TSUBAME 共同利用　平成28年度　学術利用　成果報告書

利用課題名　OpenACC による圧縮性流体解析プログラム UPACS の高速化に関する研究

利用課題責任者　星野哲也

所属　東京大学情報基盤センター

邦文抄録（300 字程度）
次世代のスーパーコンピュータにおいて主流になると考えられる，メニーコアプロセッサを用いた計算環境へのアプリケーションの対応が急務である.IHI が所有するアプリケーションである UPACS_turbo について，メニーコアプロセッサ向けの記述言語として注目されている OpenACC により並列化を行い，TSUBMAE2.5 の 1 GPU を用いた評価の結果，良好な結果を確認している.本課題では，単体の GPU 向けの最適化を行った上で，複数 GPU を効率良く利用するための手法を開発し，実際の大規模なシミュレーションデータによる評価を行うことを目的とする.

Keywords: 5つ程度
OpenACC, GPU

背景と目的

地震や気象予測，または航空機や高層ビルの設計などにおいて，今でこそ一般的となった数値シミュレーションは，計算機の急速な性能向上の恩恵を受けることで発展してきた. しかし近年における計算環境の変化は，アプリケーションの利用者にとって必ずしも喜ばしいものではない. コア単体の性能向上が物理的な制約から頭打ちになりつつある近年においては，計算機はコアの数を増やすことによって性能向上を続けているが，多数のコアを効率良く利用するためには，往々にしてアルゴリズムの見直しや並列プログラムング言語を用いた再実装が必要となるためである. またメニーコアプロセッサの登場など，実行デバイスの形態も多様化してきており，数十万行にも及ぶアプリケーションをその都度再実装することは現実的ではない. 既存のアプリケーション資源を活かしつつ，計算環境の変化に柔軟に対応するための仕組みが求められている.

そこで，既存のアプリケーションの並列化手法として，コンパイラ指示文ベースのプログラミングモデルが注目されている. マルチコア CPU 向けの並列プログラミングモデルである OpenMP や，メニーコアアクセラレータ向けの OpenACC などがこれにあたる. 指示文ベースのプログラミングモデルは，基本的には既存のアプリケーションに対し数行の指示文を加えるだけであり，対応するコンパイラが指示文を解釈し，例えば OpenMP であればマルチコア CPU での並列実行が可能である. 指示文に対応していないコンパイラは指示文行をコメントとして無視するため，元のソースコードを保全できる.

しかし，OpenACC は 2011 年に発表された比較的新しい規格であり，2015 年 10 月に新しい仕様である OpenACC2.5 が発表されたばかりの，未だ発展途上のプログラミングモデルであるため，特に実アプリケーションにおける評価が少ないのが現状である.

本稿では，実際に株式会社 IHI で利用されているアプリケーションである UPACS_turbo の OpenACC による高速化を行った.

UPACS

本研究において対象としている圧縮性流体解析プログラム，UPACS について説明する. UPACS は独立行政法人宇宙航空研究開発機構 JAXA により研究開発されている，航空宇宙分野において要求される様々な流体現象の解析に用いることを目的とした，汎用的な流体アプリケーションである. ただし，今回対象としている UPACS は，JAXA の UPACS をベースとして，株式会社 IHI が開発のために独自の拡張を施したものであり，UPACS_turbo と呼称する.

オリジナルの UPACS では圧縮性流体の数値シミュレーションを並列計算機上で行うために，マルチブロック構造格子法を用いている. マルチブロック構造格子法では，複数の構造格子を非構造的に接続することで，

複雑形状まわりの計算格子を作成し，各々の構造格子を 1 ブロック単位として MPI プロセスに割り当てることで，並列計算機上での実行を可能にしている．なお，1MPI プロセスに複数のブロックを割り当てることは可能であるが，1 ブロックに複数の MPI プロセスを割り当てることはできない．またオリジナルの UPACS は，様々な流体解析プログラムを共通的に利用できるプラットフォームの確立を目的としているため，コードの階層化，データおよび計算手法のカプセル化といった，オブジェクト指向的な考え方を用いて設計されている．これにより，数百ものモジュールを共通のデータ構造を用いて管理可能としている．

以上については UPACS_turbo も共通であり，大きな枠組みは変わっていない．UPACS_turbo においては，翼列周りに発生する乱流を解析するための解法が追加されているのが特徴であり，また計算精度をオリジナルの倍精度から単精度に落としている．

結果および考察

以前までの OpenACC 実装では，陰解法部分がボトルネックとなっていた．この部分の高速化のために CUDA Fortran によるカーネル実装を行い，OpenACC 側から呼び出す実装方式を用いた．CUDA カーネルを使う場合，host_data ディレクティブを用いることで，OpenACC 側で生成されたデバイス上のデータのポインタを CUDA カーネルに渡すことができる．

UPACS_turbo の陰解法部分について簡単に説明する．まず陽解法とは，図 1 のように左辺 Qt+1 を更新する際右辺項には Q_t しか現れない．故に同タイムステップのにおいてデータ依存がないため並列性が高く，GPU などに向いている．一方図 2 のような陰解法は，右辺に Qt+1 が現れる．故に，Qt+1(x,y,z)を計算するためには Qt+1(x-1,y,z), Qt+1(x,y-1,z), Qt+1(x,y,z-1)の計算が完了していなければならないため陽解法と比較して並列性が低く，GPU 向きではない．しかし陰解法は収束速度が早いため，アプリケーション全体でみるとどちらが速いかは明らかではない．

$$
\begin{aligned}
Qt+1(x,y,z) = {} & a1*Qt(x,y,z) + a2*Qt(x\text{-}1,y,z) \\
& + a3*Qt(x,y\text{-}1,z) + a4*Qt(x,y,z\text{-}1) \\
& + a5*Qt(x+1,y,z) + a6*Q\_t(x,y+1,z) \\
& + a7*Qt(x,y,z+1)
\end{aligned}
$$

