平成 24 年 10 月 29 日

チューニングとその周辺

渡辺宙志

東京大学物性研究所

概 要

プログラムのボトルネックの探し方や性能が出ない理由の調べ方、並列化にまつわるトラブル、そしてソフトウェアの公開方法について簡単にまとめる。

目 次

1　このテキストについて

このテキストは、2012 年 3 月 12 日〜14 日にかけて行われた CMSI 若手技術交流会のために用意されたレジュメに加筆したものである。チューニングや並列化、それにともなうプログラム設計からプログラムの公開まで、渡辺の経験を雑多にならべたものである。経験に基づくものであり、特に系統的な教育を受けたわけでもないのであまり正確でない記述があるかもしれないが、そのあたりは容赦していただきたい。このレジュメに対応するスライドも公開しているので、合わせて参照されたい [1]。

2　はじめに

2.1　並列化、高速化の必要性

並列化とは、複数のプロセスが協調してひとつの仕事を行う手段である。まず強調しておきたいのは、並列化は手段であって目的ではないということである。まったく並列化されていないコードを非自明並列化するのにはかなりの時間がかかる。並列化しないで、もしくは自明並列で目的が達成されるのであればそれに越したことはない。そもそもなぜ並列化をするのか。その目的から考えておきたい。並列化により稼ぐことができるものは、時間か空間、もしくはその両方である。時間を稼ぐとは、たとえばシングルコアでは 100 時間かかるジョブが、100 コアで並列計算することで 1 時間で終わらせるようなことで、空間を稼ぐとは、そもそも目的とする系がメモリなどの問題でシングルノードに乗らないため、複数のノードを使うことで大きな系を計算しようとするようなことである。それぞれ、後述するストロングスケーリング、ウィークスケーリングに対応する。また、単にサイズを稼ぐつもりであっても、平衡状態に興味がある場合には時間に関する制約も同時にかかる。三次元系のシミュレーションをすることを考えよう。一片の長さが L の立方体領域を計算することにする。系の自由度を N として、$O(N)$ で計算するスキームを使えば、同じステップ数だけ計算するのにかかる時間は、L に対して $O(L^3)$ で増加する。しかし、もし平衡状態に興味がある場合、系が平衡に達するまでの時間は一般に L^2 に比例するため、興味ある物理量を測定できるまでにかかる時間は $O(L^5)$ に比例することがわかる。したがって、結果が出るまでの時間を固定した場合、$O(N)$ 法を使い、かつ並列化効率が完璧である場合でも、プロセス数を P 倍にすることで系のサイズは $P^{1/5}$ にしかならない。並列化率が完璧なコードを持っていたとしても、シングルプロセスで一辺 L の系を平衡化するのにかかる時間で一辺 $2L$ の系を平衡化するには、32 コア必要、ということになる。もしくは、一辺 $2L$ の系を 8 コアで計算した場合には (同じステップ数を同じ時間で計算できるにも関わらず) 平衡化までにシングルプロセスで一辺 L の系を平衡化するのにかかる時間の 4 倍かかる、と読み替えても良い。なお、単純拡散しないような系、特になんらかの臨界点近傍や、遅いダイナミクスがある系ではさらに緩和に時間がかかるため、このスケーリングは悪くなる[1]。一般に、サイズを稼ぐ並列化よりも、時間を稼ぐ並列化の方が条件が厳しくなる。また、実際のアプリケーションでも、大きな系を短時間計算するより、ある程度小さな系を長時間計算するニーズの方が多いと思われる。従って、並列化する前のプログラムが速いに越したことはない。また、プログラムの高速化にともなって、データの保持の形が変わる場合が多いため、プログラムのチューニングが不十分のまま並列化すると、チューニングのために並列部分を書き直す必要がでてしまい、結局書き直し、ということになりかねない。以上から、まずは並列化の前に、それなりにコードを最適化、高速化しておくことが望ましい。

高速化にはいくつかの段階がある。まず一番アプリケーション寄りの階層では、アルゴリズムレベルの高速化がある。原理的に遅いアルゴリズムをどんなにチューニングしても無駄である。たとえば $O(N^2)$ の方法を $O(N)$ の計算量に落とすなど、計算量を大きく削減する方法があればそれを採用し、理論的に計算量がこれ以上削減できない、という段階となってからチューニングすべきである。次に問題となるのが

[1] ここでは平衡状態を扱ったが、一般に非平衡状態のタイムスケールは、音速など、系のサイズにあまり依存しない物理量で決まっていることが多い。この場合には計算量のスケーリングは $O(L^3)$ で済む。

メモリバンド幅である。CPU の性能向上に比べて、メモリと CPU 間の通信速度はさほど向上していない。そのため、CPU から見てメモリはどんどん遠くにあるよう見える。多くのアプリケーションにおいて、CPU とメモリの間の通信速度、すなわちメモリバンド幅がボトルネックとなる。必要なデータがなるべくキャッシュに乗るようにプログラムを工夫する必要がある。最後に、CPU の計算能力を最大限活用するためのチューニングを行う。これには SIMD 化やソフトウェアパイプライニングなど、パイプラインをにらんだ最適化が含まれる。ただし、このレベルを人力でチューニングするのはかなり難しく、またその恩恵も労力に見合ったものになりにくい。一般論として、ほとんどのプログラムはアルゴリズムレベルの最適化は行っていることが多いので、まずはメモリまわりの最適化に注力することになる。その上で、CPU まわりの最適化はコンパイラに任せた方が良い場合が多い[2]。

2.2 高速化、その前に 1 〜テストはしっかりと〜

何度も繰り返すが、計算は結果を得るための手段である。したがって、結果が正しくなければどんなに早いコードを書いても無意味である。そこで、まずは速度などはまったく考えず、絶対に正しいと信じることができるコードを書くべきである。最初から高速化をにらんだコードを書くと、得られた結果が正しいのか、正しくなければ手法による誤りなのか、高速化による誤りなのか判断できず、デバッグに無尽蔵に時間を食われてしまう。一般、小さい系においては厳密解かそれに類する結果を得ることができるので、まずはそれと比較することが大事である。また、ある程度大きな系においても、時間はかかるが信頼できる方法というのがあることが多い。それらの方法と比較することでバグを十分につぶしておく必要がある。

短距離古典分子動力学法 (Molecular Dynamics, MD) で例を挙げよう。カットオフがあるような短距離 MD では、相互作用粒子リストをあらかじめ探しておく必要がある。それを何も考えずに実装すると、Algorithm 1 に示したようなコードになるであろう。これはいわゆる $O(N^2)$ コードであり、粒子数 N が

Algorithm 1 Finding the interacting particle pairs

```
1: for i = 1 to N − 1 do
2:   for j = i + 1 to N do
3:     if |q_i − q_j| < Cutoff Length then
4:       Register pair (i, j)
5:     end if
6:   end for
7: end for
```

増えると計算量が増えるため実際のプロダクトランには向かない。一般には空間をメッシュに切り、それぞれの粒子がすむ住所を設定し、住所から逆引きして相互作用粒子を探すことで、計算量を $O(N)$ に落とすということが行われている。しかし、だからといって最初から $O(N)$ のコードを書こうとすると、ほぼ確実にバグが入り、かつ比較コードがないためそのデバッグは困難である。そこで、以下のようなプログラム開発ステップをふむ。

1. $O(N^2)$ で数百〜千粒子程度計算を行い、エネルギーが保存することを確認する (境界条件や積分スキームのデバッグ)。
2. $O(N)$ の相互作用ペアリスト作成ルーチンを書き、時間発展させる前に、同じ configuration から作成されたペアリストが $O(N^2)$ 法と $O(N)$ 法で一致することを確認する。

[2] しかし、希に「コンパイラの吐くコードがアホだ」と、ほとんど自分でハンドアセンブルに近い形でコードを書いてしまう人が存在する。個人的な意見だが、こういう人は若い頃からずっとそういうことばかりやってることが多く、訓練次第でどうにかなるものではないと思う。

