J. Plasma Fusion Res. Vol.93, No.6 (2017) 271-277

研究・技術ノート

簡約化 MHD シミュレーションコードの自動チューニング

Automatic Tuning of Reduced MHD Simulation Code

宮川大和，TRETLER Rudolf，龍野智哉
MIYAGAWA Yamato, TRETLER Rudolf and TATSUNO Tomoya
電気通信大学

（原稿受付：2016年12月 6 日／原稿受理：2017年 4 月11日）

擬スペクトル法による 2 次元簡約化電磁流体コード rmhdper を，スレッド並列化により自動的にチューニングするシステムを開発した．本システムは並列スレッド数を自動で制御するものであり，ユーザーはグリッド数やマシンを変更するたびに最適スレッド数を選定する作業から解放される．本システムはコード中で変化するメモリアクセスパターンに応じて最適なスレッド数を個別に自動選択するため，コード全体で固定されたスレッド数を用いる場合と比べても実行時間が短縮されることがある．Helios で4096×4096グリッドを用いた場合，自動チューニングシステムは処理の種類によって16, 8 と変化するスレッド数を自動的に選んでシミュレーションを行い，これはどの固定スレッド数を選んだ場合よりも高速であった．

Keywords:
magnetohydrodynamics, simulation, high performance computing, OpenMP, automatic tuning

1．はじめに

自動チューニングは，計算機の性能向上が継続してきた末に生まれた比較的若い研究分野である[1-4]．その目的にはアプリケーションを高速に動かすこと，処理のための消費電力を抑えることなど幾つかの視点があり，そのために配列の構成，アルゴリズムや利用ライブラリの選択，並列度の調整など，様々なレベルでの自動チューニングが考えられる．例えば Active Harmony[1]は独立した監視サーバを立ち上げ，計算機のリソース状況に応じてライブラリ選択やパラメータ調整を動的に行うことで最適な高速化をめざす．また Autopilot[2]など，ネットワークを介して広域に分散したシステムにも対応し，センサーからのデータをファジィ推論に基づく決定プロセスを経て動作させるという壮大な研究もある．ここでは，並列度を自動的に調整することで，数値シミュレーションコードを高速動作させることを考える．

近年の計算機では CPU の動作周波数は頭打ちしており，代わりに複数のコアを搭載することによる性能向上が図られている．大型計算ではそのような複数のCPUコアを搭載した計算機を複数台（ノード）用いることにより，ノード内，ノード間の階層的な並列化が必要となってきた．これらノード内，ノード間の異なる階層における並列処理では，共有メモリ環境で用いられるマルチスレッド並列処理と，分散メモリ環境にも対応できるマルチプロセス並列処理の二種類を効率的に用いる必要がある．近年はプロセスとスレッドの使い分け方も，シミュレーションを行う研究者を悩ませる新たな負担となっている．ここではノード内の並列化に着目し，マルチスレッド並列処理を効率的に行う問題を取り扱う．

マシンに搭載されている CPU コアを全て利用してマルチスレッド並列処理を行った場合，メモリアクセスが律速となる偏微分方程式ソルバーでは，メモリアクセスの衝突が頻発して計算性能が低下することがある．したがって，計算処理の種類に応じてあえてスレッド数を CPU コア数よりも少なくすることで，性能が向上する可能性がある．最適なスレッド数は計算機の特性や処理内容，グリッドサイズなど複数の要因に依存するため，先見的に定めることは困難である．したがって，現実には試行により定めることが多いわけであるが，この試行をプログラミングしておき，自動的に定めることができればユーザーの負担を大幅に軽減することができる．本研究では，このような試行をプログラムにより自動的に行うものを考える．

本研究では，コードの可読性と確実性を重視して設計された，単一ノードで動作する簡約化 MHD コード rmhdper [5]の高速化を考え，計算の種類に応じて最適なスレッド数を自動的に選択する自動チューニングシステム（Automatic Tuning System: ATS）を開発した．このシステムにより，ユーザーは最適なスレッド数を試行により決定するプロセスから解放される．さらに，複数のマルチスレッド並列処理部に異なるスレッド数を用い，最適な並列計算を自動で行うことができるようになった．この手法は，分散メモリと共有メモリを併用したさらに大型のハイブリッド並列計算コードにおいても，共有メモリ並列部分にそのまま用いることができる．

University of Electro-Communications, Chofu, TOKYO 182-8585, Japan

corresponding author's e-mail: tomo.tatsuno@uec.ac.jp

まず次節で本研究で用いる数値コード rmhdper の概要について述べたあと，第 3 節で OpenMP による並列化の実装方法を示す．第 4 節で我々の開発した ATS について述べ，第 5 節でまとめと今後の展望について議論する．

