3.2.4. Fortran90 による格子 QCD コードとその性能

広島大学　　中村　純

1. はじめに

Fortran90 は FORTRAN77 の後継であり、新しく導入された module 機能によりユーザーによるデータの自由な型とそれに対する演算の定義が可能となった。このことにより、可読性の向上、バグの可能性の低減、コーディングの容易さがもたらされた。大量の計算機リソースを必要とする格子 QCD 計算の分野では、新しい解析手法やアルゴリズムが次々と提案されるため、この読みやすくミスを起こしにくい Fortran90 のオブジェクト指向型コーディングは非常に重要である。しかし、得られたコードの性能が低くては科学技術計算用には問題である。

ここでは、Fortran90 によりどのようにコードが書かれるのか、またそのコードの高速化ではどのような点に配慮するべきかについて検討する。

```
TYPE(g_field1)   staple, temp1, temp2, temp3

c          x+nu temp2
c            .---------.
c            I         I
c   temp1 I            I
c            I         I
c            x        x+mu

             temp1 = u(nu)
             temp2 = nu.gshift.u(mu)
             temp3 = temp1 * temp2
             temp1 = mu.gshift.u(nu)
             staple = staple + (fac*(temp3.prodAD.temp1))
```

ここで、TYPE(g_field1) はあらかじめ以下のように定義されている。

```
TYPE g_field1
    SEQUENCE
    COMPLEX*16, DIMENSION(NC,NC,NV/NBUSH) :: g
    INTEGER parity, direction
END TYPE
```

また上記のプログラム中の演算（=, +, .gshift. , .prodAD）も同様にモジュール中で定義されている。例えば加法は、以下のように演算子「+」に関数 gadd を対応させて定義されている。

```
....
INTERFACE OPERATOR(+)
    MODULE PROCEDURE gadd
END INTERFACE
....
c-----------------------------------------------------------------c
      FUNCTION gadd(a,b) RESULT(c)
c-----------------------------------------------------------------c
      TYPE(g_field1), INTENT(IN):: a, b
      TYPE(g_field1) c

      do i = 1, NV/NBUSH

          c%g( 1, 1, i) = a%g( 1, 1, i) + b%g( 1, 1, i)
          c%g( 1, 2, i) = a%g( 1, 2, i) + b%g( 1, 2, i)
          c%g( 1, 3, i) = a%g( 1, 3, i) + b%g( 1, 3, i)
      .........

      enddo

      END FUNCTION
```

これらのモジュールで定義された演算も、コンパイラーが計算式の中の変数の型で演算を区別することによって、通常の演算同様適切に行われる．一度演算を定義すると、上記のように通常の演算と同じようにプログラム中で使用することができるので，紙の上での式とコード中の式の距離が短い。

上のコードは、かつては以下のような形で書かれていた。

```
call   movsites(jd,nvol,site(1,1),site(1,3))

call   getlinks(site(1,1),temp2,jd,nvol)
call   getlinks(site(1,3),temp3,id,nvol)
call   prodlink(temp2,temp3,temp4,nvol,1)

call   getlinks(site(1,2),temp2,jd,nvol)
call   prodlink(temp4,temp2,temp3,nvol,3)
call   prodlink(temp1,temp3,temp2,nvol,3)

call   addlink(plaq,temp2,plaq,nvol)
```

これと比較すると module のありがたみが分かる。

2. Fortran90 と計算スピード

Fortran を使う以上、計算スピードが高いことを期待するのは当然であろう。しかし、これまで国産コンパイラーで疑問に感じることが何度かあった。

- SR8800 において、module で定義した変数が引数に現れるとサブルーティンコールに異常に時間のかかることがあった。
- SX-5 において、module の演算を定義する関数の中の DO-ループは自動並列化の対象にならない。
- Fujitsu コンパイラーでは module で定義した演算は遅いのでは。

3. QCD と HPC

QCD(Quantum Chromodynamics, 量子色力学)の数値シミュレーションは大きな計算機リソースを必要とする。現在、その大部分は大規模疎行列の逆を求めるための計算で CG(Conjugate Gradient, 共役勾配法)系のアルゴリズムが使われている。CG 法は線形方程式