3. $O(N)$ 法で時間発展させた結果、エネルギーや温度などの物理量の時間発展が $O(N^2)$ 法と一致することを確認。

実際の渡辺の経験では、$O(N)$ 法に落とした時、特に周期境界条件まわりでバグが入り、そのバグをつぶすのに $O(N^2)$ との結果との比較が極めて役に立った。ペアリスト構築に限らず、一般に高速化手法を実装する前に、比較用の単純かつ信頼できる手法を実装しておき、段階的に比較することでバグをつぶしておくことが極めて大事である。これは並列化においても同様で、シングルスレッドの計算と十分に比較することでバグをつぶしておくことが大事である。

2.3 高速化、その前に 2 〜プロファイラの使い方〜

闇雲に高速化を行う前に、まず自分のコードのどの部分が時間がかかる部分であるのかを知る必要がある。一般にアプリケーションはある特定のルーチンが計算時間の大部分を占めることが多く、その場所をホットスポットと呼ぶ。まずやるべきことは、そのホットスポットの同定である。ホットスポットを探すのに一番簡単なのはプロファイラを使うことである。ホットスポットが見つかったら、そのルーチンが早いのか遅いのか、遅いとしたらどの程度「のびしろ」があるのかを確認する必要がある。プロファイラなどでサブルーチンごとの FLOPS 値などの測定ができる場合もあるが、単にピーク性能比が悪いからといってプログラムにチューニングの余地があるとは限らない。むしろ、多くの場合においてボトルネックはメモリバンド幅にあることが多い。また、サブルーチンごとにシステムサイズに対してどれくらいの計算量であるかも把握しておく必要がある。テスト用に小さい系で計算している時にはあまり時間がかかっておらず、目立たなかったルーチンが、実は $O(N^2)$ の計算量で大きな系を計算したらそこがボトルネックになる、ということもあるからである。

プロファイラには様々な種類があり、かつコンパイラを提供しているベンダーが独自のプロファイラを提供している場合もあるが、ここでは無料で使える gprof の使い方を簡単に紹介する。まず、List 1 のように適当なコードを用意する。ここでは行列積 (matmat)、行列ベクトル積 (matvec)、ベクトルの内積 (vecvec) を、効率を考えずに書き下してある。ループの回り方を見ればわかる通り、行列積は $O(N^3)$、行列ベクトル積は $O(N^2)$、ベクトルの内積は $O(N)$ であり、行列積がボトルネックであろうことが推測できる。このプログラムを `test.cc` として保存し、-pg オプションをつけてコンパイル、実行する。なお、最適化オプションによっては、関数がインライン展開されてしまって正しく関数ごとのプロファイルが得られないことがある。その場合は-fno-inline をつけてインライン展開を抑止する。ついでに time コマンドで実行時間も測っておこう。

```
$ g++ -pg -fno-inline test.cc
$ time ./a.out
./a.out 1.01s user 0.00s system 99\% cpu 1.013 total

$ ls
a.out* gmon.out test.cc
```

-pg つきでコンパイルされたプログラムを実行すると、gmon.out というプロファイル情報を格納したバイナリファイルが出力される。このファイルの内容をテキストで出力するプログラムが gprof である。gprof に実行ファイル名とプロファイル情報のファイル名を引数として実行する、つまり

```
$ gprof a.out gmon.out
```

とすると、List. 2 のような出力が得られる。なお、実行ファイル名が a.out の場合は省略できる。また、プロファイル情報のファイル名も gmon.out であれば省略できる。gprof の出力の最初に `Flat profile:` とあるのが、関数ごとの実行時間である。計算時間がもっともかかったもの、すなわち計算が重い関数から順番に並べてある。このリストを見ると、最も重いのが `matmat()`、すなわち行列積で、計算時間の 100%近くを占めていることがわかる。なお、これは 100%を超えているのはサンプリングによる誤差である。

gprof など、多くのプロファイラはサンプリングによる解析を行う。これは、一定時間間隔ごとに割り込みをかけ、「プログラムがどこを実行中であるか」を調べて、その割合をもってそれぞれの関数の「重さ」を推定する方法である。gprof の出力を見ると、`Each sample counts as 0.01 seconds.` という記述がある。これは 0.01 秒ごとに割り込みをかけて調べており、調べた時に実行中だった関数を 0.01 秒かかったと見なした、という意味であり、実際の実行時間は、ほぼ 0 から 0.02 秒までの幅があり得る。サンプリング型のプロファイラでは、このように観測時間を「量子化」して測定するため、実行時間が短いと正しく統計データが出ない。大雑把には、サンプリング間隔の 100 倍くらいの実行時間であればだいたい信頼できる、と覚えておけば良い。この例では、サンプリング間隔が 0.01 であり、`matmat` の実行時間がおよそ 1 秒であるので、サンプリング回数が 100 回程度あった、と考えられる。誤差はサンプリング回数の平方根に反比例すると思えば、誤差も含めて 1.0 ± 0.1 くらいの実行時間であったと推定される。

なお、Mac OS X では gprof を使う事ができない。-pg つきでコンパイル、実行すると gmon.out は出力されるのだが、gprof で出力された結果には何も表示されない。Mac でプロファイルデータが欲しい場合には、Xcode の Shark を使うと良い。Ctrl+Space で出てくる Spotlight に「shark」と入力すれば見つかるはずである。Shark が起動したら、一番右のプルダウンメニューから「Launch」を選ぶ。Launch はリストの一番上にあるはずである。その状態で「Start」をクリックし、実行ファイルを選んで OK を押すとサンプリングが開始される。なお、Shark を利用する際には特にコンパイル時にオプションは必要ない。サンプリングが終わると、プロファイルが出力される。この例では `matmat` が全体の 99.2%の実行時間を占めていることがわかる。Shark のデフォルトのサンプリングレートは 1ms であり、その分 gprof より精度が高くなっている。なお、プロファイル画面で関数名をダブルクリックすると、対応する関数のアセンブリコードが表示される。実行時間の重さが色で可視化されているので、さらなるチューニングの参考にしてほしい。

ここで挙げた例のように、多くの場合においてはプログラムの実行時間の大部分をごく一部の関数が占めることが多い。その場合にはその関数のチューニングに専念すれば良い。しかし、プログラムによっては、様々な関数が似たような実行時間である場合もある。また、ホットスポットの改善に成功した場合も、後は同じような計算時間を持つ関数ばかりで際立って重い関数がない、という状態になるかもしれない。このような状態となったプログラムの高速化は労力がかかる割に見返りが少ない。たとえば計算時間の 95%を占める関数があれば、そこを 2 倍高速化すれば全体も 1.9 倍となるが、10 倍高速化しても 6.9 倍にしかならない。また、10%の計算時間を占める関数が 10 個で構成されているプログラムは、一つの関数を 2 倍高速化することに成功しても 5%の速度向上効果しか得られない。闇雲にチューニングしても、手間の割に見返りが少なく、チューニングしている時間だけ計算してしまった方が結果が早く出る、ということになりかねない。最初に、どの程度までチューニングすべきかの目処をつけておくのが大事である。

3 メモリマネジメント

3.1 記憶階層とバンド幅

計算データは主記憶と呼ばれるメモリに格納されているが、計算を行うためにはメモリからレジスタまでデータを持ってこなければならない。どんなに高い計算能力があっても、データの転送が間に合わなければその時間 CPU は遊んでしまい、その計算能力を活かす事ができない。CPU の計算能力に対してどれだけメモリ転送能力があるかを表す指標として、メモリのデータ転送能力 (Byte/s) と計算能力 (FLOPS) の比を取った値がよく用いられる。これを Byte/FLOP、通称 B/F 値と呼ぶ。B/F 値は計算機によって異なるが、典型的な値は 0.5 ～ 1 程度である。この値が数値計算にとってどのくらい厳しい値であるか、例を挙げて考えてみよう。A=B*C という計算を考える。この計算を実行するには、B と C という倍精度実数を二つ取ってきて、一度掛け算し、A に書き戻す必要がある。倍精度実数は、ひとつ 8Byte である。2 回の読み込み (load) と、1 回の書き込み (store) があるから、全体で 24Byte の load/store が必要となる。

