数値計算の最適化について

Optimization of Numerical Calculations

伊藤 誠

1 はじめに

パーソナルコンピュータ (以後パソコン) からスーパーコンピュータ (以後スパコン) まで、計算機の形態には様々なものがある。歴史的には

1. 単一 CPU(単一コア[1]) 共有メモリ
2. 複数 CPU(単一コア) 分散メモリ
3. 単一 CPU(マルチコア) 共有メモリ
4. 複数 CPU(マルチコア) 共有メモリ +分散メモリ

と言った感じであろうか。最近のパソコンは 3 番目の形態であり、スパコンは 4 番目の形態にあたる。パソコンは日常業務に利用される身近な存在であるが、スパコンは昨年 (2011 年) コンピュータの計算速度 TOP500 で世界一になった日本の計算機「 京」[1] など、何やら遠い存在のように感じられる。

しかし近年のパソコンの性能は、ムーアの法則[2]にしたがって IC の集積度が向上し高速化されてきており、従来のスパコンを凌駕する計算性能がある。ただしそれにともなう消費電力の増加や発熱量の増大のため、動作周波数 (クロック周波数) を増加させるよりも有り余るトランジスタを使い、マルチコア化・多機能化されてきている [2]。パソコンが廉価になったため、複数のパソコンをネットワークで結合し、複数 CPU での計算も可能になった (「 京」には遠く及ばないが一応 4 番目の形態)。今まで高性能な計算機を利用して行ってきた数値計算でも、ちょっとした計算であれば、数値計算・計算結果の解析等はパソコンがあれば十分可能である。

ただしいくらパソコンが高速化したとしても、マルチコア化して並列計算が可能となったとしても、CPU 内でのメモリアクセスやベクトル化に関する計算機の基本、、並列化手法等数値計算を行う上で考慮すべき点はいくつかあげることができる。高速化したパソコンの性能を十分引き出すため、将来スパコンを用いた計算を行うため、数値計算の最適化に関する基本的な技術・知識を身につけることは重要である。幸い手近なパソコンを用いてこれらの技術・知識を身につけることは十分可能であるので、ここで簡単に数値計算の最適化に関する基礎をまとめておくことにする。

[1]CPU 内の演算部分をコアと言う

[2]「 IC(Integrated Circuit:集積回路) の集積度 (単位面積あたりのトランジスタ数) は 18ヶ月ごとに倍になる」というもの。

2　様々な計算処理

2.1　逐次処理とパイプライン処理

CPU による 1 つの計算処理 (命令処理) は、簡単には次のようなステージから構成されている。

- メモリからの命令読み出し (フェッチ:fetch)
- 命令の解釈 (デコード :decode)
- データの取得 (オペランド operand)
- 演算の実効 (演算:excute)
- 結果の格納 (ライトバック:write back)

1 命令はこれら複数のステージで実行されるわけだが、逐次処理では 1 命令がこれらの複数のステージの処理を終えた後、次の命令が実行される。つまり 1 命令単位で逐次実行されるわけである。一方パイプライン処理では 1 つ目の命令が decode している間に、次の命令を fetch する、、、といった具合に、1 命令の実行が終わる前に次の命令の処理が行われる。これが長く続くと 1 ステージ間隔で 1 命令が実行されることになり高速化が図られる。

近年の CPU にはパイプライン処理に加え、(数値計算にはあまり関係ないが) スーパースカラ実行、Out-of-Order 実行、分岐予測など高速処理のための様々な手法が用いられている [2]

2.2　ベクトル化処理

数値計算は単純なループ処理が多く用いられる。数値計算でもパイプライン処理が行われるわけであるが、ループ処理では一般に 1 つの命令で複数のデータを処理する場合が多い。つまり decode は 1 度だけで operand は複数といった処理が主流となる。また科学技術計算では浮動小数点数の演算が主要演算であるが、整数演算とは異なり、上述の excute の部分は単一のステージで処理することができずさらに細かく複数のステージに分かれることになる。したがってその部分をさらにパイプライン化することが考えられる。これに基づいて考えられたのが、ベクトル化であり、そのためのプロセッサがベクトルプロセッサである (実際はこれほど単純ではない)[3][4]。

