J. Plasma Fusion Res. Vol.89, No.4 (2013) 245-260

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

5．粒子シミュレーションのコーディング技法

5. Coding Techniques of Particle Simulations

内藤裕志，佐竹真介[1)]
NAITOU Hiroshi and SATAKE Shinsuke[1)]
山口大学大学院理工学研究科，[1)]核融合科学研究所核融合理論シミュレーション研究系

（原稿受付：2013年3月15日）

この章ではプラズマ中のMHD現象や輸送現象の解析に用いられるPIC法とモンテカルロ法を題材に，粒子シミュレーションのコーディングにおける並列化，高速化のポイントを紹介する．PIC法では領域分割と粒子分割のハイブリッド並列化，モンテカルロ法では乱数発生法の並列化とメモリへのランダムアクセスの抑制について主に解説する．

Keywords:
Particle-In-Cell, Monte-Carlo, parallel computing, domain decomposition, random numbers

5.1 序論

プラズマ・核融合分野に応用される代表的な粒子シミュレーション法はParticle-In-Cell法（PIC法）とモンテカルロ法が挙げられる．両者とも，プラズマが荷電粒子の集まりであるという粒子的描像を反映させて作られたシミュレーションモデルであり，3章のMHDシミュレーション，4章のVlasovシミュレーションで取り扱った3次元配位空間の電磁流体，あるいは5次元位相空間（3次元配位空間＋2次元速度空間）における荷電粒子の分布関数を連続体として扱う手法とは異なる数値計算上の工夫が必要となる．

2つの粒子シミュレーション手法がプラズマ・核融合分野のどのような研究に応用されているかをみてみると，PIC法は磁気リコネクション現象のように外部電磁場に加えてプラズマ自身が作り出す電磁場が強く影響する現象を調べる際や，電磁波と荷電粒子の相互作用，荷電粒子の集団的振る舞い，強いビーム成分の存在するプラズマの研究などに用いられる．荷電粒子間の近接相互作用，いわゆるクーロン衝突は無視され，デバイ長以上のスケールでの電荷の不均一による長距離相関に着目したモデルである．PIC法ではシミュレーション粒子1つ1つが荷電粒子を多数集めた「超粒子」とみなされ，実際の荷電粒子の運動を少ないシミュレーション粒子で代表させて現象を模擬する．

一方モンテカルロシミュレーションはプラズマ・核融合分野において主に輸送現象の解析に応用されている．例えば核融合反応で発生する高速アルファ粒子や周辺部からコアプラズマに流入する不純物イオンの輸送，その他荷電粒子以外にも中性子やプラズマ周辺領域の中性粒子がどう拡散するかを調べる際に用いられる．あるいはコアプラズマの新古典輸送を計算するために，クーロン衝突項を含むドリフト運動論方程式を粒子シミュレーションとして解くためにも応用される．いずれの手法においても，衝突散乱過程を模擬するために乱数を使うことからモンテカルロ法と呼ばれる．

PIC法とモンテカルロ法は粒子コードとして計算の並列化・高速化に関する技法に共通する部分も多いが，それぞれの手法の中でネックとなる部分には違いがある．PIC法では荷電粒子が作る電磁場の計算にかかるコストがモンテカルロ法に比べると大きく，粒子軌道計算の並列化と場の計算の並列化をどのように組み合わせて実装するかが大規模並列化の鍵となる．5.2節では3次元PICコードを例に，並列化コードのプログラミング法を，順を追って解説する．またGpic-MHDコードでの高並列化の例を紹介する．一方，モンテカルロコードを並列化する上でまず必要となるのは乱数の並列化であり，5.3節でいくつかの方法を紹介する．また，モンテカルロ法ではシミュレーション粒子の分布からアンサンブル平均量やモーメント量を取ることが多いが，その際にメモリへのランダムアクセスによる計算速度の劣化をどう防ぐかについても5.3節で解説する．

5.2 PICシミュレーション

プラズマは自由に動き回る電子とイオンの集合体である．電子とイオンは外部磁場や外部電場に加えて，自分自身が作る電磁場の影響下で運動するため，きわめて複雑かつ神秘的な振る舞いを示す．この超多体的性質を持つプラズマを，コンピュータでシミュレーションする手法の一つとして粒子的手法がある．粒子的手法では，個々の荷電粒子の運動を支配するニュートン・ローレンツの運動方程式

authors' address: Graduate School of Science and Engineering,Yamaguchi University, Ube 755-8611 Japan, [1)]National Institute for Fusion Science, Toki 509-5292 Japan
e-mail: naitou@yamaguchi-u.ac.jp, satake@nifs.ac.jp

を，電磁場の時間・空間変化を支配するマクスウェルの方程式と連立して解くことにより追跡する．粒子的手法は，基礎方程式はきわめてシンプルであるが，巨大なコンピュータ資源を要求するため，第一原理シミュレーションの一つとして分類される．

粒子的手法を用いたコードは普通 PIC（Particle-In-Cell）コードと呼ばれる．PICコードの詳しい解説は文献[1]に示されている．PIC では，個々の粒子は空間を自由に飛び回るのに対して，電場や磁場の場の量の計算には（メッシュやグリッドと呼ばれる）空間格子を用いる．空間格子の最小単位をセルと呼ぶ．1個の荷電粒子は空間内のどこかの1セル内に存在することになる．現在または近未来のコンピュータでは，プラズマ中のすべての電子とイオンの振る舞いをシミュレーションすることは不可能である．PIC コードで追跡する電子（イオン）は多数の電子（イオン）の電荷と質量を一つにまとめた超粒子であることに注意する．超粒子はその近傍の格子点上（線形補間の場合は超粒子の属するセルの各頂点）の電磁場の影響のみを受けて運動する．

本節の構成は以下のとおりである．5.2.1では，PIC コードの基本構成を簡単に紹介する．5.2.2では，読者が実際に PIC コードを並列化する際の参考になるように，簡単な並列化からより複雑な並列化に向かってコードを変更していくステップを提案する．5.2.3では，ジャイロ運動論に基づいた PIC コードである Gpic-MHD（Gyrokinetic PIC code for MHD simulation）の核融合科学研究所の Plasma Simulator（日立 SR16000）と IFERC-CSC の HELIOS（Bull 社 Bullx B510）の両スーパーコンピュータ上での並列化の経験と現状について解説する．5.2.4では粒子コードの並列化のまとめを述べる．

5.2.1 PIC コードの基本構成

簡単のため3次元の静電近似コードを例として PIC コードの基本構成を概説する．直角座標中の各辺が L_x，L_y，L_z の直方体のシステム中に，電子（$s=e$）と1種類のイオン（$s=i$）からなる超粒子プラズマが存在している．s 種の粒子の電荷を q_s，質量を m_s，個数を N_s とする．時間 t での粒子の位置と速度を $\mathbf{x}_{sj}(t)$，$\mathbf{v}_{sj}(t)$（$j=1,N_s$）とする．PIC コードは荷電粒子の運動を初期値問題として解くため時間差分を用いる．時間差分には，PIC コードの標準的手法で，粒子の位置と速度が $0.5\Delta t$ だけ異なった時間で計算される蛙飛び法（leap-frog-method）を用いることにする．ここで Δt は時間差分のステップ幅である．蛙飛び法では時間微分が中心差分で近似されるように差分化を行っている．中心差分を用いると，実際の運動方程式と同様に，差分化した運動方程式も時間反転に関して対称（time-reversal）になる．なお蛙飛び法は，1段法でありながら，2次の Runge-Kutta 法と同等の精度（誤差は $(\Delta t)^3$ のオーダー）が保証されるアルゴリズムになっている．まず，すべての粒子の位置と速度の初期値 $\mathbf{x}_{sj}(0)$，$\mathbf{v}_{sj}(-0.5\Delta t)$ を与える．例えば位置は空間的に一様に，速度はガウス乱数を用いてマクスウェル分布に従うように設定する．場の量は空間的に離散化された等間隔の格子の格子点上でのみ計算される．格子間隔を Δ で表す．格子点上での，電荷密度，静電ポテンシャル，電場を表す配列 $\rho(\mathbf{x}_G)$，$\phi(\mathbf{x}_G)$，$\mathbf{E}(\mathbf{x}_G)$ を用意する．ここで $\mathbf{x}_G$ は各格子点の座標を示す．以下では，場の量の（格子点間）補間や粒子の電荷の格子点への分配は線形補間を使用することとする．

