Vol.2016-HPC-153 No.4
2016/3/1

圧縮性流体解析プログラムのOpenACCによる高速化

星野 哲也[1,2]　松岡 聡[2]

概要：航空機の開発などに用いられる圧縮性流体解析アプリケーションには多大な演算パワーが必要とされ，近年一般的になっている演算アクセラレータを用いたスーパーコンピュータの利用が推進されている．しかし一般に，既存のアプリケーションのアクセラレータ向けの移植・最適化には多大なコストが伴うことが知られている．本稿では，実際に用いられている圧縮性流体アプリケーション UPACS へ OpenACC を適用・最適化することでその移植コストを調査し，OpenMP による移植との性能比較評価を行った．その結果，PGI コンパイラを用いた場合においては，基準となる変更なしの UPACS から 9.5 倍，OpenMP により並列化し 6 CPU コアで実行した場合と比較して 15%の性能向上を得た．またさらなる高速化に向けて，ボトルネック部分の最適化の検討，CUDA Fortran の適用に向けた予備評価を行った結果を報告する．

1. はじめに

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[5] や，メニーコアアクセラレータ向けの OpenACC[4] 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 の OpenACC による高速化を通じ，現状における OpenACC の仕様上の制限や，OpenMP 版と比較した実装コストや性能の評価を行い，PGI コンパイラを用いた場合には，基準となる変更なしの UPACS から 9.5 倍，OpenMP により 6 コアで実行した場合と比較して 15%の性能向上を得られることを確認した．また OpenACC 化を行うにあたり，陰解法部分が性能ボトルネックとなっていることを確認し，アルゴリズムの変更による高速化の検討や，CUDA Fortran の適用などの予備評価を行った．これらの結果を報告する．

2. 背景

2.1 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[6] 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

[1] 東京大学
University Tokyo
[2] 東京工業大学
Tokyo Institute of Technology

独自の拡張を施したものである．

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 IHI 版の UPACS も共通であり，大きな枠組みは変わっていない．IHI 版の UPACS 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．以前の我々の研究 [2] において，オリジナルの UPACS の Navier-Stokes 方程式を解く解法に対して OpenACC 化を行っているが，この Navier-Stokes 方程式と独立の計算フェーズとして乱流項を計算するよう実装されている．また，以下で断りなく UPACS と述べた場合，IHI 拡張版の UPACS を指すこととする．

2.2 OpenACC

OpenACC は複数のコンパイラベンダにより規定された，メニーコアアクセラレータ向けの並列プログラミング規格である．C/C++や Fortran と言った科学技術アプリケーションで多く用いられるプログラミング言語に対して，OpenMP の様にコンパイラ指示文を挿入することで，アクセラレータ環境での実行を可能にする．アクセラレータ向けのプログラミングモデルとしては，CUDA や OpenCL が広く使われているが，これらはアクセラレータのアーキテクチャを意識した低レベルな記述をする必要があり，アプリケーション移植の阻害要因になっていた．一方ソースコードの直接的な変更の必要がない OpenACC の登場により，アクセラレータ環境への移植の簡素化に期待が高まっている．

例えば**図 1** は OpenACC による行列積の例である．OpenACC を構成する主要な指示文として，並列領域指定指示文，データ移動指示文，ループ指示文の 3 つがある．並列領域指定指示文はアクセラレータで実行するべき領域を指定するためのものであり，`parallel` ディレクティブ，`kernels` ディレクティブがこれにあたる．図 1 の例では，2 行目の`!$acc kernels` から 15 行目の`!$acc end kernels` でループネストを囲むことにより，並列実行領域を指定している．

```
1  !$acc data copy(c) copyin(a,b)
2  !$acc kernels
3  !$acc loop gang
4  do j=1,n
5     !$acc loop vector
6     do i=1,n
7        cc=0
8        !$acc loop seq
9        do k= 1, n
10          cc= cc + a(i,k) * b(k,j)
11       end do
12       c(i,j) =cc
13    end do
14 end do
15 !$acc end kernels
16 !$acc end data
```

図 1 OpenACC による行列積

データ移動指示文はホスト-アクセラレータ間のデータ移動に用いられる．現在のアクセラレータはホスト CPU 側とは独立したメモリを持っているのが一般的であり，それに対応するためのものである．図 1 の例では，1 行目の`!$acc data` において，入力行列 a, b, c のホスト側からアクセラレータ側へ転送が行われる．ここで `copy(c)` とは，16 行目の`!$acc end data` において，アクセラレータ側からホスト側へのデータ転送を行うことを指示しており，一方 `copyin(a,b)` と指定した場合，アクセラレータ側からのデータ転送は行わない．データ移動指示文には他に `enter/exit data` ディレクティブ，`update` ディレクティブなどがあり，プログラムの構造により使い分ける必要がある．ループ指示文は並列化方法や並列粒度の指定を行うためのものである．OpenACC には 3 つの並列粒度，`gang`，`worker`，`vector` があり，`worker` は `vector` の集合，`gang` は `worker` の集合である．図 1 には 3, 5, 8 行目に `loop` ディレクティブが現れているが，それぞれ 4, 5, 9 行目のループ文の並列粒度を指示している．この例では，最外の `j` ループが最も粗粒度の並列化が行われ，`i` ループが細粒度に並列化される．またこの例では `worker` を指定していないため，`gang` は `vector` の集合である．`seq` の指定された 9 行目のループは並列化が行われず，`vector` 単位で逐次に実行されることとなる．

2.2.1 OpenACC vs OpenMP

同じディレクティブベースのプログラミングモデルである OpenACC と OpenMP の大きな違いのひとつは，採用しているメモリモデルにある．OpenMP は共有メモリモデ

