SR11000を使い始めた人のための コンパイルのちょっとしたコツ

佐藤周行

スーパーコンピューティング研究部門

概 要

この原稿の（実質的な点は）次にまとめられます。

- いじるのはプログラムではなくてコンパイラオプションのほう。
- コンパイラの頭はあまりよくないことは事実だが、人間ほどではない。
- プログラムは素直に書くほうが結局は速い (下手の考え休むに似たり)。

1 はじめに

この原稿は，SR11000 を使い始めた利用者が，東大センターでの標準コンパイラである日立のコンパイラをどう利用するかについて，少しノウハウを伝えられたらなぁというものです。今回は Fortran を取り上げますが，C でも状況は（基本的に）変わりません。

ん？わかってますって。常日頃から，どういじってもキャッシュの再利用なんか土台無理なプログラムの改良に苦労しているのに，行列積なんていうやさしい問題を例にとって「ほら，こんなにうまくいくでしょう」なんていうのは詐欺だって。そういう人たちにとって，この稿はあまり参考にならないかもしれません。でも，コンパイラオプションが何を制御して，どのように性能に影響するかをながめるには少しは参考になるかもしれません。また，性能を上げようと思ったらプログラムをいじらないといけない GNU のコンパイラと違って，「コンパイラオプションをいじる」ことで性能を上げるのが日立コンパイラのコツだということを理解する良い機会かもしれません。

また，この稿は性能向上の王道であるプロファイリングについてふれていません。大きな問題ですから，稿をあらためてふれることにします。

2 日立 Fortran コンパイラ

ここで相手に選ぶのは日立最適化 Fortran90 コンパイラです。名前のとおりいわゆる Fortran90 とよばれる規格に準拠していますが，Fortran95 にも対応しています[1] 。このコンパイラは，命令スケジューリングやキャッシュ，プリフェッチなどの CPU に特化した最適化を行なうのはもちろん，1980 年代から連綿と続いてきたループの再構成による最適化（当時はベクトル化を対象としていましたが）の並列アーキテクチャやキャッシュアーキテクチャ

[1] ちなみに最新の規格は Fortran2003 です。これを実装したコンパイラは世界中探してもまだありません。規格が決まって (2004 年) から 3 年目だというのに…

への適用が大々的に行なわれています。また，高速ライブラリを用意し，イディオム認識によってあるコード部分をそのままごっそり置換するような最適化も平気で行ないます。

並列化に関する能力を見るのは別の機会に譲り，今回は，基本的な最適化の性能を見てみることにしましょう。

2.1 最適化オプション

最適化オプションは基本的に以下の 3 つです。マニュアルに書いてある説明をつけておきます。

-O3	プログラム全体で大域的な最適化をする
-O4	-O3 に加えて制御構造の変換，演算順序の変更などを含め，プログラム全体を最適化する。
-Os	実行速度が速くなるように，次に示すコンパイルオプションを自動的に設定する。 -nolcheck -nodochk -approx -noconvcheck -disbracket -divmove -expmove -fma -invariant_if -ischedule=3 -loopdistribute -loopexpand -loopfuse -loopinterchange -loopreroll -O4 -parallel=2 -prefetch -prod -rapidcall -scope -swpl -noargchk -nobreak -noerstmt -noagochk

そのほかに最強の最適化オプション-Oss というのもありますが，今回は使いません。最適化オプションを指定しなければ -O3 が適用されます。最適化を一切抑止する-O0 というオプションもありますが，特別な理由がないかぎり使うことはないでしょう。

ある特定の最適化オプションを抑止したい場合はno を先頭につけます。たとえば-loopfuse を無効にしたい場合は-noloopfuse とします。

筆者の感触では，-O3 はコンパイラの教科書[2] に書いてあるような最適化がなされます。-O4 でもいわゆる「普通」にがんばっているコンパイラなら実装されているようなものです（それでも出てくるコードはだいぶ上質です）。