List 1: プロファイラサンプル

```
//------------------------------------------------------------------------
#include <iostream>
const int N = 500;
double A[N][N];
double B[N][N];
double C[N][N];
double x[N],y[N];
//------------------------------------------------------------------------
void
init(void){
  for(int i=0;i<N;i++){
    for(int j=0;j<N;j++){
      A[i][j] = static_cast<double>(i*j);
      B[i][j] = static_cast<double>(i+j);
      C[i][j] = 0.0;
    }
    x[i] = static_cast<double>(i);
    y[i] = static_cast<double>(i);
  }
}
//------------------------------------------------------------------------
void
matmat(void){
  for(int i=0;i<N;i++){
    for(int j=0;j<N;j++){
      for(int k=0;k<N;k++){
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
//------------------------------------------------------------------------
void
matvec(void){
  for(int i=0;i<N;i++){
    for(int j=0;j<N;j++){
      y[i] += C[i][j]*x[j];
    }
  }
}
//------------------------------------------------------------------------
double
vecvec(void){
  double sum = 0.0;
  for(int i=0;i<N;i++){
    sum += x[i]*y[i];
  }
}
//------------------------------------------------------------------------
int
main(void){
  init();
  vecvec();
  matvec();
  matmat();
}
//------------------------------------------------------------------------
```

List 2: gprof 出力結果 (抜粋)

```
Flat profile:

Each sample counts as 0.01 seconds.
  % cumulative self self total
 time seconds seconds calls ms/call ms/call name
100.57 0.93 0.93 1 925.26 925.26 matmat()
  0.00 0.93 0.00 1 0.00 0.00 global constructors keyed to A
  0.00 0.93 0.00 1 0.00 0.00 __static_initialization_and_destruction_0(int, int)
  0.00 0.93 0.00 1 0.00 0.00 init()
  0.00 0.93 0.00 1 0.00 0.00 matvec()
  0.00 0.93 0.00 1 0.00 0.00 vecvec()

granularity: each sample hit covers 2 byte(s) for 1.08% of 0.93 seconds

index % time self children called name
                                                 <spontaneous>
[1] 100.0 0.00 0.93 main [1]
               0.93 0.00 1/1 matmat() [2]
               0.00 0.00 1/1 vecvec() [13]
               0.00 0.00 1/1 init() [11]
               0.00 0.00 1/1 matvec() [12]
-----------------------------------------------
               0.93 0.00 1/1 main [1]
[2] 100.0 0.93 0.00 1 matmat() [2]
-----------------------------------------------
               0.00 0.00 1/1 __do_global_ctors_aux [14]
[9] 0.0 0.00 0.00 1 global constructors keyed to A [9]
               0.00 0.00 1/1 __static_initialization_and_destruction_0(int, int) [10]
-----------------------------------------------
               0.00 0.00 1/1 global constructors keyed to A [9]
[10] 0.0 0.00 0.00 1 __static_initialization_and_destruction_0(int, int) [10]
-----------------------------------------------
               0.00 0.00 1/1 main [1]
[11] 0.0 0.00 0.00 1 init() [11]
-----------------------------------------------
               0.00 0.00 1/1 main [1]
[12] 0.0 0.00 0.00 1 matvec() [12]
-----------------------------------------------
               0.00 0.00 1/1 main [1]
[13] 0.0 0.00 0.00 1 vecvec() [13]
-----------------------------------------------
```

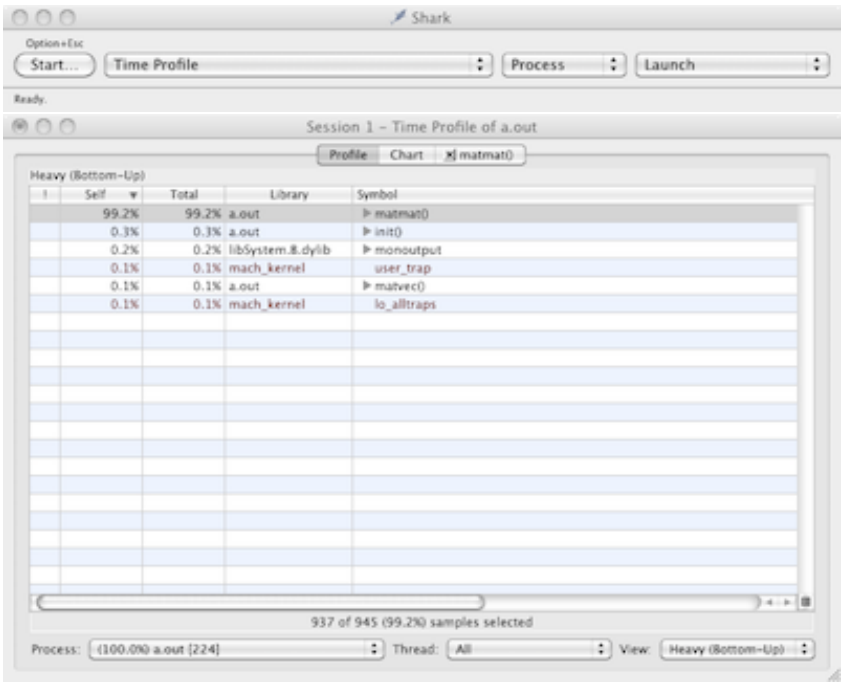


図 1: Shark の実行画面。

一方、計算は 1 度しかしていない。独立な乗算をひたすら繰り返すような計算が、ピーク性能を出すためには B/F 値が 24 以上でなければならない[3]。逆に B/F 値が 0.5 である計算機で A=B*C をただ繰り返すような計算をしようとすると、理論ピーク性能の 2%強しか使えず、残りの 98%の時間は CPU が遊んでしまうことになる。さらに、実際の計算ではメモリバンド幅だけではなく、レイテンシも重要である。メモリバンド幅とは、通信が始まった状態から通信が終わるまでにどれだけのデータを流すことができるか、という値であり、レイテンシとは CPU が必要なデータを要求してから、実際に届くまでの時間 (サイクル数) である。システムによってはレイテンシが数百サイクル以上かかる場合もあり、広大なメモリ空間をランダムにアクセスするようなプログラムでは計算時間がほとんどメモリレイテンシ、ということになりかねない。

このように B/F 値がプログラムから要求される値よりも低い場合に計算資源を有効に使う為には、メモリから持って来たデータを CPU に近いところに置いておき、使い回す必要がある。そのために CPU と主記憶の間に用意されているのがキャッシュである。キャッシュは CPU 間と高速に接続されており、キャッシュにあるデータは高速に、かつ低レンテンシでアクセスできる。最近ではキャッシュも階層化されており、CPU に近い側から L1 キャッシュ、L2 キャッシュ、L3 キャッシュなどと呼ばれる。最終的に、データの格納領域は、CPU に近い側から見てレジスターキャッシュー主記憶 (ー並列化している場合は他のノードにある主記憶) など階層構造をなしており、CPU に近いほど小容量高速低レイテンシ、CPU から遠くなるほど大容量低速高レイテンシである。プログラムを組む際には、いかに CPU に近い記憶媒体に必要なデータを置いておけるかが鍵となる。

3.2 キャッシュ利用効率の見積もり

CPU が計算に必要とするデータがキャッシュに載っていればいるほど計算は速くなる。逆に、キャッシュに入りきらないようなデータを扱うとキャッシュミスが発生し、計算は著しく遅くなる。自分のコードが