ルであり，各スレッドは同じデータにアクセス可能であることが前提として作られている．一方で OpenACC は独立メモリモデルであり，ホスト側のメモリとデバイス側のメモリは独立しているものとして扱われる．OpenACC コンパイラの実装によっては，データへのアクセス領域を解析し，暗黙的にホスト-デバイス間のデータ転送を行うことでデータの一貫性を保つ場合もあるが，基本的に OpenACC では，プログラマがメモリ空間が独立していることを意識し，明示的にデータ転送を行うことで，データの一貫性を保つ必要がある．

OpenACC が独立のメモリモデルを採用しているのは，一般に，ホスト側のメモリ容量と比べてアクセラレータのメモリ容量が少ないこと，ホスト-デバイス間のデータ転送速度が相対的に遅いこと，を考慮した結果であるが，実用上弊害もある．ひとつは，ディープコピーの問題である．C 言語における構造体中にポインタが含まれる場合や，Fortran の derived type 中に allocatable, pointer 属性のついたメンバがある場合，構造体中のポインタの持つ情報は，実際のデータが格納されたアドレス情報である．従って，ホスト側で確保されたポインタを含む構造体をデバイス側にコピーする場合，その構造体中のポインタが持つ情報とは，ホストメモリ上に確保されたデータのアドレス情報であり，デバイス側では利用できない．このポインタの参照先までコピーする方式をディープコピーと呼ぶが，メモリ容量の差などの問題から，現在の OpenACC の仕様ではディープコピーをサポートしていない．OpenACC でポインタを含む構造体を利用する際には，ポインタの参照先を個別に転送する必要がある．

2.2.2 OpenACC vs CUDA・OpenCL

CUDA や OpenCL は C++/Fortran や C 言語の拡張であり，アクセラレータ向けのプログラミングモデルとしては現在最も広く使われている．並列化，データ移動，ループマッピングなど全てを明示的に行う必要があるが，自由度が高くきめ細かな最適化を行えるため，アクセラレータの性能を十分に引き出すことができる．

既存のアプリケーションのアクセラレータへの移植という観点で OpenACC と CUDA・OpenCL の生産性を比較した場合，OpenACC では基本的に指示文を挿入するだけである一方，CUDA・OpenCL ではオリジナルのプログラムを書き換える必要があるため，OpenACC の方が生産性が高いと言える．プログラムの移植性の観点から考えると，CUDA がサポートしているアクセラレータが NVIDIA GPU のみであるのに対し，OpenCL と OpenACC は他のアクセラレータにも対応しているため，移植性が高いと言える．その一方で性能に関しては，OpenACC は CUDA・OpenCL と比較して記述の自由度が低いため，アクセラレータの性能を十分に引き出せない恐れがある．例えば OpenACC は CUDA の `syncthreads()` にあたる明示的な同期構文を有していない．様々なアクセラレータに適応できるようこのような仕様となっているが，このような機能的な制限が与える影響を調査する必要がある．

3. 関連研究

UPACS の高速化に関する研究としては，高木らによる研究 [7] がある．高木らによる研究では，主に CPU 版をターゲットとし，ループインターチェンジや配列の次元変更などの最適化による高速化を行っている．一方本研究で対象としているのは GPU などのアクセラレータである．UPACS の GPU による高速化としては，我々の以前の研究 [2] において，OpenACC1.0 の仕様のもとに，今回対象とするアプリケーションである UPACS の CUDA・OpenACC 化を行っている．しかし Navier-Stokes の解法部分のみに移植対象を限定しており，今回のメインの対象とした乱流解析部についての高速化はまだ検討されていない．また，PGI コンパイラが x86 のマルチコア CPU をターゲットとできるようになっており，充分に検証がされていない CPU 版の最適化，比較評価を本研究では行っている．

アプリケーションの OpenACC 化に関しては，Calore らの研究 [1] が比較的新しい．Calore らは Lattice Boltzmann の MPI 環境での並列化を行っている．Lattice Boltzmann では袖領域を他のプロセスに通信する必要があるが，OpenACC の async 指示節を適切に用いることで通信時間を隠蔽し，クラスタ上でのスケーラビリティを得ている．しかし彼らのアプリケーションはベンチマーク用に作られたもので，実アプリケーションとは呼べず，また OpenACC 自体の評価は行っていない．本研究では実アプリケーションの高速化に基づく OpenACC の評価を行うことを目的としている．

実アプリケーションの OpenACC による移植としては，Kraus らの研究 [3] がある．彼らの研究においては，C++ の CFD アプリケーションを CUDA，OpenACC を用いて比較評価を行っている．彼らのアプリケーションは陽解法を用いており，良好な結果が出ているが，本研究で対象としている UPACS では，陰解法部分をいかに高速に解くかが鍵となっている．

4. 研究目的・評価手法

4.1 目的

本研究における目的は主に 2 つあり，ひとつは UPACS 自体のアクセラレータ利用における高速化手法の検討，もうひとつは現状の OpenACC の評価である．UPACS の高速化という観点においては，以前の研究において対象としていなかった乱流解析部分について，どの程度高速化が見込めるか，評価することが目的である．OpenACC の評価という観点では，以前は OpenACC1.0 の仕様の元で実装

を行ったが，2.0 の仕様における productivity の評価，機能制限が与える性能への影響などが目的である．

結果として，IHI 版 UPACS を PGI コンパイラでコンパイルし，CPU1 コアでの実行と比較した場合と比較し，9.5 倍の高速化を得た．また OpenMP による実装も行い，1CPU ソケットを使った場合と比較して 15%の性能向上を得た．しかし OpenACC 版においては陰解法部分が性能ボトルネックとなることを確認し，高速化に向けた予備評価を行った．

4.2 評価手法

OpenACC の性能・移植コストの評価にあたり，OpenMP 版 UPACS の実装，一部の OpenACC カーネルの CUDA Fortran への置き換えを行い，比較評価を行う．評価を行う環境として TSUBAME2.5 を用いる．**表 1**，**表 2** にそれぞれ今回用いた計算環境，コンパイラとそのオプションを示す．

