実アプリケーションの最適化のテクニック

吉廣　保
東京大学情報基盤センター

1. はじめに

スーパーコンピューティングニュース 10 巻 4 号、5 号と 2 回に渡り、HA8000 クラスタシステム（以降 HA8000）で高性能プログラミングを行うための手法や技術を紹介してきました。しかしながら、HA8000 の利用者全員が自分でプログラムを組み、計算を実行するユーザではなく、フリーソフトウェアなどをソースからコンパイルし利用する、もしくは与えられたプログラムを使って計算を実行するユーザも多いと考えられます。

本稿では、そのようなユーザのために、プログラム本体を修正せずとも、コンパイラオプションなどを修正するだけで、プログラムの高速化ができないかということを、実際に配布されているフリーソフトウェアを例に挙げて紹介する予定でした。しかしながら、正式運用が開始された段階で、HA8000 ではチューニングを行う際に必要となるパフォーマンスモニタが提供されておらず、プロファイラと最適化ログのみでは、当初予定していた内容を紹介できないため、少し内容を変更してプログラム実行時に有効なテクニックなどから紹介し、実アプリケーションの例は簡単に紹介することにしました。アプリケーションの最適化テクニックは SR11000/J2 でも有効なので、詳細に関しては、後日別の機会に紹介できればと思います。

2. プログラム実行時の CPU、メモリを特定の場所で実行させる方法（コマンドライン編）

4 号記事[1]の「6. ccNUMA 向け最適化」では、HA8000 の計算ノードに使われている AMD Quad Core Opteron プロセッサの特徴とシステムへの実装方法から、メモリアクセススピードなどに違いがあることを紹介しました。その結果、並列プログラム実行時にプロセス・スレッドを割り当てる CPU の制御と、割り当てた CPU に近いメモリを確保する制御が必要であり、その手法についても簡単に紹介されています。

5 号記事[2]の「3. プログラムを特定の CPU で実行させる方法」「4. メモリを固定した場所に割り付ける方法」では上記の手法のうち、プログラムソース本体にコードを記述し、NUMA ライブラリ libnuma の提供する関数を利用することで制御する方法を紹介しています。

本稿では、4 号で紹介したもうひとつの手法、プログラム起動時に numactl コマンドを利用する方法について紹介します。この方法は、HA8000 クラスタシステム利用の手引き[3]の第 9 章「使用例」でも簡単に紹介されていますが、今回はもう少し詳しく、実際のプログラム利用の際に必要な情報も含めて紹介します。

numactl を利用する方法の利点は、NUMA アーキテクチャを意識せずに作成されている（CPU、メモリの明示的割り当てを行っていない）プログラムを実行する際に、実行段階で CPU への割りつけやメモリ割り当てを指定できるということです。よって既存アプリケーションを主に利用しているユーザも、NUMA アーキテクチャを意識することで今までよりも高速にプログラムが実行される可能性が出てきます。

