修 士 論 文 の 和 文 要 旨

研究科・専攻	大学院　情報理工学研究科　情報・通信工学専攻　博士前期課程		
氏　　名	宮川　大和	学籍番号	１３３１１０２
論 文 題 目	プラズマシミュレーションコード rmhdper と AstroGK の並列計算による高速化		

要　旨

一般的にプラズマシミュレーションの計算には長い時間が必要とされている．そのため大型計算機における並列処理能力が必要とされると同時にシミュレーションコードの最適化が重要となる． 本研究の目的は，オープンソースである磁気流体力学(Magneto-Hydro-Dynamics: MHD) に関するコード rmhdper およびジャイロ運動論コード AstroGK について並列計算を用いた高速化を図ることである．両シミュレーションコードは擬スペクトル法に基づく数値コードであるため，高速フーリエ変換(Fast Fourier Transform: FFT) を利用している．

まず，rmhdper に対して OpenMP によるマルチスレッド並列処理を施して高速化を図った．マシンの最大コア数を利用してマルチスレッド並列処理を行った場合，最大スレッド数が各並列計算領域によって最適なスレッド数とは限らないため，領域ごとのスレッド数を実行時に自動的に調節する自動チューニングシステムを開発した．このシステムを複数のマルチスレッド並列処理領域に利用することで各領域におけるスレッド数が最適化され，自動で最適な並列計算を行うことができるようにした．高速化実験の結果，グリッド数 4096×4096 の時に初期状態に比べ 1 ステップあたりの計算時間が 11.18 秒から 2.68 秒となり，最大で約 76%の時間短縮に成功した．

次に，MPI のみによるマルチプロセス並列処理が可能な AstroGK に対し，OpenMP を利用したマルチスレッド並列処理を施し，ハイブリッド並列処理による高速化を図った．OpenMP による並列処理領域の追加によりリダクション演算にかかる時間が短くなった．また，レイアウト変換にかかる時間も予想していた時間ほどではないが減少した．高速化実験の結果，グリッド数 128×128×64×64 の時において 4096 コアを利用した初期状態の実行時間 3.029 分に比べ，同コア数で 1024 プロセス 4 スレッド利用時の実行時間が 1.973 分となり，最大で約 35%の時間短縮に成功した．

平成 27 年度

修 士 論 文

プラズマシミュレーションコード rmhdper と AstroGK の並列計算による高速化

2016 年 1 月 29 日

電気通信大学
情報理工学研究科

1331102　　宮川　大和

指導教員 龍野　智哉　准教授

副指導教員 山本　有作　教授

目次

概要

一般的にプラズマシミュレーションの計算には長い時間が必要とされている．そのため大型計算機における並列処理能力が必要とされると同時にシミュレーションコードの最適化が重要となる．並列処理による高速化の手法は，主に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がある．本研究の目的は，オープンソースである磁気流体力学 (Magneto-Hydro-Dynamics: MHD) に関するコード rmhdper およびジャイロ運動論コード AstroGK について並列計算を用いた高速化を図ることである．両シミュレーションコードは擬スペクトル法に基づく数値コードであるため，高速フーリエ変換 (Fast Fourier Transform: FFT) を利用した．

まず，rmhdper に対して OpenMP によるマルチスレッド並列処理を施して高速化を図った．マシンの最大利用コア数を利用してマルチスレッド並列処理を行った場合，最大スレッド数が各並列計算領域によって最適なスレッド数とは限らないため，領域ごとのスレッド数を実行時に自動的に調節する自動チューニングシステムを開発した．このシステムを複数のマルチスレッド並列処理領域に利用することで各領域におけるスレッド数が最適化され，自動で最適な並列計算を行うことができるようにした．高速化実験の結果，グリッド数 $(N_x, N_y) = (4096, 4096)$ の時に初期状態に比べ 1 ステップあたりの計算時間が 11.18 秒から 2.68 秒となり，最大で約 76% の時間短縮に成功した．

次に，MPI のみによるマルチプロセス並列処理が可能な AstroGK に対し，OpenMP を利用したマルチスレッド並列処理を施し，ハイブリッド並列処理による高速化を図った．OpenMP による並列処理領域の追加によりリダクション演算にかかる時間が短くなった．また，レイアウト変換にかかる時間も予想していた時間ほどではないが減少した．高速化実験の結果，グリッド数 $(N_x, N_y, N_z, 2N_\lambda, N_E, N_s) = (128, 128, 3, 64, 64, 1)$ の時において 4096 コアを利用した初期状態の実行時間 3.029 分に比べ，同コア数で 1024 プロセス 4 スレッド利用時の実行時間が 1.973 分となり，最大で約 35% の時間短縮に成功した．

1 はじめに

一般的にプラズマシミュレーションの計算には長い時間が必要とされている．そのため大型計算機における並列処理能力が必要とされると同時にシミュレーションコードの最適化が重要となる．並列処理による高速化の手法は，主に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がある．特にマルチスレッド並列処理では利用スレッド数を各計算実行環境における最大利用コア数に固定して使用することが一般的であるが，メモリアクセスや他スレッドの同期待ち，各スレッドへの計算領域の分配や計算結果の結合によるオーバーヘッドが発生するため，必ずしも最大利用コア数が最適なスレッド数とは限らない．そこでコード中の計算時間の長い領域に対し，最適なスレッド数を各並列計算領域ごとに自動で選定する自動チューニングシステムを開発した．このシステムを利用することにより最適な高速化を自動で行うことができる．

本研究の目的は，オープンソースである磁気流体力学 (Magneto-Hydro-Dynamics: MHD) に関するコード rmhdper [1] およびジャイロ運動論コード AstroGK [1, 2] の高速化を実現することである．rmhdper は自動チューニングを利用して， AstroGK はハイブリッド並列処理を利用して高速化を図る．両シミュレーションコードは擬スペクトル法に基づく数値コードであり，高速フーリエ変換 (Fast Fourier Transform: FFT) を利用する．FFT にはマルチスレッドでの利用が可能なフリーソフトウェアの一つである FFTW (Fastest Fourier Transform in the West) [3] を用いている．

まず，比較的計算規模が小さく単一ノードでの計算が可能である rmhdper に対し，OpenMP によるマルチスレッド並列化を施し，自動チューニングシステムを実装した．次に，AstroGK の高速化を図った．現状の AstroGK はマルチプロセス並列処理に用いられる MPI のみによる並列化が施されている．このコードに OpenMP によるマルチスレッド並列処理を加えてハイブリッド並列処理による高速化を図った．

2　基本原理

本節では，本研究において使用するコードの概要および高速化技術の原理，計測に利用するハードウェアおよびソフトウェアについて説明する．

2.1　並列処理と自動チューニング

並列処理には大きく分けて二種類の手法が存在する．ひとつはスレッドを利用するマルチスレッド並列処理，もうひとつはプロセスを利用するマルチプロセス並列処理である [4] ．

マルチスレッド並列処理はメモリ空間を共有する複数の演算プロセッサをスレッドとして計算処理を分担し，データの読み書きを同一の共有するメモリ領域に対して行う共有並列である．並列化には OpenMP の指示文を追加して行うか，コンパイラの自動スレッド並列機能を用いる，もしくはスレッドの POSIX 標準である Pthreads を利用するなどの方法がある．OpenMP やコンパイラの自動スレッド並列機能を利用する場合はプログラムの構造を大きく変更する必要がなく，比較的簡単にプログラムの並列化を行うことができる．ただし，共有メモリ領域を使うためマルチスレッド並列による並列計算の性能向上はマシンに搭載されているコア数で決まり，4 から 32 並列程度が現状の上限である．

一方で，マルチプロセス並列処理は複数の計算ノードで各プロセスが異なるメモリ空間にデータを保持し，ノード間でコミュニケーションをとりながら処理を並列で行う分散並列である．異なるノード間でのやりとりや，並列計算のタイミングを制御するために，MPI ライブラリが用いられる．そのため，プログラムの並列化には一般に多くの変更を必要とするが，並列数の上限はシステムを構成する全演算コア数まで可能である．

2.1.1　自動チューニング

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 1 (a)) することが多い．しかし，スレッド数が大きい場合，共有メモリへのアクセスに制限があるため並列箇所の演算量に対してメモリアクセス量が上回ってしまったり，並列領域内の配列や計算領域を各スレッドに分配および分割，結合する時間が長くなるといったオーバーヘッドが発生する可能性がある．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．さらには同一のコード内においても各計算領域によって演算量やメモリアクセスのパターン

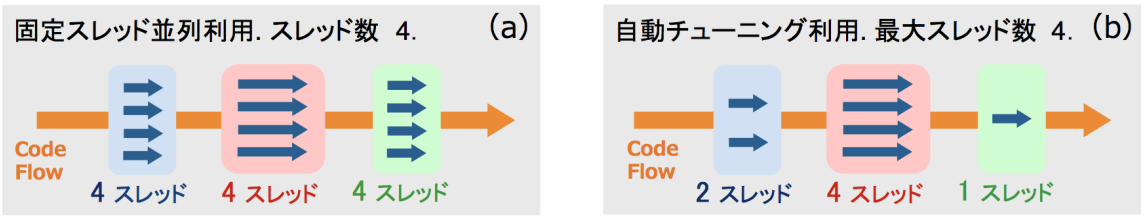


図 1: 固定スレッド並列時と自動チューニングシステム利用時との並列処理領域のスレッド数の違い．(a) 固定スレッドによる計算実行時，各並列領域におけるスレッド数は同一となる．(b) 自動チューニングシステムを利用した計算実行時，各並列領域におけるスレッド数は異なる最適化がされる．

が異なるため，最適なスレッド数は違うと考えられる．そこで，スレッド数を実行時に自動的に調節する自動チューニングシステムを開発した．このシステムは、プログラムの初期化時に数回のフルタイムステップを使用してスレッド数を変化させながら計算時間の計測を行い，各スレッド数での計測結果をプログラム内で比較することで最適なスレッド数を自動で選択するシステムである。このシステムを複数のマルチスレッド並列処理領域に利用することで各領域におけるスレッド数が最適化 (図 1 (b)) され，自動で最適な並列計算を行うことが可能となる．

2.1.2 ハイブリッド並列処理

ハイブリッド並列処理とは，複数プロセッサ（またはコア）を有するマシン複数台を用い，共有メモリ並列と分散メモリ並列を併用した処理手法である [5] ．並列計算を行うコア数が同じ場合，すべてを分散メモリ並列型だけで行う場合に比べて，ハイブリッド並列処理のほうが MPI による通信コミュニケーションの数が減るため，通信レイテンシの減少が期待できる．特に，本研究で使用するジャイロ運動論コード AstroGK では，§4.1 で述べるように巨大配列のレイアウトを変換する回数を減らすことができる．また，速度空間の積分に対応するリダクション演算が随所で必要となるが，この際に全プロセス間の同期作業を一部ノード内のスレッド間同期に置き換えられる効果がある．ただし，マルチスレッド並列処理はスレッド数が増えるとメモリアクセスの制限により性能向上が頭打ちになるため，共有メモリ型並列計算を行うスレッド数を最大にし，分散メモリ型並列計算を行うプロセス数を最小にすることが最適な並列性能を与えるとは限らないことに注意が必要である．

2.2 簡約化 MHD コード

本研究では，2 次元の簡約化 MHD 方程式 [6] について周期境界条件のもとで擬スペクトル法を使用して初期値問題を解くオープンソースのコード **rmhdper** [1] を使用する．使用する方程式は，磁束関数を ψ，流れ関数 ϕ，真空透磁率 μ_0，質量密度 ρ，動粘性率 ν，電気抵抗を η とすると，

$$\begin{cases} \dfrac{\partial}{\partial t}\Delta\phi + \{\phi, \Delta\phi\} = \dfrac{1}{\rho\mu_0}\{\psi, \Delta\psi\} + \nu\Delta^2\phi \ , & (1) \\ \dfrac{\partial}{\partial t}\psi + \{\phi, \psi\} = \eta\Delta\psi & (2) \end{cases}$$

の連立方程式から成り立っており，非圧縮性と 2 次元性を仮定して簡約化されている．ここで，Δ は $\Delta = \nabla^2$ を表し，Poisson 括弧 $\{\}$ は以下のように表される．

$$\{\phi, \psi\} = \frac{\partial\phi}{\partial y}\frac{\partial\psi}{\partial x} - \frac{\partial\phi}{\partial x}\frac{\partial\psi}{\partial y} \ . \tag{3}$$

流速を $\mathbf{u}$ とすると，非圧縮性から

$$\nabla \cdot \mathbf{u} = 0 \tag{4}$$

が成り立つ．さらに 2 次元性を仮定しているため，流速 $\mathbf{u}$ は流れ関数 ϕ を用いて

$$\mathbf{u} = \nabla\phi \times \hat{z} \tag{5}$$

$$= \begin{pmatrix} \dfrac{\partial\phi}{\partial x} \\ \dfrac{\partial\phi}{\partial y} \\ \dfrac{\partial\phi}{\partial z} \end{pmatrix} \times \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \dfrac{\partial\phi}{\partial y} \\ -\dfrac{\partial\phi}{\partial x} \\ 0 \end{pmatrix} \tag{6}$$

と表される．これは式 (4) を満たす．また，磁束密度 $\mathbf{B}$ はある向きへの任意の曲面 S を考えた場合に磁束関数 ψ より，

$$\psi = \int_S \mathbf{B} \cdot d\mathbf{S} \tag{7}$$

が成り立つ．ただし，$d\mathbf{S}$ は面積要素でその向きは曲面 S の法線を向いているとする．

本コードにおける空間の離散化には擬スペクトル法を使用し，その際に利用する高速フーリエ変換 (Fast Fourier Transform: FFT) のためのソフトウェアとしては FFTW3

[3] が使われている．また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を，非線形項について 3 次の Adams-Bashforth 法を使用している．ただし，タイムステップの 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．

2.3 ジャイロ運動論コード

本研究では，MPI のみによる並列処理が可能なオープンソースであるジャイロ運動論コード **AstroGK** [1, 2] を使用する．

粒子位置を $\mathbf{r}$，粒子速度を $\mathbf{v}$ とするとき，ジャイロ運動論では，分布関数を粒子座標（$\mathbf{r}, \mathbf{v}$）の代わりにリング中心位置 $\mathbf{R}$ とリング中心速度 $\mathbf{V}$ を用いてリング中心座標（$\mathbf{R}, \mathbf{V}$）で表す．ここで，$\mathbf{R}$ および $\mathbf{V}$ にはそれぞれ

$$\mathbf{R} = \mathbf{r} + \frac{\mathbf{v} \times \hat{\mathbf{z}}}{\Omega} \ , \tag{8}$$

$$\mathbf{V} = \mathbf{v} \tag{9}$$

が成り立つ．だたし，Ω はサイクロトロン周波数を表す．一方，速度座標は極座標（$V_\parallel, V_\perp, \Theta$）を利用する．このとき，デカルト座標との関係は

$$V_\perp = \sqrt{V_x^2 + V_y^2} \ , \tag{10}$$

$$V_\parallel = V_z \ , \tag{11}$$

$$\tan\Theta = \frac{V_y}{V_x} \tag{12}$$

で表され，$|\mathbf{V}| = V = \sqrt{V_\perp^2 + V_\parallel^2}$ が成り立つ．また，ジャイロ運動論オーダリングのもとで，1 次まで取った粒子分布関数 f は

$$f = \Big(1 - \frac{q\phi}{T_0} + \frac{\mathbf{v} \times \hat{\mathbf{z}}}{\Omega} \cdot \nabla_\perp \Big) f_0 + h \tag{13}$$

と表すことができる．ただし，q は電荷，ϕ は静電ポテンシャル，T_0 は温度，f_0 はマクスウェル分布関数，h はリング分布関数を表し，リング分布関数 h はリング中心座標 $(\mathbf{R}, \mathbf{V})$ で定義されている．

このとき，静電的ジャイロ運動論方程式はリング分布関数 $h(\mathbf{R}, V_\perp, V_\parallel, t)$ より

$$\frac{\partial h}{\partial t} + v_\parallel \frac{\partial h}{\partial Z} - \Big(\frac{\partial \langle\phi\rangle_{\mathbf{R}}}{\partial \mathbf{R}} \times \frac{\hat{\mathbf{Z}}}{B_0} \Big) \cdot \frac{\partial h}{\partial \mathbf{R}} = \frac{q f_0}{T_0} \frac{\partial \langle\phi\rangle_{\mathbf{R}}}{\partial t} + \langle C(h) \rangle_{\mathbf{R}}$$

と記述される．ただし，下付き添え字の $\perp$ および $\parallel$ は一様磁場に対する垂直，平行を意味し，$C(h)$ は線形衝突項を表す．また三角括弧 $\langle\cdot\rangle_{\mathbf{R}}$ はリング中心座標 $\mathbf{R}$ におけるジャイロ平均を表し，$\mathbf{V}=(V_\perp, V_\parallel, \Theta)$ のとき，

$$\langle F(\mathbf{r})\rangle_{\mathbf{R}} = \frac{1}{2\pi}\oint F\left(\mathbf{R}+\frac{\mathbf{V}\times\hat{\mathbf{Z}}}{\Omega}\right)d\Theta \tag{14}$$

となる．

一様磁場に垂直な面についてのフーリエ空間を $\mathbf{k}_\perp=(k_x,k_y,0)$，$k_\perp=|\mathbf{k}_\perp|$ とすると，分布関数 g は

$$g = h - \frac{qf_0}{T_0}\langle\phi\rangle_{\mathbf{R}} \tag{15}$$

が成り立つ．本コードは汎用連続体コードであり，空間の離散化には磁力線垂直方向についてフーリエスペクトル法を，沿磁力線方向について 2 次の中心差分法を，速度空間について Legendre，Laguerre スペクトル積分を使用している．また時間の離散化には線形移流項について陰的 2 次台形公式を，非線形項に 3 次の Adams-Bashforth 法を，衝突項に陰的オイラー法を使用している．

2.3.1 レイアウト変換

AstroGK では分布関数の配列として，

$$g(k_x, k_y, Z, \lambda, E, \sigma, s) \tag{16}$$

のような，7 次元の並列分布関数配列を持っている．

だたし，k_x, k_y は垂直フーリエ空間 $\mathbf{k}_\perp=(k_x,k_y,0)$，$Z$ は沿磁力線方向の座標を表す．また，λ, E, σ は速度空間を表し，エネルギー E とピッチ角 λ は

$$E = V_\perp^2 + V_\parallel^2 \ , \tag{17}$$

$$\lambda = \frac{V_\perp^2}{V^2 B_0} \ . \tag{18}$$

を満たす．σ は一様磁場に対する平行速度の符号 $\sigma=\mathrm{sgn}(v_\parallel)$，$s$ は粒子種を意味する．

この配列を計算に使用するプロセス数に区切り，複数のプロセッサーのメモリに分配することで，大きなスケールの計算を可能としている．コード内において配列は 3 次元配列の形をとっており，1 次元目に Z，2 次元目に σ，3 次元目に残りの 5 変数をまとめて一列に配置することで擬似的な 7 次元配列を作っている．一方でジャイロ運動論方程式を解くためには磁力線方向の移流項（式 (14) の第 2 項）の計算，衝突項の計算，非線形項の計

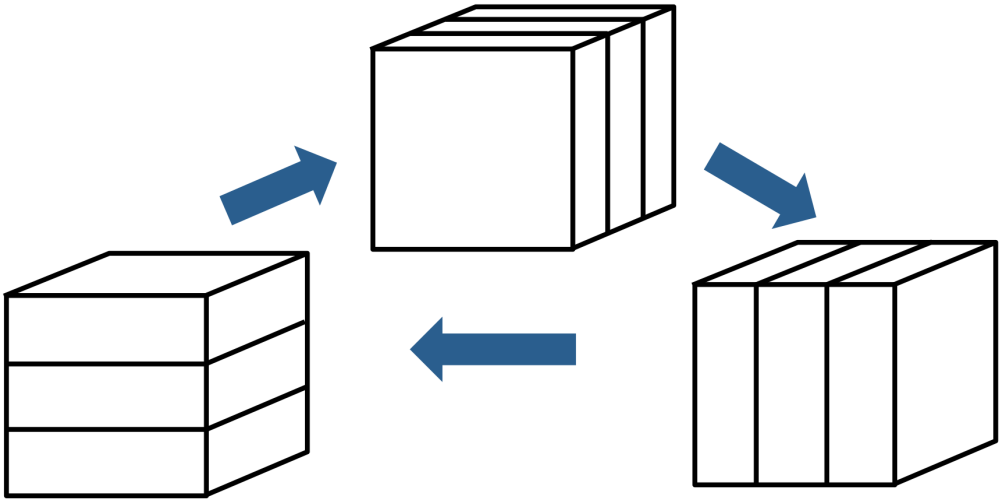


図 2: レイアウト変換イメージ.

