チュートリアル

科学技術計算の分野で始まったGPU（Graphic Processing Unit）による汎用計算は、製造業、医療、金融など様々な分野へ広まり、モバイル、組み込みの分野でも使われ始めています。当初は専用言語を使う必要がありGPUによる汎用計算は気軽に始められるものではありませんでしたが、用途が広がると同時に、従来の言語でGPU向けにプログラミングする環境が整ってきました。OpenACCはその中の一つで、既存のプログラムにディレクティブと呼ばれるヒント情報を追加するだけで、コンパイラが自動でGPUコードを生成する開発手法です。今回のチュートリアルでは、全４回のシリーズとして、このOpenACCを使ってアプリケーションを効率良くGPUで加速する方法を解説します。

OpenACCで始めるGPUコンピューティング：OpenACC概要

成瀬　彰

1　はじめに

科学技術計算に使われる計算機システムは、多様化の時代を迎えている。2008年のTop500スパコンランキング[1]では、90%以上のシステムがx86プロセッサを使用しており、HPCの世界は均質だった。その後、アクセラレータとしてNVIDIA GPUが普及、Intelはメニコアプロセッサの XeonPhiをリリース、米CORALプロジェクト[2]と供にIBMのPowerプロセッサがHPCに復帰、一部システムではARMプロセッサも使われ始め、特定のアプリケーションに特化してFPGAやDSPも活用されている。また、一つの計算ノードに、複数の種類のプロセッサを搭載することも珍しくない。競争はイノベーションを加速する。多種多様なプロセッサの存在は、高い性能を求めるユーザには朗報である。

しかし、ソースコードのポータビリティの確保が問題である。プログラム言語や開発手法も多様化しており、事前の検討が不十分だと、特定のプロセッサ以外では、開発したプログラムが動作しない、もしくは、まともな性能が出ないという状況になりかねない。後者は、性能ポータビリティと呼ばれる問題だが、多様なプロセッサが存在する中で、どうすれば性能ポータビリティを確保できるのかは、大きな課題となっている。多様化が進む中での救いは、これらプロセッサ間に少なからぬ共通点が存在することである。

一点目の共通点は、全てのプロセッサが「並列性」を高める方向で、高性能化を指向していることである。CPUは、CPUコア数を増やすと当時に、SIMD幅を広げる傾向にある。例えばIntel Xeonプロセッサのコア数は10年前の2コアから18コアと9倍になり[3]、SIMD幅も10年前のSSE2の128bitと比べてAVX-512は512bitで間もなく4倍になる[4]。つまり、粒度の異なる2つの並列性を上げて、性能を高める方針を採っている。GPUは、CPUとは異なるが、粗粒度のブロック並列性、細粒度なSIMT並列性[5]、やはり粒度の異なる2つの並列性を高めることで高性能を実現するアーキテクチャである。近年の、そして、将来のプロセッサを有効に使うには、単にアプリケーションから大量の並列性を抽出するだけでなく、異なる粒度に合った並列性を抽出する必要があると言える。

もう一つの共通点は、メモリシステム階層の深化である。CPUのメモリ階層は、主記憶（典型的にはDDRメモリ）、同一チップ内のCPUコア間でシェアされる共有キャッシュ、CPUコア内のプライベートキャッシュで構成される。一方、GPUのメモリ階層は、CPUの主記憶、GPUの主記憶（DDRメモリよりスループットの高いGDDRメモリが典型的）、複数階層のキャッシュ、スクラッチパッドメモリで構成される。

続きはWebで
日本計算工学会誌「計算工学（Vol.21, No.2）」HP:
http://www.jsces.org/Issue/Journal/

筆者紹介

なるせ　あきら
1996年、名古屋大学大学院工学系研究科修士課程修了（情報工学専攻）。同年、富士通研究所に入社、大規模サーバ開発、HPCシステム開発など、様々なプロジェクトに参加、関連するハードウェアやソフトウェアの研究開発に従事。2009年、山下記念研究賞受賞。2010年、SACSIS2010最優秀論文賞受賞。2013年、NVIDIAに参加、シニアデベロッパーテクノロジーエンジニアとして様々なアルゴリズムとアプリケーションのGPU向け並列化に従事、GPUコンピューティングの普及に努めている。