また本稿における評価には株式会社 IHI より提供された実際のデータを用いる．実験対象データは単翼周りの流れを解析するための 3 次元データであり，格子点数は約 400 万点である．この 400 万点の格子を 7 つのブロックに分割し，翼列まわりへの適合を行っている．7 つのブロックのうち最も大きいものが $83 \times 96 \times 120$ であり，最も小さいものが $110 \times 26 \times 120$ である．なお UPACS ではこれらのブロックは最大 7MPI 並列で計算することができるが，今回は MPI による評価は行わず，1 プロセスが全てのブロックの計算を行う．境界条件については，気流の翼列に対する流入口，流出口，ピッチ方向，壁で異なり，それぞれ亜音速流入境界条件，流出境界での静圧固定，周期境界条件，固定壁の滑り無し条件となっている．時間積分法には陰解法を用いるが，定常計算であるために陰解法の 1 タイムステップあたりの反復回数は 1 である．

OpenACC の適用する実装範囲としては，上述の入力によって使われる範囲のみに限っておこなう．今回の入力で用いられない範囲についてもいずれ実装するべきであるが，使われていない解法部分は基本的に計算手法が異なるだけで，OpenACC の適用手法としては大きく変わらない．OpenACC の評価として用いるには今回の入力で用いられる範囲のみで十分であると判断したためである．また前述の通り，我々は以前の研究 [2] において，オリジナルの UPACS に対して OpenACC の適用を行っている．しかし，当時から UPACS 自体が更新されていること，今回対象とする入力では Navier-Stokes を解く部分においても以前は対象としなかった計算カーネルを利用していること，OpenACC の仕様が変化していることから，IHI 版の UPACS をベースにして一からの再実装を行う．

表 1 実験環境

CPU	Intel Xeon X5670 6cores 2.93 GHz 2 sockets 54 GB Memory
GPU	NVIDIA Kepler K20X 2688 CUDA cores 6GB Memory

5. UPACS の OpenACC 実装

OpenACC を用いた実装の詳細を説明する．本章で説明する内容はエンジニアリング的な要素が強く，必ずしも本稿で説明が必要な内容ではないが，OpenACC を実アプリケーションに用いた例はそれほど多くなく，得られる資料も限定的であるため，OpenACC を用いた実装を行う際の一般的な方法や留意点について書く．

5.1 一般的な OpenACC 適用方法

OpenACC を用いた実装を行う際，その実装ステップは多くの場合以下のようになると考えられる．

(1) プロファイリングによる，アプリケーションのボトルネック部位の導出
(2) ボトルネック部位への parallel 指示文，kernels 指示文の適用による並列化
(3) data, enter/exit data, update 指示文等による，データ転送の最適化
(4) loop 指示文による並列粒度の最適化, cache 指示文等による計算部分の最適化
(5) (1)〜(4) の繰り返し

以下では，上述のステップによる UPACS の OpenACC 化を説明する．ただし今回は (4) にあたる最適化を行っていないため，(1)〜(3) に関しての説明を行う．

5.2 UPACS のプロファイリング

まず，CPU 実行時におけるボトルネック部分を特定するために，アプリケーションのプロファイリングを行う必要がある．TSUBAME2.5 表 1 の 1CPU コアにおけるプロファイリング結果を**表 3** に示す．コンパイラとして pgfortran を用い，表 2 のオプションに-Mprof=func を加えコンパイルした．-Mprof=func を加えることにより，実行時に function/subroutine レベルでの実行時間・呼出回数などのプロファイリング情報が生成され．pgprof プロファイラを用いることで可視化できる．テストデータとしては前述の株式会社 IHI による提供データを使用した．以降の評価実験についてもこのテストデータを利用する．ただし，本来のシミュレーションでは数万タイムステップの実行を行う必要があるが，プロファイリング時は 100 タイムステップに制限している．

表 3 から，UPACS にはボトルネックとなる subroutine/-function が偏在していることがわかる．表 3 に基づき，主

表 2 コンパイラ

	compiler	target	option
Intel	ifort15.0.2	CPU	-O3 -openmp -no-prec-div -xHost -no-fp-port -convert big_endian
PGI	pgfortran15.10	CPU(single)	-fast -mp -byteswapio
		CPU(OpenMP)	-fast -mp -byteswapio
		CPU(OpenACC)	-fast -acc -ta=multicore -byteswapio -Minfo=accel
		GPU(OpenACC)	-fast -acc -ta=tesla -byteswapio -Minfo=accel
		GPU(CUDA)	-Mcuda=7.0 -byteswapio

表 3 UPACS プロファイリング結果 (上位 20 function)

	subroutine	time[sec]	%
1	cellfacevariables_woedgecorner	1395.7	19
2	implhs_mfgs	1054.2	14
3	flux_roe	980.5	13
4	muscl_co	502.6	7
5	muscl_2ndorder	473.8	6
6	cellfacevariables_woedgecorner_tm	412.7	6
7	diffusion_tm_sa	291.3	4
8	rhs_convect	245.7	3
9	flux_ausm_tm	175.6	2
10	rhs_viscous	173.4	2
11	strain_rate	151.2	2
12	calcvoverdx	150.9	2
13	vorticity	149.0	2
14	minmod_co	137.5	2
15	flux_vis	123.0	2
16	calcdt_original	104.2	1
17	production_destruction_sa	72.5	1
18	lhs_gaussseidel	65.5	1
19	muscl	63.7	1
20	transferrecvp4	59.9	1
	total	7433.4	100

表 4 UPACS の主要計算フェーズ．(subroutine/function の実行時間合計と各計算単位の総実行時間が一致しないのは，表 3 で省略した 21 番目以降の影響．)

計算単位	主要な subroutine/function (数字は表 3 参照)	time[sec]	%
乱流	5, 6, 7, 9, 11, 13, 17, 18	1951.0	26.3
対流項	3, 4, 8, 14, 19	1984.5	26.7
粘性項	1, 10, 15	1692.1	22.8
時間発展	2	1115.1	15.0
total		6742.7	90.7

