
```

```
rank = 3  
000 000 000 000 000 000  
000 048 049 050 051 000  
000 052 053 054 055 000  
000 056 057 058 059 000  
000 060 061 062 063 000  
000 000 000 000 000 000
```

000 は「のりしろ」である。グローバル領域は二次元的に分割されるが、各プロセスはそれを一次元的に保持しているので、「のりしろ」を除いてデータをコピーするところを除けば、通信部分は一次元の時と同じになる。

```
void gather(std::vector<int> &local_data, MPIinfo &mi) {  
    const int lx = mi.local_size_x;  
    const int ly = mi.local_size_y;  
    std::vector<int> sendbuf(lx * ly);  
    // 「のりしろ」を除いたデータのコピー  
    for (int iy = 0; iy < ly; iy++) {  
        for (int ix = 0; ix < lx; ix++) {  
            int index_from = (ix + 1) + (iy + 1) * (lx + 2);  
            int index_to = ix + iy * lx;  
            sendbuf[index_to] = local_data[index_from];  
        }  
    }  
    std::vector<int> recvbuf;  
    if (mi.rank == 0) {  
        recvbuf.resize(lx * ly * mi.procs);  
    }  
    MPI_Gather(sendbuf.data(), lx * ly, MPI_INT, recvbuf.data(), lx * ly, MPI_INT, 0, MPI_COMM_WORLD)  
    // ここで、ランク 0 番のプロセスの保持する recvbuf にグローバルデータが入る。  
}
```

しかし、このようにして集約されたグローバルデータは、各プロセスが論理的に保持するデータと場所が異なり、こんな感じになる。

```
Before reordering  
000 001 002 003 004 005 006 007
```

```

008 009 010 011 012 013 014 015
016 017 018 019 020 021 022 023
024 025 026 027 028 029 030 031
032 033 034 035 036 037 038 039
040 041 042 043 044 045 046 047
048 049 050 051 052 053 054 055
056 057 058 059 060 061 062 063

```

数字が連番になっているのがわかるだろうか。デバッグに便利なように、そうなるようにローカルデータに数字を振っておいた。さて、論理的にはこういう配置になっていて欲しい。

After reordering

```

000 001 002 003 016 017 018 019
004 005 006 007 020 021 022 023
008 009 010 011 024 025 026 027
012 013 014 015 028 029 030 031
032 033 034 035 048 049 050 051
036 037 038 039 052 053 054 055
040 041 042 043 056 057 058 059
044 045 046 047 060 061 062 063

```

というわけで、そうなるようにデータを並び替えればよい。並び替えのための関数 `reordering` はこう書けるだろう。

```

void reordering(std::vector<int> &v, MPIinfo &mi) {
    std::vector<int> v2(v.size());
    std::copy(v.begin(), v.end(), v2.begin());
    const int lx = mi.local_size_x;
    const int ly = mi.local_size_y;
    int i = 0;
    for (int r = 0; r < mi.procs; r++) {
        int rx = r % mi.GX;
        int ry = r / mi.GX;
        int sx = rx * lx;
        int sy = ry * ly;
        for (int iy = 0; iy < ly; iy++) {
            for (int ix = 0; ix < lx; ix++) {
                int index = (sx + ix) + (sy + iy) * L;
                v[index] = v2[i];
                i++;
            }
        }
    }
}

```

以上の処理まで含めて、`gather` 完成である。

```

void gather(std::vector<int> &local_data, MPIinfo &mi) {
    const int lx = mi.local_size_x;
    const int ly = mi.local_size_y;

```

```

std::vector<int> sendbuf(lx * ly);
// 「のりしろ」を除いたデータのコピー
for (int iy = 0; iy < ly; iy++) {
    for (int ix = 0; ix < lx; ix++) {
        int index_from = (ix + 1) + (iy + 1) * (lx + 2);
        int index_to = ix + iy * lx;
        sendbuf[index_to] = local_data[index_from];
    }
}
std::vector<int> recvbuf;
if (mi.rank == 0) {
    recvbuf.resize(lx * ly * mi.procs);
}
MPI_Gather(sendbuf.data(), lx * ly, MPI_INT, recvbuf.data(), lx * ly, MPI_INT, 0, MPI_COMM_WORLD);
if (mi.rank == 0) {
    printf("Before reordering\n");
    dump_global(recvbuf);
    reordering(recvbuf, mi);
    printf("After reordering\n");
    dump_global(recvbuf);
}
}

```

送信前や送信後にデータの処理が必要となるので、やってることが単純なわりにコード量がそこそこの長さになる。このあたりが「MPIは面倒くさい」と言われる所以かもしれない。筆者も「MPIは面倒くさい」ことは否定しない。しかし、ここまで読んでくださった方なら「MPIは難しくはない」ということも同意してもらえると思う。MPIは書いた通りに動く。なので、通信アルゴリズムが決まっていれば、その手順どおりに書くだけである。実際面倒なのは通信そのものよりも、通信の前処理と後処理だったりする(そもそも今回も通信は一行だけだ)。

以上をすべてまとめたコードは以下の通り。

gather2d.cpp

main 関数だけ書いておくとこんな感じ。

```

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    MPIInfo mi;
    setup_info(mi);
    // ローカルデータの確保
    std::vector<int> local_data((mi.local_size_x + 2) * (mi.local_size_y + 2), 0);
    // ローカルデータの初期化
    init(local_data, mi);
    // ローカルデータの表示
    dump_local(local_data, mi);
    // ローカルデータを集約してグローバルデータに
    gather(local_data, mi);
    MPI_Finalize();
}

```

}

まあ、そのまま手続きを書いただけですね。

並列化ステップ2: のりしろの通信

さて、計算を実行するためには、上下左右のプロセスから自分の「のりしろ」に情報を受け取らないといけない。問題は、二次元の場合には角の情報、つまり「斜め方向」の通信も必要なことである。普通に考えると、左右2回、上下2回、角4つで8回の通信が必要となるのだが、左右から受け取ったデータを、上下に転送することで、4回の通信で斜め方向の通信も完了する。

どうでも良いが筆者は昔ブログを書いており(今は書いてないが)、「斜め方向の通信どうするかなあ」と書いたら、日記の読者二人から別々にこのアルゴリズムを教えていただいた(その節はありがとうございます)。ブログも書いてみるものある。

データの転送を図解するとこんな感じになる。まず、左右方向の通信。実際の例では 2×2 分割のため、自分から見て左にいるプロセスと右にいるプロセスが同一になってしまふが、図では別プロセスとして描いているから注意。

左から受け取り、右に送る通信

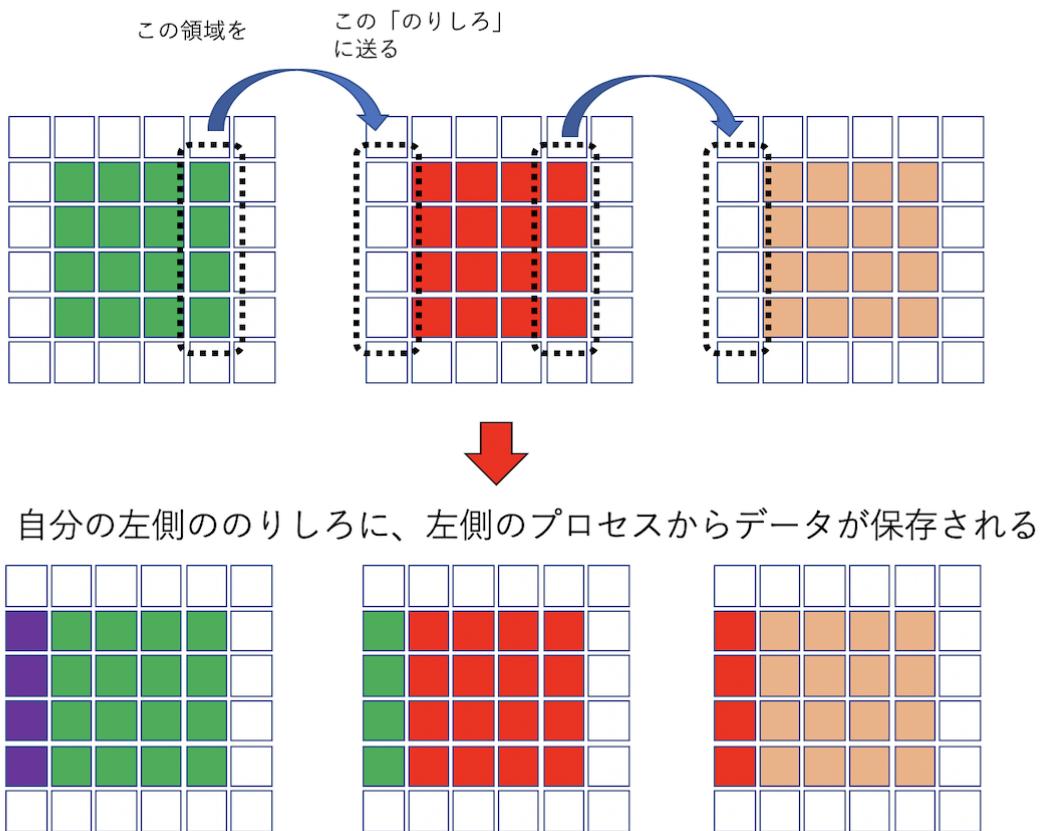


図 3: fig/sendrecv_x.png

左右の通信が終わったら、左右からもらったデータも込みで上下に転送する。以下は、「下から受け取り、上に送る」通信。

1. 左右の通信が終わった状態 2. 左右からもらったデータごと
 上に送る(下からももらう)
 2. 上辺に加え、斜めのデータも
 送ることができた

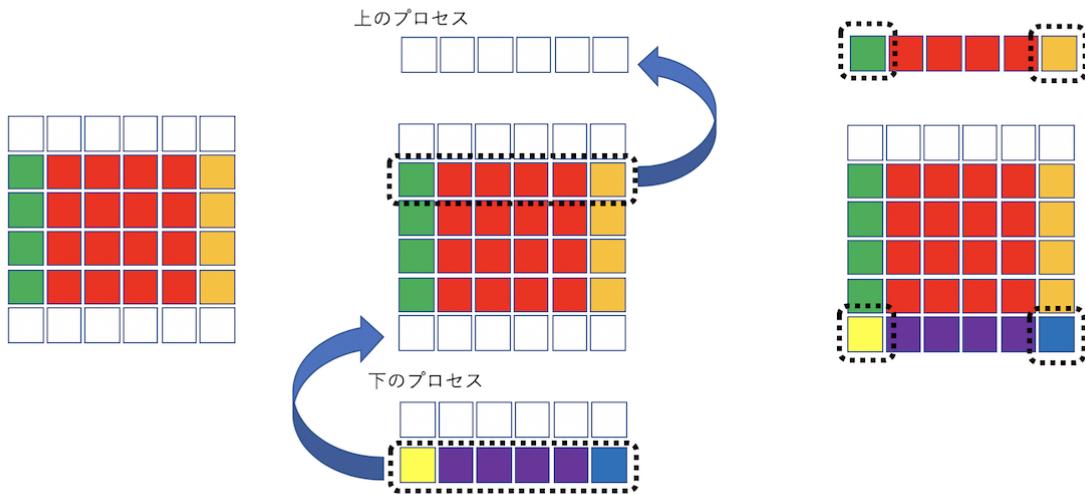


図 4: fig/sendrecv_y.png

最後の点線で囲ったデータが「斜め方向のプロセスが保持していたデータ」であり、間接的に受け取ったことになる。

まず、上下左右にいるプロセス番号を知りたい。MPIinfo に get_rank メソッドを追加しておこう。

```
struct MPIinfo {
    int rank;
    int procs;
    int GX, GY;
    int local_grid_x, local_grid_y;
    int local_size_x, local_size_y;
    // 自分から見て (dx, dy) だけ離れたプロセスの rank を返す
    int get_rank(int dx, int dy) {
        int rx = (local_grid_x + dx + GX) % GX;
        int ry = (local_grid_y + dy + GY) % GY;
        return rx + ry * GX;
    }
};
```

これを使って、左右(x 方向)に通信して、右と左の「のりしろ」データを交換するコードはこんな感じに書ける。

```
void sendrecv_x(std::vector<int> &local_data, MPIinfo &mi) {
    const int lx = mi.local_size_x;
    const int ly = mi.local_size_y;
    std::vector<int> sendbuf(ly);
    std::vector<int> recvbuf(ly);
    int left = mi.get_rank(-1, 0);
    int right = mi.get_rank(1, 0);
```

```

for (int i = 0; i < ly; i++) {
    int index = lx + (i + 1) * (lx + 2);
    sendbuf[i] = local_data[index];
}
MPI_Status st;
MPI_Sendrecv(sendbuf.data(), ly, MPI_INT, right, 0,
             recvbuf.data(), ly, MPI_INT, left, 0, MPI_COMM_WORLD, &st);
for (int i = 0; i < ly; i++) {
    int index = (i + 1) * (lx + 2);
    local_data[index] = recvbuf[i];
}

for (int i = 0; i < ly; i++) {
    int index = 1 + (i + 1) * (lx + 2);
    sendbuf[i] = local_data[index];
}
MPI_Sendrecv(sendbuf.data(), ly, MPI_INT, left, 0,
             recvbuf.data(), ly, MPI_INT, right, 0, MPI_COMM_WORLD, &st);
for (int i = 0; i < ly; i++) {
    int index = lx + 1 + (i + 1) * (lx + 2);
    local_data[index] = recvbuf[i];
}
}

```

全く同様に y 方向の通信も書けるが、先に述べたように「左右からもらったデータも転送」するため、その分がちょっとだけ異なる。

このアルゴリズムを実装するとこんな感じになる。

sendrecv.cpp

実行結果はこんな感じ。

```
$ mpic++ sendrecv.cpp
$ mpirun -np 4 ./a.out
```

```
# 通信前
rank = 0
000 000 000 000 000 000
000 000 001 002 003 000
000 004 005 006 007 000
000 008 009 010 011 000
000 012 013 014 015 000
000 000 000 000 000 000
```

```
rank = 1
000 000 000 000 000 000
000 016 017 018 019 000
000 020 021 022 023 000
```

```
000 024 025 026 027 000  
000 028 029 030 031 000  
000 000 000 000 000 000
```

```
rank = 2  
000 000 000 000 000 000  
000 032 033 034 035 000  
000 036 037 038 039 000  
000 040 041 042 043 000  
000 044 045 046 047 000  
000 000 000 000 000 000
```

```
rank = 3  
000 000 000 000 000 000  
000 048 049 050 051 000  
000 052 053 054 055 000  
000 056 057 058 059 000  
000 060 061 062 063 000  
000 000 000 000 000 000
```

```
# 左右の通信終了後
```

```
rank = 0  
000 000 000 000 000 000  
019 000 001 002 003 016  
023 004 005 006 007 020  
027 008 009 010 011 024  
031 012 013 014 015 028  
000 000 000 000 000 000
```

```
rank = 1  
000 000 000 000 000 000  
003 016 017 018 019 000  
007 020 021 022 023 004  
011 024 025 026 027 008  
015 028 029 030 031 012  
000 000 000 000 000 000
```

```
rank = 2  
000 000 000 000 000 000  
051 032 033 034 035 048  
055 036 037 038 039 052  
059 040 041 042 043 056  
063 044 045 046 047 060  
000 000 000 000 000 000
```

```
rank = 3
```

```
000 000 000 000 000 000  
035 048 049 050 051 032  
039 052 053 054 055 036  
043 056 057 058 059 040  
047 060 061 062 063 044  
000 000 000 000 000 000
```

上下の通信終了後 (これで斜め方向も完了)

```
rank = 0  
063 044 045 046 047 060  
019 000 001 002 003 016  
023 004 005 006 007 020  
027 008 009 010 011 024  
031 012 013 014 015 028  
051 032 033 034 035 048
```

```
rank = 1  
047 060 061 062 063 044  
003 016 017 018 019 000  
007 020 021 022 023 004  
011 024 025 026 027 008  
015 028 029 030 031 012  
035 048 049 050 051 032
```

```
rank = 2  
031 012 013 014 015 028  
051 032 033 034 035 048  
055 036 037 038 039 052  
059 040 041 042 043 056  
063 044 045 046 047 060  
019 000 001 002 003 016
```

```
rank = 3  
015 028 029 030 031 012  
035 048 049 050 051 032  
039 052 053 054 055 036  
043 056 057 058 059 040  
047 060 061 062 063 044  
003 016 017 018 019 000
```

先の図と比べて、正しく通信が行われていることを確認してほしい。

結局、通信プログラムとはこういうことをする。

- 送信バッファと受信バッファを用意する
- 送信バッファに送るべきデータをコピー
- 通信する

- 受信バッファに来たデータを必要な場所にコピー

通信そのものは関数呼び出し一発で難しくも面倒でもないが、送受信バッファの作業が面倒くさい。

並列化ステップ3: 並列コードの実装

通信に使うアルゴリズムの確認が終わったので、いよいよ差分法コードに実装してみよう。まず、初期化の部分を考えないといけない。初期化についてはグローバル座標で考えたいが、実際に値を入れるのは各プロセスが保持するローカルデータである。そこで、「このグローバル座標が自分の領域に含まれるか？」「含まれるなら、そのインデックスはどこか？」が知りたくなる。それを MPIinfo のメソッドとして追加しておこう。

```

struct MPIinfo {
    int rank;
    int procs;
    int GX, GY;
    int local_grid_x, local_grid_y;
    int local_size_x, local_size_y;

    // 自分から見て +dx, +dy だけずれたプロセスのランクを返す
    int get_rank(int dx, int dy) {
        int rx = (local_grid_x + dx + GX) % GX;
        int ry = (local_grid_y + dy + GY) % GY;
        return rx + ry * GX;
    }

    // 自分の領域に含まれるか
    bool is_inside(int x, int y) {
        int sx = local_size_x * local_grid_x;
        int sy = local_size_y * local_grid_y;
        int ex = sx + local_size_x;
        int ey = sy + local_size_y;
        if (x < sx)return false;
        if (x >= ex)return false;
        if (y < sy)return false;
        if (y >= ey)return false;
        return true;
    }

    // グローバル座標をローカルインデックスに
    int g2i(int gx, int gy) {
        int sx = local_size_x * local_grid_x;
        int sy = local_size_y * local_grid_y;
        int x = gx - sx;
        int y = gy - sy;
        return (x + 1) + (y + 1) * (local_size_x + 2);
    }
};

```

そうすると、初期化処理はこんな感じにかける。

```
void init(vd &u, vd &v, MPIinfo &mi) {
    int d = 3;
    for (int i = L / 2 - d; i < L / 2 + d; i++) {
        for (int j = L / 2 - d; j < L / 2 + d; j++) {
            if (!mi.is_inside(i, j)) continue;
            int k = mi.g2i(i, j);
            u[k] = 0.7;
        }
    }
    d = 6;
    for (int i = L / 2 - d; i < L / 2 + d; i++) {
        for (int j = L / 2 - d; j < L / 2 + d; j++) {
            if (!mi.is_inside(i, j)) continue;
            int k = mi.g2i(i, j);
            v[k] = 0.9;
        }
    }
}
```

要するにグローバル座標でループを回してしまって、自分の領域に入っていたら (`mi.is_inside(i, j)==true`)、ローカルインデックスを取得して、そこに値を書き込む、というだけのコードである。自分の守備範囲外もループが回って非効率に思えるかもしれないが、どうせ初期化処理は最初に一度しか走らないし、こうしておくと他の初期化をしたい時や、ファイルから読み込む時に、シリアルコードと並列コードで同じファイルが使えたりして便利である。

初期化処理が済んだら、可視化用のファイル保存コードを書こう。といっても、ステップ2で書いたコードを `int` から `double` に変えて、標準出力にダンプしていたのをファイルに保存するだけである。

```
// 各プロセスから保存用のデータを受け取ってセーブ
void save_as_dat_mpi(vd &local_data, MPIinfo &mi) {
    const int lx = mi.local_size_x;
    const int ly = mi.local_size_y;
    vd sendbuf(lx * ly);
    // 「のりしろ」を除いたデータのコピー
    for (int iy = 0; iy < ly; iy++) {
        for (int ix = 0; ix < lx; ix++) {
            int index_from = (ix + 1) + (iy + 1) * (lx + 2);
            int index_to = ix + iy * lx;
            sendbuf[index_to] = local_data[index_from];
        }
    }
    vd recvbuf;
    if (mi.rank == 0) {
        recvbuf.resize(lx * ly * mi.procs);
    }
    MPI_Gather(sendbuf.data(), lx * ly, MPI_DOUBLE, recvbuf.data(), lx * ly, MPI_DOUBLE, 0, MPI_COMM_
```

```

    reorder(reorderbuf, mi);
    save_as_dat(reorderbuf);
}
}

```

データの再配置 (reordering) もほとんど同じなので割愛。ここで、いきなり時間発展させずに初期化処理をしてからファイルに保存し、正しく初期化、保存できているか確認しておこう。

「のりしろ」の通信部分も、基本的に int を double に変更するだけなので割愛。ただし、u と v の両方を通信しないといけないので、それをまとめて行う関数を作つておこう。

```

void sendrecv(vd &u, vd &v, MPIinfo &mi) {
    sendrecv_x(u, mi);
    sendrecv_y(u, mi);
    sendrecv_x(v, mi);
    sendrecv_y(v, mi);
}

```

これを時間発展直前に呼び出せば、「のりしろ」部分の通信が完了している。ここでも、いきなり時間発展させずに初期化処理を行った後に「のりしろ通信」を行い、ローカルデータをダンプして正しく通信できているか確認しよう。

そこまでできればあとはシリアル版とほぼ同じ。main 関数はこんな感じになる。