時間を1ステップ（Δt）進めるためには以下に示す **SOURCE**，**FIELD**，**PUSH** のループを繰り返せばよい．空間は Δ で規格化されているとする．特定の粒子 $\mathbf{x}_{sj}(t)=(x_{sj},y_{sj},z_{sj})$ は，以下に示すように，(l,m,n) で代表されるセルに属している．

$$
\begin{aligned}
x_{sj} &= l+\delta x \qquad (0\le \delta x<1)\\
y_{sj} &= m+\delta y \qquad (0\le \delta y<1)\\
z_{sj} &= n+\delta z \qquad (0\le \delta z<1)
\end{aligned}
$$

ループの入り口では，すべての粒子の位置と速度 $\mathbf{x}_{sj}(t)$，$\mathbf{v}_{sj}(t-0.5\Delta t)$ が与えられている．またループの入り口では毎回 $\rho(\mathbf{x}_G)$ にすべてゼロを代入する．

1．SOURCE：各粒子の電荷を各粒子の属するセルの8個の頂点に分配し加算する（charge assignment）．加算する配列は $\rho(\mathbf{x}_G)$ である．以下で矢印は代入を示す．

$$
\begin{aligned}
\rho(l,m,n) &\leftarrow \rho(l,m,n)\\
&\quad + q_s(1-\delta x)(1-\delta y)(1-\delta z)\\
\rho(l+1,m,n) &\leftarrow \rho(l+1,m,n)\\
&\quad + q_s\delta x(1-\delta y)(1-\delta z)\\
\rho(l,m+1,n) &\leftarrow \rho(l,m+1,n)\\
&\quad + q_s(1-\delta x)\delta y(1-\delta z)\\
\rho(l+1,m+1,n) &\leftarrow \rho(l+1,m+1,n)\\
&\quad + q_s\delta x\delta y(1-\delta z)\\
\rho(l,m,n+1) &\leftarrow \rho(l,m,n+1)\\
&\quad + q_s(1-\delta x)(1-\delta y)\delta z\\
\rho(l+1,m,n+1) &\leftarrow \rho(l+1,m,n+1)\\
&\quad + q_s\delta x(1-\delta y)\delta z\\
\rho(l,m+1,n+1) &\leftarrow \rho(l,m+1,n+1)\\
&\quad + q_s(1-\delta x)\delta y\delta z\\
\rho(l+1,m+1,n+1) &\leftarrow \rho(l+1,m+1,n+1)\\
&\quad + q_s\delta x\delta y\delta z
\end{aligned}
$$

すべての粒子に対して上記の処理を行った後，適当な定数をかければ，時間 t での格子点上での電荷密度が計算されたことになる．

2．FIELD：格子点上の電荷密度 $\rho(\mathbf{x}_G)$ から時間 t での電場 $\mathbf{E}(\mathbf{x}_G)$ を計算する．通常は，ポアッソン方程式により $\rho(\mathbf{x}_G)$ から静電ポテンシャル $\phi(\mathbf{x}_G)$ を求め，$\phi(\mathbf{x}_G)$ の勾配から $\mathbf{E}(\mathbf{x}_G)$ を計算する．

$$
\begin{aligned}
\nabla^2\phi &= \rho/\varepsilon_0\\
\mathbf{E} &= -\nabla\phi
\end{aligned}
$$

3．PUSH：時間 t での各粒子の属するセルの8個の頂点の電場 $\mathbf{E}(\mathbf{x}_G)$ から個々の荷電粒子の位置での電場 $\mathbf{E}(\mathbf{x}_{sj}(t))$ を求め，

$$\begin{aligned}\mathbf{E}(\mathbf{x}_{sj}(t)) = &(1-\delta x)(1-\delta y)(1-\delta z)\mathbf{E}(l,m,n)\\ &+\delta x(1-\delta y)(1-\delta z)\mathbf{E}(l+1.m.n)\\ &+(1-\delta x)\delta y(1-\delta z)\mathbf{E}(l,m+1,n)\\ &+\delta x\delta y(1-\delta z)\mathbf{E}(l+1,m+1,n)\\ &+(1-\delta x)(1-\delta y)\delta z\mathbf{E}(l,m,n+1)\\ &+\delta x(1-\delta y)\delta z\mathbf{E}(l+1,m,n+1)\\ &+(1-\delta x)\delta y\delta z\mathbf{E}(l,m+1,n+1)\\ &+\delta x\delta y\delta z\mathbf{E}(l+1,m+1,n+1)\end{aligned}$$

この電場を使って，1ステップだけ荷電粒子の位置と速度を進める（particle pushing または particle acceleration）．

$$\mathbf{v}_{sj}(t+0.5\Delta t)=\mathbf{v}_{sj}(t-0.5\Delta t)+(q_s/m_s)\mathbf{E}(\mathbf{x}_{sj}(t))\Delta t$$
$$\mathbf{x}_{sj}(t+\Delta t)=\mathbf{x}_{sj}(t)+\mathbf{v}_{sj}(t+0.5\Delta t)\Delta t$$

なお，磁場がある場合は次式を用いる．次式も時間に対する中心差分を採用していて，時間反転に関して対称である．

$$\mathbf{v}^{-}=\mathbf{v}_{sj}(t-0.5\Delta t)+0.5(q_s/m_s)\mathbf{E}(\mathbf{x}_{sj}(t))\Delta t$$
$$\mathbf{v}'=\mathbf{v}^{-}+\mathbf{v}^{-}\times 0.5(q_s/m_s)\mathbf{B}(\mathbf{x}_{sj}(t))\Delta t$$
$$\mathbf{v}^{+}=\mathbf{v}^{-}+\mathbf{v}'\times(q_s/m_s)\mathbf{B}(\mathbf{x}_{sj}(t))\Delta t/\{1+[0.5(q_s/m_s)B(\mathbf{x}_{sj}(t))\Delta t]^2\}$$
$$\mathbf{v}_{sj}(t+0.5\Delta t)=\mathbf{v}^{+}+0.5(q_s/m_s)\mathbf{E}(\mathbf{x}_{sj}(t))\Delta t$$
$$\mathbf{x}_{sj}(t+\Delta t)=\mathbf{x}_{sj}(t)+\mathbf{v}_{sj}(t+0.5\Delta t)\Delta t$$

すべての粒子に対してこの処理をした後，シミュレーション領域（直方体）から出た荷電粒子の処理をする（粒子に関する境界条件の適用）．

ループの出口では全粒子の $\mathbf{x}_{sj}(t+\Delta t)$，$\mathbf{v}_{sj}(t+0.5\Delta t)$ が求まっている．$t \leftarrow t+\Delta t$ としてこのループを所定のステップ数だけ繰り返せばPICシミュレーションは終了する．

SOURCE では荷電粒子の電荷を空間的にランダムな格子点に対応する配列の成分に加える scatter の計算を実行している．また **PUSH** では，粒子の感じる電場は空間的にランダムな格子点に対応する配列の成分を引用する gather の計算を実行している．なお，荷電粒子が自分自身の作る電磁場で自分自身を加速しないために，**SOURCE**での電荷の分配公式と **PUSH**での電場の内挿公式は同一にしておく必要がある．PICコードでは粒子に関する統計的精度を維持するため，1セル内に数十から数百（場合によってはそれ以上）の粒子を含めてシミュレーションを実行する．このため一般に，場の量の計算と比較して，粒子に関する計算が支配的である．したがって**SOURCE**と**PUSH**の並列化がPICコードの並列化に特有の問題である．**FIELD** でのポアッソン方程式の解法では高速フーリエ変換か差分法を用いる．場の量の計算は一般の数値計算の教科書にあるものと本質的に変わらない．