本研究では，電気通信大学の研究室にあるデスクトップ機（ローカルマシンと呼ぶ）と，国際核融合エネルギー研究センターの計算機シミュレーションセンター（IFERC-CSC）にある大型計算機 Helios を利用した[6]．ローカルマシンの CPU はクアッドコアの Intel Core i7 920，Nehalem 2.67 GHz であり，OS は Linux Mint 17.1 Rebecca，コンパイラは Intel Fortran Compiler 14.0.1 を用いた．Helios の CPU は16コアの Intel Xeon E5-2680，Sandy-Bridge 2.7 GHz であり，OS は Red Hat Enterprise Linux 6.4 Santiago，コンパイラは Intel Fortran Compiler 15.0.2 を用いた．またコードのプロファイル取得には，GNU プロファイラ gprof を用いた．

2．簡約化 MHD ソルバー rmhdper の概要

2.1 支配方程式と数値解法

MHD 方程式に非圧縮性と 2 次元性を仮定して簡約化し，Alfvén 時間で規格化された 2 次元の簡約化 MHD 方程式[7]

$$\frac{\partial}{\partial t}\Delta\phi+\{\phi,\Delta\phi\}=\{\psi,\Delta\psi\}+\nu\Delta^2\phi, \tag{1}$$

$$\frac{\partial}{\partial t}\psi+\{\phi,\psi\}=\eta\Delta\psi \tag{2}$$

を考える．ここで$\phi(x,y)$と$\psi(x,y)$はそれぞれ流れ関数と磁束関数を表し，定数νとηはそれぞれ動粘性係数と電気抵抗率を表す．また$\Delta=\nabla^2$は 2 次元のラプラシアン，$\{\cdot,\cdot\}$は Poisson 括弧

$$\{f,g\}=\frac{\partial f}{\partial y}\frac{\partial g}{\partial x}-\frac{\partial f}{\partial x}\frac{\partial g}{\partial y} \tag{3}$$

を表す．プラズマの流速ベクトル$\boldsymbol{u}(x,y)$と磁場ベクトル$\boldsymbol{B}(x,y)$は，$\hat{\boldsymbol{z}}$をz方向の単位ベクトルとしてそれぞれ$\boldsymbol{u}=\nabla\phi\times\hat{\boldsymbol{z}}$，$\boldsymbol{B}=\nabla\psi\times\hat{\boldsymbol{z}}$と表される．

本研究では，周期境界条件の下で方程式（1）−（2）の初期値問題を解くオープンソースのコード rmhdper[5]を使用する．空間の離散化には擬スペクトル法[8]を用いるが，その際の高速フーリエ変換（Fast Fourier Transform: FFT）は FFTW ライブラリ[9, 10]を利用する．時間の離散化には線形項について 2 次の陰的後退差分法[11]を，非線形項について 3 次の Adams-Bashforth 法（AB3）[12]を使用する．ただし，時間ステップの 1 ステップ目には線形項および非線形項ともに Euler 法を使用し，2 ステップ目には非線形項に 2 次の Adams-Bashforth 法を使用する．

2.2 変更前コードのプロファイル

変更前の rmhdper では，処理性能にはあまり注意を払われていなかった．変更を加える前に，gprof によるプロファイル結果（一部抜粋）を**表 1** に示す．また比較のため，以降に述べる変更をすべて行った最終的なプロファイル結果も同時に**表 2** に示した．計測はローカルマシンで行い，グリッド数を$(N_x, N_y)=(2048, 2048)$，ステップ数を500とした．gprof ではサブルーチンごとに呼び出し回数，処理時間などが計測されるが，**表 1** の各列はそれぞれ以下を表す：

% time 全体の総実行時間に対する，サブルーチンの総実行時間の占める割合，

cumulative seconds プロファイルリストにおいて，該当列を含め上位に示されている全てのサブルーチンの累

表 1 変更前の rmhdper コードのプロファイルの一部．グリッド数(N_x, N_y) = (2048, 2048)，ステップ数500.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
37.22	165.47	165.47	5068	0.03	0.03	fft_mp_ktox_
15.18	232.97	67.50				_intel_sse2_rep_memset
9.84	276.70	43.73				_intel_ssse3_rep_memcpy
9.01	316.76	40.06	1500	0.03	0.10	four_mp_poisson_bracket_
5.96	343.28	26.52	500	0.05	0.42	tint_mp_advect_
4.66	363.98	20.70	1002	0.02	0.02	fft_mp_xtok_
	後略					