```

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    MPIinfo mi;
    setup_info(mi);
    const int V = (mi.local_size_x + 2) * (mi.local_size_y + 2);
    vd u(V, 0.0), v(V, 0.0);
    vd u2(V, 0.0), v2(V, 0.0);
    init(u, v, mi);
    for (int i = 0; i < TOTAL_STEP; i++) {
        if (i & 1) {
            sendrecv(u2, v2, mi);
            calc(u2, v2, u, v, mi);
        } else {
            sendrecv(u, v, mi);
            calc(u, v, u2, v2, mi);
        }
        if (i % INTERVAL == 0) save_as_dat_mpi(u, mi);
    }
    MPI_Finalize();
}

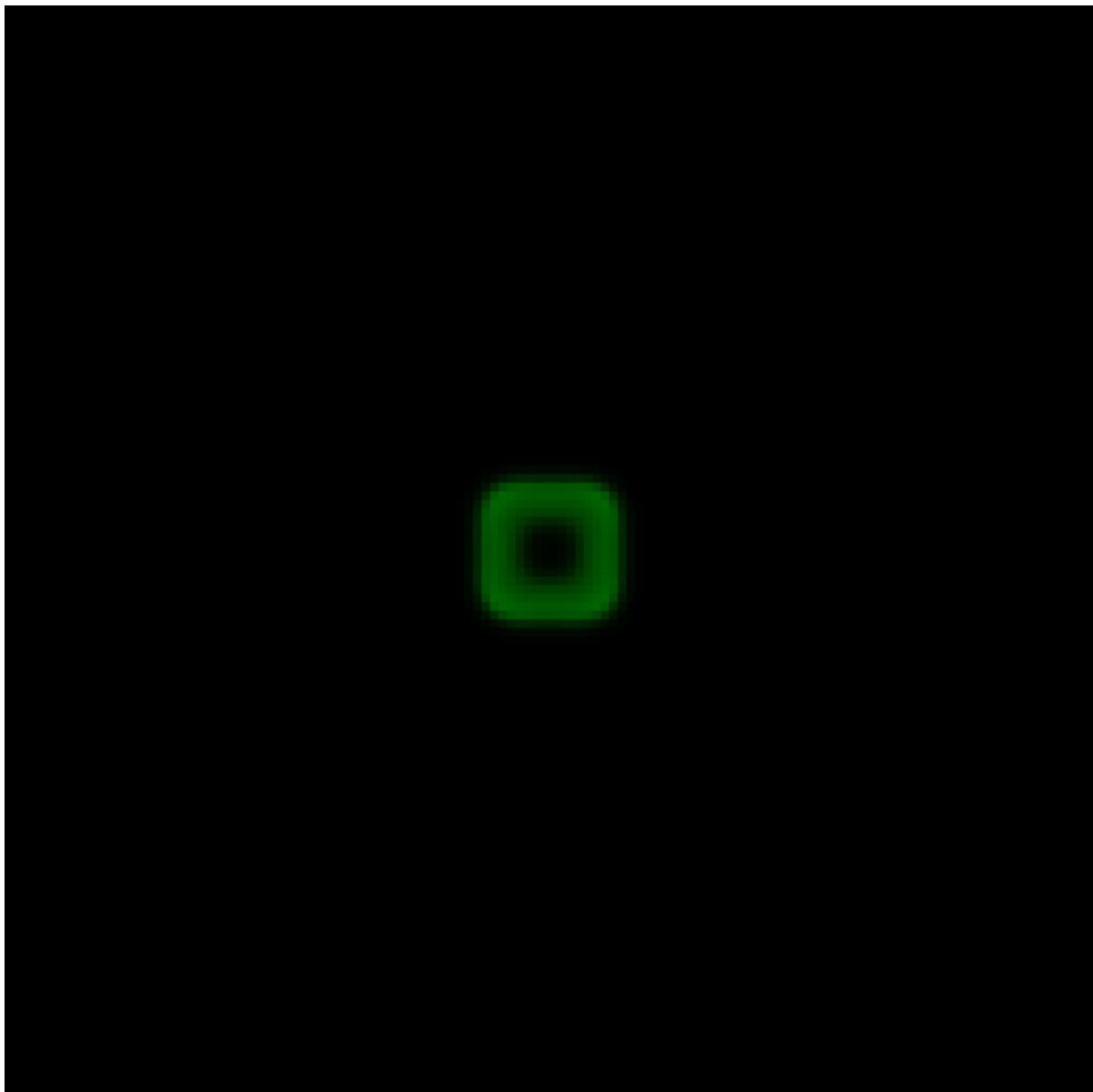
```

MPI の初期化、終了処理、および計算の直前に通信を呼んでるところ以外はシリアル版と変わらないことがわかる。

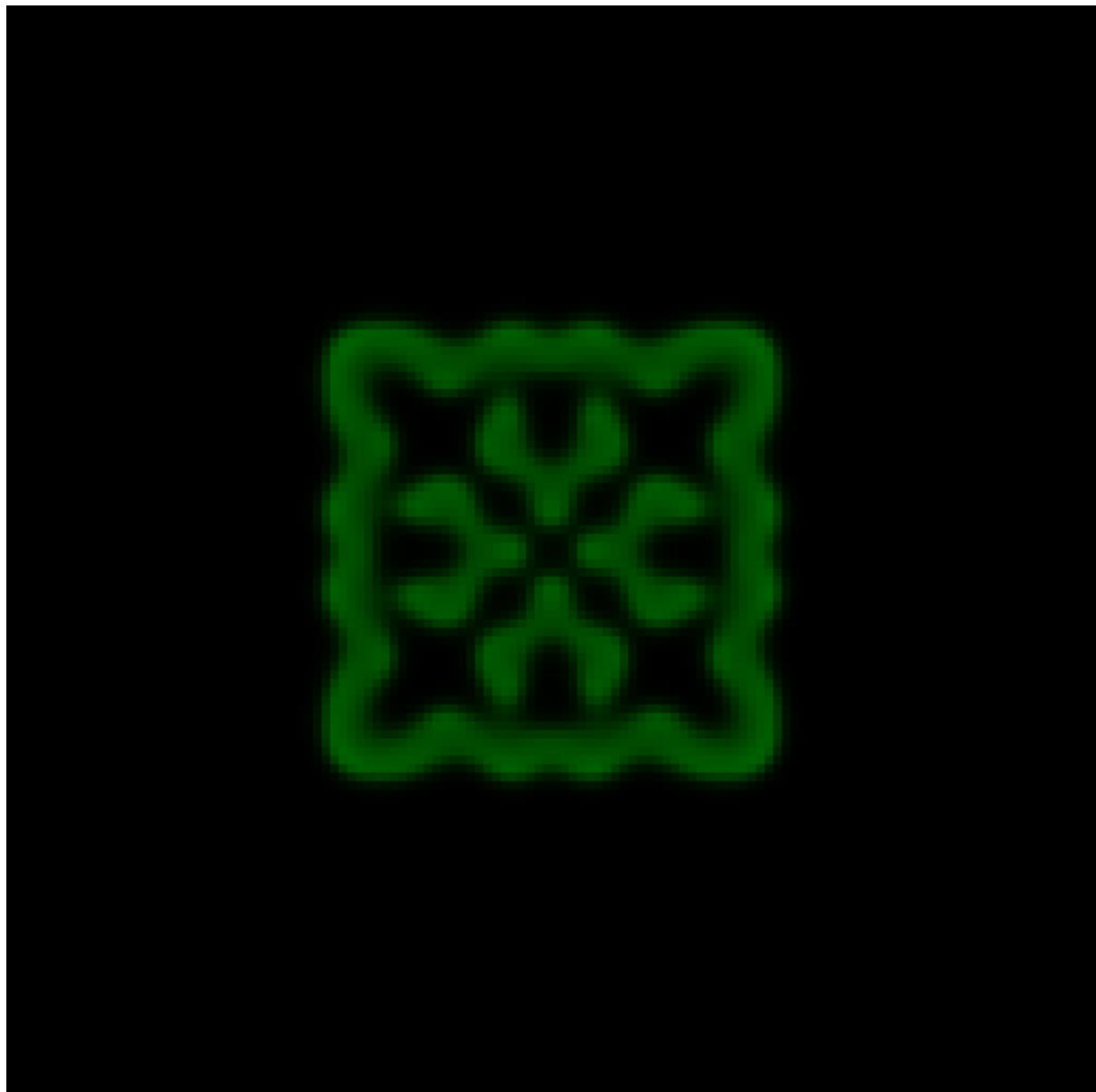
実行してみよう。普通の mpic++ を使ってしまうと clang++ が呼ばれてしまう。先程、g++ でコンパイルしたシリアル版と実行時間を比較するため、明示的に g++ でコンパイルして実行しよう。筆者の環境では MPI のヘッダやライブラリにパスが通っているので、-lmpi -lmpi_cxx をつけるだけでコンパイルできる。

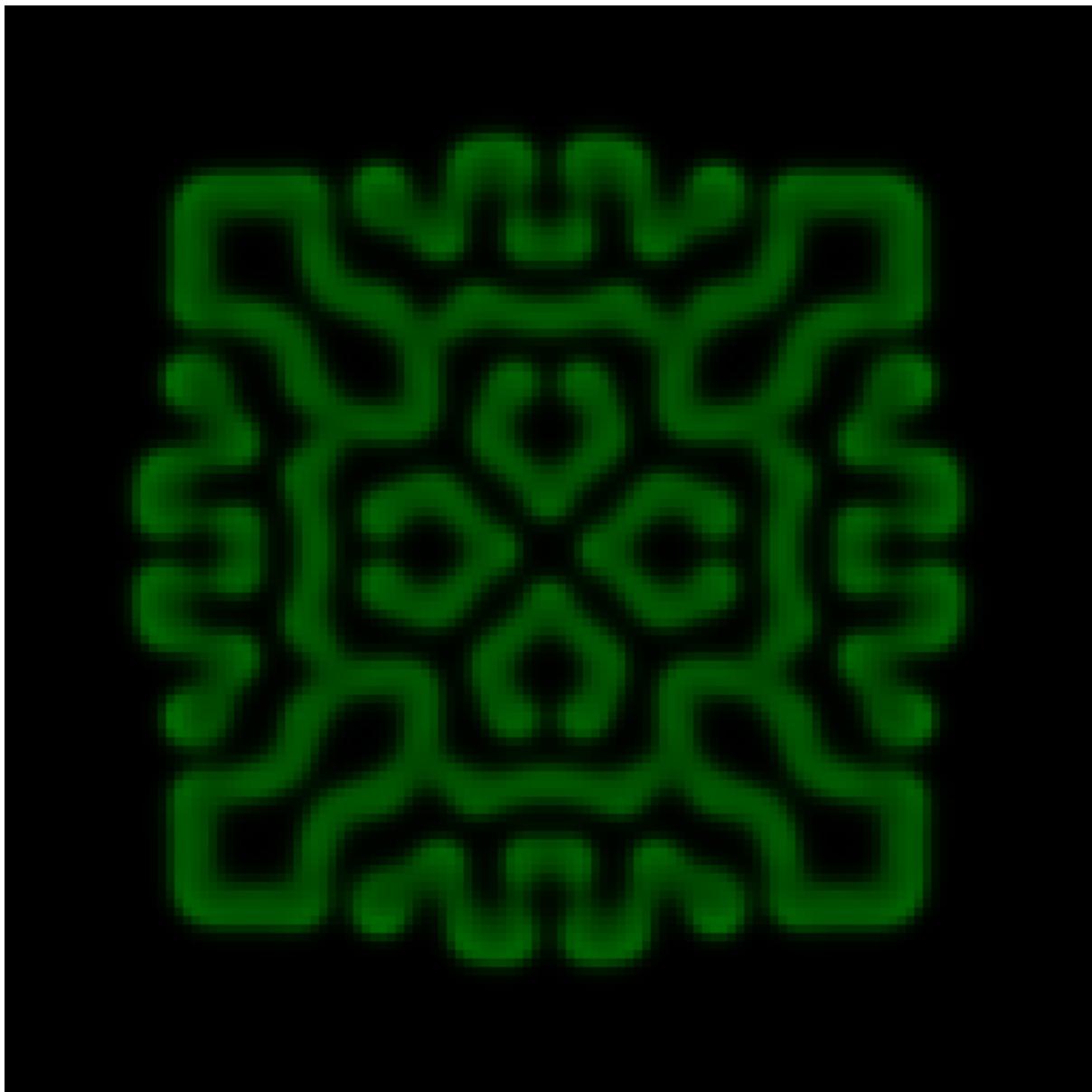
```
$ g++ -O3 gs_mpi.cpp -lmpi -lmpi_cxx
$ time mpirun -np 4 --oversubscribe ./a.out
conf000.dat
conf001.dat
conf002.dat
(snip)
conf098.dat
conf099.dat
mpirun -np 4 --oversubscribe ./a.out 2.39s user 0.29s system 321% cpu 0.832 total
```

321%とか出でるので、並列化できているようだ。実行時間も 1.697s → 0.832s と倍近く早くなっている。
実行結果も可視化して確認してみよう。



88





うん、大丈夫そうですね。

さて、いまは4コアあるローカルPCで4プロセス実行したから、理想的には4倍早くなって欲しいのに、2倍近くしか早くなっていない。つまり、並列化効率は50%程度である。

ん？並列化効率が物足りない？そういう時はウィースケーリングに逃げてサイズで殴れ！

というわけでサイズをでかくする。一辺4倍にして再度実行してみよう。

```
-const int L = 128;  
+const int L = 512;  
  
$ g++ -O3 gs.cpp  
$ time ./a.out  
(snip)  
./a.out 57.98s user 0.16s system 99% cpu 58.248 total  
  
$ g++ -O3 gs_mpi.cpp -lmpi -lmpi_cxx
```

```
$ time mpirun -np 4 --oversubscribe ./a.out
./a.out 57.98s user 0.16s system 99% cpu 58.248 total
mpirun -np 4 --oversubscribe ./a.out 68.28s user 1.72s system 382% cpu 18.305 total
```

実行時間が 58.248s → 18.305 となり、並列化効率も 80%近くに向上した。それでもなんか文句を言ってくる人がいたら、とてもローカル PC のメモリには乗りきらないほど大きな系を計算して黙らせよう。「並列化効率で悩んだらサイズに逃げろ」と覚えておくと良い。

余談：MPI の面倒くささ

本格的な領域分割コードの例として、二次元反応拡散方程式を並列化してみた。「並列化」によってどれくらいコードが増えたか見てみよう。

```
$ wc gs.cpp gs_mpi.cpp
   89      430     1969 gs.cpp
  272     1271    7345 gs_mpi.cpp
  361     1701    9314 total
```

というわけで、89 行から 272 行になった。3 倍増である。つまり、もともとの計算コードの二倍の量の通信コードがついたことになる。といっても、「通信コードそのもの」の量は大したことがない。

```
$ grep MPI_ gs_mpi.cpp
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &procs);
MPI_Dims_create(procs, 2, d2);
MPI_Gather(sendbuf.data(), lx * ly, MPI_DOUBLE, recvbuf.data(), lx * ly, MPI_DOUBLE, 0, MPI_COMM_WORLD, &st);
MPI_Status st;
MPI_Sendrecv(sendbuf.data(), ly, MPI_DOUBLE, right, 0,
             recvbuf.data(), ly, MPI_DOUBLE, left, 0, MPI_COMM_WORLD, &st);
MPI_Sendrecv(sendbuf.data(), ly, MPI_DOUBLE, left, 0,
             recvbuf.data(), ly, MPI_DOUBLE, right, 0, MPI_COMM_WORLD, &st);
MPI_Status st;
MPI_Sendrecv(sendbuf.data(), lx + 2, MPI_DOUBLE, up, 0,
             recvbuf.data(), lx + 2, MPI_DOUBLE, down, 0, MPI_COMM_WORLD, &st);
MPI_Sendrecv(sendbuf.data(), lx + 2, MPI_DOUBLE, down, 0,
             recvbuf.data(), lx + 2, MPI_DOUBLE, up, 0, MPI_COMM_WORLD, &st);
MPI_Init(&argc, &argv);
MPI_Finalize();

$ grep MPI_ gs_mpi.cpp | wc
   16      82     850
```

`MPI_Status st` の宣言を除くと 14 行だけである。それ以外はバッファの準備と整理に費やされている。これをもって「MPI は面倒くさい」というのであれば、私は同意する。しかし、「MPI の面倒くささ」の本質はそこではないように思う。

MPI を使って並列コードを書くことを「並列化 (parallelization)」と呼ぶ。「並列化」という言葉から想像されるのは、「もともとあるシリアル版のコードを改造して並列コードを書く」という作業であろう。典型的には、

1. シリアルコードを書く
2. 大きな系がやりたくなつたので、OpenMP を使ってスレッド並列をする
3. さらに MPI を使って並列版に修正する

といった開発プロセスとなりがちなのだとと思われる。しかし、既存のコードを修正して MPIを入れていく作業は極めて面倒くさく、バグが入りやすく、かつやっている最中に何をやってるかわからなくなりがちである。一度何をやってるかわからない状態になつたら、もうどこがバグなのか、バグが何に起因するのかわからず、泥沼にハマっていく。筆者は、学生さんだけでなくプログラムで飯を食っているプロな人でもそういう状態になっているのを何度も目撃している。

さて、スレッド並列はともかく、**MPI**を使った並列化とは、**MPI**向けに新規にコードを書き直す作業である。「正しい」並列化プロセスは以下の通りとなる。

1. シリアルコードを書く
2. MPI 並列化に必要な通信パターンを抽出する
3. その通信パターンだけを抜き出してテストコードを書く
4. シリアルコードとテストコードを参照しながら、新規にコードを開発する

具体的に4つ目のプロセスでは、「初期化して gather して保存し、正しいことを確認」「初期化後にのりしろ通信して、正しいことを確認」してから、次のステップに進んでいる。並列版として開発した `gs_mpi.cpp` は、シリアル版である `gs.cpp` をコピーせず、`gs.cpp` を参照しながらゼロから開発していった。MPI は面倒である。その感覚は正しい。しかし、順を追って開発していくは、別に難しくはない。ソースコードが三倍になった、というと「うっ」と思うかもしれないが、それでも 300 行も無いのだし、通信コードを書くこと自体は対して時間はかかるない。並列化に限ったことではないが、プログラムの開発時間のほとんどはデバッグでしめられている。面倒臭がらずに、通信ロジックのテストコードなどをきちんと書いていけば、さほど時間はかからずに並列化することができるだろう。

もし 2 万行のソースコードを渡されて「並列化しろ」と言われたら？ それはもうご愁傷様としか……

Day 6 : ハイブリッド並列

ハイブリッド並列とは

これまで、並列化の手段として MPI を使った「プロセス並列」を行ってきた。最初に述べたように、並列化には他にも「スレッド並列」という手段がある。プロセス並列が分散メモリ型、スレッド並列が共有メモリ型であり、スレッド並列だけではノードをまたぐことができないので、普通「スパコンを使う」というとプロセス並列が必須になる。さて、MPI を使ったプロセス並列「だけ」による並列化を「flat-MPI」と呼ぶ。一方、プロセス並列とスレッド並列を併用する並列化を「ハイブリッド並列」と呼ぶ。当然のことながら、ハイブリッド並列は、プロセス並列単体、スレッド並列単体よりも面倒になるので、できることならやりたくない。しかし、アプリケーションやサイズによっては、ハイブリッド並列を選択せざるを得ない場合もあるだろう。ここでは、スレッド並列を行うときの注意点や、ハイブリッド並列の実例について見てみよう。

仮想メモリと TLB

さて、プロセス並列ではあまり気にしなくてよかったが、スレッド並列を行う時には気にしなければいけないものとして「NUMA」というものがある。「NUMA」を気にするためには、仮想メモリについて知らないといけない。というわけで、仮想メモリについて見てみよう。

OS は実に様々なことをやっているが、特に重要な仕事に「メモリ管理」がある。物理的には「メモリ」はマザーボードに刺さった DRAM を指すが、OS の管理下で動くプロセスから見える「メモリ」は、それを仮想化したものである。プロセスにとっては連続に見えるメモリも、実は DRAM 上にバラバラに割り付けられているかもしれない。OS は、「プロセスから見えるアドレス」と「物理的に DRAM に割り当てられたアドレス」をうまいこと変換して、プロセスが物理メモリを意識しないで済むようにしている。このような仕組みを「仮想メモリ (virtual memory)」と呼ぶ。仮想メモリを扱う利点としては、

- OS がメモリを管理してくれるので複数のプロセスがお互いのメモリを気にしなくて良くなる(セキュリティ上も好ましい)
- 物理的には不連続であっても、プロセスから見ると連続アドレスに見えるようにできる
- メモリが足りない時にハードディスクなどにスワップすることで、物理メモリより大きな論理メモリ空間がとれる

などが挙げられる。なお、Windows では「ハードディスクにスワップする領域の上限」のことを「仮想メモリ」と呼んでいるようなので注意。

実際に、プロセスごとに固有の仮想メモリが与えられているのを見てみよう。こんなコードを書いてみる。

vmem.cpp

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

int rank;

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("rank = %d, address = %x\n", rank, &rank);
```

```
    MPI_Finalize();  
}
```

これは、int 型の変数 rank の値とアドレスを表示するコードである。関数の中に入れるとスタック内のアドレスになってしまうので（まあそれでもいいんだけど）、グローバル変数として宣言してある。これを Linux で実行するとこんな感じになる。

```
$ mpic++ vmem.cpp  
$ mpirun -np 4 ./a.out  
rank = 0, address = 611e64  
rank = 1, address = 611e64  
rank = 3, address = 611e64  
rank = 2, address = 611e64
```