電磁コードの場合には場の量の計算は少し複雑になる．また，ここでの解説で用いた蛙飛び法は，ニュートン・ローレンツの運動法定式を時間積分するのに適した1段法である．ドリフト近似を用いた運動方程式等を時間積分する場合は，Runge-Kutta 法や予測子・修正子法などの多段法を用いる必要がある．5.2.3で紹介する Gpic-MHD コードの場合は2次の Runge-Kutta 法（2段法）を用いている．時間積分に多段法を使用する場合や，内挿公式に高次のスプラインを用いる場合でも，ここで述べた PIC コードの基本構造は本質的に変わらない．なお PIC コードの基礎をもう少し詳しく知りたい読者は，本学会誌の講座「プラズマ計算機シミュレーション入門」の中の「2.粒子シミュレーションの基礎」[2]を参照されたい．

5.2.2 PIC コードの並列化

読者がこれから並列化PICコードを作成することを仮定して，最短の並列化の順序を提案したい．以下に示すように小規模の並列化から大規模の並列化へとプログラムを変更していく順序に従い解説する．

(1) 並列化前の最適化
(2) スレッド並列化
(3) プロセス並列化1（粒子分割）
(4) プロセス並列化2（1次元領域分割＋粒子分割）
(5) プロセス並列化3（多次元領域分割＋粒子分割）

シミュレーションしたい物理現象によっては，小規模または中規模の並列化で終了することも可である．また(2)は場合によっては省略可能である．

(1) 並列化前の最適化

並列化前の PIC コードは存在しているとする．まずは，この PIC コードを最適化しておくことが重要である．基本的には**SOURCE**と**PUSH**のサブルーチンを最適化すれば良い．実際には，最近のコンパイラは優れていて，よく知られている最適化手法を適用してもそれほどの差がでないことを認識される結果になる場合が多いと思われる．ただし，グリッドサイズが大きい場合にはキャッシュの関係で差がでてくるので注意する．このステップは PC でも可能である．一見無駄なステップであるが，コードが並列化され複雑化する前に一度試しておくことは重要だと思われる．

(2) スレッド並列化

最初にスレッド並列化をする．OpenMP または自動並列化を用いる．自動並列化は機種に依存するため可搬性を考えるなら OpenMP で書いておいたほうがよいかもしれない．OpenMP で書いておいても，コンパイルの際に OpenMP を使用しないで自動並列化を有効にすることも可能である．最近の PC では1チップ上に4コアが搭載されているものがあり，intel 系のチップでは，1物理コアを2論理コアとして動作させる hyper-threading 技術が利用できるため，8並列までのスレッド並列化が可能である．パブリックドメインのコンパイラ gfortran 等も OpenMP に対応している．この段階でFFTを使用している場合は並列化 FFT を利用するように変更しておく．

PUSH の計算は簡単にスレッド並列化できる．これは，同時に異なるスレッドから同じ配列の成分にアクセスがあった場合も，配列の値の引用のみであるため，コンピュータが適切に処理をするような設計になっていること

による．ただし，メモリの取り合いがスレッド並列化の効果を限定的にすることに注意する．

SOURCE では配列の成分へのランダムな加算計算を実行する必要があり，そのままでは並列化が困難である．これは異なるスレッドから同時に同じ配列の成分に加算するアクセスがあった場合の動作が保証されていないことによる．実際，OpenMP では，そのままでは並列化できない．このため並列化のためのアルゴリズムを工夫する必要がある．筆者はまだ試みていないが，場の量の複数のコピー配列を用いてスレッド毎に計算した後，スレッド間の総和計算を行う［(3)の粒子分割に似た手法］とか，スレッド毎に場の量の異なる領域を担当する［(4)の領域分割に似た手法］等のアルゴリズムが適用できると思われる[3]．**SOURCE** 部に現れる演算はモンテカルロ法でもボトルネックとなるものであり，スレッド並列化に関するいくつかのアイデアが5.3.2節で解説されているので参考にされたい．幸い Plasma Simulator では，HITACHI の自動並列化コンパイラが（実際どのような手法で並列化しているのが不明であるが）加算に対応しているので，DO ループの直前に

```
*poption parallel, sum（加算する配列名）
```

という並列化制御行を入れておくだけでよい．なお自動並列化の場合は，かなり複雑なループも無理に並列化してしまう危険性があるため，並列化で逆に遅くなっているループを探し出して，

```
*poption noparallel
```

という並列化制御行を入れて，並列化を禁止することが重要になる．

本節の本筋からは離れるが，加算のスレッド並列化は GPU（Graphic Processing Unit）を用いた並列化の場合にも問題になる．GPU に含まれる多数の演算コアを，図形処理ではなく，一般の計算に利用することは，GPGPU（General-purpose computing on graphic processing units）と呼ばれる．GPU を使用する場合は，各粒子を異なるスレッドで計算するように並列化することができないため，セル毎にスレッド並列化するアルゴリズムが存在する[4]．各スレッドは各セルに対応してそのセル内の粒子の処理をする．この場合には **PUSH** の後で，粒子を各セルに再配置する処理が必要になる．領域分割の手法を 1 セルの極限にまで拡張し適用したような例になっている．

並列化が終了した段階で，スレッド数を変更して高速化性能（計算時間の逆数）のスケーリングを確認しておくこと．ただし，スレッドあたりの粒子数は十分大きくなるようにする．粒子数が十分大きくない場合はすぐにスケーリングが飽和してしまうので注意する．小規模の並列化で十分な場合はここで終了である．ただし，小規模の並列化の場合はこのステップを省略して次のステップでのプロセス並列（粒子分割）のみを採用する選択もある．通常スレッド並列は DO ループ単位の並列化であり，DO ループが呼ばれる毎に，処理を異なるコアに分割して計算するため，スレッド数が多くなるにつれて高速化が飽和しやすい傾向がある．プロセス並列の場合は大きな粒度で並列化できるため並列化性能を引き出しやすい．

(3) プロセス並列化 1 （粒子分割）

プロセス並列化を開始する．(2)でスレッド並列化を済ましていると，ここからはスレッド並列とプロセス並列のハイブリッド並列になる．まず，第一段階として粒子分割（particle decomposition）の手法により並列化する．粒子分割の基本概念を**図 1** に示す．MPI を用いる．場の量（電場と磁場）にレプリカ（コピー）を用いて，各プロセスに対応させる．各プロセスは，レプリカとそのレプリカ内の粒子の計算を行う．粒子の位置と速度の初期値は，プロセス毎に変えておく必要がある．レプリカ数を N_{copy} ($= N_{\mathrm{process}}$) とすると，各プロセスは全体の粒子の $1/N_{\mathrm{copy}}$ 個の粒子を取り扱うため，粒子分割と呼ばれている．粒子に関する演算は各プロセスで完全に独立である．プロセス間の粒子の移動はない．PIC コードは基本的には粒子の計算が，場の量の計算より圧倒的に多い．このため，粒子分割は非常に有効な並列化アルゴリズムである．

SOURCEでは各レプリカの電荷密度と電流密度（静電近似コードの場合は電荷密度のみ）を計算する．**SOURCE** の直後に，全レプリカ間の総和を計算し，各プロセスに分配する総和計算（**SUM**）を実行すれば並列化は完了する．総和計算のアルゴリズムで一般によく知られているものを紹介する．各プロセスは，それぞれのデータ（配列）を保持している．プロセス数を 2^N とする．プロセス数が 8 ($N = 3$)の例を**図 2** に示す．n 番目のステップでは，全データを 2^n 個ずつのデータの組に分け，それぞれの組の中で 2^{n-1} 離れたデータのペアを作る．ペア間でデータの送受信を行い，2 個のデータの和を計算する．N ステップで総和計算は完了する．総和計算は，MPI の関数である `MPI_ALLREDUCE` を用いても計算できるが，読者には MPI でプロセス並列化する場合の練習問題として一度作成して実行してみることを勧める．また，並列化したコードは将来総和計算をカスタマイズしたい場合の基本コードとしても利用できる．