それでは、実際に numactl の利用方法を紹介します。まず HA8000 の計算ノードの構成を再度説明しておきます。

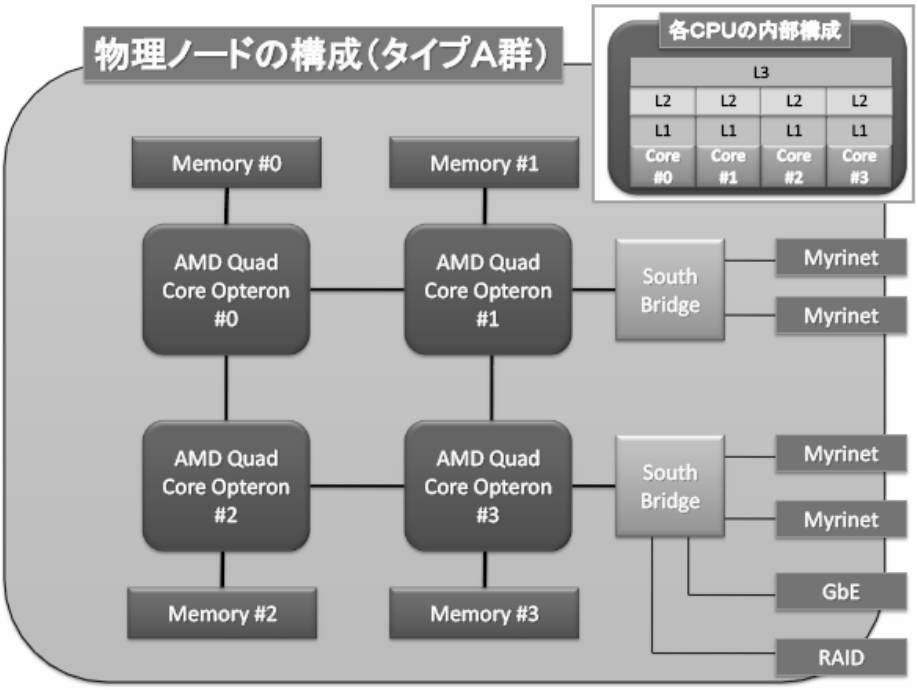

図１．計算ノード１ノード内の構成

図 1 のように計算ノード 1 ノードの中には 4 つの Opteron プロセッサがあり、それぞれの Opteron プロセッサにメモリが接続されています。4 つの Opteron プロセッサの中は、それぞれ 4 つのコアが入っており、コア毎に L1、L2 キャッシュを持ち、4 つのコアで L3 キャッシュを共有しています。

ここで、numactl を使う上での用語を整理しておきます。Opteron プロセッサ #0 と Memory #0 の組を 0 番のソケットと呼ぶことがありますが、numactl ではこれを「ノード」と呼びます。計算ノードとは違いますので区別するために、本稿では「Node」と記述することにします。また各 Opteron プロセッサ内のコアのことを「Physical CPU」と呼びますが、本稿では「CPU」と記述し、Opteron プロセッサのことは「CPU Node」と記述することにします。これをまとめると図 2 のようになります。

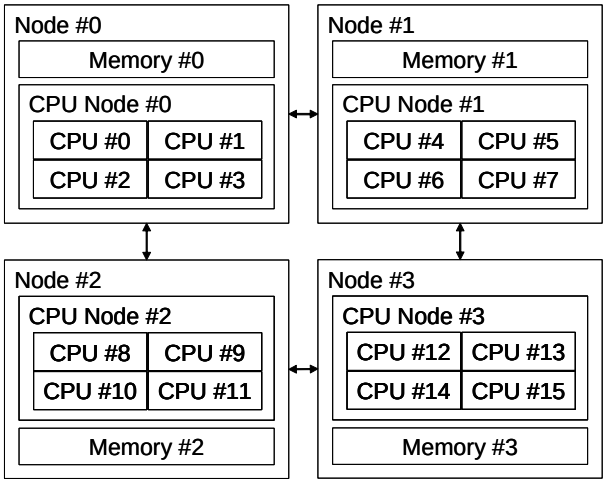

図２．numactl での各名称

まず、numactl というコマンドにどのようなオプションがあるのか見てみます。numactl にはヘルプを見るオプションが存在しないため、定義されていない「-h」などを引数として実行すると、エラーとなってコマンドの usage が表示されます。

```
$ numactl -h
numactl: option requires an argument -- h
usage: numactl [--interleave=nodes] [--preferred=node]
             [--physcpubind=cpus] [--cpunodebind=nodes]
             [--membind=nodes] [--localalloc] command args ...
      numactl [--show]
      numactl [--hardware]
      numactl [--length length] [--offset offset] [--mode shmmode] [--strict]
             --shm shmkeyfile | --file tmpfsfile | --shmid id
             [--huge] [--touch]
             memory policy

memory policy is --interleave, --preferred, --membind, --localalloc
nodes is a comma delimited list of node numbers or A-B ranges or none/all.
cpus is a comma delimited list of cpu numbers or A-B ranges or all
all ranges can be inverted with !
the old --cpubind argument is deprecated.
use --cpunodebind or --physcpubind instead
length can have g (GB), m (MB) or k (KB) suffixes
```