すべて同じアドレスであるにもかかわらず、値が異なるのがわかるだろう。これは、4 プロセス立ち上がったそれぞれから「見える」アドレス空間が、物理的には異なるアドレスに割り当てられているからである。

なお、上記のコードを Mac で実行するとこうなる。

```
$ mpirun -np 4 --oversubscribe ./a.out  
rank = 1, address = cae26d8  
rank = 2, address = fe426d8  
rank = 3, address = ff4c6d8  
rank = 0, address = 40c36d8
```

論理的には同じ場所にあるはずの rank が、てんでバラバラのアドレスにあることがわかる。これは、Mac OSX が「アドレス空間ランダム化 (Address Space Layout Randomization, ASLR)」と呼ばれるセキュリティ技術を採用しているからである。ディストリビューションによっては Linux でもこれをやっているかもしれない。まあそれはさておき。

話の本筋からはやや外れるのだが、せっかく仮想メモリの話が出たので、合わせて TLB の話もしておこう。OS は、プロセスから見える「論理アドレス」と、実際に DRAM に割り当てる「物理アドレス」は、「ページ」と呼ばれる単位で管理している。メモリをある程度のまとまりごとにわけ、それを「ページ」と呼ぶ。論理アドレスと物理アドレスの対応は「ページテーブルエントリ (PTE)」と呼ばれ、それをまとめたデータを「ページテーブル」と呼ぶ。プロセスからメモリのアクセス要求があったら、ページテーブルを見てプロセスの論理アドレスから物理アドレスに変換してやる必要がある。ページサイズを調べるには getconf を使う。

```
$ getconf PAGESIZE  
4096
```

一般に、ページサイズは 4096 Byte に取ることが多い。すると、1GB のメモリは 26 万 2144 個の PTE で管理されることになる。これだけの数をキャッシュに乗せるのは不可能であるため、基本的にはページテーブルはメモリ上に置かれる。すると、メモリにアクセスする際にキャッシュミスをすると、まず論理アドレスを物理アドレスに変換するためにメモリにアクセスし、物理アドレスがわかったらそのアドレスにアクセス、とメモリアクセスが二回発生してしまう。これを防ぐために、一度アクセスした PTE を記憶する、特別なキャッシュが用意されている。これが TLB (Translation Lookaside Buffer) である。これもキャッシュの一種であるため、キャッシュのように L1、L2 のような階層構造を持っている。キャッシュの大きさはバイト数で指定されるが、TLB はエントリ数で指定される。TLB の情報を見るには x86info が便利だ。CentOS なら、

```
sudo yum install x86info
```

で入る。他のパッケージにもだいたい入っていると思う。キャッシュの情報を見るには-c を指定する。以下は Intel Xeon Gold 6130 (Skylake) 上で実行した結果である。

```
x86info v1.31pre
Found 32 identical CPUs
Extended Family: 0 Extended Model: 5 Family: 6 Model: 85 Stepping: 4
Type: 0 (Original OEM)
CPU Model (x86info's best guess): Core i7 (Skylake-X)
Processor name string (BIOS programmed): Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz
```

Cache info

```
L1 Data Cache: 32KB, 8-way associative, 64 byte line size
L1 Instruction Cache: 32KB, 8-way associative, 64 byte line size
L2 Unified Cache: 1024KB, 16-way associative, 64 byte line size
L3 Unified Cache: 22528KB, 11-way associative, 64 byte line size
```

TLB info

```
Instruction TLB: 2M/4M pages, fully associative, 8 entries
Instruction TLB: 4K pages, 8-way associative, 128 entries
Data TLB: 1GB pages, 4-way set associative, 4 entries
Data TLB: 4KB pages, 4-way associative, 64 entries
Shared L2 TLB: 4KB/2MB pages, 6-way associative, 1536 entries
64 byte prefetching.
```

Total processor threads: 32

This system has 2 eight-core processors with hyper-threading (2 threads per core) running at an esti

キャッシュの構造について知っている人なら、命令用 TLB とデータ用 TLB が分けられていたり、L1、L2 の区別があり、L2 は命令用とデータ用が共通になっているなど、キャッシュと同じような構造をしていることがわかるだろう。さて、一番大きな TLB でも、エントリ数は 1536 しかない。デフォルトでは 1 ページが 4KB なので、これをフルに活用しても、6MB ちょっとのメモリ空間しか管理できない。したがってそれより大きなデータを扱う場合には TLB ミスが起きることになる。しかし、先程の表示で Shared L2 TLB: 4KB/2MB pages と書いてあったように、この TLB は 2MB のページを管理することもできる。1 ページを 2MB にすれば、1536 エントリで 3GB のメモリ空間をカバーすることができる。このように、通常より大きなサイズのページを使うことをラージページ (Large Page)、もしくは ヒュージページ (Huge Page) と呼ぶ。ラージページのサイズは meminfo で確認できる。

```
$ cat /proc/meminfo | grep Hugepagesize
Hugepagesize: 2048 kB
```

確かに 2MB になっていることがわかる。ラージページを使うことで、TLB ミスの低減が期待できるが、メモリの最小管理サイズが大きくなるため、メモリの利用に無駄が多くなり、実質的にメモリ使用量が増える。また、ラージページはスワップアウトされないということも覚えておくと良い。仮想メモリでは、メモリが足りなくなるとページ単位でディスクにメモリを逃がすスワッピングを行うが、ラージページはスワップされない。もしラージページをスワップアウト (ディスクへの書き出し) した場合、あとで必要になってスワッピン (メモリへの読み戻し) する時に、ラージページのサイズだけ連続した物理メモリを用意してやる必要があるが、断片化などによりそれが保証できないためである。

Small ページのページウォーク

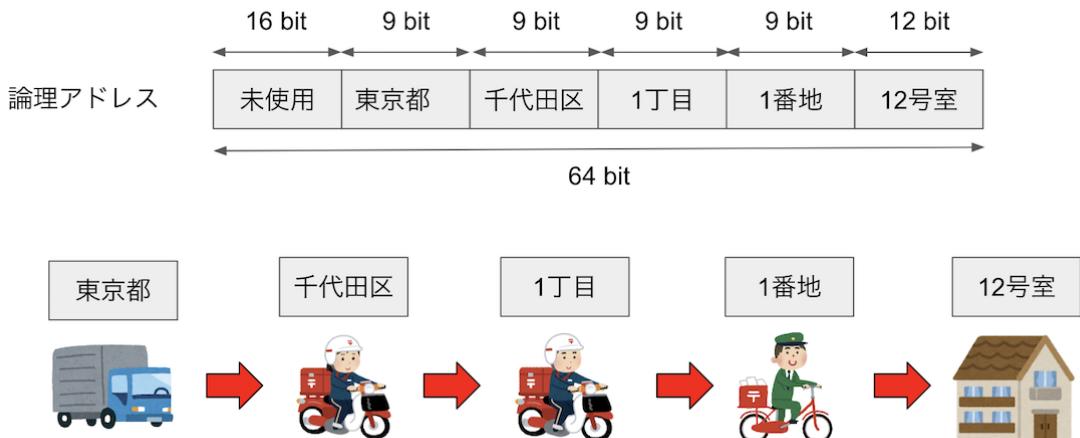


図 1: fig/smallpage.png

余談：TLB ミスについて

TLB もキャッシュの一種であるから、通常のキャッシュと同様にキャッシュミスを起こす。最近はメモリ空間が大きくなつたため、ページテーブルは多段階で管理されている。現在の一般的なマシンは 64 bit ので、理論的には 16 エクサバイトのメモリ空間を表現できる（正確には EiB、つまりエクスピバイトだがそれはさておく）。しかし、現実にはそんなに大きなメモリ空間は（まだ）実現できないため、48 bit で 256 テラバイトを表現する。現在の x86 は、この 48 ビットの論理アドレスを 5 つの部分にわける。最下位の 12 ビットが、ページ内のどの場所にあるかを表すオフセットである。12 ビット、すなわち 4096 バイトがページサイズとなる。その上位の 9 ビットを 4 セットが、ページの「住所」を表す。例えばそれぞれ「東京都」「千代田区」「一丁目」「一番地」といった具合である。イメージとしては、最初の 4 つで建物を指定し、最後のオフセットが部屋番号だと思えば良い。いま、「住所（論理アドレス）」が与えられた時、その住所が示す地球上での座標（物理アドレス）を探すことを考える。まず、都道府県の対応表（ページテーブル）を見て「東京」がどこにあるかを調べ、次に都内で「千代田区」がどこにあるかを調べ…と、4 段階の検索を行つて、ようやく論理アドレスと物理アドレスが対応付けられる。こうして、ページテーブルをたどつていつて物理アドレスを解決することをページウォークと呼ぶ。また、ラージページを扱う場合には、最後の 9 ビットもオフセットとして用いる。するとページサイズを表すビット数が $12+9=21$ ビットとなり、ページサイズが $2^{21}=2097152$ 、つまり 2MiB となることがわかる。住所の階層を一段階「粗く」して、その分最後の家を超高層ビルにしたようなイメージである。

いずれにせよ、ページウォークは時間がかかる作業であるため、一度調べたページテーブルエントリをキャッシュしたくなる。これが TLB であった。CPU はキャッシュと同様にハードウェアとして TLB を持つていて。一般に TLB ミスはページウォークを伴うためにコストが高い。しかし、TLB ミスが頻繁に起きるということは、TLB エントリ数に比べてずっと多い数のページをプログラムが触っている、ということである。これは使用メモリの局所性が低いことを意味するから、一緒にキャッシュミスも起きている可能性が高い。頻繁にキャッシュミスを起こしていれば性能は全くでないため、通常はそっちの方が問題になることが多い、それを解決すれば TLB ミスも一緒に減ることが多い（気がする）。しかし、現実にはキャッシュはあまり問題を起こしていないのに、TLB ミスが深刻な影響を与えることがたまにある。

かなりアドバンストな話題となるが、キャッシュは、ハッシュのような仕組みでデータを保存している。大雑把にいえば論理アドレスからハッシュ値を作り、そのハッシュ値によってキャッシュのどこにそのメモリ

Large (Huge) ページのページウォーク

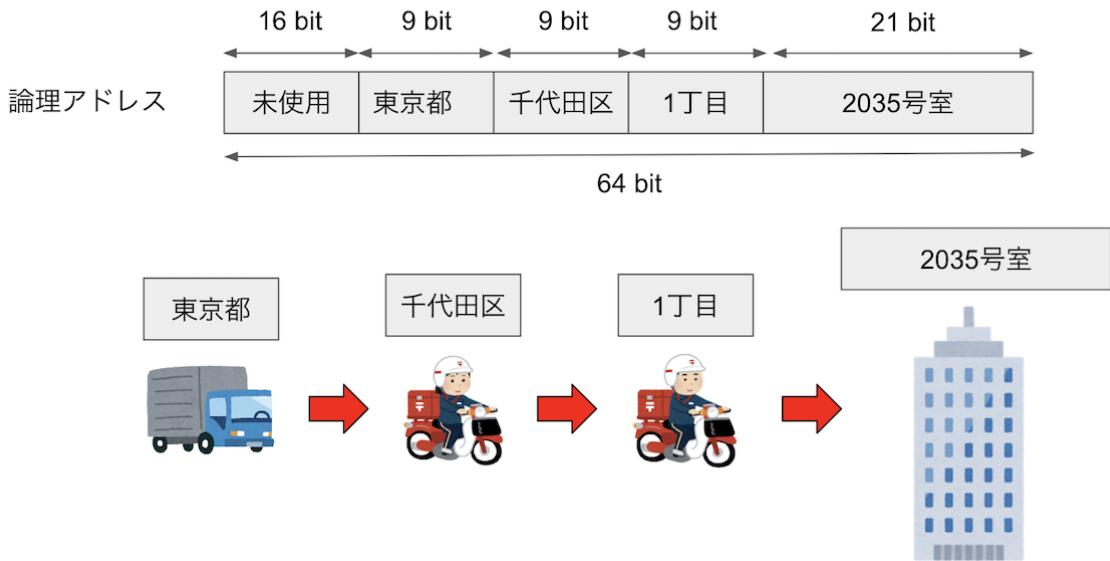


図 2: fig/largepage.png

を置くかを決める。したがって、メモリの別々の場所にあるにもかかわらず、ハッシュ値がぶつかるためにキャッシュ上の場所が同じになってしまうことがある。そして、運悪くそのデータを交互にアクセスするようなプログラムになっていると、キャッシュのサイズには余裕があるにもかかわらず、同じデータをキャッシュインしてはキャッシュアウトすることを繰り返し、性能が極端に劣化してしまう。これをキャッシュスラッシングと呼ぶ。多次元配列なんかを扱っていて、その次元を変えたとき、あるサイズだけ極端に性能が劣化したりした場合は、このキャッシュスラッシングが起きている可能性が高い。

さて、TLB もキャッシュであるから、全く同様な原理でスラッシングが起きる。これを TLB スラッシングと呼ぶ。個人的な経験だが、とあるサイトで自分のコードが flat-MPI では全く問題なく高並列で性能ができるのに、ハイブリッドにすると全く等価な計算をしているにもかかわらず高並列実行時に性能ブレが発生し、結果としてかなり性能が劣化してしまう、ということが起きた。その原因究明のために不要なルーチンを削って「犯人探し」をしていると、最終的に全く呼ばれていない関数を含むオブジェクトファイルをリンクすると性能が劣化、リンクしないと劣化しないという状況になった。繰り返すが、「使っていない関数を含むオブジェクトファイルをリンクするかどうか」で、性能が 20%近くもぶれてしまうのである。flat-MPI の場合にはこれは起きず、ハイブリッド並列の時だけ起きた。結局、これは TLB スラッシングが起きていたためとわかり、ページサイズを変えることで性能劣化が抑えられたのだが、今でも「なぜハイブリッドだと TLB スラッシングが起きたのか」はよくわかってない。あの GotoBLAS で有名な後藤さんも、いくつかの論文で行列積における TLB の影響が大きいことを指摘している。

- On Reducing TLB Misses in Matrix Multiplication (2002) by Kazushige Goto and Robert van de Geijn
- Anatomy of high-performance matrix multiplication

他にもハイブリッド実行時に性能ブレが発生し、結果として並列性能が劣化するのだが、その原因として TLB ミスが疑わしい場合が結構あるらしい。そのあたり、TLB の扱いをハードウェアでどう扱っているか、またマルチスレッドのとき、マルチプロセスのときにどうしているかなど、実装に強く依存しているっぽいので、「どうなってるかよくわからない」というのが筆者の正直な気持ちである。

なお、余談の余談となるが、論理アドレスは8バイトアラインされているため、アドレスを2進数表記すると下位3ビットは必ず0となる。glibcのmallocは、これを利用してメモリ(チャンク)の状態を記録する。詳細はkosakiさんによるmalloc動画、もしくはmallocの旅(glibc)編を参照してほしい。

NUMA

さて、計算ノードは「メモリ」と「CPU」でできているのだが、最近はCPUもマルチコアになったり、マルチソケットになったりして、ノード内はかなり複雑になっている。

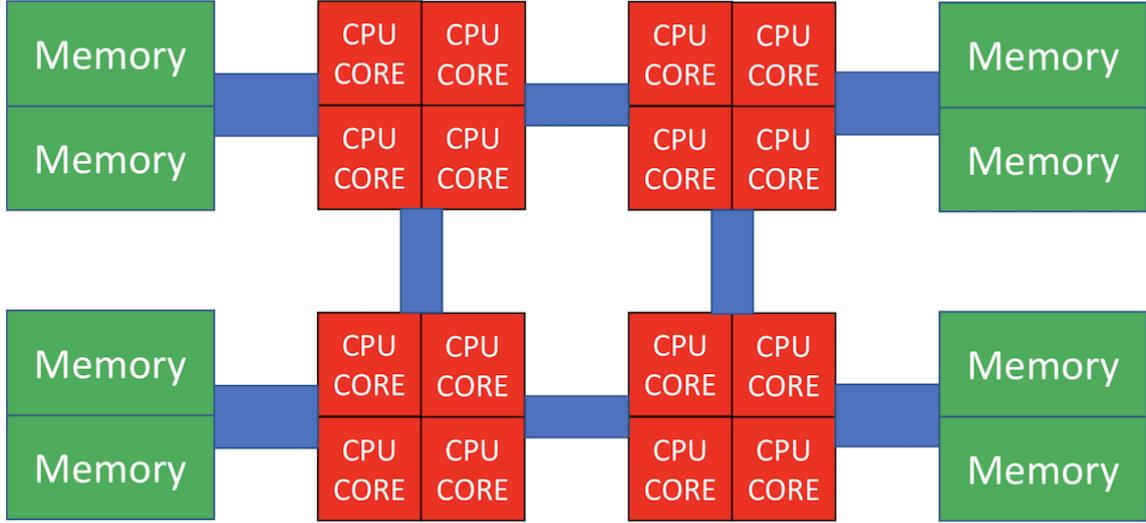


図 3: fig/numa.png

上図では、CPUが4つ搭載されており、それぞれにメモリがつながっている。CPU同士もバスで接続されており、例えば左下のCPUから右上のメモリにアクセスすることも可能であるが、自分の近くに接続されているメモリにアクセスするよりは時間がかかるてしまう。このように、CPUから見て「近いメモリ」「遠いメモリ」が存在する構成のことを「非対称メモリアクセス(Non-uniform Memory Access, NUMA)」と呼ぶ。「NUMA」をなんと呼ぶのが一般的なのかよく知らないのだが、筆者は「ぬ～ま」と呼んでいる。他にも「ぬま」や「にゅーま」と発音している人もいる。なぜNUMAが必要となるかはここでは深入りしないので、気になる人は各自調べて欲しい。

さて、論理メモリは、宣言しただけではまだ物理メモリは割り当てられない。例えば、以下のような配列宣言があったとする。

```
double a[4096];
```

倍精度実数は一つ8バイトであるため、ページサイズが4096バイトならば、この配列全体で8枚のページを割り当てる必要がある。しかし、宣言した瞬間には物理メモリは割り当てられない。物理メモリが割り当てられるのは、この配列に始めてアクセスした時である。はじめて配列に触ったとき、対応するページを物理メモリに割り当てるのだが、その物理メモリは「触ったスレッドが走っていたコアに一番近いメモリ」が選ばれる。これを「ファーストタッチの原則」と呼ぶ。一度物理メモリが割り当てられたら、開放されるまでずっとそのままである。したがって、そこから遠いコアで走るスレッドが触りにいったら時間がかかることになる。

flat-MPIをやっている場合は、各プロセスごとに独立な論理メモリを持っているため、原則として、あるプロセス(のメインスレッド)が触ったページを、他のスレッドが触りにくる可能性はない(ここではプロ

セスマイグレーションなどは考えていない)。しかし、スレッド並列をしている場合には「最初に触ったスレッドと、計算をするスレッドが異なる可能性」がある。

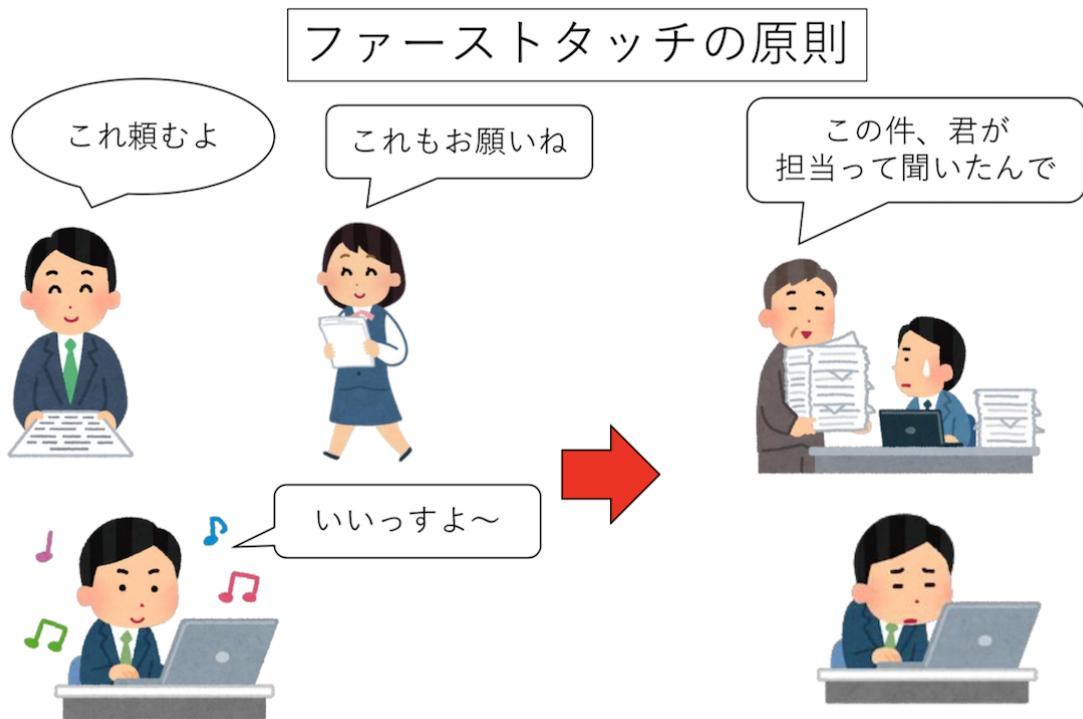


図 4: fig/firsttouch.png

これが問題となるのは「初期化処理は軽いから」と、大きな配列の初期化をメインスレッドでやって、全てのページがメインスレッドの近くに割り当てられてしまい、いざ重い処理をしようとしたらすべてのスレッドから「メインスレッドが走る CPU」にデータの要求が来て遅くなる、というパターンである。これを防ぐには、予め「あとで処理をするスレッド」が「始めてそのページを触るように」してやらないといけない。

