2.7 スレッド並列プログラムの性能測定

名古屋大学情報連携基盤センター
永井　亨

1. はじめに

名古屋大学情報連携基盤センターの Fujitsu PRIMEPOWER HPC2500（CPU の動作周波数 2.08Hz、総 CPU 数 1664、ノード数 24、トータルの理論最大性能 13.5Tflops、総主記憶容量 12TB）を使用してスレッド並列性能を測定した。本報告における測定では 1 ノード（64CPU、主記憶 512GB）を占有しておこなった。

2. 行列積

2-1. コンパイラオプション

スレッド数を 60 まで変化させたときの 4096×4096 行列積の経過時間を測定した。測定はそれぞれ 10 回おこない、その平均値をとった。これを flops 値にしたものを図 1 に示す。配列の宣言は

```
dimension a(4096+1,4096),b(4096+1,4096),c(4096+1,4096)
```

で、すべて倍精度実数型である。

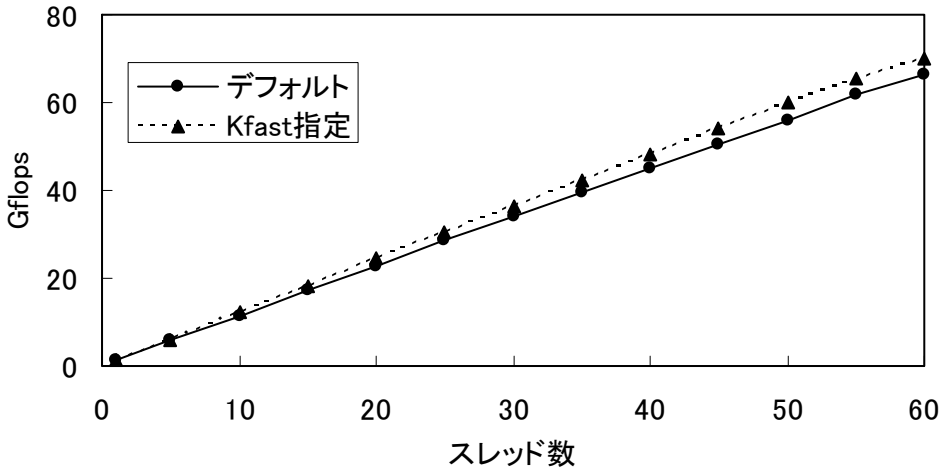

図 1

図 1 にはコンパイラの最適化オプションをデフォルト値（–Kfast_GP2=3,largepage=2,V9）のまま実行した場合（これは通常のプログラムでは最も性能が出る富士通推奨のオプションセット）と–Kfast を指定した場合の測定結果が示されている。デフォルトのほうが–Kfast を指定した場合よりも 7%程度おそい。これは–Kfast を指定した場合には 32 ビットアドレッシングモード（–KV8PLUS）が有効になるのに対し、デフォルトの 64 ビットアドレッシングモードでは 4 バイト整数から 8 バイト整数への変換やアドレス計算の増加、それに伴う一部最適化の違いなどの影響により 32 ビットアドレッシングモードより実行性能が低下する場合があるためである。

2-2. パディング

図 1 では 50 スレッドで約 60Gflops となっている。1CPU あたり 8Gflops の理論最大性能を持つことを考えるとこれはかなり低い値である。そこで、52 スレッドの場合について、配列宣言を

```
dimension a(ndim+m,ndim),b(ndim+m,ndim),c(ndim+m,ndim)
```

とし、ndim が 1024、2048、4096 のときに m を 1 から 10 まで変えてそれぞれ測定した。結果

を表 1 に示す。測定はそれぞれ 10 回おこない平均値をとった。表 1 にはコンパイラオプションとしてデフォルトの場合と–Kfast を指定した場合とが示されている。パディングの大きさに依存して性能が改善されるのはキャッシュ競合が緩和されるためと考えてよい。4096×4096 の m=1 では L1 キャッシュミスが大きくなるため、ほかの場合に比べて性能が低下する。