いくつかオプションが表示されますが、今回は CPU 割り当てに関する「--physcpubind=cpus」「--cpunodebind=nodes」とメモリ割り当てに関する「--interleave=nodes」「--membind=nodes」、亜システムの NUMA について確認を行う「--show」「--hardware」の 6 つのオプションについて説明します。それ以外のオプションについては、man numactl で表示されるオンラインマニュアルなどを参照してください。

それでは、実際に numactl で扱える CPU の数や Node の数を確認してみます。確認のために「--show」オプションを実行します。ついでに「--hardware」オプションも実行してみます。

```
$ numactl --show
policy: default
preferred node: current
physcpubind: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
cpubind: 0 1 2 3
nodebind: 0 1 2 3
membind: 0 1 2 3
```

show オプションによって、CPU は 0～15 番まで、CPU Node や Memory、Node は 0～3 番まで利用できることがわかります。policy とは、メモリの割り当て方法の選択で、interleave や membind オプションで変更が可能です。

```
$ numactl --hardware
available: 4 nodes (0-3)
node 0 size: 8062 MB
node 0 free: 5201 MB
node 1 size: 8080 MB
node 1 free: 7218 MB
node 2 size: 8080 MB
node 2 free: 7245 MB
node 3 size: 8080 MB
node 3 free: 7209 MB
node distances:
node   0   1   2   3
  0:  10  20  20  20
  1:  20  10  20  20
  2:  20  20  10  20
  3:  20  20  20  10
```

hardware オプションでは、それぞれの Node に搭載されているメモリサイズと現在の空きメモリサイズ、Node 間のソフトウェア的な距離を表示します。Node #0 と#1～#3 の距離が全部 20 となっていますが、この後のテストでもわかるとおり、対角線上に位置する Node は隣接している Node に比べると遠い（例えば#0 からの#3 までの距離は#1、#2 と比べると遠い）と考えられます。

では、5 号記事で使用された、約 1GB のメモリ領域を確保し、そのメモリ領域に 0x77 という値を書き込むプログラムを利用して、実際に numactl を使用してみます。今回のサンプルプログラムでは、5 号記事で利用された libnuma に関する関数は削除し、単純にメモリ確保、書き込み、開放のみを行うプログラムとなっています。詳細については 5 号記事を参照してください。サンプルプログラムとしては、malloc と free を使ってわかりやすく作成することも可能でしたが、今回は、5 号記事との比較のため mmap と munmap、memset を使ったサンプルプログラムになっています。

```
#include <sys/mman.h>
#include <stdio.h>
#include <string.h>

#define N 1000000000
#define RDTSC(X) asm volatile("rdtsc;shlq $32,%%rdx;orq %%rdx,%%rax":"=a"(X)::"%rdx")

int main(int argc, char *argv[])
{
  char *a;
  int i;
  unsigned long long t1, t2;

  a = mmap(NULL, N, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
  for (i = 0; i < 2; i++) {
    RDTSC(t1); memset(a, 0x77, N); RDTSC(t2);
    printf("Memory Bandwidth=%5.0f MB/s¥n",
           (double)N/1024/1024/((double)(t2-t1)/2300096000.0));
  }
  munmap(a, N);
  return 0;
}
```

このサンプルプログラムは、インラインアセンブラを用いているため gcc でコンパイルし、実行してみます（ログインノードでも実行可能です)。

```
$ gcc sample.c
$ ./a.out
Memory Bandwidth=  994 MB/s
Memory Bandwidth= 3248 MB/s
```

実行結果を 5 号記事と比較すると、同じ Node 内の CPU と Memory が利用されていると考えられます。次に、numactl を使って、実行 CPU と使用 Memory を変えながら試してみます。利用する CPU は 5 号記事に合わせて 0 番、4 番、8 番、12 番とそれぞれの Node から選択し、Memory は 1 番の Node に固定して実行してみます。CPU を指定するために physcpubind オプションを、Memory を指定するために membind オプションを使用します（ログインノードでも実行可能です)。

```
$ numactl --physcpubind=0 --membind=1 ./a.out
Memory Bandwidth=  672 MB/s
Memory Bandwidth= 2541 MB/s
$ numactl --physcpubind=4 --membind=1 ./a.out
Memory Bandwidth=  988 MB/s
Memory Bandwidth= 3255 MB/s
$ numactl --physcpubind=8 --membind=1 ./a.out
Memory Bandwidth=  591 MB/s
Memory Bandwidth= 2054 MB/s
$ numactl --physcpubind=12 --membind=1 ./a.out
Memory Bandwidth=  679 MB/s
Memory Bandwidth= 2560 MB/s
```

