高性能プログラミング（Ｉ）入門編

片桐　孝洋

東京大学情報基盤センター　特任准教授

１．はじめに

本稿は、高性能プログラミングを始める場合の入門的な解説書を目指して執筆した。特に、コンパイラが行う最適化手法について、どのような手法や技術があるか説明する。コンパイラが行う最適化についてユーザが知る必要は無い思われる方もいるかも知れない。しかし、実用コードの多くはコンパイラが解析するにはあまりに複雑な書き方がされており、自動最適化は限界がある。ユーザがコンパイラの理解しやすい書法を習得するのが望ましいが、コンパイラの最適化手法を知らないと無理である。したがって、高性能化のためコンパイラの最適化についてユーザが少しでも理解することが、ユーザのアプリケーション高速化への近道と思う。

本稿の想定事例は、ユーザのスキルに応じ以下のとおりである：（１）コンパイラが最適化しやすいコードに書きなおす；（２）コンパイラが行えない最適化を自らコーディングする；（３）コンパイラが最適化を行いやすい新実装方式を開発する。つまり、ユーザ自らが高速化できる知識をつけてもらうことを意図した。

本稿を執筆するにあたり、以下の点を注意した。

- コンピュータサイエンス分野の定式化や表記法を利用せずに直観的な説明を心がける
- できるだけ数値計算分野の実例を記載する

計算科学の研究者にとってコンパイラ分野での厳密な定式化は興味ないであろうが、コンパイラが行う最適化の理解は高速化のために致し方がないと判断されるに違いない。また、数値計算シミュレーションコードを持つユーザを想定しているため、数値計算の実例が有益であると判断した。現在の初心者ユーザの多くはＣ言語を利用していると予想されるため、本稿でのプログラム解説にはＣ言語を用いた。Fortran ユーザにおいても、基本的には言語に依存しない概念を解説しているので本稿が活用できると思う。

２．高速化の前提

高性能プログラミングを行うためには、多かれ少なかれ実行する計算機ハードウエアの前提が必要となる。本稿における計算機ハードウエアの前提は、大きく分けて以下の２つである。

（１）　メモリ階層構造

（２）　並列計算機

第一の前提として、計算機が有するメモリに図 1 の階層構造があることを前提とする。図 1 では、小容量・高速なメモリが複数ある。これら小容量・高速メモリを活用できれば、高性能なプログラムが作成できる。

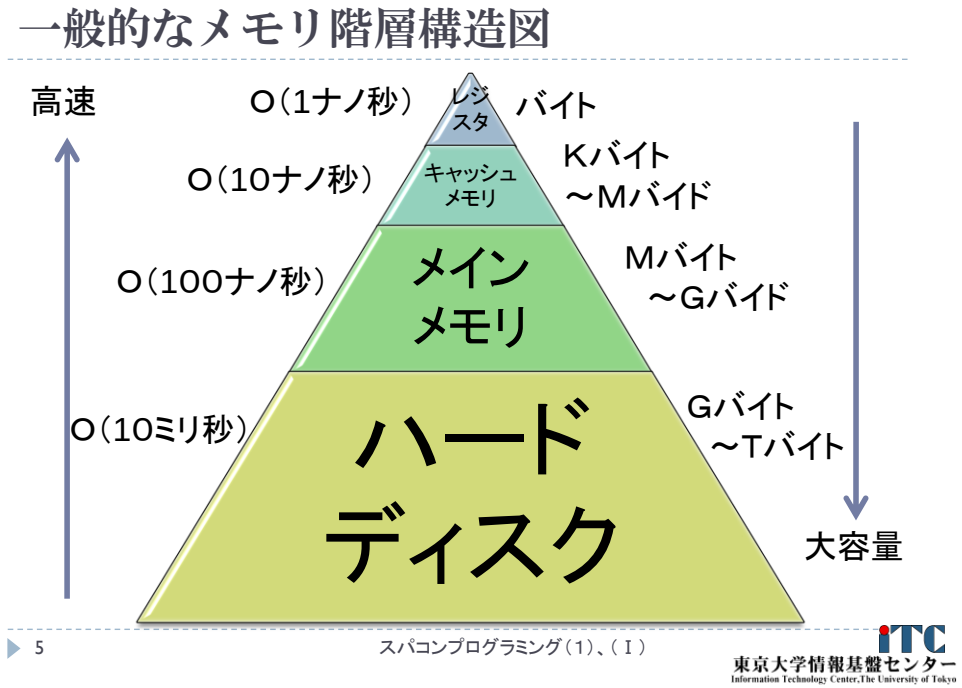


図 1　メモリ階層構造

配列について、ループ中に記載される式に現れるループ変数でアクセスされる「範囲」が大きいと、上位にある「キャッシュメモリ」からデータが追い出されてしまう。その配列のアクセスのたびに、メインメモリからデータを取ってこなくてはならなくなる。データがメインメモリから CPU に送られる速度は、キャッシュメモリからに比べざっと 10 倍は遅い。その間、CPU は計算が一切できない状況となる。したがって CPU の利用率が低下し、プログラム全体が低速化する。

逆に、ループの書き方やアルゴリズムの変更で、配列のアクセス範囲を局所化できたとしよう。この場合、高速なキャッシュメモリ上にデータがあるので、配列のデータアクセス時間がメインメモリからに比べ高速化できる。そればかりか、データをあらかじめ CPU 内に取り込むハードウエアやソフトウエア機構がうまく動作し、データを読み込む時間が完全に隠れてしまうことさえもある。この場合、CPU の理論的な演算性能の上限に近づく。極めて効率の良いプログラムが作成可能となる。

以上長々と説明したが、現在の CPU における高速プログラミングの要点は以下の通りである。

- 配列データのアクセス範囲をいかに縮小するか
- 高速なメモリに乗ったデータを何回も使いまわせるか

上記の観点での、新しいプログラミングスタイルが要求される。

第二の前提として、並列計算機の普及がある。CPU 自体が複数、もしくは、CPU 内にある演算要素（「演算ユニット」や「コア」とばれるもの）が複数あるという状況が普通となった。それらを使い切る並列性が、書かれているプログラムの「字面」としてあるかどうかが重要な論点となってきた。したがって、以下の事項が重要となっている。