表 2 変更後の rmhdper コードのプロファイルの一部．グリッド数(N_x, N_y) = (2048, 2048)，ステップ数500.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
32.85	70.91	70.91	4316	0.02	0.02	fft_mp_ktox_
11.79	96.37	25.46				_intel_ssse3_rep_memcpy
10.32	118.65	22.28	1072	0.02	0.06	four_mp_poisson_bracket_
10.30	140.89	22.24	536	0.04	0.20	tint_mp_advect_
6.23	154.34	13.45	1076	0.01	0.01	fft_mp_xtok_
	中略					
1.12	199.39	2.42				_intel_sse2_rep_memset
	後略					

計実行時間(秒),

self seconds　サブルーチン単独の総実行時間(秒),

calls　サブルーチンが呼び出された総回数,

self s/call　サブルーチン自身の一回の呼び出しにかかる平均実行時間(秒),

total s/call　一回の呼び出しあたりの，自分自身とその内部で呼び出す子サブルーチンの実行時間の和の平均(秒),

name　プロファイルを取得したサブルーチンの名前.

プロファイル結果は self seconds により降順に並べられ，**表1**ではその上位6つを示した．表中の空白は，コンパイラの内部関数など，呼び出し回数が決定されなかったことを示す．各サブルーチンの内容は以下のとおりである：

`fft_mp_ktox_`　波数空間における配列を実空間へ逆フーリエ変換(IFFT)する,

`fft_mp_xtok_`　実空間における配列を波数空間へフーリエ変換(FFT)する,

`four_mp_poisson_bracket_`　FFT, IFFTや微分演算関数を呼び出し，Poisson括弧を計算する,

`tint_mp_advect_`　Poisson括弧やAB 3を計算する関数を呼び出し，非線形項の時間ステップを進める,

`_intel_sse2_rep_memset`, `_intel_ssse3_rep_memcpy`　変数への値の代入，コピーを行うIntelコンパイラの内部ルーチン.

表1より，プロファイルが取得されたサブルーチンの中で `fft_mp_ktox_`，`four_mp_poisson_bracket_`，`tint_mp_advect_`の順に総実行時間が大きいことがわかる．高速化を図る場合，サブルーチンの呼び出し回数(calls列)を減らしたり，一回の呼び出しあたりの平均実行時間（self s/call列，total s/call列）を短くすることで高速化を実現できる．各サブルーチンのcalls列とtotal s/call列に注目すると，`fft_mp_ktox_`は呼び出し回数（calls列）が5068回と最も多いために時間がかかっていることがわかる．一方，`four_mp_poisson_bracket_`や`tint_mp_advect_`は内部に留まる平均実行時間（total s/call列）が長いために時間がかかっている．以下では，**表1**の上位6つに関する領域について高速化を図ることとする.

3．rmhdperの高速化

3.1　逐次実行レベルの高速化

前節で挙げた**表1**の上位6つのサブルーチンについて，逐次実行のままで高速化できないか検討した．まずFFTに用いられるライブラリFFTWのバージョンは2.1.5を利用していたが，これを3.3.4も利用できるように変更した．3.3.4は単独CPUの使用でも2.1.5よりいくらか高速化されているが，新たにOpenMPにも対応しており，次節での並列化に利用される．FFTWは，実際にFFTを行う前に初期化段階で変換のプランを作成する仕組みである．今回バージョン3.3.4において使用したプラン作成関数は，実数から複素数配列への`dfftw_plan_dft_r2c_2d`と複素数から実数配列への`dfftw_plan_dft_c2r_2d`である．プラン作成の際16バイト境界にアラインされた配列も指定し，変換の実行には，`dfftw_execute`を2次元の1変換あたり1度だけ実行する．キャッシュ効率を考慮して1次元変換を組み合わせることで更なる高速化を計ることも考えられるが，今回は並列度の変更による自動チューニングのみを考え，アルゴリズムの変更は含めないこととした．また，プラン作成にあたって使用した最適化オプションは`FFTW_PATIENT`である.

さらに呼び出し回数削減による高速化を2つ行った．コード作成時の確実性のために，実行時に割り当てた配列は一旦全要素にゼロ代入を行って初期化し，その後に必要な分だけ代入などの操作が行われていた．サブルーチン内で呼び出しごとに割り当てられる配列ではこの操作が多数回に及び，それが**表1**における`_intel_sse2_rep_memset`の肥大化につながっていると考えられる．実際の代入がすべての要素に及ばないこともあるため，いくらかのゼロ代入は必要であるが，必要最低限に限ることで代入回数を削減した.

