圧縮性流体差分計算のための C++コードのベクトル化

岩本 幸治[1)]、村上 匡且[2)]

[1)]愛媛大学大学院 理工学研究科

[2)]大阪大学 レーザーエネルギー学研究センター

1. はじめに

2011 年 11 月、大阪大学基礎工学研究科の河原源太先生のご紹介で著者の 1 人である村上匡且先生が取り組まれている爆縮の数値計算をすることになった。「圧縮性流体の数値計算ができればよいから」ということであったが、不勉強な筆者には爆縮なんて現象は聞いたことがなく、どういう計算をすれば良いのかを自分なりに理解するまでに 1 か月ほどかかった。要は以下のようなことであった。球面状の衝撃波が中心に向かって収縮し、その後中心から反射する現象を考える。中心から衝撃波までの距離 R が時間の累乗で変化すると仮定する（とりあえずの仮定であるが、解いてみると仮定を満たす解があることが分かる）。さらに半径位置 r を R で無次元化した無次元座標 ξ を定義する。このようにすると、流体の速度、密度、圧力などの物理量が ξ を変数とする無次元関数を含む形で表すことができ、その関数を求めておけば任意の時間、半径位置における物理量を 1 つにまとめて表現することができるというものである。これは Guderley の自己相似解として知られているそうである[1]。Guderley の自己相似解が表す流れは 3 次元球対称流れであるが、今回の計算では超球幾何形状と呼んでいる、いわばトランペット状にした壁面を使用するもので、球対称の場合よりも高圧、高密度にガスを圧縮できる。その流れを数値計算で示してほしいというものであった。

計算自体は軸対称流れ（2 次元流れに多少の付加項が付いたもの）である。筆者は学生時代に圧縮性流体の差分計算を行っていたが、その当時の計算コードは FORTRAN77 で書かれていた。計算環境が変わり、最近の筆者はもっぱら C++でコードを書くようになっていた。これからも環境は大きく変わらないと思ったので、これを機にコードを C++で書き直した。とはいえさほど大した苦労はなく、1 ヶ月ほどでできた。というのも、擬似圧縮性法[2]による 3 次元数値計算コードを C++であらかじめ作っていたからである。擬似圧縮性法は圧縮性流体の解法を非圧縮性流体に拡張したものであるため、圧縮性流体のプログラムと類似点がかなり多い。

計算結果の一例を図 1 に示す。図は比熱比 7/5 で計算したものである。τ は衝撃波が焦点に到達するまでの理論上の時間であり、負または正の時間 t においてそれぞれ衝撃波が収縮または反射する。図 1 では各時間において広がりが大きい場合と小さい場合を示しており、軸対象流れの断面のうち、上半分を表示している。広がりが大きい場合は収縮時の $t/\tau = -0.2$ において衝撃波の変形が見られる。それに対して広がりが小さい場合は反射後（$t > 0$）においても面状の衝撃波を維持している。これは球対称流れに対してより厳密に行った線形安定解析で定性的に説明できる[3]。

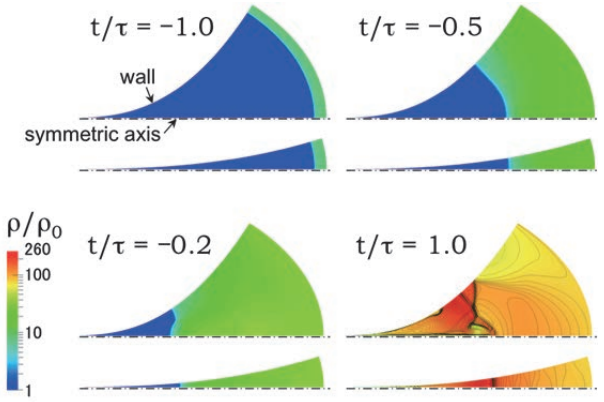

図 1：超球幾何形状中の爆縮における密度のカラーコンター[3]