- ループに存在する並列性をコンパイラが適切に解析できるか

並列性の明確な記述は、各社コンパイラが用意しているコマンド（「ディレクティブ」という）を用いたり、並列化のための標準ディレクティブである OpenMP を用いて記述ができる。しかしこれらのディレクティブが記述できるかどうかは、ループ中に書かれた式から生じるデータ依存関係による。自動並列化コンパイラを使用している場合、並列化可能かどうかは、このプログラム上の字面からのデータ依存関係をもとに判定される。しかし、本質的な並列性がある場合でも、プログラムの書き方が悪いため「偽の依存性」が生じ、並列化がまったくできなくなる場合も珍しくない。

以上から、コンパイラがどのように並列性の解析をしているのか理解することは、高性能プログラミングのための必須課題となる。次節において、並列性の解説から行う。

３．並列性の抽出

以下のループを考えよう。

```
for (i=0; i<n; i++) {
  A[i] = A[i] + B[i];
}
```

このループでは、i ループをどのような順番で計算しても、結果として得られる A[i]の値に影響しない。このようなループは、**データ依存がない**ループという。データ依存がないループでは、ベクトル化、スレッド化、MPI 並列化、OpenMP 並列化、自動並列化など、基本的な並列化がすべて行える。したがって、このようなループを構成するようにコードを書くことが高性能プログラミングの基本であり、究極の書法である。

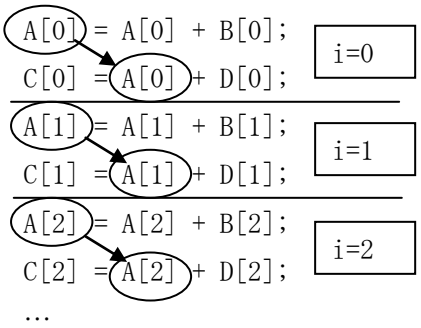
しかしながら一般には、ループに何らかのデータ依存が存在してうまくはいかない。このデータ依存の種類は、以下に説明する（１）流れ依存、（２）逆依存、（３）出力依存という３種類に分類されている。結論から言うと、（１）流れ依存以外は偽の依存である。

（１） 流れ依存

以下のループを考えよう。

```
for (i=0; i<n; i++) {
  A[i] = A[i] + B[i];
  C[i] = A[i] + D[i];
}
```

このとき、各ループの処理はループ変数 i について、以下のようになる。

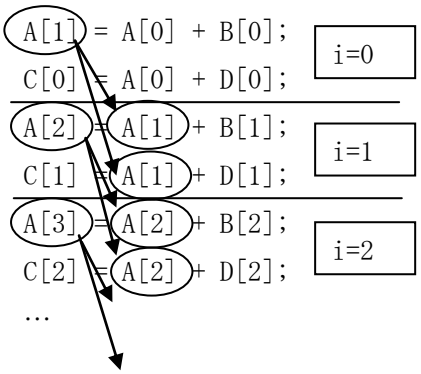


ここで、1行目の左辺のA[i]の値が計算されるまで、2行目の右辺のA[i]の値を参照できないことがわかる。このようなデータの依存を、**流れ依存**という。またこの場合、ループの中で、この流れ依存が閉じている。このような流れ依存を、**ループ独立の流れ依存**という。

一方、以下のループを考える。

```
for (i=0; i<n; i++) {
  A[i+1] = A[i] + B[i];
  C[i] = A[i] + D[i];
}
```

このとき、各ループの処理は、ループ変数iについて、以下のようになる。



この例では、A[i]についてフロー依存があるが、1行目の左辺のA[i+1]が、1行目、2行目の右辺のA[i]にループを伝搬して流れ依存している。このような依存を、**ループ伝搬流れ依存**という。

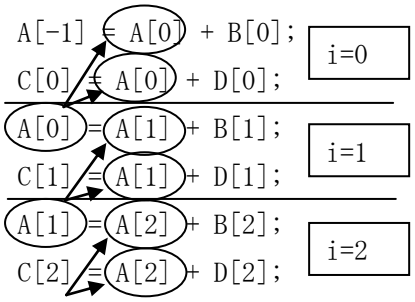
流れ依存は本質的な依存であり、コードの変更だけでは並列性を抽出できない。アルゴリズムの変更などを考慮しなくてはならない。

（2） 逆依存

以下のループを考える。

```
for (i=0; i<n; i++) {
  A[i-1] = A[i] + B[i];
  C[i] = A[i] + D[i];
}
```

このとき、各ループの処理はループ変数 i について、以下のようになる。



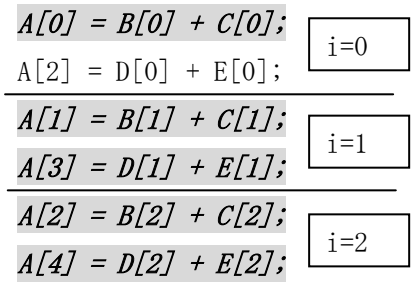
こんどは、1 行目の左辺の A[i-1]が、1 行目、２行目の右辺の A[i]それぞれについて、A[i]が参照されるまで A[i-1]の値を更新できない。このようなデータ依存を、**逆依存**という。また、上記の例の場合は、ループにまたがっているため、**ループ伝搬の逆依存**とよぶ。

（３） 出力依存

以下のループを考える。

```
for (i=0; i<n; i++) {
  A[i] = B[i] + C[i];
  A[i+2] = D[i] + E[i];
}
```

このとき、各ループの処理はループ変数 i について、以下のようになる。



これは、逐次実行としては問題を生じないが、ループ中の各行に並列性があると判断して実行すると逐次の結果と異なる。ここで、斜体の式の実行が最終的な値を決定する処理である。
以下のような並列実行をしてしまうと、逐次と異なる結果が残る。この例では、A[2]の値が逐

次と異なる。

A[0] = B[0] + C[0];	i=0
A[1] = B[1] + C[1];	i=1
A[2] = B[2] + C[2];	i=2

…

A[2] = D[2] + E[2];	i=0	←逐次と残る結果が異なる
A[3] = D[3] + E[3];	i=1	
A[4] = D[4] + E[4];	i=2	