算などの複数のステージを必要とするが，各ステージでは 7 次元の分布関数データの中で異なる次元について演算を行う．例えば，メイン関数の線形項は Z に関する差分を含み，衝突項は λ, E, σ を含んだ速度空間の差分を含み，非線形項はフーリエ変換を含むため垂直フーリエ空間の k_x, k_y を必要とする．そこで，各ステージごとでそれぞれ必要な変数を先頭に持ってきてアクセスするための**レイアウト変換**が必要となる．レイアウト変換とは配列に並ぶ 7 変数 $Z, \sigma, k_y, k_x, \lambda, E, s$ の順番を次元ごと変更することであり，図 2 はそのイメージを表している．レイアウト変換を行うためには複数のプロセッサーのメモリに分配された配列変数を再分配する必要があるため，MPI による並列処理に使用するプロセス数が多ければ多いほど，全体の計算実行時間におけるレイアウト変換の処理時間の割合が大きくなる．

2.4 空間の離散化

この節では本研究で使用する空間の離散化について説明する．rmhdper， AstroGK ともに擬スペクトル法を用いた離散化を行っているが，これらは異なる方程式に対する共通の手法なので，ここではモデル方程式について説明する．また，本研究で使用する他の数値解放の手法の説明については省略する．AstroGK で用いる速度空間及び積分の手法については以下の論文 [7] を参照していただきたい．

2.4.1 スペクトル法

$0 \leq x \leq 2\pi$ で周期境界条件を満たす $\varphi(x,t)$ に関する移流方程式

$$\frac{\partial \varphi(x,t)}{\partial t} + c\frac{\partial \varphi(x,t)}{\partial x} = 0 \tag{19}$$

があるとする．ただし，c は定数である．周期関数 $\varphi(x,t)$ は，フーリエ変換公式 [8] より

$$\hat{\varphi}(k,t) = \int_0^{2\pi} \varphi(x,t)e^{-ikx}dx =: \mathcal{F}\big[\varphi(x,t)\big] \tag{20}$$

とおける．ただし，$\hat{\varphi}(k,t)$ はフーリエ空間上の関数を意味する．また，このときフーリエ逆変換 $\mathcal{F}^{-1}$ は

$$\varphi(x,t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} \hat{\varphi}(k,t)e^{ikx}dk =: \mathcal{F}^{-1}\big[\hat{\varphi}(k,t)\big] \tag{21}$$

となる．ここで式 (19) の両辺に e^{-ikx} をかけて 0 から 2π まで x について積分を行うとする．左辺第 1 項および第 2 項についてそれぞれフーリエ変換を用いると，式 (20) と部分積分から，式 (19) 左辺第 1 項は

$$\int_0^{2\pi} \frac{\partial \varphi(x,t)}{\partial t}e^{-ikx}dx = \frac{\partial}{\partial t}\Big(\int_0^{2\pi} \varphi(x,t)e^{-ikx}dx\Big) \tag{22}$$

$$= \frac{\partial}{\partial t}\hat{\varphi}(k,t)\ , \tag{23}$$

式 (19) 左辺第 2 項は

$$\int_0^{2\pi} c\frac{\partial \varphi(x,t)}{\partial x}e^{-ikx}dx = c\big[\varphi(x,t)e^{-ikx}\big]_0^{2\pi} + ikc\int_0^{2\pi} \varphi(x,t)e^{-ikx}dx \tag{24}$$

$$= 0 + ikc\hat{\varphi}(k,t) \tag{25}$$

となる．ここで，式 (24) の右辺第 1 項は $\varphi(x,t)e^{-ikx}$ が x について 0 から 2π までの周期関数であるため 0 となった．式 (23) および式 (25) より式 (19) は

$$\frac{\partial}{\partial t}\hat{\varphi}(k,t) + ikc\hat{\varphi}(k,t) = 0 \tag{26}$$

と変換することができる．これにより，x と t からなる偏微分方程式（19）は k をパラメータとした t の常微分方程式（26）として解くことが可能となる．常微分方程式 (26) を解くと，

$$\hat{\varphi}(k,t) = \hat{\varphi}(k,0)\exp[-ikct] \tag{27}$$

となり，この $\hat{\varphi}(k,t)$ についてフーリエ逆変換 $\mathcal{F}^{-1}$ を用いることで

$$\varphi(x,t) = \mathcal{F}^{-1}\left[\hat{\varphi}(k,t)\right] , \tag{28}$$

$$= \mathcal{F}^{-1}\left[\hat{\varphi}(k,0)\exp[-ikct]\right] \tag{29}$$

と変換して $\varphi(x,t)$ を求めることができる．

2.4.2 擬スペクトル法

非線形項を含む偏微分方程式についてスペクトル法を用いるための解法として擬スペクトル法が存在する．この手法は非線形項の各フーリエ変数について，あらかじめフーリエ逆変換 $\mathcal{F}^{-1}$ を行い，分点上での非線形項の評価を行った上でフーリエ変換 $\mathcal{F}$ を行う手法である．

式（19）に非線形項を追加した場合を考える．

$$\frac{\partial\varphi(x,t)}{\partial t} + c\frac{\partial\varphi(x,t)}{\partial x} + \varphi(x,t)\frac{\partial\varphi(x,t)}{\partial x} = 0 \ . \tag{30}$$

フーリエ変換を $\mathcal{F}$，フーリエ逆変換を $\mathcal{F}^{-1}$ とすると，

$$\mathcal{F}\left[\frac{\partial\varphi(x,t)}{\partial x}\right] = ik\hat{\varphi}(k,t) \ . \tag{31}$$

すなわち

$$\mathcal{F}^{-1}\Big[ik\hat{\varphi}(k,t)\Big] = \frac{\partial\varphi(x,t)}{\partial x} \tag{32}$$

となるため，式（30）の左辺第 3 項について擬スペクトル法を用いると，左辺第 3 項は

$$\mathcal{F}\left[\varphi(x,t)\frac{\partial\varphi(x,t)}{\partial x}\right] = \mathcal{F}\left[\mathcal{F}^{-1}\Big[\hat{\varphi}(k,t)\Big]\mathcal{F}^{-1}\Big[ik\hat{\varphi}(k,t)\Big]\right] \tag{33}$$

とおくことができる．これにより式（26）と同様に，x と t からなる非線形偏微分方程式（30）は k で解くことができる．

2.5 ハードウェア

シミュレーションコードの計算時間の計測を行うにあたって，ローカルマシンとスーパーコンピュータ Helios の 2 台を利用した．

2.5.1 ローカルマシン

ローカルマシンは研究室内にあり，CPU にクアッドコアの Intel Core i7 920, Nehalem マイクロアーキテクチャ 2.67GHz を搭載している．そのため Intel Core i7 に搭載されている「Hyper-Threading」(後述) を利用して，実際に搭載さている 4 コアを仮想的に最大 8 スレッドにすることで並列処理による性能向上をさらに期待することができる．ローカルマシンの L2 キャッシュメモリサイズは 8MB である．

Hyper-Threading

Intel Core i7 には Intel 社が開発した **Hyper-Threading** [9] という技術が応用されている．Hyper-Threading とは，プロセッサ内のレジスタやパイプライン回路の空き時間を有効利用して処理を行う技術である．そのため，1 つのプロセッサを仮想的に 2 つのプロセッサとみなすことができる．このため Hyper-Threading を実装したプロセッサでは，1 つのプロセッサコアに対し 2 つのバスが存在し，クアッドコアの Intel Core i7 では Hyper-Threading を利用して仮想的に最大 8 スレッドでの並列処理による性能向上が期待できる．しかしながら，2 つのアプリケーションが同じプロセッサ要素を同時に利用できないという制約があるため，単純に性能が 2 倍になるわけではないことに注意が必要である．

2.5.2 スーパーコンピュータ Helios

スーパーコンピュータは Helios を利用した．Helios は青森県六ヶ所村の国際核融合エネルギー研究センター (International Fusion Energy Research Center: IFERC) にある BULL SA 社製のスーパーコンピュータである． CPU は Intel Xeon E5-2680 processor, Sandy-Bridge EP 2.7GHz を搭載している．ブレード（基板）は Bullx B510 を使用しており，245 個の筐体に 4410 個のコンピュータノードを持っている（各筐体に 18 ノード）．また 1 つのノードに 16 コアが搭載されており，1 ノードあたりのキャッシュメモリサイズは 20MB である．プラズマの挙動，超高密度磁場での超高温電離ガス，温度や粒子の極端な変動にさらされる素材のデザインなどの核融合に関する根本的な問題の数値計算に主に利用されている．毎年 6 月及び 11 月に更新されるスーパーコンピュータの計算性能の測定値を基にした世界ランキングでは，2015 年 11 月の時点で 60 位にランクインしている [10] ．また日本国内では第 6 位につけている．

2.6 ソフトウェア

2.6.1 OS

ローカルマシンの Operating System は Debian Linux をベースとした Linux Mint 17.1 Rebecca を利用しており，Helios は Linux version 2.6.32 を利用している．

2.6.2 コンパイラ

本研究において，コンパイルする際には **Intel Fortran Compiler** [9] を利用した．ローカルマシンでは Intel Fortran Compiler version 14.0.1 を，Helios では Intel Fortran Compiler version 15.0.2 を利用した． Intel Fortran Compiler は Intel Fortran Composer XE for Linux の一部であり，自動並列化や OpenMP を使用したマルチスレッド並列化に対応し，マルチコア・プロセッサをターゲットとするアプリケーションの開発をサポートしている．

また，GCC の一部である **GNU Fortran Compiler** は使用しなかった．これは，マルチスレッド並列化した FFTW3 のライブラリ関数を含むプログラムコードを **GNU Fortran Compiler** でコンパイルした場合に，計算速度が著しく長くなってしまったためである．そのため，本研究では **Intel Fortran Compiler** のみを使用することにした．

2.6.3 FFTW

高速フーリエ変換を実装するソフトウェアには，マルチスレッドの利用が可能である **FFTW** (Fastest Fourier Transform in the West) (ver 3.3.4) [3] を用いた．FFTW は離散フーリエ変換 (Discrete Fourier Transform: DFT) を計算するためのライブラリであり, マサチューセッツ工科大学 (Massachusetts Institute of Technology; MIT) のマテオ・フリゴ (Matteo Frigo) とスティーブン・ジョンソン (Steven G. Johnson) によって開発された [11] ．FFT を実装したフリーソフトウェアの中ではもっとも高速であるとされ，任意のサイズ N の実数および複素数のデータ配列を $N \log N$ のオーダーの時間で計算することができる．FFTW はヒューリスティックな方法または状況に合わせた最適な尺度で，適切なアルゴリズムを選ぶことで，高速な演算を実現している．また OpenMP によるマルチスレッド演算が可能である．

従来の MHD コード rmhdper およびジャイロ運動論コード AstroGK では FFTW の version 2 を使用していたためマルチスレッドによる並列処理が行えなかった．現在は

FFTW の version 3 を利用可能にするために改良を施したため，FFTW3 の選択が可能となり，FFTW によるマルチスレッド演算を行うことができる．

2.6.4 OpenMP

OpenMP (Open Multi-Processing) [12] とは，コンパイラに対する指示文，関数やサブルーチン，また実行時に用いる環境変数についての仕様であり，OpenMP の規格にしたがう記述を逐次計算プログラムに追加することにより，共有メモリシステム上でマルチスレッド演算を行うことが出来る．すでに広く普及している分散メモリシステム上の MPI によるプログラミングと比較して，OpenMP では逐次計算プログラムから並列計算プログラムへの書き換えの手間がかなり少なくてすむというのが 1 つの特徴である．

2.6.5 MPI

MPI (Message Passing Interface) とは，並列コンピューティングを利用するためにメモリ間のメッセージのやりとりを行う規格の一つである．MPI を実装するライブラリは多くあり，自由に使用できる実装としては MPICH と Open MPI が有名である．OpenMP のように既存のプログラムに簡単なディレクティブを挿入するだけで並列計算プログラムができるわけではなく，並列計算プログラムであることを明確に意識してコーディングする必要がある．本研究では AstroGK でのみ用いられ， Intel MPI Library version 5.0.3.048 を利用した．

2.6.6 プロファイリング

シミュレーションコードを解析する際に，プロファイリングを行った．プロファイリングとは性能解析または性能分析のことを指し，プログラムの実行を通して情報を収集することでプログラムの性能を解析する動的プログラム解析の一種である．プロファイリングを行うことで実行時間やメモリ使用量を最適化するためにプログラムのどの部分を改良すべきか決定することができる．本研究において，プロファイラには GNU binutils の **gprof** を使用した．主に C/C++ 向けだが，Fortran でも動作可能である．プロファイリング結果の各コラムの説明は 3.1 節で行う．

3 簡約化 MHD コード rmhdper の自動チューニング

3.1 コードの解析

高速化を図るにあたって，はじめにシミュレーションコードのプロファイリングを行った．これにより，どの関数がどれだけの処理時間を消費するか，何回呼ばれているかなどを測定し，シミュレーションコード内のどの部分に対して並列処理を施すかを決定した．

表 1 は，改良する前の MHD コード rmhdper に対して gprof を使ってプロファイリングを行った結果の一部である．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．

ここで，表 1 の各コラムの読み方は以下のとおりである．[13]

- % time
  全体の総実行時間に対するその関数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，その関数より上位に示されている全ての関数の総実行時間と，その関数の総実行時間の累計時間.
- self seconds
  その関数の総実行時間.
- calls

% 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 1: 改良前の rmhdper コードのプロファイリング結果の一部．グリッド数 $(N_x, N_y) = (2048, 2048)$，ステップ数 500

その関数が呼び出された総回数．もしその関数が呼び出されなかった場合や，呼び出される回数が決定されなかった (コンパイラの内部ルーチンなど関数がプロファイリング可能な形でコンパイルできなかった) 場合は，空白となる．

- self s/call
  その関数の一回の呼び出しに対する平均実行時間 (秒) ．プロファイリング不可能であった場合は空白となる．
- total s/call
  その関数の一回の呼び出しに対する，その関数とその関数に呼び出されたサブルーチンの平均実行時間 (秒) ．
- name
  プロファイリングされる関数の名前．

プロファイルのリストは，self seconds により各関数を降順に並べ，次に calls で降順に並べ替え，最後に name でアルファベット順に並び替えている．

表 1 では，全体の総実行時間に対するその関数の総実行時間の占める割合 (% time) の大きい順に上位 6 位までを載せた．各関数の説明は以下のとおりである．

- `fft_mp_ktox_`
  配列を波数空間から実空間へ逆フーリエ変換する関数
- `fft_mp_xtok_`
  配列を実空間から波数空間へフーリエ変換する関数
- `four_mp_poisson_bracket_`
  Poisson 括弧を計算する関数
- `tint_mp_advect_`
  Poisson 括弧や ab3 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関数
- `_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 の結果より，プロファイリングが可能であった関数の中で `fft_mp_ktox_`，`four_mp_poisson_bracket_`，`tint_mp_advect_` の順に時間がかかっていることがわかった．並列処理を利用して高速化を図る場合，関数の一回の呼び出しに対するその関数とその関数に呼び出されたサブルーチンの平均実行時間 (total s/call) を短くする事で高速化を実現できると考えられる．さらに，関数の呼び出される回数 (calls) が多い場合

は並列処理による高速化の影響が大きくなると考えられる．ここで各関数の calls と total s/call の列に注目すると，fft_mp_ktox_は呼び出された回数 (calls 列) が 5068 回と最も多いために時間がかかっている事が分かった (単純計算で $5068 \times 0.03 = 152.04$ 秒)．一方，four_mp_poisson_bracket_ および tint_mp_advect_ は関数とその関数に呼び出されたサブルーチンの 1 回あたりの平均実行時間（total s/call 列）が長いために時間がかかっている事が分かった．そこで，fft_mp_ktox_，four_mp_poisson_bracket_，tint_mp_advect_ の 3 つの関数に関わる領域に並列処理を施す事によって高速化を図る事に決めた．

3.2 コードの並列化と改良

プロファイリングの結果を参考に，コードの並列化と改良を行った．並列化には OpenMP によるマルチスレッド並列処理を使用した．

3.2.1 並列計算実装領域

プロファイリングの結果から，本コードでは 3 箇所に対してマルチスレッド並列処理を施す事にした．以下に，並列処理領域とその領域を含む関数名，選定の理由を載せた．

- FFT および IFFT の計算実行領域（fft_mp_ktox_，fft_mp_xtok_）
  本コードが擬スペクトル法に基づくコードであり，非線形項内の Poisson 括弧計算時に使われるため．
- 3 次の Adams-Bashforth 法の計算領域 （tint_mp_advect_mp_ab3_）
  時間の離散化，非線形項の 3 ステップ目以降で使われているため．
- IFFT 時の配列コピー領域（fft_mp_ktox_）
  最も呼び出される回数の多い関数 fft_mp_ktox_内でのループ文であるため．

以降，上記 3 箇所の並列領域は，FFT および IFFT の計算実行領域を **fft**，3 次の Adams-Bashforth 法の計算領域を **ab3**，IFFT 時の配列コピー領域を **ktox** と呼ぶ事にする．

