T2K オープンスパコン(東大)チューニング連載講座(その 6)

UPC のチューニング

鴨志田 良和

東京大学情報基盤センター

はじめに

既存の逐次プログラミングを並列化したり、新規に並列プログラムを作成したりするとき、逐次プログラムからの変更をなるべく少なくして記述することができればプログラムの見通しが良くなるし、デバッグも容易になると考えられます。今回の記事では、このような並列プログラミングを可能にするプログラミング言語のひとつである UPC を取り上げ、その性能を最適化していく手順を解説していきます。

高性能な並列プログラムを記述するためには MPI がよく用いられています。MPI に代表されるメッセージパッシングに基づいたプログラミングモデルでは、それぞれのプロセスが別々のメモリ空間を持ち、Send と Receive 命令によって他のプロセスと通信しながら計算をすすめます。このやり方でプログラムを作成する場合、データがプロセスを隔てた場合に扱いが面倒であり、また、全体のアルゴリズムが読みにくくなるなどの欠点があります。これに対して共有メモリモデルでの並列プログラミングはすべてのプロセスがメモリ空間を共有しているため、プログラミングが簡単になるという利点がありますが、メモリがローカル・リモートのどちらにあるかが明示されていないため、知らない間に通信が行われる可能性があり、高速なプログラムを記述するのが困難になる場合があります。

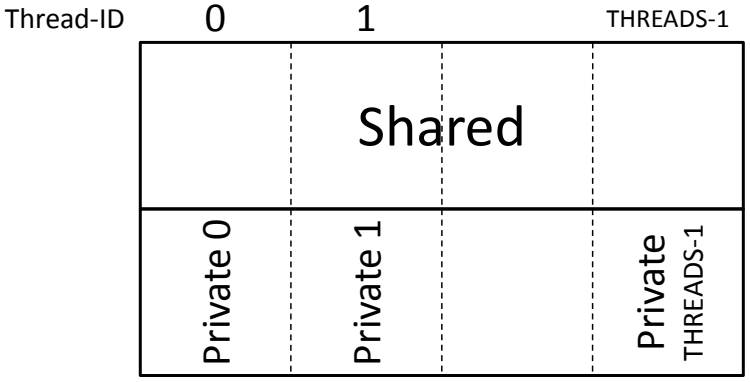
UPC(Unified Parallel C)は、C 言語を拡張して並列プログラムを記述できるようにした言語仕様で、2005 年に ver1.2 の仕様が策定されました。UPC のプログラミングモデルは共有メモリモデルを基礎としながら、ローカル・リモートのメモリの違いを型によってはっきり区別する分散共有メモリモデルを採用しているため、リモートへのメモリアクセスを最小限にするチューニングを行うことができます。OpenMP も C 言語のプログラムに並列化のためのプラグマを追加することで並列化を実現するため、もとの C 言語をもとに少ない変更で並列化を行うことができる点が似ています。

UPC の基礎

UPC では、複数のスレッドが、論理的にスレッドごとに分割された単一のグローバルアドレス空間で動作しているように見えます。各スレッドはこのグローバルアドレス空間の一部にアフィニティを持ち、また、各スレッドはプライベートなメモリ空間も保持しています(図 1)。図の Private と示された範囲は各スレッドが自分のメモリ空間に保有している領域で、自分しかアクセスできません。また、Shared と示された範囲はどのスレッドからも参照できるメモリ空間ですが、いずれかのスレッドのメモリ上に存在し、自分の持ちもの以外のメモリにアクセスする場合には通信のコストがかかります。UPC ではこの様なメモリ空間上で SPMD 型のプログラミングを行うことができます。各スレッドは、MYTHREAD という定数で自分のスレッド ID を知ることができます。スレッドの総数は THREADS という定数で知ることができます。また、

