J. Plasma Fusion Res. Vol.89, No.2 (2013) 119-127

講座 核融合プラズマシミュレーションの技法ー大規模並列計算環境の活用ー

# 3．MHD シミュレーションのコーディング技法

三浦英昭，藤堂　泰，後藤俊幸[1)]
核融合科学研究所，[1)]名古屋工業大学

（原稿受付：2012年1月6日）

この章では，MHD 方程式のシミュレーションを行うにあたっての並列化のポイントについて簡単に紹介する．取り上げる話題は，差分法によるコードの並列化と，3次元高速フーリエ変換の高速化である．

**Keywords:**
MHD, finite difference, FFT, parallelization

## 3.1　はじめに

MHD シミュレーションの一番簡単な形は，速度や電場などの物理量を空間上の離散点（格子点）の集合として表現し，空間微分をなんらかの形で近似，これを時間方向に積分することで数値的な近似解を得るものである．このため，数値シミュレーションコードの作成の際には，如何に微分を近似するか，その近似を効率よく行うためにデータをどのように主記憶に配置するかが重要となる．最も古典的な（しかし今でも頻繁に使われる）数値解法として中心差分法がある．最も粗い近似の中心差分法である2次精度中心差分法を例にとると，ある物理量 $f(x)$ の空間微分を空間上の格子点 $x_i=(L/N)i=\Delta i$，$i=0,1,2,\cdots$（$L$，$N$ はそれぞれ空間の全長と，格子点の総数）で

$$\left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_i=\frac{f_{i+1}-f_{i-1}}{2\Delta}+O(\Delta^2)$$

と近似する．このような差分法の特徴は，考えている格子点 $x_i$ とその隣接格子 $x_{i\pm1}$ の情報があれば微分が近似できる点である．（より高次の差分法では必要とする格子点数が増加する．）このため，格子点データが複数の計算ノードにまたがっていても，2.2節で紹介した MPI ライブラリによるノード間通信を行って比較的簡単に計算が可能である．このような場合についての並列化を3.2節で説明する．

中心差分法が格子点 $x_i$ の周辺の情報だけを利用するのに対して，空間全体の情報を利用する手法もある．その代表的な例が，フーリエ変換によるスペクトル法である．スペクトル法では，物理量 $f(x)$ の空間微分をそのフーリエ係数 $\hat{f}_k$ を用いて

$$\left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_i=\sum_{k=-N/2}^{N/2}ik\hat{f}_k e^{ikx_i}$$

と近似する．この手法は空間に対する周期性を利用する，空間3次元が周期的な一様乱流，一様 MHD 乱流の直接数値シミュレーションなどに用いられる．（直接数値シミュレーションとは，支配方程式を人工粘性などに頼らず極力方程式を誠実に解く場合に用いられる用語であり，必然的に数値シミュレーション手法には高精度・高解像度であることが要請される．）また，トーラスプラズマのトロイダル方向およびポロイダル方向もフーリエ変換可能であるため，動半径方向は差分法で，トロイダル・ポロイダル方向はスペクトル法を使うこともある．ここで重要な点は，フーリエ変換の演算には格子線に沿ったすべての格子点上のデータを使わざるを得ないということである．フーリエ変換を行う方向のデータが複数の計算ノードに分散して置かれている場合にはノード間通信が発生する．3.3節では，このようなノード間通信の発生を念頭に，3次元高速フーリエ変換（Fast Fourier Transform，FFT）における並列化の方法について述べる．また，近年は優れた FFT ライブラリーのソースコードが公開されていることから，これについても同節で触れる．

## 3.2　MPI 並列化のための領域分割

3.1で述べたように，計算法として差分法を用いて，MPI 通信を伴う並列化 MHD シミュレーションを行うことを考える．MPI プロセスと分割領域は一対一に対応し，各 MPI プロセスが一つの分割領域について同一のプログラムを実行するものとする．このとき境界値計算のためのデータ交換等のため他のプロセスとの通信が必要となるが，差分法で参照されるデータは隣接する格子点に限定されているので，プロセス間の通信は基本的に1対1通信である．並列化によって短縮される計算時間と比べて通信時間が十分に小さければ，プログラム全体の実行時間を短縮することができる．（このような差分法のプログラムでは，プログラム作成者やコンピュータのアーキテクチャーが異なっても領域分割の様式は比較的似通った形になりやすい．これは，プログラム作成者の考えやコンピュータのアーキテク

*3. Coding Techniques of MHD Simulations*

*MIURA Hideaki, TODO Yasushi and GOTOH Toshiyuki*　　*corresponding author's e-mail: miura.hideaki@nifs.ac.jp*

チャー次第でデータの配置が大きく変わり得る3次元FFT（次節）と対照的である.）

ここではプログラム言語としてFortranを使用する．モジュールmpiを使用し（use mpi），その中で定義されていない変数（以下の例ではmpi_err，nprocess，my_rankなど）は特に断らない限り整数型（integer）で宣言するものとする．MPI並列計算では，まず最初に

```
call mpi_init (mpi_err)
```

を実行し，計算の最後には

```
call mpi_finalize (mpi_err)
```

を実行しなければならない．

次に，MPI並列計算では全プロセスが同一のプログラムを実行するので，各プロセスは自分が何番目のプロセス（ランク，以下の例ではmy_rank）であるかをプログラム側で認識する必要がある．ランクと計算全体のプロセス数（以下の例ではnprocess）は，以下で与えられる．

```
call mpi_comm_rank(mpi_comm_world,
my_rank,mpi_err)
call mpi_comm_size(mpi_comm_world,
nprocess,mpi_err)
```