パソコンで用いられている CPU はよく知られているようにスカラープロセッサであり、ベクトル処理に特化したプロセッサではない。しかし SIMD(Single Instruction Multiple Data) レジスタを用い、SSE(Streaming SIMD Extention) 命令によるベクトル化を行うことができる。これにより 1 つの excute 処理単位で 1 命令を実行することが可能になり更なる高速処理が可能となる。

2.3　並列化処理

近年のパソコンの CPU はマルチコア化されてきている。単純に考えれば、コア数が増加し、演算処理を複数のコアで分散すれば、コアの並列度がました分計算速度も向上するように思われる。しかし並列で処理する場合様々な問題点があげられる。例えば

- 複数のコアで処理するためのプログラミング技術
- ループ内の計算依存性への対処
- 複数のコア間相互で計算の同期をとるためのオーバーヘッド
- 処理を複数のコアに振り分けるための OS(オペレーティングシステム) のオーバーヘッド

など、シングルプロセッサでは想像できなかった困難さがある。しかし諸般の問題点はあるにしても数値計算を行う上で避けて通ることはできない。

またマルチ CPU・分散メモリのシステムになると、上記に加え、例えば

- 分散メモリ間で処理を一旦統合し、同期をとるためデータ転送をするための通信のオーバーヘッド

などの問題も生じてくる。

幸いなことにマルチコア CPU での「プログラミング技術」については、並列計算可能なプログラミング言語対応のコンパイラさえあれば、個人で手軽に入手可能なパソコンで十分実践可能である。メモリやディスクといったストレージも安価になっており、個人レベルで比較的大規模な数値計算を行うことが可能である。

マルチ CPU については複数のパソコンをネットワークで結合するとか、パソコンのゲームで重用されるグラフィックスボード[3]を多数搭載したパソコンを用いることで高速な数値計算が可能である。しかし環境を個人で構築するには、ハードウェア・ソフトウェア的に敷居がやや高い。

以下ではプログラミング技術について簡単にまとめることにする。

3　具体例

ここで「具体例」と言ってもある特定のハードウェア、ソフトウエアに特化した例をあげるわけではない。あくまでも話を進める上での「例」である。したがって特定の環境における定量的な解析を行うことはしない。

[3]グラフィックスボードに搭載されている GPU(Graphic Proccessing Unit) は単純な計算しかできないが、その性能は汎用 CPU の数 10 倍から 100 倍程度あり、近年 GPGPU[2](Genreal Purpouse GPU) として並列数値計算用として流用されている。GPGPU を用いた東京工業大学の「つばめ」は数千万円の費用で TOP500 の上位を占める。ちなみに「京」はその建設費だけで 1000 億円以上である。

3.1　ハードウェアとコンパイラ

まず容易に構築可能な計算機システム (ハードウェア) として、パソコンで用いられる単一 CPU、マルチコアを考える。単一 CPU であるので、複数のコアでメモリは共有であり、並列処理の場合はデータ転送について考慮しなくてよい。しかしメモリアクセスについては留意が必要である。

コンパイラに関しては、数値計算でよく用いられるプログラミング言語 Fortran を考える。マルチコアによる並列計算では OpenMP[5] が用いられる。OpenMP ではソースプログラムは通常の Fortran の文法で記述すればよいが、並列計算のためにディレクティブと呼ばれる指示文を Fortran のコメント行としてソースプログラムに記述する手法がとられる。そのため OpenMP に対応していないコンパイラで処理してもそのまま実行することができる。OpenMP を利用できるコンパイラとして、GNU のコンパイラである gfortran や Intel Fortran コンパイラなどがある。

マルチ CPU での計算では MPI[6](Message Passing Interface) や PVM(Parallel Virtual Machine) などのライブラリを利用するのが一般的であるが、ここでは取り上げない。

3.2　プログラム

数値計算において最も時間を要するのはループ (Fortran では do ループ) であるが、CPU のパイプライン化はすでに一般的なものであるので、ここではメモリアクセス、ベクトル化、並列化について考える。