日立コンパイラの日立コンパイラたる所以は-Os の最適化の凄さにあるといっても過言ではないでしょう。日立（に限らず，いわゆる国産 3 社）は昔からコンパイラを作ってきて，ベクトルコンピュータの時代は製品としては世界一の品質を誇ってきたわけですが，その技術の歴史が全部詰め込まれている感じがします。

ここでは，ループの再構成（交換，分散，融合，タイリングなど）[3] をはじめとした，いわゆる「ソースプログラムの書き換え」レベルの最適化が適用されます。どれをどのくらい適用するかについてはコンパイラが判断します。そういうわけで

1. いじるのはプログラムではなくてコンパイラオプションのほう。

一般論ですが，この「判断」はコンパイラに任せるのが賢明です。コンパイラも時々判断ミスをすることがありますが，

2. コンパイラの頭はあまりよくないことは事実だが、人間ほどではない。

[2] これにはまるとコンピュータサイエンティストになっちゃいますから，計算科学者は読んではいけません（？）。

[3] ちなみに，ループ最適化として皆がすぐ思いつくループアンローリングは再構成には一般に含めません。適用される理論が違うからです。

2.2 最適化のリストの表示

ある特定の最適化オプションを指定したときに，どの最適化がどう適用されるか実感がわかないかもしれません。そういう時は-loglistを指定しましょう。たとえば次のようにします。

```
% f90 -Os -nodisbracket -noparallel -loglist m.f
...
```

上の例では，リストがm.logに出力されます。出力の形式はたとえば次のようになります。**に続いて，コメントが挿入されます。

```
      subroutine mmjki(a,b,c,n)
       implicit none
       integer n,i,j,k
       double precision a(n,n),b(n,n),c(n,n)
**
**    Outer loop unrolled (2 times).
**
       do j = 1,n
**
**    IF test is invarient in loop so moved to outside.
**    Outer loop unrolled (4 times).
**
         do k = 1,n
**
**    SWPL applied.
**    6 streams (A[]x4, C[]x2) pre-fetch applied.
**
             do i = 1,n
                c(i,j) = c(i,j) + a(i,k) * b(k,j)
             end do
          end do
       end do
       end
```

もっと詳細に見たいという方にはアセンブリコードを直接見るのが一つの方法です。オプションは-S。

```
% f90 -Os -S -loglist m.f
...
```

出力は上の場合，m.sに出てきます。アセンブリコードを読む必要のある場合が存在することは否定しませんが，これにはまると，生産性その他大きな問題が生じます。最後の手段にしておきましょう。

3　実験

与太話をいくらしても実感がわきませんから，具体的に最適化オプションをいろいろ変えてそれが性能にどのくらい関係するかを実感してみましょう。

実験に使うのは行列積です。行列積はLU分解とともに，最適化の余地が大きく，かつプログラムの構造が単純で，数値計算の素養がない人にも理解しやすいので，最適化の効果を見せるための格好のターゲットになってきました[4] 。

3.1　実験環境

実験は2007年7月17日に行ないました。日立Fortranコンパイラのバージョンは V01-05-/C で，Interactive環境で実行しました。

interactive環境では、他のユーザとCPUをシェアしていますから、いろいろ邪魔が入ります。センターのバッチ環境では、OSの割り込みが入る可能性はありますが、ほぼ安定した環境でプログラムが実行できると思ってよいでしょう。その意味で、interactive環境で行なった今回の実験は少し割り引いて考えてください。

3.2　実験プログラム

実験プログラムとしていずれも行列積を計算する以下のものを用意しました。表1にあげておきます。

解説

IJKは、定義に素直にプログラムを書いたものです。JKIは、データアクセスの足跡を考えれば、一番効率的だろうと思われているものです[5] 。IJKTは、行列積の改善のときに必ず問題になるAのデータの再利用を、ループの入れ替えではなくて、必要なデータをあらかじめ集めることで改善しました。IJKTTは、それを一歩進めて、行列を転置してしまったものです。作業用に余分な配列が必要になります。最後にINTRINSICはFortran90の組み込み関数matmulを使いました。