[3] ここでの議論ではレイテンシや演算器の数などは無視している。

どのくらいキャッシュを使えているかを高度なプロファイラを使って調べることができる場合もあるが、システムサイズを大きくしていって、計算性能がどうなるかを見るのが一番簡単である。分子動力学法コードで例を挙げよう。3 次元分子動力学法コードでは、扱うデータは粒子ごとに 3 次元の座標と運動量で、倍精度実数が 6 つ必要になる。倍精度実数は一つ 8Byte であるから、粒子あたり 48Byte 必要になる。物性研のシステム B は、L2 キャッシュが 256KB、L3 キャッシュが 8MB であるので、これを 48 で割るとそれぞれ 5.3×10^3 個と 1.7×10^5 個に対応する。実際に粒子数をかえて計算を行ってみた結果を図 2 に示す [2]。横軸が粒子数、縦軸が単位時間あたり何粒子をあつかうことができたかを表す値で、詳細は省くがこの値が大きければ大きいほど速いことを意味する。黒丸がキャッシュを上手く使えていない場合である。粒子数を増やしていくと、L2 から溢れたところで性能が下がり始め、L3 から溢れると急激に性能が落ちる。白丸は、キャッシュを上手く使うように工夫したコードである。全体のデータはもちろんキャッシュに収まりきらないが、必要なデータがだいたいキャッシュに載っているため、サイズを大きくしても性能がほとんど劣化しないことがわかる。この例では最大で 3 倍程度の差がついているが、よりレイテンシが効いてくるコードでは、工夫した場合としない場合で 100 倍以上の差が出ることも珍しくない。まずは自分のコードをシステムサイズを変えて計算時間を測定してみて、その性能 ($O(N)$ のアルゴリズムなら実行時間を自由度で割ったもの) が極端に低下するようであれば、キャッシュミスを疑うべきである。キャッシュの利用効率を挙げるにはブロック化やソートなど、問題に応じて様々な方法があるが、ここでは詳細は述べない。

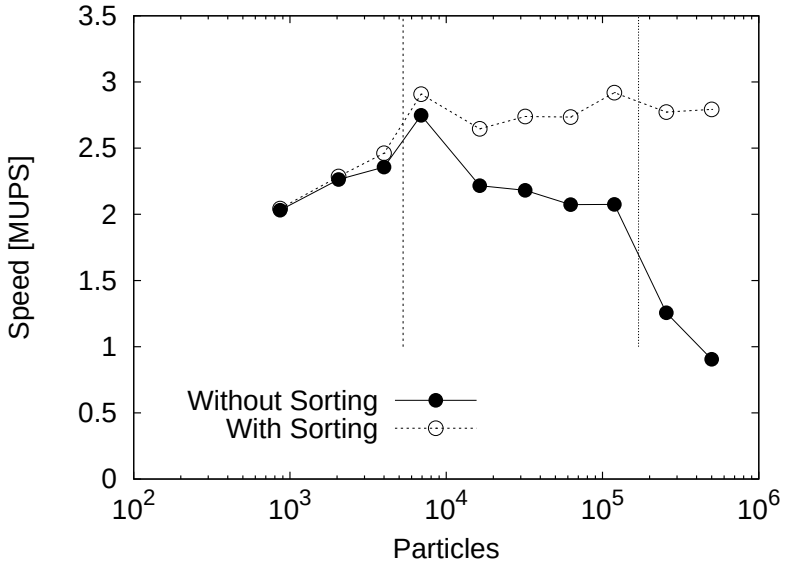


図 2: キャッシュ利用効率の見積もり。黒丸がキャッシュをうまく使えていない場合。L2 キャッシュ、L3 キャッシュのサイズに対応する粒子数に縦の点線を引いてある。ちょうど粒子が L2、L3 キャッシュから溢れるところで性能が落ちていることがわかる。

3.3 レジスタの活用例

メモリ階層の一番 CPU 側に存在する記憶媒体がレジスタである。全ての演算はレジスタを通して行われる。一般に、レジスタを効率的に使うコードを人の手で書くのは困難で、コンパイラに任せた方が良い結果がでることが多い。それでも、たまにレジスタを意識した方が良い場合もある。List 3 は、3 次元空間に粒子をランダムに配置し、全ての粒子間において 12-6 型の Lennard-Jones ポテンシャルによる力積を計算するコードを単純に書き下したものである。ここで、外側のループのインデックスが i、内側のルー

プがjであるので、それぞれのインデックスに関わる粒子をそれぞれi粒子、j粒子と呼ぶことが多い。計算の詳細を追うと、まずi粒子とj粒子の距離を計算し、距離から力積を計算し、力積を運動量に書き戻している。ここで、i粒子の運動量も、j粒子ごとに和を取っているが、実際にはここでデータを書き戻す必要はない。一度、テンポラリ変数にj粒子からの力積の和を集めておき、j粒子のループの最後に和だけi粒子の運動量に書き戻した方が、メモリアクセスの面で効率的である。そのような修正をしたのがList 4である。この例では、この修正で5%ほど高速化される[4]。

List 3: LJ ポテンシャルの計算。何も工夫しない場合。

```
#include <stdlib.h>
//------------------------------------------------------------------------
const int N = 20000;
const double L = 10.0;
const int D = 3;
const int X = 0, Y = 1, Z = 2;
double q[N][D],p[N][D];
const double dt = 0.001;
//------------------------------------------------------------------------
double
myrand(void){
  return static_cast<double>(rand())/(static_cast<double>(RAND_MAX));
}
//------------------------------------------------------------------------
void
calcforce(void){
  for(int i=0;i<N-1;i++){
    for(int j=i+1;j<N;j++){
      const double dx = q[j][X] - q[i][X];
      const double dy = q[j][Y] - q[i][Y];
      const double dz = q[j][Z] - q[i][Z];
      const double r2 = (dx*dx + dy*dy + dz*dz);
      const double r6 = r2*r2*r2;
      const double df = (24.0*r6-48.0)/(r6*r6*r2)*dt;
      p[i][X] += df*dx;
      p[i][Y] += df*dy;
      p[i][Z] += df*dz;
      p[j][X] -= df*dx;
      p[j][Y] -= df*dy;
      p[j][Z] -= df*dz;
    }
  }
}
//------------------------------------------------------------------------
int
main(void){
  for(int i=0;i<N;i++){
    p[i][X] = 0.0;
    p[i][Y] = 0.0;
    p[i][Z] = 0.0;
    q[i][X] = L*myrand();
    q[i][Y] = L*myrand();
    q[i][Z] = L*myrand();
  }
  calcforce();
}
//------------------------------------------------------------------------
```

[4] このチューニングにより5%の速度向上、という効果をどう見るかは難しいところだが、簡単にできることで、プログラムによってはもっと効果がでる場合もあるし、特に可読性を損なうわけでもないので、これくらいはやっても良いかな、という気はする

List 4: LJ ポテンシャルの計算。レジスタを意識した場合。

```
void
calcforce(void){
  for(int i=0;i<N-1;i++){
    double pix = p[i][X];
    double piy = p[i][Y];
    double piz = p[i][Z];
    for(int j=i+1;j<N;j++){
      const double dx = q[j][X] - q[i][X];
      const double dy = q[j][Y] - q[i][Y];
      const double dz = q[j][Z] - q[i][Z];
      const double r2 = (dx*dx + dy*dy + dz*dz);
      const double r6 = r2*r2*r2;
      const double df = (24.0*r6-48.0)/(r6*r6*r2)*dt;
      pix += df*dx;
      piy += df*dy;
      piz += df*dz;
      p[j][X] -= df*dx;
      p[j][Y] -= df*dy;
      p[j][Z] -= df*dz;
    }
    p[i][X] = pix;
    p[i][Y] = piy;
    p[i][Z] = piz;
  }
}
```

4 CPU チューニング

4.1 パイプライン

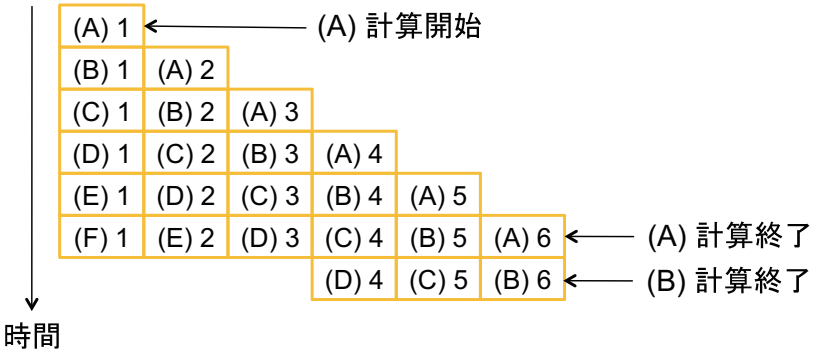
CPU にとって計算に必要なデータがほぼキャッシュに乗っている、もしくはボトルネックがメモリ転送ではなく計算待ちである、という状況になった後には、CPU の持つ計算資源をできるだけ効率的に使う CPU チューニングを行う。近年の CPU の演算器は高度にパイプライン化されており、CPU チューニングは「いかにパイプラインを埋めるか」がキーとなる。そのためにはまずパイプラインとは何かを知っておく必要があるため、以下で簡単に説明する。