近年のプロセッサには、このような並列化・階層化という共通性があるのに加え、内在するアーキテクチャの複雑性をハードウェアレベルで隠蔽せずに、ソフトウェアに解放する傾向にある。つまり、プロセッサの機能を最大限に活用するには、これまで以上に、その機能をソフトウェアから使いこなす必要があり、そのためプロセッサベンダーは各種APIやプログラミングモデルを開発者に提供している。ベンダー提供のAPIやプログラミングモデルを使うと、そのベンダーのプロセッサ上でのアプリケーション性能は向上するが、コードのポータビリティが問題として残る。高いレベルで性能とポータビリティを両立するプログラミングモデルが存在するのが理想であるが、一般的には、性能とポータビリティはトレードオフの関係にあり、単一のプログラミングモデルでそれを両立させるのは容易ではない。従って、大規模なアプリケーションになるほど、複数のプログラミングモデルを組み合わせて使用するケースが増えている。

2　高並列プロセッサ向けプログラミングモデル

本稿はOpenACCによる高並列プロセッサ向けのプログラム開発手法を説明するものだが、だからといってOpenACCのみを使用してプログラム並列化を進めよと推奨するものではない。OpenACCはシンプルで強力なプログラミングモデルだが、決して万能ではない。世の中には様々なプログラミングモデルが存在し、それぞれ長所と短所がある。我々が推奨しているのは、OpenACCを中核として、様々なプログラミングモデルのいいところを利用して、効率良くプログラム開発を進める方法である。

HPC分野で使用されているプログラミングモデルは、大別すると次の4種類である。

1. 標準言語
2. ライブラリ
3. ディレクティブ
4. 拡張言語

以下では、これらプログラミングモデルのそれぞれの特徴に触れた後、我々の推奨方法を説明する。

2.1　標準言語

C/C++やFortranなどの標準言語で開発されたソースコードは、当然ながら、最もポータビリティが高い。しかし、現段階で標準言語が提供している機能は、高並列プロセッサにとっては十分とは言えない。標準言語コミュニティ内でも高並列時代への対応が始まっているが、将来の規格で、高並列プロセッサ向け機能をどう標準化するか、議論されているところである。また、新しい規格がリリースされても、コンパイラベンダーから新機能が安定した状態で提供されるまでには数年かかるのが常である。従って、今後数年間は、高並列を指向するプロセッサ向けに、標準言語で高性能を実現するのは難しい状態が続くと考えられる。

2.2　ライブラリ

BLASなどの標準ライブラリの使用は、ポータビリティの高い方法である。APIが標準化されているため、使用プロセッサを変更しても、ソースコードを変更することなく、ライブラリ実装を切り替えるだけで、プログラムを動作させることができる。また、プロセッサベンダーは、標準ライブラリに関しては、高度にチューニングされた実装を提供する傾向にあり、高性能も実現できることが多い。例えば、BLASの中でも使用頻度の高い行列積ルーチンは、Intelプロセッサ上ではIntel MKL[6]が、NVIDIA GPU上ではNVIDIA cuBLAS[7]がそれぞれ最速であり、どちらもユーザコードではほぼ実現不可能なレベルの性能を提供する。標準ライブラリの使用は、ポータビリティと性能を高いレベルで両立できる方法であるが、残念ながら、標準ライブラリだけで開発できるアプリケーションはほとんどない。

プロセッサベンダーは標準ライブラリ以外にも、ユーザのために独自ライブラリを開発・提供することがある。独自ライブラリ使用のメリットは高性能であるが、APIが標準化されていないため、ポータビリティが問題となる。ただ、ライブラリの適用箇所は局所的であることが多いので、ベンダー独自ライブラリを使っても、アプリケーション全体としてはポータビリティを高い状態に保つことができる。

2.3　ディレクティブ

ディレクティブは、標準言語で開発された既存のプログラムに対して、コンパイラへのヒントを追加する方法である。一般的には、ディレクティブは、C/C++の場合はpragma行として、Fortranの場合はコメント行として既存コードに追加される。ディレクティブは、標準言語ではサポートされていない機能を提供する手段として使われることが多く、プログラム開発者はディレクティブを使ってコンパイラに対してどのようにコードをビルド・最適化すべきかを指示できる。