OpenMP の例

さて、いよいよ OpenMP によるスレッド並列を行うが、その前にシリアルコードのプロファイルを取ろう。プロファイルとは、実行コードの性能分析のことで、最も簡単には関数ごとにどこがどれだけ時間を使っているか調べる。性能分析には `perf` を使うのが良い。残念ながら Mac には `perf` 相当のツールがなく、同様な目的に使われる `gprof` も正常に動作しないため、以下では `perf` が使える Linux を想定する。実行環境は以下の通り。

- Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz 12 コア x 2 ソケット

まず、シリアルコードとして Day 4 で使った Gray Scott モデルの計算を使おう。純粋に計算のみをカウントするため、途中のファイル出力を削除し、また実行時間を測定するようにしたのが `gs.cpp` である。ただし、デバッグのために最終結果だけファイルに出力している。コンパイルして `perf` でプロファイルをとつてみよう。まず、`perf record` で記録を取る。

```
$ g++ -O3 -fopenmp gs.cpp -o gs.out
```

```
$ perf record ./gs.out
2527 [ms]
conf000.dat
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.113 MB perf.data (~4953 samples) ]
```

実行時間が 2527 ms で、conf000.dat を出力したことがわかる。後のためにこれを conf000.org か何かにリネームしておこう。perf によって記録されたプロファイルデータは、perf.data として保存されている。これは perf report で中身を見ることができる。

perf report

環境によるが、こんな画面が出てくる。

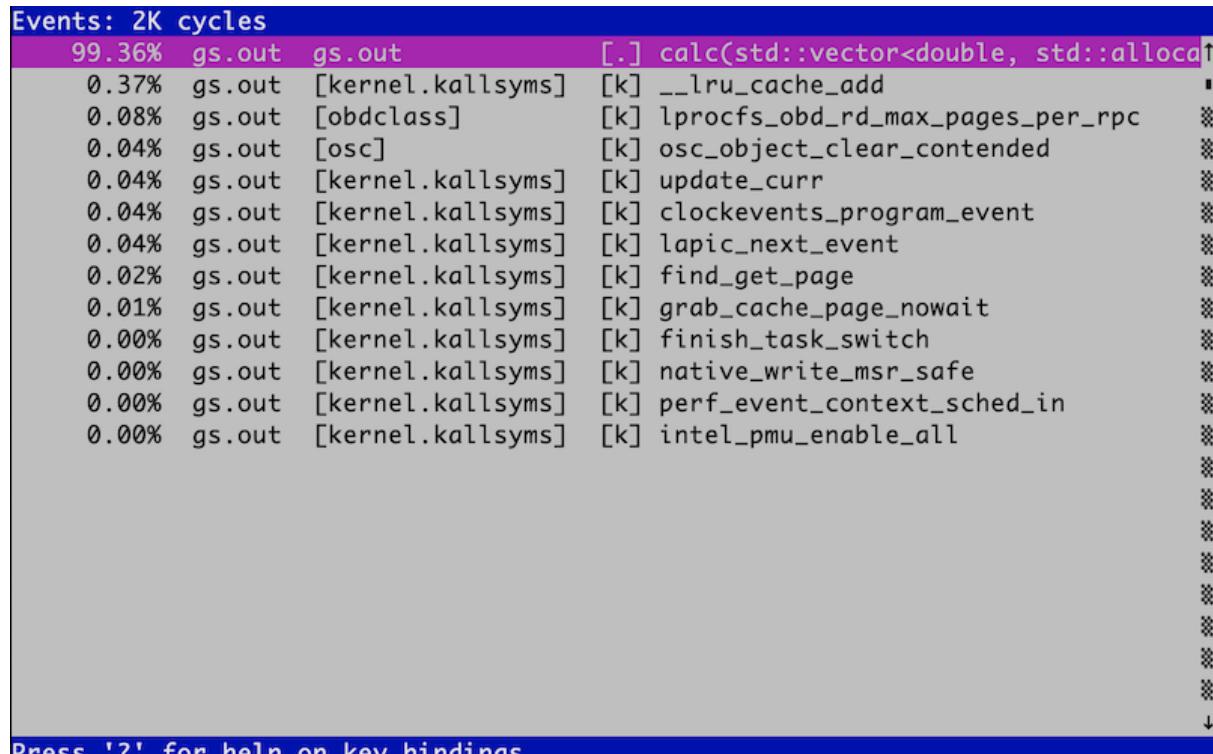


図 5: perf.png

いろいろ出てきているが、とりあえずメインの計算ルーチン calc が計算時間の 99.36%を占めるのを確認すれば良い。このように一番「重い」関数のことを ホットスポット (hotspot) と呼ぶ。ホットスポットが 90%以上を占めるような計算コードはチューニングがやりやすい。

さて、一番重い関数はこんな感じになっている。

```
void calc(vd &u, vd &v, vd &u2, vd &v2) {
// 外側ループ
    for (int iy = 1; iy < L - 1; iy++) {
        // 内側ループ
        for (int ix = 1; ix < L - 1; ix++) {
            double du = 0;
            double dv = 0;
            const int i = ix + iy * L;
```

```

        du = Du * laplacian(ix, iy, u);
        dv = Dv * laplacian(ix, iy, v);
        du += calcU(u[i], v[i]);
        dv += calcV(u[i], v[i]);
        u2[i] = u[i] + du * dt;
        v2[i] = v[i] + dv * dt;
    }
}
}

```

二重ループになっている。OpenMP は、並列実行したいループの直前にディレクティブを入れて、「このループを並列化してください」と指示することで並列化する。スレッド並列する時には、ループインデックス間に依存性がないか確認しなければならないのだが、今回はたまたまループインデックス間に全く依存関係がないので、好きなように並列化してよい(たまたまというか、そうなるように題材を選んだわけだが)。

まずは内側のループにディレクティブを入れてみよう。`#pragma omp parallel for` というディレクティブを対象ループの直前に入れるだけでよい。

`gs_omp1.cpp`

```

void calc(vd &u, vd &v, vd &u2, vd &v2) {
    for (int iy = 1; iy < L - 1; iy++) {
#pragma omp parallel for
        for (int ix = 1; ix < L - 1; ix++) {
            double du = 0;
            double dv = 0;
            const int i = ix + iy * L;
            du = Du * laplacian(ix, iy, u);
            dv = Dv * laplacian(ix, iy, v);
            du += calcU(u[i], v[i]);
            dv += calcV(u[i], v[i]);
            u2[i] = u[i] + du * dt;
            v2[i] = v[i] + dv * dt;
        }
    }
}

```

実行してみよう。スレッド数は環境変数 `OMP_NUM_THREADS` で指定する。12コア2ソケットマシンなので、全体で24コアあるから、24スレッドで走らせてみよう。ついでに `time` コマンドをかましてCPUがどれだけ使われているかも見てみる。

```

$ time OMP_NUM_THREADS=24 ./gs_omp1.out
24 threads 24078 [ms]
conf000.dat
OMP_NUM_THREADS=24 ./gs_omp1.out 573.12s user 1.72s system 2384% cpu 24.103 total

```

2384%、つまり24コア使われているのは間違いなさそうだが、シリアルコードで2527msだったのが、24078ms、つまり並列化により**10倍遅くな**ったことになる。ついでに結果が正しいことも確認しておこう(基本!)。

```
diff conf000.org conf000.dat
```

問題なさそうですね。

次に、外側を並列化してみよう。

```
gs_omp2.cpp
```

```
void calc(vd &u, vd &v, vd &u2, vd &v2) {
#pragma omp parallel for
for (int iy = 1; iy < L - 1; iy++) {
    for (int ix = 1; ix < L - 1; ix++) {
        double du = 0;
        double dv = 0;
        const int i = ix + iy * L;
        du = Du * laplacian(ix, iy, u);
        dv = Dv * laplacian(ix, iy, v);
        du += calcU(u[i], v[i]);
        dv += calcV(u[i], v[i]);
        u2[i] = u[i] + du * dt;
        v2[i] = v[i] + dv * dt;
    }
}
}
```

同じような計算をしてみよう。

```
$ time OMP_NUM_THREADS=24 ./gs_omp2.out
24 threads 411 [ms]
conf000.dat
OMP_NUM_THREADS=24 ./gs_omp2.out 9.16s user 0.02s system 2194% cpu 0.418 total

$ diff conf000.org conf000.dat
```

今度は早くなった。結果も正しいようだ。しかし、24コアを使っているのに、実行速度が6.4倍にしかなっていない。並列化効率にして27%程度である。ちなみに、12スレッド実行にしても、実行時間がたいして変わらない。

```
$ time OMP_NUM_THREADS=12 ./gs_omp2.out
12 threads 410 [ms]
conf000.dat
OMP_NUM_THREADS=12 ./gs_omp2.out 4.91s user 0.01s system 1185% cpu 0.415 total
```

並列数が半分になって、実行時間がたいしてかわらないので、並列化効率も51%にまで改善した。このように、

- 二重ループの内側と外側、どちらを並列化するかで性能が全く変わる。むしろ並列化により遅くなる場合もある。
- スレッドを増やせば増やすほど性能が良くなるわけではない。あるところからスレッドを増やすとむしろ性能が劣化する場合もある。

ということがわかる。

さて、なんで並列化して遅くなったのか見てみよう。まずは内側にディレクティブを入れた場合のコードを、1スレッド実行した場合のプロファイルである。見やすくするために、perf report の結果を cat にパイプしよう。perf report はデフォルトで結果を TUI インタフェースにより表示する (--tui) が、パイプをつないだ場合には標準出力に結果を流す (--stdio)。--sort は、ソートするキーを指定する。デフォルトでは関数ごとに表示されるが、dso を指定すると、シェアードライブラリごとにまとめてくれる。

```
$ OMP_NUM_THREADS=1 perf record ./gs_omp1.out
1 threads 3690 [ms]
conf000.dat
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.157 MB perf.data (~6859 samples) ]

$ perf report --sort dso | cat
(snip)
# Overhead      Shared Object
# ..... .....
#
68.91%  gs_omp1.out
22.04%  [kernel.kallsyms]
6.51%   libgomp.so.1.0.0
2.52%   libc-2.11.3.so
0.03%   [obdclass]
```

「Overhead」が、全体の時間に占める割合だが、自分のプログラムである gs_omp1.out が 68% しか占めていないことがわかる。libgomp.so.1.0.0 は OpenMP の実装である。同じことをスレッド並列していないコード gs.out でやるとこうなる。

```
$ perf record ./gs.out
2422 [ms]
conf000.dat
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.109 MB perf.data (~4758 samples) ]

$ perf report --sort dso | cat
# Overhead      Shared Object
# ..... .....
#
99.77%  gs.out
0.21%   [kernel.kallsyms]
0.02%   [obdclass]
```

つまり、ここで増えた kernel.kallsyms とか libgomp.so.1.0.0 が、スレッド並列化によるオーバーヘッドであることがわかる。実際、3690 ms の 68.91% は 2542ms であり、シリアルコードの実行時間とほぼ同じであることがわかる。

同様なことを外側にディレクティブを入れた場合でやってみると、変なオーバーヘッドがないことがわかる。

```
$ OMP_NUM_THREADS=1 perf record ./gs_omp2.out
2342 [ms]
conf000.dat
```

```
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.106 MB perf.data (~4615 samples) ]

$ perf report --sort dso | cat
(snip)
# Overhead      Shared Object
# ..... .
#
99.21%  gs_omp2.out
  0.39%  [kernel.kallsyms]
  0.30%  libgomp.so.1.0.0
  0.13%  libc-2.11.3.so
```

内側と外側、それぞれにディレクティブを入れた場合について、スレッド数を増やすして行った場合、計算にかかったコストと、それ以外のオーバーヘッドをグラフにした。

内側に入れた場合。

内側をスレッド並列化

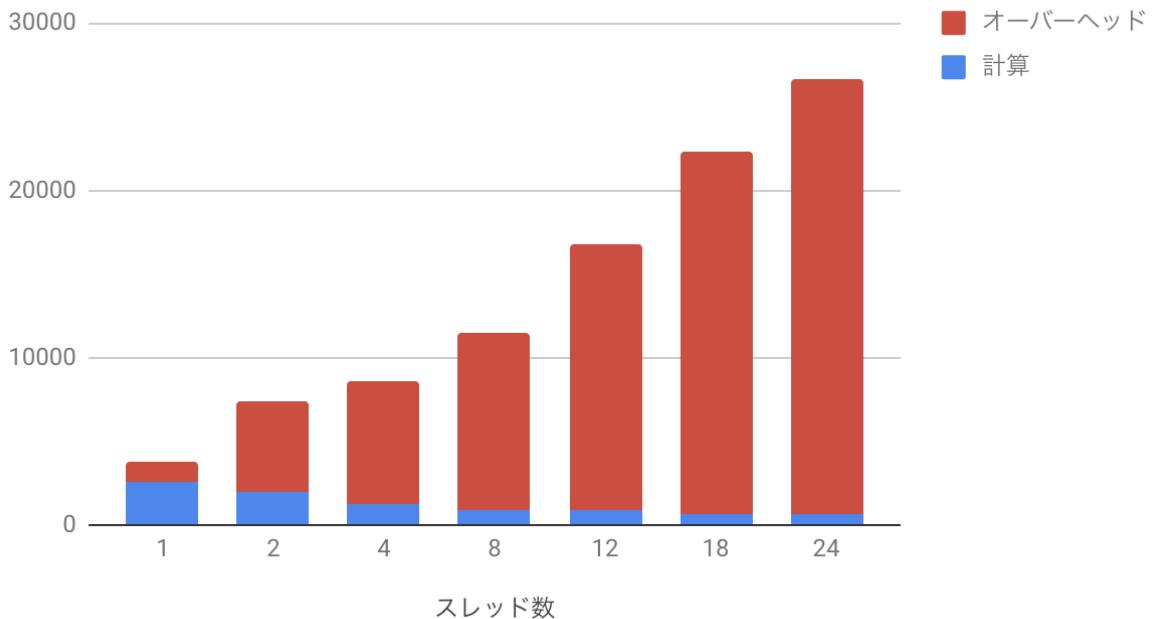


図 6: fig/inner.png

青が実際に計算しているところで、赤がオーバーヘッドである。計算時間は順調に減っているのだが、実行時間のほとんどをオーバーヘッドが占めるという、かなり悲惨なことになっている。

次に、外側にディレクティブを入れた場合。

先程と同様に青が実際に計算しているところで、赤がオーバーヘッドである。スレッド数を増やすほど計算時間は減るが、それに伴ってオーバーヘッドが増えてしまい、12スレッドのところで最も性能が良くなることがわかる。ちなみに先程は12スレッドと24スレッドはほぼ同じ時間だったが、perfをとった時には24スレッドの方が遅くなった。もともとこれだけ計算が軽い場合のスレッド並列は実行時間がかなり揺らぐので、このくらいは誤差の範囲である。

外側をスレッド並列化

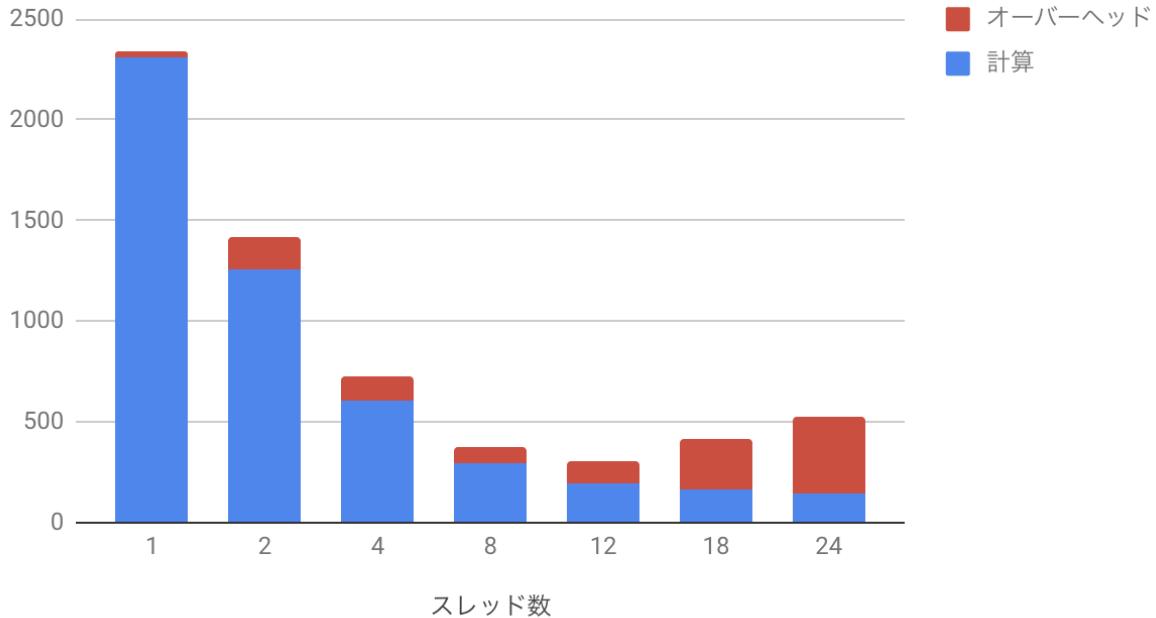


図 7: fig/outer.png

性能評価

さて、外側にディレクティブを入れて並列化した場合はそこそくうまいといったが、それでも 24 スレッド実行時の並列化効率は 27% と低い値であった。効率だけ見るとかなり改善の余地がありそうなので、いろいろチューニングをしてくる。しかし、チューニングを検討する前に、そもそもこの数字が良いのか悪いのか、悪いのなら理想に比べてどれだけ悪いのか考えて見よう。

そもそも、この計算は全部で 20000 ステップの計算をしているのであった。シリアルの計算が 2422 ms なので、1 ループあたり 120 ns くらいで回っていることになる。これを 24 スレッドで理想的に並列化したとすると、ループあたり 5 ns しかない。24 スレッドが同期するのに平均 5ns としても並列化効率 50% である。実際には、24 スレッド実行時に計算が平均 7ns、同期コストが 20ns 程度かかっている。12 スレッド実行の時には、9.7 ns、同期コストが 5.7ns である。計算したのは 12 コア 2 ソケットのマシンなので、CPU 内 12 コアの同期に 6ns 弱、2CPU 24 コアすべてが同期するのに 20ns というのは、まあ妥当な気がする。ということは、この計算はそもそも同期コストに比べて計算が軽すぎるのが問題なのであって、今後ループ結合やループ分割といった最適化を頑張ったとしても劇的に性能が向上したりする余地はなさそうだな、ということがわかるであろう。

ちなみに、内側を並列化した場合は、さらに外側のループの回転数だけ同期コストが増える。いま $L=128$ で、ループの回転数が $L-2$ なので、外側を並列化する場合に比べて同期回数が 126 倍かかる。これはかなりしんどそうだな、ということが想像できる。

ん？ 並列化効率がものたりない？だからそういう時はサイズを増やせ！

というわけで、サイズを一辺二倍にしてみよう。

```
$ ./gs.out  
10032 [ms]  
conf000.dat
```

```
$ OMP_NUM_THREADS=12 ./gs_omp2.out  
12 threads 1104 [ms]  
conf000.dat
```

```
$ OMP_NUM_THREADS=24 ./gs_omp2.out  
24 threads 1023 [ms]  
conf000.dat
```

二次元系なので、計算が単純に4倍重くなった。同期コストはほぼ変わらないため、相対的に並列化効率は良くなる。この場合は、12スレッドで並列化効率76%、24スレッドで41%である。サイズを増やせば増やすほど効率が高くなることが想像できよう。

一般論として、スレッド並列において同期コストが見えてしまった時、そのコストをなんとかしようと努力するのは、苦労のわりに実入りが少ない。なので、そういう時には計算するモデルを見直し、同期コストが見えないほど重い計算をすれば良い。繰り返しになるが並列化効率で困ったらウィースケーリングに逃げてサイズで殴れ！。せっかく「並列化」という武器を手に入れたのだから、その武器を手に入る前の敵とばかり戦うのではなく、その武器が最大限輝く敵を探す方が有益である。

本当はOpenMPについてもう少し真面目に、たとえばNUMA最適化やLoop Fusion/Fission/Collapseなども扱おうかと思ったのだが、正直面倒くさくなかった。個人的な感想で申し訳ないのだが、ディレクティブベースの最適化はコンパイラが何をやっているかが分かりづらく、隔靴搔痒の感があってどうしても好きになれない。

OpenMPには様々な構文があり、実際にいろいろなことができる。OpenMPについてはネットに良い文献が多数落ちているが、例えばIntelによる以下の文献がわかりやすかったので参照されたい。

インテルコンパイラ OpenMP 入門

また、名大の片桐先生による講義資料も詳しいので一読されたい。

計算科学技術特論第三回 OpenMP の基礎

余談：ロックの話

スレッド数が増えた時に、スレッドの同期コストが問題になると書いた。同期とは「待ち合わせ」のことである。修学旅行などで自由行動があったりするが、次の場所に移動する際には一度集合場所に集まって、全員がそろっているか確認しなければならない。この時、先に集合場所についた人は、後から来る人を待つことになる。この待ち時間が同期コストである。そう思うと、同期待ちの時にはCPUは遊んでいそうな気がするが、実際にほとんど同期待ちでCPUが「遊んでいる」はずのコードをtimeコマンドで実行してみると、

```
$ time OMP_NUM_THREADS=24 ./gs_omp1.out  
24 threads 24078 [ms]  
conf000.dat  
OMP_NUM_THREADS=24 ./gs_omp1.out 573.12s user 1.72s system 2384% cpu 24.103 total
```