ここで，my_rankがとりうる値の範囲は0からnprocess-1である．プログラム側ではコミュニケータmpi_comm_worldを定義する必要はない．コミュニケータは各通信に関係するプロセスを指定しており，mpi_comm_worldはこの通信にランク0からnprocess-1のすべてのプロセスが関係することを意味している．コミュニケータとしてmpi_comm_worldを使用すればあらゆる通信を行うことができるが，通信の効率を考えるとプログラム側でコミュニケータを定義して対象範囲を必要なプロセスに限定する方がよい．その例については後述する．

さて，ここでは2次元座標系$(x,y)$の問題を考えよう．全体でそれぞれlx_global，ly_global個の格子点を使用し，それぞれをnprocess_x，nprocess_y個の領域に均等に分割するものとする．一つの領域が一つのプロセスに対応するので，nprocess=nprocess_x*nprocess_yの関係がある．各プロセスが分担する分割領域の格子点数(lx,ly)はlx=lx_global/nprocess_x，ly=ly_global/nprocess_yである．差分法では近隣の格子点の情報を参照して差分計算を行う．そのため分割領域の端点では差分計算に参照する格子点が近隣プロセスに存在するので，差分計算に必要なデータをプロセス間の通信により交換しなければならない．このため，配列を正味の格子点だけでなく，差分計算に必要な近隣プロセス上の格子点も含めて定義しておくとよい．このように差分計算のために拡張した領域は袖領域またはのりしろ領域と呼ばれる．両側m点ずつ参照して差分計算を行うものとすると，各プロセスの格子点数は袖領域を含めて(lx+2*m,ly+2*m)となる．

ここで計算量と通信量の関係を整理しておこう．各プロセスが担当する計算量は，正味の格子点数lx*ly=(lx_global*ly_global)/nprocessに比例するので，全体のプロセス数nprocessに反比例して減少する．一方で通信量は，上下左右のプロセスとデータを交換するので，2*m*(lx+ly)=2*m*(lx_global/nprocess_x + ly_global/nprocess_y)に比例する．通信量はnprocess_xとnprocess_yの積であるnprocessではなく，nprocess_xまたはnprocess_yに反比例して減少するのである．通信量と計算量の比，(通信量)/(計算量)は2*m*(1/lx + 1/ly)に比例することになる．実際の計算における通信時間と計算時間の比は，通信量と計算量の比に加えて，計算機の演算性能および通信性能にも依存するが，lxとlyが十分に大きければ通信時間は無視できて，全体の計算時間（＝計算時間＋通信時間）をnprocessに反比例して短縮することが可能となる．このため，差分法は大規模並列計算に適しているといえる．領域分割数が増大してlxまたはlyが小さくなると通信時間が計算時間と比較して無視できなくなり，プロセス数を増やしても期待されるほど全体の計算時間は短縮されなくなる．また，nprocess一定の条件下で(通信量)/(計算量)を極小化するには(同時にlx*ly=lx_global*ly_global/nprocessも一定であることに注意)，1/lx + 1/ly=(lx + ly)/(lx*ly)であるから，lx+lyを極小化すればよいことがわかる．2次元の問題ではlx=lyとなる正方形に近い分割領域が高速計算には有利であるといえる．

次に考慮しなければならないのは，分割領域格子と全体格子の対応である．まずは各分割領域がx，y方向に何番目かをmy_rank_x，my_rank_yで定義しよう(0≤my_rank_x≤nprocess_x-1，0≤my_rank_y≤nprocess_y-1)．ここでは分割領域をx方向に左から順番に並べ，右端に達すると次はy方向に一つ移動して左から順番に並べるものとする．このときmy_rank = my_rank_x + my_rank_y*nprocess_xの関係が成り立ち，Fortranプログラムではmy_rank_x，my_rank_yは以下のように計算できる．

```
my_rank_y = my_rank/nprocess_x
my_rank_x = my_rank – my_rank_y*nprocess_x
```

x方向の境界条件は周期境界としてみよう．このとき物理的な境界条件として，全体格子上の配列GがG(1,:)=G(lx_global+1,:)を満たし（全体格子上に実際に配列を定義する必要はない），各分割領域のx方向にi番目の格子点が(1 ≤ i ≤ lx+2*m)全体ではigloba l番目の格子点であるとする．my_rank_x=0の分割領域のx方向にm+1番目の格子点が全体では1番目の格子点に対応する場合，両者の対応はiglobal=i+my_rank_x*lx-mで与えられる．例えば，my_rank_x=0の分割領域ではiglobal=i-mとなる．

次にy方向には周期境界でなく，全体格子上のjglobal=1とjglobal=ly_globalの物理量はあらかじめ与えられるものとしよう．y方向についてもx方向と同様に分割領域格子と全体格子を対応させるものとすると，それぞれの格子番号jとjglobalはjglobal=j+my_