ディレクティブにはコンパイラベンダー独自のものもあるが、広く使われているのは、OpenMP[8]やOpenACC[9]のように、産学グループで標準化され、複数のコンパイラベンダーからサポートされるディレクティブである。標準化されたディレクティブを使ったプログラムであれば、使用するプロセッサやコンパイラを変えても、プログラムをビルド・実行することができる。

一般的には、ディレクティブで最適化したプログラムは、ハンドチューニングしたコードと比べると、１～２割程度低い性能となることが多い。ディレクティブが提供する機能をフルに使いこなせば、ハンドチューニングコードに近い性能を引き出すことも可能だが、そのようにディレクティブを追加すると、対象プロセッサに特化したコードとなりやすく、性能ポータビリティが低下する。また、ソースコードの可読性も低下する。それではディレクティブのメリットが損なわれてしまう。１～２割の性能ロスの代償はあるも

のの、コードのポータビリティと可読性を確保できるため、ディレクティブは多くのアプリケーション、特に大規模なアプリケーションで採用されている。

また、追加したディレクティブは、ディレクティブに未対応のコンパイラでは認識されないが、ビルド・実行はできる。ソースコードを標準コンパイラでもビルドできる状態に保てるという意味でも、ディレクティブはポータビリティの高い方法である。

2.4　拡張言語

拡張言語とは、例えばCUDA[10]やOpenCL[11]のように、特定のプロセッサの機能を最大限に引き出すために拡張された言語である。CUDAとOpenCLはベース言語としてC/C++を採用しているが、GPU等の高並列プロセッサの性能を活用しやすいよう拡張されている。拡張言語でのプログラミングは低レベルの記述が必要になりがちであるが、拡張言語を使わないと活用できないハードウェア機能が存在するため、非常に高い性能が求められるアプリケーションでは拡張言語が使用される。

一方で、拡張言語で開発されたコードは、例えばCUDAコードやOpenCLコードは、標準のC/C++コンパイラではビルドすることができない。また、拡張言語はその特性から、特定プロセッサでしかサポートされない傾向にあり、必然的にコードのポータビリティは低くなる。拡張言語は、性能面では魅力的だが、ポータビリティに難点があるため、採用の前には十分な検討が必要である。

2.5　プログラミングモデルの併用

一つのプログラミングモデルで、性能とポータビリティを高いレベルで両立するのは難しい。しかし、単一のプログラミングモデルに依存する必要はない。ほとんどのアプリケーションでは、複数のプログラミングモデルを併用して、プログラムを開発することが可能である。従って、高並列プロセッサ向けに、高性能、かつ、メンテナンスの容易なコード開発を要求される開発者への推奨方法は次の方法となる。

1. ライブラリで可能な部分はライブラリに置換える
2. コードの並列化は、最も生産性の高いプログラミングモデルで開始する
3. 性能が必要なモジュールのみ、低レベルなプログラミングモデルを適用する

ライブラリは、標準ライブラリの使用が望ましいが、ニーズによってはベンダーライブラリもOKである。2.と3.の部分は、言い換えると、ディレクティブの使用から始めて、必要な部分だけ低レベルの拡張言語を使うということである。現時点では、GPUに対応し標準化されたディレクティブはOpenACCのみであり、GPUをターゲットプロセッサとするなら、高性能とポータビリティを両立させる現実的なアプローチはOpenACCでコード開発を始めることとなる。

3　OpenACCとは

HPC分野では、メニコアアーキテクチャやGPUなどの高並列プロセッサの出現とともに、高級言語で性能を引き出したい、コードのポータビリティと実効性能を高いレベルで両立させたいというプログラマーからのニーズが高まっていた。それに応える形で、OpenACCは、アクセレレータなど並列性の高いプロセッサ向けのプログラミングモデルとして2011年に提案された。OpenACCはディレクティブ方式を採用しており、コード開発者はコード内の並列化が可能な部分やその並列化の方法を、ディレクティブを追加することでコンパイラに指示することができる。そして、コンパイラはその情報を元に、様々な並列プロセッサ向けのコードを生成する。

3.1　アクセレレータ・モデル

OpenACCで開発されたコードが、様々なプロセッサで実行できるようにするために、OpenACCではアクセラレータ・コンピューティングを抽象化したモデルが定義されている。このモデルは、対象となるプロセッサでは様々な粒度の並列性が必要であること、メモリシステムは容量・速度・アドレスの異なる階層から構成されることを想定している。アクセラレータ・モデルの目的は、現時点で広く使われているプロセッサに対してだけでなく、これから出現するであろう全てのプロセッサに対して、OpenACCを適用可能にすることである。