と、24コアのCPUを2384%使っており、「ほぼ全てのCPUコアが忙しく動いていた」ことがわかる。これが何をやっているのか見てみよう。perf reportコマンドは、そのまま実行するとTUIモードになる。「内側」をスレッド並列化し、非常に非効率的になったコードをperf recordで記録してみよう。

```
OMP_NUM_THREADS=24 record ./gs_omp1.out
```

実行が終したら、perf report してみる。こんな画面になるだろう。

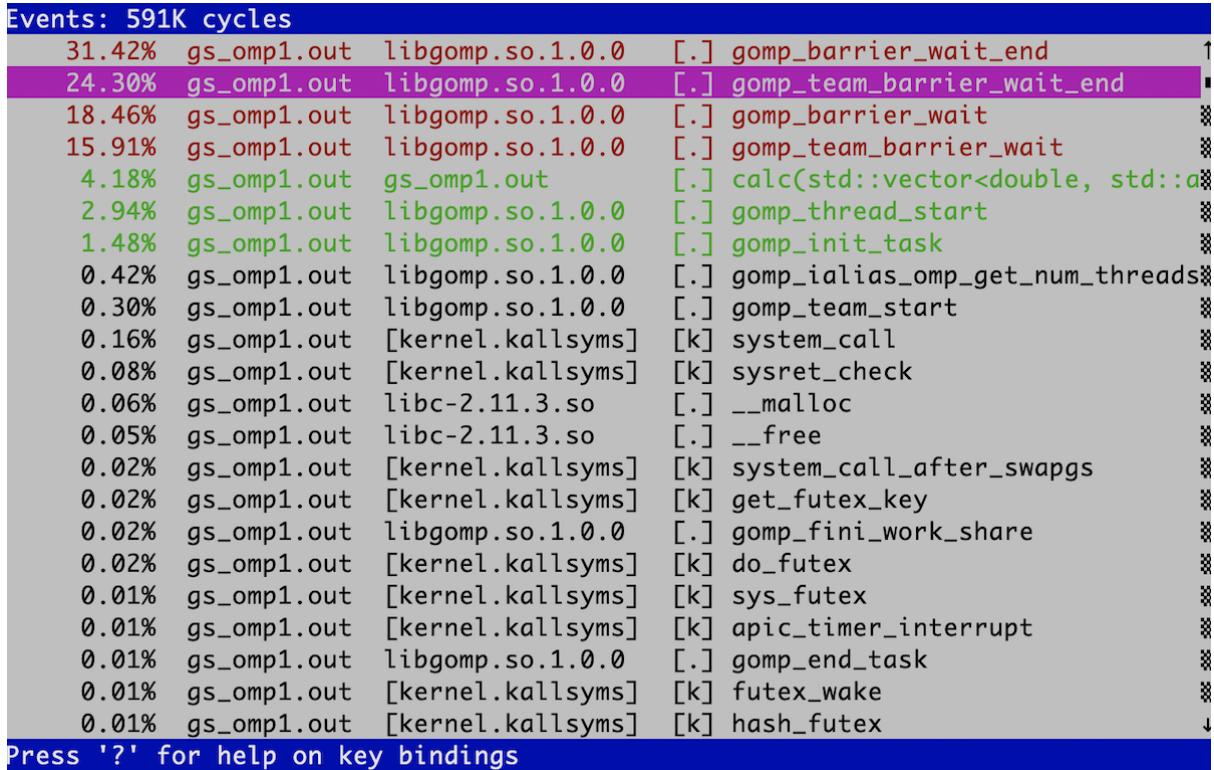


図 8: fig/perf.png

gomp_barrier_wait_end とか gomp_team_barrier_wait_end みたいな、いかにも同期待ち向けの関数が並んでいることがわかる。perf は、関数の内部を見ることができる。まずは gomp_barrier_wait を見てみよう。カーソルキーの上下で目的の関数を選び、エンターを押すと「Annotate gomp_barrier_wait」という項目があるので、それを選ぶ。こんな画面になるはずだ。

非常に単純なコードだが、気になる命令がある。lock だ。実は lock は命令ではなく、命令のプリフィックスである。「後に続く命令をアトミックにしてください」という修飾語のようなものだ。lock の後には、inc や xchg、そしてここで出てくる‘subl’など、いわゆる「read-modify-write」命令が入る。「read-modify-write」とは、「メモリから何かを読み出し、修正して、書き戻す」という一連の動作である。普通はこの動作はアトミックにならない。アトミックとは「不可分である、分けられない」という意味である。アトミックでない場合、この一連の動作の合間に他のスレッドの動作が割り込んでくる可能性がある。マルチスレッドでこういうことが起きるといろいろ困ることがある。よく言われるのが銀行の ATM の例だ。今、100 万円の貯金がある口座から、2 つの ATM からほぼ同時に 1 万円を引き出すことを考える。ATM がするべき動作は「銀行残高を読み出し (read)、1 万円を引いて (modify)、残額を書き戻す (99 万円)」の三つである。ところが、ATM1 と ATM2 で同時に「銀行残高を読み出し (read)」たとする。ATM1 も ATM2 も残高は 100 万円だと思っている。そしてそれぞれが「1 万円を引いて (modify)、残額を書き戻す (99 万円)」という残りの動作を行うと、合計 2 万円を引き出したにもかかわらず、残高が 99 万円になってしまう。こういったことを防ぐために、マルチスレッドをサポートする CPU は「read-modify-write」の一連の動作をアトミックに行う命令をハードウェアで持っている。この目的でよく使われるものが「compare-and-swap」、いわゆる CAS 命令であるが、ここでは深入りしないことにしよう。ちなみに、例えばピーターソンのアルゴリズムなど、ハードウェアサポートがなくても安全なロックを取る方法も知られているのだが、一般的に遅いので、マルチスレッドの実装にはハードウェアサポートがほぼ必須だと思われる。

さて、さっきの gomp_barrier_wait のアセンブリを見てみよう。libgomp.so を objdump すれば良い。

```

gomp_barrier_wait
:
:
:
セクション .text の逆アセンブル：

000000000000a430 <gomp_barrier_wait>:
    a430: 8b 47 04          mov    0x4(%rdi),%eax
19.16 : a433: 31 f6          xor    %esi,%esi
0.00 : a435: 83 e0 fc          and   $0xffffffffc,%eax
0.02 : a438: f0 83 6f 40 01      lock  subl $0x1,0x40(%rdi)
80.80 : a43d: 40 0f 94 c6      sete  %sil
0.00 : a441: 01 c6          add    %eax,%esi
0.00 : a443: e9 08 ff ff ff      jmpq  a350 <gomp_bar...

```

<-, -> or ESC: exit, TAB/shift+TAB: cycle hottest lines, H: Hottest

図 9: fig/perf2.png

シェアードライブラリの場所は LD_LIBRARY_PATH にあるだろうから探してみよう。

```

000000000000a430 <gomp_barrier_wait>:
    a430: 8b 47 04          mov    0x4(%rdi),%eax
    a433: 31 f6          xor    %esi,%esi
    a435: 83 e0 fc          and   $0xffffffffc,%eax
    a438: f0 83 6f 40 01      lock  subl $0x1,0x40(%rdi)
    a43d: 40 0f 94 c6      sete  %sil
    a441: 01 c6          add    %eax,%esi
    a443: e9 08 ff ff ff      jmpq  a350 <gomp_barrier_wait_end>
    a448: 0f 1f 84 00 00 00 00      nopl  0x0(%rax,%rax,1)
    a44f: 00

```

ロックを取って、あるメモリの指す値を 1 減らしたら、gomp_barrier_wait_end に飛んでいることがわかる。そこも見てみよう。perf で調べると、「重かった」のはこのループのようだ。

```

a390: 44 8b 07          mov    (%rdi),%r8d
a393: 41 39 d0          cmp    %edx,%r8d
a396: 75 20          jne    a3b8 <gomp_barrier_wait_end+0x68>
a398: 48 83 c1 01      add    $0x1,%rcx
a39c: 48 39 f1          cmp    %rsi,%rcx
a39f: f3 90          pause
a3a1: 75 ed          jne    a390 <gomp_barrier_wait_end+0x40>

```

見るとわかると思うが、あるメモリの指す場所を監視して、条件を満たしたらアドレス a3b8 に飛べ、そうでなければアドレス a390(抜き出した場所の先頭)に戻れ、というループである。つまり、スレッドは同期

待ちの間遊んでいるのではなく、条件が満たされるまでひたすらこのループを繰り返しているのである。例えるなら、赤信号で待っているあいだもずっとエンジンをぶかしている状態である。これが「同期待ちで暇そうなのに、なぜか CPU コアが忙しかった理由」である。ここでは「待ち合わせ」でぐるぐる回っていたが、ロックを獲得するのにぐるぐるループでまわる方式をスピンロックと言う。アトミックな「read-modify-write」命令があればスピンロックの実装は簡単である。詳細は先の Wikipedia の記事を参照されたい。ちなみに、京コンピュータで採用されている CPU は SPARC VIIIfx だが、この CPU にはハードウェアバリアが実装されている。どんな実装になっているか詳しくは知らないのだが、スピンロックなどのソフトウェア実装に比べて 10 倍以上高速とのことである。先程見たように、「計算が軽い」ループでは、同期コストが見えてきてしまう。ハードウェアバリアにより同期コストが減ると、その分だけ計算が軽くてもスケールしやすくなるため、プログラマとしてはありがたい。

ハイブリッド並列の実例

というわけで、ハイブリッド並列コードを作ってみよう。といっても、既に Day 5 で MPI 並列版を作っているので、その計算ルーチンに#pragma omp parallel for を一行入れるだけである。先程の知見から、二重ループの外側にディレクティブを入れよう。手元にある環境が、1 ノードに 2 ソケット、1 ソケットが 12 コアの環境で、18 ノードまでの計算がすぐに走るものだったので、系のサイズをそれにあわせて 324 くらいにしておく。

```
const int L = 9 * 36;
```

こうすると、12 や 18 の素因数である 2 や 3 をたくさん含んでいるので都合がよろしい。ついでに、時間を計測して、プロセス数、スレッド数、計算にかかった時間を表示するようにしておこう。実行スレッド数は omp_get_max_threads で取れる。

```
int num_threads = omp_get_max_threads();
```

時間計測はなんでも良いが、ここでは std::chrono::system_clock::now で計測しよう。

```
const auto s = std::chrono::system_clock::now();
// なにか時間を計測したい処理
const auto e = std::chrono::system_clock::now();
const auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(e - s).count();
```

これで elapsed にミリ秒単位の値が入る。このようにして作ったハイブリッド版の反応拡散方程式ソルバが gs_hybrid.cpp である。筆者の環境では MPI にパスが通してるので、以下のようなオプションでコンパイルできる。

```
g++ -fopenmp -O3 -mavx2 gs_hybrid.cpp -lmpi -lmpi_cxx
```

手元の Mac で 2 プロセス x 2 スレッドで実行すると以下の結果を吐く。

```
$ OMP_NUM_THREADS=2 mpexec -np 2 ./a.out
# System Size = 324
# 2 Process x 2 Threads
# Domain = 2 x 1
2 2 4635 [ms]
```

これを、手元のスペックでベンチマークを取ってみよう。石は Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz、12 コアが 2 ソケットで 1 ノードである。まずは 1 ノードで、1 プロセス 1 スレッドの計算から、スレッド数だけ 2,3,4,6... と増やした場合と、プロセス数だけ 2,3,4,6... と増やした場合の結果である。

シングルノード(12コア×2ソケット)でのスケーリング

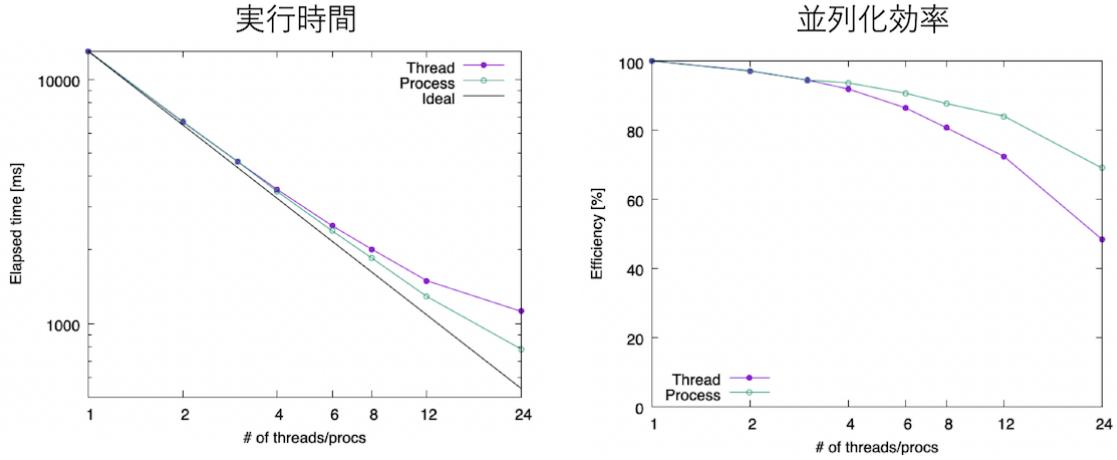


図 10: single_scaling.png

左が実行時間で両対数、右が並列化効率で片対数グラフになっている。プロセス数を増やした場合、つまり flat-MPI の方が実行効率が良いことがわかる。24 プロセス計算で、シリアル計算に比べて 16.6 倍、つまり並列化効率にして 70% 程度出ている。一方、全てスレッド並列で実行した場合も、さほど悪くはないが、24 スレッド計算で、シリアル計算に比べて 11.6 倍、並列化効率にして 48% と、50% を切ってしまう。MPI はバッファを作ったりコピーしたりという手間が発生する分、マルチスレッドより遅くなりそうだが、今回のケースでは flat-MPI の方が早かった。

今度はノードをまたいでみよう。18 ノードで計算してみる。各ノードには 2 ソケット × 12 コア、合計 24 コアあるから、全体で $18 \times 24 = 432$ 個の CPU コアがある。これを、プログラム全体を 432 スレッドのまま、プロセス数を変えてみよう。例えば、flat-MPI なら 1 プロセスに 1 スレッドなので、432 プロセスが最大プロセス数、1 ノードにフルにスレッドを立てると 24 スレッドなので、18 プロセス 24 スレッドの計算が最小プロセス数である。横軸にプロセス数、縦軸に実行時間をとった結果が以下の通り。

横軸は対数スケールになっている。全て 432 スレッドの計算で、違いは「1 プロセスが何スレッドを束ねているか」だけなのだが、ここでは 432 プロセス計算、すなわち 1 プロセスが 1 スレッドだけを持っている flat-MPI 計算が最も早くかった。

一般論として、プロセスとスレッドには最適な割合が存在し、それが一番早いかはやってみないとわからない。しかし、筆者の経験としては、非常に単純な計算で、かつそこそこ計算量がある場合は flat-MPI が一番早いことが多い。ただし、筆者はスレッド並列化にあまり慣れていないため、上記のコードも OpenMP に慣れた人がチューニングしたら、もっとハイブリッド版が早くなるのかもしれない。そのあたりはいろいろ試して見てほしい。「こうすればもっと早くなった」「ここが問題だった」といったプルリクを歓迎する。

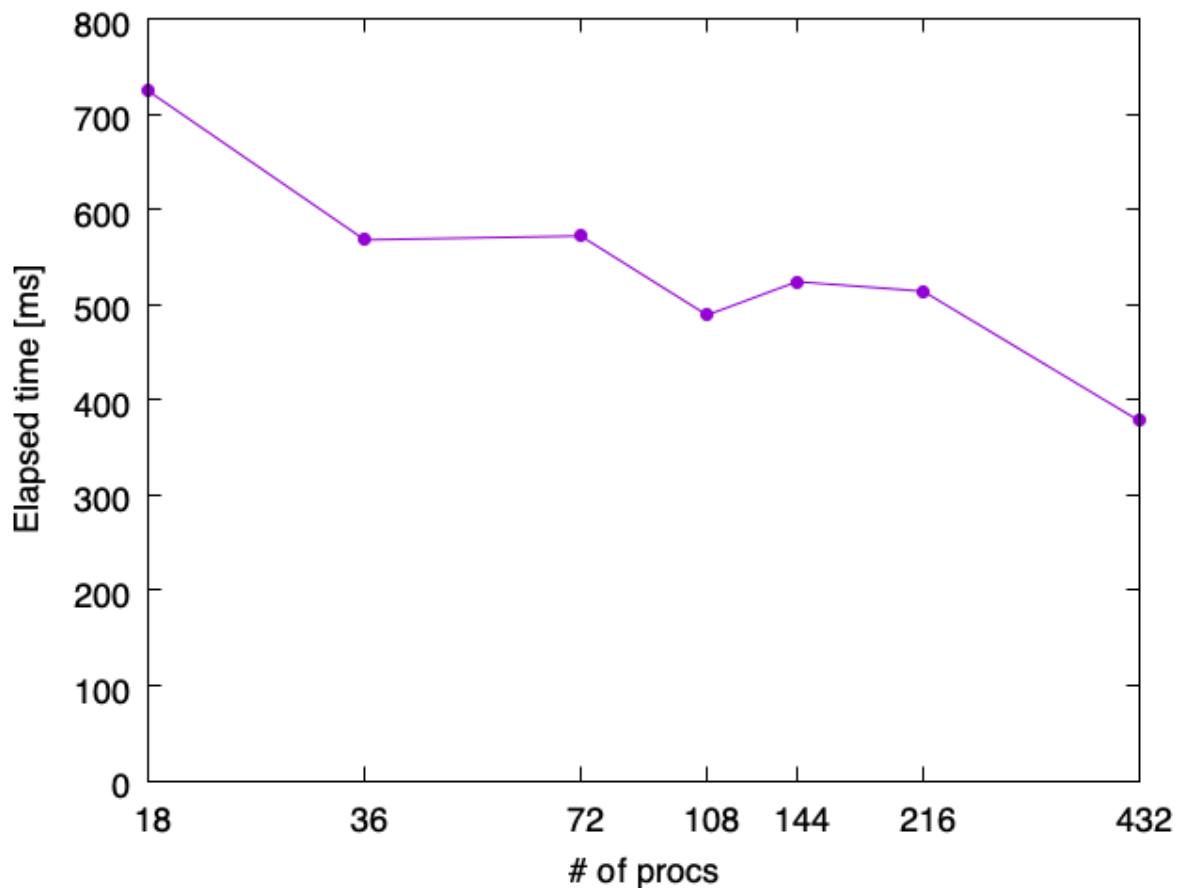


図 11: multi.png

Day 7 : SIMD 化

はじめに

ここまで読んだ人、お疲れ様です。ここから読んでいる人、それでも問題ありません。これまで、主に並列化についてだらだら書いてきたが、最後はシングルコアでの最適化技術である SIMD 化について説明してみたいと思う。

SIMD とは

スパコンプログラミングに興味があるような人なら、「SIMD」という言葉を聞いたことがあるだろう。 SIMD とは、「single instruction multiple data」の略で、「一回の命令で複数のデータを同時に扱う」という意味である。先に、並列化は大きく分けて「データ並列」「共有メモリ並列」「分散メモリ並列」の三種類になると書いたが、 SIMD はデータ並列 (Data parallelism) に属す。現在、一般的に数値計算に使われる CPU にはほとんど SIMD 命令が実装されている。後述するが、 SIMD とは 1 サイクルに複数の演算を同時に実行する技術であり、CPU の「理論ピーク性能」は、 SIMD の能力を使い切った場合の性能を指す。したがって、まったく SIMD 化できなければ、ピーク性能が数分の 1 になることと等価である。ここでは、なぜ SIMD が必要になるか、そして SIMD とは何かについて見てみよう。

計算機というのは、要するにメモリからデータと命令を取ってきて、演算器に投げ、結果をメモリに書き戻す機械である。CPU の動作単位は「サイクル」で表される。演算器に計算を投げてから、結果が返ってくるまでに数サイクルかかるが、現代の CPU ではパイプライン処理という手法によって事実上 1 サイクルに 1 個演算ができる。1 サイクル 1 演算できるので、あとは「1 秒あたりのサイクル数=動作周波数」を増やせば増やすほど性能が向上することになる。

というわけで CPU ベンダーで動作周波数を向上させる熾烈な競争が行われたのだが、2000 年代に入って動作周波数は上がらなくなった。これはリーク電流による発熱が主な原因なのだが、ここでは深く立ち入らない。1 サイクルに 1 演算できる状態で、動作周波数をもう上げられないのだから、性能を向上させるためには「1 サイクルに複数演算」をさせなければならない。この「1 サイクルに複数演算」の実現にはいくつかの方法が考えられた。

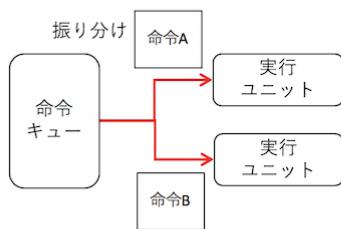
まず、単純に演算器の数を増やすという方法が考えられる。1 サイクルで命令を複数取ってきて、その中に独立に実行できるものがあれば、複数の演算器に同時に投げることで 1 サイクルあたりの演算数を増やそう、という方法論である。これをスーパースカラと呼ぶ。独立に実行できる命令がないと性能が上がらないため、よくアウトオブオーダー実行と組み合わされる。要するに命令をたくさん取ってきて命令キューにためておき、スケジューラがそこを見て独立に実行できる命令を選んでは演算器に投げる、ということをする。

この方法には「命令セットの変更を伴わない」という大きなメリットがある。ハードウェアが勝手に並列実行できる命令を探して同時に実行してくれるので、プログラマは何もしなくて良い。そういう意味において、スーパースカラとはハードウェアにがんばらせる方法論である。デメリット、というか問題点は、実行ユニットの数が増えると、依存関係チェックの手間が指数関数的に増えることである。一般的には整数演算が 4 つ程度、浮動小数点演算が 2 つくらいまでが限界だと思われる。