A X = b

の解法であるが、行列とベクトルの積 Y=A X とベクトルの内積 <X|Y>、それにベクトルの和とスカラー倍で表され、A が大規模疎行列の時に強力な手法となる。計算リソースという観点からは行列とベクトルの積が高速化されればよい。

3.1. QCD に現れる行列の形

行列 D を格子 QCD の業界用語で書くと

$$D = I - \kappa \sum_{\mu=1}^{4} (r - \gamma_\mu) U_\mu(x) \delta_{x',x+\hat{\mu}} + (r + \gamma_\mu) U_\mu^\dagger(x') \delta_{x',x-\hat{\mu}}$$

κ ： ホッピング・パラメータ
r ： Wilson 項
γ_μ ： Dirac のガンマ行列(4x4)
$\hat{\mu}$ ： 格子間隔の大きさを持った mu 方向の 4 次元ベクトル
$U_\mu(x)$ ： SU(3) 行列
$U_\mu(x)\dagger$ ： エルミート共役行列（複素・転置）

なので $\vec{Y} = D\vec{X}$ は

$$Y(x) = X(x) - \kappa \sum_{\mu=1}^{4} \left\{ (r-\gamma_\mu) U_\mu(x) X(x+\hat{\mu}) + (r+\gamma_\mu) U_\mu^\dagger(x-\hat{\mu}) X(x-\hat{\mu}) \right\}$$

となる。もう少し普通の人に分かるような表現は、フェルミオン行列は、カラー、Dirac、座標の添え字を顕わに書いて、以下となる。

$$D_{\alpha\beta}^{ab}(x,x') = \delta_{ab}\delta_{\alpha\beta}\delta_{x,x'} - \kappa \sum_{\mu=1}^{4} \left\{ (r-\gamma_\mu)_{\alpha\beta} U_\mu(x)^{ab} \delta_{x',x+\hat{\mu}} + (r+\gamma_\mu)_{\alpha\beta} U_\mu^\dagger(x')^{ab} \delta_{x',x-\hat{\mu}} \right\}$$

ここで $x \pm \hat{\mu}$ は、μ(=1,2,3,4) 方向の隣の格子点。

$\vec{Y} = D\vec{X}$ は成分を顕わに書けば、以下となる。

$$Y_\alpha^a(x) = X_\alpha^a(x) - \kappa \sum_{\mu=1}^{4} \sum_{\beta=1}^{4} \sum_{b=1}^{3} \left\{ (r-\gamma_\mu)_{\alpha\beta} U_\mu(x)^{ab} X_\beta^b(x+\hat{\mu}) + (r+\gamma_\mu)_{\alpha\beta} U_\mu^\dagger(x-\hat{\mu})^{ab} X_\beta^b(x-\hat{\mu}) \right\}$$

4. 理研でのチューニング

理研の RSCC を使わせていただくときに、チューニングを依頼した。青山幸也氏が中村の計算機コードを解析し(2005 年 10 月)、主なアドバイスは以下の 2 点であった。この結果、小規模な問題で 6.56 秒が 1.03 秒に短縮された。

- キャッシュミスの可能性を低めるために

```
do ic = 1, Nc
   do it = 1, Nt
   do iz = 1, Nz
   do iy = 1, Ny
   do ix = 1, Nx
      b%f(ic,ix,iy,iz,it,1) = ....
   enddo
   enddo
   enddo
   enddo
enddo
```

のループの順番を変更し

```
do it = 1, Nt
do iz = 1, Nz
do iy = 1, Ny
do ix = 1, Nx
   do ic = 1, Nc
      b%f(ic,ix,iy,iz,it,1) = ....
   enddo
enddo
enddo
enddo
enddo
```

