第 7 章　大規模並列計算機向け対応

7.1　GPU 向け数値予報モデルの動向 [1]

7.1.1　はじめに

スーパーコンピュータは地球大気の数値シミュレーションばかりでなく、地震や津波のような気象以外の自然現象に対する防災シミュレーションや自動車の設計等の産業分野、さらには生命科学、医療、社会科学分野に至るまで幅広い分野で用いられている。スーパーコンピュータは今後もわが国の基盤技術の一つとして必要不可欠なものである一方、スーパーコンピュータの今後の技術動向については、そのアーキテクチャの発展や省電力化の技術開発、利活用の推進などに多くの課題が残されている。

このような情勢を背景に、将来のスーパーコンピュータの開発方針については、文部科学省を中心に様々な調査・検討の取り組みが行われている。その一つとして、文部科学省の「HPCI [2] 計画推進委員会」に設置されたいくつかの作業部会での議論をまとめた HPCI 技術ロードマップ白書 [3]（以下、白書と言う）がある。これによると、CPU とアクセラレータ（CPU を補助して演算速度を加速させる装置）を組み合わせたアーキテクチャ（ヘテロジニアスアーキテクチャ）が今後のスーパーコンピュータを支える技術として有望視されており、アクセラレータの例として、GPU(Graphic Processing Unit) や MIC(Many Integrated Core) が挙げられている。また、演算性能の向上にメモリバンド幅の向上が追いつかなくなり B/F 値 [4] が低下していく状況が今後も続くことから、高速小容量のメモリと低速大容量のメモリとを組み合わせたメモリの階層構造がさらに必要となること、複雑化したメモリ構造に対応して性能向上を図るためには、メモリ配置を意識したプログラミングが必要となることなどが述べられている。白書では、主要なアプリケーションの分析も行い、様々な問題に適用可能な「汎用型」とメモリ性能を重視する「容量・帯域重視型」、逆に演算性能を重視する「演算重視型」、メモリ容量を極力削減した「メモリ容量削減型」の 4 種類のアーキテクチャが考えられると述べられている。

また、文部科学省は HPCI 計画推進委員会の下に「今後の HPCI 計画推進のあり方に関する検討ワーキンググループ [5]」を設置し、将来のスーパーコンピュータの計画推進について検討を行ってきた。ワーキンググループが平成 25 年 6 月にとりまとめた中間報告 [6] では、システム技術の動向として、GPU や MIC 等の新型プロセッサにより演算性能を向上させているシステムが多く見られるようになっていることを述べている。また、将来の日本におけるスーパーコンピュータのインフラとして、様々なアプリケーションに対応するために、幅広い分野に対応可能な高い計算性能を持つフラッグシップシステムと、フラッグシップシステムでは実効性能が低いアプリケーションに特化した複数のシステムによる構成を検討している。さらに、ワーキンググループの下に設置されたシステム検討サブワーキンググループは、フラッグシップシステムとして、汎用部に加速部を加えた、上述のヘテロジニアスアーキテクチャが妥当であると判断している [7]。

このような状況を考えると、今後はヘテロジニアスアーキテクチャによるシステムが中心となっていくと考えられ、気象庁においてもアクセラレータの利用に向けた取り組みが必要となると考えられる。上述の通り計算機性能を十分に引き出すためには、メモリ配置を意識したプログラミングが重要であり、そのための知見を蓄積していく必要がある。そこで、数値予報課では東京工業大学（以下、東工大）と共同で実際に GPU を用いた asuca の実行について研究を行っている。

GPU コンピューティングと GPU による asuca の実行については既に室井 (2011) で報告されているため、本節では、GPU コンピューティングについて簡単に述べた後に、共同研究の成果として実際に asuca のソースコードに加えた変更の概略を説明する。また、室井 (2011) 以後に GPU コンピューティングの環境は大きく変わっており、様々な GPU 向けプログラミングモデルが選択可能になるとともに、数値予報分野でも GPU 対応が進んできている状況を受け、以下では、GPU 向けプログラミングモデルと数値予報分野における GPU 対応についてレビューする。

7.1.2　GPGPU とは

GPGPU については室井 (2011) で述べられているので、以下の説明に必要な内容に絞って解説する。パソコンなどで使われている CPU に対して、GPU はグラフィック表示の処理に特化したプロセッサである。CPU は汎用的に使うために、異なるデータに対して異なる命令を実行するといった複雑な処理を行う必要がありコアを肥大化させてきた。一方、GPU が対象とするグラフィック表示においては、複数のデータに対して同じ命令を同時に実行することが重要であり、複雑な処

[1] 石田 純一、室井 ちあし
[2] High Performance Computing Infrastructure の略
[3] http://open-supercomputer.org/wp-content/uploads/2012/03/hpci-roadmap.pdf
[4] メモリバンド幅 (Byte/s) と演算性能 (Flops) の比。
[5] 気象庁から著者の室井が委員として参画している。
[6] http://www.mext.go.jp/b_menu/shingi/chousa/shinkou/028/gaiyou/__icsFiles/afieldfile/2013/07/10/1337595_1.pdf
[7] http://www.mext.go.jp/b_menu/shingi/chousa/shinkou/028/028-1/gaiyou/__icsFiles/afieldfile/2013/11/26/1341630_1.pdf