各プロセス（レプリカ）は，総和計算の結果の電荷密度

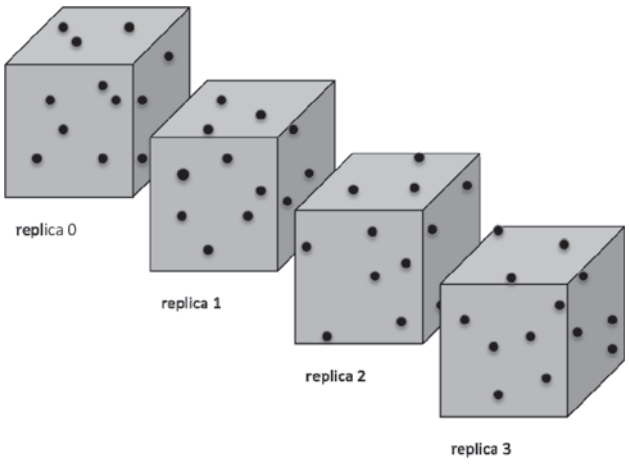

図 1　粒子分割によるプロセス並列化の基本概念．

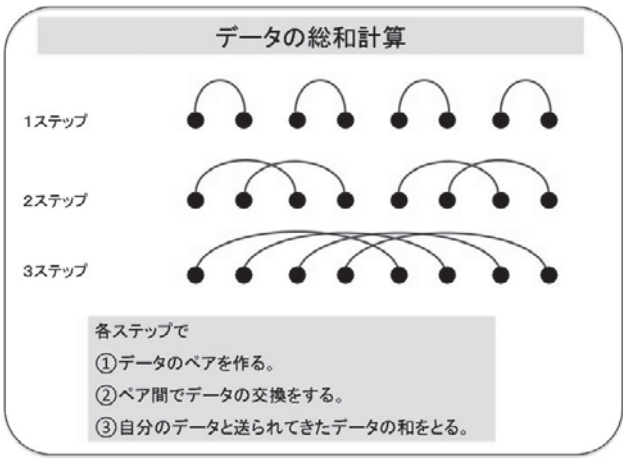

図2　総和計算のプロセス並列化のアルゴリズム．

と電流密度を用いて，それぞれ電場と磁場を計算する．この部分は全プロセスでまったく同じ計算をすることになるので冗長計算になっている．

粒子分割の1ループの手順をまとめると以下のようになる．ここで追加した部分を網掛で示している．

1．**SOURCE**
2．**SUM**（総和計算）
3．**FIELD**
4．**PUSH**

粒子並列の並列化性能を確認するために，各プロセスの粒子数を固定して，プロセス数（＝レプリカの数）を変化させて計算時間のスケーリングを計測する．総和計算にかかる時間は，レプリカ数を N_{copy} とすると，$N_{\mathrm{copy}} \log_2 N_{\mathrm{copy}}$ に比例する．総和計算を除いた部分は各プロセスで完全に独立であるため，計算時間に変化がない．このため総和計算を除いた計算時間は，理論的にも実測でもプロセス数に関係なくまったく同一になる．総和計算にかかる計算時間がその他の部分の計算時間と同程度になった時点で，粒子並列による性能向上は飽和することになる［粒子シミュレーションでは超粒子のノイズを低減して粒子数無限大（無衝突の極限）へのスケーリングを調べる必要がある場合がある．上記のスケーリングはこの場合のスケーリングの導出の際にも必要になる］．ここでの並列化は各プロセスに対する負荷分布が一様である．粒子並列のプログラムは基本的には総和のプロセスを導入するだけであるので簡単に短時間で並列化可能である．ここで並列化を終了しても，シミュレーションしたい物理現象によっては十分である．場の量の配列が大きく（場合によっては各コアのメモリに入りきれない場合を含む），場の量の計算時間が粒子を1ステップ進める時間と比較して無視できない場合は領域分割による並列化を採用する必要がある．

(4) プロセス並列化2（1次元領域分割＋粒子分割）

1次元の領域分割（domain decomposition）をコードに組み込む．ここでは場の量を z 方向に，N_{divz} 個に等分割するとして説明する．簡単のため，x，y，z 方向のメッシュ数 N_x，N_y，N_z は同じとする．まず，$N_{\mathrm{copy}} = 1$ にして，純粋な1次元領域分割のプロセス並列化の概念を**図3**に示す．場の量に関する領域分割は3.2節で述べられているものと基本的に同じである．隣り合う分割領域（subdomain）に共有される袖領域［ガードセル（guard cell)］の取り扱いに注意すればよい．

以下，すべての方向に高速フーリエ変換（FFT）を利用するとして解説する．3.3節で述べられた転置（transpose）の手法を用いる．まず x，y 方向についてFFT（順FFT）を実行する．z 方向のFFTを実行する前に，分割の方向を例えば x 方向に転置しておく．転置後は，z 方向のFFTの計算が異なるプロセスにまたがることはない．フーリエ空間で場の量を求めたら，z 方向の逆FFTを実行し，その後で分割の方向を x 方向から z 方向になるように逆転置を行い，最後に x，y 方向の逆FFTを実行すれば，実空間での場の量が求まる．

粒子コードの場合は，分割領域がそれぞれ多数の粒子を含んでいる．このため，**PUSH**で1ステップ粒子の位置と速度を進める毎に，異なる分割領域に出ていく粒子を行き先の分割領域に送り出し，異なる分割領域から入ってくる粒子を受け取る必要がある．このプロセスを**MOVE**で表すことにする．いかなる粒子も隣接する分割領域にしか到達できないことを仮定してプログラムするのが簡単である．このため，時間ステップ幅は最も高速の粒子でも，隣の分割領域にしか到達しないように選択しておく必要がある．プロセス間のデータ通信はパケット毎に行われるため，まとまった量を同時に送受信する必要がある．このため，各分割領域で，送り出す粒子と受け取る粒子を保存するバッファー用配列を準備する必要がある．分割領域から出ていく全粒子のデータを送信用バッファー配列（行き先は2方向あるので各2配列を用意する必要がある）に格納し，隣接する分割領域に送信する．このとき受信も同時に行う．`MPI_Isend/Irecv` を用いた非同期送受信を行う方法は2.4節で述べられている．受信用バッファー用配列（これも2方向に対応して2配列を用意する必要がある）で受け取った粒子の情報を，各分割領域の粒子用配列に格納する．各分割領域の粒子数は平均して一様になるが，分割領域間の粒子の移動があるため，各分割領域の粒子数は時間

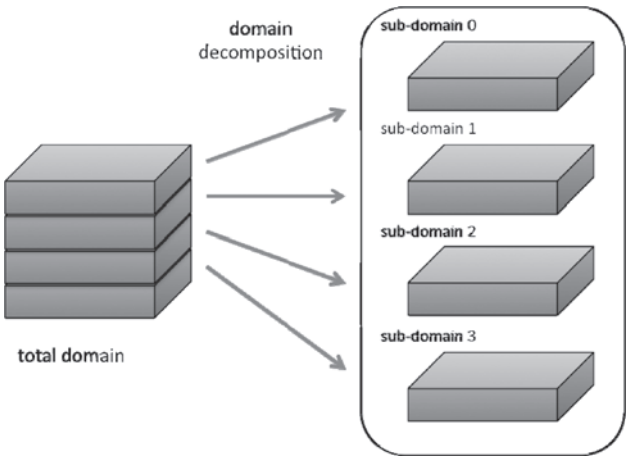

図3　領域分割によるプロセス並列化の基本概念．

的に変化する．このため，粒子に関する情報を格納する配列の大きさは十分余裕をもって設定しておく必要がある．

この段階で，分割数に関するスケーリングをよく検証しておく．各分割領域の粒子数を固定して分割数N_{divz}を増やしていくと，理想的には場の量の計算は$1/N_{\mathrm{divz}}$に比例して小さくなる．実際には袖領域や**MOVE**の計算に関するプロセス間通信が加わる．FFTを使用する場合はフーリエ変換されたデータがモード毎に実部，虚部の順に入るため，フーリエ空間での計算が異なる分割領域（厳密にはフーリエ空間での分割領域になっている）に属しないためには，分割可能な分割数の最大値は$N_{\mathrm{divz}} = N_z/2$になる場合があることに注意する．通常，分割数は最大値に固定してよいが，念のため計算時間の，分割数に対するスケーリングを実測し，最速の分割数を確認しておくことが重要である．