呼び出し回数削減の2つ目として，IFFTの回数を削減した．**表1**からステップ数が500の場合，`four_mp_poisson_bracket_`の回数が1500回，`fft_mp_ktox_`の回数が約5000回であることがわかる（余剰分は結果の診断などに用いられる68回）．つまり，**表1**では1時間ステップあたり`four_mp_poisson_bracket_`が3回，`fft_mp_ktox_`が10回呼び出されていたことになる．本コードで使用する方程式(1)，(2)では，Adams-Bashforth法を用いる場合，1時間ステップあたり3つのPoisson括弧を1度ずつ評価する必要がある．これらをFFT $\mathcal{F}$，IFFT $\mathcal{F}^{-1}$ を陽に用いて表せば

$$\{\phi,\Delta\phi\}=\mathcal{F}^{-1}(ik_y\hat{\phi})\mathcal{F}^{-1}(-ik_xk^2\hat{\phi}) - \mathcal{F}^{-1}(ik_x\hat{\phi})\mathcal{F}^{-1}(-ik_yk^2\hat{\phi}), \quad (4)$$

$$\{\phi,\Delta\psi\}=\mathcal{F}^{-1}(ik_y\hat{\psi})\mathcal{F}^{-1}(-ik_xk^2\hat{\psi}) - \mathcal{F}^{-1}(ik_x\hat{\psi})\mathcal{F}^{-1}(-ik_yk^2\hat{\psi}), \quad (5)$$

$$\{\phi,\psi\}=\mathcal{F}^{-1}(ik_y\hat{\phi})\mathcal{F}^{-1}(ik_x\hat{\psi}) - \mathcal{F}^{-1}(ik_x\hat{\phi})\mathcal{F}^{-1}(ik_y\hat{\psi}), \quad (6)$$

となる．ただしここで　$\hat{\phi}:=\mathcal{F}(\phi)$, $\hat{\psi}:=\mathcal{F}(\psi)$ である．各物理量は基本的に波数空間でメモリに保存されており，Poisson括弧の計算でのみ実空間に変換して，この後再び波数空間に変換して時間発展計算を行う．(4)-(6)では12回のIFFT($\mathcal{F}^{-1}$)が現れるが，これをいくらか削減して10回で行っていたのが**表1**である．つまり，Poisson括弧は $\{\phi,f\}=\boldsymbol{u}\cdot\nabla f$ であることを利用し，$\boldsymbol{u}$ を先に計算しておく．`four_mp_poisson_bracket_`をベクトル $\boldsymbol{u}$ とスカラー量 f を受け取るように設計することで，(4)と(6)で重複する $\mathcal{F}^{-1}(ik_y\hat{\phi})$, $\mathcal{F}^{-1}(ik_x\hat{\phi})$ の計算を1回ずつ削減していたわけである.

ところが，(6)に出てくる残りの $\mathcal{F}^{-1}(ik_x\hat{\psi})$, $\mathcal{F}^{-1}(ik_y\hat{\psi})$ は既に(5)で計算したものである．これらも同時に再利用

することで，(6)は four_mp_poisson_bracket_を呼び出すことなく計算できることがわかる．したがって1時間ステップあたり，four_mp_poisson_bracket_を2回，fft_mp_ktox_を8回呼び出すだけでよい．

3.2 OpenMPによる並列計算の実装

本節では，**表1**に挙げられたコード中の処理時間が大きい3つの領域について，個別にスレッド並列処理を実装する．

まず**表1**のサブルーチンの中で，fft_mp_ktox_やfft_mp_xtok_は内部でFFTWライブラリを呼んでおり，FFTW3のスレッド並列処理を利用することができる．この並列処理領域をfftと呼ぶことにする．FFTW3の変換プラン作成では，プラン作成前にマルチスレッド処理命令文を追加することで並列化が可能となる．**図1**のように!$から始まるOpenMP指示文を利用して，FFTW3のプラン作成前にマルチスレッド処理命令文を追加した．fftの並列処理に使用するスレッド数はnthreads_fftで指定される．ただし，iretは FFTW3のスレッド初期化時のエラーチェック変数であり，エラーが発生した場合は0が，正常な場合は1が代入される．