バリアやロックなどの同期のための命令もあります。

図 1: UPC のメモリモデル



通常の C と同じように宣言された変数は Private 空間に確保されます。共有されたメモリ空間上にメモリを確保する場合には、以下のように shared という記憶クラスを指定して宣言を行います。この例の場合は、ary という int の配列が確保され、ary[0]はスレッド 0 に、ary[1]はスレッド 1 に・・・、ary[THREADS]はスレッド 0 に、ary[THREADS+1]はスレッド 1 に・・・というように、ラウンドロビンにアフィニティが設定されます。

```
shared int ary[THREADS*16];
```

実行環境の実装にもよりますが、たとえば Berkeley UPC では、それぞれのスレッドは、実際はプロセスに対応し、スレッド間の通信には、UDP、MPI、Myrinet や Infiniband など、いろいろな通信方法を選ぶことができます。また、POSIX thread (Pthread)をこれらの通信方法の一つと組み合わせて用いることができます。

コンパイルとインストール

現在、いくつかの大学や企業から、UPC のコンパイラや実行環境などの実装が公開されていますが、今回は、Berkeley UPC 2.8.0 を用いました。Berkeley UPC は http://upc.lbl.gov/で公開されています。Berkeley UPC のうち、今回コンパイルして使用するのは、実行環境とフロントエンドの部分のみで、UPC を C に変換するトランスレータは upc-translator.lbl.gov で Web サービスとして提供されている、HTTP ベースのトランスレータを使用します。

Berkeley UPC のようにフリーソフトウェアとして公開されているソフトウェアは、HA8000 クラスタシステム上でコンパイルして、各自のホームディレクトリにインストールするなどして利用することができます。HA8000 クラスタシステムの各ノードは Opteron CPU を搭載した Linux のシステムです。このアーキテクチャは PC でも広く利用されているものですから、多くのフリーソフトウェアは問題なくコンパイルできます。

まずは UPC 実行環境のソースコードを取得して展開します。ファイルの展開やコンパイルは、/tmp の中で実行するとよいでしょう。/tmp にはローカルディスクのパーティションが割り当てられているので、ネットワークへのアクセスをファイル操作のたびに行う必要がある

HSFS(/home や/short)よりも、レスポンスが優れています。/tmp の中の、しばらく使用されていないファイルは定期的に削除されます。

```
$ mkdir /tmp/$HOME
$ wget http://upc.lbl.gov/download/release/berkeley_upc-2.8.0.tar.gz
$ tar –C /tmp/$HOME -xvzf berkeley_upc-2.8.0.tar.gz
$ cd /tmp/$HOME/berkeley_upc-2.8.0
```

gcc でコンパイルが可能なフリーソフトウェアは多いですが、HA8000 クラスタシステムにインストールされている gcc4 のオプティマイザに含まれるエラーと Berkeley UPC が使用している機能が競合するため、configure を行うときにエラーで停止してしまいます。gcc3.4 を使用することも可能ですが、今回は Intel コンパイラを使用することにします。バックエンドで使用する MPI は、mpich-mx-intel を使用することにします。以下のコマンドを使うと、PATH などの環境変数を設定できます。

```
$ . /opt/itc/mpi/mpiswitch.sh mpich-mx-intel
$ . /opt/intel/Compiler/11.0/074/bin/iccvars.sh intel64
```

コンパイルとインストールを行います。インストール先は、$HOME/work/bupc-2.8.0 としました。$HOME/work/bupc-2.8.0/bin に PATH を通すと、upcc(コンパイラ)、upcrun(起動プログラム)等を使用することができるようになります。

```
$ ./configure CC=icc CXX=icpc MPI_CC=mpicc MPI_CXX=mpicxx ¥
   --prefix=$HOME/work/bupc-2.8.0 --disable-udp ¥
   --enable-mpi --enable-par --disable-aligned-segments
$ make
$ make install
```

動作確認をかねて、単純なプログラムを実行してみます。以下の内容を、hello.upc というファイル名で作成します。

```
#include <upc_relaxed.h>
#include <stdio.h>

int main(int argc, char **argv){
  printf("Hello, my ID = %d, num of threads = %d¥n",
        MYTHREAD, THREADS);
  return 0;
}
```

以下のようなコマンドでコンパイルできます。-T オプションはスレッドの数です。

```
$ upcc --network=mpi -T4 hello.upc -o hello
```