このように 5 号記事と同じ傾向の実行結果が得られました。この結果、HA8000 の NUMA アーキテクチャでは、メモリアクセスは同じ Node 内が一番速く、次に隣接する Node（Node #1 と#0、Node #1 と#3)、一番遅いのは対角線上にある Node（Node #1 と#2）ということがわかります。どのように CPU と Memory を組み合わせても同様の傾向の結果が得られます。また、libnuma を用いていない一般的なプログラムも、numactl によって実行時に CPU や Memory の割り当てを指定することができることがわかりました。

最初に numactl 無しで実行した場合、同じ Node 内の CPU と Memory が使用されましたが、どの組み合わせが使われるかは OS 任せになるので、可能な限り numactl を用いて使用する CPU と Memory を指定した方が良いでしょう。

3. バッチジョブでの numactl の利用

ここまでは、コマンドラインから numactl を使い逐次実行型のプログラムを 1 台の計算ノード内でのみ実行する方法を説明してきました。しかし、実際には HA8000 ではバッチジョブシステムを利用して並列実行型のプログラムで計算を行いますので、その方法について紹介します。

並列実行型プログラムと言っても、そのプロセス管理手段、通信手段などでいくつかの種類に分類することができます。本稿では、HA8000 の標準として用いられる MPI（mpich-mx）を使った並列プログラム（MPI 並列プログラム）と、日立製コンパイラによる自動並列化（または OpenMP による並列化）プログラム（SMP 並列プログラム)、MPI 並列と SMP 並列を組み合わせた Hybrid 型の並列プログラムの 3 種類について、バッチジョブで numactl を利用する方法について説明します。本稿では SMP 並列は日立製コンパイラの自動並列化機能で作成されたものと仮定しています。

i. MPI 並列プログラムの場合

MPI 並列プログラムでは、計算ノード内も計算ノード間もすべて MPI で通信を行います。生成されるプロセスは、それぞれ独立したものとなるので、CPU と Memory の Node を合わせることに気をつければ問題ないと考えられます。

ii. SMP 並列プログラムの場合

SMP 並列プログラムは計算ノード内でのみの並列化となります。HA8000 の場合だと 16CPU を使った並列実行が最大の並列数となります。SMP 並列化はスレッドを用いた並列化となり、それぞれのスレッドでメモリを共有することが可能です。よって CPU に関して OS 任せで、Memory はすべての Node のメモリを平均して利用する interleave モードで利用する方が、性能が得られると考えられます。ただし並列数 4 や 8 など計算ノード内の CPU をすべて使用し

ない場合は、使用 CPU を固定する方が良いと思われます。

iii. Hybrid 並列プログラムの場合

Hybrid 並列プログラムでは、計算ノード内では MPI 並列と SMP 並列を組み合わせて計算を実行し、計算ノード間は MPI 並列での実行となります。計算ノード内の MPI プロセスと SMP スレッドの組み合わせによって、CPU、Memory の割り当ての最適値は変化します。計算ノード内の MPI 並列と SMP 並列の組み合わせがわかりやすいように、MPI 並列数×SMP 並列数の Hybrid 並列と表現することにします。例えば、各 CPU Node では SMP 並列による実行で、CPU Node 間は MPI 並列で実行する場合、4MPI プロセス×4SMP スレッドということになるので、4×4Hybrid 並列と呼びます。考えられる組み合わせとしては、1×16、2×8、4×4 という組み合わせが主に使用されることになると考えられます。1×16Hybrid 並列の場合、計算ノード内だけをみると単純な SMP 並列となるため、B)と同様に Memory を interleave モードで使用することになります。2×8Hybrid 並列の場合、4 つの Node を分割する必要があり、プロセスの割りつけを 0 番 1 番 CPU Node、Memory を 0 番 1 番での interleave、同様に 2 番 3 番 Node を利用するといったパターンになると思いますが、プログラムによっては、Memory を interleave モードではなく、通常の NUMA モードで利用する方が良い場合もあるかもしれません。4×4Hybrid 並列では、CPU Node と Memory を合わせるように実行すれば良いでしょう。