次に，IFFTにおけるエイリアシング[13]に関連した配列コピー領域ktoxの並列化を考える．**図2**にサブルーチンktoxの一部を示すが，これは波数空間におけるcomplex型の2次元配列faを受け取って，real型の2次元配列fyxを返すものである．**図2**に示されたとおり，サブルーチンfft_mp_ktox_全体に使用するスレッド数をnthreads_ktoxで指定して並列処理実行指示文を与えた．ループ構文と構造化されたブロックに対してはdo指示文とworkshare構文によって並列処理命令を行い，FFTの実行部分に対してはmaster構文を使用することでマスタースレッドのみがdfftw_execute関数をコールするようにした．つまりFFTのスレッド並列化は，スレッドの立ち上げを含め全てがFFTW側で処理されている．またmaster構文はスレッド間の同期を自動で行わないため，IFFT実行後にbarrier指示文によって明示的に同期を行っている．

```
!$ call dfftw_init_threads (iret)
!$ if (iret == 0) then
!$  write(error_unit(),*) &
!$    "Error message."
!$  stop
!$ end if
 ! enable multi-thread calculation
!$ call dfftw_plan_with_nthreads &
!$      (nthreads_fft)
    (以下，プランの作成)
```

図1 fftに関するスレッド並列計算．

最後に，tint_mp_advect_から呼び出される3次のAdams-Bashforth法の計算領域ab3では，ブロック分割とサイクリック分割[14]，およびそれらの中間と言えるブロックサイクリック分割を試した．これら3手法で並列化効率に有意な違いは見られなかったため，本研究では最後に試したブロックサイクリック分割を用いている．

4．自動チューニングシステム(ATS)

4.1 背景

マルチスレッド並列処理による並列計算を行う場合，各マシンの最大利用コア数を利用スレッド数として固定して計算[**図3**(a)]することが多い．しかし，スレッド数が大きい場合，メモリアクセス量が演算量を上回ったり，並列処理のための分配，分割および結合といったオーバーヘッドが大きくなる可能性がある．そのため，共有並列計算に利用可能な搭載コア数が必ずしも最適なスレッド数とは限らない．さらに，コード内にある様々な計算領域によって演算量やメモリアクセスのパターンが異なるため，最適なスレッド数も異なることが考えられる．

そこで，スレッド数を実行時に自動的に調節する自動

```
 subroutine ktox (fa, fyx)
  ⋮
!$OMP PARALLEL DEFAULT(shared)
    NUM_THREADS(nthreads_ktox)
!$OMP DO PRIVATE(iky)
    do iky=1, nky
       out2d(iky,1:nkxh) = fa(1:nkxh,iky)
       out2d(iky,nkxh+1:nx-nkxh+1) = 0.0
       out2d(iky,nx-nkxh+2:nx)
            = fa(nkxh+1:nkx,iky)
    end do
!$OMP END DO NOWAIT
!$OMP WORKSHARE
    out2d(nky+1:, :) = 0.0
!$OMP END WORKSHARE
!$OMP MASTER
    call dfftw_execute (p2db)
!$OMP END MASTER
!$OMP BARRIER
!$OMP WORKSHARE
    fyx(:ny,nx+1) = fyx(:ny,1)
    fyx(ny+1,:) = fyx(1,:)
!$OMP END WORKSHARE
!$OMP END PARALLEL
 end subroutine ktox
```

図2 ktoxの並列領域の実装箇所．最初の!$OMP PARALLELの行は，紙面の都合上折り返されている．

チューニングシステム（ATS）を開発した．このシステムは，初期化時にスレッド数を変化させながら数回のフルタイムステップを計算し，経過時間の計測結果をコード内で比較することで最適なスレッド数を自動で選択しようとするものである．このシステムによりコード中の各領域におけるスレッド数が個別に最適化［**図 3** (b)］され，計算中のスレッド数変更も含めた最適な並列計算を自動で行うことが期待できる．

4.2　システムの概要

前節で決定した 3 箇所の並列領域 `fft`, `ktox`, `ab3` について自動チューニングを行った．本コードでの ATS の処理手順は以下のとおりである：

1．インプットファイル中で設定された論理変数 `autotune` の値が真の場合，コードの初期化時に自動チューニングが開始される，
2．まず `ab3`, `ktox` のスレッド数を 1 に固定し，`fft` のスレッド数のみを1, 2, 4, …と変化させながら実行時間を計測し，比較する．最も実行時間が短い時のスレッド数を `fft` の本計算でのスレッド数とする，
3．決定した `fft` のスレッド数はそのまま，`ktox` のスレッド数を 1 に固定し，`ab3` のスレッド数を1, 2, 4, …と変化させながら実行時間の計測，比較を行う．最も実行時間の短いスレッド数を `ab3` の本計算でのスレッド数とする，
4．決定した `fft` と `ab3` のスレッド数はそのまま固定し，`ktox` のスレッド数を1, 2, 4, …と変化させながら実行時間の計測，比較を行う．最も実行時間の短いスレッド数を `ktox` の本計算でのスレッド数とする，
5．各領域で使用するスレッド数が決定され，本計算を行う．