パラメータを変えて図 1 のような計算を計算しているうちに、高圧縮条件（壁面の曲率が大きい）では衝撃波が中心に収束する際に非常に小さな時間ス

テップをとらないと負の絶対温度にアンダーシュートしてしまうことが分かった。そのため計算時間が予想以上にかかってしまうようになった。少しでも計算が速くなるよう村上先生と相談したところ、「阪大 ILE にあるスパコンを使えばどうか？」というご提案をいただいた。そこで阪大 ILE を訪問し、ILE の長友英夫先生や計算機室の福田優子さんからベクトル型スパコン SX-8R の使い方をご説明いただいた。このとき初めてベクトル計算機の実態を知り、ベクトル化のためにコードの書き直しが必要で、筆者が好んで使用している C++で書き直している人はほとんどいないために自身でノウハウを蓄積するしかない、という現実を知らされることになった。

少しずつ問題を解決し、1 年弱を経てようやくベクトル計算機を使う甲斐のある計算が可能になった。ベクトル演算率も 97%にまで向上した。本稿では C++でベクトル化コードに書き直した際に参考にしたもの、初心者では気づきにくい点、およびベクトル演算率向上に貢献したテクニックを紹介する。

2. C++によるベクトル化コードの作成

2.1 参考にした資料

コンパイラの使い方は阪大サイバーメディアセンターのポータルシステムから入手できる手引書[4]から学んだ。基本的にはこれだけで十分であった。ただ、発生する様々なエラーを調べているうち、ベクトル計算とは何か？という基本的なことを理解しないと解決できない気がしてきた。そこでベクトル化についても学ぶようになった。それには講習会の資料[5]が役立った。

後述するように、最初の段階ではほとんどのループがベクトル化されなかった。言語に特有の問題があるかもしれないと考えて Web で検索したところ、C 言語のソースを SX4 用にチューニングした方による記事[6]を見つけた。この記事は日記風に書かれており、短くまとめられていて助かった。とくに組み込みでない関数をループ内で呼び出すとベクトル化しないことをこの記事で初めて知った。（手引書[4]125 ページにも記載はあるが、当初は見つけられなかった。）

2.2 extern "C"ブロックでリンクエラーを起こす場合

あまり起こらないかもしれないが、最初に経験したエラーを紹介する。多くの流れ場の差分計算プログラムがそうであるように、ソースは複数のファイルに分散され、Makefile で管理している。Makefile の中ではそれぞれのソースファイルをまず

> sxc++ -c ソースファイル.cpp（または c）

（オプション-c でコンパイルのみを行い、リンカを起動させない）で拡張子 o のオブジェクトファイルに変換し、最後に

> sxc++ オブジェクトファイル 1.o オブジェクトファイル 2.o …

というふうにリンクしていた。ここで include するヘッダファイルのプロトタイプ宣言が extern "C"ブロック内に入っているとリンクエラーを起こす。理由は簡単で、sxc++ -c によってオブジェクトファイルを作るとソースの拡張子が何であれ全て C++のソースとして扱われるからである。よってコンパイルされて出来る関数は全てマングルされた名前を持つ（関数の仮引数の型情報を付け加えた名前に変換される）。一方、extern "C"ブロック内にプロトタイプ宣言がある関数はマングルされていない名前でプログラムから探される。そんな名前の関数はあるはずもなく、リンクエラーを起こす。今回は extern "C"ブロックがなくてもよいプログラムであったので、それらを全て取り除いた。これによってコンパイルが通るようになった。

知識のある方ならば extern "C"ブロック内にプロトタイプ宣言がある関数のソースは sxcc でコンパイルしなければいけないと即座に指摘されるであろう。しかし、それが分からなかった。というのは Linux の g++やオプション/Tp つきの Visual C++でコンパイルする場合はリンクエラーが起こらないからである。C++用であるにもかかわらず、これらのコンパイラは extern "C"ブロック内で宣言されている関数に対しては名前をマングルしない。これを全てのコ

