# HPC2500 落穂拾い
## ～ HPC2500（スーパーコンピュータ）とうまく付き合うために～

津　田　知　子

## Ⅰ．はじめに

スーパーコンピュータ Fujitsu PRIMEPOWER HPC2500 が本センター導入されて 1 年が経ちました。ベクトル型の並列計算機 VPP5000 からスカラ型の並列計算機 HPC2500 への機種変更は，プログラムの処理性能を追求するユーザにとって，そのチューニング作業は，結構大変な作業となっています。エンドユーザにとっては，コンパイラの自動並列化や最適化などコンパイラの機能をフルに使うことで性能を出すことができれば，万々歳なのですが，そう簡単にことは済まないのが現実です。HPC2500 のいろいろなふるまいを知ることは，効率の良いプログラム作成の第一歩といえるでしょう。そこで，この 1 年，筆者の周りで目や耳にした HPC2500 の性能に関するふるまいをまとめてみました。

## Ⅱ．HPC2500 との付き合い方

**【その 1】可能なら単精度をつかうべし！――単精度は倍精度に比べて処理が速い――**

数値計算を行う場合，多くのプログラムでは，データは倍精度を使います。精度の差による処理性能を測定したところ，つぎのような結果が得られました。

(1) 配列の初期化処理

```
do j = 1,4000
   do i= 1,4000
      a(i,j)= 0.0
   end do
end do
```

| | 単精度 | 倍精度 |
|---|---|---|
| CPU 時間 | 0.094 秒 | 0.183 秒 |

(2) 配列演算処理（加算）

```
do j = 1,4096
   do i= 1,4096
      a(i,j)= a(i,j)+1.0D0
   end do
end do
```

| | 単精度 | 倍精度 |
|---|---|---|
| CPU 時間 | 0.095 秒 | 0.194 秒 |

これらの処理性能の差は，キャッシュミスによるもので，単精度のデータの方がキャッシュの利用効率が良くなることによるものです。もし，単精度で十分な結果が得られる計算であるなら，単精度で計算すべきです。スカラマシンでは，プログラムは何でもかんでも倍精度にしない方が得策のようです。

【その２】ラージページ機能は，すぐれもの！

ラージページ機能とは，メモリ管理の単位であるページサイズを通常の 8KB から 4MB に拡張し，さらに実行に先立ち固定的にメモリを確保するものです。この機能を使うことにより，アドレス変換やページングのオーバヘッドが削減でき，実効性能の向上が期待できます。本センターでは，hpc システムの Fortran，C，C++ の各コンパイラは，いずれも `largepage` オプションがデフォルトとなっています。

(1) ラージページ化でメモリアクセスの局所化をカバー

メモリアクセスの局所化を行うことで，プログラムの処理性能を上げることができる場合があります。これを loop tiling といいます。例えば，行列の転置のプログラム（A）に loop tiling を施すとプログラム（B）のようになります。

プログラム（A）

```
do i = 1,k
   do j = 1,k
      a(i,j)= b(j,i)
   end do
end do
```

プログラム（B）

```
do ii = 1,k,ll
   do jj = 1,k,ll
      do i=ii,min(ii+ll-1,k)
```

```
            do j=jj,min(jj+ll-1,k)
            a(i,j)= b(j,i)
            end do
            end do
          end do
        end do
```

これらのプログラムの処理時間は，つぎのようになりました。ここで，配列は倍精度を使用しています。

| | largepage | | nolargepage | |
|---|---|---|---|---|
| 行列のサイズ | プログラム（A） | プログラム（B） | プログラム（A） | プログラム（B） |
| 300 | 0.0007 | 0.0002 | 0.0044 | 0.0002 |
| 3000 | 0.1065 | 0.1252 | 0.9865 | 0.1430 |

単位：秒

この結果から，ラージページ機能を使えば，loop tiling のような面倒なことをしなくてもそれなりの性能が得られるということがわかります。特に，行列がキャッシュに載りきらないほど大きい場合には，ラージページ機能を用いて loop tiling しない方が処理性能は，わずかばかり良くなっています。

(2) perl プログラムもラージページ対応で速くなる！

perl の大容量のメモリを使用するプログラムで perl のモジュールをラージページ化することにより，実行速度が約 2 倍になる大幅な性能改善が得られました。

| | 経過時間 | User_CPU 時間 | system_CPU 時間 |
|---|---|---|---|
| perl プログラム通常実行 | 1133.92 | 519.37 | 585.54 |
| perl プログラムラージページ対応 | 564.52 | 555.74 | 7.04 |