3.2.2 各並列領域の実装方法

各並列領域の並列化実装方法を説明する．

まず，FFT および IFFT の計算実行領域 **fft** は FFTW3 ライブラリを利用して並列化を行った．FFTW は実際の FFT 計算の前に初期化段階でプランを作成する仕組みであるが，特に FFTW3 ではそのプラン作成前にマルチスレッド処理命令文を追加する事で

fft 並列処理領域

```
# ifdef OPENMP
 call dfftw_init_threads (iret)
 if (iret == 0) then
  write(error_unit(),*) "Error during FFTW3 thread initialization."
  stop
 end if
 ! enable multi-thread calculation
 call dfftw_plan_with_nthreads(nthreads_fft)
# endif
```

図 3: **fft** の並列領域の実装箇所

並列化が可能となる（A.1 参照）．図 3 のようにプリプロセスを利用して，FFTW3 のプラン作成前にマルチスレッド処理命令文を OpenMP マクロを定義することで追加できるようにした．**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 を出力し，正常な場合は 1 を出力する．

次に，3 次の Adams-Bashforth 法の計算領域 **ab3** の並列化について説明する．**ab3** は 3.2.3 節で述べるように図 4 のブロックサイクリック分割を利用して並列化を行った．使用するスレッド数は nthreads_ab3 で指定される．

最後に，IFFT 時の配列コピー領域 **ktox** の並列化について説明する．**ktox** は図 5 のように配列コピー領域をメインとした関数 fft_mp_ktox_ 全体に対して並列処理実行指示文を与えることで並列化を行った．使用するスレッド数は nthreads_ktox で指定される．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，FFTW 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実行するようにした．ただし，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．ここで，図 5 において fa，out2d は complex 型，fyx は real 型の 2 次元配列である．

ab3 並列処理領域

—ブロック分割—

```
!$OMP PARALLEL NUM_THREADS(nthreads_ab3)
!$OMP WORKSHARE
  fa(:,:,1) = fac%a(0) * &
              ( - fac%a(1)*fa(:,:,2) - &
              fac%a(2)*fa(:,:,3) + &
              fac%b(1)*dadt(:,:,1) + &
              fac%b(2)*dadt(:,:,2) + &
              fac%b(3)*dadt(:,:,3))
!$OMP END WORKSHARE
!$OMP END PARALLEL
```

—ブロックサイクリック分割—

```
!$OMP PARALLEL NUM_THREADS(nthreads_ab3)
 do iy=1, nky
!$OMP DO SCHEDULE(static, 2)
  do ix=1, nkx
   fa(ix,iy,1) = fac%a(0) * &
                 ( - fac%a(1)*fa(ix,iy,2) - &
                 fac%a(2)*fa(ix,iy,3) + &
                 fac%b(1)*dadt(ix,iy,1) + &
                 fac%b(2)*dadt(ix,iy,2) + &
                 fac%b(3)*dadt(ix,iy,3))
  end do
!$OMP END DO
 end do
!$OMP END PARALLEL
```

図 4: **ab3** の並列領域の変更箇所

ktox 並列処理領域

```
 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 5: **ktox** の並列領域の実装箇所

3.2.3 チャンクの変更

ab3 の並列処理領域に対してチャンクサイズを小さく指定して，配列を複数行ごとに分割するブロックサイクリック分割による並列化を行った．ただし，ここでは 2 次元配列を行列とみなし，配列の 1 次元目を「列」，2 次元目を「行」として話を進める．OpenMP では通常はブロック分割が行われ，例えば 1〜100 のループを 2 スレッドで並列処理する場合には 1〜50 をスレッド 0 に，51〜100 をスレッド 1 に割り当ててブロックごとに処理を実行する．それに対して，サイクリック分割では 1, 3, 5, ... をスレッド 0 に，2, 4, 8, ... をスレッド 1 に割り当てるような配列の 1 列（または 1 行）ごとの分割を行って処理を実行する．本コードで利用したブロックサイクリック分割はブロック分割とサイクリック分割の中間となる分割法である．ここで，チャンクとは 1 つのスレッドに割り当てられる連続した反復演算のまとまりのことであり，チャンクサイズを小さく指定することでキャッシュメモリに保存されたデータに効率良くアクセスすることが可能となる．特に，グリッド数が大きくなりアドレスの参照する範囲が拡大した場合でも，チャンクサイズが小さいために並行に演算する複数のスレッドが互いに近いアドレスを参照し，キャッシュ上のデータにアクセスする確率が高くなるため高速化が期待できる [14].

これに伴い，**ab3** の並列処理領域は図 4 で示されているとおり，`workshare` 構文を使用した並列処理命令から，`do` 二重ループ文の内側ループに `do` 指示文を使用する並列処理命令に変更した．これは，`workshare` 構文が，囲まれたコードの演算を複数のブロックに分割し，それぞれのブロックが 1 回だけ実行されるように各スレッドに割り当てるブロック分割方式を取っており，チャンクサイズの指定ができないためである．一方で `do` 指示文では反復計算のスケジューリングを変更することができるため，静的なスケジュール（`static`）に設定し，各スレッドに割り当てられる反復計算の回数（チャンクサイズ）を指定した.

ここでどのチャンクサイズが最適か，また実際にブロック分割よりサイクリック分割による並列処理の方が高速化が期待できるのかを検証するために計算時間の比較を行った．図 6 は rmhdper コードの **ab3** の並列処理領域に対してブロック分割，サイクリック分割（チャンクサイズ 1），ブロックサイクリック分割（チャンクサイズ 2, 4）をそれぞれ使用して実行した時の計算時間の比較結果を示している．ただし，比較結果はスレッド数を 8 に固定した場合の 1 ステップあたりの時間 [秒/1 ステップ] を利用しており，スーパーコンピュータ Helios で行っている．グリッド数は図 6(a) が 2048×2048，図 6(b) が 4096×4096 となっている．また比較結果を見やすくするため，両図の縦軸のスケールは調整されている.

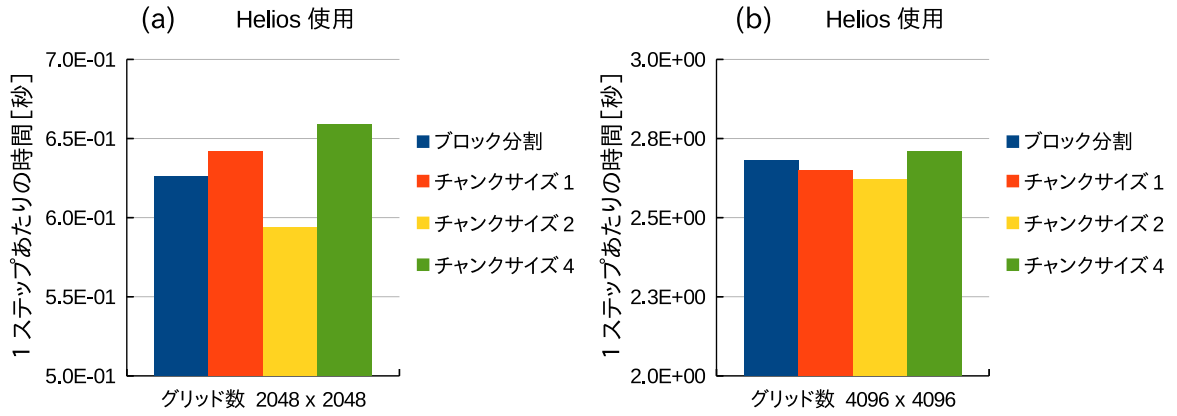


図 6: 分割方法変更時の計算時間比較図（8 スレッド固定）．(a) 2048×2048 スーパーコンピュータ helios，(b) 4096×4096 スーパーコンピュータ helios

図 6 の結果からチャンクサイズが 2 のブロックサイクリック分割の時がもっとも実行時間が短くなることが分かった．これは，本コードで使用している 1 つの命令で複数のデータ処理を行う SIMD (Single Instruction/Multiple Data) 演算と関係があると考えられる．Helios では CPU に Sandy-Bridge-EP を利用（§2.5.2 参照）しており，浮動小数点 SIMD 演算が 256 ビットで行われる Intel AVX 拡張命令セットを使用している．そのため，倍精度浮動小数点演算を 4 要素並列（4×64bit = 256bit）で演算することが可能となる．チャンクサイズ 2 の時の実行時間が最小となる理由は，**ab3** の並列領域で計算する配列が `complex` 型で宣言されており，2 要素並列（2×128bit = 256bit）で行なわれているためだと考えられる．この結果から，rmhdper コードではチャンクサイズを 2 としたブロックサイクリック分割を **ab3** の並列処理領域に使用した．

3.2.4 ヒープメモリの利用

rmhdper では 2 次元の複数の配列を確保する必要があるため，グリッド数が大きい場合に大きなメモリ数を必要とする．そのため，本研究ではローカルマシンで Intel Fortran Compiler によって rmhdper をコンパイルする際に`-heap-arrays` というコンパイルオプションを追加した．これは自動配列および一時的な計算用に作成される配列を，スタックではなくヒープ上に割り当てるためのオプションである．そのため，スタックメモリを使用する場合にくらべヒープメモリを使用するほうがメモリアクセスに時間がかかるので，ローカルマシンでは本来の実行時間より多少時間が長くかかっている．

一方，スーパーコンピュータ Helios ではグリッド数が大きい場合でもスタック上に十分なメモリが存在していたため，このオプションを利用しなかった．

3.2.5 コードの改良

並列化とともにコードに対して 3 点の改良を施した．

1 点目は，配列への 0 代入とコピーの一部削除である．3.1 節の表 1 のプロファイリング結果から`_intel_sse2_rep_memset` および `_intel_ssse3_rep_memcpy` に長い時間がかかっている事が分かった．この原因は，特に関数 `fft_mp_ktox_` 内において，配列への 0 代入とコピーを行っていたためである．配列への 0 代入の一部は本コードの計算には不要である事が判明したため削除した．

2 点目は，FFTW の利用可能なバージョンの拡張である．初期状態のコードでは FFTW の version 2 のみが使用可能であり，マルチスレッドによる並列処理を行う事ができなかった．これに対して FFTW の version 3 を利用可能にすることで並列処理が可能となり，同時に使用する FFTW ライブラリの関数がより高速なものとなった．

3 点目は，Poisson 括弧の演算の計算回数の削減である．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 回であることが分かる．つまり，初期状態のコードでは 1 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事になる．本コードで使用する方程式（2.2 節の式（1），(2）参照）では以下の 3 つの Poisson 括弧が存在しており，通常の擬スペクトル法を用いれば，計 12 回のフーリエ逆変換が必要となる事が分かる．

$$\{\phi, \Delta\phi\} = \frac{\partial\phi}{\partial y}\frac{\partial\Delta\phi}{\partial x} - \frac{\partial\phi}{\partial x}\frac{\partial\Delta\phi}{\partial y} \ , \tag{34}$$

$$\{\psi, \Delta\psi\} = \frac{\partial\psi}{\partial y}\frac{\partial\Delta\psi}{\partial x} - \frac{\partial\psi}{\partial x}\frac{\partial\Delta\psi}{\partial y} \ , \tag{35}$$

$$\{\phi, \psi\} = \frac{\partial\phi}{\partial y}\frac{\partial\psi}{\partial x} - \frac{\partial\phi}{\partial x}\frac{\partial\psi}{\partial y} \ . \tag{36}$$

ここで初期状態のコードを見ると，式（34）と式（36）で一致する一部の要素 $\frac{\partial\phi}{\partial x}, \frac{\partial\phi}{\partial y}$ のフーリエ逆変換は 1 回のみ行い，計 2 回フーリエ逆変換の省略を行っていたために `fft_mp_ktox_` の呼び出し回数が 10 回になっていた事が分かった．しかし，式（35）と式（36）でも一部の要素 $\frac{\partial\psi}{\partial x}, \frac{\partial\psi}{\partial y}$ が一致するため，(36）に存在する全要素 $\frac{\partial\phi}{\partial x}, \frac{\partial\phi}{\partial y}, \frac{\partial\psi}{\partial x}, \frac{\partial\psi}{\partial y}$ のフーリエ逆変換後の配列を保存して再利用する事で関数 `four_mp_poisson_bracket_` の呼び出し回数を 1 回減らす事にした．これにより，1 ス

テップあたりの four_mp_poisson_bracket_ の呼び出し回数が 2 回，fft_mp_ktox_ が 8 回となり，計算時間が短くなった．

3.3 自動チューニング

3.2.1 節で決定した 3 箇所の並列領域に対して自動チューニング（ 2.1.1 節参照）を行った．本コードでの自動チューニングシステムの計算手順は以下のとおりである．

自動チューニングシステムの計算手順

1. コンパイルの段階で OpenMP マクロが定義されている場合，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 **fft** のスレッド数を決定するために， **ab3** ， **ktox** のスレッド数を 1 に固定し， **fft** のスレッド数のみを 1，2，4，8 と変化させながら各スレッド数での実行時間を計測し，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決定する．
3. 決定した **fft** のスレッド数はそのまま固定し，つぎに **ab3** のスレッド数を 1，2，4，8 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を **ab3** の本計算でのスレッド数と決定する．
4. 決定した **fft** と **ab3** スレッド数はそのまま固定し，最後に **ktox** のスレッド数を 1，2，4，8 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を **ktox** の本計算でのスレッド数と決定する．
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，並列領域の自動チューニングは **fft**，**ab3**，**ktox** の順番で行われる．これは，各並列領域の計算比重が **fft**，**ab3**，**ktox** の順に大きいためである．また，上記の例では 8 スレッドまでの実行時間の比較を行っているが，比較したい最大スレッド数はインプットファイル内にて設定が可能である．ただし，未設定の場合は使用する計算機の利用可能なコア数が設定される．

また，自動チューニングに利用するフルタイムステップの計算回数は 2 回から 100 回を取り，これはグリッド数に依存する．すべてのグリッド数で各計測時間が平均約 1 秒で終わるように設定した．設定にあたり，各グリッド数での自動チューニング利用時の 1 ステップあたりのフルタイムステップ実行時間を計測し，数値のスケーリングを測った．計測時間は N_x を x 方向のグリッド数とすると $1.5 \times 10^{-6} \times N_x^{2.1}$ でほぼ近似できる事が分

かったため，コードにて以下のように計算回数 nsteps を設定した．

```
integer :: nsteps, nx
nsteps = ((2048/nx)**2.1)+2
```

nsteps は整数型であるためグリッド数 nx が 2048 以上の場合は 2 が代入される．グリッド数が大きい場合，2 回のフルタイムステップの利用で十分な精度の比較が可能であるためこの値を最小値として設定した．

また，図 7 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，計測はローカルマシンで行い，グリッド数は $(N_x, N_y) = (512, 512)$ に設定した．図 7 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．計測時間の結果，各並列処理領域でのスレッド数は **fft** が 4 スレッド，**ab3** が 2 スレッド，**ktox** が 2 スレッドと決定されていることが分かる．

3.4 数値実験

MHD コード rmhdper について，

1) 初期状態のコード（並列処理領域なし）
2) 並列処理領域追加後のコードに対して **fft**，**ab3**，**ktox** のすべてのスレッド数を固定した場合
3) 自動チューニングシステムを利用した場合

の計算時間の比較を行った．これらの時間を比較することで，自動チューニングシステムを利用時に計算時間が短くなっているかを検証した．

計測には研究室のローカルマシン（2.5.1 参照）とスーパーコンピュータ **Helios**（2.5.2 参照）を利用した．ローカルマシンにはクアッドコアの Intel Core i7 を搭載しており，Hyper-Threading を利用して仮想的に最大 8 スレッドまでマルチスレッド並列処理による性能向上の可能性がある．一方，スーパーコンピュータ Helios では Hyper-Threading を利用せずに最大 16 スレッドまでのマルチスレッド並列処理が可能である．プログラムは Fortran90 をベースとし，一部にプリプロセスを利用することで汎用性を確保している．コンパイラは Intel Fortran Compiler を使用した．

また，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］を使用している．しかし，計測時間を調整するために，計測に利用した本計算のステップ数は各グリッド数ごとに異なる．グリッド数が 64×64 の時はステップ数を 80000 回，128×128 の時はステッ

自動チューニングの出力例