パイプラインとは、計算をいくつかのステージにわけて流れ作業で行う仕組みである。たとえば、6 段のパイプラインを考える[5]。すると、一つの計算の実行を始めてから終わるまで 6 サイクルかかる。しかし、最初の計算の 2 ステージ目の作業をしている時、次の計算の 1 ステージ目の作業が同時に実行可能である。例えば、独立な足し算を連続で行う事を考える。

```
fadd fp2,fp0,fp1 (A)
fadd fp5,fp3,fp4 (B)
fadd fp8,fp6,fp7 (C)
....
```

それぞれを (A)(B)(C)··· と名前をつけ、(A) の計算の最初のステージを (A)1 などと表記すると、パイプラインでは以下のような作業が行われている。

[5] ここでは分かりやすさのためにパイプラインのステージがレイテンシと等しいとしているが、実際の仕組みは異なるので注意。

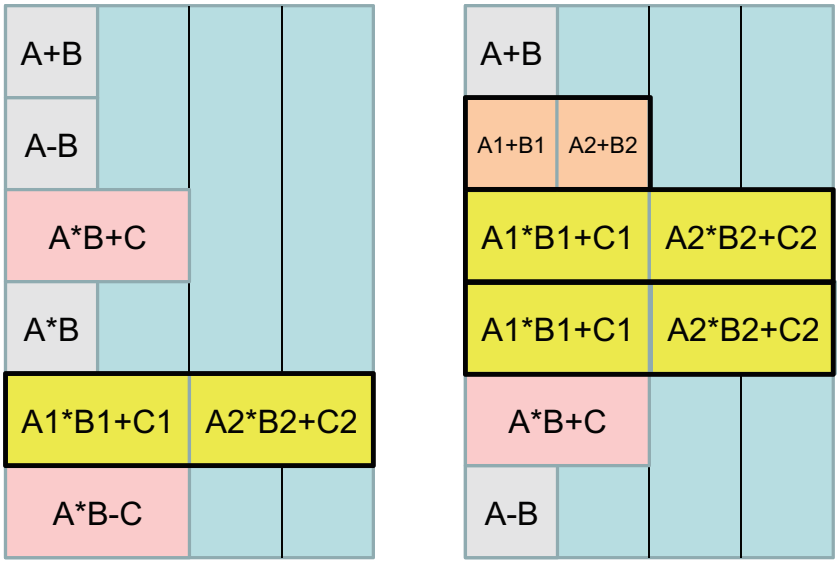


この例では、6 サイクルかかる計算を同時に 6 つ処理できている。従って、十分な数の独立な計算があれば、1 サイクルで 1 回の計算ができているように見える[6]。これを「スループット (Throughput) が 1 である」と表現する。計算機のピーク性能は、演算器がフル稼働した場合の値を指す。最近のアーキテクチャでは、パイプラインが 2 本あることが多いので、レイテンシが 6 の場合には、最低でも独立な計算が 12 個ないとピーク性能が出せないことになる。また、SIMD も重要である。SIMD とは Single Instruction Multiple Data の略で、一つの命令で複数の処理を行うことである。科学計算では、主に複数の浮動小数点演算を扱う命令を指す。そのうちの一つが積和命令である。積和命令とは、$A \times B + C$ のような計算を一度に行うような命令で、この命令一つで二つの浮動小数点演算と数える。さらに、複数の独立した乗算、加減算、積和命令を一度に行う SIMD 命令も存在する。パイプラインに流れる計算の間には依存関係があってはいけない。例えば

```
a = b + c
d = a * e
```

という計算は、前の計算が終わって a の値が確定しなければ次の計算を行うことができない。従って、レイテンシの分だけ次の計算が待たされることになり、それだけでピーク性能比の 1/6 まで性能が落ちる。

以上をまとめると、以下のような図になる。



つまり、幅 4、長さ 6 のベルトコンベアがあり、それぞれには、大きさ 1、2、4 の荷物を流せるが、同時に一つしか置けないという制約がある。ピーク性能比とはベルトコンベアの最大積載量 (4*6*2=48) に対してどれだけ荷物が載っているかの比であり、高い性能を出す、ということはなるべくおおきさ 4 の荷物

[6]実際には最初のレイテンシの分だけ遅くなるが、例えば 100 個の独立な計算が 106 サイクルで、1000 個ならば 1006 サイクルと、計算量が増えればほとんど計算量/サイクルの比は 1 に近づいて行く。

を隙間無く流すことに対応する。独立な計算を増やす方法としては、ループアンローリング、ソフトウェアパイプライニング、同時マルチスレッディング (Simultaneous Multithreading; SMT) などがある。

4.2 if 分岐削除

演算器の全てのパイプラインが埋まっている状態になってはじめて CPU はピーク性能を出せるのであった。逆に、パイプラインが埋まらなければ、それだけ性能が落ちることになる。この、パイプラインがスムーズに流れるのを阻害する要因のことをパイプラインハザードと呼ぶ。if 文による分岐などがその典型例である。分子動力学法では、力の及ぶ範囲にカットオフを入れることがほとんどである。従って、力の計算において、粒子間距離が相互作用距離よりも長いか短いか判定し、相互作用距離内の時のみ力の計算をする、とい処理が必要になる。List 3 に、適当にカットオフを入れたのが List. 5 である。この例では、カットオフ距離よりも粒子間距離が長い場合、単純に continue により次のペアの計算に飛んでいる。if 文はパイプラインハザードとなるが、特にジャンプを伴う分岐のコストは重い場合が多い。そこで、カットオフ距離よりも長い時にも計算を実行してしまい、運動量に書き戻す時に力積を 0 にクリアしてしまうようにしたのが List 6 である。このコードを、Intel Xeon 2.93GHz と IBM POWER6 3.5GHz で比較した結果を図 3 に示す。Xeon では、if 文で continue をしてしまった方が速い。しかし POWER では、continue 文がある場合に比べて、無駄な計算をしてでもジャンプ命令をなくした方が二倍近く速い。1 行書き換えるだけで 2 倍の速度向上を得る事ができる。これは IBM POWER のような RISC 系の石ではジャンプ命令のペナルティが大きい事と、Intel 系の石では投機的実行がうまく働く場合が多く、ジャンプ命令にあまりペナルティがないこと等による。理由はともかく、大事なことは **CPU チューニングは CPU に強く依存する**ということである。一見当たり前のようであるが、これは使うスパコンによって、ホットスポットのコードを変える必要がある、ということを意味する。メモリマネジメントが、(もちろん機種依存するところもあるものの) だいたいの計算機で効果的であるのに対して、CPU チューニングは使う CPU アーキテクチャに強く依存する。そのため、CPU チューニングを施したコードは、少なくともホットスポットに関しては複数のバージョンの面倒を並行して見る必要がある。HPC を志す以上、CPU チューニングは当然という考え方と、そこまでのチューニングは手間がかかり過ぎるという考え方、どちらの立場もあろう。いずれにせよ、CPU チューニングをするならば、計算機の種類ごとにコードを管理することになる、ということだけは意識しておいたほうが良い。

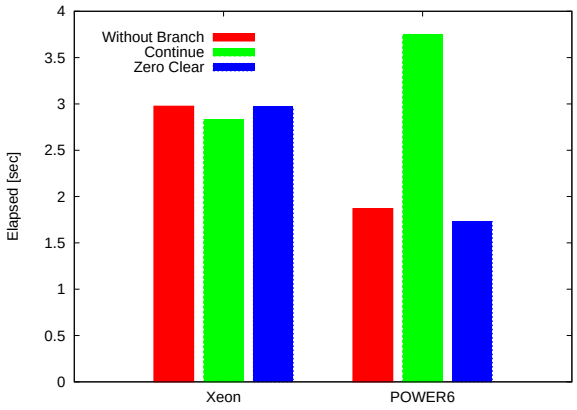


図 3: if 分岐削除の効果。if 文で continue をした場合 (continue) と、continue せずに力を計算し、後で力積をゼロにしてしまう方法 (zero clear) の比較。左側が Intel Xeon 2.93GHz、右側が IBM POWER6 3.5GHz。Xeon では無駄な計算をする分遅くなるが、POWER では分岐命令のペナルティが大きい分、無駄な計算があってもジャンプ命令を削った方が速い。