理を行う必要がない分、GPU のコアはシンプルで済む。さらに、コアがシンプルであるため、多数のコアを搭載することができ、トータルの演算能力を向上できる。近年では、スーパーコンピュータの評価において消費電力をいかにして抑えるか、といった観点が重要となってきている。この点で、GPU は複雑な処理を行うことはないものの、シンプルな処理に特化して省電力で高速計算を行えることから注目を浴びている。Green500 [8] というエネルギー消費効率の良いスーパーコンピュータのランキング [9] において、 2013 年 11 月には GPU を利用した東工大のスーパーコンピュータシステム (TSUBAME KFC) が第 1 位となっている。

当初の GPU はグラフィック表示に特化したものであった。一方、科学技術計算においては複数のデータに対して同じ命令を同時に実行することが圧倒的に多く、グラフィック表示で必要とする処理と似ている部分がある。そこで、GPU を汎用的な処理（科学技術計算）に利用できるように進化させたものが GPGPU(General Purpose computing on GPUs) である。気象の数値予報分野においても、このような事情が当てはまることから、GPU の利用が徐々に広まりつつある（第 7.1.6 項)。なお、以降では特に理由が無い限り GPGPU と GPU とを区別せずに GPU と記述する。

GPU の特徴として、多数のスレッドを同時に高速に処理できる点がある。スレッドとは CPU のプログラミングでも用いられる概念であり、例えば、繰り返し計算において、どの要素に対しても依存関係がなく同じ処理を行う場合に、その計算を複数に分割して行うことを指す。気象の数値予報計算では計算領域に配置された多くの格子に同じ計算を繰り返し実行するためこのような処理が大半である。なお、GPU を利用するための言語 CUDA(Compute Unified Device Architecture) では、スレッドが実行するコードをカーネルと呼んでいる。また、スレッドをまとめてブロックと呼び、さらにブロックをまとめてグリッドと呼んでいる。ソースコードの書き換えにあたってはこのような概念を意識する必要がある (次項で簡単にソースコードを紹介するが、そこではグリッドについては触れない)。

GPU のもう一つの特徴としてメモリの階層構造の制御があげられる。CPU のメモリ階層構造では、コアに近いメモリほど容量が少なく高速であり、遠くなるにつれて低速になるが容量が増えるような配置が一般的である。GPU においても同様に、コアから遠い順に全てのスレッドから参照可能なグローバルメモリ (相対的に大容量・低速)、複数のスレッドで参照可能だが全てのスレッドからは参照できないシェアードメモリ（小容量・高速)、スレッド間で共用しないレジスタ（小容量・高速：シェアードメモリよりも高速）となっていて、CUDA のプログラミングにおいては変数毎にどのメモリにおくかコントロールすることで、より性能を引き出すことができる。より高い性能を引き出すためには、スレッドの起動が高速であることを活かすと共に、メモリ階層を意識したコーディングが必要である。

[8] http://www.green500.org/
[9] 毎年 2 回 6 月と 11 月に更新される。

7.1.3 GPU による asuca の実行へのアプローチ

ここでは asuca を GPU で利用するために移植した際の方針について説明する。

冒頭に述べた asuca を GPU で実行させるための共同研究を開始した時点では、数値予報モデルを GPU で実行することはまだ一般的ではなく、WRF のプログラムコードの一部を GPU で動作するように書き換えて高速化したことが報告されていた程度であった。WRF の高速化とは、ソースコードで見ると全体の 1%に過ぎない雲物理過程が計算時間では 25%を占めていてボトルネックとなっていたことから、雲物理過程を GPU 向けに書き換えることによって 20 倍の高速化を達成し、全体として 1.2 − 1.3 倍の高速化を達成した (Michalakes and Vachharajani 2008) ものである。このようにソースコードの一部を書き換えるアプローチでは、GPU による高速化の効果は局所に限定され実際に得られる高速化率は限られる。上記の通り、GPU 向けに書き換えた部分が 20 倍に高速化されるとした場合、計算時間の 95%を占める部分を GPU 向けに書き換えても、全体の高速化は $1/\{(1-0.95)+0.95/20\} \sim 10.25$ 倍にしかならない。さらに、CPU と GPU はメモリを共有していないため、実際には CPU と GPU の間のメモリ転送に大きな時間がかかることも考慮する必要がある。数値予報モデルにおいては、力学過程や物理過程で計算した時間変化率を用いて時間積分を行い、そこで変更された予報変数やそこから診断された変数を用いて再び、力学過程や物理過程の計算を行うことを繰り返す。時間変化率の計算または時間積分の計算のたとえ一部であっても GPU で実行されない場合、時間積分の各ステップで CPU で計算した結果の GPU への転送及び GPU で計算した結果の CPU への転送が必要となり、多くの時間を要する。そこで、この共同研究においては、モデルの時間積分全体を GPU 向けに書き換えることとした。すなわち、CPU で初期値等を読み込んだ後に、データを GPU に転送し、力学過程・物理過程のそれぞれの時間変化率の計算と時間積分を全て GPU で行うというアプローチである。この場合、時間積分の各ステップでの CPU と GPU の間のメモリ転送は不要となり、大幅な高速化が可能となる。

7.1.4 GPU 向けソースコードの変更例

ここでは、GPU 向けソースコードの例を示す。まず、室井 (2011) に示されている通り、Fortran から C 言語へ書き換え、さらに C 言語から CUDA C [10] へと書き