```
# find optimal number of threads:
 nthreads_fft :   threads   time
                          1   1.2541
                          2   1.0447
                          4   0.8952
                          8   0.9559
nthreads_ab3 :  threads   time
                          1   0.9504
                          2   0.8937
                          4   0.9288
                          8   1.8136
nthreads_ktox :  threads    time
                          1   1.1220
                          2   0.8449
                          4   0.9369
                          8   1.6233
# optimal number of threads:
      fft: 4          ab3: 2        ktox: 2
```

（以降，本計算アウトプット）

図 7: ローカルマシン，グリッド数 $(N_x, N_y) = (512, 512)$ の時の自動チューニングの様子

プ数を 20000 回，256×256 の時はステップ数を 5000 回，512×512 の時はステップ数を 2000 回，1024×1024，2048×2048 および 4096×4096 の時はステップ数を 500 回にして計測を行った．また出力データの書き出しは全条件において初期状態と最終状態のみを出力するように設定した．

数値実験を行うにあたり，計測方法は，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 3 つの値の平均値を出力結果とする方法を使用した．

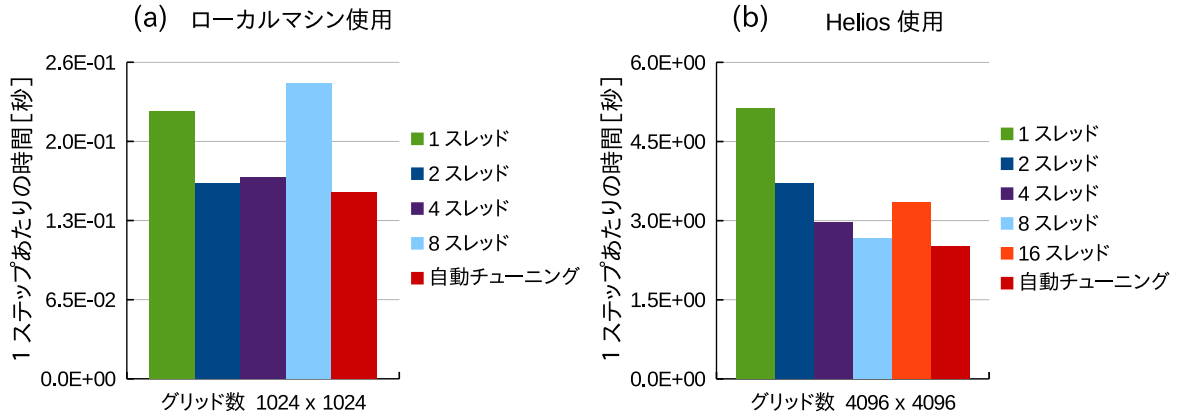


図 8: 計算時間比較図．(a) 1024×1024 ローカルマシン，(b) 4096×4096 スーパーコンピュータ helios

図 8 は計算時間の比較結果の一部である．図 8(a) はローカルマシンでグリッド数 1024×1024 の時の計算時間比較図，図 8(b) は Helios でグリッド数 4096×4096 の時の計算時間比較図である．

図 8 から，自動チューニングシステム利用時の計算時間がどの固定スレッド利用時よりも短いことが分かる．これは，自動チューニングシステムでは各並列処理領域ごとに最適なスレッド数を選定しているためである．

さらに比較するグリッド数を追加して，各グリッド数における計算時間の比較結果を図 9 および図 10 にまとめた．図 9 はローカルマシンでの実行結果を，図 10 は Helios での実行結果で表している．

計算時間の比較図，図 9 および図 10 において自動チューニングシステムを利用時の計算時間は赤い折れ線で示されている．図より，自動チューニングシステム利用時の計算時間が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分かる．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（茶色折れ線）はグリッド数が大きい時は計算時間が短いがグリッド数が小さい場合は計算時間が比較的長いことが分かる．

このように，最大スレッドを固定スレッドとして利用した場合の計算時間が最も短いわけではないことが分かる．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，自動チューニングシステム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で調整している．

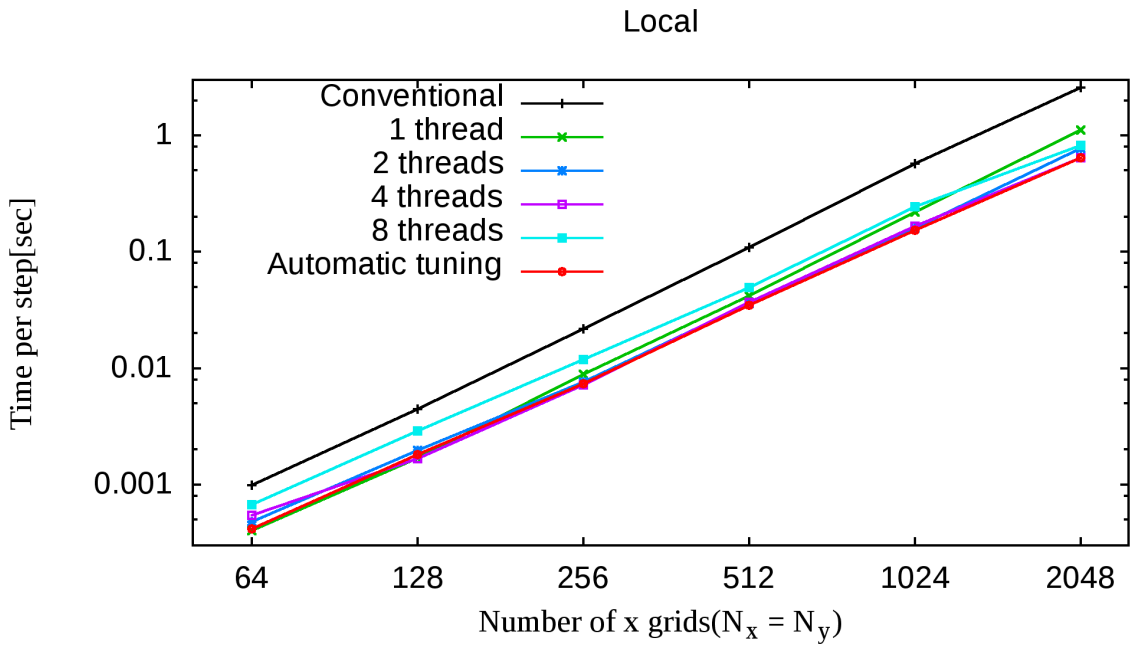


図 9: ローカルマシンでの計算時間比較図

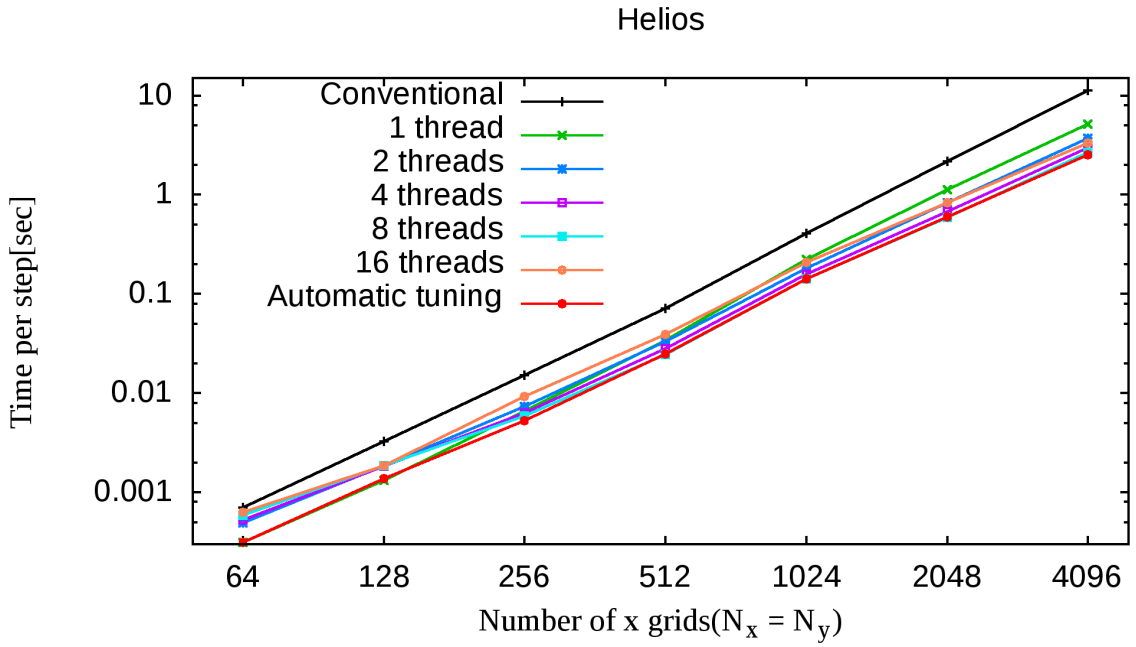


図 10: Helios での計算時間比較図

また，すべての条件において初期コードより固定スレッド利用時および自動チューニングシステム利用時の方が計算時間が短いことが分かる．特に，同じ逐次実行状態である 1 スレッドでの実行結果と大きく差があることが分かった．これは，§3.2.5 で述べたコードの改良にあたって FFTW のバージョンを FFTW2 から FFTW3 にバージョンアップし，gprof を利用したプロファイリングにより本計算に無関係な不要な領域をコードから省いたためである．

以上のことから，自動チューニングシステムを利用したことで最適な高速化を実現できたと言える．

ローカルマシンと Helios の計測結果の比較

ローカルマシンと Helios の自動チューニングシステム利用時の計算時間の比較を行った．図 11 はその比較結果である．

図 11 より，グリッド数が小さい時は Helios の計算時間の方が短いが，グリッド数が大きい時はローカルマシンの計算時間とほとんど差がないことが分かった．

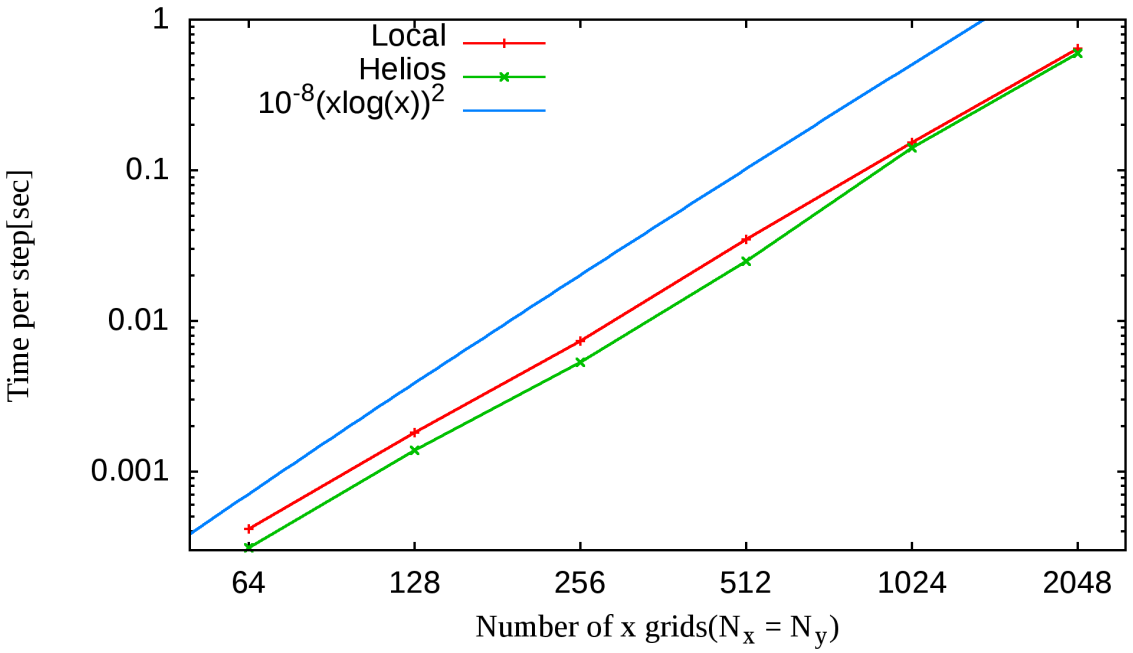


図 11: 自動チューニング利用時の計算時間比較図

改良後のコードのプロファイリング結果

改良後の rmhdper コードに対して，gprof を使ってプロファイリングを行った．表 2 はその結果の一部を表している．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．

表 2 では，全体の総実行時間に対するその関数の総実行時間の占める割合 (% time) の大きい順に上位 5 位まで載せた．表 1 と比べると，関数 `fft_mp_ktox_`，`four_mp_poisson_bracket_`，`tint_mp_advect_` すべてにおいて関数の 1 回の呼び出しに対するその関数とその関数に呼び出された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分かる．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 3.2.5 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，`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．

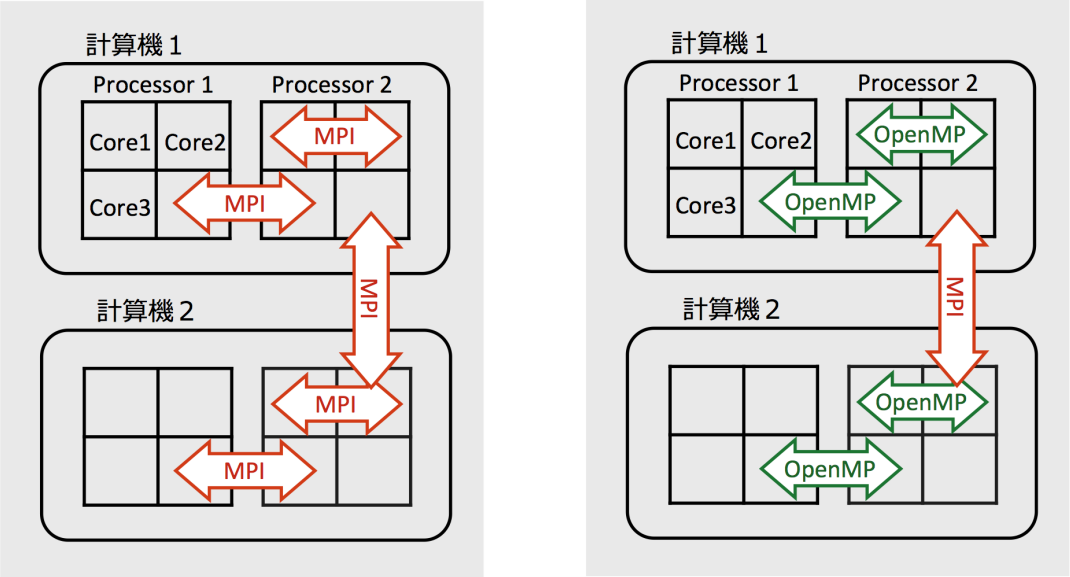
% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
32.85	70.91	70.91	4316	0.02	0.02	`fft_mp_ktox_`
11.79	96.37	25.46				`_intel_ssse3_rep_memcpy`
10.32	118.65	22.28	1072	0.02	0.06	`four_mp_poisson_bracket_`
10.30	140.89	22.24	536	0.04	0.20	`tint_mp_advect_`
6.23	154.34	13.45	1076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

表 2: 改良後の rmhdper コードのプロファイリング結果の一部．グリッド数 $(N_x, N_y) = (2048, 2048)$，ステップ数 500

4　ジャイロ運動論コード AstroGK のハイブリッド並列処理

4.1　高速化の指針

ジャイロ運動論コード AstroGK は MPI のみによる分散メモリを仮定した並列処理が用いられている．本章の目的は，コードの一部に対して OpenMP による共有メモリを利用した並列処理を追加することで高速化を図ることである．現状の AstroGK コードでは共有メモリが利用可能なノード内でも，全コアにプロセスを割り当てて分散メモリ並列処理を行っているため，常にコア間での MPI によるコミュニケーションが必要とされている（図 12a）．これに対して，OpenMP による並列処理を利用してノード内は共有メモリ並列処理を行うことで，コア間におけるコミュニケーションはノード間のみと最小限に抑えることができ（図 12b）全体のコミュニケーションにかかる時間を減らすことができるため，高速化されると予想される．こうした共有メモリ並列と分散メモリ並列を併用した処理の手法はハイブリッド並列処理（2.1.2 節参照）と呼ばれている．



(a)　(b)

図 12:　(a) ノード内でも MPI によるコミュニケーションを実行．(b) ノード内は OpenMP を利用した共有メモリを利用した並列処理．

また，本コードでは MPI による並列処理に使用するプロセス数が多ければ多いほどレイアウト変換（2.3.1 節参照）の処理時間の割合が大きくなる．ハイブリッド並列処理を利用することでレイアウト変換にかかる時間を減少させ，高速化を図ることができると期待される．

4.2 スケーリング

ハイブリッド並列処理に利用するコア数を決定するために，現在のジャイロ運動論コード AstroGK の実行時間が，問題の大きさを保ちながらコア数（プロセス数）を増加させた場合にどのように変化するかのスケーリング（強スケーリングと呼ぶ）をとった．強スケーリングにより評価を行った場合，理想的な実行時間はコア数に反比例して減少する．計測はすべて Helios で行い，コンパイラには Intel Compiler を利用した．また，MPI ライブラリは Intel MPI Library version 5.0.3.048 を利用した．

図 13 は各条件における強スケーリングの結果である．各図におけるグリッド数は，図 13(a) が $(N_x, N_y, N_z, 2N_\lambda, N_E, N_s) = (128, 128, 3, 64, 64, 1)$，図 13(b) が $(N_x, N_y, N_z, 2N_\lambda, N_E, N_s) = (256, 256, 3, 128, 128, 1)$ とした．ただし，N_x, N_y, N_z は実空間のグリッド数，N_λ, N_E は速度空間のグリッド数を表し，N_s は粒子種を表している．また，ピッチ角 λ は一様磁場に対する平行速度の向き $\sigma = \mathrm{sgn}(v_\parallel)$ が正負を取るため，$2N_\lambda$ と表現している．これら 2 種類のランを，総グリッド数の大小に応じて **medium** サイズ，**large** サイズと呼ぶことにする．

図 13 から，どちらの条件においても $1/N$ の定数倍に沿ってスケーリングが途中まで伸びて実行時間が短くなっていることが分かった．図 13(a) の条件では，1024 コアのところでスケーリングが伸び悩みそれ以降は計算時間が短くなっていない．一方，図 13(b) の条件では，4096 コアあたりからスケーリングが伸び悩み 8192 コアで計算時間が少し短くなったものの，16384 コアでは計算時間が長くなってしまっている．

このように，コア数が大きくなると実行時間の減少がみられなくなるのは，1 コアあたりの問題サイズが減少し，MPI による通信時間やレイアウト変換にかかる時間の占める割合が増大するためである．そこで，一部に OpenMP を利用してハイブリッド並列処理を施すことで通信時間とレイアウト変換にかかる時間の占める割合を減らし，スケーリングの伸び悩んでいるコア数での実行時間を減少させることができれば，高速化を実現することができると期待される．

以降の数値実験では **meduim** サイズでは 2048 コア，**large** サイズでは 8192 コアを中心として実行時間の比較を行うことにした．

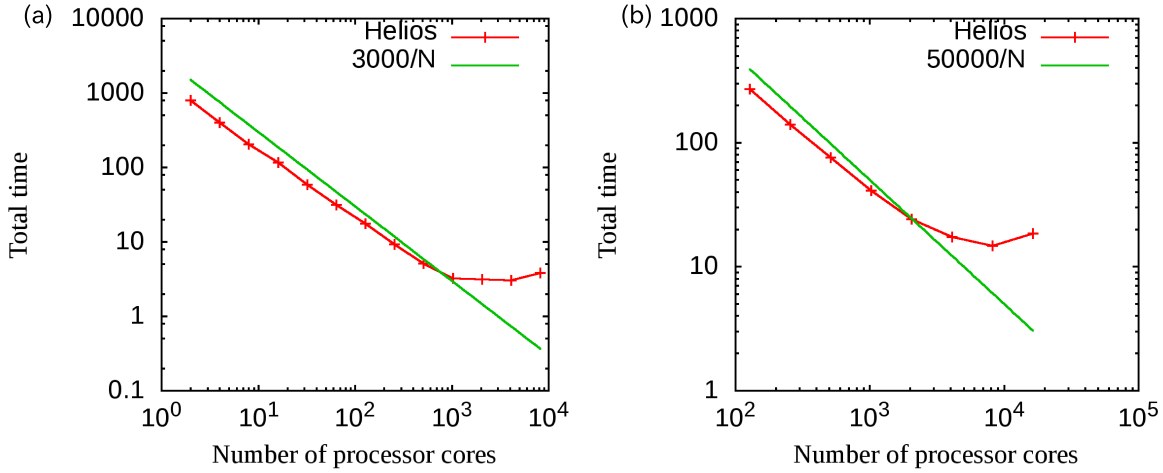


図 13: 初期状態における強スケーリング．(a) $(N_x, N_y, N_z, 2N_\lambda, N_E, N_s) = (128, 128, 3, 64, 64, 1)$ (**medium** サイズ)，(b) $(N_x, N_y, N_z, 2N_\lambda, N_E, N_s) = (256, 256, 3, 128, 128, 1)$ (**large** サイズ)

4.3 コードの並列化と改良

AstroGK に対し，OpenMP によるコードの並列化と改良を行った．並列化を実装する並列領域は各レイアウトで依存性のないループなどプロセスが独立に計算を行う箇所が中心となっている．

4.3.1 コードの解析

rmhdper の高速化と同様に，はじめに AstroGK のプロファイリングを行った．これにより，どの関数がどれだけの処理時間を消費するか，何回呼ばれているかなどを測定し，シミュレーションコード内のどの部分に対して並列処理を施すかを決定する．

表 3 は，改良する前のジャイロ運動論コード AstroGK に対して gprof を使ってプロファイリングを行った結果の一部である．計測は Helios で行い，グリッドサイズを **medium** サイズ，ステップ数を 1000 とした．

表 3 では，全体の総実行時間に対するその関数の総実行時間の占める割合（% time）の大きい順に載せた．各コラムの読み方は 3.1 節にて説明されている通りである．プロファイリングの結果から，`agk_layouts_mp_*`の呼び出される回数（calls）が多いことが分かった．これらはレイアウト変換を必要とする擬似的な 7 次元の分布関数配列から必要

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
7.65	5.39	5.39				_intel_fast_memcmp
6.49	9.97	4.58	205213630	0.00	0.00	agk_layouts_mp_is_idx_g_
6.46	14.52	4.55				for_cpstr
5.53	18.41	3.90	121188835	0.00	0.00	agk_layouts_mp_ie_idx_g_
4.38	21.50	3.09	186097980	0.00	0.00	agk_layouts_mp_it_idx_g_
3.68	24.09	2.60	14634620	0.00	0.00	set_source_
3.29	26.41	2.32	2002	0.00	0.02	dist_fn_mp_timeadv_
3.26	28.71	2.30				fftw_hc2real_128
2.84	30.71	2.00	2000	0.00	0.00	nonlinear_terms_mp_add_nl_
2.67	32.59	1.88				fftw
2.50	34.35	1.76	204376635	0.00	0.00	agk_layouts_mp_ik_idx_g_
2.34	36.00	1.65	4004	0.00	0.00	measure_gather_32_
2.33	37.64	1.64	14634620	0.00	0.00	dist_fn_mp_get_source_term_
2.17	39.17	1.53	4004	0.00	0.00	conserve_diffuse_
2.14	40.68	1.51	11753	0.00	0.00	integrate_moment_c34_
2.10	42.16	1.48	69167220	0.00	0.00	agk_layouts_mp_il_idx_g_
2.10	43.64	1.48	4000	0.00	0.00	transform2_5d_accel_
2.00	45.05	1.41	4404	0.00	0.00	measure_scatter_23_
1.79	46.31	1.26	4004	0.00	0.00	conserve_lorentz_
1.72	47.52	1.21	7317310	0.00	0.00	dist_fn_mp_invert_rhs_1_
1.58	48.63	1.12	89825280	0.00	0.00	agk_layouts_mp_idx_e_
	後略					

表 3: AstroGK コードのプロファイリング結果の一部．グリッドサイズ $(N_x, N_y, N_z, 2N_\lambda, N_E, N_s) = (128, 128, 3, 64, 64, 1)$ (**medium** サイズ)，ステップ数 1000

な変数を各プロセスが取り出すための関数であるため，大量に呼ばれている．しかし，これらの関数はすでに関数ポインタを用いた高速化が施されているため，本研究では直接これらの関数の内容を書き換えることは行わなかった．

一方で，呼び出される回数（calls）はそれほど多くないが，総実行時間（self seconds）が長い関数 `dist_fn_mp_timeadv_`，`nonlinear_terms_mp_add_nl_`なども存在することが分かった．そこで，本研究ではこれらの関数に関わる領域にマルチスレッド並列処理を施すこととした．

4.3.2 並列処理化実装箇所

コードに対して行った並列化の並列箇所を説明する．以下に，並列領域とその領域を含む関数名を一覧に載せた．

- 線形移流項の計算領域
  `dist_fn_mp_invert_rhs`
- 分布関数 g から h への変換の計算領域
  `dist_fn_mp_g_adjust`
- 衝突項の計算領域
  `collision_mp_coserve_diffuse`，`collision_mp_coserve_lorentz`，
  `collision_mp_solfp_lorentz`，`collision_mp_solfp_ediffuse`
- フーリエ変換計算領域
  `agk_transform_mp_transform2_5d_accel`, `agk_transform_mp_inverse2_5d_accel`
- 非線形項の計算領域
  `nonlinear_terms_add_nl`
- 速度空間での積分計算領域
  `le_grids_mp_integrate_moment_c34`，`le_grids_mp_integrate_species1`

各並列領域においては図 14 にあるように，各レイアウト領域でプロセスによる並行処理がされているループ構文に対して OpenMP による並列処理命令を行った．ただし，図 14 は関数 `dist_fn_mp_invert_rhs` 内における並列処理領域であり，`g_lo` は基本レイアウトの構造体である．MPI により各プロセスが自身の担当する配列の数だけループ構文を計算するため，OpenMP による並列処理命令を施した．

また，並列領域の追加にあたり，一部の関数では図 15 のように `reduction` 節を配列に対して利用することでループ構文の並列化を行った．データ依存性がループ構文内に含まれる場合でも正しく演算を行うために利用される `reduction` 節は C 言語では単一変数の

みに限られるが，Fortran では配列に対しても利用可能（B.1 参照，[15] ）である．図 15 は関数 `le_grids_mp_integrate_moment_c34` 内における並列領域であり，4 次元配列である `total` について総和を求める際にデータ依存性によるデータレース（競合）が発生してしまうため，`reduction` 節を利用した．

4.3.3 コードの改良

MHD コード rmhdper と同様に FFTW の利用可能なバージョンの拡張を行った．初期状態のコードでは FFTW の version 2 のみが使用可能であったため，FFTW の version 3 でも利用可能な状態に変更した．この拡張を行うにあたり, FFTW2 と FFTW3 とではプラン作成と計算実行関数の仕様が大きく違っていた（A.9 参照)．

rmhdper は 2 次元の場の量について時間発展を行うコードであるが，AstroGK はより高次元の分布関数について，その 2 次元部分の FFT を多数回行う．したがって，rmhdper の時のように 1 つの 2 次元 FFT を並列化するのではなく，独立な多数回の 2 次元 FFT を並列実行することとした．

4.3.4 MPI-3.0 と OpenMP の比較

2.1 節でも述べたように，一般的に OpenMP は並列処理に共有メモリを利用し，MPI は並列処理に分散メモリを利用する．しかし，MPI-3.0 では共有メモリを利用した並列処理が利用可能になった [16] ． そこで，共有メモリを利用した並列処理を行う場合に，OpenMP と MPI-3 とでどちらの方が性能が良いかを調べるためにテストプログラムコードを作成して比較を行った．

並列処理領域の例

