SuperCon のためのベクトル計算入門 (2015 年版)

菊池誠 (大阪大学サイバーメディアセンター)

1 はじめに

ベクトル計算とはひとことでいえば、**配列の各要素に対して同じ計算を行なう**ことです。したがって、対象となるのは**配列に対するループ計算**です。PC のようなスカラー計算機ではそのような場合でもひとつの要素についての計算が終了したら次の要素という具合に順番に実行していきますが、ベクトル計算機では前の計算が終了するのを待たずに次の要素の計算を実行します。非常に単純化していうと、各要素についての計算をほぼ同時に行なうのです。NEC SX-ACE の CPU は、最大で 256 個の要素についての計算をほぼ同時に実行します。したがって、ベクトル計算機で高速に計算するためには、プログラムの中で計算時間のかかる部分をベクトル計算に向いたループとして上手に書くことが必要になります。以下では、ベクトル計算の基本を説明していきますが、ベクトル計算をうまく利用するためのプログラミングの基本方針は**あまり凝ったことをせず、素直なループを書く**ということに尽きます。

2 ベクトル化の例

ベクトル計算に向いた典型的なループはたとえば次のようなものです

```
double a[1000],b[1000],c[1000];
int i;
for(i=0; i<1000; ++i){
a[i] = b[i] * c[i];
}
```

これは配列の要素同士の掛け算をして、その結果を別の配列にいれています。あるいは配列の各要素に対して同じ (といっても、ループ変数に依存する) 操作をする

```
double a[1000],b[1000];
int i;
for(i=0; i<1000; ++i){
a[i] = b[i] + (i*2);
}
```

なども典型的なもので。このような計算はコンパイラが自動的に**ベクトル計算**と判断して、(ほぼ) 同時に実行してくれます。これはコンパイラによる**自動ベクトル化**と呼ばれます。ループの繰り返し回数を**ベクトル長**と呼びます。上のふたつの例ではベクトル長は 1000 です。一般にベクトル長が長いほど、計算の効率が高く (したがって、計算ひとつあたりの所要時間は短く) なります。SX-ACE では 256 の計算を同時に実行できますから、ベクトル長 256 までは効率がどんどん高くなり、それ以上のベクトル長では最大効率を維持します。逆にあまりに短いループ (ベクトル長が 5 以下) は却って時間がかかるため、ベクトル化されません。

C 言語でプログラムを書く場合は、以下に説明するように、コンパイラが「この計算はベクトル化できるな」と判断しやすいように書かなくてはなりません。なお、ベクトル化は for ループだけ

ではなく、while や if-goto などでも (条件さえあえば) 行なわれますが、多くの場合は for ループを使うことになるでしょう。

上の例では、ループの中に「代入」のほかに「和」や「積」の演算子が登場しましたが、ほかにもたいていの演算子 (+、−、∗、/、% など) は使えると思ってかまいません。条件判断を含む三項式 ((A)?B:C のタイプ) も大丈夫です。また、ビット演算 (ビット毎の論理和 (|)、論理積 (&) など) もたいてい問題なく使えますが、ビットをシフトする演算 (演算子 ≪ と ≫) はシフトする数 (演算子の右辺) が変数だとベクトル化できないようです。

さて、for ループならどんなものでもベクトル化されるのかというと、もちろんそういうわけにはいきません。ベクトル化できるループは各要素に対する計算が**独立**していることが必要です。独立というのは、**順序を変えても計算結果が変わらないこと**です。実際、プログラム上は順番に実行することになっているものを同時に実行しようというのですから、実際の計算順序はプログラムに書かれているものと違ってしまいます。それでも計算結果が変わらないようなループでなくてはならないのです。上の例は一見して、順序を変えても大丈夫であることがわかります。i=0 に対する計算 (a[0] = b[0]*c[0]) はたとえば i=100 の計算 (a[100] = b[100]*c[100]) の結果とまったく関係ありませんから、異なる i の値についての計算をいくつでも同時に行なうことができ、その結果は順番に計算した場合と同じです。これが**独立**の意味です。詳しくは後の「定義・参照関係」で説明します。