コードが正しく動いていることが検証できたら，総和計算の部分を分割領域に対応するようにコードを書き換えればこの段階でのプログラムの変更は終了する．この場合は各分割領域が，それぞれのレプリカをもつことになる．領域分割と粒子分割によるプロセス並列化の基本概念を**図4**に示す．ここでは，グループという概念を導入した方が説明しやすい．グループ数はレプリカ数N_{copy}に等しい．それぞれのグループ内の粒子の運動はグループ内で閉じている．このため**MOVE**の計算はグループ内の通信のみで達成される．また**TRANSPOSE**と袖領域の計算もグリープ内で閉じている．分割領域毎の総和計算**SUM**のみがグループをまたがった通信を必要とする．

ここで，N_{divz}を最適値に固定し，プロセス毎の粒子数を固定して，計算時間のN_{copy}依存性を調べ，N_{copy}をどの程度まで問題なく増やせるか確認しておく．N_{copy}が大きくなると，総和計算の時間が無視できなくなるので，N_{thread}を増やした方が，計算時間が短縮される場合がある．このため，使用コア数を固定し，コピー数を増やした場合と，スレッド数を増やした場合の最適値がどのようになるかを検証しておく必要がある．ここで，全使用コア数は以下のようになる．

$$N_{\mathrm{core}} = N_{\mathrm{thread}} \times N_{\mathrm{process}} = N_{\mathrm{thread}} \times N_{\mathrm{divz}} \times N_{\mathrm{copy}}$$

この等間隔の領域分割は不安定性等が生じても分割領域間で粒子密度の非一様性が大きくない物理現象の場合に有効であることに注意する．ビーム不安定性のように粒子のバンチングによる大きな疎密が生じる場合には，分割領域境界の動的変更などの動的領域分割手法を取り入れる必要がある．

領域分割と粒子分割を併用した場合の各ステップに対応するループの構造は以下のように表される．

1．**SOURCE**
2．**SUM**（分割領域毎の総和計算＋ガードセルの処理）
3．**FIELD**（＋**TRANSPOSE**）
4．**PUSH**
5．**MOVE**（分割領域間の粒子の移動）

通常，1次元の領域分割数は使用可能な計算機のコア数よりはるかに少ない．このため，領域分割と粒子分割を併用したハイブリッドプロセス並列を使用する必要がある．このハイブリッドプロセス並列は，最近ではdomain cloningと呼ばれることがある．

この段階で数千コアの高並列コンピュータの性能を十分引きだせるコードになっていると思われる．この程度の並列化で十分予定していたシミュレーションが可能ならここで並列化を終了してよい．さらに大規模のシミュレーションを，数万コアから数十万コアの高並列コンピュータで実行したい場合には，次に示す多次元の領域分割に進む必要がある．

(5) プロセス並列化3（多次元領域分割＋粒子分割）

2次元の領域分割に対応したコードを作成する．1次元の領域分割を2方向に繰り返すことで2次元領域分割版のコードを作成することができる．さらに必要ならもう1方向に領域分割し3次元の領域分割コードを作成する．ここでそれぞれの方向の分割数を，N_{divz}，N_{divy}，N_{divz}とする

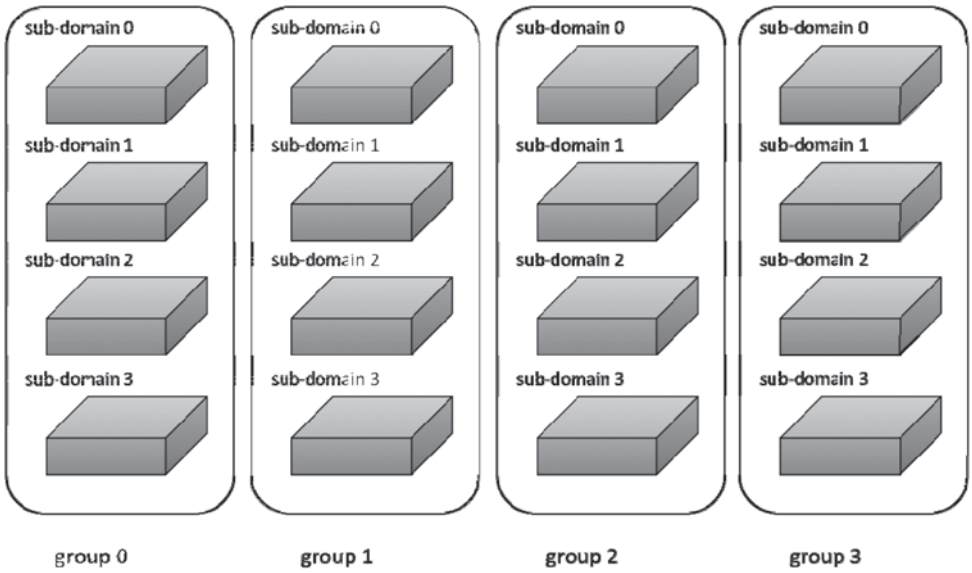

図4　領域分割と粒子分割によるプロセス並列化の基本概念．

と，3次元領域分割版での使用コア数は，

$$N_{\mathrm{core}} = N_{\mathrm{thread}} \times N_{\mathrm{divx}} \times N_{\mathrm{divy}} \times N_{\mathrm{divz}} \times N_{\mathrm{copy}}$$

になるため，数万から数十万の高並列コンピュータの計算機資源を十分利用可能になる．

ここでの例では，直角座標系を採用しているので特に問題はない．ただし複雑な座標系を使用している場合は，多次元の領域分割は容易ではない．例えば円筒座標の場合を考える．これは5.2.3で述べるGpic-MHDコードの場合に対応している．軸方向は問題なく領域分割できる．方位角方向は軸上の特異点の存在のため領域分割の方向に採用するのは難しい（軸上では粒子が任意の方位角方向分割領域に移動する可能性があるためMOVEの計算が複雑になる）．このため3次元の領域分割は実際的でない．2次元目の領域分割の方向として半径方向を採用する．半径方向の分割は，各分割領域に含まれる粒子数を一定にすると，非等間隔の分割になる．この場合，場の量の計算負荷は半径方向の分割領域毎に一様でなくなる．このため，半径方向の領域分割の効果は軸方向の領域分割の効果と比べるとすこし限定的である．軸方向と半径方向の2次元領域分割の例を**図5**に示す．

5.2.3　Gpic-MHDコードの並列化

ジャイロ運動論シミュレーションは第4章で紹介されたようにPIC法と，Vlasovシミュレーション法がある．ここでは，PIC法に基づくジャイロ運動論コードの一つであるGpic-MHD　コード（Gyrokinetic PIC code for MHD simulation）の並列化について解説する．Gpic-MHDは運動論的効果を含みながら，流体系のコードと比較して「クロージャー」の問題がないため，流体コードの結果の正当性・健全性をチェックするためにも有用である．

Gpic-MHDは直角座標に対応したGYR 3 Dコードを基礎としている．どちらも，トカマクの第0次近似である直線トカマクをシミュレーションしている．z方向がトロイダル方向を示し，この方向には周期境界条件が適用されている（なお，Gpic-MHDコードの最終目標はトロイダル版である．）基礎方程式系は文献[5]に示されている．荷電粒子（マーカー）の運動方程式はジャイロ運動論で表される．これはジャイロ運動論Vlasov方程式の特性曲線に対応している．またデルタエフ（δf）法を使用している．δf法では分布関数fを熱平衡成分f_0とそこからの変化分δfの和として表し，マーカー法を用いてδfの変化を計算する．各マーカーは重み（weight）$w_{sj}(t)$を持つ．マーカーの電荷と質量は，$q_s w_{sj}(t)$，$m_s w_{sj}(t)$になる．δf法では，運動方程式に加えて，重みの時間変化を表す式を同時に解く．