List 5: カットオフ付き LJ ポテンシャルの計算。if 文で continue をしている。

```
const double CUTOFF = L*L*0.9*0.9;
void
calcforce(void){
  for(int i=0;i<N-1;i++){
    for(int j=i+1;j<N;j++){
      const double dx = q[j][X] - q[i][X];
      const double dy = q[j][Y] - q[i][Y];
      const double dz = q[j][Z] - q[i][Z];
      const double r2 = (dx*dx + dy*dy + dz*dz);
      if (r2 > CUTOFF) continue;
      const double r6 = r2*r2*r2;
      double df = (24.0*r6-48.0)/(r6*r6*r2)*dt;
      p[i][X] += df*dx;
      p[i][Y] += df*dy;
      p[i][Z] += df*dz;
      p[j][X] -= df*dx;
      p[j][Y] -= df*dy;
      p[j][Z] -= df*dz;
    }
  }
}
```

List 6: カットオフ付き LJ ポテンシャルの計算。if 文で力積を 0 クリアしている。

```
const double CUTOFF = L*L*0.9*0.9;
void
calcforce(void){
  for(int i=0;i<N-1;i++){
    for(int j=i+1;j<N;j++){
      const double dx = q[j][X] - q[i][X];
      const double dy = q[j][Y] - q[i][Y];
      const double dz = q[j][Z] - q[i][Z];
      const double r2 = (dx*dx + dy*dy + dz*dz);
      const double r6 = r2*r2*r2;
      double df = (24.0*r6-48.0)/(r6*r6*r2)*dt;
      if (r2 > CUTOFF) df = 0.0;
      p[i][X] += df*dx;
      p[i][Y] += df*dy;
      p[i][Z] += df*dz;
      p[j][X] -= df*dx;
      p[j][Y] -= df*dy;
      p[j][Z] -= df*dz;
    }
  }
}
```

4.3 アセンブリの確認

パイプラインの項目で述べたように、現在の CPU アーキテクチャでは性能を出す為には SIMD 命令を使うのが必須となっている。SIMD ではないスカラ命令が実行されているということは、それだけで性能が半分に、さらに積和命令のあるアーキテクチャで積和命令が出ていない場合にはさらに性能が半分になってしまう。コンパイラによっては、コンパイル時メッセージにどこでどれくらい SIMD 命令を使えたかのメッセージを出してくれる場合もあるし、プロファイラを使っての解析も可能であるが、実際にコンパイラがどんな命令を出しているのか直接見るのが一番手っ取り早い。ここでは、SIMD や積和命令がちゃんと出ているかどうか、アセンブリを出力して確認する方法について説明する。

コンパイル時に-S オプションをつけると、ファイル名.s という名前のアセンブリファイルができる。しかし、プログラム全体のアセンブリは把握しづらいので、見たいところだけ抜粋したファイルを作り、その部分だけをアセンブリ出力させると見やすくて便利である。たとえば、一次元ベクトル **a**,**b**,**c** にたいして、

$$a_i = a_i \times b_i + c_i \tag{1}$$

という演算をすることを考えよう。ただし a_i は **a** の i 番目の要素である。その部分だけのソースを切り出すと List 7 のようになるだろう。

List 7: アセンブリを見るために切り出したソース。

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

アセンブリを見るだけであれば実行可能にする必要はないので、main 関数なども必要ない。この例における配列のサイズ N など、コンパイル時に必要な定数などだけ適宜補ってやれば良い。これを、たとえば MacPro で gcc でアセンブリを出力し、SIMD 命令が出ているか grep をかけてみよう。

```
$ g++ -O3 -S asm.cc
$ grep mulpd asm.s

$ grep mulsd asm.s
       mulsd (%rsi), %xmm0
       mulsd (%rsi,%rax,8), %xmm0
```

ここで、mulsd は乗算のスカラ命令、mulpd は SIMD 命令である。この例では SIMD 命令が一つも出ていない、つまりどんなにがんばってもピーク性能の半分までの性能しか出ない、ということがわかる。このソースを、別の Xeon マシンで、Intel コンパイラで同じことをしてみよう。

```
$ icpc -S asm.cc
$ grep mulsd asm.s
       mulpd (%rdi,%rax,8), %xmm1 #5.19
       mulpd 16(%rdi,%rax,8), %xmm3 #5.19
       mulpd 32(%rdi,%rax,8), %xmm5 #5.19
       mulpd 48(%rdi,%rax,8), %xmm7 #5.19
       mulpd (%rdi,%rax,8), %xmm0 #5.19
       mulpd 16(%rdi,%rax,8), %xmm1 #5.19
       mulpd 32(%rdi,%rax,8), %xmm2 #5.19
       mulpd 48(%rdi,%rax,8), %xmm3 #5.19

$ grep mulsd asm.s
       mulsd (%rsi), %xmm0 #5.19
       mulsd (%rsi,%rcx,8), %xmm0 #5.19
       mulsd (%rsi,%rax,8), %xmm0 #5.19
```

内容はともかく、「いっぱい mulpd が出力されているな」ということが分かれば良い。ちなみに同じ環境で gcc でコンパイルすると以下のようになる。

```
$ g++ -O3 -S asm.cc
$ grep mulpd asm.s
       mulpd %xmm2, %xmm0

$ grep mulsd asm.s
       mulsd (%rsi,%rax), %xmm0
```

一応 SIMD 命令が出ているが、その数が少ない事がわかるであろう。より詳細にアセンブリを見ると、インテルコンパイラはループアンロールとソフトウェアパイプライニングをしてなるべくパイプラインが埋まるように工夫している (らしい) のに対し、gcc では素直に二倍のループ展開をして SIMD 命令を使っていることがわかるが、ここではとりあえず「SIMD 命令が出ているか、出ていないか」だけがわかれば良い。

IBM POWER や、SPARC のように積和命令のあるアーキテクチャでは、積和命令が出ていないとそれだけで性能が半分になる。同じコードを、IBM の POWER6 マシンでコンパイルして、積和命令が出ているか確認してみよう。

```
$ g++ -S asm.cc
$ grep fmadd asm.s

$ grep -E "fmul|fadd" asm.s
       fmul 13,13,0
       fadd 0,13,0
```

`fmul` が乗算命令、`fadd` が加算命令、そして `fmadd` が積和命令である。デフォルトオプションでは積和命令が呼ばれておらず、素直に乗算と加算が行われていることがわかる。最適化オプションを上げて同じ事をやってみよう。

```
$ g++ -O3 -S asm.cc
$ grep -E "fmul|fadd" asm.s

$ grep fmadd asm.s
       fmadd 0,0,12,13
       fmadd 0,0,12,13
```

今度は逆に、乗算命令と加算命令がなくなり、積和命令のみになった。積和命令がさらに SIMD 化されているような場合は、SIMD 化された積和が出ている場合に比べ、SIMD 化されていない積和、もしくは SIMD 化されている乗算、加算、減算命令が出て半分、さらに SIMD 化されていない乗算や加減算が出ていたら演算器の 1/4 の能力しか使えていない、ということになる。一般に、SIMD 化されていないコードを SIMD 化するのは難しい。しかし、SIMD 化されているのかどうか、されているならどのくらいされているのかくらいはアセンブリを見ればすぐにわかる。ここで紹介したようにソースの見たいところだけを抜粋してアセンブリを出力するのはわりと役に立つので、覚えておいて損はない。

5 並列化で起きるトラブル

シングルノードや研究室のクラスタでの動作確認が済んでおり、本番環境でも小規模計算では動くのに、大規模計算ジョブが失敗した、ということは良くある。こういった大規模実行時に初めて現れる問題は、知らなければ事前の対策が難しい。以下、筆者が体験したトラブルを列挙する。