[10] CUDA には C 言語を基にした CUDA C と Fortran を基にした CUDA Fortran とがあるため、区別して CUDA C と呼ぶ。

```
!$OMP PARALLEL DO
do j = 1, ny
do i = 1, nx
do k = 1, nz
  dens_ptb_v_rk_s(k,i,j) = dens_ptb_v_s(k,i,j) &
 & - ( mom_xi_v(k,i,j) - mom_xi_v(k,i-1,j) &
 &   + mom_yi_v(k,i,j) - mom_yi_v(k,i,j-1) &
 &   + mom_zi_v(k,i,j) - mom_zi_v(k-1,i,j) ) &
 & * dt_rk_s
end do
end do
end do
!$OMP END PARALLEL DO
```

図 7.1.1 Fortran コードの例。3 次元配列を取り、内側からインデックスを鉛直 (k)、東西 (i)、南北 (j) 方向にとる。

```
for(int j= ny_mgn; j<ny + ny_mgn; j++){
for(int k= nz_mgn; k<nz + nz_mgn; k++){
for(int i= nx_mgn; i<nx + nx_mgn; i++){
  int ix = lnx*lnz*j     + lnx*k     + i;
  int im = lnx*lnz*j     + lnx*k     + i-1;
  int jm = lnx*lnz*(j-1) + lnx*k     + i;
  int km = lnx*lnz*j     + lnx*(k-1) + i;
  dens_v_rk[ix] = dens_v[ix]
    - ((mom_xi_v[ix] - mom_xi_v[im])
     + (mom_yi_v[ix] - mom_yi_v[jm])
     + (mom_zi_v[ix] - mom_zi_v[km]) ) * dts;
}
}
}
```

図 7.1.2 C 言語へ書き換えた例。3 次元配列が 1 次元配列となる。また、連続するメモリへの配置順序を東西方向、鉛直方向、南北方向と変えている。

換える手順を踏む。これは、共同研究の開始時点では CUDA C が GPU の利用にあたり最適と考えたためである。以下、同じソースコードが、Fortran、C 言語、CUDA C でどのように変わるかを見る。

asuca の計算の多くは 3 次元のループになっている。この例として図 7.1.1 に split-explicit 法の密度の時間積分 (第 2.3.3 項を参照) を行う部分の Fortran によるソースコードを示した [11]。あるセル（格子点）の密度の時間変化率の計算において、セル境界のそれぞれの面でその面と直交する運動量を参照する。そして、求められた時間変化率に積分時間間隔を乗じて現在値に加えることにより次の時刻における値が求められる。Fortran では 3 次元配列を用い、インデックスの内側から鉛直方向 (k)、東西方向 (i)、南北方向 (j) としている。この部分にはループ順序の依存性はなく、配列の内側から外側に向かってループを書いている。次に、CUDA C に移る前の段階として C 言語に書き換える。C 言語によるソースコードは CUDA C の検証コードとしても用いられる。C 言語に書き換えたソースコードを図 7.1.2 に示す。本研究に利用したモデルの設定では東西及び南北方向の格子数が鉛直方向の格子数よりも多いために、東西方向のループを最内側としてループ長が長くなるように順序を入れ替えている。次に、CUDA C へ書き換える際に、GPU の効率を引き出すために、以下の方針とした。

- C 言語に書き換えた 3 重ループの多くは図 7.1.2 で示したように書かれている。これは、GPU で用いられる概念であるスレッド・ブロックを用いると、「スレッドをまとめたブロックを 2 次元とし（ここでは内側のループである i と k を指す）、残りの 1 次元の方向（ここでは j）に進んでいく」と表現する。このとき、スレッドの数が多いほど効率が良いため、ブロックとする 2 次元にはもっとも格子数が多いものを取る。
- 3 重ループの中に依存がなければ、ブロックとする 2 次元は任意に選択できるが、依存がある場合は任意にはとれない。例えば、鉛直 1 次元の連立方程式を消去法で解く場合は鉛直方向に依存があり、その方向に計算を進める必要があるため、i と j を内側として k の方向に進むように計算する。
- 計算を進めていく方向に参照する変数については他のスレッドから参照できなくても良いため、最も高速なレジスタに置く。それ以外の変数についてはレジスタの次に高速なシェアードメモリに置く。図 7.1.2 の例で言えば、あるセル (i,k,j) を計算したスレッドはその次には (i,k,j+1) のセルに対して計算を行う。従って、両者のセル境界にある変数 mom_yi_v はこの 2 つのセルの計算に用いられるだけで、他のセルの計算には用いられない。すなわち、他のスレッドから参照する必要がないためレジスタに置くことができる。一方、mom_xi_v はセル (i,k,j) とセル (i+1,k,j) の計算に用いられるなど別のスレッドから参照されるため、シェアードメモリに置く。mom_zi_v も同じ理由でシェアードメモリに置く。
- 複数の GPU 計算においては MPI を用い、計算と通信をオーバーラップする。すなわち、各 MPI プロセスでは予報領域を水平 2 次元分割した 3 次元領域を持っており、隣接 MPI プロセスが持つ情報が必要な領域は数格子幅の東西南北の壁付近の領域だけである。そこで、隣接 MPI プロセスが持つデータの通信と並行して各 MPI プロセスは中心付近の格子に対して演算を行う。

ここで示した対応策のうち、最初の 3 点は GPU 向けの書き換えに固有の変更である一方、最後に示した計算と通信のオーバーラップについては現在の計算機でも有効になりうる対応である [12] 。