さて、スーパースカラの問題点は、命令の依存関係チェックが複雑であることだった。そこさえ解決できるなら、演算器をたくさん設置すればするほど性能が向上できると期待できる。そこで、事前に並列に実行できる命令を並べておいて、それをそのままノーチェックで演算器に流せばいいじゃないか、という方法論が考えられた。整数演算器や浮動小数点演算器、メモリのロードストアといった実行ユニットを並べておき、それらに供給する命令を予め全部ならべたものを「一つの命令」とする。並列実行できる実行ユニットの数だけ命令を「パック」したものを作り、一つの命令にするため、命令が極めて長くなる。そのため、こ

スーパースカラ

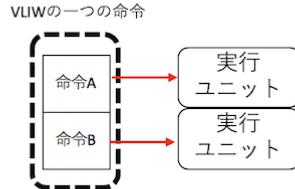
ハードウェアにがんばらせる



命令を複数取ってきて、スケジューラが振り分ける

VLIW

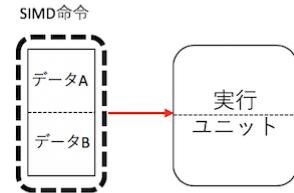
コンパイラにがんばらせる



事前に並列実行できる命令をひとつにまとめておく

SIMD

プログラマにがんばらせる



複数のデータに同じ演算を一度に行う



命令の後方互換性を保てる



依存関係チェックが不要
→ハードウェアが簡単に



ハードウェアは簡単
後方互換性も保てる



実行ユニットが増えると
命令振り分けで死ぬ



神のように賢いコンパイラが必要
後方互換性を失う



プログラムが大変

図 1: fig/simd.png

の方式は「Very Long Instruction Word (超長い命令語)」、略して VLIW と呼ばれる。実際にはコンパイラがソースコードを見て並列に実行できる命令を抽出し、なるべく並列に実行できるように並べて一つの命令を作る。そういう意味において、VLIW とはコンパイラにがんばらせる方法論である。

この方式で性能を稼ぐためには、VLIW の「命令」に有効な命令が並んでなければならない。しかし、命令に依存関係が多いと同時に使える実行ユニットが少なくなり、「遊ぶ」実行ユニットに対応する箇所には「NOP(no operation = 何もしない)」が並ぶことになる。VLIW は Intel と HP が共同開発した IA-64 というアーキテクチャに採用され、それを実装した Itanium2 は、ハイエンドサーバやスパコン向けにそれなりに採用された。個人的には Itanium2 は(レジスタもいっぱいあって)好きな方なのだが、この方が輝くためには「神のように賢いコンパイラ」が必須となる。一般に「神のように賢いコンパイラ」を要求する方法はだいたい失敗する運命にある。また、命令セットが実行ユニットの仕様と強く結びついており、後方互換性に欠けるのも痛い。VLIW は組み込み用途では人気があるものの、いまのところハイエンドな HPC 向けとしてはほぼ滅びたと言ってよいと思う。

さて、「ハードウェアにがんばらせる」方法には限界があり、「コンパイラにがんばらせる」方法には無理があった。残る方式はプログラマにがんばらせる方法論だけである。それが SIMD である。

異なる命令をまとめてパックするのは難しい。というわけで、命令ではなく、データをパックすることを考える。たとえば「 $C = A + B$ 」という足し算を考える。これは「 A と B というデータを取ってきて、加算し、 C としてメモリに書き戻す」という動作をする。ここで、「 $C_1 = A_1 + B_1$ 」「 $C_2 = A_2 + B_2$ 」という独立な演算があったとしよう。予め「 $A_1:A_2$ 」「 $B_1:B_2$ 」とデータを一つのレジスタにパックしておき、それぞれの和を取ると「 $C_1:C_2$ 」という、2つの演算結果がパックしたレジスタが得られる。このように複数のデータをパックするレジスタを SIMD レジスタと呼ぶ。例えば AVX2 なら 256 ビットの SIMD レジスタがあるため、64 ビットの倍精度実数を 4 つパックできる。そしてパックされた 4 つのデータ間に独立な演算を実行することができる。ここで、ハードウェアはパックされたデータの演算には依存関係が無いことを

仮定する。つまり依存関係の責任はプログラマ側にある。ハードウェアから見れば、レジスタや演算器の「ビット幅」が増えただけのように見えるため、ハードウェアはさほど複雑にならない。しかし、性能向上のためには、SIMD レジスタを活用したプログラミングを行わなければならない。SIMD レジスタを活用して性能を向上させることを俗に「SIMD 化 (SIMD-vectorization)」などと呼ぶ。原理的にはコンパイラによって SIMD 化することは可能であり、実際に、最近のコンパイラの SIMD 最適化能力の向上には目を見張るものがある。しかし、効果的な SIMD 化のためにはデータ構造の変更を伴うことが多く、コンパイラにはそういうグローバルな変更を伴う最適化が困難であることから、基本的には「SIMD 化はプログラマが手で行う必要がある」のが現状である。

SIMD レジスタを触ってみる

SIMD 化とは、CPU に実装されている SIMD レジスタをうまく使うコードを書いて実行速度を加速させることである。そのためには、まず SIMD レジスタを使う必要がある。SIMD レジスタとは、要するに複数のデータを一度に保持できる変数である。これを読んでいる人の大多数は AVX2 に対応した x86 系 CPU のパソコンを持っているだろう。まずは AVX2 の 256bit レジスタ、YMM レジスタを使ってみよう。以下、変数としては倍精度実数型を使う。倍精度実数は 64 ビットなので、256 ビットレジスタは倍精度実数を 4 つ保持できる。

SIMD を明示的に扱うには、まず `x86intrin.h` を include する。すると、SIMD レジスタに対応する `_m256d` という型が使えるようになる。これは 256bit の YMM レジスタを、倍精度実数 4 つだと思って使う型である。この型の変数に値を入れるには、例えば `_mm256_set_pd` という組み込み関数を使う。

```
__m256d v1 = _mm256_set_pd(3.0, 2.0, 1.0, 0.0);
```

この関数は、右から下位に値を放り込んでいく。上記の例なら、一番下位の 64 ビットに倍精度実数の 0、次の 64 ビットに 1 … と値が入る。さて、SIMD レジスタとは、変数を複数同時に保持できるものである。今回のケースでは、`_m256d` という型は、ほぼ `double [4]` だと思ってかまわない。実際、これらはそのままキャスト可能である。SIMD 化を行う時、こんなデバッグ用の関数を作ておくと便利である。

```
void print256d(__m256d x) {
    printf("%f %f %f %f\n", x[3], x[2], x[1], x[0]);
}
```

`_m256d` `x` が、そのまま `double x[4]` として使っているのがわかると思う。この時、`x[0]` が一番下位となる。先程の代入と合わせるとこんな感じになる。

print.cpp

```
#include <cstdio>
#include <x86intrin.h>

void print256d(__m256d x) {
    printf("%f %f %f %f\n", x[3], x[2], x[1], x[0]);
}

int main(void) {
    __m256d v1 = _mm256_set_pd(3.0, 2.0, 1.0, 0.0);
    print256d(v1);
}
```

これを `g++` でコンパイルするには、AVX2 を使うよ、と教えてあげる必要がある。

```
$ g++ -mavx2 print.cpp
$ ./a.out
3.000000 2.000000 1.000000 0.000000
```

ちゃんと SIMD レジスタに値が代入され、それが表示されたことがわかる。

さて、SIMD レジスタの強みは、SIMD レジスタ同士で四則演算をすると、4つ同時に計算が実行できることであった。それを確認してみよう。

単に`_m256d`同士の足し算をやる関数を書いてみる。`_m256d`の型は、そのまま四則演算ができる。`double[4]`の型をラップしたクラスを作り、オペレータのオーバーロードをしたようなイメージである。

```
#include <cstdio>
#include <x86intrin.h>

__m256d add(__m256d v1, __m256d v2) {
    return v1 + v2;
}
```

アセンブリを見てみる。少し最適化したほうがアセンブリが読みやすい。

```
g++ -mavx2 -O2 -S add.cpp
```

アセンブリはこうなる。

```
--Z3addDv4_dS_:
    vaddpd %ymm1, %ymm0, %ymm0
    ret
```

`vaddpd` は SIMD の足し算を行う命令であり、ちゃんと YMM レジスタの足し算が呼ばれていることがわかる。

実際に 4 要素同時に足し算できることを確認しよう。

add.cpp

```
#include <cstdio>
#include <x86intrin.h>

void print256d(__m256d x) {
    printf("%f %f %f %f\n", x[3], x[2], x[1], x[0]);
}

int main(void) {
    __m256d v1 = _mm256_set_pd(3.0, 2.0, 1.0, 0.0);
    __m256d v2 = _mm256_set_pd(7.0, 6.0, 5.0, 4.0);
    __m256d v3 = v1 + v2;
    print256d(v3);
}

$ g++ -mavx2 add.cpp
$ ./a.out
10.000000 8.000000 6.000000 4.000000
```

$(0,1,2,3)$ というベクトルと、 $(4,5,6,7)$ というベクトルの和をとり、 $(4,6,8,10)$ というベクトルが得られた。このように、ベクトル同士の演算に見えるので、SIMD 化のことをベクトル化と呼んだりする。ただし、線形代数で出てくるベクトルの積とは違い、SIMD の積は単に要素ごとの積になることに注意。実際、さっきの和を積にするところとなる。

mul.cpp

```
#include <cstdio>
#include <x86intrin.h>

void print256d(__m256d x) {
    printf("%f %f %f %f\n", x[3], x[2], x[1], x[0]);
}

int main(void) {
    __m256d v1 = _mm256_set_pd(3.0, 2.0, 1.0, 0.0);
    __m256d v2 = _mm256_set_pd(7.0, 6.0, 5.0, 4.0);
    __m256d v3 = v1 * v2; // 積にした
    print256d(v3);
}

$ g++ -mavx2 mul.cpp
$ ./a.out
21.000000 12.000000 5.000000 0.000000
```

それぞれ、 $0*0$ 、 $1*5$ 、 $2*6$ 、 $3*7$ が計算されていることがわかる。

あと SIMD 化で大事なのは、SIMD レジスタへのデータの読み書きである。先程はデバッグのために `_mm256_set_pd` を使ったが、これは極めて遅い。どんな動作をするか見てみよう。

setpd.cpp

```
#include <x86intrin.h>

__m256d setpd(double a, double b, double c, double d) {
    return _mm256_set_pd(d, c, b, a);
}
```

このアセンブリを見てみる。

```
g++ -mavx2 -O2 -S setpd.cpp

_Z5setpd:
vunpcklpd %xmm3, %xmm2, %xmm2
vunpcklpd %xmm1, %xmm0, %xmm0
vininsertf128 $0x1, %xmm2, %ymm0, %ymm0
ret
```

これは、

1. a と b を xmm2 レジスタにパック
2. c と d を xmm0 レジスタにパック
3. xmm2 レジスタの値を ymm0 レジスタの上位に挿入

ということをしている。ここで、`xmm0` レジスタと `ymm0` レジスタの下位 128 ビットは共有していることに注意。つまり、`xmm0` レジスタに読み書きすると、`ymm0` レジスタの下位 128 ビットも影響を受ける。上記の例はそれを利用して、最終的に欲しい情報、つまり 4 要素をパックしたレジスタを作っている。

とりあえず 4 つの要素を YMM レジスタに載せることができれば、あとは 4 要素同時に計算ができるようになるのだが、4 要素をパックする際に `_mm256_set_pd` を使うとメモリアクセスが多くなって性能が出ない。そのため、メモリから連続するデータをごそっとレジスタにとってきてたり、書き戻したりする命令がある。例えば、`_mm256_load_pd` は、指定されたポインタから連続する 4 つの倍精度実数をとってきて YMM レジスタに入ってくれる。ただし、そのポインタの指すアドレスは 32 バイトアラインされていなければならぬ。

利用例はこんな感じになる。

load.cpp

```
#include <cstdio>
#include <x86intrin.h>

void print256d(__m256d x) {
    printf("%f %f %f %f\n", x[3], x[2], x[1], x[0]);
}

__attribute__((aligned(32))) double a[] = {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0};

int main(void) {
    __m256d v1 = _mm256_load_pd(a);
    __m256d v2 = _mm256_load_pd(a + 4);
    __m256d v3 = v1 + v2;
    print256d(v3);
}
```

ここで `__attribute__((aligned(32)))` が、「後に続くデータを 32 バイトアライメントにしてください」という指示である。メモリアライメントについてはここでは深入りしないが、「利用メモリの端がちょうど良いところから始まっていること」とおぼえておけば良い。具体的には、配列の先頭アドレスが 32 で割り切れる値になる。メモリアライメントが合わないと遅くなったり、実行時に SIGSEGV で落ちたりする。

コンパイル、実行はこんな感じ。

```
$ g++ -mavx2 load.cpp
$ ./a.out
10.000000 8.000000 6.000000 4.000000
```

`_mm256_load_pd` が何をやっているか（どんなアセンブリに対応するか）も見てみよう。こんなコードのアセンブリを見てみる。

loadasm.cpp

```
#include <x86intrin.h>
__m256d load(double *a, int index) {
    return _mm256_load_pd(a + index);
}
```

```
g++ -O2 -mavx2 loadasm.cpp
```

例によって、少し最適化をかけておく。

```
__Z4loadPdi:  
    movslq %esi, %rsi  
    vmovapd (%rdi,%rsi,8), %ymm0  
    ret
```

ymm0 レジスタに、`vmovapd` 一発でデータがロードされていることがわかる。

ここでは読み出しを見たが、書き戻し (`_mm256_store_pd`) も同様である。

さて、これで SIMD 化の基礎は終わりである。ようするにデータを複数個とってきて、SIMD レジスタに乗せて、レジスタでなんか計算して、それを書き戻せばよろしい。簡単でしょう？ただし、バラバラのデータを pack/unpack すると遅いので、なるべく連続したデータをロードしたりストアしたりするように心がける必要がある。実際に使おうとすると、「あれ？if 文があるんだけど、どうすればいいんだろう？」とか「メモリの配置とレジスタに載せたい配置が微妙に違う」とかいろいろ出てくると思うが、それに対応して SIMD のための補助命令（シャッフルとか）が死ぬほどある。まあ、それは必要に応じて覚えておけばいいと思う。以下、簡単な例で SIMD 化の実際を見てみよう。

余談：アセンブリ言語？アセンブラ言語？

アセンブリの話になると、必ずといっていいほど「アセンブリ言語 (assembly language)」が正しく、アセンブライ語 (assembler language) は誤り」と言い出す人がいる。もちろんほとんどの場合において、アセンブリ言語 (assembly language)」をアセンブル (assemble) して機械語 (machine language) に変換するのがアセンブラー (assembler) である、という認識で良い。しかし、「アセンブライ語 (assembler language)」という用法が無いわけではない。もっとも有名なのが IBM である。IBM は昔から「アセンブライ語 (assembler language)」という呼び方をしてきた。IBM が現在サポートしているアセンブリ言語は IBM High Level Assembler(HLASM) であり、メインフレームである System z 上で動作する。HLASM のマニュアル内でも「アセンブライ語 (assembler language)」という呼び方をしている。しかし、System z の祖先である System 360 上で動作するアセンブリ言語は「IBM Basic Assembly Language (BAL)」と呼ばれており、「Assembly language」と「Assembler language」の使い分けは微妙である。しかし、IBM 自身は BAL のことを Basic Assembler Language と表現しているので、もしかしたら BAL のことを IBM は Basic Assembler Language、サードパーティは Basic Assembly Language と呼んでいるのかもしれない。

IBM 以外でも、ARM が規定したアセンブリ言語である UAL は「Unified Assembler Language」の略である。直訳すると「統合アセンブライ語」とでもなるだろうか。しかし、ARM 自身はアセンブリ言語のことを「assembly language」と呼んでおり、やはりその使い分けは微妙である。これは筆者の経験になるが、日本でも昔は「アセンブリ言語」のことを指して「アセンブラー」と呼ぶのがかなり一般的であった記憶がある。C 言語などを使わずに書くことを「フルアセンブラーで書く」といった具合である。いまでもそのような使い方をしている人を見かける。どうでもいいが、当時 x86 のアセンブラーを書いていた人は、機械語のことを「マシン語」と呼ぶのが一般的であった気もする。

繰り返しになるが、現在は「アセンブリ言語 (assembly language)」の方が一般的な用語であると思われる所以、「アセンブリ言語をアセンブラーがアセンブルして機械語にする」と表現することになんの問題もない。しかし、誰かが「アセンブラーを書く」もしくは「アセンブライ言語」と言ったときに、脊髄反射で「アセンブリ言語が正しい」とマウントを取る前に、上記のような事情を思い出していただけたらと思う。

余談の余談となるが、アセンブリで書かれたものを手で機械語に翻訳する作業を「ハンドアセンブル」と呼ぶ。昔のアセンブリはほぼ機械語と一対一対応しており、「便利なマクロ付き機械語」といった趣であつ

ため、ハンドアセンブルはさほど難しい作業ではなかった。しかし、現在の機械語、特に x86 の機械語はかなり複雑になっており、アセンブリから機械語に翻訳するのはかなり大変になっている。そのあたりは例えば x86_64 機械語入門なんかを参照してほしい。

簡単な SIMD 化の例

では、実際に SIMD 化をやってみよう。こんなコードを考える。一次元の配列の単純な和のループである。

func.cpp

```
const int N = 10000;
double a[N], b[N], c[N];

void func() {
    for (int i = 0; i < N; i++) {
        c[i] = a[i] + b[i];
    }
}
```

これを普通にコンパイルすると、こんなアセンブリになる。

```
g++ -O1 -S func.cpp

xorl %eax, %eax
leaq _a(%rip), %rcx
leaq _b(%rip), %rdx
leaq _c(%rip), %rsi
movsd (%rax,%rcx), %xmm0
addsd (%rax,%rdx), %xmm0
movsd %xmm0, (%rax,%rsi)
addq $8, %rax
cmpq $80000, %rax
jne LBB0_1
```

配列 a,b,c のアドレスを%rcx, %rdx, %rsi に取得し、`movsd` で a[i] のデータを%xmm0 に持ってきて、`addsd` で a[i]+b[i] を計算して%xmm0 に保存し、それを c[i] の指すアドレスに `movsd` で書き戻す、ということをやっている。これはスカラーコードなのだが、xmm は 128 ビット SIMD レジスタである。x86 は歴史的経緯からスカラーコードでも浮動小数点演算に SIMD レジスタである xmm を使う（後述の余談参照）。

さて、このループを AVX2 を使って SIMD 化することを考える。SIMD 化の基本は、ループをアンロールして独立な演算を複数作り、それを SIMD レジスタで同時に演算することである。AVX2 の SIMD レジスタは ymm で表記される。ymm レジスタは 256 ビットで、倍精度実数 (64 ビット) が 4 つ保持できるので、

- ループを 4 倍展開する
- 配列 a から 4 つデータを持ってきて ymm レジスタに乗せる
- 配列 b から 4 つデータを持ってきて ymm レジスタに乗せる
- 二つのレジスタを足す
- 結果のレジスタを配列 c のしかるべき場所に保存する

ということをすれば SIMD 化完了である。コードを見たほうが早いと思う。

func_simd.cpp

```

#include <x86intrin.h>
void func_simd() {
    for (int i = 0; i < N; i += 4) {
        __m256d va = _mm256_load_pd(&(a[i]));
        __m256d vb = _mm256_load_pd(&(b[i]));
        __m256d vc = va + vb;
        _mm256_store_pd(&(c[i]), vc);
    }
}

```

先程みたように、4つ連続したデータを持ってくる命令が`_mm256_load_pd`であり、SIMD レジスタの内容をメモリに保存する命令が`_mm256_store_pd`である。これをコンパイルしてアセンブリを見てみよう。

```

g++ -O1 -mavx2 -S func_simd.cpp

xorl %eax, %eax
leaq _a(%rip), %rcx
leaq _b(%rip), %rdx
leaq _c(%rip), %rsi
xorl %edi, %edi
LBB0_1:
    vmovupd (%rax,%rcx), %ymm0      # (a[i],a[i+1],a[i+2],a[i+3]) -> ymm0
    vaddpd (%rax,%rdx), %ymm0, %ymm0 # ymm0 + (b[i],b[i+1],b[i+2],b[i+3]) -> ymm0
    vmovupd %ymm0, (%rax,%rsi)      # ymm0 -> (c[i],c[i+1],c[i+2],c[i+3])
    addq $4, %rdi      # i += 4
    addq $32, %rax
    cmpq $10000, %rdi
    jb LBB0_1

```

ほとんどそのままなので、アセンブリ詳しくない人でも理解は難しくないと思う。配列のアドレスを`%rcx`, `%rdx`, `%rsi`に取得するところまでは同じ。元のコードでは`movsd`で`xmm`レジスタにデータをコピーしていたのが、`vmovupd`で`ymm`レジスタにデータをコピーしているのがわかる。どの組み込み関数がどんな SIMD 命令に対応しているかは Intel Intrinsics Guide が便利である。

念の為、このコードが正しく計算できるかチェックしよう。適当に乱数を生成して配列 `a[N]` と `b[N]` に保存し、ついでに答えも `ans[N]` に保存しておく。

```

int main() {
    std::mt19937 mt;
    std::uniform_real_distribution<double> ud(0.0, 1.0);
    for (int i = 0; i < N; i++) {
        a[i] = ud(mt);
        b[i] = ud(mt);
        ans[i] = a[i] + b[i];
    }
    check(func, "scalar");
    check(func_simd, "vector");
}