では

```
double a[10000], x;
int i;
x=0;
for(i=0; i<1000; ++i){
x += a[i];
}
```

はどうでしょうか。計算順序を変えるとループの途中では x の値が違ってしまいますから、ベクトル化できないようにも思えます。しかし、これは結局、配列 a の 0 から 999 番目までを全部 x に足すだけなので、どの順序で足そうと最後の結果は同じですね。実はこのようなループは、コンパイラが**総和の計算**と判断して、きちんとベクトル化してくれます。もちろん、和の代わりに積でも大丈夫です。「総和」以外にも「最大値・最小値」を求める計算など、いくつかの典型的なパターンはベクトル化されます。たとえば最小値の計算は

```
double a[10000], xmin;
int i,imin;
xmin=a[0];
imin=0;
for(i=0; i<1000; ++i){
if(a[i] < xmin) { xmin = a[i]; imin = i;}
}
```

のようなものです。if が含まれていて、一見ベクトル化できるのかなと思いますが、この場合はコンパイラが「最小値計算」というパターンと判断します。xmin や imin がループ全体で共有されていて、これも問題になりそうですが、これは大丈夫です。if が含まれる一般のループでもベクトル化できますが、それはまたあとで

ベクトル化されるループには文がいくつ含まれていてもかまいません。たとえば

```
double a[1000],b[1000],c[1000];
int i;
double xmin;
a[0] = b[0] * c[0];
xmin = a[0];
for(i=0; i<1000; ++i){
a[i] = b[i] * c[i];
if(a[i] < xmin) { xmin = a[i];}
}
```

はベクトル化できる文の組み合わせなので、全体をベクトル化します。ちなみに、実際には、コンパイラがこれを

```
for(i=0; i<1000; ++i){
a[i] = b[i] * c[i];
}
for(i=0; i<1000; ++i){
if(a[i] < xmin) { xmin = a[i];}
}
```

と解釈してベクトル化します。つまり、プログラムとは計算順序が変わるわけです。この点は、どういうループがベクトル化可能かを考える上で重要なヒントになるので、頭に入れておいてください。なお、ループの中にベクトル化できない文が含まれる場合には、ベクトル化できる文だけがベクトル化されます。

3 関数

libm に含まれる組み込み関数の多くはベクトル化されますので、たとえば

```
x = 0;
for(i=0; i<1000; ++i){
x += sqrt(a[i] * a[i] + b[i] * b[i] );
}
```

のような計算もできます。これは平面上で二点間の距離の総和 (というより、空間ベクトルの長さですが) を求めていますね。当然、もっとも短い距離を求める

```
xmin = sqrt(a[0] * a[0] + b[0] * b[0] );
imin=0;
for(i=1; i<1000; ++i){
if(( x = sqrt(a[i] * a[i] + b[i] * b[i] ) ) < xmin){
xmin = x; imin=i;}
}
```

も大丈夫です。

しかし、どんな関数の呼び出しが含まれていてもいいのかというと、そういうわけではなく、libm は特別と考えておいてください。自分で作った関数をループ内で使っていると、そのループはベクトル化されません。その場合には関数の中身をループ中に直接書いてしまいましょう。また、ループ中に入出力があるようなものもベクトル化できません。

4 制御変数・要素番号

配列の要素番号がずれていても、もちろん大丈夫です。たとえば

```
double a[1000],b[1000],c[1000];
int i;
for(i=1; i < 998; ++i){
a[i] = b[i-1] * c[i+2];
}
```

は問題なくベクトル化されます。

もちろん、ループの制御変数は 1 ずつ増えなくてもかまいません。2 ずつとか 5 ずつとか、「定数」で増える場合や逆に減る場合もベクトル化されます。その場合の「ベクトル長」は実際に繰り返される回数です。

要素番号が制御変数の計算式になっていても大丈夫です。たとえば

```
double a[1000],b[2000],c[3005];
int i;
for(i=0; i < 1000; ++i){
a[i] = b[i*2] * c[i*3+5];
}
```

また、よく使われるのは、要素番号が別の配列で決まる場合です。これは「リストベクトル」と呼ばれます。たとえば

```
double a[1000],b[1000],c[100];
int ix[1000], iy[1000];
for(i=0; i < 1000; ++i){
a[ix[i]] = b[i] * c[iy[i]];
}
```

のようなものです。もちろん、ix や iy の値が a や c の要素番号を指すように (a の要素番号は 0 から 999 までしかないのに、ix[100] の値が 1000 だの 10000 だのになったりしないように)、プログラム中できちんと決めておかなくてはならないことは言うまでもありません。