要な計算単位毎に subroutine/function の実行時間をまとめたものが**表 4** である．乱流，対流項，粘性項，時間発展の 4 つの計算で実行時間の 90%以上を占めることがわかる．本稿では主にこの 4 つの計算部分について説明するが，最終的にはタイムステップを刻むための時間ループ内のほとんどを OpenACC 化している．

一般的に UPACS のような流体系のアプリケーションにおいては，時間ループの内部を極力全てデバイス内で実行しなければ高速化は難しい．流体系のアプリケーションにおいては，支配方程式の偏微分方程式を解析的に解くために，解きたい領域の時間・空間を離散 (格子) 化するが，基本的に各格子の計算は連続する周囲の格子のみとの相互作用計算によって行われる．これはデータ量に対する計算量が比較的少なくメモリ律速な計算パターンとなるが，対する計算機のメモリ性能は，デバイスメモリ内 > ホストメモリ内 > ホスト-デバイス間通信である．デバイス/ホスト側で更新したデータをホスト/デバイス側で利用する場合，データの一貫性を保つためにはホスト-デバイス間の通信が必須であるが，時間ループ内にホスト側の計算が残っていると，結局より低速なホスト-デバイス間通信に律速されてしまうため，時間ループ内部を極力全てデバイス内で実行する必要がある．UPACS の OpenACC 実装においても，極力ホスト-デバイス間の通信回数を少なくするために，時間発展ループ内のほとんど全てを OpenACC 化することを目指す．

5.3 ボトルネック部位の並列化

プロファイリングにより特定されたボトルネック部分に指示文を適用することにより，アクセラレータ上での実行を可能にする．しかし UPACS は多数の解法を提供しており，コードサイズが非常に大きいため，今回の入力で用いられる部分のみを適用範囲とする．また，元プログラムに直接指示文を適用できないケースがある．以下では，UPACS のボトルネック部分の OpenACC 化について説明する．

5.3.1 元プログラムの変更

OpenACC 化を行う前提として，プログラムが並列化可能であるかどうかを調べる必要がある．UPACS は計算領域をブロック単位で分割し，ブロック毎に計算プロセスを割り当てることで MPI 並列を行っている．しかしブロック内部の計算は明示的には並列化されておらず，富士通コンパイラの自動並列化に任せていた形跡がみられる．調査の結果，主要な subroutine/function においては，表 3 中の 2, 18 番目にあたる implhs_mfgs と lhs_gaussseidel にデータ依存性があり並列化不可であったが，UPACS では同等の計算を行い並列化可能にした implhs_mfgs_vector と lhs_gaussseidel_vector を提供しているため，それぞれを置き換えることで並列化を行うこととした．なお置き換えによる性能への影響は後述する．

UPACS の各サブルーチンに対しての OpenACC ディレクティブの適用についてであるが，OpenACC の仕様上，またはコンパイラの実装上の理由で，元プログラムを変更せざるを得ない箇所がいくつかあった．**表 5** にソースコードを変更せざるを得なかった理由と，それによって書き換えが生じた行数について記す．影響が最も大きかった構造体中のポインタメンバであるが，前述の通り，現在の OpenACC の仕様ではディープコピーをサポートしていない．しかし仕様上サポートされていないのはディープコピーであり，適切なプロセスでコピーを行えばポインタを含む構造体も利用できるはずである．ここでいう適切なプロセスとは以下である．

(1) あらかじめ構造体自体をデバイス上にコピーする．このときディープコピーは行われないが，構造体中のポインタはデバイス上に生成される．
(2) 使用する構造体中のポインタを指定し，ポインタが指すデータのコピーを行う．これにより，先にデバイス上に生成された構造体中のポインタが転送されたデータを適切に指すようになる．

このプロセスにより，実際にデバイス上にデータが転送されていることは確認できた．しかし，並列領域内でこのポインタを用いて計算しようとした場合，Illegal address エラーにより計算が実行できない．この原因は現在調査中である．ゆえに行数増加のほとんどは，このバグを回避するために元プログラムの書き換えを行なった結果である．

二つ目は，並列計算領域内からの関数呼び出しの問題である．OpenACC は仕様 1.0 においては関数呼び出しをサポートしていなかったが，2.0 より `routine` ディレクティブが追加され，対応できるようになった．しかし 2.0 の仕様においては，Fortran の pure function は利用できないとの記述がある．OpenACC2.5 の仕様においては pure function の制限についての記述が消えているため，今後改善されると予想される．現在我々が利用できる最新のコンパイラにおいても未だ OpenACC2.5 の仕様はサポートされていないため，今回は元プログラム関数呼び出し部分を直接展開することで解決した．

最後の理由は pointer 属性のついた構造体の配列にである．ここでの構造体は，allocatable/pointer 属性のついたメンバを含まない，OpenACC でサポートされている構造体である．この構造体の配列を用いるとき，構造体に pointer 属性がついていると，Illegal address エラーが起こってしまうため，allocatable 属性に変更している．この原因も調査中である．