ンパイラで共通の仕様と勘違いしていた。g++やVisual C++に既に慣れておられる方は陥るかもしれない問題なので念のために記しておいた。

なお、後に分かったことであるがsxc++でもextern "C"ブロック内にプロトタイプ宣言がある関数の名前をマングルさせないようにすることも出来る。それは以下のように、途中でオブジェクトファイルを作らずに一気にリンクまで行うことである。

```
> sxc++ ソースファイル1.cpp（または c） ソースファイル2.cpp（または c） …
```

2.3 自作クラスが保有するポインタによる自動ベクトル化の阻害

ようやくコンパイルが通るようになったが、編集リスト（手引書[4]199 ページ）を確認すると、ほとんどのループが自動ベクトル化されていなかった。原因の 1 つは、前述のように組み込みでない関数をループ内で呼び出していたことである。呼び出している関数の命令を全てループ内に直接書き込み、再びコンパイルしてみた。しかし、まだベクトル化されない部分が多々あった。手引書[4]を調べるうちに、自作クラスが保有するポインタが自動ベクトル化を阻害していることが分かった。これらのポインタは配列を動的に確保するために使用している。自作クラスが保有するポインタは関数の単位を超えて生存するため、別の関数でアドレスを共有する命令を追加することもできる。もしアドレスが共有されていたら、データ依存関係が発生するかもしれない。とすればふつうのスカラー計算と異なる計算をベクトル計算が行う可能性がある。よって自動ベクトル化をコンパイラが回避していたのである。よく見ると手引書[4]120 ページに同じ趣旨が記されているが、クラスが保有するポインタにも当てはまるということに当初は気がつかなかった。

プログラマが「アドレス共有の心配はない」ということをコンパイラに指示すれば上記の問題は解決でき、その方法は手引書[4]に示されている。コンパイラオプション-pvctl,nodep と-Orestrict=value である。前者は繰り返しループにおいてデータ依存関係がないこと、後者は value の値によってアドレス共有がない範囲をプログラマが保証する。筆者の場合、アドレスを共有するポインタが一切ないので、-Orestrict=all を使用した。（次節で示すように、実際はアドレスを共有するポインタを作っている。しかしそれらのポインタがループ内で同時に現れることがないため、-Orestrict=all で問題ない。）

2.4 ベクトル長を長くするための工夫

オプション-Orestrict=all により多くのループが自動ベクトル化された。2 次元流れのプログラムなので、その時点での筆者のプログラムは各次元でループを持ち、2 重ループになっていた。この場合、ベクトル化は内側のループにのみ適用される。内側のループの繰り返し数（=対応する方向の格子点数）が小さい場合、ベクトルレジスタを持て余して非効率である。手引書[4]を見ると、最大ベクトルレジスタ長は 256 と書いてある。今回の計算では、256 回よりもかなり小さい繰り返しになることもある。そこでループを一重化し、レジスタを効率よく活用することを考えた。ループの一重化はコンパイラオプション-pvctl,collapse によってもできるが、思わぬ副作用が起こっても困るので、1 次元配列に 2 次元データを直線的に格納し、さらにループが一重になるようにプログラムを書き換えた。ただし、以下のような工夫を行い、必要に応じて 2 次元配列にもなり得る配列を使用した。