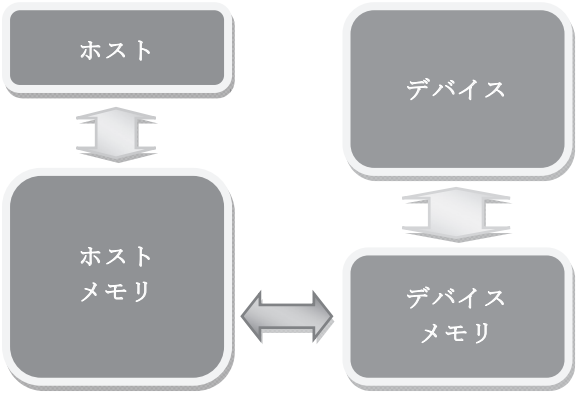

図1　OpenACCのアクセラレータ・モデル

OpenACCの実行モデルでは、プログラム実行を主導するのはホストであり、デバイス（アクセラレータ）はホストの指示に従って動作する。ホストは、プログラム内の計算負荷の重い部分をアクセラレータにオフロードし、必要な場合はホストメモリとデバイスメモリ間でデータを転送する。図1にアクセラレータ・モデルの概略を示す。この図は、近年のGPU搭載ノードで見られるような、ホストとしてCPUを使い、デバイスとしてGPUを使う形態を模している。この形態のシステムでは、ホストとデバイスでメモリ空間が分離しているので、アクセラレータにオフロードされる計算で

使われるデータは、事前に、OpenACCランタイムによりホストからデバイスへ転送される。同様に、デバイスメモリに書き込まれたアクセラレータの計算結果は、ホストプロセッサがそれを必要とする前に、OpenACCランタイムによりデバイスからホストへと転送される。

図1より、ホストとデバイスが別個のプロセッサであることをこのモデルは前提としているように見えるかもしれない。実際には、ホストとデバイスは同じアーキテクチャでも問題なく、同一チップでも構わない。その場合のプログラムの実行方式は、マルチコアCPU上でのOpenMPプログラムの実行に近いものとなる。メモリも同様であり、ホストメモリとデバイスメモリが同一のメモリでも問題ない。この場合は、単にホストとデバイス間のデータ転送が不要になるだけである。

3.2　メリットと制限

OpenACCのメリットは、高級言語でアクセラレータ向けコード開発を行えること、そして、開発したコードの適用対象が特定プロセッサに限定されないことである。プログラマは、OpenACC化した単一のソースコードから、様々なプロセッサ上で動作するプログラムを開発できる。例えば、PGIのOpenACCコンパイラ[12]は、コンパイルオプションを変えるだけで、単一のソースコードからNVIDIA GPU、AMD GPU、Intel マルチコアCPU向けのコードを生成できる。しかし、この高いポータビリティは性能面では制限となることもある。

OpenACCのアクセラレータモデルは、想定される様々なアクセラレータの共通項に基づいて決定されている。これは、特定のアクセラレータに固有な機能をモデルに取り入れることにより、OpenACCで開発されたコードのポータビリティが低下するのを回避するためである。従って、CUDAやOpenCLのような、低レベルなプログラミングモデルで容易に実装できる最適化手法の中には、OpenACCでは実現するのが難しいものもある。例えばNVIDIA GPUの場合、チップ内の「共有メモリ」と呼ばれるスクラッチパッドメモリは、CUDAなら比較的容易に活用できるのに対して、OpenACCには「cache」ディレクティブが用意されているものの、OpenACCで共有メモリを活用するのは容易ではない。同様の事例は、他のプロセッサやアクセラレータにも存在する。特定のプロセッサにのみ有効な最適化手法ほど、そのプロセッサに密接した低レベルなプログラミングモデルなら容易に実現できるにもかかわらず、OpenACCでは実現が難しい傾向にある。