[11] 解説のために実際のソースコードを一部書き換えている。

[12] ただし、今までに HITACHI SR16000M1 でテストした限りではオーバーラップによる効果はそれほど見えなかった。

density.cu の一部

```
001: ...
002: dim3 threads(BLOCK_DIM_X, BLOCK_DIM_Y);
003: dim3 blocks(get_griddim(grid_->lnx(), threads.x), get_griddim(grid_->lnz(), threads.y));
004: density_gpu::dyn_hevi_dens_gpu <0, 0, 0><<<blocks, threads>>>(grid_->parameters(),
005:     dens_v_rk, dens_v, mom_xi_v, mom_yi_v, mom_zi_v,dts);
```

density_gpu.cu の一部

```
001: global__ void dyn_hevi_dens_gpu(const struct ::GridParameters grid,
002:    FLOAT *dens_v_rk, const FLOAT *dens_v,
003:    const FLOAT *mom_xi_v, const FLOAT *mom_yi_v, const FLOAT *mom_zi_v, FLOAT dts)
004: {
005:     ...
006:     __shared__ FLOAT s_mom_xi_v[(block_dim_x+1)*(BLOCK_DIM_Y+1)];
007:     __shared__ FLOAT s_mom_zi_v[(block_dim_x+1)*(BLOCK_DIM_Y+1)];
008:     ...
009:     for (int jj=0; jj< (NJ <= 0 ? grid.ny + NJ : NJ); jj++) {
010:         const int ix     = grid.lnx*grid.lnz*j     + grid.lnx*k     + i;
011:         const int im     = ix - 1;
012:         const int km     = ix - grid.lnx;
013:         s_mom_xi_v[s_ix] = mom_xi_v[ix];
014:         s_mom_zi_v[s_ix] = mom_zi_v[ix];
015:         const FLOAT mom_yi_v_ix = mom_yi_v[ix];
016:         if (!(i == grid.nx_mgn - 1 || k == grid.nz_mgn - 1)) {//continue;
017:             dens_v_rk[ix] = dens_v[ix]
018:                 - ( (s_mom_xi_v[s_ix] - s_mom_xi_v[s_im])
019:                     + (mom_yi_v_ix - mom_yi_v_jm)
020:                     + (s_mom_zi_v[s_ix] - s_mom_zi_v[s_km]) ) * dts;
021:         }
022:        mom_yi_v_jm = mom_yi_v_ix;
023:    }
024:}
```

図 7.1.3 CUDA C へ書き換えた例。ただし、例外処理の一部を削除するなど、全てのソースコードは掲載していない。

これらの対応策を踏まえて、GPU 対応のために固有な変更を加えたソースコードを図 7.1.3 に示す（説明のため行番号を付加している）。おおまかに構造を述べると density_gpu.cu [13] の 1 行目から始まる dyn_hevi_dens_gpu という関数が前述のカーネルである。そして、この関数を呼び出している部分が density.cu の 4,5 行目であり、“<<<” と “>>>” で囲まれた中にブロック数やスレッド数を記述する。また、dyn_hevi_dens_gpu の 6,7 行目にある __shared__ がシェアードメモリを確保する宣言文である。実際の計算を行っているのが、17–20 行目である。配列のインデックス計算等のいろいろな処理が追加されている。基本的には上述のスレッドの制御とメモリの階層構造に応じた変数宣言を行うことにより GPU 対応を図っており、性能を引き出すためには大規模な書き換えが必要なことがお分かりいただけよう。

東工大との共同研究においては、このように asuca の全てのコードを書き換え、2010 年 11 月に稼働した東工大の TSUBAME2.0 において、3990GPU を利用して 145TFLOPS という高い実行性能を達成している (室井 2011; 下川辺ほか 2011)。

[13] CUDA は C 言語を元にしているが、C 言語とは異なるため、拡張子として “cu” を用いている

7.1.5 近年の GPU 向けプログラミングモデル

東工大との共同研究で用いた CUDA C の言語仕様は C 言語と非常によく似ているものの、同一ではなく、CPU で走らせるコードと GPU で走らせるコードとは共有できない。さらに、数値予報システムの多くのコードが現在は Fortran で書かれていることを考えると、C 言語への移行コスト (ソースコードの移植にかかるコストだけでなく、開発者が習熟するためのコストも含む) も必要となる。GPU の持つ高い実行性能と容易なプログラミングとの両立を目指して、近年では、GPU でプログラムを動かすためのアプローチには複数の選択肢がある。以下で簡単に説明する。

(1) CUDA Fortran

CUDA C が C 言語をベースとしているのに対して、CUDA Fortran は Fortran をベースとした言語である。CUDA C と同様に、GPU ベンダーである NVIDIA 社から提供されている [14]。コンパイラは PGI 社との共同開発であり、PGI Fortran で利用可能である (バージョンは PGI2010 以降) 。利用可能な機能が CUDA C とわずかに異なるが、この違いは C 言語と Fortran の言語仕様の違いに起因している。Fortran のソースコードから GPU の性能を最大限に引き出すにはもっとも対応しやすいアプローチだと思われる。