```

倍精度実数同士が等しいかチェックするのはいろいろと微妙なので、バイト単位で比較しよう。ここでは

配列 c[N] と ans[N] を unsigned char にキャストして比較している。

```
void check(void(*pfunc)(), const char *type) {
    pfunc();
    unsigned char *x = (unsigned char *)c;
    unsigned char *y = (unsigned char *)ans;
    bool valid = true;
    for (int i = 0; i < 8 * N; i++) {
        if (x[i] != y[i]) {
            valid = false;
            break;
        }
    }
    if (valid) {
        printf("%s is OK\n", type);
    } else {
        printf("%s is NG\n", type);
    }
}
```

全部まとめたコードはこちら。

simdcheck.cpp

実際に実行してテストしてみよう。

```
$ g++ -mavx2 -O3 simdcheck.cpp
$ ./a.out
scalar is OK
vector is OK
```

正しく計算できているようだ。

さて、これくらいのコードならコンパイラも SIMD 化してくれる。で、問題はコンパイラが SIMD 化したかどうかをどうやって判断するかである。一つの方法は、コンパイラの吐く最適化レポートを見ることだ。インテルコンパイラなら-qopt-report で最適化レポートを見ることができる。

```
$ icpc -march=core-avx2 -O2 -c -qopt-report -qopt-report-file=report.txt func.cpp
$ cat report.txt
Intel(R) Advisor can now assist with vectorization and show optimization
report messages with your source code.
(snip)
LOOP BEGIN at func.cpp(5,3)
    remark #15300: LOOP WAS VECTORIZED
LOOP END
```

実際にはもっとごちゃごちゃ出てくるのだが、とりあえず最後に「LOOP WAS VECTORIZED」とあり、SIMD 化できたことがわかる。しかし、どう SIMD 化したのかはさっぱりわからない。-qopt-report=5 として最適レポートのレベルを上げてみよう。

```
$ icpc -march=core-avx2 -O2 -c -qopt-report=5 -qopt-report-file=report5.txt func.cpp
$ cat report5.txt
```

```
Intel(R) Advisor can now assist with vectorization and show optimization
report messages with your source code.
```

(snip)

```
LOOP BEGIN at func.cpp(5,3)
    remark #15388: vectorization support: reference c has aligned access [ func.cpp(6,5) ]
    remark #15388: vectorization support: reference a has aligned access [ func.cpp(6,5) ]
    remark #15388: vectorization support: reference b has aligned access [ func.cpp(6,5) ]
    remark #15305: vectorization support: vector length 4
    remark #15399: vectorization support: unroll factor set to 4
    remark #15300: LOOP WAS VECTORIZED
    remark #15448: unmasked aligned unit stride loads: 2
    remark #15449: unmasked aligned unit stride stores: 1
    remark #15475: --- begin vector loop cost summary ---
    remark #15476: scalar loop cost: 6
    remark #15477: vector loop cost: 1.250
    remark #15478: estimated potential speedup: 4.800
    remark #15488: --- end vector loop cost summary ---
    remark #25015: Estimate of max trip count of loop=625
LOOP END
```

このレポートから以下のようなことがわかる。

- ymm レジスタを使った SIMD 化であり (vector length 4)
- ループを 4 倍展開しており (unroll factor set to 4)
- スカラーループのコストが 6 と予想され (scalar loop cost: 6)
- ベクトルループのコストが 1.250 と予想され (vector loop cost: 1.250)
- ベクトル化による速度向上率は 4.8 倍であると見積もられた (estimated potential speedup: 4.800)

でも、こういう時にはアセンブリ見ちやつた方が早い。

```
icpc -march=core-avx2 -O2 -S func.cpp
```

コンパイラが吐いたアセンブリを、少し手で並び替えたものがこちら。

```
xorl      %eax, %eax
..B1.2:
    lea      (,%rax,8), %rdx
    vmovupd a(%rax,8), %ymm0
    vmovupd 32+a(%rax,8), %ymm2
    vmovupd 64+a(%rax,8), %ymm4
    vmovupd 96+a(%rax,8), %ymm6
    vaddpd  b(%rax,8), %ymm0, %ymm1
    vaddpd  32+b(%rax,8), %ymm2, %ymm3
    vaddpd  64+b(%rax,8), %ymm4, %ymm5
    vaddpd  96+b(%rax,8), %ymm6, %ymm7
    vmovupd %ymm1, c(%rdx)
    vmovupd %ymm3, 32+c(%rdx)
    vmovupd %ymm5, 64+c(%rdx)
    vmovupd %ymm7, 96+c(%rdx)
    addq    $16, %rax
```

```
cmpq    $10000, %rax  
jb     ..B1.2
```

先程、手でループを 4 倍展開して SIMD 化したが、さらにそれを 4 倍展開していることがわかる。

これ、断言しても良いが、こういうコンパイラの最適化について調べる時、コンパイラの最適化レポートとにらめっこするよりアセンブリ読んじやった方が絶対に早い。「アセンブリを読む」というと身構える人が多いのだが、どうせ SIMD 化で使われる命令ってそんなにないし、最低でも「どんな種類のレジスタが使われているか」を見るだけでも「うまく SIMD 化できてるか」がわかる。ループアンロールとかそういうアルゴリズムがわからなくても、アセンブリ見て xmm だらけだったら SIMD 化は(あまり)されてないし、ymm レジスタが使われていれば AVX/AVX2 を使った SIMD 化で、zmm レジスタが使われていれば AVX-512 を使った SIMD 化で、`vmovupd` が出てればメモリからスムーズにデータがコピーされてそうだし、`vaddpd` とか出でればちゃんと SIMD で足し算しているなどとか、そのくらいわかれば実用的にはわりと十分だったりする。そうやっていつもアセンブリを読んでいると、そのうち「アセンブリ食わず嫌い」が治って、コンパイラが何を考えたかだんだんわかるようになる……かもしれない。

余談：x86 における浮動小数点演算の扱い

数値計算においては、浮動小数点演算が必須である。この浮動小数点数は、単精度なら 32 ビット、倍精度なら 64 ビットで表現される。その表現方法は IEEE 754 という仕様で決まっている。CPU は演算をレジスタで行うが、整数演算を行う汎用レジスタと、浮動小数点演算を行う浮動小数点レジスタは別々に用意されているのが一般的である。そして、倍精度実数演算をサポートする CPU は 64 ビットの浮動小数点レジスタを持っていることが多いのだが、x86 は例外で 64 ビットの浮動小数点レジスタがない。ではどうするかというと、普通の浮動小数点演算に 128 ビットの XMM という SIMD レジスタを使う。これには x86 における浮動小数点演算のサポートの歴史的な事情がある。

もともと、x86 はハードウェアレベルで浮動小数点演算ができなかった。したがって、浮動小数点演算をやろうとすると、汎用レジスタでソフトウェア的に浮動小数点演算をエミュレートしてやる必要があり、極めて遅かった。そこで、浮動小数点演算をサポートするために x87 というコプロセッサが用意された。コプロセッサとは、CPU の近くに外付けして機能拡張する装置で、コプロセッサに対応する命令が来ると、CPU からコプロセッサに処理が依頼される。ユーザからは CPU の命令セットが拡張されたように見える。この x87 コプロセッサは 80 ビットマシンであり、単精度(32 ビット)でも倍精度(64 ビット)でも、一度内部で 80 ビットに変換されてから計算され、演算結果をまた単精度なり倍精度なりに変換する。なお、`long double` 型は、80 ビットそのままで計算され、変換を伴わない。x86 系の石で `long double` が 80 ビットなのはこういう事情による。

x87 コプロセッサは、x86 の進化に合わせて一緒に進化していった。Intel 8086 には Intel 8087 が、80186 には 80187 が、80286 には 80287 が、といった具合である。なので x86 と同様に x87 と呼ばれる。x87 はしばらく外付けのコプロセッサだったが、80486 から x87 が CPU 内に内蔵されるようになった。

さて、浮動小数点演算にはもう一本の歴史がある。SIMD である。CPU の処理能力が上がると、音声や動画のエンコード、デコードや、3D 処理などの処理能力がもっとほしい、という要望が出てきた。そうした声に答える形で AMD が 3DNow! という SIMD 命令拡張を発表した。これは Intel による MMX を拡張し、64 ビットの MMX レジスタを使って単精度実数の演算を二つ同時にできるようにしたものだ。その後、Intel も SSE という SIMD 命令拡張を作り、ここで 128 ビットである XMM レジスタが導入された。当初、XMM レジスタは単精度実数の演算しかできなかったが、SSE2 で倍精度実数を扱えるようになる。AMD は倍精度実数の計算にデフォルトで XMM レジスタを使うようになり、その後 Intel もそうなったと思われる(そのあたりの前後関係はよく知らない)。

そんなわけで、x86 には浮動小数点レジスタとして、x87 で導入された 80 ビットのレジスタと、SSE で導入された 128 ビットの XMM レジスタがあるが、「普通の」64 ビットの浮動小数点レジスタはついに導入されなかった。現在の x86 では、SIMD ではない普段使いの倍精度実数演算でも、SIMD レジスタである XMM レジスタの下位 64 ビットを使って演算する。

その後、SIMD 命令拡張は AVX、AVX2、AVX-512 と発展を続け、SIMD 幅も 256 ビット、512 ビットと増えていった。それに伴って SIMD レジスタも YMM、ZMM と拡張されていく。YMM の下位 128 ビットが XMM、ZMM の下位 256 ビットが YMM になるのは、汎用レジスタ AX、EAX、RAX に包含関係があるのと同様である。

もう少し実戦的な SIMD 化

上記で触れた SIMD 化は非常に単純なもので、コンパイラが自動でできるものだった。しかし、一般には SIMD 化はレジスタ内をシャッフルしたり、データ構造を考えたりといろいろ複雑である。ここでは「もっとガチな SIMD 化をしたい！」というコアな SIMDer のために、もう少しだけ実践的な例を挙げてみよう。

磁場中の荷電粒子の運動を考える。磁場ベクトルを \vec{B} 、速度ベクトルを \vec{v} 、位置ベクトルを \vec{r} とする。簡単のため、素電荷も質量も光速も 1 とする単位系を取ろう。運動方程式はこんな感じになる。

$$\dot{\vec{v}} = \vec{v} \times \vec{B}$$

$$\dot{\vec{r}} = \vec{v}$$

ここで、速度の時間微分を要素をあらわに書くとこうなる。

$$\dot{v}_x = v_y B_z - v_z B_y$$

$$\dot{v}_y = v_z B_x - v_x B_z$$

$$\dot{v}_z = v_x B_y - v_y B_x$$

磁場中の荷電粒子の運動は、磁場と平行な向きには等速直線運動、垂直な向きには円運動をするため、結果として螺旋を描いてすすむ。こんな感じ。

さて、適当な方向に向いた磁場中に、ランダムな向きに初速を持った荷電粒子たちをばらまいた系を計算してみよう。なお、粒子同士の相互作用も無視する。三次元シミュレーションなので、三次元ベクトルを構造体で表現する。

```
struct vec {
    double x, y, z;
};
```

粒子数を N とし、位置ベクトル、速度ベクトルを構造体の配列にとる。

```
const int N = 100000;
vec r[N], v[N];
```

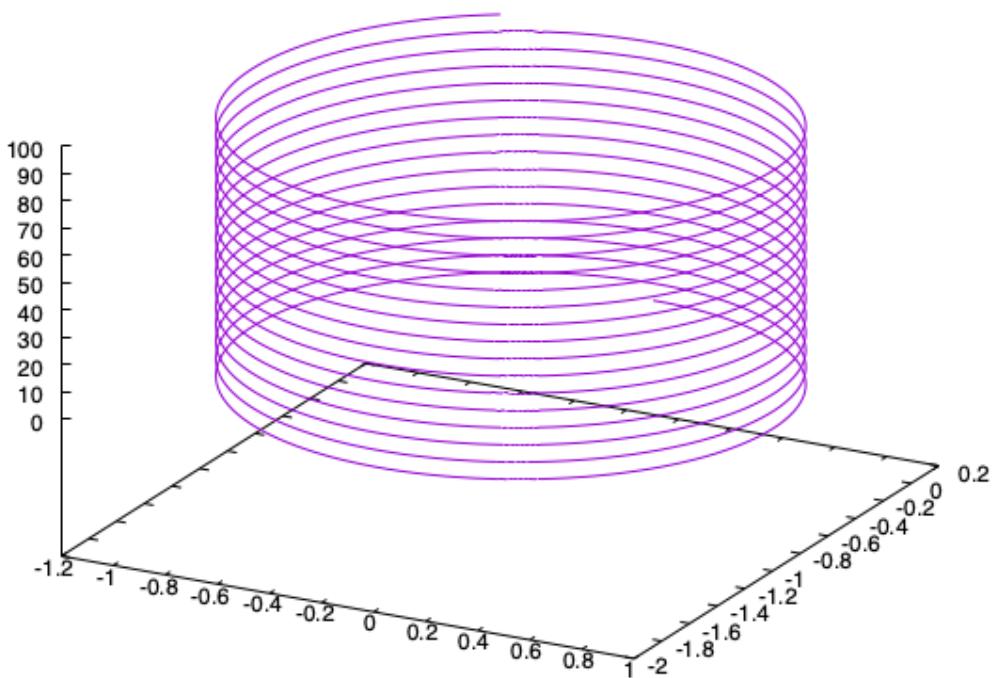


図 2: magnetic/one.png

この時、一次のオイラー法で時間発展を書くとこんな感じになる。

```
void calc_euler() {
    for (int i = 0; i < N; i++) {
        double px = v[i].y * BZ - v[i].z * BY;
        double py = v[i].z * BX - v[i].x * BZ;
        double pz = v[i].x * BY - v[i].y * BX;
        v[i].x += px * dt;
        v[i].y += py * dt;
        v[i].z += pz * dt;
        r[i].x = r[i].x + v[i].x * dt;
        r[i].y = r[i].y + v[i].y * dt;
        r[i].z = r[i].z + v[i].z * dt;
    }
}
```

しかし、よく知られているように一次のオイラー法は非常に精度が悪い。あなたが物理を学んだのなら、「磁場は荷電粒子に仕事をしない」ということを知っているはずである。つまり、全エネルギーは保存しなければならない。粒子間相互作用がないため、エネルギーは運動エネルギーだけである。

```
double energy(void) {
    double e = 0.0;
    for (int i = 0; i < N; i++) {
        e += v[i].x * v[i].x;
        e += v[i].y * v[i].y;
        e += v[i].z * v[i].z;
    }
    return e * 0.5 / static_cast<double>(N);
}
```

ちなみに、運動エネルギーなどは「一粒子あたり」の量にしておく(つまり平均エネルギーとする)と、粒子数の増減にエネルギーがよくなくなって便利である。時間発展はこんな感じにかけるだろう。

```
init();
double t = 0.0;
for (int i = 0; i < 10000; i++) {
    calc_euler();
    t += dt;
    if ((i % 1000) == 0) {
        std::cout << t << " " << energy() << std::endl;
    }
}
```

平均エネルギーの時間発展はこうなる。

1st-Euler とあるのが1次のオイラー法である。保存するべきエネルギーがどんどん増えてしまっていることがわかる。

さて、精度を上げる数値積分法はいくらでもあるが、ここでは簡単に二次の Runge-Kutta(RK) 法を採用しよう。

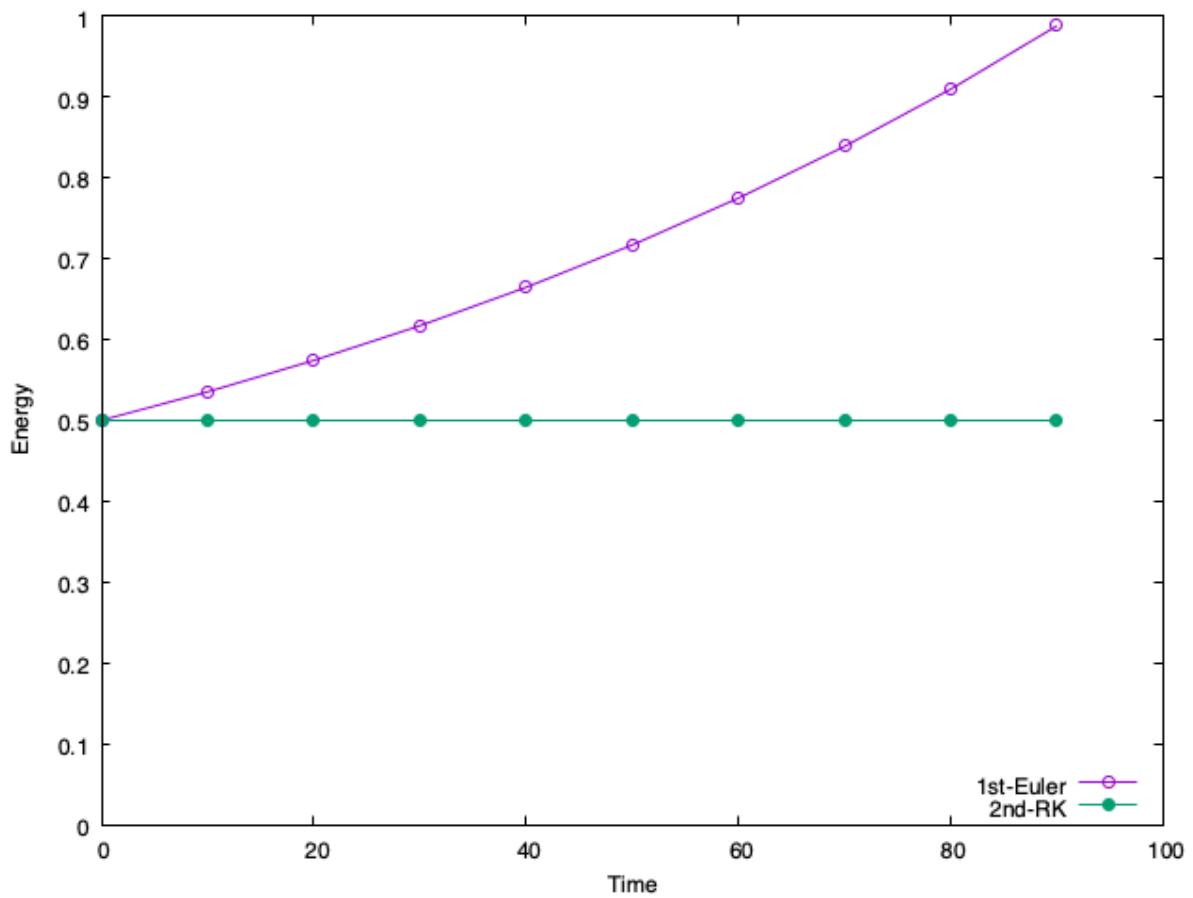


図 3: magnetic/energy.png

```

void calc_rk2() {
    for (int i = 0; i < N; i++) {
        double px = v[i].y * BZ - v[i].z * BY;
        double py = v[i].z * BX - v[i].x * BZ;
        double pz = v[i].x * BY - v[i].y * BX;
        double vcx = v[i].x + px * dt * 0.5;
        double vcy = v[i].y + py * dt * 0.5;
        double vcz = v[i].z + pz * dt * 0.5;
        double px2 = vcy * BZ - vcz * BY;
        double py2 = vcz * BX - vcx * BZ;
        double pz2 = vcx * BY - vcy * BX;
        v[i].x += px2 * dt;
        v[i].y += py2 * dt;
        v[i].z += pz2 * dt;
        r[i].x += v[i].x * dt;
        r[i].y += v[i].y * dt;
        r[i].z += v[i].z * dt;
    }
}

```

二次のRKは、まず1次のオイラー法で時間刻みの半分だけ系を仮想的に時間発展させ、その場所において再度時間微分を計算し、その微分係数をもとに現在時刻から時間発展させる方法である。一般にRunge-Kuttaというと4次の方法を指すが、ここでは手抜きして2次にする。また、座標の更新はどうせ同じなので1次のオイラーのままにする。

こうして計算したエネルギーが先程の図の「2nd-RK」と記されたデータ点である。エネルギーがきちんと保存していることがわかるだろう。この関数 `calc_rk2` を SIMD 化して見ることにしよう。

先に、SIMD化では、連続したデータを取ってくるのが重要であると書いた。いまはYMMレジスタを使うので、4つの要素を取ってきたい。しかし、三次元シミュレーションなので、各粒子は3要素の速度と3要素の位置を持っている。まずこれをなんとかしよう。

具体的には、3要素のベクトルを4要素にしてしまう。

```

struct vec {
    double x, y, z, w; // wを増やした
};

```

こうすると、一命令で粒子の速度がごそっとレジスタに乗る。

上図の例では、ポインタ`&v[i].x`が、`i`番目の粒子の速度ベクトルの先頭位置を示すため、

```
// vv <- (w, z, y, x)
__m256d vv = _mm256_load_pd((double *)(&(v[i].x)));
```

とすると、レジスタに3つのデータを一度に持ってくることができる（一要素は無駄になる）。

次に、レジスタに載せたデータから微分を計算したいのだが、そのままではベクトル積の形にならない。具体的にやりたいのはこんな計算であった。

```

double px = v[i].y * BZ - v[i].z * BY;
double py = v[i].z * BX - v[i].x * BZ;
```