…

このようなデータ依存のことを、**出力依存**という。

（4）解決策

さて、以上の「偽の」データ依存をどのように解決できるであろうか。結論からいうと、メモリやレジスタなどの計算機資源割り当ての都合から生じる偽の依存が逆依存と出力依存なため、解決策がある。具体的には、余分なメモリが確保できれば依存性を消滅でき、場合により格段に並列性を増せる。以下に、典型的な解決のための技法を述べる。

- **ノードスプリット[1]**

依存を生じるデータについて別配列を用意して、依存を断ち切る方法である。
先ほどの逆依存を生じるループ

```
for (i=0; i<n; i++) {
  A[i-1] = A[i] + B[i];
  C[i] = A[i] + D[i];
}
```

を、以下のように変更する。

```
for (i=0; i<n; i++) {
  TMP[i] = A[i];
  A[i-1] = TMP[i] + B[i];
  C[i] = TMP[i] + D[i];
}
```

こうすると、ループ伝搬の逆依存が消滅し並列化が可能となる。

```
TMP[0] = A[0];
A[-1] = TMP[0] + B[0];   i=0
```

```
C[0] = TMP[0] + D[0];
-------------------------------
TMP[1] = A[1];
A[0] = TMP[1] + B[1];   i=1
C[1] = TMP[1] + D[1];
-------------------------------
TMP[2] = A[2];
A[1] = A[2] + B[2];     i=2
C[2] = A[2] + D[2];
-------------------------------
…
```

● スカラエクスパンジョン[1]

以下のループを考える。

```
for (i=0; i<n; i++) {
  T = A[i] - SIGMA;
  B[i] = T * C[i];
}
```

このループでは、変数 T についてループ伝搬の逆依存がある。そこで、変数 T を配列 TMP[n]に拡張すると、以下のループになる。

```
for (i=0; i<n; i++) {
  TMP[i] = A[i] - SIGMA;
  B[i] = TMP[i] * C[i];
}
```

上記のループでは、逆依存が消滅して、完全に並列化できるループが構成される。

● ループ分割[1]

以下ループを考える。

```
for (i=0; i<n; i++) {
  A[i] = B[n-i-1] ;
  B[i] = C[i] * T;
}
```

このループでは､1 行目の B[n-i-1]のデータの参照について､i=n/2 以降において 2 行目の B[i]で更新された値を参照しなくてはならない。すなわち、1 行目右辺の B[n-i-1]は、2 行目左辺の B[i]に逆依存している。

だが、逆にいうと、i=1～n/2-1 においては、B についてはデータ依存がない。そこで、以下の

ようにループを分割すると、前半のループは並列に実行できる。

```
for (i=0; i<n/2; i++) {
  A[i] = B[n-i-1];
  B[i] = C[i] * T;
}
for (i=n/2; i<n; i++) {
  A[i] = B[n-i-1] ;
  B[i] = C[i] * T;
}
```

すなわち完全ではないが、部分的に並列実行されるループが構成されることになる。

以上のループ変換例は、コンパイラが自動並列化の際に行うこともある。しかしながら、ループの開始値と終了値が変数になっているだけでも、この並列化が困難となる場合がある。これは実例であるが、以下のようなループを考える。

```
for (i=i_start; i<i_end; i++) {
  ...
}
```

このループで、i_start、i_end がグローバル変数な場合は、コンパイラによる解析が困難となり、高性能なコードが生成されない場合がある[1]。また、ループの長さが解析できないので、並列化に関して十分な最適化ができない。

さらにループ中の式が複雑になるとコンパイラはデータ依存解析を断念し並列化が十分になされない。このような場合は、ユーザのもつ知識を活用し上記のループ変換をユーザ自らが行うことになる。

（5）データ依存がプログラムの字面から判断できない例

以上の例題は、プログラムの字面からデータ依存解析ができる例を示した。ところが一般のアプリケーションコードは、プログラムを実行するまでデータ依存が有るのか無いのか判明しない場合がある。数値計算の場合は、疎行列に関する演算で現れる。以下に実例を示す。

疎ベクトルの定数倍（x[] = x[] * c）を考える。以下のようなコードになる。

```
for (i=0; i<m; i++){
  x[ind[i]] *= c;
}
```

[1] i_start, i_end をローカル変数にするだけでも、性能改善がなされる場合がある。

ここで、このループに依存があるかどうかは、ind[]の値によって決まる。ind[]の値に重複がなければ、並列性がある。一方、ind[]の値が重複すると、フロー依存が生じ並列化ができない。一般に ind[]の値は実行時にならないと確定しないため、プログラムの字面だけでは並列性の判定ができない。

このように並列性が不明な場合は、コンパイラは「依存性あり」と判断して最適なコードを生成しない。もしさらなる最適化を望むのであれば、ind[]の値が重複しないというユーザ知識をコンパイラに伝えるという手段（ディレクティブ）が用意されているので、このディレクティブをユーザが記述することになる。

一方、分散メモリ型並列計算機環境（MPI を用いた並列化など）では、ind[]の値が重複しない場合においても ind[]の値の分布により並列実行性能に大きく影響する事態を引き起こす。これは、x[]を異なる分散メモリに分割する場合において、理想の場合（ind[i]が 0～m まで順番に並んでいる場合）は一切の通信なしで実行できること、および最悪の場合は、x の値を全ての分散メモリから収集しないと計算ができない（ind[]が x[]を分割した順番に並んでいる場合）からも容易に想像できよう。

以上のことから、コンパイラが行う並列化の処理についてユーザが知ることは高速化のカギであることがお分かり頂けると思う。

（６）数値計算での実例

今までの例はトイ・プログラム的であった。ここでは、数値計算ライブラリにおける実装で効果のあった実例を述べる。

実数対称行列の固有値計算では、二分法という方法がよく用いられる。二分法中で必要となる、ある値 SIGMA 以下の固有値数をカウントする部分（スツルムカウントという）は、以下のループ構造となる。

```
NEG = 0.0;
S = 0.0;
for (i=0; i<m; i++) {
  T = S - SIGMA;
  DPLAS = A[i] + T;
  S = T * B[i] / DPLAS;
  if (DPLAS < 0.0) NEG++;
}
```