[14] http://www.nvidia.co.jp/object/fortran_jp.html

(2) F2C-ACC

f2c とは Fortran のソースコードを C 言語のソースコードに変換するツールとして古くから使われていたものであるが、F2C-ACC とは Fortran のソースコードを CUDA C のソースコードに変換するコンバータである。これは、米国海洋大気庁地球システム調査研究所 (National Oceanic and Atmospheric Administration / Earth System Research Laboratory: NOAA/ESRL) で開発されており [15] 、2009 年 6 月にリリースされ、現在は 2013 年 6 月にリリースされたバージョン 5.2 が最新である。NOAA/ESRL は NIM と呼ばれる正 20 面体格子を用いた非静力学モデルを開発しており、F2C-ACC はその GPU 対応に用いられている。Fortran77 と Fortran90 の様々な機能をサポートしているが、完全に準拠していないため、今後徐々に機能を追加していくことを計画している。CUDA C 向けに変換した後にチューニングが必要とのことでもあり、どの程度効率が良い CUDA C のコードに自動的に変換できるかが鍵となるだろう。

(3) OpenCL

OpenCL とは Open Computing Language の略であり、その名の通り、オープンな仕様の並列プログラミング環境である [16]。前述の CUDA は NVIDIA が提供していることもあり、NVIDIA の GPU には最適であり、またサポートも充実していると思われるが、CUDA の言語仕様がオープンでないため、他社 (例えば AMD) の GPU では動作すらしない。一方、OpenCL はオープンな規格であり、GPU のみならずマルチコア CPU でも動作可能となっている。OpenCL は C 言語をベースとして拡張された言語である。

(4) OpenACC

CUDA C, CUDA Fortran, OpenCL のいずれも C 言語もしくは Fortran をベースとして、GPU のための拡張を行った言語であった。この場合、拡張した言語に対応したコンパイラがないと動作させることもできない。このことは、GPU 環境のみを利用する場合には問題とはならないが、様々なプラットフォームで動作させようとすると問題となる。それに対して、OpenACC はソースコードに指示行を入れていくことにより GPU へ対応させることができるプログラミングモデルである [17] 。指示行の例として、図 7.1.1 の 1 行目が挙げられる。行頭が “!” になっていることから分かるように、これは Fortran 言語から見ると単なるコメント行である。指示行は OpenMP(Open Multi-Processing) でも用いられる手法であり、OpenMP をサポートしない Fortran コンパイラではこの指示行は無視してコンパイルを行い、サポートする Fortran コンパイラは OpenMP による並列化を行う。OpenACC も同様のプログラミングモデルであり、追加する指示行は OpenACC をサポートしないコンパイラから見ると単なるコメント行として扱われるので、様々な環境で利用可能となる。また、OpenACC は C 言語と Fortran をサポートしている。ただし、利用できる機能には制限があり、前述のレジスタとシェアードメモリの使い分けはできないようである。OpenACC は後発のプログラミングモデルであり (現在のバージョンは 1.0。ただし、2.0 の仕様が策定された)、今後状況は変わると考えられる。また、指示行を挿入することによるプログラミングモデルである以上、どうしても、CUDA や OpenCL に比べると性能は出しにくいと考えられる。ただし、ユーザーにとってソースコードの可搬性は魅力的であり、OpenACC が広まるかどうかは、今後どの程度 GPU の性能を引き出せるかにかかっているであろう。

(5) Hybrid Fortran

OpenACC は汎用の指示行によるプログラミングモデルであるため、CUDA や OpenCL といった GPU 向けに言語仕様を拡張したプログラミングモデルよりも性能向上が限られると考えられることは前述した。しかし、asuca あるいは気象予測プログラムの性能向上を目的とするのであれば、必ずしも汎用のプログラミングモデルではなく、気象予測プログラムの特徴を活かしたプログラミングモデルも有効である。現業運用を考え、スーパーコンピュータシステムで動作させることを考える場合、単にソースコードの可搬性を保つだけでなく、OpenMP による指示行が適切に使われる必要があることから、GPU と CPU の双方に対応可能であるプログラミングモデルが望ましい。そこで、気象庁、東工大、理化学研究所の共同研究では、GPU 向けと CPU 向けとに適切なソースコードを生成するプログラミングモデルとして Hybrid Fortran に取り組んでいる [18] 。これは、Fortran のソースコードに Hybrid Fortran の指示行を挿入し、コンパイルの前にスクリプトを実行して、GPU 向けには CUDA Fortran のコードを生成し、CPU 向けには OpenMP の指示行を挿入したコードを生成するというプログラミングモデルである。現時点では、物理過程ライブラリと asuca を念頭において開発を行っている。

7.1.6 諸外国の数値予報における GPU 対応の動向

最近の諸外国の数値予報モデルの GPU 向け対応の動向について述べる。ここでは、2013 年 9 月に米国大気研究センターで開催された Programming weather, climate, and earth-system models on heterogeneous multi-core platforms ワークショップの講演資料を引用する。講演資料はウェブサイト [19] に掲載されているの

[15] http://www.esrl.noaa.gov/gsd/ab/ac/Accelerators.html
[16] http://www.khronos.org/opencl/
[17] http://www.openacc.org/
[18] http://typhooncomputing.com/
[19] http://data1.gfdl.noaa.gov/multi-core/

で、適宜ご覧いただきたい。なお、このワークショップは第 7.1.1 項で述べたアクセラレータ全般を取り扱っており、前述の MIC についても様々な講演が行われているが、紙幅の関係もあり省略する。