具体的には以下の 4 つのマクロを作成した。マクロ NEW_1D_ARRAY はメモリ確保失敗時に警告を表示する。宣言時にポインタを 0 で初期化しておけば、DELETE_1D_ARRAY によって不用意な解放を行うリスクを避けることができる。これらはいずれも補助的なものであり、2 次元計算で用いる配列の確保および解放はマクロ NEW_2D_ARRAY_AS_1D と DELETE_2D_ARRAY_AS_1D が行う。

```
#define NEW_1D_ARRAY(/* 要素の型 */ type, ¥
    /* ポインタ */ ptr, /* 要素数 */ imax) ¥
    if(!(ptr = new(std::nothrow) ¥
    type[imax])) { ¥
```

```
        printf("%s (%u): ", ¥
        __FILE__, __LINE__); ¥
        perror(NULL); ¥
        exit(EXIT_FAILURE); ¥
    }    // ptr[0], …, ptr[imax - 1]でアクセス
         // #include<new>が必要
#define DELETE_1D_ARRAY(/* ポインタ */ ptr) ¥
    if(ptr) { ¥
        delete[] ptr; ¥
        ptr = 0; ¥
    }    // NEW_1D_ARRAY で確保したメモリの解放
#define NEW_2D_ARRAY_AS_1D( ¥
    /* 要素の型 */ type, ¥
    /* 2 次元配列としてのポインタ */ ptr2D, ¥
    /* 1 次元配列としてのポインタ */ ptr1D, ¥
    /* 第 1 次元方向要素数 */ imax, ¥
    /* 第 2 次元方向要素数 */ jmax, ¥
    /* 作業整数 */ j) { ¥
    NEW_1D_ARRAY(type, ptr1D, (imax)*(jmax)); ¥
    NEW_1D_ARRAY(type *, ptr2D, jmax); ¥
    for(j = 0; j < (jmax); j++) { ¥
        ptr2D[i] = ptr1D + j*(imax); ¥
    } ¥
}    // ptr2D[0][0], …,
     // pt2Dr[jmax - 1][imax - 1]でアクセス,
     // ptr[j][i] = ptr1D[j*imax + i]
#define DELETE_2D_ARRAY_AS_1D( ¥
    /* 2 次元配列としてのポインタ */ ptr2D, ¥
    /* 1 次元配列としてのポインタ */ptr1D) { ¥
    DELETE_1D_ARRAY(ptr1D); ¥
    DELETE_1D_ARRAY(ptr2D); ¥
}    // NEW_2D_ARRAY_AS_1D で確保した
     // メモリの解放
```

計算空間の第 1, 2 次元方向の格子点数をそれぞれ imax, jmax とし、格子点ごとの物理量を double 型で格納する配列を動的に確保、解放する場合を例としてマクロの使用法を述べる。配列には 1 および 2 次元配列の 2 つの表現があり、それぞれのポインタを a1D、a2D とする。まず

```
double *a1D = 0, **a2D = 0;
int j;    // 作業整数
NEW_2D_ARRAY_AS_1D(double, a2D, a1D, imax, jmax, j);
```

によってポインタ宣言および配列を動的に確保する。実際に double 型の値を格納するのは a1D に割り当てられた 1 次元配列であり、その要素数は imax*jmax である。これに格子点ごとの物理量が直線的に格納される。具体的に述べると、計算空間内の任意の格子点の第 1、2 次元のアドレスをそれぞれ i (= 0 ~ imax - 1)、j (= 0 ~ jmax - 1)とすれば、その格子点での物理量を a1D[j*imax + i]に格納するようにする。a1D を一重化ループで使用すれば繰り返し数は imax*jmax になり、imax または jmax で繰り返すよりもベクトルレジスタを有効に使える。a1D は単調な繰り返しになるループ、特に差分計算のループで用いる。差分でよく用いられる a1D[k]（k はループ内で用いるインデックス）に隣接する要素にアクセスする場合、第 1、2 次元方向でそれぞれ a1D[k±1]、a1D[k±imax]を利用すれば良いだけであり、コードの可読性を落とすこともない。

ただし境界近傍でステンシル構成格子点を変化させたり、境界条件で値を代入したりする場合で a1D を使っているとコードの可読性が落ちる。そこで用いられるのが配列 a2D である。a2D には jmax 個の要素をもつポインタ配列が割り当てられ、各要素が a1D で i = 0 になる場所を指すようにマクロ内の for 文で指定している。これにより、a1D[j*imax + i] = a2D[j][i]の関係を満足させる。a2D を用いれば、例えば i = imax - 1 の点は a2D[j][imax - 1]と表記でき、a1D[j*imax + imax - 1]と書くよりも可読性が高い。C 言語の 1 次元配列の定義どおりに第 1 次元方向のインデックスを最後にしている、つまり a2D [i][j]ではなく a2D [j][i]でアクセスすることに注意が必要かもしれない。境界条件などは適用させる格子点数が差分に比べて小さいため、ループをベクトル化してもそれほど効率は上がらない。ならば可読性を優先させた方が得策と考えた訳である。なお、メモリの解放では以下のように書けばよい。

```
DELETE_2D_ARRAY_AS_1D(a1D, a2D);
```