実行するときは、以下のようなジョブスクリプトを qsub すればよいでしょう。

```
#!/bin/sh
#@$-r test -q debug -N 4 -J T1 -lm 2gb -lT 300
cd "$PBS_O_WORKDIR"
upcrun ./hello
```

正常に実行されれば、以下のような出力を得ることができます。

```
Hello, my ID = 3, num of threads = 4
Hello, my ID = 0, num of threads = 4
Hello, my ID = 1, num of threads = 4
Hello, my ID = 2, num of threads = 4
```

ソートプログラムのチューニング

単純な並列化

UPC を使って記述したアプリケーションの並列化とチューニングの例として、この記事ではソーティングを行うプログラムを取り上げます。ソートのアルゴリズムは、並列化するときの考え方がわかりやすい、バイトニックソートを考えることにします。バイトニックソートは、1つずつの増加列と減少列からなるバイトニック列を考え、二つのバイトニック列をひとつのバイトニック列にマージする操作を繰り返してソートを行います。バイトニックソートは、要素数が 2^n のとき、各要素を比較する処理が、常に完全に並列に(同時に 2^{n-1} 組の比較が)行える、並列度の高いアルゴリズムとしてよく知られています。要素数が 2^n の配列 a を並べ替えるコードは、以下のような 3 重ループになります。

```
01| void sort(double *a, int n){
02|  int i, m = 1 << n; /* 2^n = m = size of array */
03|  for(i = 0; i < n; ++i){  /* merge(i) */
04|    int j, s, k = 2 << i;
05|    for(j = k/2; j > 0; j /= 2){ /* swap(j) */
06|      for(s = 0; s < m; ++s) {
07|        double tmp;
08|        int d, d2 = (s/j) % 2; /* d is direction; 0: ascending
                                                    1: descending */
09|        if(d2) continue;
10|        d = (s/k) % 2;
11|        tmp = a[s+j];
12|        if(((d == d2)&&(a[s] > tmp)) || ((d != d2)&&(a[s] <= tmp))){
13|          a[s+j] = a[s];
14|          a[s] = tmp;
15|        }
16|      }
17|
18|    }
19|  }
20| }
```

これを UPC で並列化する場合、以下のように 3 か所を書き換えます。

```
01| void upc_sort(shared double *a, int n){
...
06|      upc_forall(s = 0; s < m; ++s; &a[s]) {
...
17|      upc_barrier;
```

1 行目は、配列を shared をつけて宣言することで、共有領域に配列を置いています。upc_barrier は、すべてのスレッドが upc_barrier を実行するまで待つという、同期のための命令です。upc_forall は、for ループを並列化するための命令です。通常の for と違い、;で区切られた 4 つ目のセクションがあります。ここには、整数か shared 領域へのポインタを記述することができます。この部分で、どのスレッドが特定のループを実行するかが決まります。整数 x を指定した場合は、スレッド ID が x % THREADS のスレッドが実行します。ポインタの場合は、そのポインタが差す場所にアフィニティを持つスレッドが実行します。ここでは s 番目のループは&a[s]にアフィニティがあるスレッド(s % THREADS)で実行されることを示しています。プログラムの構造がほとんど変わらずに、並列化できていることに注目してください。

コンパイルと実行は、以下のようなコマンドラインで行います。-T, -np オプションでスレッドの数を指定します。

```
$ upcc -O –T16 -pthreads sort_upc.upc -o sort_upc
$ upcrun -np 16 ./bitonic_simple_upc
```

2^{24}(=約 1600 万)要素を HA8000 クラスタシステムの 1 ノード上で実行した結果は以下のようになりました(表 1)。とても少ないコード変更でマルチコア CPU での高速化をある程度実現することが出来ました。UPC 化していない元のプログラム(表 1 では Native C で表示)と比べると、1 スレッドの時のオーバーヘッドが 7.8%ありますが、2 スレッド使用すると元のプログラムより実行時間が 38 秒ほど短くなりました。16 スレッドの場合は約 4 倍の速度になりました。