単位：秒

＜ラージページ化対応の perl コマンドの使用方法＞

```
hpc% /usr/bin/sparcv9/lpgexec  /opt/local/cent/perl/bin/perl.lpg   perl-program
```

【その３】スタックサイズにご用心！

スレッド並列の実行では，スレッドごとにスタック領域が必要です。このスタック領域には，自動並列化された DO ループ内でのローカル変数が割り付けられます。この領域のサイズは，プロセスのスタックサイズと同じ値が採られます。hpc でのスタックサイズは，バッチジョブでは 128MB，TSS では，8MB が標準となっています。TSS に比べてバッチジョブのスタックサ

イズが大きく設定されているのは，ある種のアプリケーションの実行で必要なためです。ジョブの実行に必要なスレッドのスタック領域は，以下の式であらわされます。

スレッドのスタック領域　＝　スレッドごとのスタックサイズ×スレッド数

したがって，スレッド数をどんどん増やしていくと，ジョブの実行中に

「jwe0911i-u 領域が不足しているので，作業領域が獲得できません」

「mpc0911i-u A work area cannot be reserved because of insufficient area」

などのエラーが出力される場合があります。これは，スタック領域のための領域が不足したことが原因として考えられます。以下に示す方法で対処してください。

（対処法１）スタックサイズを 8MB にする。

```
limit stacksize 8M
./a.out
```

（対処法２）環境変数によりスレッドごとのスタックサイズを 8MB にする。環境変数 THREAD_STACK_SIZE の値は，K バイト単位で指定する。

```
setenv THREAD_STACK_SIZE  8192
./a.out
```

（対処法３）qsub コマンドの -lM オペランドでメモリサイズを大きく指定する。

```
# @$-q p128 -lp 100 -eo -o result -lM 20gb
./a.out
```

【その４】行列積の計算はライブラリに任せろ！

プログラム的には余りないことかもしれませんが，行列積の計算は，ライブラリ matmul に任せることにより，およそ１桁速く計算できます。

コンパイル時に

“jwd8331i-I "main.f", line 144: このループはライブラリ呼出し（matmul）に変換されました。”

のメッセージが出力されれば，その行列積は，ライブラリ matmul により計算されます。

測定結果を以下に示します。ここで，配列は，倍精度を用いています。

| 行列のサイズ | ライブラリ呼出し | ライブラリ呼出しなし |
|---|---|---|
| 1000 | 4.690 | 0.422 |
| 2000 | 4.710 | 0.423 |
| 3000 | 4.787 | 0.421 |

単位：Gflops

【その５】配列の添字入れ替えは行うべきか？

プログラムによっては，配列の参照がプログラム (A) で示すようにストライドアクセスになる場合があります。プログラム (B) のように配列の添字を入れ替えた一時配列を用いて配列の参照を連続アクセスとすることにより，処理性能の向上が期待できます。

プログラム (A)

```
do iter =1,loop
   do k =1,kk
     do j = 1,kk
       do i = 1,kk
         a(i,j,k)= a(i,j,k) + b(k,i,j) * c
       end do
     end do
   end do
  c = c + h
end do
```

プログラム (B)

```
 do j =1,kk
   do i = 1,kk
     do k = 1,kk
       bt(i,j,k)= b(k,i,j)
     end do
   end do
 end do
;
 do iter =1,loop
   do k =1,kk
     do j = 1,kk
       do i = 1,kk
         a(i,j,k)= a(i,j,k) + bt(i,j,k) * c
       end do
     end do
   end do
  c = c + h
end do
```

処理結果は，つぎのようになりました。

| 繰り返し回数 | プログラム (A) | プログラム (B) |
|---|---|---|
| 10 | 18.84 | 17.11 |
| 30 | 57.11 | 48.07 |
| 300 | 573.95 | 470.96 |

単位：秒

配列の添字の入れ替えを行うかどうかは，当然のことながらそれ以降の計算でその配列をどのくらい利用するかに依存します。上記のプログラムでは，繰り返し回数が 300 で，約 20％程度性能が向上しています。

【その６】配列から配列への代入はコスト高！

配列に対する初期化，代入，加算の時間を調べてみました。

* 初期化 *

```
do k =1,kk                    ! kk=512
  do j = 1,kk
     do i = 1,kk
        a(i,j,k)= 1.0
     end do
  end do
end do
```

* 代入 *

```
do k =1,kk
  do j = 1,kk
     do i = 1,kk
        b(i,j,k)= a(i,j,k)
     end do
  end do
end do
```