以上より，OpenACC の仕様上の理由に書き換えざるを得ない部分はそれほど多くなく，現状において問題となっている部分も今後改善されていくものと思われる．

5.3.2 OpenACC ディレクティブの適用

上述の変更を施したプログラムに対し，OpenACC のディレクティブを適用する．基本的には，以下の手順でディレクティブを適用する．

(1) 対象となるループネストに `kernels` ディレクティブを適用する．
(2) `pcopy`，`pcopyin`，`pcopyout` を `kernels` の clause として用いて，データの転送を行う．これらの `p...` は `present_or_...` を意味し，データが既にデバイス上に存在しなければ/存在すれば，コピーを行う/行わないことを意味する．
(3) `loop` ディレクティブを用いて並列粒度の指定を行う．基本的には，最外ループを `gang` レベル並列，最内ループを `vector` レベル並列する．
(4) `loop` ディレクティブの `private` clause を用いて，各スレッドでローカルに扱うべき配列変数の指定を行う．これを行わない場合，PGI コンパイラは配列変数をデバイスメモリ上に確保し，全てのスレッドがアクセスを行うことになるため，計算結果に誤りが生じる．
(5) コンパイルオプション (表 2)-Minfo=accel の出力を確認し，並列化されていることを確認する．並列化可能であるはずのループをコンパイラが並列性を抽出できず，並列化されない場合があるが，この場合は対象の `loop` ディレクティブに `independent` clause を指定する．

この際，オリジナルの実行と同じ結果を得られるか，常に結果を確認しながら実装を進めることが重要である．この段階ではアプリケーションの速度は気にせず，ホスト側とのデータの一貫性を保守的に保ちながら実装を進める．このとき役に立つのが，`if` clause である．例えば `!$acc kernels if(.false.)` のように記述すれば，その部分はホスト側で実行されることになる．

以上を計 49 のループネストに適用した．

表 5 ソースコードの書き換え理由と書き換え行数

書換理由	原因	書換行数 [行]
構造体内部のポインタ	エラー回避 (調査中)	709
Fortran pure function	OpenACC2.0 の仕様	39
Pointer 属性のついた構造体の配列	エラー回避 (調査中)	41

5.4 データ転送の最適化

メモリ律速なアプリケーションに OpenACC を適用する場合，最も重要なプロセスがデータ転送の最適化である．前述の通り，ホスト-デバイス間のデータ転送速度が相対的に低速なためである．このプロセスにおいても，結果が一致するかどうかを常に確認しながら行う必要がある．データ転送がどこで発生しているのか調査するためには，PGI コンパイラを用いる場合なら，環境変数 PGI_ACC_TIME を 1 にするのが簡単な方法である．

データ転送の最適化は，末端のループネストから，徐々に上流のサブルーチンで転送を行うようにし，最終的に時間ループの外に追いやる．UPACS の場合では，まずは表 4 で示した計算単位ごとでデータ転送をまとめ，その後に時間ループの外側で転送を行うようにした．ブロック間のデータ通信など，どうしてもホスト側で扱う必要があるデータに関しては，update ディレクティブを用いることで転送を行っている．

6. OpenACC 版 UPACS の評価

6.1 OpenMP vs OpenACC

OpenACC 版 UPACS の評価を行うために，比較対象として OpenMP 版を実装した．OpenMP のディレクティブを指定したのは，OpenACC と同じ 49 ループネストである．実装は!$omp parallel do の指定のみであり，スレッドスケジューリングの指定などは行っていない．また OpenMP は OpenACC 適用前のコード変更処理を行ったプログラムに適用している．

OpenMP 版との比較評価の結果を**図 2** に示す．図 2 は，変更を加える前の IHI 版 UPACS を PGI コンパイラによりコンパイルし CPU1 コアを用いて実行した時の性能 (グラフの左端) を 1 とした相対性能である．計測時間には初期化処理などを含んでおらず，1 タイムステップあたりの実行時間を計測しており，このときの実行時間が 70.4 秒である．また計算環境である TSUBAME の 1 ノードには CPU ソケット ×2，メモリ ×2，GPU×3 があり，GPU はそれぞれ独立したメモリを持っており GPU 間での共有は行われないが，CPU 側ではメモリは共有されている．本評価では 1 対の CPU ソケット・メモリで評価を行うために，numactl により物理的に近い側のメモリにのみアクセスするよう指定している．これは，1GPU に対する比較対象としては 1CPU ソケット・メモリが妥当であるとの考えによる．評価軸については**表 6** に記載する．表 6 の V0 は前述した変更を一切行っていないものであり，一部並列化不可の部分があるため，1CPU コアのみの評価である．V1 は前述の並列版への差し替えやバグ回避などの変更を行ったものである．V2 は，対流項・粘性項，また乱流項計算の一部において用いられている構造体を，CPU・GPU それぞれに適した形に変更したものである．書き換え行数には，プログラム自体の書き換えと，挿入した OpenACC ディレクティブを含む．また表 2 中の Intel や PGI は使用したコンパイラを指し，OMP, ACC はそれぞれ OpenMP 実装版，OpenACC 実装版を示す．

まず，Intel コンパイラと PGI コンパイラの比較であるが，同じ 1CPU コアでの実行であっても，Intel コンパイラによりコンパイルされたものが 4.0 倍程度高速であることがわかった．この原因については調査中であるが，SIMD 化などの差によるものと見ている．実験環境の CPU の SIMD 長は 128bit であり，IHI 版の UPACS は単精度であるため，理論上 4 倍の高速化が見込める．本来メモリバンド幅律速なアプリケーションに対して SIMD 命令による高速化は効きにくいが，CPU1 コアではバンド幅を使い切ることができないために，演算律速になっていると考えられる．しかし 4 倍というのは理論値であり，PGI コンパイラを用いた場合でも-fast オプションを付加することにより SIMD 命令の発行が試みられるはずであるため，この他にも原因があると考えられるが，現在調査中である．また，V0 から V1 への書き換えにより，PGI コンパイラ版では 2.2 倍程度高速化されている一方，Intel コンパイラ版では 27%程性能が低下している．PGI コンパイラ版では前述のバグ回避が結果的に最適化を助け，Intel コンパイラ版では並列可能カーネルへの差替えが悪影響を起こしたと考えられる．前述のバグ回避では，OpenACC カーネルのコンパイル時にポインタを含む構造体部分の解析を失敗する現象を防ぐために，ボトルネックのループネスト内で構造体中のポインタを使わないように書き換えている．この結果カーネルが単純になり，PGI コンパイラによる最適化が効きやすくなったため，並列可能カーネルへの差替えによる悪影響を上回り，高速化したと考えられる．Intel コンパイラ版での性能低下は自然なことである．詳細は後述するが，差替え前の陰解法カーネルでのメモリアクセスが連続的であるのに対し，差替えを後のカーネルは不連続なメモリアクセスを行うためである．バグ回避による書き換えの影響と，並列可能版への差替えは本来別々に調査すべきであるが，本稿においてはあまり本質的な問題ではないため，今後の課題とする．またメモリアクセス規則の改善のために行った構造体の変更 (V2) により，どちらも性能改善が見られる．