図 1 陽解法

$$
\begin{aligned}
Qt+1(x,y,z) = {} & a1*Qt(x,y,z) + a2*Qt+1(x\text{-}1,y,z) \\
& + a3*Qt+1(x,y\text{-}1,z) + a4*Qt+1(x,y,z\text{-}1) \\
& + a5*Qt(x+1,y,z) + a6*Q\_t(x,y+1,z) \\
& + a7*Qt(x,y,z+1)
\end{aligned}
$$

図 2 陰解法

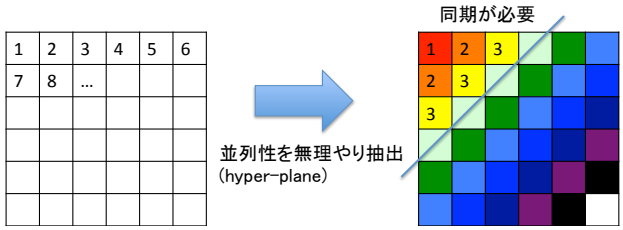

図 3

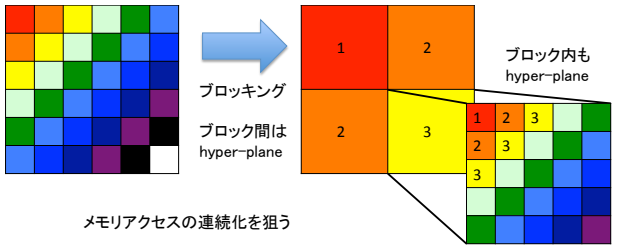

図 4

UPACS において用いられている陰解法の並列化手法である Hyper Plane 法(超平面法)は図 3 の右のような順番で解くことで，データの順序依存関係を保ちながら並列化する手法である．しかしこの方法はメモリアクセスが非連続になるため，一般的に性能が落ちる．

そこで今回我々は，この Hyper Plane 法の高速化手法として，図 4 のようなブロッキングを検討した．ブロッキングの際，例えば小ブロックを 32x1x1 のブロックとすることで，ブロック内部での依存関係を X 方向のみに限定できる．これにより，Qt+1(x,y,z) を更新する際に，依存のある a2*Qt+1(x-1,y,z)$以外の点は各スレッドが連続的読み込むことができるため，読み込みの局所性が高くなる．Qt+1(x,y,z) の更新と a2Qt+1(x-1,y,z) の読みは逐次で行う必要があるが，UPACS では係数 ai を計算するために多数の点にアクセスする必要があ

るため，結果として高速化が可能である．適用の結果を図 5 に示す。

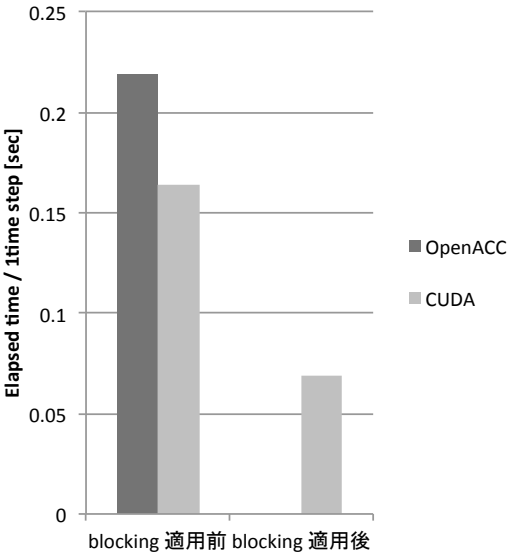

図 5

この最適化により，CUDA 版においては 2.39 倍の高速化が得られているが，この手法は必ず同期を必要とするため，同期構文を持たない OpenACC では適用できない手法である．

また，今までの OpenACC 版 UPACS_turbo では，逐次実行部分が実行時間の大半を占めていることが問題だった．原因は fortran の組み込み関数である reshape が，OpenACC 対応コンパイラである PGI コンパイラでは非常に低速であったためである．その部分について最適化を施した．図 6 が最適化を施す前であり，図 7 は最適化適用語である．報告書のフォームが Word であるため図表の自由が効かず，見辛いと思うが，図 6 と図 7 はどちらも 2 タイムステップ分の実行時間を示しており，図 6 で空白の部分が CPU で実行されている部分であり，図 7 では空白部が明らかに少なくなっていることがわかる．また，本実験については TSUBAME 環境ではなく，P100 GPU にておこなったものである．

図 6

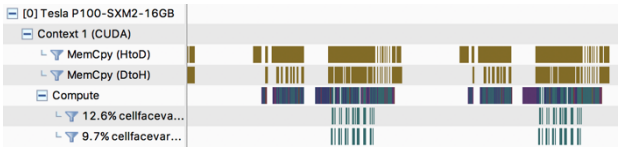

図 7

まとめ、今後の課題

当初の目的であった，大規模環境下でのロードバランシングなどは完了していないため，今後は大規模並列環境での高速化を目指す．