性能が重視されるアプリケーションにOpenACCを使うのか、それとも低レベルな言語を採用するかは、それぞれのメリットとデメリットを把握した上で、開発者が慎重に決定する必要がある。しかし、性能が非常に重要で、低レベルな言語を使うしかない場合であっても、OpenACCは低レベル言語と組み合わせて使うことができるので、アプリケーションの大部分はOpenACCで高速化して、コード内の必要最小限の部分だけ低レベル言語を採用するアプローチも可能である。

3.3　ディレクティブの構文

本節ではOpenACCの構文を説明する。OpenACCの様々なディレクティブの機能に関しては、後述するサンプルコードを事例として、アプリケーションをOpenACCで高速化するプロセスの各過程で、関連するディレクティブの機能を詳細に説明する予定である。なお、OpenACCの構文はOpenMPの構文を踏襲しており、OpenMPの経験者には違和感が少ないだろう。

OpenACCディレクティブの中身は、基本的には言語に関係なく共通であるが、コードへの追加方法は、以下に示す通り、C/C++ではpragma形式、Fortranでは特殊なコメント行形式で追加する方法である。

[C/C++]

```
#pragma acc directive-name [clause [clause [...]]]
```

[Fortran]

```
!$acc directive-name [clause [clause [...]]]
```

キーワードaccの後に記述された内容、上記例の「*directive-name [clause [clause [...]]]*」の部分がディレクティブの中身である。この中の*directive-name*の箇所に記載する内容がコンパイラへの指示内容のメインである。角括弧内はオプションであり、*clause*の箇所に記載する内容は前記*directive-name*の指示内容をより詳細化するためのものである。一つの*directive-name*に対して、複数の*clause*を付けることもできる。

OpenACCディレクティブの中には、コード領域に対して適用されるディレクティブがある。例えば、kernelsディレクティブはコード内の並列化可能領域をコンパイラに指示するものであるが、C/C++の場合は、kernelsディレクティブを追加した行の次行から始まる波括弧で囲まれたコード領域、いわゆる構造ブロックが、その適用対象領域となる。

[C/C++]

```
#pragma acc kernels
{
    (computations …)
}
```

Fortranでは、C/C++のように構造ブロックをコード内に記述するのが一般的ではないため、kernelsディレクティブを追加した行の次行から、それに対応する終了ディレクティブ（end kernelsディレクティブ）を追加した行の前行までのコード領域が、適用対象領域となる。

[Fortran]

```
!$acc kernels
    (computations …)
!$acc end kernels
```

4　OpenACCによるアプリケーション並列化の推奨方法

本章ではアプリケーションをOpenACCで並列化する場合の推奨方法を説明する。OpenACCに限らず、アプリケーションを並列化するときに、コード開発者が頻繁に直面する問題の一つは、プログラムの計算結果が並列化前と並列化後で一致しないことである。並列化により演算順序が変わり、浮動小数点演算の丸め誤差が変わったことがその原因かもしれない。これは、数学的にはどちらも正しいので、許容すべき不一致と言える。しかし、並列化の過程でバグが混入、もしくはバグが表面化したのかもしれない。こちらは修正が必要な不一致である。結果が一致しない原因は、許容して良いものと、修正が必要なバグが存在するので、それぞれ明確に切り分ける必要がある。しかし、開発者がプログラムの多数の箇所を並列化した後に、この結果が一致しない問題に直面すると、並列化したコードの中で、どの部分はOKで、どの部分は修正が必要なのか、その切り分け作業に膨大な時間を費やすことになる。場合によっては、並列化前のコードに戻って、全てをやり直すことになりかねない。

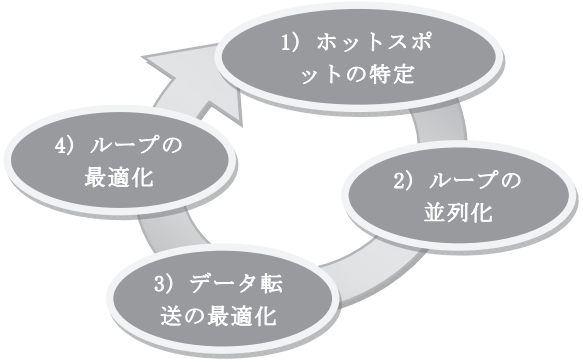

図2　OpenACCによるアプリケーション並列化のサイクル

それ故に、我々はアプリケーションを一挙に並列化するのではなく、重要な部分から一つずつ順番に並列化していく方法を推奨している。その具体的なプロセスは、図2の通りである。1）コード内の実行時間の長い場所（ホットスポット）の特定、2）ホットスポットをOpenACCで並列化、3）ホストとアクセラレータ間の無用なデータ転送を削減、4）対象アクセラレータ向けにループを最適化。そして、アプリケーション全体の性能が所望の性能に達するまで、このプロセスを繰り返すのである。