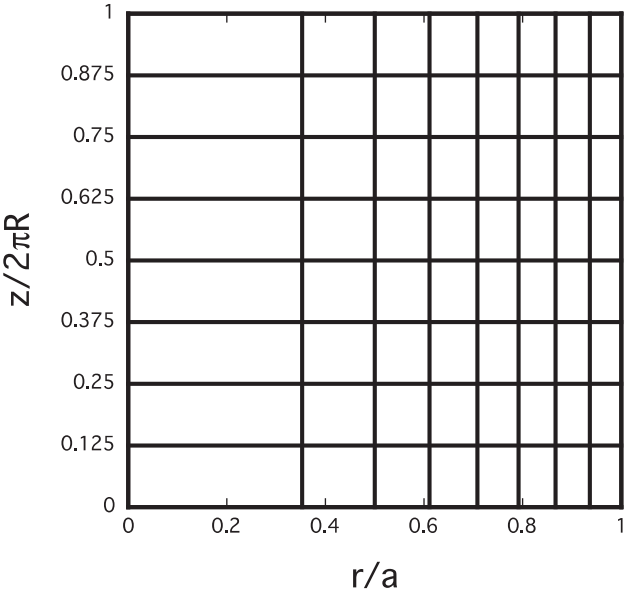

図5　軸方向と半径方向の2次元領域分割．
（文献[10]Fig.1より転載）．

GYR3Dの基礎方程式と，トカマクの鋸歯状振動の崩壊過程に関連する無衝突内部キンクモードの非線形シミュレーションは文献[5,6]に示されている．これはPICコードで，無衝突磁気再結合を含む巨視的・運動論的MHD現象（トカマクの内部崩壊現象に対応）を世界で初めてシミュレーションしたものである．GYR3Dの並列化は領域分割と粒子分割のプロセス並列化に対応していた[6]．日本語の解説としては，本学会誌の小特集「超並列計算機のシミュレーションとプラズマ」中の「1．プラズマの粒子シミュレーションコードの並列化」[7]を参照されたい．GYR3Dは，直角座標で書かれていたため，シミュレーションしたい物理に不必要な短波長で高周波数のアルベン波を含むため時間ステップ幅をきわめて小さくとる必要があり，計算機に対する負荷はきわめて大きいものであった．このため，円筒座標に対応したGpic-MHDコードを開発した．円筒座標には，半径方向のみに依存する平衡解が存在すること，トカマク中の固有モードの表示に適した軸方向と方位角方向のフーリエモード展開が利用できること等の利点ある．Gpic-MHDコードには，単一ヘリシティを仮定した2次元版Gpic-MHDと，マルチヘリシティに対応した3次元版Gpic-MHDが存在する．2次元版Gpic-MHDは，単一ヘリシティのモードのみを含むため，時間ステップ幅を大きくとれること，基本的に2次元コードであることで，3次元コードと比較してコンピュータに対する負荷が小さく，特に$m = 1/n = 1$（m，nはポロイダルとトロイダルのモード数を表す）の内部キンクモードのシミュレーションに最適である．また3次元版Gpic-MHDも不必要な高周波モードをフーリエ空間で除去できるため，適切な時間ステップ幅を選択することができる．

Gpic-MHDをPlasma Simulatorにインストールし，並列化性能を検証した．ただしここで示す結果は，増強前のPlasma Simulator（Phase-1）を使用した結果である．Plasma Simulatorでは1物理コアが2論理コアに対応している．2次元版2 D-Gpic-MHDは，スレッド並列（自動並列）とプロセス並列（MPI）によるハイブリッド並列により並列化した．プロセス並列は粒子分割による．8192論理コアまで良好な並列化性能が得られることを実証した[8]．

3次元版3 D-Gpic-MHDは，場の量の計算が増大するため，並列化のため領域分割を利用している．トロイダル方向のみの1次元領域分割と粒子分割を利用した場合の例[8,9]を以下に示す．なお，並列化はスレッド並列（$n_{\mathrm{thread}} = 2$）

とプロセス並列（$N_{process} = N_{copy} \times N_{divz}$）のハイブリッド並列になっている．メッシュは $N_r \times N_\theta \times N_z = 129 \times 128 \times 128$ とし，コア当たりの平均粒子数を百万個に固定し，1000ステップの計算を実行した．$N_{divz} = 64$ とし，N_{copy} を1から64（$N_{core} = 128 - 8192$）に変更してFLOPS値の論理コア数依存性を調べたものを**図6**に示す．比較的良好なスケーリングが Plasma Simulator の最大コア数まで得られることを実証した．なお，8192論理コアを使用して得られた2.4 TFLOPS は Plasma Simulator の理論最大値77 TFLOPS の3.1%である．

このスケーリングを主なループの構成要素に分けて分析した．プロセス間通信を必要としない**PUSH**，**SOURCE**，**FFT**（**FIELD**中の**FFT**だけを取り出したもの）に関する計算時間は論理コア数に依存せずほとんど一定であることが実証された（**図7**参照）．通信に関する部分**MOVE**，**TRANSPOSE**，**SUM**の計算時間の論理コア数依存性を**図8**に示す．**SUM**の計算時間が N_{copy} の増加とともに増大するのは自然である．**TRANPOSE**と**MOVE**の計算はグループ内にとどまるため，もし各グループで通信回線が独立であれば，計算時間は一定のはずである．このため，**TRANPOSE**と**MOVE**の計算時間の，N_{copy} の増加による増大は，通信回線が共通であることによる通信容量の制限に起因すると思われる．

トカマクの計算では有理面近傍の微細構造を解像するためにも，半径方向の解像度を増やす必要がある．半径方向のメッシュ数を増やし，$N_r \times N_\theta \times N_z = 1025 \times 128 \times 128$ として，計算時間を測定した．トロイダル方向と半径方向の2次元領域分割の場合1次元の領域分割より高速化されていることを実測した．これは，場の量の計算時間が半径方向の領域分割により短縮されていることによる．2次元の領域分割についても論理コア数に対する良好なスケーリングを得た[9,10]．増強後（Phase-2）の Plasma Simulator と HELIOS でのスケーリングについては現在検証中である．Plasma Simulator では16384コアまで，HELIOS では32768コアまでのスケーリングを検証しているが，特に顕著な飽和は観測されていない．

標準的なジャイロ運動論的PICコードのアルゴリズムに加えて先進的アルゴリズムの開発研究も行った．標準的なジャイロ運動論的 PIC コードは，大規模・高ベータのプラズマを取り扱う場合，電流密度から磁場を求める際に大きな誤差が生じることが知られている（「キャンセレーション」の問題）．この問題の解決法として拡張 split-weight-scheme があるが，我々は場の量の計算に渦方程式と磁力線方向の一般化オームの法則を用いる方法を提唱し，この方法による新しい2D-Gpic-MHDを作成した．また，新しい2D-Gpic-MHD を用いて大規模・高ベータの領域でも精度