ここで、変数 T と DPLAS はループ伝搬の逆依存があることがわかる。このことから、このままでは並列化ができない。変数 T と DPLAS に関しては、メモリに余裕があればスカラエクスパン

ジョンが適用できる。しかし変数Sに関しては、ループ伝搬のフロー依存がありコード変換のみでは解消ができない。このため、ループ自体の並列化ができない。

そこでアルゴリズムまで遡り、並列性を検討した。変数Sの計算に関して、計算する固有値ごとの並列性があり、その並列性を利用できることが判明した。最終的に、以下のようなループに再構築ができた。

```
NEG[0] = 0.0;
S[0] = 0.0;
for (j=0; j<k; j++) {
  for (i=0; i<m; i++) {
    T[j] = S[j] - SIGMA;
    DPLAS[j] = A[i] + T[j];
    S[j] = T[j] * B[i] / DPLAS[j];
    if (DPLAS[j] < 0.0) NEG[j] ++;
  }
}
```

すなわち、変数 T, DPLAS, S, および NEG について、スカラエクスパンジョン相当の方法が適用できた。上記のループでは、最も外側の j ループに並列性がある。したがって、並列化により速度向上が望めるループに書き換えられた[2]。

４．高速化に向くコード書法

今までは、ループに現れる並列性の抽出を意図した例であった。これは、OpenMP や自動並列化でのスレッド並列化を意図したものである。以降説明する高速化に向くコード書法は、CPU 内部にある演算ユニットレベルの並列性（あるいは、命令レベルの並列性）を意図したものが含まれている点に注意されたい。

（１）条件分岐情報の利用

- IF 文

  以下の IF 文を考える。

```
for (i=0; i<n; i++) {
...
  if (i == n) {
     A
  } else {
     B
```

[2] HITACHI SR11000/J2 モデルの１ノードでは、このループを含む実用的なコードにおいて、最大で 6.4 倍の高速化を得た。

```
  }
...
}
```

このループでは、n 回ループが回るうち、最後の 1 回だけ A の部分が実行される。このように、IF 文の成立条件が分かっている場合、成立確率が高いものを先に記載すると、高速化がなされる場合がある。すなわち、以下のようなコードにすると良い。

```
for (i=0; i<n; i++) {
...
  if (i != n) {
     B
  } else {
     A
  }
...
}
```

- swich 文

swich 文では、指定される変数のとりうる値の範囲が連続であるとき、高速な処理が期待できる。以下のように、離散的な値を持つ場合は、if-else 文で書いたほうが良い場合がある[3]。

```
swich (a)
{
  case 8:
    a=8 用の処理;
    break;
  case 16:
    a=16 用の処理
    break;
  case default:
    default 用の処理;
    break;
}
```

以上のコードは、以下のほうが良い。

```
if (a == 8) {
  a=8 用の処理;
  } else if (a == 16) {
```

```
      a=16 用の処理;
     } else {
       default 用の処理;
    }
  }
}
```

（２）コードの移動

● ループ内の IF 文

ループ中にループ外に出すことができる IF 文があれば、出すほうが良い。IF 文は時間がかかる処理なので、安易にループ中に書くべきでない。
たとえば、以下の例を考える。

```
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    if (b[j] != 0.0) {
      a[i] = a[i] * b[j];
    } else {
      a[i] = a[i] * EPS;
   }
  }
}
```

以上の IF 文では、i ループ中では b[j]の値は定数となるため、以下のように IF 文の実行回数が少なくなるように書くべきである。

```
for (j=0; j<n; j++) {
  if (b[j] != 0.0) {
    for (i=0; i<n; i++) {
       a[i] = a[i] * b[j];
    }
  } else {
    for (i=0; i<n; i++) {
      a[i] = a[i] * EPS;
    }
  }
}
```

また、ループ中の IF 文の移動のために、アルゴリズムから根本的に改める必要がある場合もある。行列の対角要素だけ特殊な処理をする以下の例を考える。

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    if (i == j) a[i][j] = 1.0;
    else        a[i][j] = b[i][j];
  }
}
```

上記の例では、ほとんどが else 文中での実行となるため、以下のように考え方を変えて記述するべきである。

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    a[i][j] = b[i][j];
  }
}
for (i=0; i<n; i++) {
  a[i][i] = 1.0;
}
```

● 共通部分式の削除

以下のコードを考える。

```
a = b + c / d;
e = c / d  + e;
```

以上の式では、共通な計算 c / d が表れるので、以下のようにするほうがよい。

```
dtemp = c / d;
a = dtemp + b;
e = dtemp + e;
```

● 割り算の移動

除算は一般に、実行時間が乗算や加算に比べてかかる。以下の正規化処理のループ中に除算が書かれている場合、低速となる。

```
for (i=0; i<n; i++) {
  a[i] = a[i] / dtemp;
}
```

したがって、以下のように逆数を取り、乗算化するほうが良い。

```
dinv = 1.0 / dtemp;
for (i=0; i<n; i++) {
  a[i] = a[i] * dinv;
}
```

ただし以上の２例（共通部分式の削除、割り算の移動）での変更は、場合により、変更前の計算結果と異なる場合がある。したがって、注意を要する最適化手法である。

（３）配列確保の順番

配列を宣言する時、大きな領域を必要とする倍精度実数型から順番に宣言をしないと、速度が低下する場合がある。これは、メモリからデータを取り込むハードウエアの実装が、倍精度実数型の境界（8バイト）単位に作られていることが多いためである。すなわち、4バイト変数と8バイト変数の宣言が混在して８バイト境界がずれるときに、速度低下が起こる場合がある。以下の順番で変数宣言されている場合を考える。

```
int  i, j, k;
double  x, y, z;
int  ia, ib;
float  v;
```

もし8バイト境界でメモリがアクセスされる場合､上記の書き方で配列が確保されてしまうと、倍精度変数ｘ、ｙ、ｚは８バイト境界にない。この場合、いままで１クロックでメモリから読み込めたとすると、少なくとも２クロックは必要となり大幅な速度低下となる。

そこで、以下のように８バイト境界を意識して宣言する。

```
double  x, y, z;
int  ia, ib;
int  i, j, k;
float  v;
```