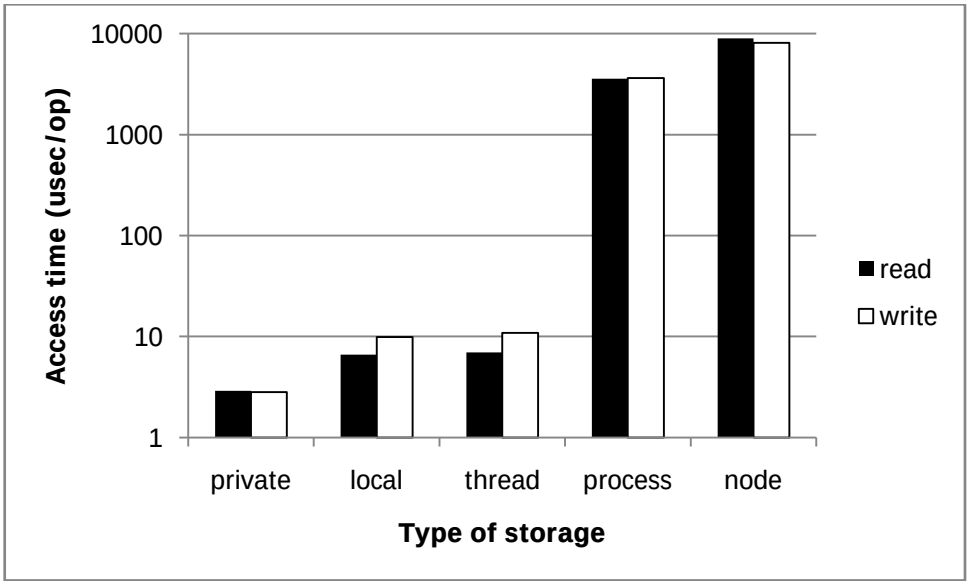
表 1: 単純な並列化の効果

スレッド数	実行時間(msec)	1 スレッドに対するスピードアップ
1(Native C)	180,161	107.8%
1	194,275	100.0%
2	142,123	136.7%
4	93,110	208.7%
8	62,573	310.5%
16	48,862	397.6%

メモリアクセス速度

UPC では通常のプライベートなメモリ空間にアクセスするのとほとんど同じようにして共有メモリ空間にもアクセスできますが、当然のことながら両者のコストは同じではありません。また、共有メモリが、ローカルスレッドにアフィニティを持つ場合と別のスレッドにアフィニティを持つ場合でもアクセス速度は変わってきます。さらに、そのスレッドが同一ノードに存在するのか、別のノードなのかなどでも、アクセス速度が違います。単純なメモリアクセスを行うプログラムを実行して、共有メモリにアクセスするためにはどの程度のオーバーヘッドがかかるのかを確かめてみましょう。図 2 は、メモリの種類によってアクセス速度がどのように違うかを示したものです。2^{20}(約 100 万)要素の double 配列に対して連続で読み書きを行い、かかった時間からアクセス 1 回あたりの時間を算出しました。横軸の private は private 領域、残りはいずれも shared 領域へのアクセス時間で local はローカルアクセス、thread は同一プロセスの別スレッド、process は同一ノードの別プロセス、node は別ノードのスレッドへのアクセス時間を示しています。プロセス間・ノード間の通信には MPI を使っています。プロセス内では private 領域内の 2 倍以上、別プロセス・別ノードになると 3 ケタの差があることがわかります。高性能な並列プログラムを作成するには、このメモリアクセス速度の差を考慮してプログラミングする必要があります。

図 2: メモリのアクセス速度



並列ソートのチューニング

先に解説したとおり、UPC は少ないコードの変更でプログラムを並列化できます。これをさらに高速化することを考えてみましょう。よりよい性能を得るためには、以下のような事項に注意します。

・無駄な同期を減らす
・メモリアクセスの局所性を活用する
・リモートアクセスを減らす
・ローカルな共有領域へのアクセスを減らす

今回紹介しているプログラムでは、同期の数はさほど多くないので、メモリアクセスを最適化することを考えます。UPC では共有メモリをブロック化することができます。以下の二つの定義を見てください。

```
shared int x[10];
shared [2] int y[10];
```