1 重ループに関しては、ループ内の演算に依存関係がなければベクトル化も、並列化も容易であるのでここでは考えない。メモリアクセスについては次の 2 重ループで考察する。

2 重ループの簡単な計算例として、ベクトルの一次変換を考える。

$$\mathbf{y} = A\mathbf{x} \tag{1}$$

成分で書けば

$$y_i = \sum_{j=1}^{n} = a_{ij}x_j \quad (i = 1 \sim n) \tag{2}$$

であるから、Fortran プログラムとしては上記の式をそのまま記述すれば

・プログラム (1)

```
do i = 1, n
  y(i) = 0.0
  do j = 1, n
    y(i) = y(i) + a(i,j)*x(j)
  end do
end do
```

となる。しかしよく知られているように Fortran の場合配列 `a(i,j)` はメモリ内で前の添字に関して連続に配置されるので、上記のプログラムでは `j` に関するループで `a(i,j)` へ

のアクセスがメモリ内で飛び飛びになり、効率が著しく低下する (メモリに連続アクセスする場合の数 10~100 倍程度の計算速度の差がある)。この様な場合コンパイラは自動的に i、j に関するループの順番を入れ替え、添字 i に関して連続してアクセスするように最適化してくれる。しかしプログラマの手で明示的に

・プログラム (2)

```
do k = 1, n
  y(k) = 0.0
end do
do j = 1, n
  do i = 1, n
    y(i) = y(i) + a(i,j)*x(j)
  end do
end do
```

とすることも可能である。あるいは A の転置行列 A^T の成分を at(j,i)=a(i,j) として

・プログラム (3)

```
do i = 1, n
  y(i) = 0.0
  do j = 1, n
    y(i) = y(i) + at(j,i)*x(j)
  end do
end do
```

とすることも可能である。プログラム (2)、(3) は最内ループに対してベクトル化可能であり効率のよいプログラムである[4]。

次に並列化について考える。OpenMP による並列化の記述方法は例えば

・プログラム (4)

```
program hello
  !$use omp_lib
  !$OMP parallel
    do ...
      do ...
      end do
    end do
  !$OMP end parallel
end program hello
```

[4](1) のプログラムもコンパイラの最適化により自動的にループの順番を入れ替えた場合、ベクトル化は可能である

である。ここで!で始まる行は Fortran のコメント行で、!$OMP で始まる行が OpenMP のディレクティブ (指示文) である。いくつのコアで do ループを並列処理するのかは、実行時ルーチンで指定するかあるいは環境変数で指定する。

上記のプログラムでは!$OMP parallel と!$OMP end parallel の間が並列処理の対象となる。単純に考えればコアの数に反比例して計算時間は短くなり高速処理が出きるようになる。

しかし計算の同期とループ内の計算依存性は考慮する必要がある。依存性がなければ並列化可能である。ただ 2 重ループ以上の場合、並列化の対象となるループは最外ループに対してであるので、上述のプログラムの場合並列化可能なプログラムは (1) と (3) であり、(2) のプログラムは (4) の形式のままでは並列化することができない。また (1) のプログラムは最外ループを並列化すると、ループの順番を入れ替えることができなくなり最内ループはベクトル化できなくなる。この場合使用するコアに反比例して計算時間は短くなるが、ベクトル化の効率の方が勝るので数個 (2~4 個) のコアでは計算時間で単一コアのベクトル化に劣る。数十個ないしは数百個のコアで並列化すればよいのであろうが、一つのプロセスを並列化するよりも、ベクトル化した複数のプロセスをコアの数だけ用意して並列実行した方が効率がよくなってしまう。並列化することによってかえって計算機資源の無駄遣いになり本末転倒な結果となってしまう。その点 (3) のプログラムは並列化、メモリアクセス、ベクトル化の点で最も最適化されたものとなっている[5]。また並列化と同時に最内ループのベクトル化が行われるかどうかはコンパイラ依存であり、並列化の時点で人間による関与 (ベクトル化指示行) が必要になる場合がある。マルチコアで単に並列化すれば最適化・高速化が図られるわけではない。