以上のようなソートでの宣言が無理であれば、4バイトの宣言（たとえば、float）が８バイト境界に合うように、ダミー変数を宣言するという方法がある。このようにダミー変数を導入してメモリアクセスの改良を図る方法を**パディング**とよぶ。

５．ループ最適化

（１） ループアンローリング

ループアンローリングとは、ループ変数の刻み幅を１から2、4などに増加させ（ループを展

開し)、ループ自体のオーバヘッドの削減に加え、ループ中に書かれている式のデータ再利用性を向上させ（主にレジスタにデータを載せ）て高速化を狙う技法である[3]。

● 完全ループアンローリング

ループを完全に展開して、ループを消滅させる技法である。特にあとで説明する AMD64 プロセッサでは、ループ内部の命令が 100 命令に満たないときは、すべてをアンローリングするほうが高速である[3]。

```
for (i=0; i<3; i++) {
  a[i] = a[i] + b[i] * c;
}
```

完全アンローリング後：

```
a[0] = a[0] + b[0] * c;
a[1] = a[1] + b[1] * c;
a[2] = a[2] + b[2] * c;
```

● 部分ループアンローリング

ループ長が長い場合において完全アンローリングをすると、ループに関するオーハベッドが削減されるが、命令列を保持する高速メモリである命令キャッシュ（命令 L1 キャッシュ）に実行命令がヒットしなくなり、かえって速度が低下する場合がある。その場合行うのが、部分アンローリングという技法である。

例：内積計算

```
dtemp = 0.0;
for (i=0; i<n; i++) {
  dtemp = dtemp + a[i]*b[i];
}
```

部分アンローリングの例：

```
dtemp1 = 0.0;
dtemp2 = 0.0;
dtemp3 = 0.0;
dtemp4 = 0.0;
j=0;
```

[3] なお、コンパイラ用語のループアンローリングとは、複数入れ子になったループの最も内側を展開することを意味している。ここでのループアンローリングとは広義に解釈し、多重ループのどのループでも展開することを意味する。したがって、コンパイラ用語では別の技法（ループリストラクチャリングなど）を意味することがあるので注意が必要である。

```
  for (i=0; i<n/4; i++) {
    dtemp1 = dtemp1 + a[j  ]*b[j  ];
    dtemp2 = dtemp2 + a[j+1]*b[j+1];
    dtemp3 = dtemp3 + a[j+2]*b[j+2];
    dtemp4 = dtemp4 + a[j+3]*b[j+3];
    j += 4
  }
  dtemp = (dtemp1 + dtemp2) + (dtemp3 + dtemp4);
  for (i=(n/4)*4; i<n; i++) {
    dtemp = dtemp + a[i]*b[i];
  }
```

以上の部分アンローリングでは、i ループのカウントを 4 つ飛びに行っている。一般に、k 飛びにアンローリングすることを、k 段のアンローリングという。

さらに上記の部分アンローリングでは、ループ中の dtemp1～dtemp4 の計算自体に自明な 4 つの並列性が「字面として」表れている。この場合は、ループアンローリングすることで並列性も抽出ができるようになることがわかる。

- 配列の移動

ところで、ループアンローリングを配列について行うと、冗長な配列がループ中に現れることがある。たとえば、以下の行列 - ベクトル積演算

```
for (i=0; i<n; i++) {
  y[i] = 0.0;
  for (j=0; j<n; j++) {
    y[i] = y[i] + A[i][j] * x[j];
  }
}
```

を、i ループについて 2 段アンローリングすると、

```
for (i=0; i<n; i+=2) {
  y[i  ] = 0.0;
  y[i+1] = 0.0;
  for (j=0; j<n; j++) {
    y[i  ] = y[i  ] + A[i  ][j] * x[j];
    y[i+1] = y[i+1] + A[i+1][j] * x[j];
  }
}
```

となる。ここで、x[j]がループ中に 2 回現れる。この場合、コンパイラが適切に最適化をすれば、x[j]をメモリ上から 2 回読みこむことはない。しかし、ループ中の式が複雑になってくるとレジスタなどの高速なメモリに x[j]の値を保持できなくなり、メモリに値を書き戻すなどの速度低下が生じることがある。これを避けるには、明示的に x[j]の値の保持を記述する。それが、以下のコードである。

```
for (i=0; i<n; i+=2) {
  y[i  ] = 0.0;
  y[i+1] = 0.0;
  for (j=0; j<n; j++) {
    dx = x[j];
    y[i  ] = y[i  ] + A[i  ][j] * dx;
    y[i+1] = y[i+1] + A[i+1][j] * dx;
  }
}
```