上の配列 x には、x[0]がスレッド 0、x[1]がスレッド 1、・・・というように 1 要素ずつラウンドロビンにスレッドへのアフィニティが設定されることはすでに述べました。下のように宣言を行った場合、y[0]と y[1]がスレッド 0、y[2]と y[3]がスレッド 1、・・・というように、2 要素ずつアフィニティが設定されます。このことを利用し、あらかじめいくつかの連続した要素が同じスレッドに属するようにしておき、さらにそれらの要素をいったん private 領域にコピーしてからソートを行うことで、ローカルな共有領域へのアクセスオーバーヘッドを減らすことができます。

具体的には、まず関数の引数の型を以下のように変更します。ブロックサイズを、$B=2^{LOGB}$

であるとします。

```
01| void upc_sort(shared double *a, int n){
                ↓
01| void upc_sort(shared [B] double *a, int n){
```

そして最外ループの最初のLOGB回のメモリアクセスはすべてローカルアクセスになることを考慮して、以下のように最外ループを分割します。

```
03|  for(i = 0; i < n; ++i){  /* merge(i) */
                ↓
A1|  double b[B];
A2|  upc_forall(i = 0; i < m; i += B; &a[i]){
A3|    upc_memget(b, &a[i], B*sizeof(double));
A4|    sort(b, LOGB); /* sort locally */
A5|    private2shared(a, i, b, B, 1, (i/B)%2);
A6|  }
A7|  upc_barrier;
03|  for(i = LOGB; i < n; ++i){  /* merge(i) */
```

ここで、upc_memget の呼び出し部分は、shared 配列の a[i]から B 要素分を private 配列の b にコピーすることを、private2shared の呼び出し部分は、以下のように、private 配列の b から、a[i]・・・a[i+B-1]にデータを正順または逆順にコピーすることを意味します。sort は逐次版の sort をそのまま使いました。

```
void private2shared_reverse(a, i, b, m, reverse){
  int j;
  for(j=0; j<m; ++j) a[i+(reverse?(m-1-j):j)] = b[j];
}
```

3 重ループの 2 つ目のループも、j が B/2 以下になるとローカルアクセスだけになります。この部分についても同様に、いったん private 領域にコピーしてからまとめて処理するようにします。具体的にはまず、2 つ目のループを B で区切ります。

```
05|    for(j = k/2; j > 0; j /= 2){ /* swap(j) */
                ↓
05|    for(j = k/2; j >= B; j /= 2){ /* swap(j) */
```

そして、ループの後半は、以下のように書きなおします。一番内側のループも B ごとに区切

って実行したほうが、upc_forall が実行される回数が減るため、効率が良くなります。

```
18|    }
19|  }
                ↓
18|    }
B1|    upc_forall(s = 0; s < m; s += B; &a[s]) {
B2|      upc_memget(b, &a[s], B*sizeof(double));
B3|      int j2, r, d = (s/k) % 2;
B4|      for(j2 = j; j2 > 0; j2 /= 2){
B5|        for(r = 0; r < B; ++ r){
B6|          int d2 =((s+r)/j2) % 2;
B7|          if(d2) continue;
B8|          double tmp = b[r+j2];
B9|          if(((d == d2)&&(b[r] > tmp)) ||
Ba|             ((d != d2)&&(b[r] <= tmp))){
Bb|            b[r+j2] = b[r];
Bc|            b[r] = tmp;
Bd|          }
Be|        }
Bf|      }
Bg|      upc_memput(&a[s], b, B*sizeof(double));
Bh|    }
Bi|    upc_barrier;
19|  }
```