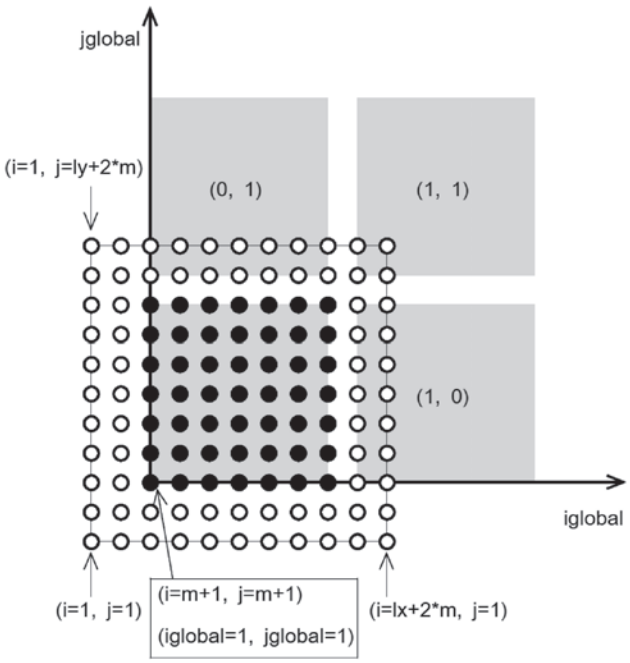



図 1　分割領域（my_rank_x=0, my_rank_y=0）の格子（i, j）と全体格子（iglobal, jglobal）の対応．黒丸と白丸は内部領域と袖領域の格子点をそれぞれ示している．灰色で塗られた部分は各分割領域の内部領域に対応し，（ ）内の数字は（my_rank_x, my_rank_y）を示している．

rank_y*ly-m で関係づけられる．分割領域格子と全体格子の対応を**図 1** に示す．m が 2 以上の場合には，jglobal=2,…,m および jglobal=ly_global-m+1,…,ly_global-1 の格子点では境界に近い側で m 個の格子点を参照することができず，他の格子点とは異なる差分法を適用することになる．これらの全体格子点に対応する格子点を含む my_rank_y=0 および nprocess_y-1 のプロセスでは，境界付近の専用差分法を実行する必要がある．

分割領域のうち袖領域を除いた正味の格子の x 方向の最初と最後の点をそれぞれ lxstart，lxend，y 方向にはそれぞれ lystart と lyend とする．ここまでに述べた分割領域格子と全体格子の対応を用いると，lxstart=m+1，lxend=lx+m，lystart=m+1，lyend=ly+m である．x 方向に隣接する領域間（my_rank_x および my_rank_x+1）のデータ交換は**図 2** のように行われる．y 方向についても同様である．

ここまでに説明した分割領域格子と全体格子の対応では，my_rank_y=0 および nprocess_y-1 のプロセスにおいて，境界付近の専用差分法を実行する必要があることをすでに述べた．ランクによる計算の分岐があるとプログラムが複雑になり，バグが発生しやすくなる．MHD シミュレーションでは差分計算を頻繁に行うので，差分計算はランクに依存した計算の分岐を含まないことが望ましい．そこで，各プロセスで j=2,…,m および j=ly-m+1,…,ly-1 の格子点では共通の専用差分法を適用することを考える．この場合には，my_rank_y=0 の領域において全体格子と分割領域格子の下端が一致し，my_rank_y=nprocess_y-1 の領域では上端が一致しなければならない．具体的には，(1) my_rank_y=0 の領域では，jglobal=j，lystart=1，lyend=ly，(2) my_rank_y=nprocess_y-1 の領域では，jglobal=j+my_rank_y*ly-2*m，lystart=2*m+1，lyend=ly+2*m，(3) これら以外の領域では，jglobal=j+my_rank_y*ly-m，lystart=m+1，lyend=ly+m とそれぞれ設定される．この設定は，全体格子上で開発された非 MPI 並列差分計算プログラムを MPI 並列化する際にも役立つ．

**図 2** に示した袖領域の通信では，ユーザーが新たに定義するコミュニケータを使用して通信の対象を限定するとよい．ここでは mpi_comm_split を用いてコミュニケータを定義する．x 方向の通信では my_rank_y の値が等しいプロセス間でデータを交換するので，コミュニケータ mpi_comm_x を以下のように定義する．

```
icolor = my_rank_y
ikey = my_rank_x
call  mpi_comm_split(mpi_comm_world,
icolor,ikey,mpi_comm_x,mpi_err)
```

ここで，icolor はこの値が同じプロセス間で通信することを意味し，この通信グループ内のランクは ikey の順番に 0 から定められる．y 方向通信のコミュニケータ mpi_comm_y は，icolor = my_rank_x，ikey = my_rank_y として同様に定義される．このように新しく定義したコミュニケータを**図 3** に示す．

次に袖領域の通信について説明する．

```
real(8)::aaa(lx+2*m,ly+2*m)
```

と宣言されている配列 aaa の通信を考える．y 方向の通信は，対象となるデータがメモリ上に連続に格納されているので簡単である．この場合は上方と下方の隣接プロセスとの通信が発生する．my_rank_y=0 または my_rank_y=nprocess_y-1 の場合は，それぞれ下方または上方の隣接プロセスとの通信が発生しないので，通信先として mpi_proc_null を指定して通信を抑制する．非同期型通信 mpi_isend と mpi_irecv を使うとデッドロックの心配がない．

```
node_up = my_rank_y + 1
node_down = my_rank_y - 1
if(my_rank_y.eq.0) node_down =
mpi_proc_null
if(my_rank_y.eq.nprocess_y-1)  node_up  =
```