前述の通り、NOAA/ESRL で開発している NIM は F2C-ACC による GPU 対応に取り組んでいる。また、F2C-ACC が完全には Fortran90 の機能をサポートしていないことから、OpenACC による GPU 対応も行っているが、F2C-ACC よりも高速化はできていない [20]。

米国の WRF に対しては雲物理過程、移流スキーム等のスキームに対して CUDA を使ったベンチマークカーネルを作成している [21] 。

第 1.3 節で述べた COSMO コンソーシアムの COSMO も GPU 対応を進めており、CUDA C と OpenACC の組み合わせを採用している [22] 。比較的ソースコードの分量が少なく、かつ計算時間を多く要する力学コアについては CUDA C を用いて全面的に書き換え、ソースコードの分量が多く、様々な開発者が開発を行い、また複数のモデルでソースコードを共有する物理過程は OpenACC による対応としている。これは、計算効率と開発コストとのバランスを考慮したものと考えられる。

また、COSMO コンソーシアムにおける次期のモデルとなる ICON も GPU 対応が行われている。単一ノードでの利用に対しては、OpenCL と CUDA Fortran を用いて力学コアの実装を完了し、複数ノードの利用については OpenACC を用いて力学コアを移植している [23]。

NVIDIA 社のワークショップの講演資料 [24] によると WRF, COSMO, NIM 等の GPU 対応には NVIDIA 社が関わっていることや、OpenACC によるアプローチが急激に広がっていることなどが示されている。

7.1.7 終わりに

スーパーコンピュータは数値予報の高度化を支える根幹の技術であり、その性能向上は気象予測の精度向上に直接結びついている。スーパーコンピュータの利用者にとっては、現在のアーキテクチャ、現在の数値予報システム・プログラムのままで高度化ができれば開発作業としては最も作業コストが低く望ましいが、今後のスーパーコンピュータの技術動向や省電力の要請等を踏まえると、様々な選択肢を考慮しておく必要がある。その取り組みの一つとして、東工大との共同研究による asuca の CUDA を利用した GPU 対応の取り組みを紹介してきた。この成果は先駆的かつ画期的なものであったと考えている。その後、システム技術の動向として GPU を含むアクセラレータを用いたヘテロジニアスアーキテクチャが有力となりつつあり、また、ソフトウェア技術として CUDA Fortran の登場や OpenACC といったより容易なアプローチも可能になり、GPU の利用は今後より一層広がるであろう。

GPU あるいは MIC といった特定の計算機アーキテクチャに特化したチューニングを行うと、そのアーキテクチャにおいては高性能を引き出せるが、可搬性が失われる事態にもなる。一方で、何にでも適用可能な汎用的アプローチでは、計算機アーキテクチャの持つ性能のポテンシャルは引き出せない。一般に、良い性能を得るためには目的を特化したチューニングが必要である。

そのため、自らのプログラム固有の特性に応じたアプローチを考えていく必要がある。数値予報モデルの GPU 向け対応においては、その特性を最も熟知している数値予報モデルの開発者と様々な計算機アーキテクチャを熟知している専門家と共に問題解決にあたることが効果的であり、共同研究といったアプローチが重要となるだろう。前述のワークショップにおいては、NVIDIA 社や Intel 社からの講演もあり、その資料によると両社は様々な機関と共同で GPU 対応、あるいは MIC 対応を行っている。ベンダーによる囲い込みという見方もできるが、計算機の専門家とアプリケーションの専門家が協力して問題に対応することが重要であり、様々な機関がそのようなアプローチをしている状況と言える。asuca 及び物理過程ライブラリの GPU 向けの開発においては、東工大との共同研究を行ってきたところであり、今後も最新の技術動向や大学・研究機関の最新成果も反映しつつ、数値予報システムの高度化を進めていく必要があると考えている。

[20] http://data1.gfdl.noaa.gov/multi-core/presentations/govett_6b.pdf
[21] http://www.mmm.ucar.edu/wrf/WG2/GPU/
[22] http://srnwp.met.hu/Annual_Meetings/2013/download/thrusday/Philippe_Steiner.pdf
[23] http://data1.gfdl.noaa.gov/multi-core/presentations/sawyer_6b.pdf
[24] http://data1.gfdl.noaa.gov/multi-core/presentations/posey_6a.pdf

参考文献

Michalakes, J. and M. Vachharajani, 2008: GPU Acceleration of Numerical Weather Prediction. *Parallel Processing Letters*, **18**, 531–548.

室井ちあし, 2011: 省電力 GPU コンピューティング. 平成 23 年度数値予報研修テキスト, 気象庁予報部, 53–55.

下川辺隆史, 青木尊之, 石田純一, 河野耕平, 室井ちあし, 2011: メソスケール気象モデル ASUCA の TSUBAME2.0 での実行. ながれ, **30**, 75–78.