```
!$OMP PARALLEL DO DEFAULT(shared) PRIVATE(iglo)
  do iglo = g_lo%llim_proc, g_lo%ulim_proc
     call invert_rhs_1 (phi, apar, bpar, phinew, aparnew, bparnew, &
          istep, iglo, sourcefac)
  end do
!$OMP END PARALLEL DO
```

図 14: 関数 dist_fn_mp_invert_rhs における並列処理領域の様子

reduction 節の配列への利用

```
!$OMP PARALLEL DO PRIVATE(iglo, ik, it, ie, is, il, fac)&
!$OMP REDUCTION(+:total)
    do iglo = g_lo%llim_proc, g_lo%ulim_proc
       ik = ik_idx(g_lo,iglo)
       it = it_idx(g_lo,iglo)
       ie = ie_idx(g_lo,iglo)
       is = is_idx(g_lo,iglo)
       il = il_idx(g_lo,iglo)
       fac = w(ie,is)*wl(il)

       total(:, it, ik, is) = total(:, it, ik, is) &
               & + fac*(g(:,1,iglo)+g(:,2,iglo))
    end do
!$OMP END PARALLEL DO
```

図 15: 関数 le_grids_mp_integrate_moment_c34 における並列処理領域の様子

図 16 および図 17 は 2 次元配列に対して FFT と IFFT を複数回ループさせて実行し，その時の 1 ステップあたりの実行時間を比較した結果である．図 16 はグリッド数を $(N_x, N_y) = (4096, 4096)$ に固定し，プロセス数（またはスレッド数）を変化させて比較した時の結果である．一方，図 17 は使用するプロセス数（またはスレッド数）を固定し，グリッド数を変化させて比較した時の結果である．計測はすべて Helios で行い，FFT の実行には FFTW ライブラリ version 3 を使用した．

図 16 の結果より，グリッド数が $(N_x, N_y) = (4096, 4096)$ で固定された場合は 1 プロセスまたは 1 スレッドで実行した時を除き，OpenMP 利用時の方が実行時間が短くなることが分かった．また，図 17 の結果からほぼすべてのグリッドサイズにおいて OpenMP 利用時の方が実行時間が短くなることが確認できた．

これは，MPI-3 では共有メモリを使用可能ではあるが処理の分配に時間がかかるために OpenMP より時間がかかったと考えられる．また，FFTW は MPI には対応しているが FFTW の仕様により MPI を利用するには配列のサイズを通常より大きくする必要があり（A.8 参照），MPI-3 で共有メモリを利用する場合でも配列のサイズを調整して大き

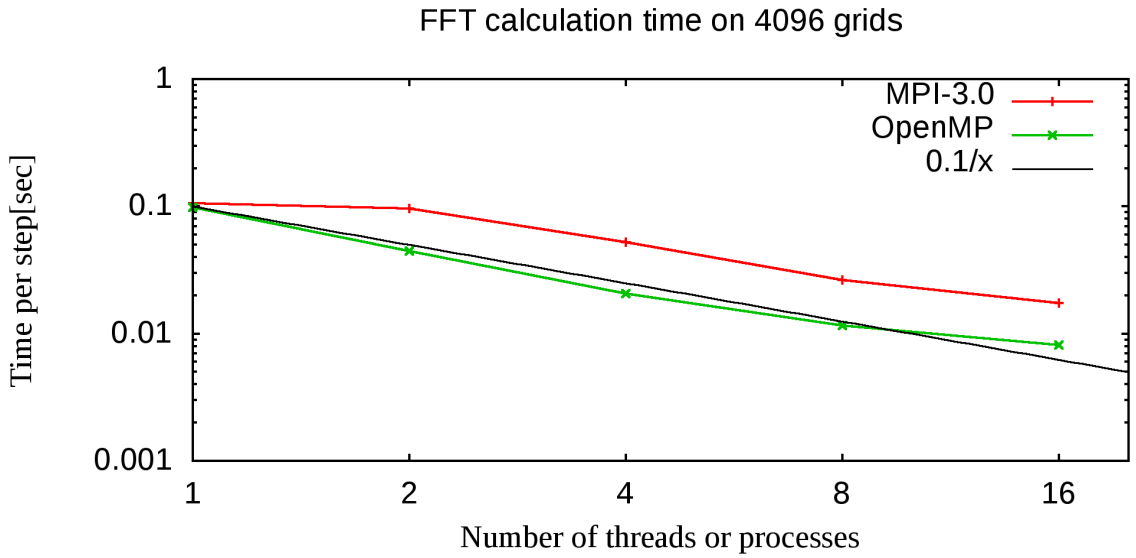


図 16: グリッド数 $(N_x, N_y) = (4096, 4096)$ 時のプロセス数（スレッド数）の変化に対する実行時間の比較

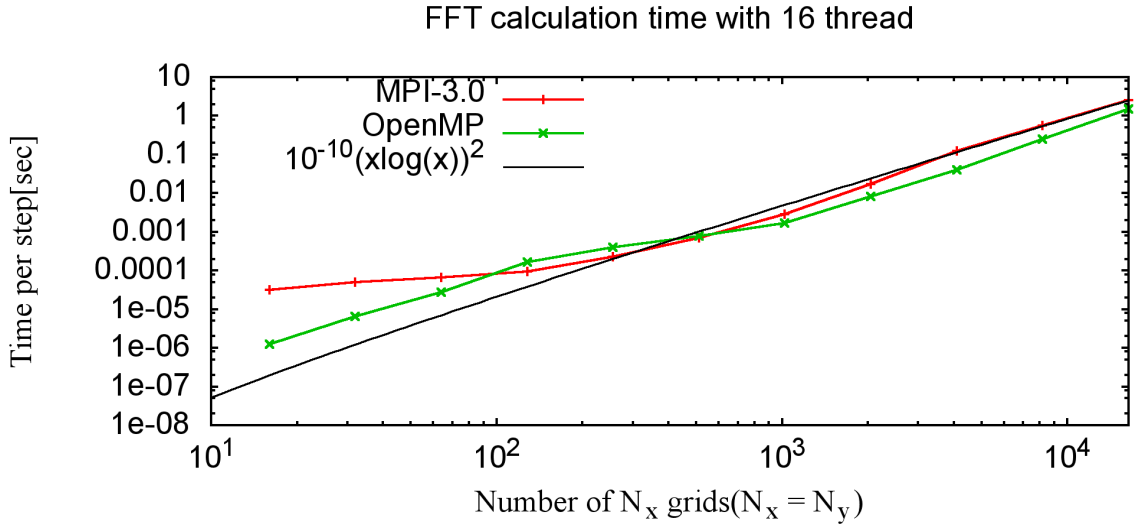


図 17: 16 プロセス（または 16 スレッド）固定時のグリッド数の変化に対する実行時間の比較

く確保する必要であるために実行時間が長くなった可能性がある．以上の結果から，共有メモリを使った並列処理を行う場合は OpenMP のほうが良いと判断した．これによって以降の数値実験では OpenMP を使用することにした．

4.4 数値実験

OpenMP によるマルチスレッド並列処理領域を追加したコードに対して，複数の数値実験を行った．計測はすべて Helios で行った．

プログラムは Fortran90 をベースとして，一部にプリプロセスを利用することで汎用性を確保している．

4.4.1 ジョブ命令方法

Helios においてコードを実行するには，使用したいプロセス数やノード数などを指定して，実行命令を記述したジョブスクリプトを作成し，実行依頼する必要がある．コア数が等しい場合でも，コアに対してどのようにプロセスとスレッドを割り当てるかは実行ごとに異なるため，ジョブスクリプトで指定する必要がある．

図 18 はジョブスクリプトの一例である．`-N` で使用するノード数，`-n` で使用するプロセス数，環境変数 `OMP_NUM_THREADS` で各ノードごとのスレッド数を指定する．図 18 ではノード数が 256，プロセス数が 1024，スレッド数が 2 に指定されている．また，`--ntasks-per-node` によって 1 ノードあたりのプロセス数を指定し，`-c` によって 1 プロセスに割り当てるコア数を指定するため，1 ノードあたり 16 コアをもつ Helios では二つの数字をかけた値が 16 になる必要がある（図 18 では 4×4=16）．

図 18 の例のように，Helios にジョブスクリプトを渡して実行させる場合の実行コマンドは，基本的に `srun` コマンドを利用した．これは大規模な計算ノード・クラスターのためのクラスター・マネージャーおよびジョブ・スケジューリング・システムである SLURM (Simple Linux Utility for Resource Management) [17] の実行コマンドである．大規模な並列計算を行う場合に，Intel MPI の実行コマンド使用時と比べて初期化にかかる時間を減らすことができ，さらに`--place` を加えることによって CPU へプロセスを割り当てる最適なバインディングを自動で行うことができる．ただし，512 ノード以上を使用する場合は `srun` を利用すると計算時間が不安定になる場合があるため，Intel MPI の `mpirun` を使用した．

ジョブスクリプトの例

```
#!/bin/bash
#SBATCH -J agk                      # job name
#SBATCH -o agk.log                  # stdout file name
#SBATCH -e agk.err                  # stderr file name
#SBATCH -N 256                      # number of nodes
#SBATCH --ntasks-per-node 4
#SBATCH -n 1024                     # number of processes
#SBATCH -c 4                        # number of cores per process
#SBATCH -t 00:10:00                 # max run time (hh:mm:ss)
#SBATCH --place                     # CPU bind option