* 加算 *

```
 do k =1,kk
    do j = 1,kk
      do i = 1,kk
        temp= temp+a(i,j,k)
      end do
    end do
```

```
end do
```

| プログラム | 処理時間（秒） |
|---|---|
| 初期化 | 1.514 |
| 代　入 | 2.200 |
| 加　算 | 0.042 |

上記の結果から，配列から配列への単純な代入のコストが高いのがよくわかります。このことから，プログラミングに際しては，DO ループの中では，参照した配列要素（キャッシュに載っているものも含む）は，できるだけ使い回し，配列への代入は極力避けるようにできれば，性能改善につながります。とはいっても，そう都合よくはいかないものですが，このことを頭の隅に入れてプログラムを眺めてみることは，性能の観点からは無駄ではなさそうです。なお，HPC2500 のキャッシュの大きさは，１次キャッシュ：命令 128KB，データ 128KB，２次キャッシュ：4MB です。

### 【その７】ループの融合で性能アップ！

スカラマシンでは，キャッシュにあるデータをいかに有効利用するかが，性能向上の一つのポイントです。以下のプログラムでは，DO ループを融合することで，キャッシュにあるデータを使うことができるため，性能アップしています。処理時間を測定したところ，下表に示すようにいずれも 30%以上の性能改善が得られています。

プログラム (A)

```
do j = 2,nn-1
   do i= 2,nn-1
      a(i,j)= a(i+1,j)+alpha*b(i,j)
1             +b(i-1,j)+b(i,j-1)+b(i+1,j)+b(i,j+1)
   end do
end do

      ;

do j = 2,nn-1
   do i= 2,nn-1
      a(i,j)= a(i+2,j)+beta*b(i,j)
1             +b(i-1,j)+b(i,j-1)+b(i+1,j)+b(i,j+1)
   end do
end do
```

プログラム (B)

```
do j = 2,nn-1
   do i= 2,nn-1
      a(i,j)= a(i+1,j)+alpha*b(i,j)
1              +b(i-1,j)+b(i,j-1)+b(i+1,j)+b(i,j+1)
   end do
     ;
   do i= 2,nn-1
      a(i,j)= a(i+2,j)+alpha*b(i,j)
1             +b(i-1,j)+b(i,j-1)+b(i+1,j)+b(i,j+1)
   end do
end do
```

| 配列のサイズ | プログラム (A) | プログラム (B) |
|---|---|---|
| 3000 | 0.2047 | 0.1313 |
| 5000 | 0.5713 | 0.3960 |

単位：秒

また，つぎのプログラムでは，ループ融合することにより，約 40% も性能が改善されています。

プログラム (A)

```
do k = 1,nz                              ! nz =256
   do j = 1,ny                           ! ny =256
      do i = 1,nx                        ! nx =256
         u_work(i,j,k)= b(i)*u(i,j,k)
      end do
   end do
end do
do k = 1,nz
   do j=2,ny-1
      u_work(1,j,k) = u_work(1,j,k)*0.5
   end do
end do
do k = 1,nz
   do j = 2,ny-1
      do i = 1,nx-1
         d(i,j,k) = fn(u_work(i+1,j,k),u_work(i,j,k))
```

```
      end do
   end do
end do
```

プログラム (B)

```
do k = 1,nz
   do j = 2,ny-1
      temp(1) = b(1)* u(i,j,k)* 0.5d0
      do i = 2,nx
         temp(i) = b(i) * u(i,j,k)
      end do
      do i = 1,nx-1
         d(i,j,k) = fn(temp(i+1),temp(i))
      end do
   end do
end do
```

| プログラム (A) | プログラム (B) |
|---|---|
| 0.465 | 0.266 |

単位：秒

なお，上記プログラムは，本センターで行われた“新システム hpc のプログラミング講習会”で配布された資料「プログラムの性能向上手法」富士通株式会社（補足資料）から引用しました。

【その８】ループ分割で性能アップ！

前項では，ループ融合により性能向上している例を示しましたが，逆に，ループ分割することで，キャッシュ上のデータを有効に利用できる場合があります。下記のプログラムでは，配列のサイズが 10000 と大きい場合には，70％も性能向上しています。

プログラム (A)

```
do j = 2,nn
   do i= 1,nn
      b(i,j) = a(i,j)-a(i,j-1)              ! (1)
      c(j,i) = a(j,i)-a(j-1,i)              ! (2)
   end do
end do
```