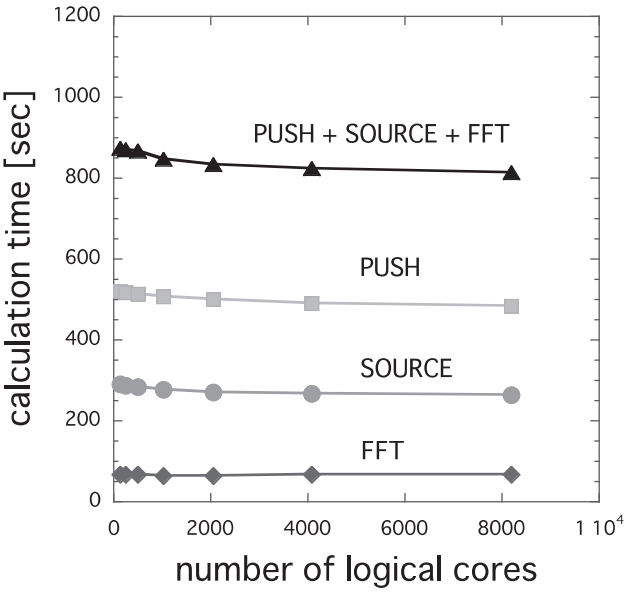

図7　通信に無関係な部分の計算時間の論理コア数依存性．
（文献[9]Fig.3 より転載）

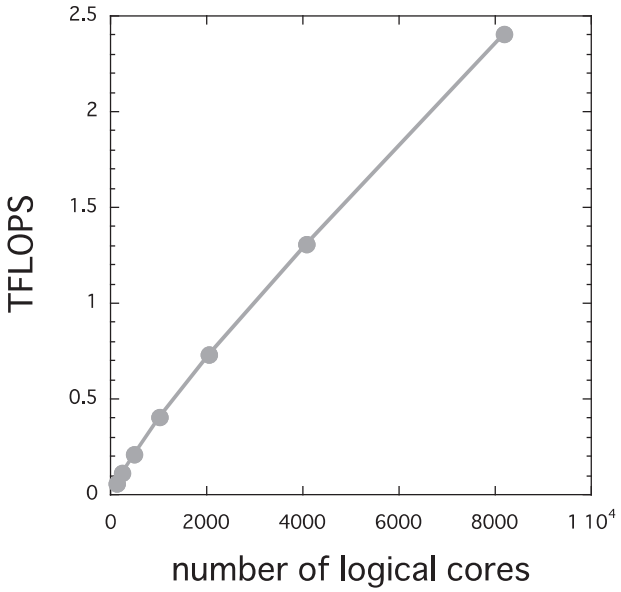

図6　FLOPS 値の論理コア数依存性．
（文献[9]Fig.2 より転載）．

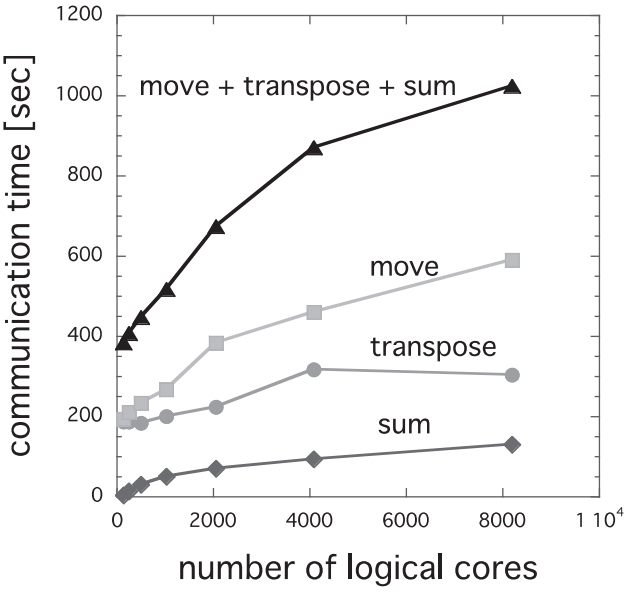

図8　通信に関係する部分の計算時間の論理コア数依存性．
（文献[9]Fig.4 より転載）

よく $m = 1/n = 1$ の運動論的内部キンクモードがシミュレーションできることを実証した[11].

今後のスケジュールとしては，新しいアルゴリズムを採用した3 D-Gpic-MHD の作成とトロイダル版3 D-Gpic-MHD の作成がある．

5.2.4 PIC コードの並列化のまとめ

5.2節では PIC コードの並列化の手順について段階的に解説するとともに，Gpic-MHD コードの高並列コンピュータでの並列化の現況について報告した．Gpic-MHD の並列化に関しては，(1)プロセス並列における2次元の領域分割の場合の最適化と(2)スレッド並列のさらなる最適化が課題として残された．Gpic-MHD コードの並列化では，Plasma Simulator で自動並列化コンパイラが利用できたため，特別なアルゴリズムは採用しなかった．高度のチューニングが可能な OpenMP を利用し，プロセス内のスレッド間での粒子分割や領域分割を取り入れた並列化は今後の課題である．

超並列計算になればなるほど，個々のコンピュータに対する最適化が必要になる．異なるコンピュータの個性もあるので，高度の並列化はきわめて専門色の強いものになる．計算物理の研究者は機種依存性の少ない，物理モデルと並列アルゴリズムの開発に専念して，機種依存性の強い最適化は，専門家にまかせるのが賢明であると思われる．

なお，次世代の高並列コンピュータでは，GPU のような計算アクセラレータが搭載される可能性が高い．マルチコアやメニーコアに対するスレッド並列化に加えて，計算アクセラレータによるスレッド並列化のための準備も必要と感じる．

PIC コードの並列化については，プラズマ・核融合学会主催の第22回専門講習会「プラズマ・核融合分野での計算機シミュレーション技法とその応用」(2009年12月16日，東京大学山上会館）で「PC クラスタの原理と基礎」という題目で講演した．特に2次元粒子コードについて解説し，プログラムの一部も掲載しているので参考にされたい[12]．また，2次元の静電近似 PIC コードの非並列版と粒子並列化版のコードは学会 HP[13]からダウンロードできるので参考にされたい．

5.3 モンテカルロシミュレーション

一口にモンテカルロシミュレーションと言っても，そこには広範囲な研究対象と実装方法がある．何らかの物理現象（あるいは社会現象や生態系などでもよい）のシミュレーションをする際に，その素過程が確率論的な現象であるとみなせる場合，あるいは素過程を方程式として記述することができないか困難である場合に，ある確率分布を持って振る舞うモデルとして近似できる場合に用いられるのがモンテカルロ法である．さらに，ある型の偏微分方程式の境界値問題や，複雑な境界形状をもつ積分区間での数値積分の技法として用いられることもある．一般的に，モンテカルロ法を用いた数値解法は扱う問題が多次元化してもアルゴリズムを変える必要がなく，収束速度の問題を別にすれば決定論的なアルゴリズムに比べて多次元化が容易であるという利点をもつ．また，扱う問題に系からのフィードバックのない場合（例えば時間的に固定された場の中でのテスト粒子の拡散問題や，モンテカルロ積分法など），解いている問題を独立事象の和とみなすことができ，非常に並列化効率の高い計算が実行可能である．

本節で取り上げるのは5.1節で紹介したモンテカルロ法のプラズマ・核融合分野への応用例のうち，特にプラズマ中の輸送現象をドリフト運動論あるいはジャイロ運動論モデルで，かつ粒子シミュレーションとして扱う事例を念頭においた並列化コーディング技法であるが，乱数の並列化に関して（5.3.1）はテスト粒子的モンテカルロ法や熱浴モンテカルロ法，モンテカルロ積分法など，統計的サンプリングを多ノード数で並列実行する場合にも役に立つであろう．またランダムアクセスの抑制（5.3.2）や MPI_REDUCE_SCATTER を用いた並列化の応用例（5.3.3）については，PIC 法など他の粒子コードを記述する上でもヒントになると思われる．

5.3.1 乱数の並列化

粒子シミュレーションモデルのドリフト運動論やジャイロ運動論におけるモンテカルロ法とは，フォッカー・プランク方程式の形で記述されたクーロン相互作用による2体衝突項を乱数によって模擬することであり，具体的にはマーカー粒子の速度 v を，ある確率密度分布 $p(\Delta v, x, v, t)$ に従うランダムな Δv だけ変位させることである．ここで，p は一般的に各マーカーの位置 x と速度 v，および時刻 t におけるプラズマの背景パラメータ（密度，温度など）に依存する．クーロン衝突項をモンテカルロ法で模擬する具体的な方法については[14-16]を参照するとよい．