4.1　ホットスポットの特定

アプリケーションの並列化に取り組む前に、まず、アプリケーションのどのサブルーチン、どのループが全実行時間のどの程度の時間を要しているかを把握することは重要である。アムダールの法則[13]によれば、並列化したアプリケーションの最終性能は、アプリケーションの逐次部分の実行時間に支配される。つまり、並列性能を高めるには、逐次実行時間の削減が重要ということである。逐次実行時間の長いところ、CPU実行時のホットスポットから並列化に取り組むというのは、効率良く並列化の効果が得られるという意味でも理にかなっている。Gprof[14]などフリーのツールから、より詳細に解析できる有償のツールまで、CPU向けの性能解析ツールは豊富に存在するが、効率良くホットスポットを特定できるのであれば、ツールは何でもOKである。ユーザタイマーでも構わない。

4.2　ホットスポットの並列化

高速化すべき場所を決定したら、次はその中のどの部分が、どのループが並列化できるかを調査し、OpenACCディレクティブを使って、どのループをどう並列化するかをコンパイラに指示する。コンパイラへの指示は、最初から完璧を目指す必要はない。多くのOpenACCコンパイラは、どう並列化したかのフィードバック・メッセージをコンパイル時に出力するので、それを参考にして並列化方法を詳細化していけばよい。

また、この段階では、ホストとデバイス間で冗長にデータ転送が行われていることが多く、その結果として、アクセラレータを使用する前より、性能が低下することも多い。この段階で無理にアクセラレータ上の計算性能を最適化する必要はない。

4.3　無用なデータ転送の削減

CPU+GPUのように、現在のアクセラレータ搭載ノードでは、ホストとアクセラレータでメモリが分離していることが多い。この条件で正しい結果を得るためには、コンパイラは、ホストとアクセラレータ間で適切にデータを転送するコードを生成しなくてはいけない。しかし、ソースコードからそのアプリケーションに関してコンパイラが読み取れる情報は、その一部でしかない。その結果として、正しい計算結果が得られることを保証するため、コンパイラが生成するコードでは、必要以上にホストとアクセラレータ間でデータ転送が行われることが多い。コード開発者はそのアプリケーションに関してコンパイラより多くの知識を持っている。加えて、OpenACCにはデータ転送をコントロールするディレクティブが用意されている。コード開発者はこのディレクティブを使って、できる限り長い間、データをアクセラレータメモリ上に滞在させ、ホストとアクセラレータ間のデータ転送を最小化することができる。現行のGPU搭載ノードは、CPUとGPU間が、主記憶のメモリバンド幅と比べると、かなり帯域の狭いPCIエクスプレスで接続されていることもあり、このデータ転送の最適化がアプリケーション性能に寄与する効果は大きい。

4.4　高並列向けにループを最適化

シンプルにループが並列化可能であることをコンパイラに指示するだけでも、コンパイラは自身の知識に基づいて、様々な粒度の並列性をコードから抽出し、

それをターゲットプロセッサの並列特性にマップする。しかし、前述の通り、コンパイラが持つアプリケーションの知識には制限があるので、コード開発者がより詳細に並列化方式を指示することで、更に性能を向上させることも可能である。

それよりも性能向上の可能性が高いのは、コードの再構成である。計算負荷の重い部分は、複数の多重ループから構成されていることが多い。往々にして、これらコードはマルチコアCPUをターゲットに開発されていることが多く、マルチコアCPUには十分な並列性、つまり数～数十程度の並列性は抽出できている。しかし、ループの構成を見直すことで、より多くの並列性、数千、数万、それ以上の並列性を抽出することも可能である。GPUは超並列プロセッサであり、その性能を最大限に活用するには、数万のオーダーの並列性が必要である。このようなコードの再構成は、アクセラレータ上で高い性能を得られるのはもちろんのこと、今後、全てのプロセッサが高並列へと向かっていることを考えると、将来的には全てのプロセッサにとって有意義と言える。