図 3 に、チューニング前後でどの程度実行速度が変化したかを示します。BASE が単純な並列化を行った時、OPT1 が最外ループの分割を行った時、OPT2 が内側のループの分割をさらに行った時です(B=2^{20} としました)。図 3 には、BASE・1 スレッドの時の実行時間を、各ケースの実行時間で割ったものをプロットしています。BASE や OPT1 はスレッド数が増加すると性能の伸びが鈍化しているのに対して、OPT2 はより線形に近い性能の伸びを示しています(OPT2-16 スレッドを、OPT2-1 スレッドと比較した並列化効率は 88%)。また、OPT2 では、一番内側のループを B ごとに分割して実行した効果でキャッシュ効率が良くなり、1 スレッドでの性能も 154 秒と速くなりました。

ブロックコピーの最適化

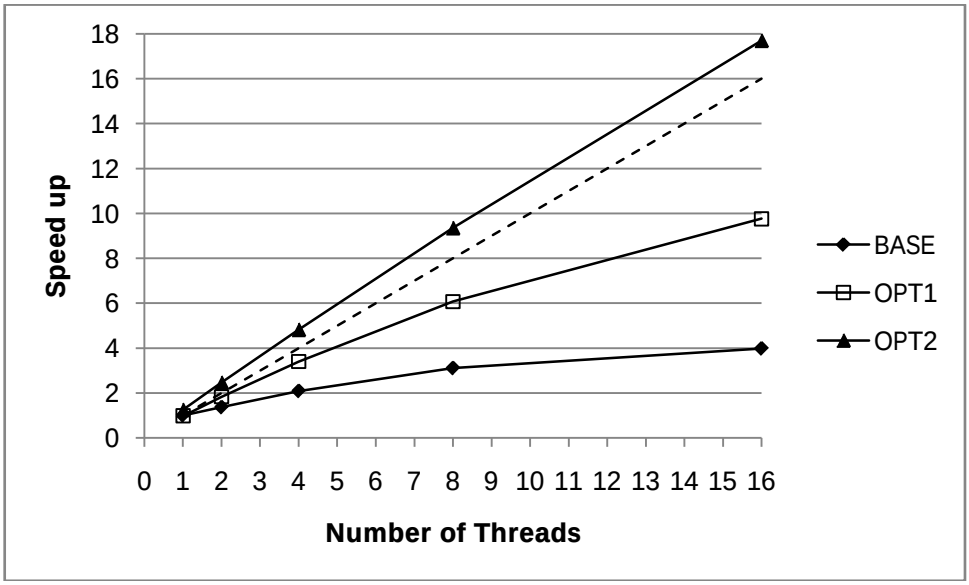
今までの例では、upc_memget や upc_memput のコピーは、共有配列の添え字が連続した領域と、プライベートな配列の間のコピーであるように見えるようなものばかりでしたが、実際は、これらの命令は、引数の共有ポインタが指し示す領域にアフィニティをもつ単一のスレッドがアフィニティを持っている配列の要素だけをコピーします。たとえば、以下のようなコードは、

```
    shared [2] double s[1024]
...
    upc_memget(d, &s[0], 5*sizeof(double));
```

以下とほぼ同じ意味です。この事実を使うと、複数のブロックを一度にコピーすることなどにより、工夫次第でさらに高度な最適化を行うことができる余地があります(この記事では省略します)。

```
    d[0] = s[0];
    d[1] = s[1];
    d[2] = s[THREADS*2];
    d[3] = s[THREADS*2+1];
    d[4] = s[THREADS*4];
```

図 3: ソート時間の SpeedUp



おわりに

この記事では、並列バイトニックソートを例に、UPC で記述されたプログラムのチューニングの手順を示し、その効果を確認しました。UPC を使うと、逐次版の C のコードに大きな変更を加えずに並列化を行うことができます。UPC では他のノードやプロセスが持つメモリに透過的にアクセスすることができますが、ノード間、プロセス間の通信は、スレッド間のアクセスと比べるとメモリアクセス速度に大きな差があります。今回の記事では、単一ノードでの実行についてのみチューニングを行い、性能評価をした結果を紹介しましたが、複数ノードでプログラムを実行する場合はこのメモリアクセス速度の差に注意してチューニングを行う必要があります。

また、これまでのチューニング連載講座で紹介されてきた、SSE 命令や numactl などによる高速化と組み合わせることにより、より高度な高速化を行うことも可能です。