5 if

配列のうちで特定の条件を満たすものについてだけ計算する場合、if 文を使います。これもベクトル化できるのですが、実際に計算機が行なうのは、条件に合う要素だけを集めた短い配列を作って、それをベクトル化することです。というわけで

```
for(i=0; i<10000; ++i){
if(i% 2 == 0 ){ a[i] = b[i];}
}
```

はベクトル化されます。もっとも、この場合は

```
for(i=0; i<10000; i+=2){ a[i] = b[i];}
```

が普通ですね。if を使わずにすむなら、使わないほうが計算は高速になります。

もっと複雑な例としては、ある長さ r0 以上の空間ベクトルがいくつあるかを求める

```
n =0;
for(i=1; i<1000; ++i){
if(sqrt(a[i] * a[i] + b[i] * b[i] ) > r0 ){ ++n; }
}
```

などでも大丈夫です。

もちろん、else や else if があってもかまいません。ただし、if のあとに goto を使う場合には、その行き先がループ内にあって、かつ goto で飛ぶ先が goto よりもあとになくてはならない (前に戻ってはいけない) という規則があります。ループ内に複数の文があるときには計算順序が変わるということを思い出せば理解しやすいでしょう。つまり、

```
n =0;
m =0;
for(i=1; i<1000; ++i){
if(sqrt(a[i] * a[i] + b[i] * b[i] ) > r0 )goto A;
++n;
A: ++m;
}
```

のような計算はベクトル化されるということです。もっとも、これなら if...else で書いたほうがきれいですし、そうするべきですが。

いっぽう、ループの外へ飛び出す if は特殊な形のみがベクトル化されます。基本的には、「ある条件に合う要素を見つける」場合だけと考えておけばよいでしょう。以下のタイプです。

```
for(i=1; i<1000; ++i){
if(a[i] == 1 )break;
}
```

6 多重ループ・多次元配列

多重ループの場合、基本的には一番内側のループがベクトル化されます。その際、

```
for(j=0; j<1000; ++j){
for(i=0; i<1000; ++i){
a[i][j] = b[i][j] * c[i][j];
}
```

と

```
for(i=0; i<1000; ++i){
for(j=0; j<1000; ++j){
a[i][j] = b[i][j] * c[i][j];
}
```

では計算効率が違います。多次元配列は計算機のメモリ上で、もっとも右側の添え字が連続になるように配置されます (a[i][1] と a[i][2] は隣同士)。いっぽう、左側の添え字が 1 違っているだけでも、メモリ上では離れたところに配置されます (a[1][j] と a[2][j] は離れた位置に置かれる)。メモリから値を呼び出す際、なるべくメモリ上で連続した位置から呼び出すほうが効率がよいので、上のふたつの例では j を内側のループにした下のプログラムのほうが速いのです。同様に、右側の添え字に関するベクトル長をもっとも長くするのも、計算効率を高くするポイントです。

実は上の例のように簡単なループは 2 重ループであっても全体をベクトル化します。つまり、コンパイラはあたかもベクトル長 1000000 の 1 重ループであるかのように扱うわけです。ただし、多重ループ全体をベクトル化するのは、実行される文が最内側のループの中にしか含まれていないような簡単な場合に限られます。実際にはコンパイラがかなり賢く、内側と外側のループを入れ替えて全体をベクトル化可能にするなどの操作を自動的にやってくれますが、どういう場合にそれが可能かについてはいろいろと条件があります。

7 定義・参照関係

ベクトル化できるかできないかは、ループ内での定義と参照の関係、つまり代入の左辺にあるものと右辺にあるものとの関係で決まります。ループ内に複数の文があるときは、ベクトル化によって実行順序が変わるので、それで計算結果が変わってしまうようなループはベクトル化できません。

```
double a[1001],b[1000],c[1000];
int i;
for(i=0; i<1000; ++i){
c[i] = a[i];
a[i+1] = b[i];
}
```

は明らかに

```
for(i=0; i<1000; ++i){ c[i] = a[i];}
for(i=0; i<1000; ++i){ a[i+1] = b[i];}
```

とは違う計算なので、このままではベクトル化できません。いっぽう

```
for(i=0; i<1000; ++i){
a[i+1] = b[i];
c[i] = a[i];
}
```