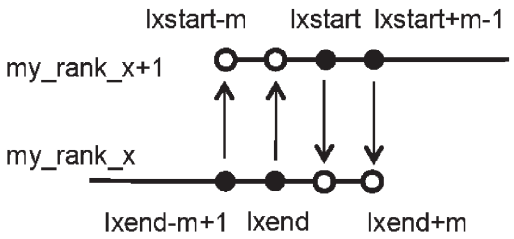



図 2　x 方向に隣接する領域間（my_rank_x および my_rank_x+1）のデータ交換．黒丸と白丸は内部領域と袖領域，矢印の向きはデータの通信方向をそれぞれ示している．

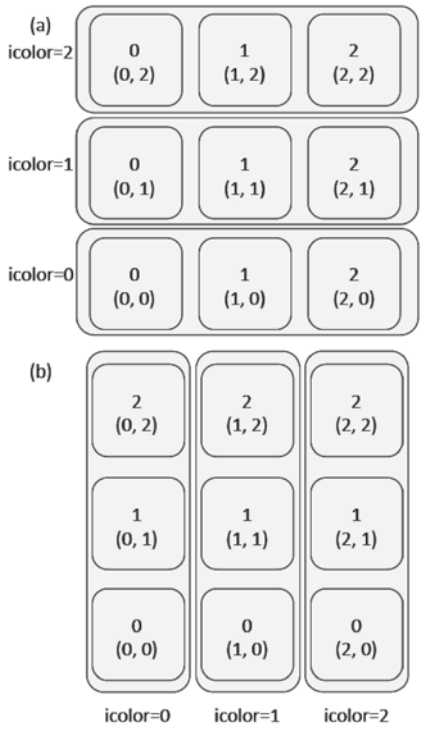



図3 新しく定義したコミュニケータ(a)mpi_comm_xと(b)mpi_comm_y．正方形と長方形はMPIプロセスと通信グループをそれぞれ表している．各MPIプロセスの中心の数字は新コミュニケータでのランク，( )内の数字はmpi_comm_worldにおける(my_rank_x, my_rank_y)を示している．

```
mpi_proc_null

jsize=(lx+2*m)*m
call  mpi_isend(aaa(1,lyend-m+1),jsize,
                mpi_double_precision,&
                node_up,1,mpi_comm_y,
                jsend1,mpi_err)
call mpi_isend(aaa(1,lystart),jsize,
                mpi_double_precision,&
                node_down,1,mpi_comm_y,
                jsend2,mpi_err)
call mpi_irecv(aaa(1,lyend+1),jsize,
                mpi_double_precision,&
                node_up,1,mpi_comm_y,
                jrecv1,mpi_err)
call  mpi_irecv(aaa(1,lystart-m),jsize,
                mpi_double_precision,&
                node_down,1,mpi_comm_y,
                jrec2,mpi_err)
call mpi_wait(jsend1,my_status,mpi_err)
call mpi_wait(jsend2,my_status,mpi_err)
call mpi_wait(jrecv1,my_status,mpi_err)
call mpi_wait(jrecv2,my_status,mpi_err)
```

x方向の通信では参照されるデータがメモリ上に連続に格納されていないため，バッファ配列を使用する．以下のバッファ配列を宣言しておく．

```
real(8)::up_send_x(m,ly+2*m),up_recv_x
(m,ly+2*m)
real(8)::down_send_x(m,ly+2*m),down_recv
_x(m,ly+2*m)
```

隣接プロセスのランクは，周期境界条件を考慮して

```
node_up=mod(my_rank_x+1,nprocess_x)
node_down=mod(my_rank_x-1+nprocess_x,
nprocess_x)
```

と定義される．次に送信用バッファ配列にデータを格納する．

```
do j=1,ly+2*m
  up_send_x(1:m,j)=aaa(lxend-m+1:lxend,
  j)
  down_send_x(1:m,j)=aaa(lstart:lxstart
  +m-1,j)
end do
```

通信は以下のようになる．

```
isize=(ly+2*m)*m
call mpi_isend(up_send_x(1,1),isize,
                mpi_double_precision,&
                node_up,1,mpi_comm_x,
                isend1,mpi_err)
call  mpi_isend(down_send_x(1,1),isize,
                mpi_double_precision,&
                node_down,1,mpi_comm_x,
                isend2,mpi_err)
call mpi_irecv(up_recv_x(1,1),isize,
                mpi_double_precision,&
                node_up,1,mpi_comm_x,
                irecv1,mpi_err)
call  mpi_irecv(down_recv_x(1,1),isize,
                mpi_double_precision,&
                node_down,1,mpi_comm_x,
                irecv2,mpi_err)
call mpi_wait(isend1,my_status,mpi_err)
call mpi_wait(isend2,my_status,mpi_err)
call mpi_wait(irecv1,my_status,mpi_err)
call mpi_wait(irecv2,my_status,mpi_err)
```

最後に，バッファ配列に受け取ったデータを元の配列に格納する．

```
do j=1,ly+2*m
  aaa(lxend+1:lxend+m,j)=up_recv_x(1:m,
  j)
  aaa(lxstart-m:lxstart-1,j)= down_recv_
  x(1:m,j)
end do
```

MPI通信では毎回オーバーヘッドが発生するので，複数

の配列について通信を行う場合は，以下で宣言されるようなバッファ配列を用いて通信を1回にまとめるとよい．ここでnarrayは通信する配列の個数である．

```
real(8)::up_send_xm(m,ly+2*m,narray),up_
recv_xm(m,ly+2*m,narray)
real(8)::down_send_xm(m,ly+2*m,narray),
down_recv_xm(m,ly+2*m,narray)
```