次にマルチコア CPU 上での比較であるが，PGI コンパイラ，Intel コンパイラそれぞれによる OpenMP 版と OpenACC のマルチコア CPU ターゲット版についての比較を行う．pgfortran15.10 より，NVIDIA 社製 GPU，AMD 社製 GPU に加え，Intel 社の x86 CPU も OpenACC のターゲットにできるようになった．ターゲットの切り替えは-ta=multicore により行っている．ターゲット multicore が指定された場合，ホスト-デバイス間のデータ転送は不要であるため，データ転送指示文は無視される．PGI コンパイラではマルチコア CPU 向けの並列化は gang レベルにおいてのみ行われ，vector レベルの並列化は無視されるようである．ただし，OpenMP における OMP_NUM_THREADS にあたるスレッド数を外部から指定するためのインターフェースは見つかっておらず，実装されているかどうかは調査不足でわからなかった．そのため 1CPU ソケットのみを対象にした評価ができず，2 ソケット 12 コアでの評価のみとなっている．OpenACC のマルチコア版との比較のために，PGI 版 OpenMP は 12 コアでの評価も行った．まず

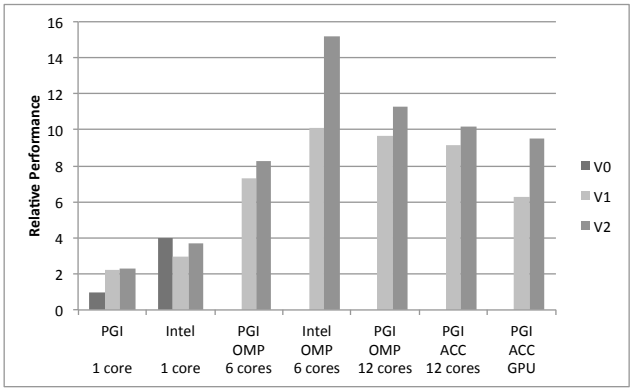

図 2 UPACS OpenMP 版と OpenACC 版の比較

表 6 UPACS への変更と書換行数

	書換内容	書換行数 [行]
V0	IHI 版 UPACS のオリジナル	0
V1	並列版への差替え・バグ回避版	3213
V2	構造体変更最適化版	512

OpenMP 版について V2 での性能を比較すると，1CPU ソケット (6 CPU コア) で実行した場合，1 コアでの実行時と比較して，PGI 版，Intel 版それぞれ 3.6 倍，4.1 倍程度の高速化が得られた．特に Intel 版については基準から 15.2 倍高速化しており，GPU 実行の OpenACC 版より 1.6 倍程度高速であった．次に OpenACC のマルチコア CPU 版との比較であるが，同じ PGI コンパイラによる OpenMP 実装と比較して，90%程度の性能であった．OpenACC マルチコア版ではデータ転送などのオーバーヘッドはないはずであり，OpenMP と全く同じループを並列化しているため，差がどこで生じたのかは今後調査する必要があるものの，今回の場合は OpenACC の機能制限が障壁となることもなかったため，基本的にはコンパイラの実装による差であると考えられる．

最後に GPU ターゲット版の OpenACC についてであるが，CPU の結果と比較すると，V1 から V2 への性能向上が特に大きく，1.51 倍程度の高速化が得られた．また基準となる PGI 1core 版に対して，9.53 倍の性能向上が得られ，PGI OMP 6core 版に対しても 1.15 倍程度の性能向上が得られたが，Intel OMP 6core 版に対しては 63%程度の性能しか得られなかった．基本的な性能が GPU ＞ CPU であることを考慮すると満足な結果であるとは言えず，さらなる改善が望まれる．

6.2 OpenACC 版 UPACS の詳細解析

OpenACC 版 UPACS で充分な性能が得られなかった原因についての詳細解析を行う．**図 3** は，Nvidia Visual Profiler を用いて，GPU 上での実行のタイムラインをとったものである．図 3 は 1 タイムステップあたりの実行を抜き出したものであり，図の緑のバーの左端から図中に赤字で記した C の右端までが 1 タイムステップである．Compute(図中緑色の部分) が GPU による実行が行われている部分であり，Memcpy(図中黄土色の部分) がホスト-デバイス間のデータ通信が行われている部分である．すなわち，C の部分では GPU による計算が行われていない．また，図中 A と B が低速であることがわかるが，これは並列化不可であり差し替えたカーネル，lhs_gaussseidel_vector と implhs_mfgs_vector であり．最も実行時間のかかっている lhs_gauussseidel は，プロファイリング結果 (表 3) ではもともと 18 番目であったカーネルである．

また，**図 4** は 1 タイムステップあたりの各計算単位における実行時間の比較である．このグラフ中の”その他”にあたる部分には OpenACC 指示文も OpenMP 指示文も使われておらず，どちらも CPU1 コアでの実行である．調査したところ，PGI コンパイラ版では，袖領域通信部分が Intel 版と比べて 10 倍近く低速であった．なお今回の実験では MPI 並列を行っていないため，全てノード内のコピーで完結する．UPACS では袖領域通信の準備段階として，データが連続に並ぶよう前処理を行うが，この部分が非常に低速なようである．図 4 の結果から，陽解法である対流項・粘性項の計算部分では，OpenACC が 1.75 倍程度高速であることがわかったが，陰解法である時間発展部分の計算は OpenMP と同程度，lhs_gauussseidel を含む乱流解析部分では 55%程度の性能であった．