5.1 Flat-MPI か、ハイブリッドか

現代使われている並列計算機のほとんどは、CPU コアを複数持つノードを、さらに多数束ねた形となっており、ノード内は複数のコアがメモリを共有し、ノード間はメモリが分散している、階層的なメモリアーキテクチャを採用していることが多い。メモリ共有型の並列モデルとしてディレクティブベースの OpenMP、メモリ分散型の並列モデルとしてライブラリベースの MPI が広く使われており、事実上の標準となっている。ノードをまたぐ通信では MPI が必須となるため、超並列計算では、全て MPI で書く flat-MPI、ノード間通信には MPI を用いるがノード内並列には OpenMP を用いるハイブリッド計算のどちらかを選択することになる[7]。アプリケーションにも依るが、一般論としては flat MPI が可能であれば、hybrid よりも速い事が多い。しかし、MPI プロセスは OpenMP スレッドに比べてメモリ使用量が大きい。

東大物性研の SGI Altix ICE 8400EX の 1024 コアにおいて、flat MPI (1024 プロセス) とハイブリッド (128 プロセス× 8 スレッド) の計算を行った結果を表 1 に示す。詳細は省くが、それぞれの場合において、同じコアには全く同じ計算が割り当てられるようにしてある。Flat-MPI の方が実行速度は早いが、69TB だけ余計にメモリを使っていることがわかる。詳細は実装に依存するが、一般に並列数が大きくなればなるほど MPI の「プロセスあたり」のメモリ使用量が大きくなる傾向にある。従って、100 並列では大丈夫だった計算が、ウィークスケーリング (ノードあたりの計算規模を固定してノード数を増やす方法) であるにもかかわらず 1000 並列ではメモリ不足で失敗する、ということがおきる。その場合は環境変数を修正することで MPI の利用メモリを減らしたり、ハイブリッド化により MPI のプロセス数を減らすなどの工夫が必要となる。

並列化手法	実行時間 [s]	利用メモリ [TB]
Flat-MPI	249.36	268
Hybrid	257.09	199

表 1: Flat-MPI 並列 (1024 プロセス) と Hybrid 並列 (128 プロセス× 8 スレッド) の性能比較。カットオフ付き Lennard-Jones ポテンシャルの計算。直径を 1 として $800 \times 800 \times 1600$、密度 0.5 で 5 億 1 千万粒子を 1000 ステップ計算するのにかかった時間と利用実メモリ量。

5.2 MPI 利用における資源枯渇

また、MPI が利用する資源はメモリだけではない。MPI は裏で様々なバッファを利用するが、それらのデフォルトの大きさはシステムによってだいたいのアプリケーションにおいて問題なく性能がでる値に設定されているおり、ユーザは通常は意識する必要はない。しかし、アプリケーションによってはデフォルトの値では性能がでなかったり、大規模実行時にエラーを起こして止まってしまったりするため、環境変数の見直しをしたり、コードを書き換えたりしなければならないことがある。

特に多いのが、ノンブロッキング通信の多用によるトラブルである。通信時間の隠蔽のため、MPI_Isend などのノンブロッキング通信を使うことも多いだろう。しかし、一般にノンブロッキング通信はブロッキング通信よりも多くのメモリ、多くの種類のバッファを使うため、大規模並列時に問題を起こしやすい。例えば小規模並列では問題なかったプログラムを大規模並列実行した時に「MPI_REQUEST_MAX が足りない」といったエラーメッセージとともにジョブが失敗することがある。MPI_REQUEST_MAX は同時に行うことができるノンブロッキング通信の最大数であり、これが不足する場合には、通信を小分けにするか、一部をブロッキング通信に置き換えるなどの工夫が必要となる。また、REQUEST_MAX 以外にも「MPI_○○○_MAX が足りない」というタイプのエラーはよく出現する。その際は対応する環境変数

[7] これ以外にも、ノードに分散するメモリを仮想的にグローバルなメモリ空間として使える PGAS(Partitioned Global Address Space) という並列パラダイムもあるが、筆者は良く知らない。

を上げる必要があるが、その背景にはメモリ不足が疑われるため、ノードあたりのプロセス数を減らしたり、問題サイズを小さくしたりする必要がある。

また、自分はブロッキング通信を行っているつもりでも、システムによっては性能を上げる為に裏でノンブロッキング通信が行われる場合もある。例えば送受信サイズによって、ノンブロッキング、ブロッキング通信が切り替わったりする。ブロッキング通信しか行っていないのにノンブロッキング通信特有のエラーが出た場合は、ノンブロッキング通信とブロッキング通信を切り替えるサイズを変更するなどして対応する必要がある。

5.3 プロダクトランに向けて

シングルノードでのチューニングも十分に済み、大規模並列でも満足できる性能が出たとして、それでもまだ科学論文を書く事はできない。分子動力学法であれば、温度制御や圧力制御、境界条件なども正しく並列化されていなければならない。さらに重要なのは観測・解析ルーチンである。大規模計算を実行中、その全ての情報をディスクに出力していくのは現実的ではないため、一般には物理量を粗視化しながら出力する操作が必要になる。これは、観測ルーチンも並列化しなければならないことを意味する。圧力や温度といったスカラ量ならまだしも、密度分布や気泡検出などを並列化しようとすると、それだけでかなりの作業量となる。例えば気泡検出は、もっとも単純には空間を小さく分割して局所密度がある閾値以下であれば気相と判断するなどの手法を取るが、気泡サイズ分布を得るためには、クラスタリングの並列化を行う必要がある。粗視化した密度分布だけ出力しておいてクラスタリングをポスト処理で行うべきか、クラスタリングまで実行中に行ってしまうべきかは場合による。もし並列化を志すならば、完成した並列化プログラムをプロダクトラン用に改良する作業量は、ベンチマークが大規模でちゃんとスケールするコードを作る作業量と同じかそれ以上かかると見ておいた方が良い[8]。

6 ソースの公開について

6.1 ソース公開のススメ

無事にプログラムの開発が進み、ある程度の成果が出始めたら、今度はそれを他の人にも使ってもらいたい、というフェーズになるであろう。そのとき、オープンソースソフトウェアとして公開することを強く勧めたい。誓っても良いが、大多数の人がかかわって、それなりに成長・成功したコードを、「さぁ、ソースを公開しましょう」といってもうまくいかない。権利関係がややこしくなるし、ノウハウの流出を警戒して反対する人も出てくるだろうし、そもそも人に見せるようにかかれていないソースコードを公開することはプログラマーにとって極めて恥ずかしいことである。しかし、これも誓っても良いが、アカデミアから発信するソフトウェアでソースが公開されていなければ、そのソフトウェアの寿命は多くの場合プロジェクトの予算と運命をともにする。こうして予算がつぎ込まれて作られたは良いが、プロジェクトの終焉とともに誰にも使われずに消えていくソフトウェアが量産されてしまうことになる。このような状況を防ぐためには、ソフトウェアを最初からオープンソースプロジェクトとして立ち上げるしかない。協力者には、「このソフトウェアはソースを公開します」と宣言し、それでも協力してくれる人のみをメンバーとしてソフトウェアを開発するのである。

ソフトウェアの公開場所は、できれば大学やプロジェクトのウェブサイトではないほうが良い。開発者が異動したり、プロジェクトが終了後にウェブサイトが閉鎖 (かそれに近い状況) になって、ソフトウェアの公開場所がころころ変わると、ユーザにとってソフトが公開終了になったと勘違いされたりして、成果

[8] 要するにベンチマークができたところから論文を書くまでかなり時間がかかる、ということ。特に任期があるような若い人は、そのベンチマークコードをプロダクトランコードまで持って行く期間も織り込んで研究計画を立てなければならない。

の公開の効果が薄れてしまう。また、ソースやソフトをダウンロードする際に所属や名前、メールアドレスを要求するのもソフトウェアの普及という観点からは望ましくない。

このような目的には、SourceForge(ソースフォージ) と呼ばれるソースコードリポジトリを利用するのが良い。SourceForge はオープンソースのソフトウェアを開発、公開するためのサイトで、ソフトウェアがオープンソースライセンスであれば誰でも利用可能で、利用料金は無料である。SourceForge には、本家 (SourceForge.net, http://sourceforge.net/) と、その日本語版 (SouceForge.jp, http://sourceforge.jp/) がある。基本的な機能はかわらないが、個人的な意見だが日本語版のほうが使いやすいので、そちらを利用するのがお勧めである。SourceForge では、ソフトウェアのウェブサイト公開、CVS や Subversion、Git といったバージョン管理システムのサーバ機能、ダウンロードカウンタ、掲示板設置など、プログラムの開発、公開に必要な機能が一通りそろっている。これらは自前のサーバに用意することもさほど難しくはないが、その場合にはプログラム開発者の異動によって URL が変わらないように、ドメインを別途取得しておくなどの措置が必要になる。