module purge
module load intel intelmpi srun

export OMP_PROC_BIND=true
export OMP_NUM_THREADS=2

srun ./agk medium
```

図 18: 256 ノード，1024 プロセス，2 スレッドで実行依頼をする場合のジョブスクリプト

4.4.2 一定コア数での実行時間比較

まず，使用するコア数を一定にしながらスレッド数を変化させて実行時間の比較を行った．図 19 はプロセス数とスレッド数の割合の変化に対する総実行時間とレイアウト変換の時間の比較図である．測定時のグリッドサイズは **medium** サイズを使用し，128 ノード，2048 コアに固定して比較を行った．そのため，2048 プロセスの時は 1 スレッド，1024 プロセスの時は 2 スレッド，512 プロセスの時は 4 スレッド，256 プロセスの時は 8 スレッド，128 プロセスの時は 16 スレッドとなる．

図 19 の結果より，1 スレッドの時に対して，2 スレッド，4 スレッドの時に実行時間が短くなっていることが分かる．特に，総実行時間は 1 スレッド時（2.84 分）から 2 スレッ

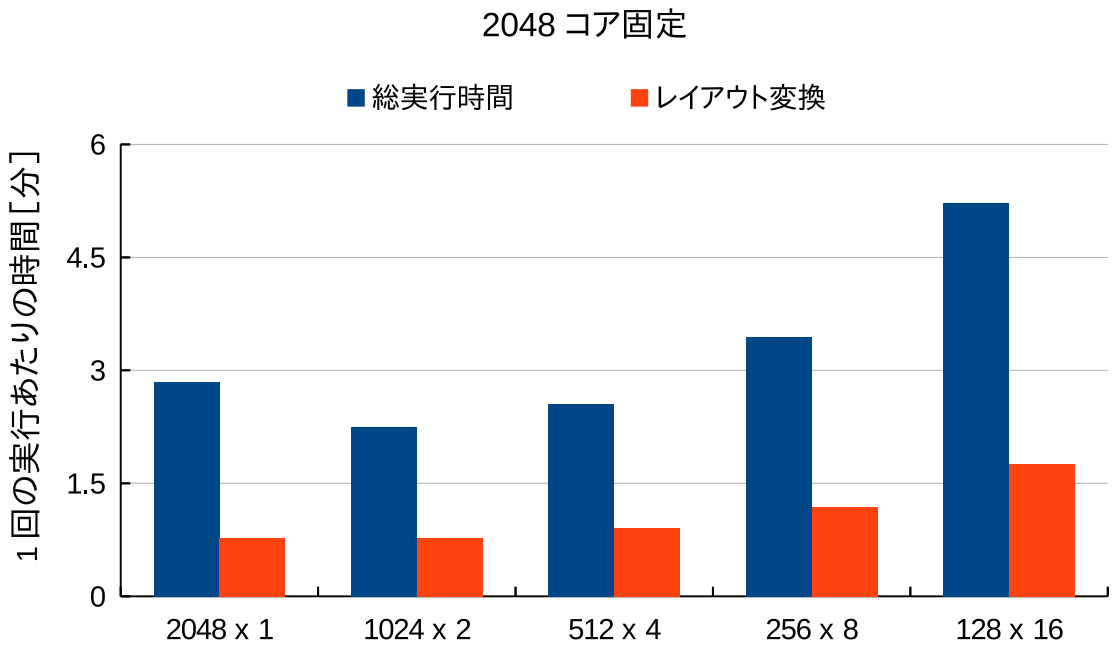


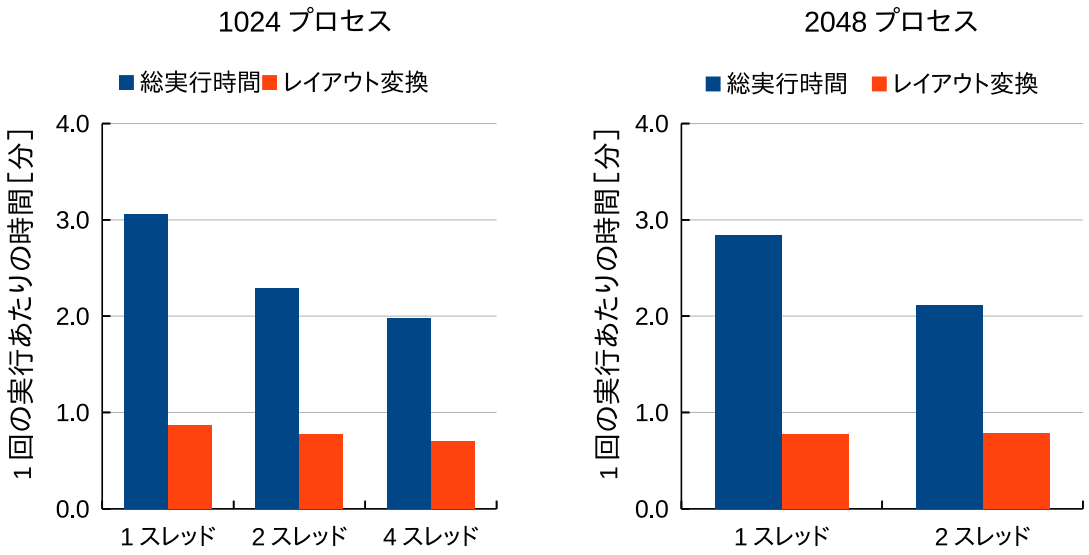
図 19: 2048 コア固定時の総実行時間比較図，グリッドサイズ $(N_x, N_y, N_z, 2N_\lambda, N_E, N_s) = (128, 128, 3, 64, 64, 1)$ (**medium** サイズ)

ド時（2.29 分）へ 19.3% の減少が見られた．しかし，レイアウト変換の時間を減少させることができなかった．これはスレッド数が大きくなることでレイアウト変換の回数は減るが転送データ量が増加したためである．また，スレッド数を大きくしすぎた場合には OpenMP のオーバーヘッドが MPI の通信オーバーヘッドを上回るために再び実行時間が大きくなった．

4.4.3 マルチスレッド並列処理による効果

次にマルチスレッド並列処理による実行時間の変化を調べた．プロセス数のみを固定し，ノード数とともにスレッド数のみを変化させることで総実行時間とレイアウト変換の時間がどのように変化するかの比較を行った．図 20a および図 20b はスレッド数の変化に対する総実行時間とレイアウト変換の時間の比較図である．プロセス数は図 20a では 1024 プロセス，図 20b では 2048 プロセスに固定されている．グリッドサイズは **medium** サイズを使用した．

図 20a および図 20b の結果より，スレッド数の増加に伴って総実行時間が短くなって



(a) (b)

図 20: プロセス数固定時の総実行時間比較図．(a) 1024 プロセス, (b) 2048 プロセス．グリッドサイズ $(N_x, N_y, N_z, 2N_\lambda, N_E, N_s) = (128, 128, 3, 64, 64, 1)$ (**medium** サイズ)

いることが分かった．一方で，レイアウト変換にかかる時間はほぼ一定に保たれている．MPI による通信コミュニケーションやレイアウト変換にかかる時間はプロセス数に依存するため，OpenMP を用いたマルチスレッド並列処理を利用してスレッド数を増やすことで，実行時間を減らすことができたと言える．

4.4.4 使用ノード数の拡張

4.4.2 および 4.4.3 では計測にあたり，常に確保したコア全てにプロセスまたはスレッドを割り当てるようなノード数を利用した．例えば 2048 コアを使用する場合，Helios では 1 ノードあたり 16 コアを持っており 128 ノード ×16 コア = 2048 コアであるため，128 ノードを使用して常に 1 ノードあたり 16 コアを使用していた．しかし，1 ノードあたりの使用するコア数を減らせば 1 ノードあたりの割り当てられるグリッド数が減り，各プロセスでの通信に余裕をもたせることができるため実行時間が短くなる可能性がある．そこで，1 ノードあたり 8 コアを使用した場合と 1 ノードあたり 16 コアを使用した場合の実行時間の比較を行った．図 21 はノード数の変化に対する総実行時間とレイアウト変換の時間の比較図である．1024 プロセス，4 スレッドに固定し，ノード数が 256 ノード（1

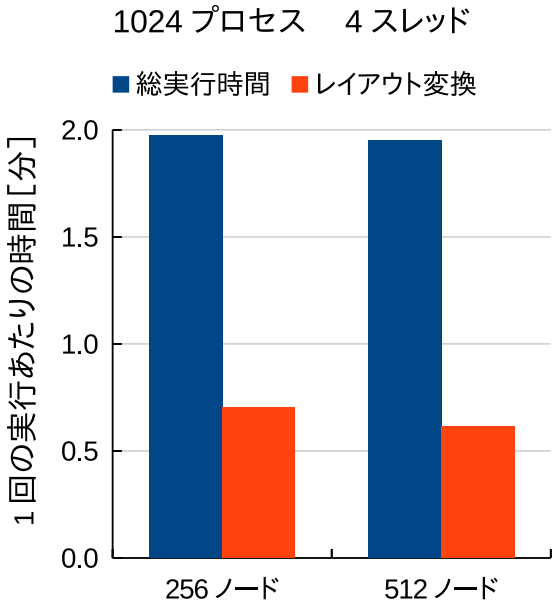


図 21: 2048 プロセス 4 スレッドでのノード数の変化に対する総実行時間比較図，グリッドサイズ $(N_x, N_y, N_z, 2N_\lambda, N_E, N_s) = (128, 128, 3, 64, 64, 1)$ (**medium** サイズ)

ノードあたり 4 プロセス ×4 スレッド）の場合と 512 ノード（1 ノードあたり 2 プロセス ×4 スレッド）の場合とで比較を行った．グリッドサイズは **medium** サイズを使用した．

図 21 より，ノード数を増やしてもほとんど変化がないことが分かった．これは，通信コミュニケーションの回数やレイアウト変換をおこなう要素の数がプロセス数に依存しており，1 つの通信機構を多数のプロセスで共有することに問題があるため，各ノードに割り当てられたグリッド数が減少しても効果が薄かったと考えられる．

4.4.5　初期状態との比較

MPI によるマルチプロセス並列処理のみが可能であった初期状態の AstroGK コードと，OpenMP によるマルチスレッド並列処理を実装しハイブリッド並列処理を利用した時の実行時間の比較を行った．図 22 は各条件における初期状態のコードとハイブリッド並列処理時との総実行時間とレイアウト変換の時間の比較図である．比較時に使用するコア数は各条件ごとに同じであり，ハイブリッド並列処理利用時の結果はその使用コア数のなかで実行結果が最も短い結果を使用している．図 22(a) および図 22(b) は **medium** サイズ，図 22(c) および図 22(d) は **large** サイズをグリッドサイズとして使用した．またコ

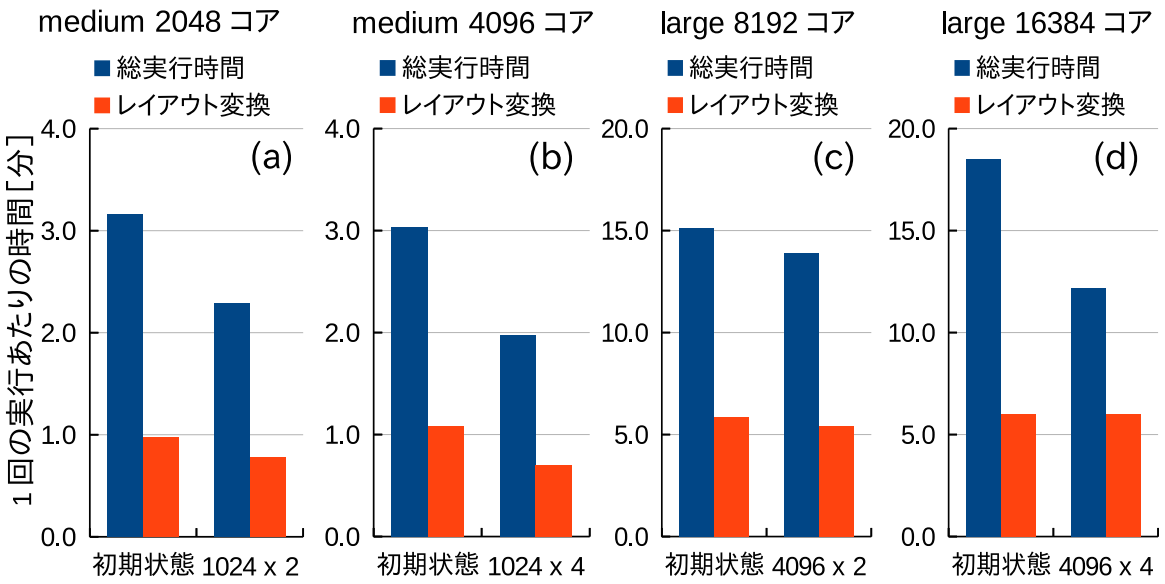


図 22: 初期状態との実行時間比較図． (a) 2048 コア, (b) 4096 コア, (c) 8192 コア, (d) 16384 コア

ア数は図 22(a) が 2048 コア，図 22(b) が 4096 コア，図 22(c) が 8192 コア，図 22(d) が 16384 コアとなっている．

図 22 に見られるように，すべての条件においてハイブリッド並列処理を利用して実行時間を短縮することに成功した．特に，**medium** サイズで 4096 コアを使用した場合は，初期状態で 3.029 分かかっていた総実行時間が 1.973 分（1024 プロセス，4 スレッド利用時）となり，34.9% の時間短縮に成功した（図 22(b)）．また **large** サイズで 16384 コアを使用した場合は，初期状態で 18.491 分かかっていた総実行時間が 12.163 分（4096 プロセス，4 スレッド利用時）となり，34.2% の時間短縮に成功した（図 22(d)）．以上の結果から，ハイブリッド並列処理を利用したことにより，MPI によるマルチプロセス並列処理効果に加えて OpenMP によるマルチスレッド並列処理効果が追加され，従来より高速化がされたと言える．特に，図 13 の強スケーリングにおいて計算効率が落ちるコア数に対してマルチスレッド並列処理を追加することでより大きな効果を得ることができると分かった．新しいコードでの強スケーリングについては 4.4.6 で述べる．

しかし一方で，レイアウト変換にかかる時間はそれほど短縮されず，図 22(b) では 1.084 分から 0.704 分と総実行時間から見て 12.6% 時間短縮されたものの，ほぼすべての条件において横ばいとなった．

高速化の原因

ハイブリッド並列処理を使用したことで初期状態と比べて時間短縮を実現した原因は複数考えられる．

ひとつは，FFTW を FFTW2 から FFTW3 にバージョンアップしたことが挙げられる．図 23 は FFTW2 と FFTW3 を使った場合の実行時間の比較結果を示している．ただし，図 23(a) は 1024 プロセス 2 スレッド，図 23(b) は 512 プロセス 4 スレッドで比較を行い，両図ともグリッドサイズは **medium** サイズとした．両図の結果から FFTW のバージョンをアップしたことによって実行時間が短くなったことが確認できた．特に図 23(a) の時は実行時間が 2.682 分から 2.438 分となり 9.1% の時間短縮に成功した．

もうひとつは reducition 演算の影響が考えられる．MPI による分散メモリ並列から OpenMP による共有メモリ並列に変えたことによって高速化された可能性がある．図 24 は reduction 演算領域に対して，MPI と OpenMP とで並列処理を行った場合の実行時間の比較結果を示している．ただし，図 24(a) は 1024 プロセス 2 スレッド，図 24(b) は 1024 プロセス 4 スレッドで比較を行い，両図ともグリッドサイズは **medium** サイズと

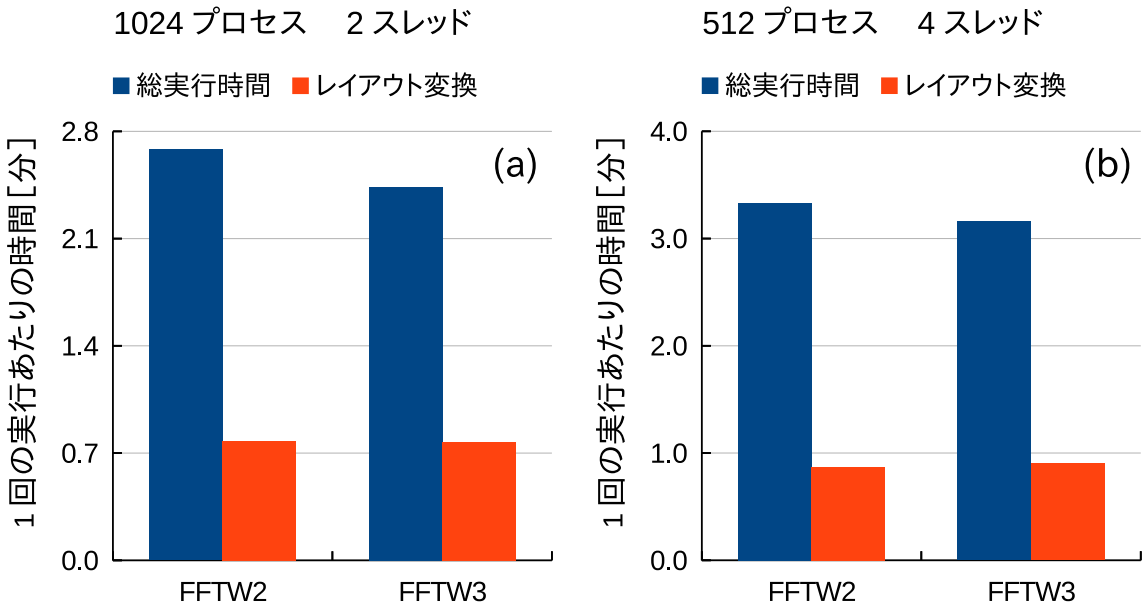


図 23: FFTW の変化に対する総実行時間比較図，**medium** サイズ． (a) 1024 プロセス 2 スレッド, (b) 512 プロセス 4 スレッド

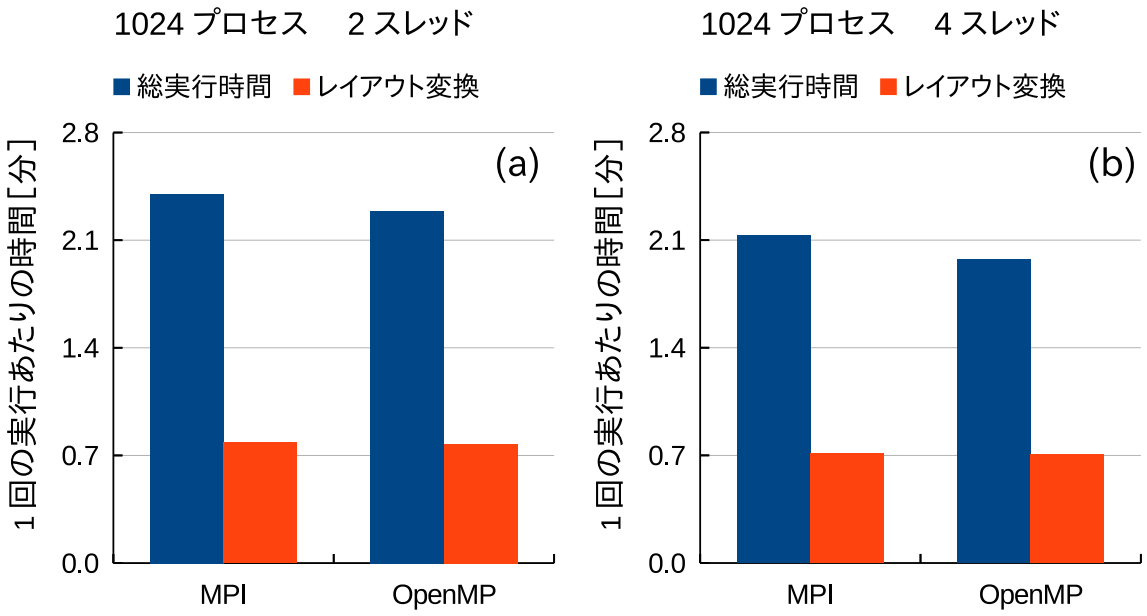


図 24: reduction 演算の変化に対する総実行時間比較図，**medium** サイズ．(a) 1024 プロセス 2 スレッド, (b) 1024 プロセス 4 スレッド

した．両図の結果から reduction 演算領域のみを OpenMP に変化させたことによって実行時間が少し短くなったことが分かる．特に，図 24(b) の時は実行時間が 2.133 分から 1.973 分となり 7.5% の時間短縮に成功した．

以上のように MPI から OpenMP に並列処理方法を変更したことによってレイアウト変換以外の箇所で時間を短縮することに成功したことが高速化につながった．

プロセス数とレイアウト変換時間

図 19 のようにプロセス数を減らした場合でもレイアウト変換の時間が短くならない理由は，1 プロセスあたりのコミュニケーション回数と変換する要素数の関係に原因があると考えられる．それは，プロセス数の増減によって 1 プロセスあたりのレイアウト変換にかかるコストが変化するためである．

簡単のため 64 個の要素を持ったグリッド数 16×16 の 2 次元配列のレイアウト変換を行う場合を考える．2 プロセスの場合は 1 プロセスあたり 32 個の要素を持ち（図 25），4 プロセスの場合は 1 プロセスあたり 16 個の要素を持つ（図 26）．プロセスの数だけ縦に分割していた領域を，横に再分割するレイアウト変換を行った場合，2 プロセスの場合は

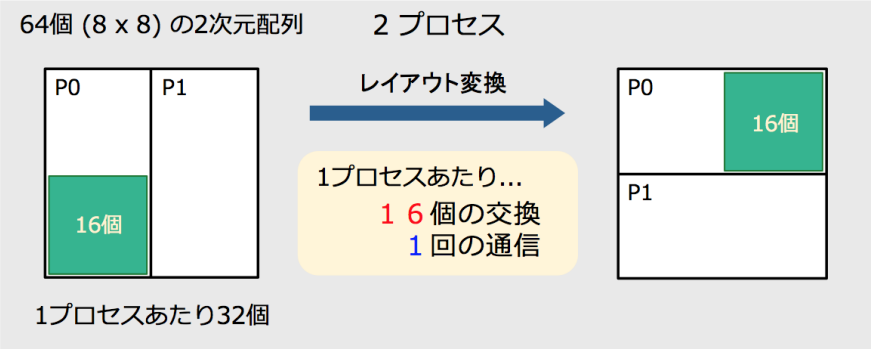


図 25: 2 プロセスでのレイアウト変換にかかるコスト

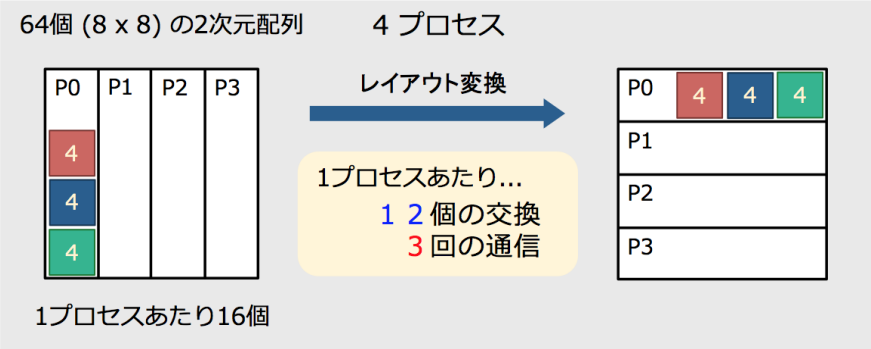


図 26: 4 プロセスでのレイアウト変換にかかるコスト

1 プロセスあたり 1 回の双方向通信と 16 個の要素の交換が必要となる（図 25）．一方，4 プロセスの場合は 1 プロセスあたり 3 回の双方向通信と計 12 個の要素の交換が必要となる（図 26）．このように，プロセス数を増やした場合，1 プロセスあたりのコミュニケーション回数が増加し，交換する要素数が減少する．逆にプロセス数を減らした場合は 1 プロセスあたりのコミュニケーション回数は減少するが，交換する要素数は増加する．そのため，プロセス数が少ない場合はレイアウト変換時に 1 プロセスあたりのメモリ参照数が多くなる．

ここで，ルーフラインモデルを考える [14]．このモデルでは，理論演算性能 F (Flops), 理論メモリ帯域幅 B (Byte/s) の計算機上で演算数 f (Flop), メモリアクセス数 b (Byte)

のプログラムを実行する際の実行演算性能 S (Flops) が

$$S = \frac{f}{f/F + b/B} \tag{37}$$

と与えられる．この関係式を変形すると，

$$\frac{S}{F} = \frac{f/b}{f/b + F/B} \tag{38}$$

とプログラムの対ピーク性能比 $\frac{S}{F}$ がプログラムの演算密度 f/b と計算機の B/F 比によって与えられる．式 (38) の関係式より，演算密度 f/b が低い領域ではメモリ帯域幅 B の影響が大きくなることが分かる．大きい要素数をもつデータのレイアウト変換を行う場合，プロセス数が減少すると 1 プロセスあたりの変換に必要な要素数が大きくなりメモリアクセス数 b が大きくなる．これにより演算密度 f/b が小さくなるため，プロセス数が減少した場合，レイアウト変換にかかる時間を短くするためにはノード間で広いバンド帯域幅が求められると考えられる．一台の計算機におけるハンド帯域幅は固定であるため，プロセス数を小さくすればするほど対ピーク性能比 $\frac{S}{F}$ が悪くなり，レイアウト変換にかかる時間が長くなる．

以上のことから，ハイブリッド並列処理によってレイアウト変換にかかる時間を短くするためには，バンド帯域幅がボトルネックとなっていることが分かった．本研究で使用したスーパーコンピュータ Helios のバンド帯域幅は 76.8 GB/sec だった．これに対して，バンド帯域幅が広いことで知られているメニーコア・コプロセッサを搭載した Xeon Phi 7120P はバンド帯域幅が広い計算機を用いる場合は 352 GB/sec であるため，Xeon Phi を使用したスーパーコンピュータを使用すれば，レイアウト変換の時間を短くすることが可能となる可能性がある．

4.4.6 ハイブリッド並列処理後のスケーリング

一定数以上のコア数に対してハイブリッド並列処理を行った場合の実行時間を含めて，改めてスケーリングをとり，初期状態との比較を行った．

図 27 は初期状態とハイブリッド並列処理後の強スケーリングの比較図である．**medium** サイズ，**large** サイズともにスケーリングによる実行時間の減少の様子が MPI のみでの並列処理の場合に比べ，ハイブリッド並列処理の場合の方が伸びていることが分かる．特に，コア数が大きい場合には特にその差が顕著にあわられ，伸び悩みの曲線が緩やかになったことが見て取れる．

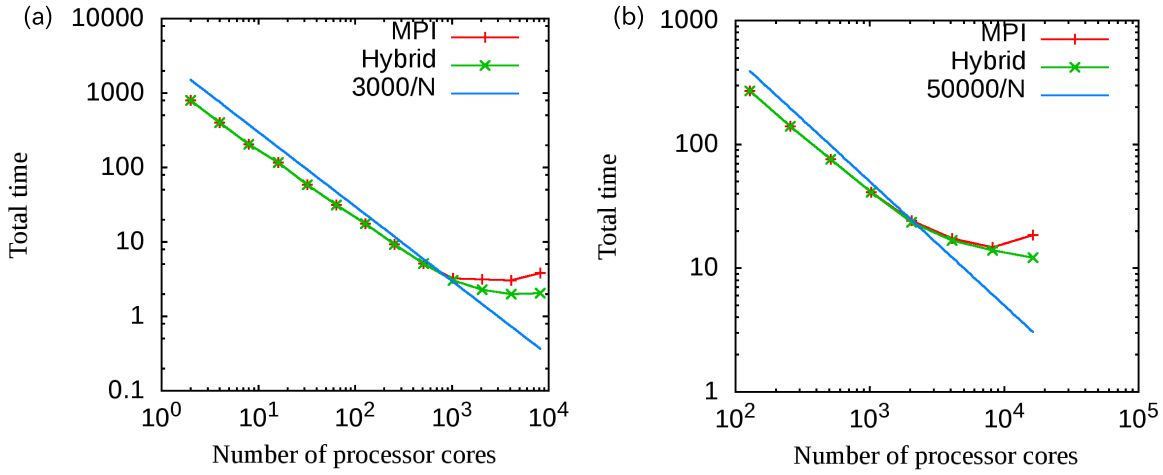


図 27: 強スケーリング．(a) $(N_x, N_y, N_z, 2N_\lambda, N_E, N_s) = (128, 128, 3, 64, 64, 1)$ (**medium** サイズ)，(b) $(N_x, N_y, N_z, 2N_\lambda, N_E, N_s) = (256, 256, 3, 128, 128, 1)$ (**large** サイズ)

5 おわりに

本研究では，プラズマシミュレーションコードに対して並列処理を施すことで高速化を図った．プラズマシミュレーションコードにはオープンソースの簡約化 MHD コード rmhdper とジャイロ運動論コード AstroGK を使用した．両シミュレーションコードは擬スペクトル法に基づく数値コードであるため，高速フーリエ変換を FFTW3 ライブラリを用いて行った．高速化のために用いる並列処理には，OpenMP による共有メモリを使用したマルチスレッド並列処理と MPI による分散メモリを使用したマルチプロセス並列処理，そして共有メモリ並列と分散メモリ並列を併用したハイブリッド並列処理を利用した．

まず第 3 章では，簡約化 MHD コード rmhdper には OpenMP によるマルチスレッド並列処理を施して高速化を図った．高速化の実現にあたり，マシンの搭載利用コア数を利用してマルチスレッド並列処理を行った場合に発生するオーバーヘッドを減らすために，スレッド数を実行時に自動的に調節する自動チューニングシステムを開発した．このシステムを複数のマルチスレッド並列処理領域に利用することで各領域におけるスレッド数が最適化され，自動で最適な並列計算を行うことができるようにした．並列処理を施したことにより，グリッド数 $(N_x, N_y) = (4096, 4096)$ の時に初期状態に比べ 1 ステップあたり

の計算時間が 11.18 秒から 2.68 秒へ最大で約 76% の時間短縮に成功した．

次に第 4 章では，MPI のみによるマルチプロセス並列処理が可能なジャイロ運動論コード AstroGK に対し，OpenMP を利用したマルチスレッド並列処理を施すことでハイブリッド並列処理による高速化を図った．ハイブリッド並列処理によりコアの一部をスレッドとして利用することで，プロセス間で発生するコミュニケーションとレイアウト変換にかかる時間が短くなり高速化が実現されると期待した．OpenMP による並列処理領域の追加の結果，リダクション演算にかかる時間が短くなった．一方で，レイアウト変換にかかる時間も少し短くなったものの，期待していたほどではなかった．高速化実験の結果，グリッド数 $(N_x, N_y, N_z, 2N_\lambda, N_E, N_s) = (128, 128, 3, 64, 64, 1)$ の時において 4096 コアを利用した初期状態の実行時間 3.029 分に比べ，同コア数で 1024 プロセス 4 スレッド利用時は実行時間が 1.973 分となり最大で 34.9% の時間短縮に成功した（図 22(b)）．レイアウト変換にかかる時間がそれほど短くならなかった原因は，プロセス数の減少に伴う 1 プロセスあたりの変換に必要な要素数が増加したのに対して，十分なバンド帯域幅がなかったためだと考えられる．

付録 A　FFTW の記録

A.1　FFTW3 でのマルチスレッドの利用

FFTW を利用して FFT に対してマルチスレッドでの計算を可能にするためには，プログラム内においてプラン作成前にいかに示されている文を挿入し，コンパイルオプションに「-lfftw3_threads」を加える必要がある．ただしプログラムは Fortran90 を使用している．

FFTW のマルチスレッド並列処理命令文