では、それぞれの場合のバッチジョブスクリプトの記述方法について説明します。通常のバッチジョブでは、バッチジョブスクリプト中の mpirun コマンドによって、実行したいプログラムを起動することになりますが、numactl を使用する場合には、一旦シェルスクリプトを介して実行プログラムを起動します。

A) MPI 並列プログラムの場合

まず、以下のようなバッチジョブスクリプトを用意します。8 計算ノードを使用し、それぞれの計算ノードでは 16MPI プロセスを実行するジョブとなります。

```
#!/bin/sh
#@$-q parallel
#@$-N 8
#@$-J T16
#@$-lm 7GB
#@$-lT 01:00:00
#@$

cd ${QSUB_WORKDIR}
mpirun ./numarun.sh ./a.out,
exit
```

次に、以下のような numarun.sh を作成します。

```
#!/bin/sh

MYRANK=$MXMPI_ID
CPU=$(expr $MYRANK % 16)
NODE=$(expr $MYRANK / 4 % 4)

numactl --physcpubind=$CPU --membind=$NODE $@
```

$MXMPI_ID は、HA8000 システムで使用する mpich-mx に依存する環境変数です。mpirun によって起動されたプログラムには RANK 番号という一意の番号が割り当てられますが、それと同じものが環境変数 MXMPI_ID に代入されています。この RANK 番号からプロセスを割り当てる CPU 番号と Memory 番号を計算します。1 計算ノードあたり 16 プロセス起動されるので、RANK 番号を 16 で割った余りが、各計算ノードでの CPU 番号となります。次に Memory を割り当てる Node 番号の計算ですが、1Node あたり 4 プロセス割り当てられるので、RANK 番号を 4 で割ると使用する Node 数が計算できます。これを 4 で割った余りが Memory の Node 番号として使用できます。もしくは、すでに求めた計算ノード内での CPU 番号を 4 で割った商でも Memory の Node 番号は求められます。CPU 番号と Memory 番号が決まったところで、numactl の physcpubind オプションと membind オプションを使用して引数（$@）として渡されるプログラム（今回は./a.out）を起動します。これにより、RANK0～3 のプロセスは CPU#0~#3、Memory#0 で起動され RANK4～7 のプロセスは CPU#4～#7、Memory#2 と起動され、RANK16～19 のプロセスは次の計算ノードの CPU#0～#3、Memory#0 で起動されることになります。

B) SMP 並列プログラムの場合

SMP 並列の場合は、1 計算ノード内でのみの実行になるため、以下のバッチスクリプトとなります。

```
#!/bin/sh
#@$-q parallel
#@$-N 1
#@$-J T1
#@$-lm 28GB
#@$-lT 01:00:00
#@$

cd ${QSUB_WORKDIR}
export LD_LIBRARY_PATH=/opt/ofort90/lib/64:/opt/optc/lib64:
export HF_PRUNST_THREADNUM=16
numactl --interleave=all ./a.out,
exit
```

起動するプロセスは 1 プロセスのみなので、mpirun は使用していません。よって直接バッチスクリプト中に numactl を記述しています。日立製コンパイラの自動並列化機能で作成されたプロ

グラムは、起動されたプロセスが環境変数 HF_PRUNST_THREADNUM で指定された数の SMP 並列スレッドを作成し実行されます。また、numactl の interleave オプションで all を指定しているので、すべてのスレッドは Memory#0～#4 を決まったブロックサイズずつ順番に使用します。