最後にメモリアクセスについて言及しておく。連続したメモリにアクセスすれば効率はよくなるが、実際のメモリアクセスはビット単位ではなく、メモリを構成している格子 (バンク) の行単位 (通常は K ビット単位) で行われる [2]。配列が連続したメモリに配置されていたとしても、同一格子内の別のメモリにアクセスする必要がある場合、一旦バンク内の行からセンスアンプに読み込まれたデータ (配列) を破棄して、別な行からデータを読み書きする必要がある。さらにメモリアクセスクロックは、CPU の動作クロックに比してかなり遅いため、メモリは通常複数のチャネル (複数枚のメモリモジュール) から構成されクロック数の縮めるような工夫がされている。どの配列がどのメモリモジュールのどのバンクに配置されているかを考えてプログラムを作成する必要がある。アルゴリズムだけ考えていても最適化されるとは限らないのである[6]。メモリ構成を含めたハードウェアアーキテクチャも考慮してプログラムを作成する必要がある。実際汎用コンパイラである gfortran と Intel Fortran コンパイラとの差は歴然である (数倍の計算時間の差がある)。ハードウェアアーキテクチャに最適化された CPU メーカによるコンパイラの自動ベクトル化や自動並列化に頼るべきなのかもしれない。しかし最終的には人の手によるチューニングが必要であろう。

ここでは触れてないがマルチ CPU・分散メモリシステムで計算する場合、分散されたメモリ間でデータを相互に同期を取る必要が出てくる場合がある。その場合個々の CPU・メモリがどのように結合されているかを考慮し、データ転送の順番 (どの CPU からどの順に

[5]ただし転置行列を用いるため、(2) 式の数学的記述とは相容れない。

[6]メインメモリ以外にキャッシュメモリも考慮する必要があるかもしれないが、数値計算で扱う配列は一般にキャッシュメモリに収まりきる大きさではない。

転送するのが速いか。) や転送バスの構造等を考えなければならないであろう。

4　結語

大学に設置された大型計算機を利用していた時代は、計算をするには費用がかかった。与えられた予算の気にしながら計算をしていた。プログラムのリストを印刷するのに大型計算機にジョブを依頼したり、印刷結果を受け取りに計算機センターまでわざわざ行った。パソコンが次第に普及して行く中、パソコンでプログラムを入力・実行することが出来るようになり、プログラム作成もずいぶん楽になった。しかし、デバック行を一行 (例えば `write` 文を一行) 加えて再コンパイルするのに数分を要した。当時は世界中にスパコンと言われる計算機が 10 数台しかなかったため、実際の計算には 1 週間待たされることもあった。

現在では当時のスパコンの性能を凌駕するパソコンを個人で 1 台 (あるいは数台) 保有でき、様々な方向から試行錯誤してプログラムの最適化を実践できる時代へと変わってきた。世界最高のスパコンといえどもその多くは一般の PC で使われている CPU・GPU の寄せ集めであることが多い。スパコンの性能を余すことなく引き出すための前段階の環境は我々の身近にある。

参考文献

[1]「 究極の Linux 機!スパコン世界一「 京」の全貌」宇野・ 星屋 (日経 Linux 2011 年 10 月号)
[2]「 プロセッサを支える技術」Hisa Ando 著 (2011 年技術評論社)
[3]「 スーパーコンピュータ」日本物理学会編 (1985 年培風館)
[4]「 スーパーコンピュータ科学技術計算への適用」村田・ 小国・ 唐木 (1985 年丸善)
[5]「 OpenMP による並列プログラミングと数値計算」牛島 (2006 年丸善)
[6]「 MPI 並列プログラミング」パチェコ (2001 年培風館)

Abstract

This note was summarized when optimizing numerical calculations. Under the personal environment supposing PC which recent years formed into the multi-core, the important points about fundamentals of vectorization, parallelization, memory access, etc. were arranged for the double loop of simple matrix calculation for the example.
Key words: Numerical Calculation,Optimization,vectorization,parallelization
キーワード :数値計算、最適化、ベクト ル化、並列化