以上のアイディアは実は筆者オリジナルのものではない。学生時代にご指導いただいた現青山学院大の横田和彦先生から FORTRAN77 で書かれた圧縮性流体の数値計算プログラムをいただいたことがある。そのプログラムにこのアイディアが盛り込まれていた。ただし、実装には EQUIVALENCE 文（メモリ共有を指定する）が使用されていた。具体的には以下のように書くだけでよい。

```
DIMENSION A1D(IMAX*JMAX), A2D(IMAX, JMAX)
EQUIVALENCE(A1D, A2D)
```

FORTRAN の場合、2 次元配列の構成要素は DO 型並びで途切れなくメモリ上に並ぶことが保証されている。これと 1 次元配列でメモリを共有することにより、今の例では A1D((J - 1)*IMAX + I) = A2D(I, J) の関係を持たせる（FORTRAN ではインデックスが 1 から始まるため、表現が多少異なる）。この機能を C++で実現させるために先に示したマクロを作成した。

2.5 ISNAN、ISINF の取り扱い

前説のマクロを使った配列を用いて一重化ループに書き直し、コンパイルしたところ、それらのループは 1 つを除いて全て自動ベクトル化された。ベクトル化されない部分のメッセージを見ると、

```
Vectorization obstructive procedure reference.: _Dtest
```

と表示されていた。_Dtest という手続きが何なのか分からなかったが、ベクトル化できたループとそのループが異なるのはループ内でマクロ isnan と isinf を使っていたことであった。これらは計算の発散を検出するために使用している。実際にこれらのマクロをコメントアウトすると自動ベクトル化できた。しかし、これらのマクロの機能をどうしても使いたかった。というのは、計算が発散したときの対策は決まっており、時間ステップを小さくして再計算するしかない。ならばプログラムが自動で行った方が計算機の稼働率が高くできるからである。そこで自動ベクトル化を阻害せずに isnan や isinf と同じ機能を持つ以下の 2 つのマクロを作成した。

```
#define ISNAN(x) ((x) != (x))
#define ISINF(x) ((x) == INFINITY || ¥
  (x) == -INFINITY)  // #include<math.h>が必要
```

マクロ ISINF(x)は単に等価演算子によって x が±INFINITY に等しい時は非 0、それ以外は 0 を返すだけである。これと同じ方法で ISNAN(x)を定義すると、実際に x が NAN であっても 0 を返してしまい、上手く行かない。そこで上のように定義した。これは NAN は NAN 自身と比較しても等しくならない性質を利用したものである。これらを組み込んでテストしたところ自動ベクトル化され、なおかつ問題なく動作した。

2.6 最終的に利用しているコンパイルオプション

これまでの改良を経て、筆者が最終的に使用するようになったコンパイルオプションを以下に示す。

```
> sxc++ -K exceptions -pi auto -O restrict=all -R summary diaglist fmtlist -P auto -pvctl fullmsg ソースファイル.cpp（または c） …
```

自動ベクトル化はできないが自動並列化ならば可能という部分もあったため、オプション-P auto も使用している。また、簡易解析機能（どの関数がどのくらい時間を消費しているかなどの情報を得ることができる。手引書[4]210-222 ページに記載）を使用するときには上に加えて-ftrace オプションを使用する。これらオプションの説明を手引書[4]から抜粋して表 1 に示す。