表 1　（単位は Gflops）

m	1024×1024		2048×2048		4096×4096	
	デフォルト	–Kfast	デフォルト	–Kfast	デフォルト	–Kfast
1	83.2	193.5	145.2	150.6	58.0	61.9
2	86.9	202.6	149.4	152.7	121.8	122.5
3	90.6	204.5	143.4	149.7	134.5	133.1
4	82.3	156.8	137.4	143.4	156.4	157.6
5	91.8	204.5	138.3	144.4	152.2	153.4
6	78.7	204.5	142.0	145.0	157.0	157.3
7	91.4	204.5	141.3	142.6	157.9	157.4
8	71.8	127.8	132.9	136.7	152.9	154.7
9	91.8	202.6	137.8	141.2	159.7	161.3
10	92.2	204.5	140.5	145.0	156.6	157.2

表 1 で–Kfast を指定した場合の 1024×1024 がほかの場合よりも性能が大きく上回ることは注目される。これは、主にコンパイル時に組み込まれる matmul 関数の処理論理に起因する。matmul 関数は、CPU 数により配列を最適な小行列に分割して行列積を実行する。 HPC2500 では L2 キャッシュは 4MB (1MB×4WAY)であるのにたいして、たとえば、52 スレッドでは 1024×1024 行列の matmul 関数内での小行列サイズは 2.8MB 程度になるが、2048×2048 行列および 4096×4096 行列の場合のサイズは 4MB を越える。このため後者では L2 キャッシュミスが増加し、性能が低下する。ちなみに 52 スレッドで実測した際のプロファイラが出力した L2 キャッシュミス率を表 2 に示す。

表 2

配列宣言	L2 キャッシュミス率(%)
(1024+10,1024)	0.0356
(2048+10,2048)	0.1051
(4096+10,4096)	0.1076

富士通によれば、行列積(matmul)の性能見積もりは 1CPU あたり 4.6Gflops 程度である。したがって、52 スレッド×4.6 = 239Gflops であるから–Kfast を指定した場合の 1024×1024 行列の値は妥当であるといえる。そこで m を 10 より大きい値にすることによりほかの場合でも 性能がさらに向上するか測定してみた。52 スレッドで m の値を 20、50、100 と変えた場合の表 3 に示す。表 1 と比べると、1024×1024 行列と 4096×4096 行列では明らかに改善されている。実効性能をあげるにはパディングの大きさを適切に選ぶことが重要である。

表 3　（単位は Gflops）

m	1024×1024		2048×2048		4096×4096	
	デフォルト	–Kfast	デフォルト	–Kfast	デフォルト	–Kfast
20	176.0	200.7	139.8	146.1	164.0	169.1
50	146.1	150.2	138.0	148.2	168.4	168.9
100	195.2	202.6	151.0	153.0	180.6	183.0

3. 2 次元拡散方程式

2 次元拡散方程式(1)を有限差分法で解く問題を考える。

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \qquad (\alpha > 0) \qquad (1)$$

時間を 1 次精度前進差分、空間を 2 次精度中心差分で離散化し、$t = k\Delta t$ 、 $x = i\Delta x$ 、 $y = j\Delta y$ 、とおいて $u(t, x, y)$ を $u_{i,j}^k$ と表すと、

$$u_{i,j}^{k+1} = s_x u_{i+1,j}^k + (1 - 2s_x - 2s_y) u_{i,j}^k + s_x u_{i-1,j}^k + s_y u_{i,j+1}^k + s_y u_{i,j-1}^k \qquad (2)$$