プログラム (B)

```
do j = 2,nn
   do i= 1,nn
      b(i,j)= a(i,j)-a(i,j-1)
   end do
end do
```

```
do i = 1,nn
   do j = 1,nn-1
      c(i,j) =  a(j,i)-a(j-1,i)
   end do
end do
```

| 配列のサイズ | プログラム (A) | プログラム (B) |
|---|---|---|
| 3000 | 0.209 | 0.204 |
| 7000 | 1.280 | 1.127 |
| 10000 | 8.200 | 2.540 |

単位：秒

【その９】リストアクセスでの性能改善は？

プログラムによっては，下記のようにリストアクセスを用いる場合があります。しかも，このようなループがプログラム全体の中で高いコストを占め，かつ，そのループの２次キャッシュミス（プログラムのチューニングツールのプロファイラの出力では，“L2 キャッシュミス”として表示される）が高い場合（このプログラムでは 4％以上となっていた），どのようにしたら性能向上が望めるのでしょうか？

＜リストアクセスの代表的なループ＞

```
      do 560 l=0,mh-1            !! l=0,127
      do 560 m3=1,kmax           !! m3=1,256
      do 560 kk1=k1,k2           !! kk1=-256,255
         kk3=ms+m3
         dam1 = c(kk1,m3-1,2*l)
         dam2 = c(kk1,m3-1-kmax,2*l)
         c(kk1,ie1(kk3),2*l+1)=dam1+dam2
         c(kk1,ie2(kk3),2*l+1)=trigs(kk3)*(dam1-dam2)
560     continue
560     continue
560     continue
```

このような場合，L2キャッシュミスが多いリストベクトルを用いた配列の代入演算に対して，ロードのL2キャッシュミスを低減させるようにします。これは，最適化制御行の指定による手動プリフェッチ機能により行います。以下に例を示します。

＜リストアクセスでの性能改善の一例＞

```
          do 560 l=0,mh-1
          do 560 m3=1,kmax
          do 560 kk1=k1,k2
   !ocl prefetch_read(c(kk1+70,m3-1,2*l),level=2)
   !ocl prefetch_read(c(kk1+70,m3-1-kmax,2*l),level=2)
   !ocl prefetch_write(c(kk1+70,ie1(kk3),2*l+1),level=2)
   !ocl prefetch_write(c(kk1+70,ie2(kk3),2*l+1),level=2)
            kk3=ms+m3
            dam1 = c(kk1,m3-1,2*l)
            dam2 = c(kk1,m3-1-kmax,2*l)
            c(kk1,ie1(kk3),2*l+1)=dam1+dam2
            c(kk1,ie2(kk3),2*l+1)=trigs(kk3)*(dam1-dam2)
  560      continue
  560      continue
  560      continue
```

この最適化指示行の挿入により，以下のプロファイラの結果が示すように，このループで約30%の性能改善が見られました。

（オリジナル）

```
 Prefetch Information
  CPU(Sec) Commit Prefetch L1-op(%) Mem-acc(Sec) Mem-acc-para Cover(%)
 ------------------------------------------------------------------------
 3.527397e+02 2.109282e+11 9.804078e+09 2.8974 5.331419e-09 4.8140 99.3
```

（最適化指示行挿入）

```
 Prefetch Information
  CPU(Sec) Commit Prefetch L1-op(%) Mem-acc(Sec) Mem-acc-para Cover(%)
 ------------------------------------------------------------------------
 2.507152e+02 2.906897e+11 3.196270e+10 2.1108 7.622982e-09 7.6626 99.3
```

なお，プロファイラの利用については，参考資料［1］～［3］を参照されたい。

## Ⅲ. おわりに

一口にチューニングといってもチューニングする方法は，プログラムによってそれぞれ異なってきます。チューニングの事例に関しては，本センターのホームページ（http://www2.itc.nagoya-u.ac.jp/sys_riyou/hpc/index.htm）の「プログラムチューニング事例集」に掲載していますが，まだまだ事例が少なく，現実問題として余り参考にならないかもしれません。プログラムのこの部分で予想以上に時間が掛かっているとか，このように変更したら速くなったなどの事例がありましたら，センターまでお寄せいただければ幸いです。事例集の充実に役立てていきます。なお，プログラムの性能面でお困りの場合は，遠慮なくセンターまでお申し出ください。

**【参考資料】**

［1］津田知子：新スーパーコンピュータ（HPC2500）利用のしおり
名古屋大学情報連携基盤センターニュース ,Vol.4, No.2, pp.104, 2005.5
［2］スーパーコンピュータ及びアプリケーションサーバ利用の手引
名古屋大学情報連携基盤センター
［3］プログラミング支援ツール使用手引書 , 富士通マニュアル

（つだ　ともこ：名古屋大学情報連携基盤センター学術情報開発研究部門）