2.7 実行結果

プログラム実行後に標準エラー出力ファイルに実行結果が出力される。ようやくコンパイルが通った時点でのコード（改良前）とこれまでに述べた改良を行った後のコード（改良後）で同一の計算を実行してみた。実行結果のうち、ベクトル計算に関するものを表 2 に示す。改良前と比べると、計算にかかる時間は約 1/13 になり、2.4 節で述べた方法により平均ベクトル長が 4 から 102 に増加した。ベクトル

表 1：コンパイラオプション（手引書[4]より抜粋）

オプション	説明
-K exceptions	例外処理機能の使用を許可
-pi auto	自動インライン展開機能を使用
-O restrict=all	全てのポインタ、リファレンスが restrict 修飾されたものとして、最適化、自動ベクトル化、自動並列化を適用
-R summary diaglist fmtlist	サマリリスト、診断メッセージリスト、編集リストを出力
-P auto	自動並列化機能を使用
-pvctl fullmsg	ループに関する詳細なベクトル化/並列化診断メッセージを出力
-ftrace	簡易性能解析機能(ftrace 機能)により性能解析できるオブジェクトファイル、実行ファイルを生成

表 2：実行結果

	改良前	改良後
Real Time（経過時間）(sec)	1092	82.14
Vector Time（ベクトル命令実行時間）(sec)	54.87	60.33
Inst. Count（全命令実行数）（= ex）	5.632×10^{11}	1.176×10^{10}
V. Inst. Count（ベクトル命令実行数）（= vx）	1.255×10^{9}	2.812×10^{9}
V. Element Count（ベクトル命令実行要素数）（= ve）	5.136×10^{9}	2.876×10^{11}
A. V. Length（平均ベクトル長）（= ve/vx）	4.093	102.3
V. Op. Ratio（ベクトル演算率）（= 100*ve/(ex - vx + ve)）(%)	0.906	96.98

演算率も 1%から 97%に増加させることができ、アムダールの法則[5, 6]により期待したベクトル化性能が得られるとされる 95%に近い値になった。

3. おわりに

C++で書かれた 2 次元圧縮性流体の差分コードを SX-8R 向けに改良し、可能な限り多くの自動ベクトル化を達成できた。本稿ではその際の注意点やテクニックを紹介した。このコードを用いて、今後も多くのパターンの超球幾何形状中の爆縮計算を行い、その特性を明らかにしていきたい。

今回紹介したテクニックは、それほど難しいものではない。また、2.4 節で述べた方法は OpenMP によるループ並列化（ループの手前に#pragma omp parallel for を書き込む）にも有効である（持て余すスレッドが少なくなる）。現在、パーソナルワークステーション（1 年前購入、100 万円程度）でも計算を行っているが、g++と OpenMP によって SX-8R と同程度の性能を引き出すことができる。次期 SX-Ace の性能に期待したい。

参考文献

(1) Landau, L. D., Lifshitz, E. M., Fluid Mechanics 2nd Ed., Elsevier, pp. 406-411, (1987).

(2) Stuart E. Rogers, Dochan Kwak, AIAA Journal, 29-4, pp. 603-610, (1991).

(3) Murakami, M., Sanz, J., Iwamoto, Y., EPL, 100, 24004, 6 pp. (2012).

(4) 日本電気株式会社, SX システムソフトウェア C++/SX プログラミングの手引（G1AF28-13）, 入手先<https://portal.hpc.cmc.osaka-u.ac.jp/secure/manual/R18.1J/g1af28j/g1af28-13.pdf>, (参照日 2014 年 4 月 27 日), (1999).

(5) 日本電気株式会社,スーパーコンピュータシステムを利用した例題によるベクトル化・並列化入門, 入手先<http://www.hpc.cmc.osaka-u.ac.jp/j/tebiki/sx-vecpara.pdf>, (参照日 2014 年 4 月 27 日), (2007).

(6) 増田耕一, 平田和久, “SX4 ベクトル化日記”, 神戸大学情報基盤センター広報誌 MAGE, 27-19, 入手先<http://www.istc.kobe-u.ac.jp/activity/mage/m27/27_SX4.pdf>, (参照日 2014 年 4 月 27 日), (1998).