とすること（但しコンパイラーが自動的に行う場合もある)、

- <u>module で定義される演算を subroutine で</u>書くこと

その後、コンパイラーのバージョンアップなどもあったはずなので、以下のように一部を module の演算から subroutine にして時間を測定したところ､行列とベクトルの積が 1.65 秒から 1.05 秒になったので、やはり module の演算は subroutine より遅い。

```
      ! ...   the iteration starts

      CALL clock(cpu1,0,2)

      do i = 1, imax

        ! ...   q = W * p
        if(iflag==1)   then
*           q = wxvect(p,2)
            CALL xwxvect(p,q,2)
        else if(iflag==2)   then
*           q = wxvect(p,3)
            CALL xwxvect(p,q,3)
        endif

        ......

      enddo

      CALL clock(cpu2,0,2)
      WRITE(*,*) "cpu: ", cpu2-cpu1

      ....

      END
c-----------------------------------------------------------------c
      SUBROUTINE xwxvect(x,wxvect,iflag)
c-----------------------------------------------------------------c
c     wxvect = x              for iflag=1
c                W*x                        2
c                (W_adj)*x                  3
c-----------------------------------------------------------------c

          ......

          do nu = 1,4

*         temp1 =     nu   .fshift. x
          CALL xfshift(nu,x,temp1)

*         temp2 = (-nu) .fshift. x
          CALL xfshift(-nu,x,temp2)

*         temp = temp + ((hopp(nu)*temp1) + (hopm(nu)*temp2))
          CALL xdmul(hopp(nu),temp1,temp1)
          CALL xdmul(hopp(nu),temp2,temp2)
          CALL xfadd(temp,temp1,temp)
          CALL xfadd(temp,temp2,temp)

          .....
```

5. 考察

関数の場合は、戻り値が一旦一時的な領域にコピーされるため、本コードのようにその本体が巨大な配列である場合はある程度はロスがあるのはやむを得ないと思われる（【補足資料】モジュール内関数とサブルーチンの性能差について 参照）。また、最適化が個々の演算の中で閉じてしまうので、

a = b + (s*c)

で　a, b, c が大きな配列、s がスカラーの場合など右辺全体に渡る最適化が難しくなる。

今後、Fortran90 の大きな利点である module による演算定義を、数値計算の高速化の阻害を最小限にするように行うことが重要である。このためには、いろいろなコーディング例が公開され、検討されることが望ましい。

以上

【補足資料】モジュール内関数とサブルーチンの性能差について

富士通株式会社　　鈴木 清文

「Fortran90 による格子 QCD コードとその性能」で報告されているモジュール内関数とサブルーチンの性能差について詳細を報告する。

1.　現象

モジュール内で定義された演算（利用者定義関数）をサブルーチン形式に書き換えると高速化される。以下に例を挙げる。

【構造型の定義】

```
      TYPE f_field
        SEQUENCE
        COMPLEX*16, DIMENSION(NC,0:NX+1,0:NY+1,0:NZ+1,0:NT+1,4) :: f
                        ! (Color, x,      y,      z,      t, Dirac)
      END TYPE
```

ここで、NC=3, NX=4, NY=4, NZ=4, NT=4。

【関数の型の定義】

```
      FUNCTION fadd(a,b) RESULT(c)
c-----------------------------------------------------------------c
      TYPE(f_field), INTENT(IN):: a, b
      TYPE(f_field) c
```

【サブルーチンのインターフェースの定義】

```
      SUBROUTINE Sub_fadd(a,b,c)
c-----------------------------------------------------------------c
      TYPE(f_field), INTENT(IN):: a, b
      TYPE(f_field), INTENT(OUT):: c
```

2.　関数の方が遅い原因

関数で書いた方が遅い原因は、「Fortran90 による格子 QCD コードとその性能」で指摘されている通り、関数内で関数結果を保持する領域（上記の例では RESULT の c）から関数を呼び出した手続き内の変数にその関数結果をコピーする処理がオーバーヘッドとして見えるためである。上記の f_field という構造型の場合、15552 要素=248832 バイトのコピーが関数から復帰する時に暗黙の内に実行される。

【呼出元手続きにおける処理】

```
～=fadd(～) ～～      →      関数結果域=fadd(～)   ★ここでコピー実行
                            ～=関数結果域 ～～

                    コンパイラは関数結果域を生成し、右のように命令を展開する
```

関数は式の途中に記述することができるため、関数結果を一旦受け取るための領域が必要となる。

サブルーチンの形で書かれた場合は、引数として渡された呼び出し元手続きの変数に直接計算結果を格納する形になるため、上記のオーバーヘッドがないことになる。

関数結果のコピーをコンパイラが暗黙の内に実行するという動作は関数の実行論理上必要なものである。関数結果として配列や配列を含む構造型の場合は、その大きさによりコピーのオーバーヘッドが目に見える形で現れてくるため、上記の特性を踏まえて使い分けるような対処が必要となる。

以上