C) Hybrid 並列プログラムの場合

Hybrid 並列では、主に 1×16、2×8、4×4 の組み合わせが考えられることを説明しました。それぞれの場合について例を示します。

- 1×16Hybrid 並列の場合

  バッチスクリプトは以下のようになります。8 計算ノードで、それぞれ 1MPI プロセス起動し、計算ノード内では MPI プロセスから 16SMP スレッド起動することになります。日立製コンパイラの自動並列化機能を使用した場合、実行時に LD_LIBRARY_PATH でコンパイラの用意したライブラリがあるディレクトリを指示する必要があります。

```
#!/bin/sh
#@$-q parallel
#@$-N 8
#@$-J T1
#@$-lm 28GB
#@$-lT 01:00:00
#@$

cd ${QSUB_WORKDIR}
export LD_LIBRARY_PATH=/opt/ofort90/lib/64:/opt/optc/lib64:
export HF_PRUNST_THREADNUM=16
mpirun ./numarun.sh ./a.out,
exit
```

numarun.sh スクリプトは以下のようになります。Memory の Policy の変更のみなので、バッチスクリプトに直接記述でも問題ないと思われますが、今回はスクリプトとして用意しました。

```
#!/bin/sh

numactl --interleave=all $@
```

- 2×8 Hybrid 並列の場合

  バッチスクリプトは以下のようになります。8 計算ノードで、それぞれ 2MPI プロセス起動し、計算ノード内ではそれぞれの MPI プロセスから 8SMP スレッド起動することになります。

```
#!/bin/sh
#@$-q parallel
#@$-N 8
#@$-J T2
#@$-lm 14GB
#@$-lT 01:00:00
#@$

cd ${QSUB_WORKDIR}
export LD_LIBRARY_PATH=/opt/ofort90/lib/64:/opt/optc/lib64:
export HF_PRUNST_THREADNUM=8
mpirun ./numarun.sh ./a.out,
exit
```

numarun.sh スクリプトは以下のようになります。今回は RANK 番号が偶数か奇数かによって使用する Node を決める必要があります。

```
#!/bin/sh

MYRANK=$MXMPI_ID
MYVAL=$(expr $MYRANK % 2)

if [ “$MYVAL” = “0” ]; then
  NODE=”0,1”
else
  NODE=”2,3”
fi
numactl --cpunodebind=$NODE --interleave=$NODE $@
```

RANK 番号が偶数のプロセス・スレッドを Node #0 と#1 で実行し、Memory も#0 と#1 の 2 つを使用した interleave モードで使用します。また、RANK 番号奇数のプロセス・スレッドは Node、Memory 共に#2 と#3 を使用します。プログラムによっては、interleave オプションの代わりに membind オプションを利用する方ことで性能が良くなる場合があるかもしれません。

- 4×4 Hybrid 並列の場合

  バッチスクリプトは以下のようになります。8 計算ノードで、それぞれ 4MPI プロセス起動し、計算ノード内ではそれぞれの MPI プロセスから 4SMP スレッド起動することになります。

```
#!/bin/sh
#@$-q parallel
#@$-N 8
#@$-J T4
#@$-lm 7GB
#@$-lT 01:00:00
#@$

cd ${QSUB_WORKDIR}
export LD_LIBRARY_PATH=/opt/ofort90/lib/64:/opt/optc/lib64:
export HF_PRUNST_THREADNUM=4
mpirun ./numarun.sh ./a.out,
exit
```

numarun.sh スクリプトは以下のようになります。今回は 4MPI プロセス×4SMP スレッドということで、SMP スレッドは同一 CPU Node 内で実行することが望ましいため cpunodebind オプションを使用します。またメモリも Node 毎に使用することが最適と考えられるため、membind オプションを利用します。

```
#!/bin/sh

MYRANK=$MXMPI_ID
NODE=$(expr $MYRANK % 4)

numactl --cpunodebind=$NODE --membind=$NODE $@
```