は

```
for(i=0; i<1000; ++i){ a[i+1] = b[i];}
for(i=0; i<1000; ++i){ c[i] = a[i];}
```

と一致するのでベクトル化できます。同じ変数が二度出てくるときは、定義 (代入の左辺) が参照 (右辺) より先にあれば大丈夫ということです。ただし、このくらい簡単なループなら、コンパイラが文の順序を入れ替えてでもベクトル化してくれるようですが、基本的にはできないと考えてください。

では

```
double a[1000],b[1000]
int i;
for(i=0; i<1000; ++i){
x = a[i];
b[i] = x;
}
```

は

```
for(i=0; i<1000; ++i){ x = a[i];}
for(i=0; i<1000; ++i){ b[i] = x;}
```

と違うではないかと思うかもしれませんが、この場合の x は**作業変数**といって、ループの各繰り返しごとにいったん値を覚えるだけのものであることをコンパイラが判断して、正しい解釈でベクトル化します。ここでも定義が参照より先にあることが重要で、逆の

```
double a[1000],b[1000]
int i;
for(i=0; i<1000; ++i){
b[i] = x;
x = a[i];
}
```

はベクトル化されません。なお、作業変数は積極的に使ってください。これを無理に配列にすると却って計算が遅くなります。

上で見た例を少し変えて

```
double a[1001],b[1000],c[1000];
int i;
for(i=0; i<1000; i+=2){
c[i] = a[i];
a[i+1] = b[i];
}
```

とすると (i が 2 ずつ変わる)、今度は定義参照関係に問題がなくなって、ベクトル化できます。なぜなら、これは

```
for(i=0; i<1000; i+=2){ a[i+1] = b[i];}
for(i=0; i<1000; i+=2){ c[i] = a[i];}
```

と同じ計算だからです。定義されているのは a の奇数番要素で、参照されているのは a の偶数番要素なので、順番を変えても結果に影響しないのです。

ところで、同じ配列の違う要素が式の両辺にある場合

```
double a[1000],b[1000];
int i;
for(i=0; i < 999; ++i){
a[i] = a[i+1] * b[i];
}
```

はなんとなくベクトル化できそうなのに対して (実際、できます)、逆の

```
double a[1000],b[1000];
int i;
for(i=1; i < 1000; ++i){
a[i] = a[i-1] * b[i];
}
```

はベクトル化されそうにありません。しかし、これも「漸化式」と呼ばれる特殊なパターンとしてベクトル化されます (積ではなく、和や差でも大丈夫です)。

8 ベクトル化指示行

一見同じように見えるループでも、微妙な違い (計算としては大きな違いですが) で定義参照関係がベクトル化向きだったりベクトル化に不向きだったりするわけです。これをコンパイラが正しく判断できればいいのですが、できない場合もあります。リストベクトルを使うときがその例です。たとえば

```
double a[1001],b[1000],c[1000];
int i, ix[1000], iy[1000];
for(i=0; i<1000; ++i){
c[i] = a[ix[i]];
a[iy[i]] = b[i];
}
```

となっていたら、実際にはベクトル化してかまわないループでも、コンパイラはそう判断してくれません。そういうときは、**ベクトル化指示行**という特別な命令を使って、「ベクトル化してよい」という情報をコンパイラに教える必要があります。上の例では以下のように書きます

```
double a[1001],b[1000],c[1000];
int i, ix[1000], iy[1000];
#pragma vdir nodep
for(i=0; i<1000; ++i){
```

```
c[i] = a[ix[i]];
a[iy[i]] = b[i];
}
```

#pragma vdir nodep と書いたのがベクトル化指示行です。これは指示行のすぐ次のループでは定義・参照関係がきちんとベクトル化向きになっているよ、ということをコンパイラに教えています。nodep は no dependence(依存関係なし) の略です。ベクトル化指示には他にもさまざまなものがあるのですが、一番よく使うのはこの nodep です。おそらく、SuperCon ではこれだけを知っていればいいでしょう。なお、ベクトル化指示行は C 言語の一部ではありません。ここで使った nodep も SX-ACE 独自の命令です。#pragma は処理系ごとの独自の命令を書くためのものでなので、これを含むプログラムを LINUX の gcc でコンパイルしても単に無視されるだけで、影響はありません。