上記の計算手順において，並列領域のスレッド数を決定する順序は，計算時間の大きい順に `fft`, `ab3`, `ktox` とした．また，比較したいスレッド数の最大値は 2 の乗数のうちからインプットファイル内で設定することができ，未設定の場合は環境変数 `OMP_NUM_THREADS` の値が用いられる．自動チューニングに利用する時間ステップ数は，各グリッド数で 1 回の計測が 1 秒程度に収まるよう，グリッド数に応じて 2 から100の間から適当に選んだ．

図 4 は実際に rmhdper コードを実行した時の自動チューニング結果の出力例である．ただし，実行はローカルマシンで行い，グリッド数は $(N_x, N_y) = (512, 512)$ を用いた．**図 4** では，各並列処理領域ごとにスレッド数 `threads` と計測時間 `time` が並んで出力されている．時間計測の結果，各並列処理領域でのスレッド数は `fft` が 8 スレッド，`ab3` が 2 スレッド，`ktox` が 2 スレッドに決定されていることがわかる．

4.3　性能評価

簡約化 MHD コード rmhdper について，

1）初期状態のコード（Before），
2）固定スレッド数を用いた並列化（n threads），
3）ATS を利用した場合，

における計算時間の比較を行い，性能を検証した．

計測には研究室のローカルマシンとスーパーコンピュータ Helios を利用した．計測結果はすべて 1 ステップあたりの時間［秒/ 1 ステップ］に焼き直すが，本計算のステップ数はグリッド数ごとに異なる．グリッド数が64×64の時は

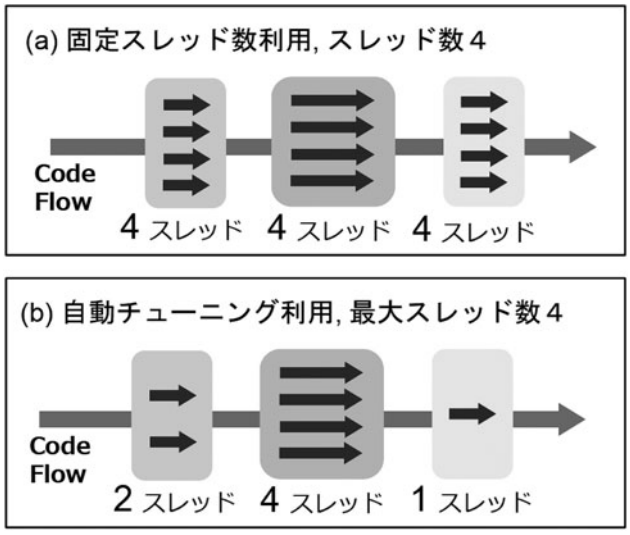

図 3　固定スレッド並列時と ATS 利用時との並列処理領域のスレッド数の違い．(a)固定スレッドによる計算実行時，各並列領域におけるスレッド数は同一となる．(b)ATS を利用した計算実行時，各並列領域におけるスレッド数は異なる最適化がされる．

```
# find optimal number of threads:
 nthreads_fft :   threads   time
                      1     0.9236
                      2     0.7650
                      4     0.9734
                      8     0.6469
 nthreads_ab3 :   threads   time
                      1     0.7228
                      2     0.6757
                      4     0.6973
                      8     1.0146
nthreads_ktox :   threads   time
                      1     0.7758
                      2     0.6731
                      4     0.6760
                      8     0.9993
# optimal number of threads :
     fft :  8    ab3 :  2    ktox :  2

     (以降，本計算アウトプット)
```

図 4　ローカルマシンにおけるグリッド数(N_x, N_y) = (512, 512)の自動チューニングの様子．

ステップ数を80000回，128×128の時は20000回，256×256の時は5000回，512×512の時は2000回，1024×1024，2048×2048および4096×4096の時は500回に調整した．シミュレーションデータの出力は，計測時間には含めないこととした．性能評価を行うにあたり，実行時間の決定は，同条件の計算を5回ずつ行い，それぞれの計算時間を計測後，その中の最高値と最低値を除いた残り3つの値の平均値を用いた．