7. さらなる高速化に向けての予備評価

前述の実験結果より，OpenACC 版では陰解法部分がボトルネックとなっていることがわかる．この高速化のために CUDA Fortran によるカーネル実装を行い，予備評価を行った．本予備評価に関しては，前述の提供データによるものではなく，テスト用に生成した仮のデータを用いて行っている．なお計算精度は提供データと同じく単精度で，提供データ中最大の $83 \times 96 \times 120$ の 1 ブロックを用いている．本来は IHI 版 UPACS に実装して評価を行う予定であったが，問題が発覚したため予備評価とした．CUDA Fortran を用いる場合，リンク時のコンパイラオプションに-Mcuda を指定するが，このオプションを指定した場合，CUDA プログラムのリンクの有無にかかわらず，一部の OpenACC カーネルにおいて実行時にエラーが生じることがわかったためである．この原因は現在調査中である．なお 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，今回の予備評価において CUDA カーネルが OpenACC 側から呼び出せることを確認している．

陰解法部分の高速化に向けて，陰解法について簡単に説明する．まず陽解法とは，**図 7** の上の数式の形で表される．左辺 Q_{t+1} を更新するために，右辺の計算を行うが，右辺には Q_t しか現れない．故に同タイムステップのにお

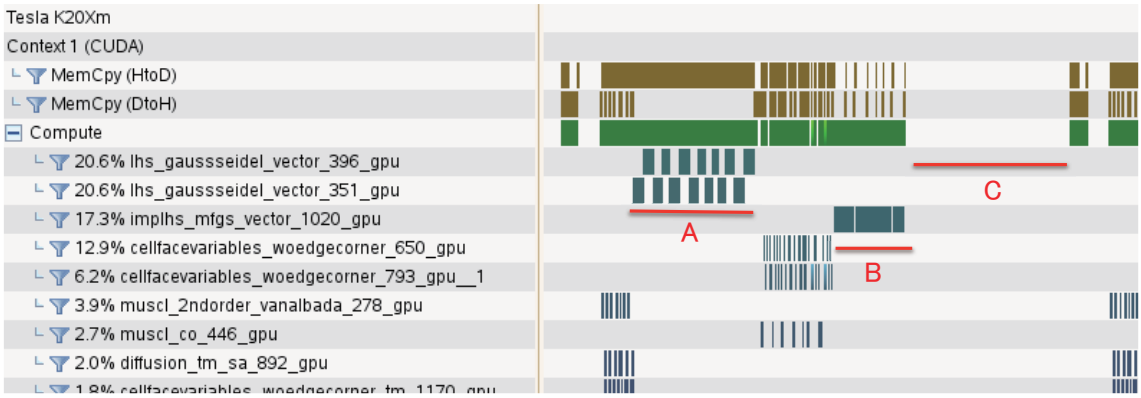

図 3 Nvidia Visual Profiler によるプロファイリング

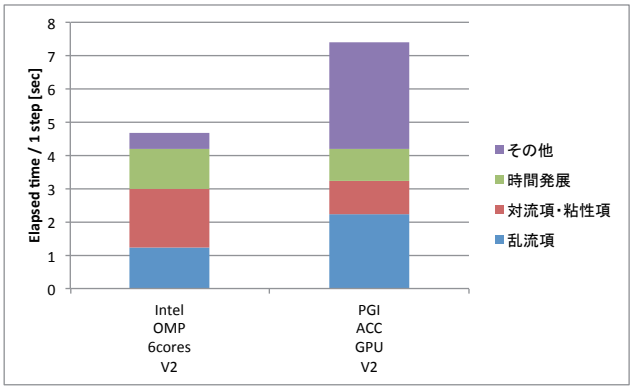

図 4 Intel 版 OpenMP 6 並列と GPU OpenACC の比較

$$
\begin{aligned}
Q_{t+1}(x,y,z) =& a_1 Q_t(x,y,z) + a_2 Q_t(x-1,y,z) \\
&+ a_3 Q_t(x,y-1,z) + a_4 Q_t(x,y,z-1) \\
&+ a_5 Q_t(x+1,y,z) + a_6 Q_t(x,y+1,z) \\
&+ a_7 Q_t(x,y,z+1)
\end{aligned}
$$

$$
\begin{aligned}
Q_{t+1}(x,y,z) =& a_1 Q_{t+1}(x,y,z) + a_2 Q_{t+1}(x-1,y,z) \\
&+ a_3 Q_{t+1}(x,y-1,z) + a_4 Q_t(x,y,z-1) \\
&+ a_5 Q_t(x+1,y,z) + a_6 Q_t(x,y+1,z) \\
&+ a_7 Q_t(x,y,z+1)
\end{aligned}
$$

図 5 7 点ステンシル陽解法 (上)，陰解法 (下)

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 7 の下の数式の形で表される陰解法は，右辺に Q_{t+1} が現れる．故に，$Q_{t+1}(x,y,z)$ を計算するためには $Q_{t+1}(x-1,y,z), Q_{t+1}(x,y-1,z), Q_{t+1}(x,y,z-1)$ 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早いかは明らかではない．

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 6** の右のような順

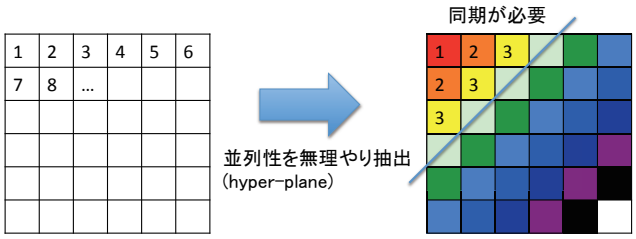

図 6 hyperPlane 法による並列化 (2D)．自格子の更新に更新後の上格子と左格子の値を使うため，本来は左図の数字の順で計算するが，右図のような順番で並列計算が可能である．