RANK 番号を 4 で割った余りが、計算ノード内での Node 番号と一致するためこのような numarun.sh スクリプトとなります。

以上がバッチジョブの実行形態によるスクリプトの作成方法となります。ちなみに HA8000 の開発元である日立製作所では、HA8000 での並列実行には 4×4Hybrid 並列実行を推奨しています。本稿の最後で、Hybrid 並列の 3 種類の性能の違いを実アプリケーションで計測してみたいと思います。

4. 実アプリケーション PHASE コンパイル時の利用ライブラリによる性能の違い

ここまでで、すでにあるプログラムを実行する場合に、なるべく HA8000 の性能を引き出せるような起動方法を紹介してきました。この節では、本来予定していた内容のほんの入り口ですが、コンパイル時の工夫で出来る限りプログラムを高速化する手段について、第一原理擬ポテンシャルバンド計算ソフトウェア PHASE という実際のアプリケーションを例題として説明していきます。

PHASE は東京大学生産技術研究所で実施されていた、文部科学省次世代 IT 基盤構築のための研究開発「革新的シミュレーションソフトウェアの研究開発（RSS21）」プロジェクトで開発されたソフトウェアです。RSS21 プロジェクト自体は昨年度で終了しましたが、現在は革新的シミュレーション研究センター（CISS）[4]で開発が継続されています。このプロジェクトで開発されたソフトウェア群は、非営利目的であれば無償で利用でき、PHASE のソースも CISS のホームページから入手することが可能

です。

PHASE は様々なコンパイラや数値演算ライブラリに対応しており、研究開発には SR11000/J2 も使用されていたため、日立製コンパイラの自動並列化や日立製数値計算ライブラリ MATRIX/MPP への対応もされている非常にすぐれたソフトウェアです。今回は、PHASE を MPI 並列版、Hybird 並列版の 2 種類でコンパイルし、使用する数値計算ライブラリを標準的な Netlib の BLAS、LAPACK、日立製 BLAS、LAPACK（逐次版、自動並列化版）と変更し、FFT ルーチンも PHASE 標準のものと、日立製 MATRIX/MPP に含まれるものとに変更して、サンプルデータの計算時間がどのように変化するかを検証してみます。インテルコンパイラと Math Kernel Library、PGI コンパイラと ACML ライブラリなどの組み合わせもコンパイル、実行可能ですが、今回の検証では省略しました。

現在配布されている、PHASE Ver.7.01 に含まれる configure スクリプトは HA8000 に対応していないため、少し修正を加えています。また、小さなバグが残っており、計算の終盤でログを書き出す際に HA8000 ではエラーになってしまう箇所があります。こちらも修正して検証しています。もし本稿を読んで、自分でも試してみたいという方は、修正点などについて、お気軽に情報基盤センターまでお問い合わせください。

実際の検証では、以下の 4 パターンで PHASE をコンパイルし、サンプルデータに含まれる水分子の振動解析（samples/phonon/H2O）の計算にかかる時間を計測しました。

① MPI 並列、Netlib BLAS/LAPACK、Built-in FFT
② MPI 並列、日立製 BLAS/LAPACK（逐次版）、MATRIX/MPP（逐次版）
③ Hybrid 並列、Netlib BLAS/LAPACK、Built-in FFT
④ Hybrid 並列、日立製 BLAS/LAPACK（自動並列版）、MATRIX/MPP（逐次版）

計算対象が水分子という小さな分子なため、1 計算ノードでの実行を行い、MPI 並列はそのまま 16MPI プロセスでの並列実行、Hybrid 並列では 4MPI プロセス×4SMP スレッドの並列実行を行い、出力ファイルに記述される最終計算時間について比較しました。その結果は以下のようになりました。

①MPI並列版（Normal）	1462.31秒
②MPI並列版（日立Lib）	1984.41秒
③Hybrid並列版（Normal）	9559.57秒
④Hybrid並列版（日立Lib）	977.62秒