図5に計算時間の比較結果を示す．**図5**(a)はローカルマシンでグリッド数1024×1024のとき，**図5**(b)はHeliosでグリッド数4096×4096のときの1時間ステップ当たりに掛かる時間を示す．変更前（ラベルB）と固定1スレッド（ラベル1）はいずれも逐次実行であるが，ケース(b)において11.2秒から4.44秒へ短縮されており，これは3.1節で述べた高速化による．自動チューニング（ラベルA）が最も計算時間が短く［ケース(b)では1.72秒］，どの固定スレッドよりも高速化されていることがわかる［ただしケース(a)では4スレッド（ラベル4）とほぼ同等］．

さらに比較するグリッド数の範囲を広げた結果を**図6**および**7**に示す．**図6**はローカルマシンでの実行結果を，**図7**はHeliosでの実行結果を表す．**図6**，**7**においてATS利用時の計算時間は●印付き実線で示されており，常に他の固定スレッド利用時より短い，もしくは同等となっていることがわかる．一方，固定スレッド利用時は，例えばHeliosの16スレッド利用時（○印付き破線）を見ると，グリッド数が大きいときは計算時間が短いが，グリッド数が小さくなると計算時間が比較的長いことがわかる．

このように，搭載CPUコア数を使用スレッド数として固定した場合の計算時間が最も短いわけではない．ATSが実際に選んだ最適なスレッド数のうち，特徴的なものを**表3**に示す．実行のタイミングによって最適スレッド数は多少変化することがあるが，ローカルマシン，Heliosの双方でグリッド数が大きいほどスレッド数も大きくなる傾向を示す．3つの並列化部分のうちでは`fft`が最もスレッド数が多く，`ab3`や`ktox`は同程度で，`fft`に比べて少なめのスレッド数が使われている．これは，並列計算のためのオーバーヘッドが発生しているためであり，ATSを利用した場合はオーバーヘッドが並列化によって得られる高速化効率を上回らないように並列数を自動的に調整している．

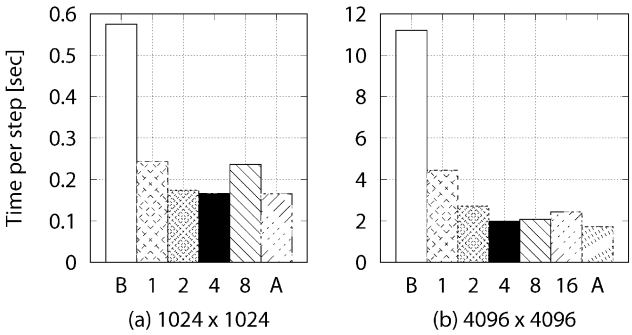

図5　計算時間比較図．(a)1024×1024ローカルマシン，(b)4096×4096スーパーコンピュータHelios．横軸のラベルは，Bは変更前の逐次実行コード(Before)，数字は固定されたスレッド数，AはATSを表す．

4.4　変更後コードのプロファイル

変更後のrmhdperコードに対して，gprofを使って変更前と同条件で再度プロファイルを取得した結果の一部を**表2**に示す．**表1**と比べると，関数`fft_mp_ktox_`，`four_mp_poisson_bracket_`，`tint_mp_advect_`すべてにおいて，1回の呼び出しに対する自分自身と子サブルーチンの平均時間（total s/call）が短くなっていることがわかる．これは，マルチスレッド並列処理を施し上記3つの関数に関わる領域を高速化したためだと考えられる．また，`four_mp_poisson_bracket_`の呼び出し回数（calls）が1500から1072に減少したのは第3節で述べたように，Poisson括弧の演算回数を1ステップあたり2回呼び出しを行うように変更し，また自動チューニングの初期計測によって72回呼び出されたためである．これに伴い，