3.3　実験結果

行列のサイズを200と2000にし，最適化オプションを4種類用意しました。結果をMFLOPSを単位として表2に書き出します。INTRINSICについては，一度$2 \cdot n^3$の掛け算をしたあとにn^2の足し算をしているようにみえますが，やっていることは他のルーチンと同じですから，これも計算量は$2 \cdot n^3$とすることにします。

[4] もちろん，現実のプログラムの中に頻繁に現れることも理由のひとつです。

[5] なぜ？この説明を講義その他で今までされたことがない人は，これを確認しましょう。ついでに，データの格納方法についてのCとFortranの違いを一度確認しておきましょう。ちなみにCでは最適なパターンがIKJになります。column majorとrow majorというのがキーワードです。

表 1: 実験に使ったプログラム

IJK:

```
do i = 1,n
  do j = 1,n
    do k = 1,n
       c(i,j)=c(i,j)+a(i,k)*b(k,j)
    end do
  end do
end do
```

JKI:

```
do j = 1,n
  do k = 1,n
    do i = 1,n
       c(i,j)=c(i,j)+a(i,k)*b(k,j)
    end do
  end do
end do
```

IJKT:

```
do i = 1,n
  d = a(i,:)
  do j = 1,n
    do k = 1,n
       c(i,j)=c(i,j)+d(k)*b(k,j)
    end do
  end do
end do
```

IJKTT:

```
d = transpose(a)
do i = 1,n
  do j = 1,n
    do k = 1,n
       c(i,j)=c(i,j)+d(k,i)*b(k,j)
    end do
  end do
end do
```

INTRINSIC:

```
c = c+matmul(a,b)
```

表 2: 実験結果

n=200: (MFLOPS)

オプション	IJK	JKI	IJKT	IJKTT	Intrinsic
`-O3 -noparallel`	758.8	1250.5	757.0	748.1	216.7
`-O4 -noparallel`	790.1	2344.6	788.8	778.7	755.1
`-Os -nodisbracket -noparallel`	5110.5	5123.4	2818.2	5062.7	1117.9
`-Os -noparallel`	6940.7	6949.6	4305.7	5065.9	6147.3

n=2000: (MFLOPS)

オプション	IJK	JKI	IJKT	IJKTT	Intrinsic
`-O3 -noparallel`	141.1	902.1	607.5	622.9	134.4
`-O4 -noparallel`	199.4	1243.0	628.5	641.6	198.6
`-Os -nodisbracket -noparallel`	3333.1	3333.3	1480.1	3834.5	295.5
`-Os -noparallel`	6116.5	6116.5	1660.0	3819.4	5998.1

3.4 観察

まず観察すべきは，最適化オプションで性能が劇的に異なるということです。キャッシュが効く（サイズ 200）と，一桁ですみますが[6] ，二桁になるとそれは大問題でしょう。

さてと，行列のサイズが 200 ではキャッシュの効果が如実に現れますからデータアクセスのコストをあまり考えない最適化で大丈夫です。ただ，2000 になるとキャッシュがあふれてしまいますから，いわゆる「キャッシュ最適化」をきちんとしないと性能が出ません[7] 。

さて，見るべきポイントは 200 と 2000 の差だけではありません。つっこみどころは実は満載なんですが，ここではポイントを 3 つにしぼりましょう。

1. データアクセスのコストを考えた最適化が必須で，それは最適化レベルを上げないときには有効。
2. IJK と JKI の差は，最適化のレベルを上げると消えてしまう。
3. `-Os` にすると，結局同じコードになってしまう。

`-O3` では，m.log には何も出てきません。`-O4` になるとソフトウェアパイプライニングが適用されるというメッセージが出てきます。この効果は数割にとどまるというのは表を見ればよくわかります。`-Os -nodisbracket` にすると，日立コンパイラの頭の中がだんだん透けて見えるようになってきます。リストで確認してみましょう。

```
       subroutine mmijk(a,b,c,n)
       implicit none
       integer n,i,j,k
       double precision a(n,n),b(n,n),c(n,n)
**
**    SWPL applied.
**    6 streams (A[]x4, C[]x2) pre-fetch applied.
**
       do i = 1,n
**
**    Loop interchanging (I,J,K)->(J,K,I) applied.
**    Outer loop unrolled (2 times).
**
         do j = 1,n
**
**    IF test is invarient in loop so moved to outside.
**    Outer loop unrolled (4 times).
**
             do k = 1,n
```