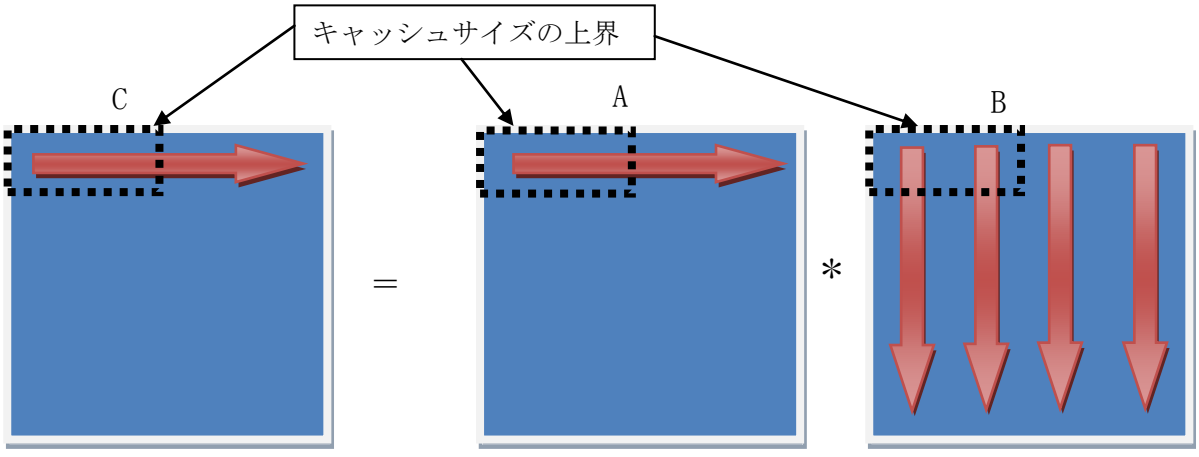
したがって、コンパイラの最適化を補助する書法としては、上記のものが良い。

（２） キャッシュブロック化（タイリング）

以下の行列 - 行列積コードを考える。

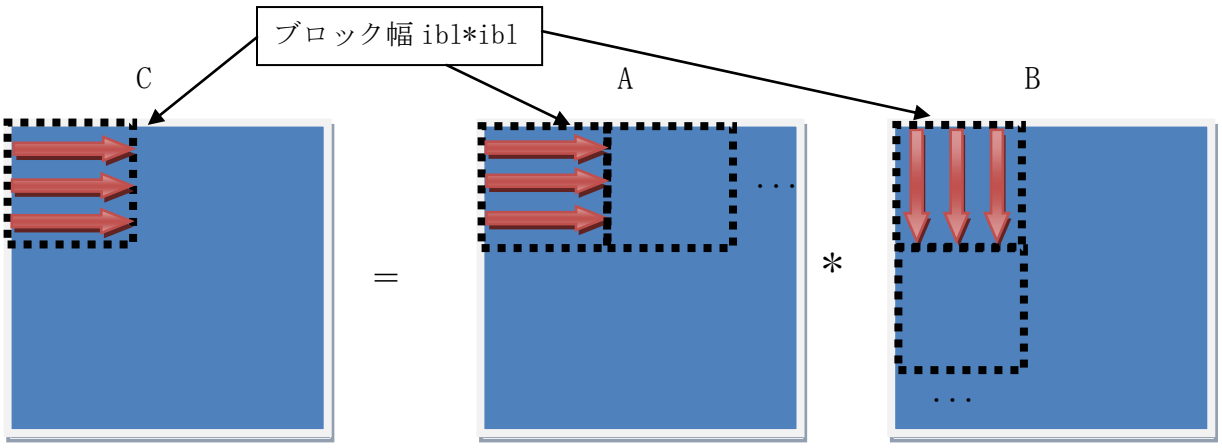
```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
     for (k=0; k<n; k++)
        C[i][j] += A[i][k] *B[k][j];
```

このコードでは n が大きくなると、配列 A、B、C について局所性がなくなり、速度の低下が起きる。これは、キャッシュと呼ばれる高速で小容量のメモリからデータが溢れてしまうこと（キャッシュミスヒット）により生じる。



行列 - 行列積では、ループ中に記載されている式が単純なため、配列 A、B、C のデータアクセス範囲を小さくするように分割して処理ができる。このような最適化手法を、**ブロック化**という。以下がそのコード例である。

```
ibl = 16;
for (ib=0; ib<n; ib+=ibl) {
  for (jb=0; jb<n; jb+=ibl) {
    for (kb=0; kb<n; kb+=ibl) {
      for (i=ib; i<ib+ibl; i++) {
        for (j=jb;  j<jb+ibl; j++) {
          for (k=kb; k<kb+ibl; k++) {
            C[i][j]  +=  A[i][k]  *  B[k][j];
          }  }  }  }  }  }
```



上記のコードでは、行列積がブロック幅 ibl 単位の正方領域（ibl*ibl）ごとになされており、配列 A、B、C のデータアクセスが（ibl*ibl）の範囲で局所化されている。つまり、正方領域（ibl*ibl）をキャッシュサイズ内に小さくできれば、速度向上が期待できるコードとなる。

なお、コードの修正のみでブロック化を実現することを、コンパイラ用語で**タイリング**という。タイリングのみではブロック化が困難で、アルゴリズムまで遡りブロック化を行う場合もある。このようにデータが局所化されるように変更されたアルゴリズムのことを、**ブロック化アルゴリズム**という。

行列-行列積の場合、ブロック化とループアンローリングが同時に適用できる。以下に、ブロック化に加え、i ループと j ループを同時に 2 段アンローリングを適用した例を載せる。

```
ibl = 16;
for (ib=0; ib<n; ib+=ibl) {
  for (jb=0; jb<n; jb+=ibl) {
    for (kb=0; kb<n; kb+=ibl) {
```

```
    for (i=ib; i<ib+ibl; i+=2) {
      for (j=jb; j<jb+ibl; j+=2) {
        for (k=kb; k<kb+ibl; k++) {
           C[i  ][j  ] += A[i  ][k] * B[k][j  ];
           C[i+1][j  ] += A[i+1][k] * B[k][j  ];
           C[i  ][j+1] += A[i  ][k] * B[k][j+1];
           C[i+1][j+1] += A[i+1][k] * B[k][j+1];
        } } } } } }
```

以上のコードのように、ブロック化とアンローリングを組み合わせることで、配列 A、B について共通部分の抽出も可能になったことが分かる。この共通部分の値をレジスタなどの高速なメモリに置くコードをコンパイラが生成すれば、局所化によるキャッシュの有効利用に加えて、レジスタの利用ができるようになり、相乗効果により大幅な速度向上が期待できる。さらに、字面においても 4 つの並列性が確認でき、並列性の向上が期待できる。