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．それ故に前述の Intel 版 (図 2) での性能低下が起きたと考えられる．さらに GPU は CPU と比べて非連続なアクセスを苦手としているため，GPU では低速であったと考えられる．

そこで今回我々は，この Hyper Plane 法の高速化手法として，ブロッキング (**図 7**) を検討した．ブロッキングの際，例えば小ブロックを $32 \times 1 \times 1$ 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，$Q_{t+1}(x,y,z)$ を更新する際に，依存のある $a_2 Q_{t+1}(x-1,y,z)$ 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．$Q_{t+1}(x,y,z)$ の更新と $a_2 Q_{t+1}(x-1,y,z)$ の読みは逐次で行う必要があるが，UPACS では係数 a_i 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．

この手法を implhs_mfgs_vector に適用した予備評価が**図 8** である．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．

8. おわりに

本稿では，実際に利用されている圧縮性流体アプリケーション UPACS の OpenACC による高速化を検討し，OpenMP 版と比較することにより評価を行った．結果，仕

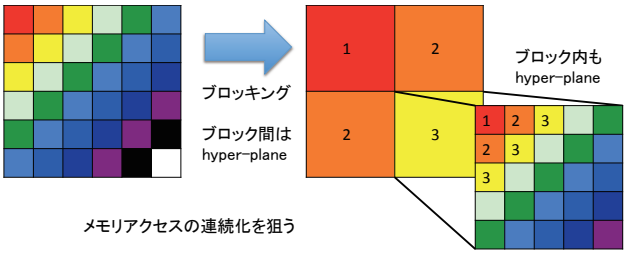

図 7 hyperPlane 法のブロッキング (2D)．ブロック外・ブロック内の 2 段階の Hyper Plane 並列を行う．

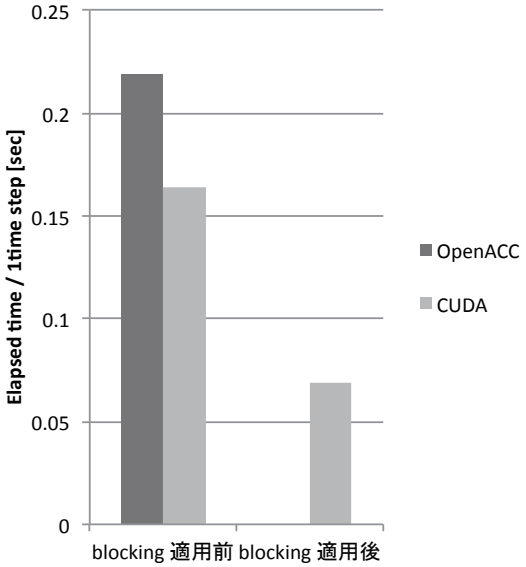

図 8 時間発展部のブロッキング

様上の問題で元プログラム大幅に書き換える必要は少なく，単純な移植を行うだけであれば OpenACC の煩雑さは緩和されていると言える．また性能面では，PGI コンパイラを用いた場合においては，基準となる変更を加えない UPACS から 9.5 倍，OpenMP により 6 CPU コア並列で実行した場合と比較して 1.15 倍の性能向上を得た．しかし Intel コンパイラによる OpenMP 版と比較すると 63%程度の性能しか得られておらず，充分な結果とはいえなかった．そこで得に低速であった陰解法部分に関しての高速化手法を検討し，CUDA Fortran による予備評価を行った．その結果，時間発展部分において 2.39 倍程度の高速化が見込めることがわかったが，この手法はスレッド間の同期を必要とするため，OpenACC では実装できない．

今後の課題として，陰解法部分へのさらなる高速化の検討，Red-black 法などとの比較を行うとともに，OpenACC と CUDA Fortran の比較を充実させることで，UPACS の高速化，OpenACC の評価を行う必要がある．

謝辞 本研究にあたり，アプリケーション及び実験データを提供してくださった，株式会社 IHI の平川香林様をはじめとする皆様に，感謝の意を表する．

参考文献

[1] Calore, E., Kraus, J., Schifano, S. F. and Tripiccione, R.: *Euro-Par 2015: Parallel Processing: 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*, chapter Accelerating Lattice Boltzmann Applications with OpenACC, pp. 613–624 (online), DOI: 10.1007/978-3-662-48096-0_47, Springer Berlin Heidelberg (2015).

[2] Hoshino, T., Maruyama, N., Matsuoka, S. and Takaki, R.: CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application, *Cluster Computing and the Grid, IEEE International Symposium on*, Vol. 0, pp. 136–143 (online), DOI: http://doi.ieeecomputersociety.org/10.1109/CCGrid.2013.12 (2013).

[3] Kraus, J., Schlottke, M., Adinetz, A. and Pleiter, D.: Accelerating a C++ CFD Code with OpenACC, *Proceedings of the First Workshop on Accelerator Programming Using Directives*, WACCPD '14, Piscataway, NJ, USA, IEEE Press, pp. 47–54 (online), DOI: 10.1109/WACCPD.2014.11 (2014).

[4] OpenACC-standard.org: `http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf` (2011).

[5] OpenMP: `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf` (2011).

[6] Takaki, R., Yamamoto, K., Yamane, T., Enomoto, S. and Mukai, J.: The Development of the UPACS CFD Environment, *High Performance Computing* (Veidenbaum, A., Joe, K., Amano, H. and Aiso, H., eds.), Lecture Notes in Computer Science, Vol. 2858, Springer Berlin / Heidelberg, pp. 307–319 (2003).

[7] 高木亮治：三次元圧縮性流体解析プログラム UPACS の性能評価，（オンライン），入手先 ⟨http://www.ssken.gr.jp/MAINSITE/download/wg_report/smpt/2.2_takaki.pdf⟩ (2006).