7.2 スーパーコンピュータ「京」での asuca の実行 [1]

7.2.1 はじめに

「京」(以下では単に京と記す) は文部科学省の次世代スーパーコンピュータ計画の一環として、理化学研究所と富士通が開発したスーパーコンピュータの愛称である。京は 2011 年 6 月には LINPACK ベンチマークで 8.162PFLOPS を記録して、世界で最も高速なコンピュータシステムの上位 500 位までをランク付けするプロジェクト TOP500 で 1 位になっている [2]。8 コアで構成される 1CPU あたり 128GFLOPS の計算性能を有する SPARC 64VIIIfx を約 88,000 個使用しており [3]、京は超並列による大規模計算の機会を多様な分野の研究者に提供している (野村 2013)。

気象庁は、「HPCI 戦略プログラム分野 3 防災・減災に資する地球変動予測 課題 (1) 防災・減災に資する気象・気候・環境予測研究 目標 1) 地球規模の気候・環境変動予測に関する研究」に独立行政法人海洋研究開発機構 (JAMSTEC) らと共に参加している。この研究の一環として行われる全球力学コアの比較の研究として、第 6 章に述べた asuca-Global を京の上で実行し、他の全球力学コアと比較する予定である。

本節では、その準備作業として緯度経度格子版の asuca を京に移植して実行した結果を述べる。次項以下では、第 7.2.2 項で asuca を京に移植する場合の可搬性について述べ、第 7.2.3 項では京での実行時間について記述する。

7.2.2 京への可搬性

京でユーザーに提供されるのは、標準的な Linux の環境だが、ジョブ投入を行う時に使用するモジュールの作成は指定のクロスコンパイラを使用する必要がある。

asuca は出力データの形式として NuSDaS[4]、NetCDF[5]、4byte 浮動小数点形式が選択可能である。4byte 浮動小数点形式で出力を行う場合には、物理過程ライブラリ以外のライブラリを導入する必要はないが、ある程度の大きな格子数と長い予測期間の計算を行う場合は、実行後のデータ転送やディスクスペースの都合により、NuSDaS、または NetCDF で出力を行う必要があり、これらのライブラリも導入する必要がある。これらのライブラリの作成についても、クロスコンパイラのオプションを調整する必要がある。しかし、クロスコンパイラを使ったモジュール作成環境と、モジュールとライブラリに適合するオプションの設定さえ行っておけば、プログラムの書き直しはほとんど必要ない [6]。従って、前節に述べた GPU を使用する場合に比べると、京の上では比較的簡単に大規模並列の計算を試みることができる。ただし、本稿の執筆時点では、物理過程のメモリ共有並列を京の上で実行させる場合の設定の一部に問題が残っている。

図 7.2.1 に京での実行結果の例を示す。この図は 2013 年 6 月 28 日 00UTC を初期値として、緯度方向、経度方向ともに 0.05 度の格子を 601 個、鉛直方向は上空約 20 km までを 57 層で、Δt =20 秒で asuca を実行した場合の 12 時間予測における前 6 時間降水量と海面更正気圧である。この計算には積雲対流を扱う物理過程が含まれていないので、雲物理過程の働きによって短期間に大きな熱量が放出されて、対流による鉛直流が強く表現される場合があり、Δt は短めに設定してある。

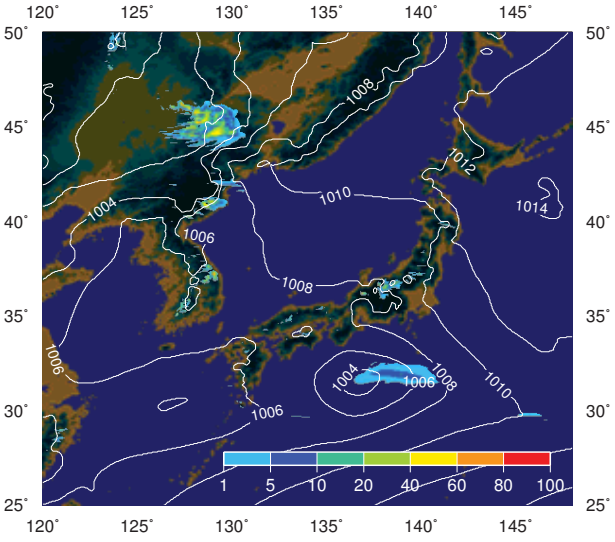

図 7.2.1 2013 年 6 月 28 日 00UTC からの 12 時間予測の海面更正気圧 (コンター、単位は hPa) と前 6 時間降水量 (陰影、単位は mm)

7.2.3 京での計算時間

京での asuca の計算性能を調べるために、前項の計算を並列の設定を変えて実行した結果をこの項に示す。

表 7.2.1 の上段に、1CPU あたり 1MPI プロセスを割り当てて、使用する CPU の数を変えて実行した場合の実行時間を示す。60CPU を使用した場合の実行時間に対して、120CPU を使用した場合の実行時間は 1/2、240CPU を使用した場合は 1/4、360CPU を使用した場合は 1/6 に近い値になっている。表 7.2.1 の下段は、1CPU に 2MPI プロセスを割り当てた場合の実行時間

---

[1] 坂本 雅巳

[2] 京は 2011 年 11 月にも 1 位になった。

[3] 京では、1 つの計算に対して、最大で 82,944CPU までが利用可能である。ただし、この設定が使用できるのは各研究課題で必要性を認められたユーザーのみである。

[4] NuSDaS は、NWP Standard Dataset System の略で、数値予報格子点データを格納するために作られたデータ形式である。C および Fortran で NuSDaS データを読み書きするためのサブルーチン集を NuSDaS ライブラリという。

[5] Network Common Data Form。配列指向のデータアクセスライブラリ NetCDF ライブラリによって作成される。

[6] 総称名の手続きなどにそのまま移植できない部分があり、これらの改修は行った。

表 7.2.1 使用する CPU の数と実行時間

1CPU あたり 1MPI プロセスの場合

CPU の数	60	120	240	360
MPI プロセス数	60	120	240	360
実行時間 [秒]	13869	7342	3833	2689
60CPU の実行時間を 1 とした実行時間比	1	0.53	0.28	0.19