（3）ループ交換

再度、以下の行列 - 行列積のループを考える。

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
     for (k=0; k<n; k++)
        C[i][j] += A[i][k] *B[k][j];
```

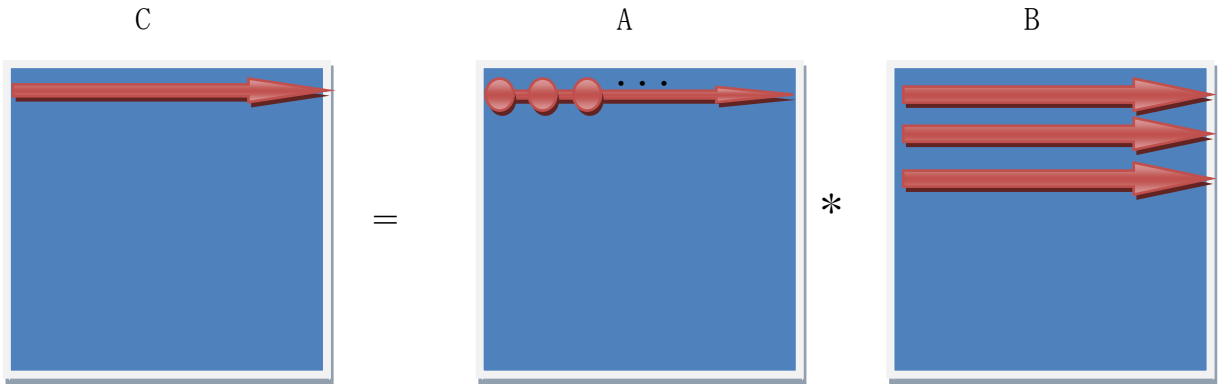
以上のループでは、ループ中に書かれている式に関して、i、j、k の 3 ループの順番を入れ替えても演算結果に影響しない。つまり、この場合には 6 通りの実現方法がある。このような場合には、ループ中の配列が連続的にアクセスするようにループを構成するほうが高速となる。これは、連続アクセスにより CPU へのデータの取り込みがハードウエアやソフトウエア的に促進される機構が実装されており、結果としてデータアクセス時間を削減できるからである。

C 言語の場合、配列は行方向に連続に確保されている。すなわち、C[i][j] の場合、第 2 添字 j 方向に連続にデータが確保されている。したがって、最も内側のループが、この第 2 添字の変数で書かれていることが望ましい。

上記の行列 - 行列積の場合、配列 B のアクセスは k ループにおいて連続アクセスにならない。この改善を試みる。以下のループを考えよう。

```
for (i=0; i<n; i++)
  for (k=0; k<n; k++)
    for (j=0; j<n; j++)
       C[i][j] += A[i][k] * B[k][j];
```

このループは、行方向アクセスが中心のデータアクセスパターンに改良される。
したがって C 言語環境では、高速化が期待できる。



しかしながら、ループ交換だけでは、キャッシュからのデータの溢れの問題は解決されない点に注意する。つまり、n が大きくなるとキャッシュミスヒットが生じ、性能低下が起こる。キャッシュブロック化とループ交換による連続アクセス化の併用が、メモリ階層を有す現在のプロセッサではきわめて効果的な高速化技法となる点に注意されたい。

６．ccNUMA 向けの最適化

（１）ジョブ・メモリ割り当ての最適化

T2K オープンスーパーコンピュータ（東大版）のノードの CPU は、AMD quad core Opteron プロセッサである。これは、Cache-Coherent Nonuniform Memory Access（ccNUMA）型の CPU である。これは、4 つのメインメモリ上にあるデータのアクセス時間が、4 つある CPU の場所によって異なるという計算機の型[4]である。

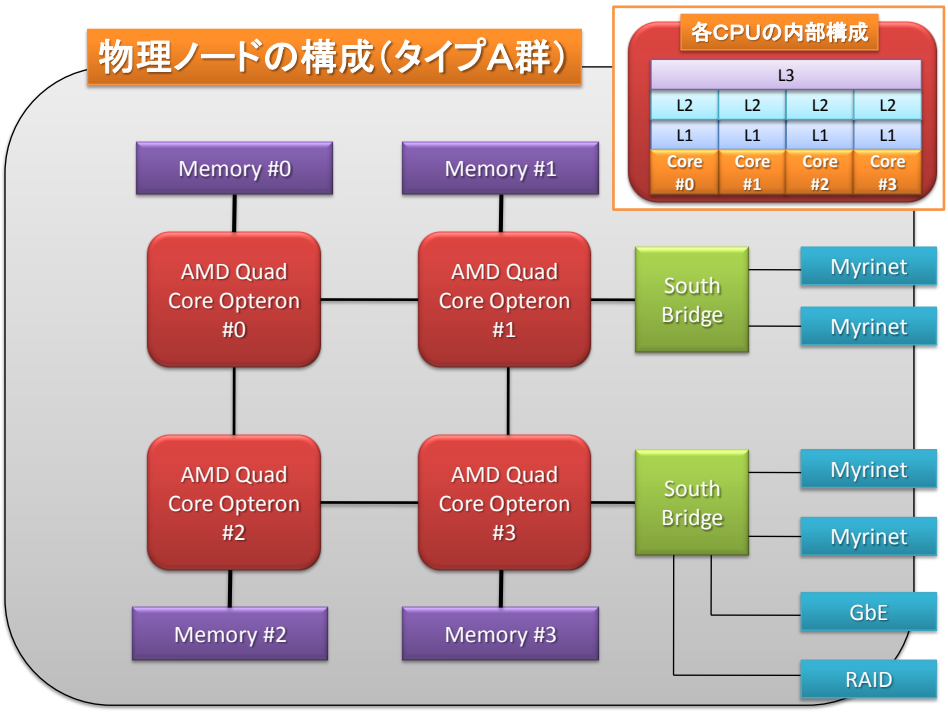


図 2　T2K オープンスパコン（東大版）のノード構成

図 2 のように、ノードには CPU が 4 つある（4 ソケット)。この 4CPU の中には、4 つのコアとよばれるプロセッサが入っている。１ノードは、16 プロセッサから構成されている。また、１ソケット内には、キャッシュがコアごとに 2 階層（L1 キャッシュ、L2 キャッシュ）ある。さらに L3 キャッシュもあり、これは 4 つのコアで共通化されているキャッシュである。キャッシュの容量は L1 が最も小さく、L3 が最も大きい。コアからのデータアクセス時間は、L1 が最も高速で、L3 が最も低速である。現在最新鋭のプロセッサでは、複雑なメモリ階層を有していることがご理解いただけると思う。
またノード間通信を行う場合、South Bridge を経由して行われるが、メモリアクセスの観点からノード内でもっとも高速なメモリは 2 か所となる（図 2 では、Memory＃1 と Memory＃3)。

ccNUMA の理由から、pthread を用いて自分でスレッドプログラミングする場合、コンパイラの自動並列化を用いる場合、OpenMP で並列化する場合、および、MPI で並列化する場合において、（１）並列実行後にスレッド・プロセスを割り当てる CPU の制御、（２）その割り当て CPU の近くのメモリを確保する制御、が性能の観点から必要になる。これは、先進計算機アーキテクチャ（一般に、マルチコアプロセッサとよばれる）の普及により生じた新技術要求である。

Linux の場合、（１)、（２）を実現する方法は、以下の２つがある。
I.　起動時に numactl コマンドを利用する方法
II. libnuma を使ってプログラミングする方法

I では、実行ファイル a.out 起動時に以下のコマンドを実行すると、起動時のジョブを指定の CPU#0 に固定でき、かつ、利用するメモリも#0 に固定できる。