[6] 原稿の都合上，こう書きましたが，性能が 7 倍くらいちがうと，1 日で終わるジョブが 1 週間流れるということになります。この感覚は無視すべきではないと考えます。

[7] だからわかってますって。みなさんのプログラムがキャッシュ最適化の恩恵を必ずしも受けない，受けようと思ったらアルゴリズムからの再検討が必要になるということは。キャッシュ最適化がすべてを解決するかのような幻想を与えるつもりはありません。この意味で，今もってベクトル計算機を作り続けている会社，またその利用者からの恒常的な批判は十分意識する必要があります。

```
            c(i,j) = c(i,j) + a(i,k) * b(k,j)
         end do
      end do
   end do
```

このリストと，-loglist の例として出したリストの違いがループの交換だけというのもついでに確認しておきましょう。

一般に-O3 や-O4 では，メモリ，キャッシュ，レジスタの階層を持つデータアクセスのコストを考慮した最適化がなされないようです。性能を出そうと思ったら-Os が必須です。これに背を向けてたとえばIJKT とかIJKTT などのように自分でデータアクセスを最適化する手もあります。これもある程度効果がありますが，結局，-Os -nodisbracket 以上になると投資の効果が消えてくるということでしょうか。

投資の効果が消えるということでは，IJK とJKI の差の解消も見逃せません。ちょっと前に「JKI がなぜ速いか理解せよ」と書きましたが，日立コンパイラはここらへんも解析して最適のデータアクセスのパターンを選択してくれます。このご利益が実感できるのも-Os 以上になります。

ちなみに-nodisbracket の有無で出てくるコードに差があるのは，日立の親切によるもので演算順序の違いを（規格上無視するのが許されているのに）利用者のコードに書かれていた意思を尊重することによるものです。-nodisbracket をつけずに-Os にすると，問答無用で日立の開発した高速行列積ライブラリが呼び出されます。これをリストで観察しましょう。

-Os でのリスト:

```
**
**     行列積ライブラリ化 (_hf_mat_matmul).
**
       do j = 1,n
          do k = 1,n
             do i = 1,n
                c(i,j) = c(i,j) + a(i,k) * b(k,j)
             end do
          end do
       end do
```

本当は5 種類のコードが全部置き換えられなければならないのですが[8]，それでもIJK, JKI，Intrinsics が同じコードに落ち着きました[9]。問題は，とても速いコードに置き換えるためには，それなりにプログラムをシンプルに保たなくてはいけないということです。IJKT やIJKTT のように,「変な」工夫をすると，コンパイラがその意図を見つけきれないことがおきます。そこで

3. プログラムは素直に書くほうが結局は速い (下手の考え休むに似たり)。

ここではふれませんでしたが，並列化でも同じことが言えます。プログラムはできるだけシンプルにして保守するのがお勧めです。

[8] ここらへんができるという論文を昔書いたことがあるのですが，まあいいや。
[9] ただし，Intrinsic については，イディオム認識にもう少し改良の余地がありそうです。

4　おわりに

ここでは，行列積を例にとって日立のコンパイラがどのように最適化を行なっているかについて解説をしました。まとめは概要に書いた 3 点に集約されます。繰り返します。

- いじるのはプログラムではなくてコンパイラオプションのほう。
- コンパイラの頭はあまりよくないことは事実だが、人間ほどではない。
- プログラムは素直に書くほうが結局は速い (下手の考え休むに似たり)。

もっともこれはコンパイラの研究者にとってとても都合の良い結論ですから，計算科学者はあまり信用してはいけません。ただし，コンピュータサイエンティストから何かアドバイスがあったときにそれを受容するゆとりを持っていただければ，チューニングがさらに進むと思います。