```
integer :: iret, nthreads_fft

call dfftw_init_threads(iret)
call dfftw_plan_with_nthreads(nthreads_fft)
```

使用したいスレッド数は，integer 型の変数 `nthreads_fft` に代入する．また，`call dfftw_init_threads(iret)` の返り値である `iret` は，FFTW3 のスレッド初期化時にエラーが発生した場合は 0 を出力し，正常な場合は 1 を出力する．

A.2　FFTW のビルドとインストール

FFTW を利用するにあたり，ローカルマシンにおいて Intel Fortran Compiler を使用して独自にビルドとインストールを行った．これは，ローカルマシンで FFTW を Intel Fortran Compiler に対応させ，単精度および倍精度での利用を可能にするためである．以下の文は FFTW3 倍精度用のビルド時の configure オプションである．

Intel Fortran Compiler 用の FFTW ビルドオプション

```
CC=icc F77=ifort CFLAGS="-O3" FFLAGS="-O3 -march=corei7"
./configure --enable-threads --enable-openmp --enable-sse2
```

`--enable-threads` はコンパイル時の FFTW マルチスレッドライブラリを利用可能にし，`--enable-openmp` は FFTW の OpenMP 命令文での並列化を可能にするオプションである．また，`--enable-sse2` はベクトルユニットを利用可能にするオプションであ

る．一方で単精度での FFTW3 の利用を可能にするためには--enable-float を追加すれば良い．また，Helios ではデフォルトでインストールされている FFTW ライブラリを利用している．

A.3 動的メモリの割り当て

FFTW では，より最適なパフォーマンスを得るために，メモリ中で 16 バイト境界にアラインされた配列を使用することが良いとされている．アラインすることによって，プラン作成時に FFTW の実行関数を高速に使用することができる．しかし，スタンダードな Fortran の配列は使用するコンパイラ次第ではアラインが保証されないため，C 言語の外部関数を利用する必要がある．これは Fortran の配列ポインタを利用することにより可能となる．以下にアラインされた配列の作成例を示す．

FFTW 用のアラインされた配列作成方法

```
  use, intrinsic :: iso_c_binding

  real (C_DOUBLE), pointer :: in2d(:,:)
  complex (C_DOUBLE_COMPLEX), pointer :: out2d(:,:)
  type (C_PTR) :: pr, pc
  integer :: nx, ny

  pr = fftw_alloc_real (int(ny * nx, C_SIZE_T))
  pc = fftw_alloc_complex (int((ny/2+1) * nx, C_SIZE_T))

  call c_f_pointer (pr, in2d, [ny,nx])
  call c_f_pointer (pc, out2d, [ny/2+1,nx])

---in2d, out2d は通常の 2 次元配列として利用---

  call fftw_free (pr)
  call fftw_free (pc)
```

上記の例では，2 次元配列である real 型の in2d，complex 型の out2d についてアライメントを行っている．ポインタは fftw_free によって解放される．また，C オブジェクトポインタと関数ポインタとの相互利用性を支援して C 言語に対応する Fortran のポインタを作成する C_PTR 型を利用するために，組込みモジュール iso_c_binding を追加する必要がある．

A.4 プランの作成とフラッグ

FFTW では FFT を実行する前に FFTW の関数からプランを作成し，そのプランのデータを実行関数に引き渡すことで FFT を実行する．FFTW のプラン作成関数には integer 型の flag という引数が存在し，これを変更することでプランの作成方法を変更することが可能である．以下にフラッグの一覧と簡単な説明を載せる．

フラッグ一覧

FFTW_ESTIMATE
アルゴリズムの比較を行わないが，最もヒューリスティックにプランを作成する．

FFTW_MEASURE
複数回の FFT を行い実行時間を計測することで最適なプランを作成する．デフォルトではこのフラッグが選択されている．

FFTW_PATIENT，FFTW_EXHAUSTIVE
FFTW_MEASURE より広範囲にわたってアルゴリズムの比較を行うため，より最適なプランを作成可能だが，同時にアルゴリズムの決定に時間を要する．

FFTW_WISDOM_ONLY
wisdom（A.6 参照）が存在するときにのみプランを作成させるためのフラッグである．そのため
・wisdom ファイルの中身もしくはファイル自体が無い場合
・フラッグを FFTW_PATIENT から FFTW_EXHAUSTIVE へ変更時
・実行するグリッドサイズの変更時
上記の時は null が返されるため，プラン作成を行うことができない．

アルゴリズムの選定フラッグの他に FFTW_UNALIGNED というフラッグも存在する．これは FFTW 用にアラインされていない配列でも利用可能にするためのフラッグであり，一度作成されたプランを別の配列でも再利用可能にするために使われる．ベクトル

演算を必要としないアルゴリズムを作成するので計算速度が遅くなるため，通常は必要としない．また配列の宣言に fftw_malloc や fftw_alloc_real を利用している場合も必要としない．

A.5 単精度と倍精度

FFTW では倍精度だけでなく単精度，四倍精度への対応が可能である．しかし，各精度を利用するための関数名が違うため注意が必要である．以下はその違いを示している．

FFTW での型対応の関数例

```
double 型
    call dfftw_execute (plan)
    call fftw_free (pr)
single 型
    call sfftw_execute (plan)
    call fftwf_free (pr)
long-double 型
    call lfftw_execute (plan)
    call fftwl_free (pr)
```

各実行関数の先頭に double 型では d を，single 型では s を，long-double 型では l をつける必要がある．また，FFTW を double 型以外で利用する場合には新たな FFTW をビルドする際に--enable-float，--enable-long-double をオプションで加え，出来上がったライブラリをリンクする必要がある．同様にコンパイル時はオプションとして-lfftw3f，-lfftw3l をそれぞれ加える必要がある．

A.6 wisdom の利用

FFTW では一度作成したプランの保存および復元が可能である．FFTW では，こうしたプランの保存と復元を行うメカニズムを wisdom と呼んでいる．FFTW の wisdom を活用するための関数を利用することで，ファイルへの保存およびファイルからの呼び出しが可能となっている．A.4 で説明したように，FFTW ではプランの作成方法をフラッグにより変更可能であるが，FFTW_PATIENT もしくは FFTW_EXHAUSTIVE を利用した場合は最適なプランが作成可能な一方で，アルゴリズムの決定に時間を要する．そこ

で，FFTW_PATIENT もしくは FFTW_EXHAUSTIVE を wisdom と一緒に利用した場合，初回のみアルゴリズムの決定に時間を要するが，2 回目以降はデータサイズが等しい場合は wisdom によって保存されたプランを使うことができるため，初期化にかかる時間を削減することができる．

A.7 FFTW のコンパイルオプションの順番

FFTW2 ライブラリを使用する場合，リンク用のコンパイルオプションの順番に注意が必要である．コンパイルのオプションは，`-lrfftw -lfftw` の順番で加えなければ正しくコンパイルが通らない．

A.8 MPI の利用と Fortran のバージョン

FFTW は MPI との互換性を持ち，MPI 用の FFTW の実行関数が存在する．しかし，MPI 用の FFTW3 は Fortran 2003 以降にのみ対応しており，かつその場合はメモリの割り当て方法とプラン作成の方法が異なる．

2 次元配列 (N_x, N_y) に対して FFT を行いたい場合，実空間を `[Ny, Nx]`，フーリエ空間を `[Ny/2+1, Nx]` と宣言するのに対し，Fortran 2003 用の関数に対しては MPI を利用する場合に実空間の配列を `[2*(Ny/2+1), Nx]`，フーリエ空間を `[Ny/2+1, Nx]` と宣言する必要がある．これは，MPI 利用時において FFT を実行する際にパッディングが必要となるためである．

MPI を利用した FFTW のプランを作成する場合は，C 言語にバインディングされているプラン作成関数を Fortran2003 の関数として使用するため，プラン作成時の配列の表記は

```
p2df = fftw_mpi_plan_dft_r2c_2d(nx, ny, in, C, comm_all%MPI_VAL, flag)
```

のように，`nx`，`ny` の順番で入力する．ただし，通常の FFTW を利用する場合は `ny`，`nx` の順番に変数を代入する．

A.9 FFTW のバージョン間の違い

FFTW2 と FFTW3 とでの最大の違いはプランを使用する配列の指定を行うタイミングである．FFTW2 では FFT 実行時に配列を指定するのに対し，FFTW3 ではプラン作成時に配列を指定する必要がある．

FFTW の関数呼び出し比較

---FFTW2---

プラン作成関数

```
  call rfftw2d_f77_create_plan(plan, ny, nx, dir, flag)
```

FFT **実行関数**

```
  call rfftwnd_f77_real_to_complex(plan, howmany, in, &
         istride, idist, out, ostride, odist)
```

---FFTW3---

プラン作成関数

```
  call dfftw_plan_many_dft_c2r)(plan, fft_dimension, &
         vector_sizes_real, howmany, &
         in, vector_sizes_complex, istride, idist, &
         out, vector_sizes_real, ostride, odist, flag)
```

FFT **実行関数**