1CPU あたり 2MPI プロセスの場合

CPU の数	60	120	240	360
MPI プロセス数	120	240	480	720
実行時間 [秒]	8313	4195	2388	1864
60CPU の実行時間を 1 とした実行時間比	1	0.50	0.29	0.22
1CPU あたり 1MPI プロセスに対する比率	0.60	0.57	0.62	0.69

である。同様に CPU の数にほぼ反比例して実行時間は減少している。1CPU あたり 1MPI プロセスを割り当てた場合の実行時間に対する比は概ね 0.6 から 0.7 倍であり、CPU の数を増やす場合よりも計算時間を短縮する効果は小さいが、CPU の数を変えずに MPI プロセスを増やすだけでも計算時間がかなり短縮されている。京ではメモリ非共有並列とメモリ共有並列を併用するアプリケーションが推奨されており、asuca も設計としてはそのように作られている。CPU の数を増やさずに MPI プロセスを増やすだけで、実行時間短縮の効果が大きいことは、メモリ共有並列計算の効率があまり高くないことを示唆している可能性がある。

図 7.2.2 の上図に 1CPU あたり 1MPI プロセスで、使用する CPU の数を 120、240、360 にした場合の計算時間を、出力処理 (output)、物理過程 (physics)、力学 (dynamics) に分けて示す。入力は力学の一部に含めている。また、asuca では、非同期通信を用いることで、通信を行っていない計算プロセスを待たせずに計算を続けさせる工夫が施されているので、通信単体の実行時間を分離していない。図 7.2.2 の上図では、出力、物理過程、力学のいずれも CPU の数にほぼ反比例して実行時間が短縮されている。

図 7.2.2 の下図に 120CPU 120MPI プロセスの実行時間（1CPU あたり 1MPI プロセス、上図の上と同じもの）と、120CPU 240MPI プロセス (1CPU あたり 2MPI プロセス) の実行時間の比較を示す。上図の上に示した 120CPU 120MPI プロセスから中央の 240CPU 240MPI プロセスへと CPU を 2 倍に増やす場合は、CPU を増やす前に比べて力学が 55%、物理過程が 50%、出力が 50%の実行時間になる。下図の下に示した 120CPU 240MPI プロセスへと MPI プロセス数だけを 2 倍に増やした場合では、力学が 64%、物理過程が 50%、出力が 57%になる。物理過程については、CPU を変えないで MPI プロセスを増やすだけで同様の短縮効果が得られていることが確認できる。物理過程がメモリ共有並列を十分に利用できていないのは、前出の移植時の設定の問題によるものと思われる。出力にかかる時間も MPI プロセスを増やすだけで、ある程度短縮されている。これは、1 時間毎の出力を行う設定で計算を行ったことと、asuca が複数の MPI プロセスを切り替えて出力させる仕組みになっていることに関係があると考えられる。出力にかかる時間は全体の実行時間に比して短いので、どの程度の MPI プロセス数が適当であるかは、力学、物理過程に費やす時間とのバランスを考慮すべきである。力学については、隣接計算領域を担当するプロセスを同一 CPU 内に割り当てる設定ができれば、他の CPU との通信が減るので、MPI プロセス数だけを増やした場合の時間短縮効果が幾分大きくなる可能性はある。しかし、京で推奨された設定を考えると、メモリ共有並列計算の効果を高める工夫を行う方が良さそうである。

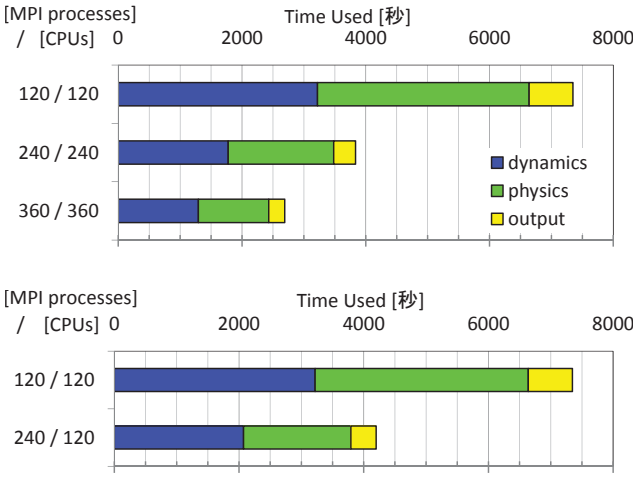

図 7.2.2 2013 年 6 月 28 日 00UTC からの 12 時間予測を行う場合の出力 (output、黄色)、物理過程 (physics、緑色)、力学 (dynamics、青色) の計算時間。縦軸は使用した MPI プロセス数と CPU の数、横軸は計算時間 [秒]。

7.2.4 まとめ

この節では、京での asuca の実行時間の計測結果を示した。前節の GPU クラスタを利用する場合と比べれば、プログラムを作り直す必要がないので、京への移植は比較的容易であった。現時点での測定結果でも、asuca はスケーラビリティの良さを京でも実現できている。物理過程のプログラムの移植時の問題が解消し、メモリ共有並列の効果を高める工夫を行っていけば、asuca は京での実行時間を更に短縮できるであろう。

参考文献

野村稔, 2013: 世界のスーパーコンピュータの動向. 科学技術動向, **137**, 11–18.