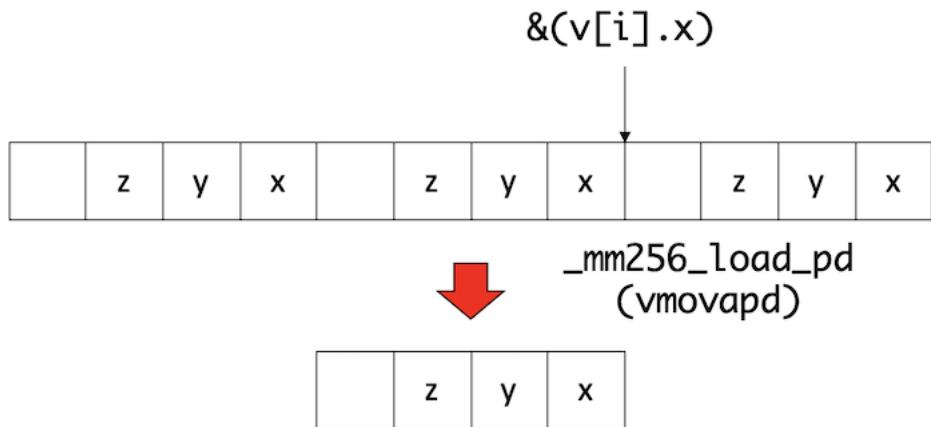


図 4: fig/load_pd.png

```
double pz = v[i].x * BY - v[i].y * BX;
```

この計算をデータをレジスタに載せたままで実行するためには、レジスタ内で (x,y,z,w) と並んでいるデータを (y,z,x,w) という順番に並び替えなければならない。このようなレジスタ内の要素の並び替えをするためにシャッフル命令が用意されている。シャッフルのやり方については、例えば AVX の倍精度実数シャッフル系命令チートシートを参照してほしいが、シャッフル後の要素の並び方を四進数で表現してやる。たとえば (0,1,2,3) とあるレジスタを (1,2,0,3) の順番にしたければ、

```
const int im_yzx = 64 * 3 + 16 * 0 + 4 * 2 + 1 * 1;
```

という数字を引数に与えてやれば良い。上記の数を四進数表記すると 3021 になることに注意。これを

```
// (w,x,z,y) <- (w,z,y,x)
__m256d vv_yzx = _mm256_permute4x64_pd(vv, im_yzx);
```

注：このあたり、左右どちらを下位に取るかいつも混乱する。レジスタは右に下位ビットを取るのが慣例であるため、その意味では (w,z,y,x) という順番で並んでいる。しかし、普通の数学の意味でのベクトルは最初の要素を左に書くのが慣例なので (x,y,z,w) と書きたくなる。そのあたり混乱しがちだが、適当に補って読んでほしい。

これによりレジスタの中身を適宜並び替えて、足したりかけたり引いたりすればよろしい。磁場については、使うベクトルが (z,x,y,0) と (z,x,y,0) のパターンしかないので、最初に宣言しておこう。

```
__m256d vb_zxy = _mm256_set_pd(0.0, BY, BX, BZ);
__m256d vb_yzx = _mm256_set_pd(0.0, BX, BZ, BY);
```

先程の図の中央に縦に並んだベクトルの計算は、実際のコードでは

```
__m256d vp = vv_yzx * vb_zxy - vv_zxy * vb_yzx;
```

と一行で書ける (SIMD がベクトル演算と呼ばれる所以である)。

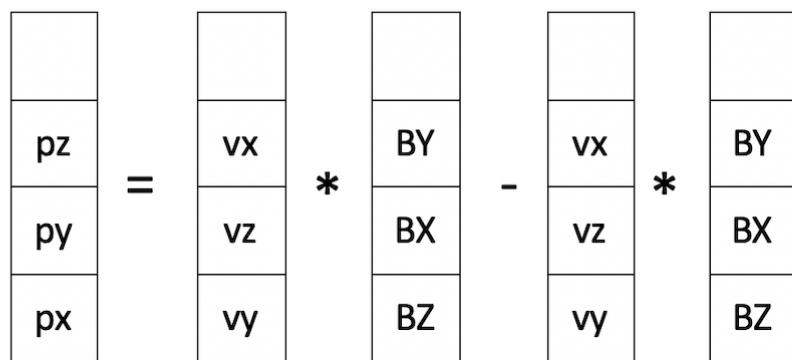
これを使って中点の微分係数を求めるのはさほど難しくなく、中点の微分係数がもとったら、それを使って速度を更新するのも難しくないだろう。

さて、速度ベクトルが、`__m256d vv` に保存されたとする。これはレジスタであるため、メモリに書き戻し

```

double px = v[i].y * BZ - v[i].z * BY;
double py = v[i].z * BX - v[i].x * BZ;
double pz = v[i].x * BY - v[i].y * BX;

```



`__m256d vv`



`_mm256_permute4x64_pd
(vperm pd)`



`__m256d vv_yzx`



図 5: fig/permute.png

てやる必要がある。これはロードと同様にポインタ&v[i].x が指す場所にストアしてやればよろしい。

```
// (w, z, y, z) -> v[i]
_mm256_store_pd((double *)(&(v[i].x)), vv);
```

位置の更新も同様に、更新された速度を使って

```
__m256d vr = _mm256_load_pd((double *)(&(r[i].x)));
vr += vv * vdt;
_mm256_store_pd((double *)(&(r[i].x)), vr);
```

とすれば良い。以上をまとめると、時間発展ルーチンの SIMD 版はこんな感じにかける。

```
void calc_rk2_simd() {
    __m256d vb_zxy = _mm256_set_pd(0.0, BY, BX, BZ);
    __m256d vb_yzx = _mm256_set_pd(0.0, BX, BZ, BY);
    __m256d vdt = _mm256_set_pd(0.0, dt, dt, dt);
    __m256d vdt_h = _mm256_set_pd(0.0, dt * 0.5, dt * 0.5, dt * 0.5);
    const int im_yzx = 64 * 3 + 16 * 0 + 4 * 2 + 1 * 1;
    const int im_zxy = 64 * 3 + 16 * 1 + 4 * 0 + 1 * 2;
    for (int i = 0; i < N; i++) {
        __m256d vv = _mm256_load_pd((double *)(&(v[i].x)));
        __m256d vr = _mm256_load_pd((double *)(&(r[i].x)));
        __m256d vv_yzx = _mm256_permute4x64_pd(vv, im_yzx);
        __m256d vv_zxy = _mm256_permute4x64_pd(vv, im_zxy);
        __m256d vp = vv_yzx * vb_zxy - vv_zxy * vb_yzx;
        __m256d vc = vv + vp * vdt_h;
        __m256d vp_yzx = _mm256_permute4x64_pd(vc, im_yzx);
        __m256d vp_zxy = _mm256_permute4x64_pd(vc, im_zxy);
        __m256d vp2 = vp_yzx * vb_zxy - vp_zxy * vb_yzx;
        vv += vp2 * vdt;
        vr += vv * vdt;
        _mm256_store_pd((double *)(&(v[i].x)), vv);
        _mm256_store_pd((double *)(&(r[i].x)), vr);
    }
}
```

19 行が 23 行になっただけなので、さほどややこしいことはないと思う。増えた 4 行も、磁場ベクトルを作るところとシャッフルのインデックスの準備だけである。いきなりこのルーチンを作るとバグが入りやすいが、シリアルコードに埋め込む形で SIMD 化をすすめ、各ステップごとにシリアルルーチンと SIMD レジスタの内容が一致することを確認しながらやればさほど難しいことはない。

さて、これは SIMD 化はしたが、演算順序その他は全くえていないため、丸め誤差も含めて完全にシリアルコードと一致するはずである。それを確認するため、時間発展後に位置ベクトルのダンプを取ろう。

```
void dump() {
    for (int i = 0; i < N; i++) {
        std::cout << r[i].x << " ";
        std::cout << r[i].y << " ";
        std::cout << r[i].z << std::endl;
```

```
}
```

時間発展後に座標をダンプして、その結果を比較しよう。シリアル版を mag.cpp、SIMD 版を mag_simd.cpp としておき、以下のようにコンパイル、実行、結果の比較をする。

```
$ g++ -std=c++11 -O3 -mavx2 -mfma mag.cpp -o a.out
$ g++ -std=c++11 -O3 -mavx2 -mfma mag_simd.cpp -o b.out
$ time ./a.out > a.txt
./a.out > a.txt 4.58s user 0.27s system 99% cpu 4.876 total

$ time ./b.out > b.txt
./b.out > b.txt 2.54s user 0.29s system 99% cpu 2.849 total

$ diff a.txt b.txt # 結果が一致
```

実行時間が 4.58s → 2.54s と高速化され、かつ実行結果が一致していることがわかる。

さて、この結果を見て「おお！まあまあ SIMD 化の効果あるじゃん！」と思うのは早計である。先程のデータ構造は、構造体の配列の形になっていた。このような構造を **Array of Structure (AoS)** と呼ぶ。逆に、同じデータを、配列の構造体で表現することもできる。

```
double rx[N], ry[N], rz[N];
double vx[N], vy[N], vz[N];
```

ここでは配列を構造体にまとめていないが、このようなデータ構造を **Structure of Array (SoA)** と呼ぶ。AoS と SoA、どちらが良いかは場合によるのだが、SIMD 化においては SoA にしたほうが性能がでる場合が多い。

先程の AoS のシリアルコードと全く同じ内容を、データ構造を SoA にして書いたものを mag_soa.cpp とする。例えば、時間発展ルーチンはこんな感じにかける。

```
void calc_rk2() {
    for (int i = 0; i < N; i++) {
        double px = vy[i] * BZ - vz[i] * BY;
        double py = vz[i] * BX - vx[i] * BZ;
        double pz = vx[i] * BY - vy[i] * BX;
        double vcx = vx[i] + px * dt * 0.5;
        double vcy = vy[i] + py * dt * 0.5;
        double vcz = vz[i] + pz * dt * 0.5;
        double px2 = vcy * BZ - vcz * BY;
        double py2 = vcz * BX - vcx * BZ;
        double pz2 = vcx * BY - vcy * BX;
        vx[i] += px2 * dt;
        vy[i] += py2 * dt;
        vz[i] += pz2 * dt;
        rx[i] = rx[i] + vx[i] * dt;
        ry[i] = ry[i] + vy[i] * dt;
        rz[i] = rz[i] + vz[i] * dt;
    }
}
```

`v[i].x` が `vx[i]` とかになっているだけで、そのまんまなのがわかるかと思う。これも実行して、結果を比較しよう。

```
$ g++ -std=c++11 -O3 -mavx2 -mfma mag_soa.cpp -o c.out
$ time ./c.out > c.txt
./c.out > c.txt 1.20s user 0.28s system 98% cpu 1.493 total

$ diff a.txt c.txt # 結果が一致
```

手で SIMD 化した場合に比べて、 $2.54\text{s} \rightarrow 1.20\text{s}$ と倍以上高速化されたことがわかる。もともとのシリアルコードが 4.58s だったので、4 倍近い。つまり、このコードはコンパイラによる自動 SIMD 化により 4 倍早くなったわけで、理想的な SIMD 化ができたことがわかる。

実際にコンパイラはこんなコードを吐いている。

L10:

```
vmovapd (%rdi,%rax), %ymm0
vmovapd (%r9,%rax), %ymm11
vmovapd (%r8,%rax), %ymm10
vmulpd %ymm7, %ymm0, %ymm2
vmulpd %ymm11, %ymm5, %ymm1
vfmsub231pd %ymm6, %ymm0, %ymm1
vmulpd %ymm4, %ymm1, %ymm1
vmulpd %ymm10, %ymm6, %ymm3
vfmadd132pd %ymm8, %ymm10, %ymm1
vfmsub231pd %ymm11, %ymm7, %ymm3
vfmsub231pd %ymm10, %ymm5, %ymm2
vmulpd %ymm4, %ymm3, %ymm3
vmulpd %ymm4, %ymm2, %ymm2
vfmadd132pd %ymm8, %ymm0, %ymm3
vfmadd132pd %ymm8, %ymm11, %ymm2
vmulpd %ymm6, %ymm1, %ymm9
vfmsub231pd %ymm7, %ymm2, %ymm9
vmulpd %ymm5, %ymm2, %ymm2
vfmadd132pd %ymm4, %ymm0, %ymm9
vfmsub231pd %ymm6, %ymm3, %ymm2
vmovapd %ymm9, (%rdi,%rax)
vfmadd132pd %ymm4, %ymm10, %ymm2
vfmadd213pd (%rsi,%rax), %ymm4, %ymm9
vmovapd %ymm2, (%r8,%rax)
vmulpd %ymm7, %ymm3, %ymm0
vfmadd213pd (%rdx,%rax), %ymm4, %ymm2
vfmsub231pd %ymm5, %ymm1, %ymm0
vmovapd %ymm9, (%rsi,%rax)
vfmadd132pd %ymm4, %ymm11, %ymm0
vmovapd %ymm2, (%rdx,%rax)
vmovapd %ymm0, (%r9,%rax)
vfmadd213pd (%rcx,%rax), %ymm4, %ymm0
vmovapd %ymm0, (%rcx,%rax)
```

```

addq $32, %rax
cmpq $800000, %rax
jne L10

```

最内ループだけ抜き出したが、基本的に `ymm` レジスタだらけであり、理想的に SIMD 化されていることがわかる。また、シャッフル命令も全く出ていないことがわかる。コンパイラはループを素直に 4 倍展開し、各レジスタに (`x1, x2, x3, x4`) のような形でデータを保持して計算している。ループカウンタは `%rax` で、毎回 32 ずつ増えており、800000 になったら終了なので、このループは 25000 回転することがわかる。

ちなみに、先程手で SIMD 化したループのアセンブリはこうなっている。

L13:

```

vmovapd (%rax), %ymm2
addq $32, %rax
addq $32, %rdx
vpermpd $201, %ymm2, %ymm0
vpermpd $210, %ymm2, %ymm1
vmulpd %ymm3, %ymm1, %ymm1
vfmsub132pd %ymm4, %ymm1, %ymm0
vfmadd132pd %ymm6, %ymm2, %ymm0
vpermpd $201, %ymm0, %ymm1
vpermpd $210, %ymm0, %ymm0
vmulpd %ymm3, %ymm0, %ymm0
vfmsub231pd %ymm4, %ymm1, %ymm0
vfmadd132pd %ymm5, %ymm2, %ymm0
vmovapd %ymm0, -32(%rax)
vmovapd %ymm0, %ymm1
vfmadd213pd -32(%rdx), %ymm5, %ymm1
vmovapd %ymm1, -32(%rdx)
cmpq %rcx, %rax
jne L13

```

`vpermpd` がシャッフル命令である。ループボディがかなり小さいが、このループは 100000 回まわるため、25000 回しかまわらないコンパイラによる自動 SIMD 化ルーチンには勝てない。大雑把な話、ループボディの計算コストが半分だが、回転数が 4 倍なので 2 倍負けた、という感じである。

上記の例のように、「いま手元にあるコード」をがんばって「そのまま SIMD 化」して高速化しても、データ構造を変えるとコンパイラがあっさり自動 SIMD 化できて負けることがある。多くの場合「SIMD 化」はデータ構造のグローバルな変更を伴う。先のコードの AoS 版である `md.cpp` と、SoA 版である `md_soa.cpp` は、全く同じことをしているが全書き換えになっている。今回はコードが短いから良いが、10 万行とかあるコードだと「やっぱり SoA の方が早いから全書き換えで！」と気軽に言えないだろう。また、デバイスによってデータ構造の向き不向きもある。例えば「CPU では AoS の方が早いが、GPGPU では SoA の方が早い」なんてこともざらにある。こういう場合には、ホットスポットルーチンに入る前に AoS \leftrightarrow SoA の相互変換をしたりすることも検討するが、もちろんその分オーバーヘッドもあるので面倒くさいところである。

まあ、以上のようにいろいろ面倒なことを書いたが、ちゃんと手を動かして上記を試してみた方には「SIMD 化は(原理的には)簡単だ」ということには同意してもらえると思う。MPI も SIMD 化も同じである。いろいろ考えることがあって面倒だが、やること自体は単純なので難しくはない。今回はシャッフル命令を取り上げたが、他にもマスク処理や `gather/scatter`、`pack/unpack` など、SIMD には実に様々な命令がある。

しかし、「そういう命令欲しいな」と思って調べたらたいがいある。あとは対応する組み込み関数を呼べばよい。要するに「やるだけ」である。ただし、MPI 化は「やれば並列計算ができ、かつプロセスあたりの計算量を増やせばいくらでも並列化効率を上げられる」ことが期待されるのに対して、SIMD 化は「やっても性能が向上するかはわからず、下手に手を出すよりコンパイラに任せた方が早い」なんてこともある。全く SIMD 化されていないコードに対して SIMD 化で得られるゲインは、256bit なら 4 倍、512 ビットでも 8 倍程度しかなく、現実にはその半分も出れば御の字であろう。SIMD 化はやってて楽しい作業であるが、手間とコストが釣り合うかどうかは微妙だな、というのが筆者の実感である。

おわりに

2003年、自分が博士課程の学生の頃、地球シミュレータのアカウントを手に入れることができた。当時、地球シミュレータは二位以下に大差をつけて圧倒的な存在として君臨する、世界No.1のスパコンだった、私は地球シミュレータを使うため、あるプロジェクトに所属していた。自分で言うのもなんだが、それなりに戦力として期待されていたのではないかと思う。しかし僕はD3であり、博士論文を書くのに忙しかった。さらに当時、地球シミュレータはネットワーク越しの利用が許されておらず、使いたければ端末を予約して新杉田にある海洋研まで行く必要があった。これは言い訳になってしまふが、自分には博士論文を書きつつ、遠くにある端末まで行ってコードを開発、チューニングする余力はなかった。結局、ほとんど地球シミュレータを使うことができないまま卒業、就職し、プロジェクトも終了。卒業した年(2004年)の11月、地球シミュレータもTop500の一位の座をBlue Geneに明渡した。

「せっかく世界一のスパコンを触る機会があったのに、ほとんど触ることができなかつた」

この悔しい思いは、「次に日本が国策スパコンを作ったら、絶対に使いこなしてやる」という強い思いとして残った。

時は流れて2011年、僕はまだ一般公開前の「京」の評価をするため、家族を連れて半年ほど神戸に詰めていた。僕はかかわっていないのだが、ちょうどこの時「京」を使ったHPLの測定が行われていた。2011年6月、京は地球シミュレータ以来、7年ぶりとなるTOP500一位を獲得する。さらに糸余曲折あって2015年3月、僕は自分の提案したプロジェクトが認められ、とある論文を投稿するために、「京」にフルノードジョブを投げていた。普段、「京」は多くのユーザで共有して使っており、フルノードジョブが実行できる機会は月に1度しかなかつた。論文の締切が4月頭で、1月から3回のチャンスがあつた。事前に4096ノード程度で問題ないことを確認していたジョブだったが、82944ノード実行時に二度失敗してしまい、後がなかつた(ちなみに失敗原因は、ノード数に比例して大きくなるテンボラリバッファを使っており、4096ノードでは特に問題とならないが、82944ノード実行するとメモリを食いつぶしてしまう、というものだった)。ジョブを投入すると、ジョブの開始時と終了時にメールが飛んでくる。したがつて、ジョブの開始メールと終了メールが同時に飛んできたら「あ、ジョブがこけたな」ということがわかる。三度目のチャレンジ、ジョブの開始メールが飛んできて、でも終了メールが飛んでこない。これは今、「京」フルノードを自分のジョブが占有して走っていることを示す。リアルに「あのスパコンで走ってるジョブ、俺のなんすよwww」状態である。しかし、ジョブが終了するまでは安心できない。ジョブ終了メールが飛んでくるまで、祈るような気持ちで待つ。そしてジョブ終了後、正常終了を示すログを見たあと、ダンプファイルを可視化したところ、そこには大きな泡があつた(当時、僕は分子動力学法による気泡生成シミュレーションをしていた)。「このジョブが失敗したら論文の投稿ができない」というジョブは、なんとか無事に完走してくれ、おかげで論文を投稿することができた。当時の緊張と、失敗時の落胆、三度目のジョブ実行中の高揚感を今でも思い出せる。

仕事柄、日常的にスパコンに触れているのだが、スパコンには夢があると思う。スパコンには、普通のパソコンではできないようなことができそうな雰囲気がある。最初、僕がスパコンを使い始めた時には「こんなもの自分で使えるんだろうか」という不安感があった。しかし、使ってみるとわりとあっさりと使え「スパコンを使いこなして俺スゲー!」という気持ちになった。さらに使っていくと、どうもスパコンはもっと可能性を秘めているのに、「自分が使いこなしていない感」が強くなってきた。今でも、「スパコンのポテンシャルを引き出した!面白い計算ができた!」と納得できる仕事はできていない。でもやっぱり、スパコンは面白いと思う。

スパコンには大きなお金がからむため、ニュースなどで話題となるのはネガティブなものが多い印象である。せっかくスパコンで面白い計算ができたり、面白い技術が開発されたりしてもあまり取り上げられず、「二位じゃだめなんでしょうか」発言に代表されるような、スパコンそのものの楽しさから離れたトピックが話題となることが多い。その際、あまりスパコンに詳しくなさそうな人による「解説」に、多くの人が

賛同したりしているのを見て、それなりにスパコンと長い付き合いがある身としては歯がゆい思いをしていた。どんな分野でもそうだと思うが、実際に手を動かして使ってみなければ「スパコンとはどういうものか」は体感できない。値段、ベンチマークの数値や理論性能値など、そういうものでスパコンを議論するのは自由だが、少なくとも僕はそれで「スパコン」を理解できる気はしない。この文章も、「スパコンを知らない人がざっと読んでわかった気になること」を目的として書いたものではない。並列計算というのを思ったより簡単であり、誰でもすぐスパコンが使えるようになること、でもそのあと「スパコンを使い倒そう」とすると、とたんに様々な「闇」があつて難しいこと、でもそれが面白いこと、そんなこんなを実感して欲しい、僕はそういう気持ちで書いた。

近い将来、学会の懇親会などで『一週間でなれる！スパコンプログラマ』を読んでスパコンプログラマになりました」と、見知らぬ学生さんに突然話かけられるのを夢見て、筆を置くことにする。

2018年 Hiroshi Watanabe