具体的な手順は配列1個の場合と同様なので割愛する．

### 3.3 3次元FFT

次に，スペクトル法で用いられる3次元FFTの並列化について述べる．多くの計算機センターのスパコンにはすでにチューニングされたFFTが備わっている．並列計算機向けのFFTとして有名なものにはFFTW[1]とそれをもとにしたP3DFFT[2]があり（これらについては最後に触れることにする），いまさら私家版の3次元FFTを開発するというのはいささか世間の流れに抗っている感がないわけではない．しかし，FFTではそのデータ配置こそが高速化の観点からも使い勝手の観点からも最も重要なポイントの一つである．レディーメイドの服はファッショナブルで安価ですぐ手に入るが，個人の体型やサイズに必ずしもフィットしているわけではなく我慢しなければならないこともある．一方，オーダーメードの服は高価だが，体型にフィットし使い勝手がよくそれだけに長い期間愛用することができる（著者は誂えた服を着たことはないが）．同じことが，各人の必要とする3次元FFTや他の基本ルーチンにもあてはまると思う．

ここで紹介する3次元FFTは，乱流およびMHD乱流のスペクトル法による直接数値シミュレーションに向けて開発されたものである．高レイノルズ数での乱流では高い空間解像度が要求されるので，1方向あたりの格子点数Nが$2^{10}$以上の3次元FFTを念頭に置いている．また，乱流理論との関連性やスペクトル法では波数空間で方程式を取り扱うことが多いので，以下では波数空間を中心にした解説を行う．プログラムはFortranで書かれているものと想定する．後述の公開されたFFTライブラリーはC言語などで書かれているものが多いが，多くの場合はFortranプログラムからサブルーチンをコールできるように配慮されている．

関数$f(x_1,x_2,x_3)$のデータは3.2と同様に各ノードに分散して配置されているものとする．ただし，ここでは2次元領域分割を考えるので，一つの次元については全データが一つのノード内で共有されている．FFTでは座標線方向にすべてのデータを使って計算するため，データが複数のノードにまたがっている場合にはMPIによる通信を行う必要がある（MPI通信の初期化等については3.2と同様なのでそちらを参考されたい．）．FFTの煩瑣なバタフライ演算を複数のノードにまたがったデータに対して行うのは相当に大変であるため，次のような手順を考える．最初に，全区間データが1つのノードに格納されている座標方向についてFFTによるフーリエ変換を行う．次に他の方向のフーリエ変換を行う際にはデータの転置を行い，一旦全区間データをすべて同一ノードに格納してからFFTを実行する．このため，3.2の差分法とは異なり，おおがかりなMPI通信が複数回発生する．この結果，大規模分散並列計算では時として，全計算時間の50%以上が3次元FFTの通信時間という事態が発生し得る．以下では，このような実態を踏まえて転置の部分を主眼に解説する．誌面の都合上，バタフライ演算などFFTの細部については割愛する．適宜参考書や論文を参考にされたい．

多くの複素表現のFFTでは関数値$f_i = f(x_i)$，$i = 1, ..., N$に対して変換，逆変換を返すように作られているので，波数空間での$ik$をかけるという数学演算は単純ではなく`K(i) = (i - 1) - [(i - 1)/(N/2)]*N`のように修正された波数をかける操作になる．ここでは数学的表現とプログラム上での表現が同等になるように複素フーリエ変換を構成する．実空間の離散点$(2\pi/N)(x_1,x_2,x_3)$での関数値を$f(x_1,x_2,x_3)$，フーリエ係数を$\hat{f}(k_1,k_2,k_3)$（$x_i$，$k_i$は整数）とすると3次元フーリエ級数を以下のように定義する．

$$f(i,j,k) = \sum_{k_1=-N/2}^{N/2-1} \sum_{k_1=-N/2}^{N/2-1} \sum_{k_1=-N/2}^{N/2-1} \hat{f}(k_1,k_2,k_3) \times \exp\left( i \frac{2\pi}{N} (k_1x_1 + k_2x_2 + k_3x_3) \right)$$

実数条件より$\hat{f}(k_1,k_2,k_3) = \hat{f}^*(-k_1,-k_2,-k_3)$であるから，メモリ消費を抑えるためにこの特性を用いて波数空間の半分の領域のみを用いる（**図4**左図）．例えば，$k_3 \geq 0$の領域のみで$\hat{f}(k_1,k_2,k_3)$を定義する（$k_3 = 0$では実数条件を課した冗長な配列を用いる）．一方，物理空間での$-N/2 \leq x_3 \leq -1$における実数関数配列$g(i,j,k)$

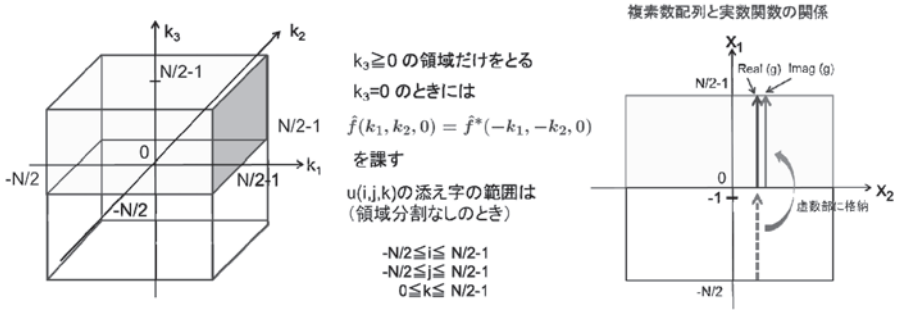



図4　3次元FFTにおける変数の格納の仕方．

は，複素数宣言された配列$\hat{f}(i,j,k)$に対して

$$\hat{f}(i,j,k)=g(i,j,k)+\sqrt{-1}g(i,j,k-\frac{N}{2}) \quad (0\le k\le N/2-1)$$