となる。ただし、$s_x = \alpha\Delta t / \Delta x^2$ 、 $s_y = \alpha\Delta t / \Delta y^2$ である。式(2)の計算は図 2 のように書ける。時間積分を行う A の部分と、積分値の更新を行う B の部分の経過時間を測定した結果を表 4 に示す。ただし、$0 \le i \le 2000$ 、$0 \le j \le 2000$ とし、A および B をそれぞれ 1000 回繰り返して平均値をとった。表 4 をみると A の部分と B の部分とで処理時間に大きな差はないことがわかる。prefetch 機能はコンパイラの標準オプションとなっているため、より高速に実行するためにはメモリコピーをおこなう B の部分をなくすプログラミングが必要になる。B の部分をなくす方法としては以下の 3 つが考えられるであろう。

```
    !  calculate temporal integral
p           do j=1,jmax-1
pu            do i=1,imax-1
pu              wk(i,j)=sx*u(i-1,j)
              *        +(1.0d0-2.0d0*sx-2.0d0*sy)*u(i,j)
              *        +sx*u(i+1,j)+sy*u(i,j-1)+sy*u(i,j+1)
pu            enddo
pu          enddo
    !  renew the values
p           do j=1,jmax-1
p             do i=1,imax-1
p               u(i,j)=wk(i,j)
p             enddo
p           enddo
```

図 2

表 4　（単位は msec）

スレッド数	A	B
1	42.07	41.77
10	6.20	6.11
20	0.67	0.47
30	0.46	0.33
40	0.35	0.27
50	0.30	0.23
60	0.26	0.21

[案 1]

図 3 に示すように u を 3 次元配列に変更し、計算ステップ数 istep が奇数のとき k1=1、k2=2、偶数のとき k1=2、k2=1 としてスイッチする。ただし、この場合には do ループの直前に指示行を挿入し、配列 u が回帰参照ではないことを明示する必要がある。指示行を有効にするためコンパイルオプションに–Kocl,vppocl を追加した。

```
            dimension u(:,:,2)
                 :
            if(mod(istep,2).eq.1) then
              k1=1
              k2=2
            else
              k1=2
              k2=1
            endif
      !ocl norecurrence(u)
p           do j=1,jmax-1
pu            do i=1,imax-1
pu              u(i,j,k2)=sx*u(i-1,j,k1)+(1.0d0-2.0d0*sx-2.0d0*sy)*u(i,j,k1)
           *            +sx*u(i+1,j,k1)+sy*u(i,j-1,k1)+sy*u(i,j+1,k1)
pu            enddo
p           enddo
```

図 3

[案 2]

図 4 に示すように、計算ステップ数 istep の奇偶よって、u の 3 次元目の値を明示的に指定する。通常のコンパイルオプションのみで並列化される。

```
            dimension u(:,:,2)
                 :
            if(mod(istep,2).eq.1) then
p             do j=1,jmax-1
pu              do i=1,imax-1
pu                u(i,j,2)=sx*u(i-1,j,1)+(1.0d0-2.0d0*sx-2.0d0*sy)*u(i,j,1)
           *              +sx*u(i+1,j,1)+sy*u(i,j-1,1)+sy*u(i,j+1,1)
pu              enddo
p             enddo
            else
p             do j=1,jmax-1
pu              do i=1,imax-1
pu                u(i,j,1)=sx*u(i-1,j,2)+(1.0d0-2.0d0*sx-2.0d0*sy)*u(i,j,2)
           *              +sx*u(i+1,j,2)+sy*u(i,j-1,2)+sy*u(i,j+1,2)
pu              enddo
p             enddo
            endif
```

図 4

[案 3]

図 5 に示すように、同じサイズの 2 次元配列 u1、u2 を用意し、計算ステップ数 istep の奇偶よって左辺と右辺の配列を入れ替える。この場合も通常のコンパイルオプションのみで並列化される。

案 1～案 3 までを実測した結果を表 5 に示す。比較のため、図 2 の A と B の部分の処理時間（表 4 の A、B および A+B）もならべて示した。また、表 5 の A+B の 1 スレッドの経過時間を 1 としたときの台数効果を図 6 に示す。案 1 から案 3 までのどの場合も明らかな性能向上がみてとれる。案 1 と案 3 は値がほとんど同じであるためグラフが重なって一本の線のようにみえる。案 2 が他の 2 つより若干性能が下回るのはループ内のレジスタ見積りに関するコンパイラの最適化状況に差があるためである。なお、いずれの場合についても配列 u の 1 次元目のパディングの大きさを変えて実行してみたが、有意な性能差はみられなかった。