ソフトウェアを公開する際には、そのソフトウェアにライセンスを設定しておく必要がある。ライセンスというのは、「このソフトはどういう用途に使ってよいか、このソフトウェアを使って得られた成果物の権利は誰に帰属するか」といった条件を定めたもので、後々のトラブルを防ぐためにはきちんと設定することが望ましい。オープンソースソフトウェアで広く使われているライセンスは、大きく分けて「GNU General Public License (GPL)」と「BSD ライセンス」の二つであり、それらを用途にあわせて修正して使うのが便利である。

「GNU GPL」は、そのソフトウェアのソースを含む二次的著作物にも全て GPL が適用されることを要求する。たとえ元のソフトウェアが無料配布されていても、二次的著作物を有償で販売することは自由である。しかし、その二次的著作物のソースは完全にアクセス可能な形で公開されなければならない、というのが GPL の要求である。GPL のソースコードを一部にでも含んだソフトウェアには GPL が適用される。この意味で、GPL は「感染するライセンス」と呼ばれることもある[9]。ちなみに再配布しなければ、改変したものを公開する義務は負わないため、このライセンスでソフトウェアを公開しても、それが産業界で使われないということは意味しない。

もうひとつのライセンス、「BSD ライセンス (Berkeley Software Distribution License)」は、GPL とは対照的に、一言で言えば「なんでもあり」のライセンスである。BSD ライセンスにはいくつかの派生ライセンスがあるが、良く使われているのは三条項 BSD ライセンスと呼ばれるもので、「無保証」であることの明記、「著作者」の表示、そしてライセンス条文そのものの表示を義務付けるものである。BSD ライセンスで公開されているソフトウェアを、少し修正して販売することも自由であり、その修正版のソースコードを公開する義務も負わない。商用化しやすいため、産業界への展開を考えているなら、こちらの方を選択しておけば問題が少ないと思われる。

また、アカデミックでよく利用されるのが、いわゆる「Cite me ライセンス」と呼ばれる形態で、無保証と著作者表示のほか、そのソフトウェアを使って何か成果を挙げたら、対応する論文を引用することや、ソフトウェアの配布元を表示することを義務付けるライセンスである。論文引用を義務付けることで、どのくらい広く使われているかが Web of Science などで検索しやすくなる、というメリットがある。渡辺が開発している分子動力学法コードも、基本的には BSD ライセンス+Cite me ライセンスとして配布している。

6.2 ソース公開の実例

実際に、Sourceforge を使ったプロジェクトの公開がどのようになるか、いくつかのソフトウェアを例に上げて見てみよう。まず、渡辺が開発している分子動力学法コード (MDACP) については Sourceforge の本家で公開している。プロジェクトを登録するには、登録を申請し承認される必要があるが、原則としてプ

[9] このライセンスの提唱者、リチャード・ストールマンは「感染」という言葉を好まないようである。

ロジェクトは承認されるようである。そのプロジェクトの URL は `http://mdacp.sourceforge.net/` と、プロジェクト名+ `sourceforge.net` というアドレスになる。このページはプロジェクト管理者が自由に編集することができる。一般のユーザにはこのページを公開する。また、プロジェクトの管理ページも自動で作られる。その名前は `http://sourceforge.net/projects/mdacp/` と、`sourceforge.net/projects/` + プロジェクト名となる。ここでダウンロードやバージョン管理、(もし運営していれば) メーリングリストの過去ログなどが閲覧できる。

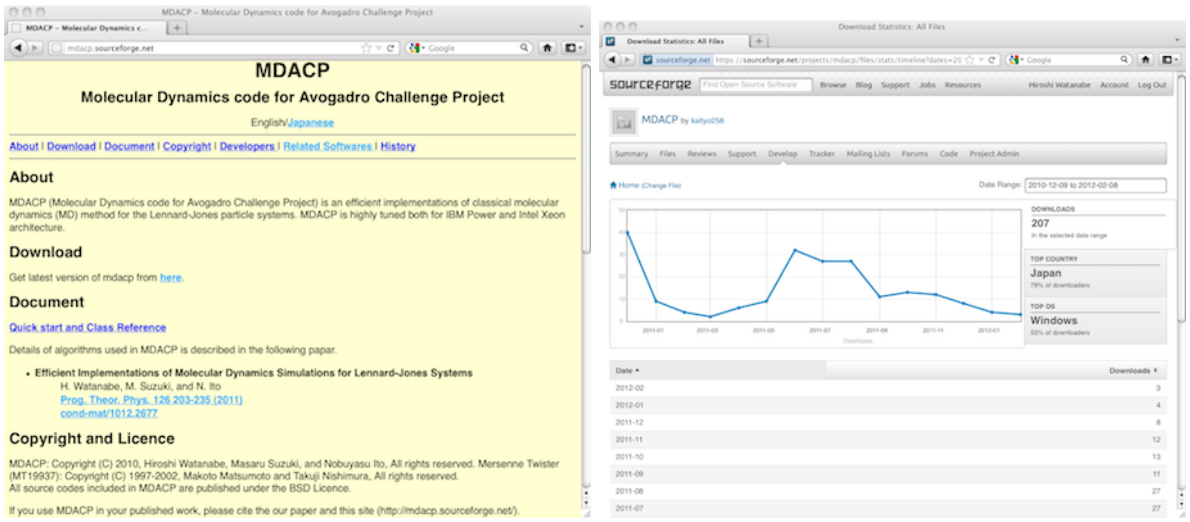


図 4: (左) MDACP プロジェクトのホームページ@SF 本家。ここは好きなように編集できる。(右) プロジェクトの管理ページ。ダウンロード数や履歴なども閲覧できる。

今度は、Sourceforge.jp の例を挙げよう。量子計算シミュレータ QCAD など、渡辺はここでいくつかプロジェクトを公開している。QCAD であれば、プロジェクトのホームページは `http://qcad.sourceforge.jp/` と、プロジェクト名+`sourceforge.jp` となる。プロジェクト管理ページは `http://sourceforge.jp/projects/qcad/` と `sourceforge.jp/projects/` + プロジェクト名となるのも本家と同様である。

図 5: (左) QCAD プロジェクトのホームページ@SF.jp。ここは好きなように編集できる。(右) プロジェクトの管理ページ。ダウンロード数や履歴なども閲覧できる。

7　おわりに

以上、プログラムのチューニング、並列化、そして公開について簡単にまとめた。一般論として、速度を考えずに組まれたプログラムは、少しの修正で大きく速度が向上するが、ある程度チューニングが進んでくると、速度を向上させるのが難しく、向上しても数%、という場合が多い。チューニングをどこまでやるべきかは問題に依存するが、とりあえず遅いのか速いのか、遅いとしたらなぜ遅いのかくらいは調べた方がよい。その上で最低限の目標速度を決め、その速度を達成した後は速度向上は目指しつつも、科学的成果をあげることに注力したほうが良いと思う。

なお、分子動力学法の並列化コードのチューニングの詳細を論文にまとめた [2]。メモリマネジメントや条件分岐削除、割り算削除なども具体的に書いたので興味のある人は参考にしてほしい。この原稿を書くにあたり、CCS HPC サマーセミナー 2007 の講義資料、特に高橋先生の資料を参考にさせていただいた [3]。また、基本的なチューニングについては、理化学研究所情報基盤センターの講習会テキストも詳しい [4]。

参考文献

[1] http://www.slideshare.net/kaityo256/tuning-etc

[2] H. Watanabe, M. Suzuki, and N. Ito, Prog. Theor. Phys. **126**, 203-235 (2011).

[3] http://www.ccs.tsukuba.ac.jp/workshop/HPCseminar/2007/

[4] http://accc.riken.jp/HPC/training.html