という具合に，$g(i,j,k)$のうち$k$が負の領域をそのまま$N/2$だけ第3変数の正の方向に平行移動して，$\hat{f}(i,j,k)$の虚数部に格納する(**図4**右図)．3番目の変数についての変換は後で述べる転置を行って第2変数の位置に移動させることにより，長さ$N$の領域を確保したうえで$z<0$の領域に展開して変換を行うようにする．以下，順にみていく．

1．関数は倍精度複素変数とする．配列サイズが大きいので領域を分割する．大事なのは，フーリエ変換（三角関数と変数との積と和によるバタフライ演算が主要部）を行う変数の方向にはプロセス間通信を入れないようにすることである．第1変数については疑似ベクトル処理機能を有効に働かせるようデータの連続アクセスをはかり，キャッシュの大きさに応じてベクトル長を最適なものにするよう分割数Npxを決める．第3変数についてはNpz個のプロセス並列に加えてスレッド並列を行う．格子点数が少ない場合には第1変数の大きさをNにとってフーリエ変換を行うことも可能である．しかし，64kBのキャッシュでは格子点サイズが1024を超えるとキャッシュに入りきらなくなるので，第2変数の方向に$N$個の格子点をとってFFTを行うようにする．したがって，第1，第3変数について2次元分割を行う(**図5**)．各変数の動く範囲は，0≦i≦N/Npx-1，-N/2≦j≦N/2-1，0≦k≦N/Npz-1となる．直感的には，第1，3変数の組で構成される(N/Npx)×(N/Npz)の面について第2変数方向にFFTをかけるというイメージになる．分割にあたっては，2種類のコミュニケータを必要とするので，mpi_comm_splitを用いて，

```
!!!--- Split MPI Rank (Type A)
  color = mod(rank,Npx) ; key = rank/Npx
  call MPI_comm_free(Ncomm,err)
  call  MPI_comm_split(comm,color,key,
  Ncomm,err)
  call MPI_comm_rank(Ncomm,Nrank,err)
  call MPI_comm_size(Ncomm,Nnpe ,err)
```

のように行う．変換の基数が4で$N=4^n$の時には，FFTは以下のようになる．

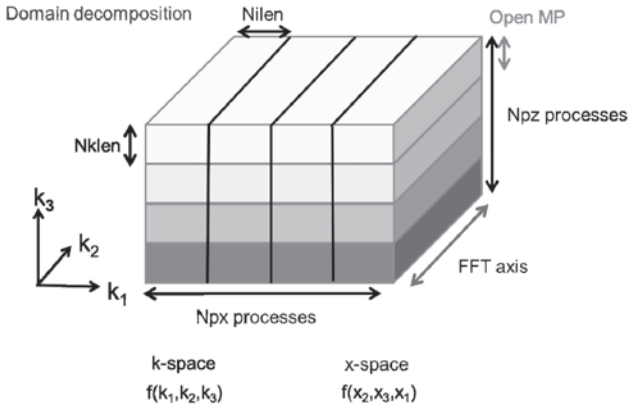



図5　3次元FFTにおける空間分割．

```
!$OMP PARALLEL DO private(i,j,k,....)
                          <-- k  軸方向スレッド並列
  do k = 0,(N/2)/Npz-1
    do j=1,n      <-- j 軸方向のフーリエ変換における n 段のバタフライ演算ループ(N = 4^n に注意)
    j1=..; j2=..; j3=..; j4=..
                  <-- バタフライ演算で用いられる4個の点の配列インデックス
    trig2=..; trig3=..; trig4=.. <-- 三角関数
    do i=0,(N/Npx)-1    <-- i 軸方向連続アクセス
      w1 = f(i,j1,k)+f(i,j3,k)
      w2 = f(i,j2,k)+f(i,j4,k)
      w3 = f(i,j1,k)-f(i,j3,k)
      w4 = f(i,j2,k)-f(i,j4,k)
      f(i,j1,k) = w1+ w2
      f(i,j2,k) = (w3-ai*w4)*trig2
                        <--ai=(0.0,1.0)虚数単位
      f(i,j3,k) = (w1- w2)*trig3
      f(i,j4,k) = (w3+ai*w4)*trig4
    end do; end do
  end do
!$OMP end parallel do
```

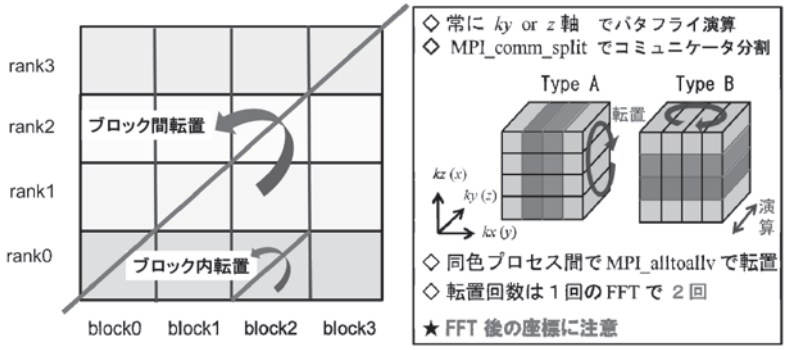



図6　3次元FFTにおけるブロック間転置とブロック内転置(左図)，およびバタフライ演算の方向と2回の転置(右図)．

キャッシュを有効に使うため，同じ配列 f(i,j1,k) についてロードとストアを行うようにする．このため 4，5 行目の配列変数と回転因子(三角関数表)をうまくならべる必要があるが，この詳細は 1 次元 FFT の解説書にゆだねる．