```
  call dfftw_execute_dft_r2c(plan, in, out)
```

図 28: FFTW2 と FFTW3 の比較

また、FFTW は一回の関数呼び出しで複数回の FFT を実行することが可能である．それと同時に，FFT を実行する配列の変換する要素を指定することが可能となっている．この機能は FFTW2，FFTW3 ともに存在するが，バージョン間での仕様の違いにより，FFTW2 では要素の指定を FFT 実行時に可能だが，FFTW3 では実行時には要素の指定までは行うことができない．そのため，両バージョンを使うにはソースコード内で書き換える必要がある．要素数を指定する場合の FFTW2 と FFTW3 とでのプラン作成と実行関数呼び出しの違いを図 28 にまとめた．ただし、図 28 の howmany は一回の FFT 実行における回数を示し，dir は FFT か IFFT かの判定を行うための変数である．また istride/ostirde は，配列 in/out における FFT の配列内の要素間を示し，idist/odist は in/out における次の FFT の配列先頭番号までの距離を示す．

付録 B　OpenMP の記録

B.1　reduction 節の配列への利用

並列処理領域内のループ構文において共有変数にデータ依存性が生じた場合，データレースを防いで正しく出力するために `reduction` 節が存在する．この `reduction` 節は以下のように記述される [12].

`reduction（` **オペレータ** `：` **変数名リスト** `）`

オペレータには和 (+) や差 (-) などループ構文内のデータ依存性が生じる文の演算オペレータを入力し，**変数名リスト**にはデータ依存性が生じる変数を入力する．このとき，Fortran では変数名リストとして配列の利用が可能である [15]．本研究では AstroGK のマルチスレッド並列処理を施す際に，一部の配列に対して `reduction` 節を利用した．

B.2　OpenMP における並列処理速度比較

OpenMP の性質を調べるために，サンプルコードを作成していくつかの数値実験を行った．ローカルマシン（2.5.1 節参照）を利用し，特に説明がない限りスレッド数は 4 に指定している．

B.2.1　数値実験 1　サブルーチン内での変数の属性

OpenMP による並列処理領域より呼び出されたサブルーチン内の変数がどのような挙動を示すかを調べるための数値実験を行った．

サブルーチンに対する仮引数 (parameter) と実引数 (argument) を用意し，各スレッドに序数を代入するコードを作成した．仮引数にはメイン関数内で数値を代入し，実引数にはサブルーチン内で数値を代入した．サブルーチン内でそれぞれの値を出力することで各変数の値に変化が見られるか，正しく出力ができているかを調べた．

出力結果 1

まず両変数に対して PRIVATE 指示節による属性付与をしなかった場合の挙動を調べた，出力結果は以下の通りである．ただし，すべての変数にはそれぞれのスレッド番号 (ID) を代入しているため，PRVATE 属性であれば自分のスレッド番号が出力される．

出力結果 1

```
$ ./omp
# Use parameter:
My ID is  3. My ordinal number is 3rd
My ID is  1. My ordinal number is 3rd
My ID is  2. My ordinal number is 3rd
My ID is  0. My ordinal number is 3rd
# Use argument:
My ID is  2. My ordinal number is 2nd
My ID is  1. My ordinal number is 1st
My ID is  3. My ordinal number is 3rd
My ID is  0. My ordinal number is 0th
```

メイン関数内で数値を代入した仮引数 (parameter) のみ全スレッドが同じ数値を出力しており，SHARED 属性であることが分かる．一方，サブルーチン内で数値を代入した実引数 (argument) の出力結果は全スレッドが異なる数値を出力した．つまり，サブルーチンで宣言された変数は PRIVATE 属性となることが確認できた．

出力結果 2

PRIVATE 指示節を使用して仮引数を PRIVATE 属性にした．出力結果は以下の通りである．

出力結果 2

```
$ ./omp
# Use parameter:
My ID is  3. My ordinal number is 3rd
My ID is  2. My ordinal number is 2nd
My ID is  0. My ordinal number is 0th
My ID is  1. My ordinal number is 1st
# Use argument:
My ID is  2. My ordinal number is 2nd
My ID is  3. My ordinal number is 3rd
My ID is  0. My ordinal number is 0th
My ID is  1. My ordinal number is 1st
```

出力結果から，仮引数として渡す変数の値を各スレッドでそれぞれ指定および保持したい場合は，PRIVATE 指示節を利用する必要があることが分かった．

B.2.2 数値実験 2　do ループ構文内でのサブルーチン呼び出しと変数の保存

並列処理領域にある do ループ構文内で呼び出されたサブルーチンでの変数の挙動を調べた．integer 型の変数 count をサブルーチン内に用意し，do ループを用いてループが 1 週まわるたびにサブルーチン内で count の値が 1 ずつ加算されるプログラムコードを作成した．count が PRIVATE 属性であれば，各スレッドで自身が回したループの数を count の値として出力され（5 回のループを 5 スレッドで回した場合，すべてのスレッドが 1 を出力する），count が SHARED 属性であれば，do ループ文の回数が加算されながら出力される（5 回のループを 5 スレッドで回した場合，各スレッドは 1, 2,..., 5 のいずれかを出力する）状態となれば良い．現段階で，変数 count に対して属性の付与および初期化等は行っていない．サンプルコードの一部は以下のとおりである．

サンプルコード

```
program test2
  integer :: i
!$OMP PARALLEL
   ⋮
!$OMP DO
  do i=1, 5
    call num_count(i)
  end do
!$OMP END DO
!$OMP END PARALLEL

  subroutine num_count(i)
    integer, intent(in) :: i
    integer :: count

    count = count + 1
    write (*, '("My ID is ", I2, ". Count: ", I3, 3x, "i: ", I3)')
        omp_get_thread_num(), count, i
  end subroutine num_count
end program test2
```

出力結果 1

上記のサンプルコードを実行した結果，以下のとおりとなった．

出力結果 1（count 未定義時）

```
$ ./omp
# Parallel Region:
My ID is  2. count:   2   i:   4
My ID is  3. count:   2   i:   5
My ID is  1. count:   2   i:   3
My ID is  0. count:   2   i:   1
My ID is  0. count:   3   i:   2
```

count が未定義のため，初めから count =1 の状態となっている，また count は PRIVATE 属性になっており，0 番スレッドのみが do ループ文を繰り返しているために count の値が更新されて 3 として返している．そもそも未定義は良くない．

出力結果 2

次に並列処理領域内のサブルーチンにおいて count を宣言する際に

```
integer :: count=0
```

のように初期化を行い，サンプルコードを実行した．

出力結果 2（count 初期化時）

```
$ ./omp
# Parallel Region:
My ID is  1. count:   4   i:   3
My ID is  2. count:   4   i:   4
My ID is  3. count:   4   i:   5
My ID is  0. count:   4   i:   1
My ID is  0. count:   5   i:   2
-------
$ ./omp
# Parallel Region:
My ID is  1. count:   1   i:   3
My ID is  3. count:   4   i:   5
My ID is  2. count:   4   i:   4
My ID is  0. count:   4   i:   1
My ID is  0. count:   5   i:   2
```

count は初期化されたが，count が SHARED 属性になってしまいデータの競合が発生している事が分かった．これは変数 count に対して

```
integer, save :: count
```

のように save 属性を付けた時と同じ状態である．つまり，

1. 宣言時に初期化
2. save 属性を付与

するとサブルーチン内において変数は SHARED 属性になることが分かった．

出力結果 3

では初期化したいが PRIVATE 属性もつけるためにはどうするべきか．その場合は，2 通り方法がある．（他にも方法はあるかもしれないが，発見したのは主に 2 通り．）

PRIVATE 属性変数の初期化方法

1. THREADPRIVATE 指示文の利用
   THREADPRIVATE 指示文を以下のように宣言部にて利用する。[18]

```
  integer, save :: count
!$OMP THREADPRIVATE (count)
```

   THREADPRIVATE 指示文はスレッド間でのデータ環境を定義するための宣言文 [15] であり，これは count=0 と宣言した時にも同様に使用することができた．OpenMP による実行をせず，逐次実行のみの場合も対応できることを確認した．
2. logical 変数の利用
   logical 変数を新たに作成し、以下のように利用する。

```
  integer :: count
  logical :: FirstCall

  if (FirstCall) then
    count = 0
    FirstCall = .FALSE.
  endif
```

   count，FirstCall 共に未定義であるため PRIVATE 属性になっており，各スレッドが if 文内を一度だけ読み込むように設定した，未定義の場合，初め FirstCall は.TRUE. を持つ，こちらも逐次実行でも実行可能である．

出力結果は以下の通りである．

出力結果 3（count 初期化および PRIVATE 属性）

THREADPRIVATE 指示文を利用

```
$ ./omp
# Parallel Region:
My ID is  1. Count:   1   i:   3
My ID is  0. Count:   1   i:   1
My ID is  0. Count:   2   i:   2
My ID is  3. Count:   1   i:   5
My ID is  2. Count:   1   i:   4
```

logical 変数を利用

```
$ ./omp
# Parallel Region:
My ID is  3. Count:   1   i:   5
My ID is  0. Count: ***   i:   1
My ID is  0. Count: ***   i:   2
My ID is  1. Count:   1   i:   3
My ID is  2. Count:   1   i:   4
```

logical 変数を利用する方法の場合，変数 count が未定義であるためマスタースレッドにおいて適当な値を取ってしまうためうまく出力されていないと思われる，しかし count を初期化すると，SHARED 属性となってしまいうまく値が出力されない．

一方，THREADPRIVATE 指示文を利用する方法は，各スレッドの count の値がすべて初期化され，0 番スレッドのみが 2 回 do ループ文を回っていることも確認できた．

したがって，THREADPRIVATE 指示文を利用したほうが良いように思われる．

出力結果 4

しかし，THREADPRIVATE 指示文を利用するのが完璧ではない場合もある．上記の 2 通りの方法を用いたサブルーチン関数に対してまず逐次計算を行い，そのまま引き続き並列計算を行うサンプルコードを作成し実行した．両実行時において関数 num_count は do ループ文を用いてそれぞれ同じ回数が呼び出される．出力結果はそれぞれ以下の通りである．

出力結果 4-1（THREADPRIVATE 指示文を利用時）

```
$ ./omp
# Serial Region:
My ID is  0. Count:   1   i:   1
My ID is  0. Count:   2   i:   2
My ID is  0. Count:   3   i:   3
My ID is  0. Count:   4   i:   4
My ID is  0. Count:   5   i:   5
# Parallel Region:
My ID is  3. Count:   1   i:   5
My ID is  2. Count:   1   i:   4
My ID is  1. Count:   1   i:   3
My ID is  0. Count:   6   i:   1
My ID is  0. Count:   7   i:   2
```

各々のスレッドで count が save 属性をもつため，0 番スレッドのマスタースレッドのみが count をそのまま更新する状態となった．

出力結果 4-2（logical 変数を利用時）

```
$ ./omp
# Serial Region:
My ID is  0. Count:   1   i:   1
My ID is  0. Count:   2   i:   2
My ID is  0. Count:   3   i:   3
My ID is  0. Count:   4   i:   4
My ID is  0. Count:   5   i:   5
# Parallel Region:
My ID is  2. Count:   1   i:   4
My ID is  1. Count:   1   i:   3
My ID is  3. Count:   1   i:   5
My ID is  0. Count: ***   i:   1
My ID is  0. Count: ***   i:   2
```

一方でこちらは全スレッドで値が 0 に戻るが，出力結果 3 の時と同様に 0 番スレッドで

は値がうまく出力されなかった．

実験 2 のまとめ

サブルーチン内の変数を初期化し，かつ PRIVATE 属性にするには THREADPRIVATE 指示文を利用した方法が最も適していると思われるが，その場合も注意が必要である．逆に SHARED 属性を付けることも可能だが，その場合は値がしっかりと計算される環境をコーディングする必要がある．

B.2.3 実験 3 do ループ構文中のサブルーチン内での並列処理

実験 2 までで，並列処理領域内で呼び出されたサブルーチンで宣言された変数を初期化，PRIVATE 属性もしくは SHARED 属性にする方法を調べた．そこで次は，並列処理領域内の `do` 指示文で囲まれた `do` ループ構文中に呼び出されたサブルーチン内で演算処理を行った時の挙動を調べるためのサンプルコードを作成した．変数に仮引数 `prm` と実引数 `arg` を利用し，サブルーチン内でさらに `do` ループ構文を用いて加算演算を行った．サンプルコードの一部は以下の通り．

サンプルコード

```
program test3
  integer :: prm = 0
  integer :: i
!$OMP PARALLEL
!$OMP DO
  do i=1, 5
    call add (prm, i)
  end do
!$OMP END DO
!$OMP END PARALLEL
  write (*, '("# Serial Region:", /, 4x, "prm:" I6)') prm

  subroutine add (prm, i)
    integer, intent (inout) :: prm
    integer, intent (in) :: i
    integer, save :: arg
    integer :: j

    do j=1, 10
      prm = prm + j
      arg = arg + j
    end do
    write (*, '("My ID is ", I2, ". prm:", I4, &
        4x, "arg:", I4, 4x, "i: ", I3)') &
        omp_get_thread_num(), prm, arg, i
  end subroutine add
end program test3
```

サンプルコードでは，5 回繰り返される do ループ文の中に 10 回繰り返される do ループ文が存在し，SHARED 属性の変数に対して 1 から 10 までの加算を 5 回行うので 275 が値として最終的に出力される．逐次実行した場合，出力結果は以下の通り．

出力結果（逐次実行時）

```
$ ./omp
# Serial Region:
My ID is  0. prm:  55    arg:  55    i:   1
My ID is  0. prm: 110    arg: 110    i:   2
My ID is  0. prm: 165    arg: 165    i:   3
My ID is  0. prm: 220    arg: 220    i:   4
My ID is  0. prm: 275    arg: 275    i:   5
```

出力結果 1

OpenMP を利用してサンプルコードを並列処理実行した.

出力結果 1（並列実行時）

```
$ ./omp
# Parallel Region:
My ID is  0. prm: 107    arg: 105    i:   1
My ID is  0. prm: 162    arg: 160    i:   2
My ID is  3. prm: 162    arg: 160    i:   5
My ID is  1. prm: 162    arg: 160    i:   3
My ID is  2. prm: 162    arg: 160    i:   4
# Serial Region:
    prm:   162
-------
$ ./omp
# Parallel Region:
My ID is  1. prm: 186    arg: 210    i:   3
My ID is  0. prm: 186    arg: 210    i:   1
My ID is  0. prm: 241    arg: 265    i:   2
My ID is  2. prm: 241    arg: 265    i:   4
My ID is  3. prm: 241    arg: 265    i:   5
# Serial Region:
    prm:   241
```

prm，arg 共に SHARED 属性であるため，加算演算中に競合が発生し，最終的に正しい出力結果を得ることができなかった．

出力結果 2

SHARED 属性であっても出力結果として正しい値が出力されるようにしなければならない．そこで， critical 指示文を利用する．critical 指示文はあるスレッドが共有変数に更新作業を行っている間，他のスレッドにその共有変数の更新作業を行わせないよう，明示的に制御する指示文である [19]．この指示文では同時実行を制限する領域に識別子をつけ制御する．今回はサブルーチン関数 add にて

```
  do j=1, 10
!$OMP CRITICAL(c_prm)
    prm = prm + j
!$OMP END CRITICAL(c_prm)
!$OMP CRITICAL(c_arg)
    arg = arg + j
!$OMP END CRITICAL(c_arg)
  end do
```

のように書き加えた．同様の処理を行うための指示構文として，reduction 指示節，atomic 指示文が存在するが，reduction 指示節は do 指示文宣言時に使用しなければならないため使いづらい．また atomic 指示文は指示文の直後の一行だけを，同時実行を制限する対象とし，適用可能な命令は加減算もしくは Fortran の組み込み関数による共有変数の更新に限られるため制限が多い．

書き換えたサンプルコードの実行結果は以下のとおりである．

出力結果 2（CRITICAL 指示文利用時）

```
$ ./omp
# Parallel Region:
My ID is  1. prm: 220    arg: 220    i:   3
My ID is  3. prm: 220    arg: 220    i:   5
My ID is  0. prm: 220    arg: 220    i:   1
My ID is  0. prm: 275    arg: 275    i:   2
My ID is  2. prm: 275    arg: 275    i:   4
# Serial Region:
    prm:   275
```

prm，arg 共に SHARED 属性であるが，出力結果として正しい値が出力されることが分かる．つまり，サブルーチン内などで SHARED 属性の変数に対し競合が発生する可能性のある計算をしたい場合は，critical 指示文を利用すれば良い．ただし，critical 指示文は強制的な同期を取る必要があるため，低速化の原因となる可能性があるため使用には注意が必要である．

謝辞

本研究を行う際，いつも丁寧で親切なご指導をしていただきました指導教官である龍野智哉准教授に深く感謝を申し上げます．また Tretler Rudolf 研究員には様々なアドバイスを頂きましたことを心より感謝致します．さらにはメリーランド大学の William Dorland 教授には留学中に研究の指導から留学のサポートまでお世話になりました．心から感謝を申し上げます．最後になりますが，最後まで一緒に頑張ってきた研究員の同期の皆様に感謝致します．

参考文献

[1] “Sourceforge gyrokinetics.” https://gyrokinetics.svn.sourceforge.net/svnroot/gyrokinetics/, January 2016.

[2] R. Numata, G. G. Howes, T. Tatsuno, M. Barnes, and W. Dorland, “Astrophysical gyrokinetics code,” *Journal of Computational Physics*, vol. 229, pp. 9347–9372, 2010.

[3] “FFTW Homepage.” http://www.fftw.org, January 2016.

[4] 渡邉智彦, “核融合プラズマシミュレーションの技法-大規模並列計算環境の活用- 1. はじめに,” **プラズマ・核融合学会誌**, vol. 89, pp. 45–48, 2013.

[5] 坂上仁志, “核融合プラズマシミュレーションの技法-大規模並列計算環境の活用- 2. 並列コードを開発するための基本技法,” **プラズマ・核融合学会誌**, vol. 89, pp. 49–56, 2013.

[6] Strauss R. H, “Nonlinear, three-dimensional magnetohydrodynamics of noncircular tokamaks,” *The Physics of Fluids*, vol. 19, pp. 134–140, 1976.

[7] M. Barnes, W. Dorland, and T. Tatsuno, “Resolving velocity space dynamics in continuum gyrokinetics,” *Physics of Plasmas*, vol. 17, p. 032106, 2010.

[8] 吉田正廣, 松浦武信, 富山薫順, and 小島紀男, **フーリエ変換の計算法**. 現代工学社, 1994.

[9] “Intel Homepage.” http://www.intel.co.jp/content/www/jp/ja/homepage.html, January 2016.

[10] “Top 500 Supercomputer Sites.” http://www.top500.org/list/2012/11/, January 2016.

[11] M. Frigo and S. G. Johnson, “The design and implementation of FFTW3,” *Proceedings of the IEEE*, vol. 93, pp. 216–231, 2005.

[12] 牛島省, *OpenMP* **による並列プログラミングと数値計算法**. 丸善株式会社, 2006.

[13] 澤田淳二, 斎藤宏樹, and 勝崎俊樹, “gprof ゼミ.” http://mikilab.doshisha.ac.jp/dia/seminar/2002/pdf/gprof.pdf, October 2002.

[14] 渡邉智彦 and 井戸村泰宏, “核融合プラズマシミュレーションの技法-大規模並列計算環境の活用- 4. Vlasov シミュレーションのコーディング技法,” **プラズマ・核融合学会誌**, vol. 89, pp. 171–179, 2013.

[15] *OpenMP Application Program Interface*, version 4.0 ed., July 2013.

[16] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, version 3.1 ed., June 2015.

[17] “slurm Homepage.” http://slurm.schedmd.com, January 2016.

[18] 岩下武史, “OpenMP 基礎.” http://ais.sys.i.kyoto-u.ac.jp/~iwashita/OpenMP-1.pdf, November 2014.

[19] 南里豪志, “OpenMP 入門 (3).” https://www.cc.kyushu-u.ac.jp/scp/system/library/OpenMP/openmp0209.pdf, November 2014.