4.5　結果検証とツール使用

本章の最初に述べた通り、OpenACCでアプリケーションを並列化・高速化する過程では、継続的に計算結果が正しいことを確認することが非常に重要である。シミュレーション結果の一部、例えばある物理量が格納されている配列内の最小値・最大値・平均値など、また収束計算なら残差・収束回数など、計算結果を検証できるデータを定期的に出力するコードを追加することを推奨する。定常的に計算結果が正しいことを検証できれば、コード開発の手戻りが減り、結果的にはコード開発を効率良く進めることができる。また、Git[15]などのソースコードのバージョン管理ツールを使うことも合わせて推奨する。こまめにスナップショットを保存しておけば、それまでの作業工数を大幅に無駄にすることなく、素早く正常なソースコードに戻ることができる。

5　サンプルコード: ヤコビ反復法

本チュートリアルの2回目と3回目では、比較的シンプルなサンプルコードを題材として、前章で説明した、OpenACCによるアプリケーション並列化のプロセスに沿って、具体的なOpenACCディレクティブの使い方とその機能を説明していく。最初に用いるサンプルコードは、ヤコビ反復法による2次元ラプラス方程式の解法である。本サンプルコードのメインループ部分は、C/C++とFortranで、それぞれ以下の通りである。

[C/C++]

```
1    while ( error > tol && iter < iter_max ) {
2      error = 0.0
3      for (int j = 1; j < N-1; j++ ) {
4        for (int i = 1; i < M-1; i++ ) {
5          Anew[j][i] = 0.25 *
6            (A[j][i-1] + A[j][i+1] + A[j-1][i] + A[j+1][i]);
7          error = fmax(error, fabs(Anew[j][i] – A[j][i] ));
8        }
9      }
10     for (int j = 1; j < N-1; j++ ) {
11       for (int i = 1; i < M-1; i++ ) {
12         A[j][i] = Anew[j][i];
13       }
14     }
15     if (iter % 100 == 0) printf( “%d, %f¥n” , iter, error)
16     iter++;
17   }
```

[Fortran]

```
1    do while ( error > tol .and. iter < iter_max )
2      error = 0.0
3      do j=2, N-1
4        do i=2, M-1
5          Anew(i,j) = 0.25 * &
6            (A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1))
7          error = max(error, abs(Anew(i,j) – A(i,j) ))
8        end do
9      end do
10     do j=2, N-1
11       do i=2, M-1
12         A(i,j) = Anew(i,j)
13       end do
14     end do
15     if (mod(iter,100)==0) write(*,*) iter, error
16     iter = iter + 1
17   end do
```

計算の内容は、隣接4点の平均を各点の次の値とし、その値と現時点の値との差分が許容値を下回るまで、同じ計算を繰り返すというシンプルなものである。このように収束的に解を求める計算方法は、科学技術計算では典型的な方法であり、このサンプルコードを並列化する過程で習得できるOpenACCの知識・ノウハウは、大規模なアプリケーションをOpenACCで加速するときに大いに役に立つだろう。

■参考文献:

[1] Top500 Supercomputer Sites, www.top500.org
[2] CORAL: Collaboration of Oak Ridge, Argonne, and Lawrence Livermore Labs, asc.llnl.gov/CORAL/
[3] Intel Xeon, ja.wikipedia.org/wiki/Xeon
[4] SSE (Streaming SIMD Extensions), ja.wikipedia.org/wiki/Streaming_SIMD_Extensions
[5] SIMT (Single Instruction Multiple Threads), en.wikipedia.org/wiki/Single_instruction,_multiple_threads
[6] Intel MKL (Math Kernel Library), software.intel.com/en-us/intel-mkl
[7] NVIDIA cuBLAS, developer.nvidia.com/cublas

[8] OpenMP, www.openmp.org
[9] OpenACC, www.openacc.org
[10] CUDA, developer.nvidia.com/cuda-toolkit
[11] OpenCL, www.khronos.org/opencl/
[12] PGI Accelerator Compilers, www.pgroup.com/resources/accel.htm
[13] Amdahl's law, en.wikipedia.org/wiki/Amdahl%27s_law
[14] GNU gprof, sourceware.org/binutils/docs/gprof/
[15] Git, git-scm.com/