2．第 1，3 変数についての変換には，第 1，3 変数を第 2 変数の位置に持ってくるように転置が必要になる．転置はブロック転置とブロック内転置からなり，通信が発生するのはブロック転置においてである．転送するデータの塊を大きくかつ変数が連続になるようにして，できるだけ MPI 通信の回数を減らすことが望ましい．そのために，例えば，2 次元配列に詰め替えた後(packing)，全対全(MPI_All_to_Allv)通信を用いてデータを交換する(**図 6**)．ブロック転置が済んだら，2 次元配列をまた 3 次元配列に戻す（unpacking）と同時にブロック内転置を行う．プログラムは以下のようになる．

```
!!!--- pack                          ←---- 詰め替え
      Nklen1=(N/2)/Npz-1;  Nilen=N/Npx;
      Nilen1=Nilen-1
      btsize2 =Nilen*Nilen*Nklen
                          ←---- 1ブロックあたりの
                                転送データサイズ
      do i=0,Npx-1
         msize2(i) = btsize2
          disp2(i) = i*(btsize2+1)
      end do

      allocate( b2(0:btsize2,0:Npx-1) )
      do iz=0,Nklen1
         do ky=0,Npx-1; kyg=-N/2+ky*Nilen
          do iy=0,Nilen1; iyp=kyg+iy
              do  ix=0,Nilen1;  nl=Nilen*
              Nilen*iz+Nilen*iy+ix
                 b2(nl,ky)=c(ix,iyp,iz)
                   ←---- 2次元配列にまとめる
      end do; end do; end do ; end do
      deallocate(c)
      allocate( c2(0:btsize2,0:Npx-1) )

!!!--- Block Transpose in x and y Directions
                   ←---- ブロック転置　全対全通信
      call  MPI_alltoallv(b2(0,0),msize2,
                          disp2,MPI_com-
                          plex16,&
                          c2(0,0),msize2,
                          disp2,MPI_com-
                          plex16,&
                          Ncomm,err)
      deallocate(b2)
      allocate(c(0:Nilen,k1:k2,0:Nklen1))

!!!--- Unpack                        ←---- 荷ほどき
      do iz=0,Nklen1
       do kx=0,Npx-1; kxg = k1+kx*Nilen
         do iy=0,Nilen1
          do ix=0,Nilen1;
            ixp=kxg+ix; nl=Nilen*Nilen*iz
            +Nilen*iy+ix
            c(iy,ixp,iz)=c2(nl,kx)
                          ←---- 局所転置も行う
      end do; end do; end do ; end do
      deallocate(c2)
```

データを密にパックして送り出すやり方でかなり計算時間が変わってくる．上のプログラム例では，わかりやすくするために 2 次元配列を用いたが，mpi_type_vector などを使って新しい変数を定義してパックせず直接送る仕方もある．パッキングをするかしないかは送るデータのサイズにも依存する．もしパッキングするのであれば，バタフライ演算（3 行目）の最終段（j=n）の時に，ストアと同時にパックしてしまう方法もあるが，ストアのコストも考慮する必要がある．

3．配列の転置にかかる全対全の通信に多くの時間がかかるので，転置の回数は最小限（2 回）にとどめる．**図 7** に示した例では，波数空間では $(k_1, k_2, k_3)$ の順に並んでいたものが，実空間では $(x_2, x_3, x_1)$ の順に並ぶので注意が必要

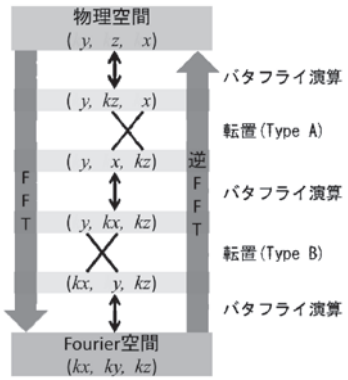



図 7　3 次元 FFT における 2 回の転置による独立変数の順序．

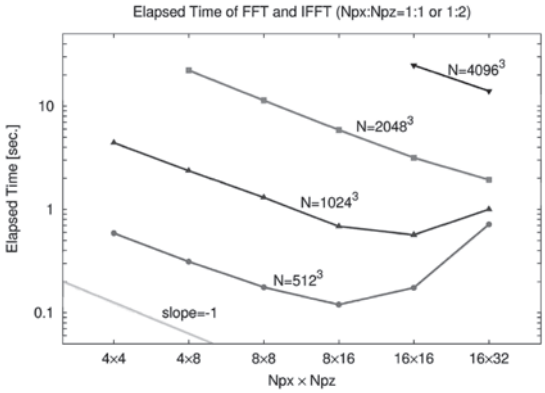



図 8　3 次元 FFT のスケーラビリティ．

である．順序は転置の順番をどうとるかによって変化するので，都合の良いように入れ替えればよい．たとえば，実空間での方程式を扱うために$(x_1, x_2, x_3)$の並びとしたければ，波数空間で$(k_3, k_1, k_2)$の並びになるように入れ替えればよい．