これらの結果より作業領域をもちいずに時間積分するプログラミングが有効であることが示された。もちろん、ここで示した例は非常に単純なプログラムであるから、より実用的なプログラムで検証する必要がある。

```
              dimension u1(:,:),u2(:,:)
                   :
              if(mod(istep,2).eq.1) then
p                do j=1,jmax-1
pu                  do i=1,imax-1
pu                     u2(i,j)=sx*u1(i-1,j)+(1.0d0-2.0d0*sx-2.0d0*sy)*u1(i,j)
             *            +sx*u1(i+1,j)+sy*u1(i,j-1)+sy*u1(i,j+1)
pu                  enddo
p                enddo
              else
p                do j=1,jmax-1
pu                  do i=1,imax-1
pu                     u1(i,j)=sx*u2(i-1,j)+(1.0d0-2.0d0*sx-2.0d0*sy)*u2(i,j)
             *            +sx*u2(i+1,j)+sy*u2(i,j-1)+sy*u2(i,j+1)
pu                  enddo
p                enddo
              endif
```

図 5

表 5　（単位は msec）

スレッド数	図 2			案 1	案 2	案 3
	A	B	A+B			
1	42.07	41.77	83.84	41.51	41.99	41.77
10	6.20	6.11	12.31	7.16	7.16	7.14
20	0.67	0.47	1.14	0.72	0.82	0.73
30	0.46	0.33	0.79	0.51	0.57	0.50
40	0.35	0.27	0.62	0.40	0.45	0.40
50	0.30	0.23	0.53	0.34	0.38	0.34
60	0.26	0.21	0.47	0.31	0.35	0.31

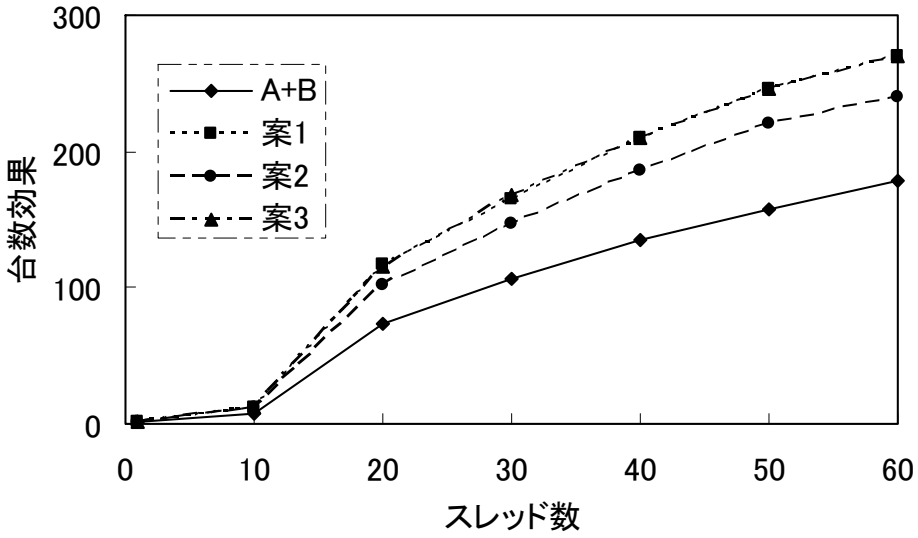

図 6

4. おわりに

簡単な 2 つのプログラム（行列積および 2 次元拡散方程式）を使用して HPC2500 上でのスレッド並列処理性能を測定した結果を報告した。SMP 上で処理性能を向上させるにはキャッシュミスの有無を意識したプログラミングが必要である。陽解法の場合に作業領域を用いずに時間積分する手法が実用的プログラムにおいても有効かどうかは今後の課題である。