9 関数

関数に配列をわたす方法は基本的に普通と同じです。ループの回数が引数になったりしていてコンパイル時に不定でも問題ありません。ただ、コンパイラが「定義・参照関係」を理解できないと自動ベクトル化できないので、凝ったポインターの使い方をしないほうがいいでしょう。プログラムとしては美しくなくても、配列をグローバル変数にしてしまったほうが簡単で確実な場合もあると思います。また、リストベクトルを使うとうまくベクトル化できることもあるでしょう。関数の中ではコンパイラが定義・参照関係を理解できない場合も多いので、#pragma vdir nodep を適宜使ってください。

10 並列化

並列化はベクトル化とはまったく違う概念です。ベクトル化がループ部分を同時計算するための特別な演算器で行なわれるのに対し、並列化は計算を多数の CPU に振り分けて行ないます。並列化のほうができることの範囲が広くて応用もしやすい代わり、ベクトル化できるループについてはベクトル化のほうが高い計算効率を持ちます。SX-ACE はベクトル演算器を持つ CPU を複数つないだ**ベクトル並列型計算機**です。せっかくなので、並列化の機能も活用しましょう。

SX-ACE では細かく指定すれば複雑な並列計算ができますが、今回は**自動並列化**に限ることにします。といっても、これだけでもかなり賢く並列化してくれます。自動並列化のためにはコンパイル時にオプション-Pauto を指定します。主として並列化されるのは多重ループの外側ループです。これをいくつかに分けて、異なる CPU に振り分けて計算します。もちろん、並列化できるのは並列に実行しても結果が変わらないようなループに限られています。ベクトル化と違い、並列化によって総 CPU 時間は減りません。それどころか、並列化に伴う余分な手順のために、CPU 時間としてはむしろ増えることになります。しかし、それが同時に行なわれるので、実時間で見れば短くなるわけです。1 ノードの SX-ACE は 4 個の CPU で構成されており、最大で 4CPU での並列計算が可能です。多数のノードを使えばもっと大規模な並列計算ができますが、SuperCon では 4 並列までとしましょう。それでも、並列化しやすいプログラムを作れば、計算時間は 1/4 に短縮できます。

注意したいのは、並列化が**関数単位**で行なわれることです。並列に実行されるループを含む関数は、ループ以外も含めた関数全体が並列化の対象となります。ループ以外はすべての CPU でまっ

たく同じ計算が行なわれることになります。ただし、自動並列化では、ひとつの関数の中をさらにいくつかの関数に分けて、並列化できる部分だけを並列化しようとするので、あまり妙なことは起きないかもしれません。このあたりは、コンパイラまかせの部分が多いので、結果を見てください。

もうひとつの重要な注意は、並列化によってメモリーの使用量が増えることです。特に関数内でローカルに定義した配列は並列計算の数だけコピーされるので、場合によってはメモリー使用量が爆発的に増えます。いっぽう、グローバル変数ならそのようなことにはならず、同じメモリーを CPU 間で共有します。したがって、非常に巨大な配列はグローバルに確保することを薦めます。もっとも、SuperCon の問題ではそれほど問題にはならないだろうとは思います。

11 おわりに

PC でのプログラミングに慣れていると、ベクトル化や並列化にはとまどう点が多いと思います。効率よい計算のためには計算量の少ないアルゴリズムを工夫すべしという基本は変わりません。しかし、同じ C 言語で書くといっても、PC 用にがっちりとチューニングしたプログラムではベクトル計算機で効率が出ないでしょう。ベクトル化をうまく使うには、C 言語の粋を極めたような凝ったプログラムよりは、シンプルで素直なプログラムのほうが向いています。たとえば、再帰的プログラムはまずうまくいかないと思っておいてください (もちろん、ベクトル化されないだけで、計算はしてくれますが、計算効率は期待できません)。また、メモリーもずっと大きく、PC では malloc を使わないと確保できないような大きな配列も気にせずに使うことができます。メモリーの節約を考える必要はまずありません。大きな配列を取ってしまって、大きなループをシンプルに回すようなプログラミング・スタイルがいいでしょう。PC で速いプログラムの書き方とベクトル計算機で速いプログラムの書き方とはかなり違うのだなあということをこの機会に感じてみてください。