③が極端に遅くなってしまった原因を考察すると、①、③で使用される Netlib BLAS/LAPACK は PHASE のソースに含まれるものをコンパイルするのですが、その際にも自動並列化が指定されており、実行時、PHASE 本体のプロセス・スレッドと BLAS/LAPACK のスレッドが重複してしまい CPU リソースを奪い合った結果遅くなったということが考えられます。そこで、①でコンパイルした BLAS/LAPACK（逐次版）を③のコンパイル時に利用するように修正し、再度実行したところ、計算時間に大きな変化はなく、上記考察は間違えていたことがわかりました。本来ならば、ここでパフォーマンスモニタやプロファイラを用いて、計算時間が非常にかかっているルーチンを比較、特定するのですが、最初に断ったようにパフォーマンスモニタが利用できないため、また紙面と筆者の都合上、後日改めて続編記事で紹介したいと思います。ちなみに、実行するたびに大きく計算時間にぶれが発生しており、②を例に取ると早いときは 1600 秒程度で終了する場合があります。これが低速なファイル I/O に起因しているのかなど、調査すべき項目が増えたので、こちらも可能ならば後日報告できればと考えています。

このように、利用するライブラリや並列化手法の組み合わせだけでも、計算性能に大きく影響を及ぼすことがあるので、プログラムソースが手元にあり、自分でコンパイルできる場合には、様々な可能性

を検討していただきたいと思います。

5. Hybrid 並列版 PHASE の MPI プロセスと SMP スレッドの組み合わせによる性能の違い

最後に、前章の実験で一番性能の良かった④の Hybrid 並列版で、MPI プロセス数と SMP スレッド数の組み合わせによって、どの程度性能に違いが出るのかを見てみたいと思います。

使用した例題は同じ水分子の振動解析で、1MPI×16SMP、2MPI×8SMP、4MPI×4SMP の 3 種類に関して計算実行を行うのですが、2MPI×8SMP では、メモリの利用方法によっても差が出る可能性があるため、「--interleave」「--membind」の両方のオプションで計算を実行してみました。その結果、4 種類の計算実行時間は以下のようになりました。

①1MPI×16SMP	1092.12秒
②2MPI×8SMP（interleave）	956.16秒
③2MPI×8SMP（membind）	1033.63秒
④4MPI×4SMP	977.62秒

結果から見ると、水分子の振動解析の場合は、2MPI×8SMP で Memory は interleave モードで利用すると良いようです。ただし、これはあくまで一例なため、計算対象分子が変わって、複数の計算ノードを利用する場合など、条件の変化で結果も大きく違ってくると思います。Hybrid 並列版プログラムを利用する場合にも、様々な可能性が検討できると思います。

6. おわりに

本来予定をしていました日立製コンパイラによるパフォーマンスモニタと最適化ログ、プロファイルを用いて、コンパイラオプションの変更やコンパイラ指示文の挿入程度でアプリケーションを高速化する手法については、なるべく近いうちに 4 章、5 章のプログラムの解析を含めて紹介したいと考えています。

4 号～本号までで紹介してきた内容以外にも様々なテクニック、手法が存在しており、すべてを紹介することは難しいです。そこで、プログラムチューニングについての入門的な内容のマニュアルを日立製作所から提供していただいています。HA8000 のユーザであれば、オンラインマニュアルのページ[5]で参照可能ですので、是非一度読んでみることをお勧めいたします。

参考文献・参考リンク

[1] 片桐孝洋、T2K オープンスパコン（東大）チューニング連載講座　高性能プログラミング（I）入門編、スーパーコンピューティングニュース Vol.10 No.4
http://www.cc.u-tokyo.ac.jp:16080/publication/news/VOL10/No4/200807tuning.pdf

[2] 黒田久泰、T2K オープンスパコン（東大）チューニング連載講座　高性能プログラミング（II）上級編、スーパーコンピューティングニュース Vol.10 No.5
http://www.cc.u-tokyo.ac.jp:16080/publication/news/VOL10/No5/200809tuning.pdf

[3] HA8000 クラスタシステム利用の手引き
http://www.cc.u-tokyo.ac.jp:16080/ha8000/ha8000-tebiki.pdf

[4] 革新的シミュレーション研究センター（CISS）
http://www.ciss.iis.u-tokyo.ac.jp/

[5] HA8000 クラスタシステムオンラインマニュアル
https://ha8000.cc.u-tokyo.ac.jp/