4．レイテンシーの問題を回避しメモリアクセス回数を逓減するためFFTの基数を大きいものにとる．上のプログラム例では基数を4とした．キャッシュサイズをみながら，領域分割数（Npx）を多めにとり基数8のFFTを導入することも考えられよう．

2次元分割における分割パターン（Npx，Npz）によってもFFTの性能に違いが出てくる．**図8**は，Fujitsu FX 1において計算時間がプロセス数とともにどれほど減少するかを各格子点数についてみたものであり，曲線の傾きが－1に近いほど理想的なスケーラビリティとなる．プロセス数（Npx×Npz）の増加とともに計算時間が減少していることが見て取れる．しかし，プロセス数がある程度大きくなると通信のオーバーヘッドが大きくなり性能は低下する．ただし，このデータを取った時点では1プロセス1スレッド（Flat MPI）の計算を行っていた．これは，MPI並列と自動並列によるハイブリッド並列を行うと，前者の場合より約10%から20%程度，計算時間が長くなるためであった．MPI並列とOpenMPによるハイブリッド並列を用いると，Flat MPI並列と同等の計算時間となった．

3次元FFTのチューニングは未だ発展途上であり，上に述べた以外にも高速化，高効率化の手法が考えられる．たとえば(a)物理変数が実数の時には，複素数表現の実数部と虚数部を使って2つの関数を同時に変換する方法，(b)ある$k_z$面でFFTの計算を行っているときにすでにFFTが完了した$k_z-1$の面内の計算結果を転送できれば，通信時間を少なくすることができる（通信の隠ぺい）．

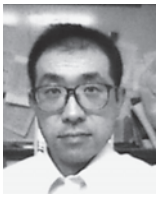
このFFTは3次元の流体力学乱流やMHD乱流のシミュレーション向けに開発されたものであるが，乱流の普遍的統計法則は高いレイノルズ数において現れると考えられており，この状態を計算機上で実現するためにはより高解像度の計算が必要である．流体力学の世界では，格子点数$2048^3$での計算は今や当たり前になってきており，格子点数$4096^3$での計算も特定のグループで行われ，$8192^3$での計算にも手が届きつつある．このような大規模な計算が可能な3次元FFTの開発は乱流研究だけでなくあらゆる分野で必須である．

最後に，公開されているFFTライブラリの利用について触れたい．最初に述べたように，有名なFFTライブラリとしてはFFTW[1]と，これを利用したP3DFFT[2]がある．FFTWはC言語で記述されているが，Fortranのためのインターフェースが用意されている．また，スレッド並列およびMPIを用いた並列化のための関数が用意されている．P3DFFTはFFTWなどを利用した3次元FFTである．本章で紹介した3次元FFTと同様に2次元の領域分割が行われているため，メニーコアのスーパーコンピュータで高効率の3次元FFTが実行可能となっており，乱流の大規模3次元シミュレーションなどに利用されている．この他の2次元領域分割型FFTについては[3]が参考になる．このウェブサイトにはFFTW以外の公開FFTライブラリや，2次元領域分割に関する説明もあり，公開ライブラリの利用者に親切な作りになっている．ここに紹介されている多くの3次元FFTは2次元領域分割を行うが，FFTW-3D[4]のように3次元領域分割を行うものもある．（著者はまだこの3次元領域分割ライブラリを使っていないが．）ウェブサイト[3]では紹介されていない公開ライブラリとしては，たとえばCREST事業で開発された国産ライブラリFFTSS[5]が上げられる．FFTSSではFFTW 3とインターフェイスレベルで互換性があり，FFTWからの移植が容易であるとされている．この他にも多数のFFTが公開されているので，興味を持った読者は検索してみるとよい．

トーラスプラズマのMHDシミュレーションではトロイダル方向，ポロイダル方向にはフーリエ変換を行うが，動径方向には差分法などを使う場合，あるいはトロイダル方向にはフーリエ変換を使い，ポロイダル断面上では有限要素法を使う場合が多い．これらの場合には，FFTWなどのライブラリから一次元FFTを用いて必要な方向にフーリエ変換を行うと良い．フーリエ変換を行う次元については共有メモリー内でのスレッド並列化を行う一方で，動径方向には差分法による離散化を行い，MPIによる並列化を行うのが一番簡単であろう．差分法を用いた場合のMPIによる並列化については前節を参考にされたい．

## 参考文献


［1］http://www.fftw.org/
［2］http://www.sdsc.edu/us/resources/p3dfft/; http://code.google.com/p/p3dfft/
［3］http://www.2decomp.org/
［4］http://www.gromacs.org/Developer_Zone/Programming_Guide/FFTW-3D/
［5］http://www.ssisc.org/fftss/index.html.ja.euc-jp



三浦英昭

核融合科学研究所ヘリカル研究部所属．流体力学・一様等方性流体の乱流シミュレーション，渦同定研究などを経て，現在は高温プラズマ・不安定性シミュレーションの研究に従事する．

後藤俊幸

名古屋工業大学大学院，研究分野：乱流理論と乱流の計算科学．福井県出身．荷風にならって，街中をぶらぶら歩くことが好きです．それと，近くの日帰り温泉につかってのほほんとしています．

藤堂泰

総合研究大学院大学物理科学研究科核融合科学専攻および核融合科学研究所・教授．流体シミュレーションと粒子シミュレーションを連結して，核融合プラズマにおいて高エネルギー粒子が不安定化するアルフベン固有モードとアルフベン固有モードによる高エネルギー粒子輸送を研究している．この研究を物理として発展させたい．学生募集中．