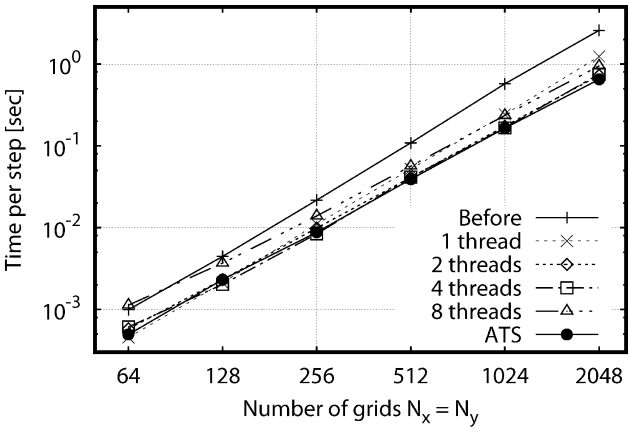

図6　ローカルマシンでの計算時間比較図．

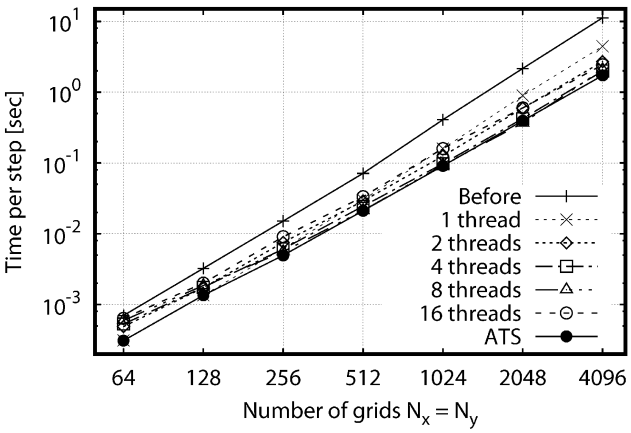

図7　Heliosでの計算時間比較図．

表3　ATSが選んだグリッド数別の特徴的なスレッド数．

	grid	fft	ab3	ktox
local	128^2	1	1	2
	512^2	8	2	2
	2048^2	8	2	2
Helios	256^2	16	1	2
	1024^2	16	8	8
	4096^2	16	8	8

`fft_mp_ktox_`の呼び出し回数(calls)も減っている．さらに，省略可能な配列への代入文を削除したことにより，`_intel_sse2_rep_memset`の実行時間がかなり減少した．

5．まとめ

本研究では，擬スペクトル法による簡約化 MHD コード rmhdper を簡易に高速化するため，スレッド並列度を自動で調節するチューニングシステムを開発した．このシステムにより，複数のマルチスレッド並列処理領域におけるスレッド数が個別に最適化され，自動で最適な並列計算を行うことができるようになった．利用者はグリッドサイズや計算機を変更するたびに最適なスレッド数を試行により選定する作業から解放され，さらにどの固定スレッド数よりも高速な実行が実現される．

グリッド数$(N_x, N_y)=(4096, 4096)$のシミュレーションを Helios で行ったところ，1 ステップあたりの計算時間は，固定 1 スレッドの 4.44 秒から，8 スレッドで 2.08 秒，16スレッドで 2.43 秒，自動並列処理（16，8 スレッド混在）で 1.72 秒へと短縮された．

今回作成した自動並列システムは MPI との併用が可能である．AstroGK[15]などの MPI 並列コードに今回の手法を実装し，ハイブリッド並列にも応用することができるだろう．

謝　辞

本研究の大部分は，国際核融合エネルギーセンターの計算機シミュレーションセンター（IFERC-CSC）における大型計算機 Helios を用いて行われた．兵庫県立大学の沼田龍介氏には，システムの開発にあたり多くの有益な助言を得た．名古屋大学の前山伸也氏には文献[14]を紹介いただいた．また匿名の査読者には，本原稿の改善につながる多数の有益なコメントをいただいた．

参 考 文 献

［1］C. Tapus *et al.,* the IEEE/ACM SC2002 Conf. (2002).
［2］R.L. Ribler *et al.,* Future Gener. Comput. Syst. **18**, 175 (2001).
［3］弓場敏嗣：電気通信大学紀要 **20**, 1 (2007).
［4］須田礼二：情報処理学会研究報告　2011-HPC-129 (14) (2011).
［5］The Gyrokinetics Project, https://sourceforge.net/p/gyrokinetics/code/HEAD/tree/rmhdper/
［6］日本原子力開発機構 IFERC-CSC 利用委員会，IFERC 事業チーム，IFERC-CSC ホームチーム：プラズマ・核融合学会誌 **92**, 157 (2016).
［7］H.R. Strauss, Phys. Fluids **19**, 134 (1976).
［8］J.P. Boyd, *Chebyshev and Fourier Spectral Methods,* 2nd ed. (Dover, Mineola, 2001).
［9］M. Frigo and S.G. Johnson, Proc. IEEE **93**, 216 (2005).
［10］FFTW Homepage, http://www.fftw.org/
［11］C.W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations* (Prentice-Hall, Englewood Cliffs, 1971).
［12］J.D. Hoffman, *Numerical Methods for Engineers and Scientists,* 2nd ed. (Marcel Dekker, New York, 2001).
［13］S.A. Orszag, J. Atmos. Sci. **28**, 1074 (1971).
［14］渡邉智彦，井戸村泰宏：プラズマ・核融合学会誌 **89**, 171 (2013).
［15］R. Numata *et al.,* J. Comput. Phys. **229**, 9347 (2010); *ibid.* **245**, 493 (2013).