```
numactl  --cpunodebind=0  --membind=0  a.out
```

また、複数のＣＰＵに割り当てられたスレッドがランダムにメモリをアクセスするプログラムの場合、メモリ割り当てを固定せず、すべてのメモリに分散してデータ配置をしたい場合がある。この場合は、以下のようにメモリへインターリーブ指定をする。

```
numactl --interleave=all  a.out
```

なお、Linux でサポートされるインターリーブポリシは、ラウンドロビン方式のみである[4]。ラウンドロビン方式とは、OS が管理するページサイズごとに、メモリへ循環するように割り当てる方式である。この大きさは、通常ページが 4KB、ラージページが 2MB である。

II では、NUMA 用ライブラリ libnuma が提供するライブラリを用いて、CPU 割り付け方法や、確保したメモリの NUMA 配置ポリシの指定を行う。CPU 割り当てについては、sched_setaffinity()関数でプロセス ID を任意の CPU に指定できる。メモリ配置は mbind()関数がある。指定アドレスから指定の大きさのメモリ範囲について、その NUMA 配置ポリシ（指定ノード限定、指定ノード交互、優先ノードから）を指定できる。

どのようにCPUを割り当てるか、どのようなNUMAメモリ配置ポリシを指定するかは、アプリケーションごとに異なる。したがって、ユーザは適する方式にチューニングする必要がある。

numactl は、OpenMP、自動並列化、MPI、およびMPI とOpenMP の混合環境、からでも利用できる。numactl 利用法に関しては、センター発行による利用手引[5]を参照されたい。

（２）単体性能の最適化

AMD64 Opteron の単体性能最適化は、コンパイラ最適化でカバーできない最適化については、今まで説明した技術を駆使して行わざるを得ない。ここで、ハードウエアの基本仕様を考慮する必要がある。

T2K オープンスパコンで採用されているAMD64 Opteron の基本仕様は表 1 とおりである[3][6]。

表 1 AMD64 Opteron 8356 の基本仕様（2.3GHz,4 コア）

項目	値
L1 キャッシュ	64 Kbytes（命令キャッシュ、データキャッシュ、双方は分離） 2 Way Associativity（ライトバック、3 サイクル） キャッシュライン：64 bytes、LRU 置換
L1 データ TLB	Full Associativity 2Mbyte Page:8 エントリ, 4Kbyte Page:32 エントリ
L2 キャッシュ	512 Kbytes（2300 Mhz）
L3 キャッシュ	2048 Kbytes
命令デコード	3 Way
演算実行	3 Way 整数演算、アドレス生成演算、浮動小数点演算（加算、乗算、ストア）
SIMD 命令	MMX, SSE, SSE2
レジスタ	● レガシーモード：汎用 32bit: 8 個、128bit-XMM:8 個、64bit MMX:8 個（x87 互換用:8 個と同一） ● 64bit モード：汎用 64bit:16 個、128bit-XMM:16 個、64bit MMX:8 個（x87 互換用:8 個と同一）
システムバス	1000 MHz

キャッシュブロック化では、L1 キャッシュサイズ（64Kbytes）を考慮する必要がある。アンローリングで抽出したデータの再利用においては、物理的なレジスタ数が考慮されるべきである。また、式として現れる並列度は、演算実行数（Way 数および演算対象）が考慮されるべきである。

なおLinux では、CPU の基本仕様情報が /proc/cpuinfo に記載されているので確認いただきたい。

行列 - 行列積などの数値計算処理は、128bit レジスタを用いた SIMD 命令（SSE2）を利用して実装すると高速化が期待できる。このようなアセンブラレベルのプログラミングは、次回「高性能プログラミング（Ⅱ）上級編」で解説を行う。

７．おわりに

本稿では、網羅的ではなく代表的な高速化プログラミング手法を説明した。紙面と時間の都合から、以下の項目については重要であると認識しているが説明ができなかった。

- プロファイリング
- 計算機言語に依存した最適化
- 分散メモリ型並列計算機での最適化（MPI による通信方式の最適化）
- 数値計算ライブラリ利用法

これらのトピックについては、次回以降に検討したい。

本稿で示した技法は、いずれもコンパイラの最適化オプションの影響を受ける。たとえば、アンローリングやタイリングについては、コンパイラが自動で行うので、それと衝突すると性能がでない（改悪となる）場合がある。しかしながら、コンパイラオプションでの最適化方式のいくつかは本稿で解説した方式が使われているので、ユーザにとって少しでも理解が深まったものと信じる。

最後に、本稿により、計算科学分野の研究進展に少しでも寄与できれば至上の喜びである。

参 考 文 献

[1] 並列コンピュータ工学、富田眞治著、昭晃堂、1996 年

[2] ハイ・パフォーマンス・コンピューティング、RISC ワークステーションで最高のパフォーマンスを引き出すための方法、Kevin Forwd 著、久良知真子訳、オーム社、1994 年

[3] Software Optimization Guide for AMD64 Processors, Advanced Micro Devices, Inc., Publication #25112, Revision:3.06, September 2005

[4] Performance Guidelines for AMD Athlon™ 64 and AMD Opteron™ ccNUMA Multiprocessor System, Application Note, Advanced Micro Devices, Inc., Publication #40555, Revision:3.00, June 2006

[5] HA8000 クラスタシステム　利用の手引、東京大学情報基盤センター、2008 年
http://www.cc.u-tokyo.ac.jp/misc/ha8000.html

[6] AMD64 Architecture Programmer's Manual Volume1: Application Programming, Advanced Micro Devices, Inc., Publication #24592, Revision:3.14, September 2007