さて，このようなモンテカルロシミュレーションを並列計算機で実行する場合，乱数をどのように生成すればよいか考えよう．今，MPI 並列数が nmpi で，1MPI プロセスあたり np 個のマーカーを用い，5.2節の PIC 法と同様に場の量のレプリカを nmpi 個使った並列化を行うとする（総マーカー数は nmpi×np 個）．乱数は衝突項の計算で Δv を与える場合だけでなく，マーカーの初期分布をある種の統計分布（マクスウェル分布など）に従って発生させる場合にも必要となる．任意の確率密度分布 p に従う Δv は，[0,1]区間の一様乱数をもとに直接法，棄却法，合成法[17] などを用いて生成することができるから，各 MPI プロセスで必要なのは[0,1]区間で発生させた np 個の乱数である．ただし，半導体の熱雑音などを利用した物理乱数[18] ではない，数値計算ライブラリなどに用意されている乱数は正確には疑似乱数と呼ばれるものであり，一般的にある種の漸化式を解くことによって統計的に十分乱数のように振る舞う*1数列を決定論的プロセスから生成するものである．数値ライブラリ毎に形は異なるが，非並列コードの場合以下のような手順で呼び出される．

＊1 疑似乱数がどれだけ「乱数っぽい」のか，またそれをどう検定するのという議論は，それだけでかなりの研究がなされている奥の深い研究テーマである．興味のある読者は[17,19,20]やそこに紹介されている文献を参照することをお勧めする．

（サンプルコード 5.A）

```
integer :: i,iwk(nwk),id,np
real :: rnd(np),dv(np),v(np)
id=123456
call init_rnd(iwk,id)
                    ! 乱数発生ルーチンの初期化
call gen_rand01(rnd,iwk,np)
          ! [0,1]区間の乱数列を配列 rnd に入れる
call rnd_to_dv(rnd,dv,np)
     ! rnd を用いてある確率分布に従う dv を求める
do i=1,np
  v(i)=v(i)+dv(i)
end do
```

ここで，iwk は乱数発生のための作業配列であり，その大きさ nwk は利用する疑似乱数ルーチンに依存する．iwk は疑似乱数を計算するのに必要な漸化式の履歴が保存されている「状態行列」であり，漸化式の続きを計算するためにその値を保持しておく必要がある．init_rnd は iwk の初期値を与えるもので，計算の最初に 1 回呼び出せばよい．また，途中で止めた計算を再開する際には，init_rnd を呼ぶ代わりに前回終了時にファイルに記録しておいた iwk を読み込めばよい．init_rnd の引数 id の値は乱数の初期値を決める．ここで注意すべき点は，疑似乱数列はそれぞれ固有の周期長をもっており，idはその数列のどこから乱数を読み始めるかを指定しているに過ぎないということである．衝突による速度変化 Δv を乱数で模擬するためには，疑似乱数列 rnd が相関を持たない白色ノイズのように振る舞う必要がある．実際には決定論的に生成されているので完全に白色ノイズというわけではないが，メジャーな数値ライブラリで使われている周期長の長い（現在では 2^{100} 以上のものが一般的である）に関しては，疑似乱数列全体に渡って統計的によく乱数のように振る舞うと考えてよい．ただし，だからといって次のようにモンテカルロコードを MPI 並列化することは間違いである．

1．乱数列初期化の id を，MPI プロセスごとに適当にずらす．例：id=myrank*100+1
2．他の乱数ルーチンを用いて，各 MPI プロセス毎に id を乱数で決める．
3．1．や 2．だけでは心配なので，そのように各 MPI プロセスでバラバラの id の値を使って初期化した 2 つの乱数ルーチンの平均値を採用する．

なぜならば，単に id の値を変えるということは乱数列の読み取りの始点を動かすことに過ぎず，ある MPI プロセスで j 番目に読まれた乱数は他のプロセスにおいて j+k 番目に読まれる値と同一であるからである．もし k の値が偶然小さかった場合，この 2 つの MPI プロセスには強い相関関係があるということになり，MPI 並列化されたコードの各プロセスで計算される衝突項が独立事象であることが保障されないのである．「周期長が長い乱数列だから id をプロセス毎に大きく離せば重ならないのでは」という考え方も間違っている．なぜなら，一般的に乱数の初期化ルーチンは引数 id を基に内部で別の乱数ルーチンを使って iwk の初期値を決めており，idの値に大きな違いがあっても乱数列の始点位置が大きく離れることは保証されないからである．実際上は，このような間違ったやり方で乱数発生を並列化しても目立った問題が起こらないかも知れない．しかしそれはたまたまあってはならない相関関係が目立たずにシミュレーションが実行できたというだけで，並列数 nmpi を増やした時に常に正しく動く保証はない．

それでは，並列コードの中で乱数はどのように呼び出せばよいかというと，次のような方法が挙げられる．

1．単一の乱数列を，MPI プロセス毎に重複しない区間を選んで利用する．
2．互いに独立であるという保証のある乱数列を，nmpi 個用意して利用する．
3．物理乱数を利用する．

まず，1．については前提として周期長の非常に長い疑似乱数ルーチンが必要であり，その実装の仕方として 2 通りある．1 つ目は，各 MPI プロセスが同一の乱数列から互いに重ならないように飛び飛びに読んでいくもので，以下のようになる．（サンプルでは mpi_init，mpi_rank など MPI コードの実行に必要な部分は省略してある．）

（サンプルコード 5.B）

```
integer :: i,iwk(nwk),id,np,myrank,nmpi
                  ! myrank=[0,1,2,…,nmpi-1]
real :: rnd0(nmpi*np),rnd(np),dv(np),v(np)
id=123456
       ! 注：この値は各 rank で共通のものを使う．
call init_rnd(iwk,id)
call gen_rand01(rnd0,iwk,nmpi*np)
do i=1,np
  rnd(i)= rnd(i)=rnd0(np*myrank+i) !
end do
call rnd_to_dv(rnd,dv,np)
        ⋮
```

注意点として，id と np の値は各 rank で同一にしなければいけない．**図 9** にこの例の模式図を示す．ただしこの方法は各 rank で必要な乱数より余分な乱数列を発生させていることになり，乱数生成にかかる時間は非並列時の nmpi 倍になる．並列数が多くなった場合，どんなに乱数生成ルーチンが速くても計算コストが無視できなくなる可能

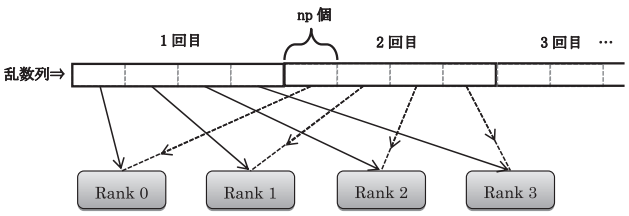

図 9　サンプルコード 5.B における MPI 並列コードにおける乱数の並列化例．nmpi=4 の場合を示している．

性がある．逆にいえば，マーカー初期分布の生成を並列化する場合など開始時に1回使うだけなら，この方法で十分かもしれない．

重複しないように乱数列を読むもう一つの方法は，あらかじめ非常に遠く離れた位置の状態行列 iwk の値を nmpi 通り用意しておき，それをシミュレーションの開始時に読み込むことである．**図10**に模式図を示す．今，各 MPI プロセスで np 個のマーカーを用いたモンテカルロ計算で，充分多く見積もって nmax 回衝突計算などに乱数を呼び出す可能性があるとする．この場合，各 rank の乱数列の呼び出しの始点を np*nmax ずつずらしておけば，シミュレーションの中でお互いに重複する乱数列を使う事は起こらず，互いに独立な計算をしていることになる．そのような iwk の初期値は，次のような非 MPI コードを事前に回して求めておく．

（サンプルコード　5.C）

```
integer, parameter :: nmpi=***,np=***,&
                    &     nmax=***,nwk=***
integer :: iwk0(nwk,nmpi), iwk(nwk),&
           & i,id,k,m
real :: rnd(nmpi*np)
id=123456
call init_rnd(iwk,id)
  do m=1,nmpi
    do k=1,nmax ! 乱数発生のから回しループ
      call gen_rand01(rnd,iwk,np)
    end do
    iwk0(:,m)=iwk(:)
  end do
  write(10) iwk0
end
```

そして，メインプログラム実行時に init_rnd を呼ぶ代わりに

```
read(10) iwk0
iwk(:)=iwk0(:,myrank+1)
```

として iwk の初期値を各 rank に与えればよい．その後の乱数ルーチンの呼び出し方は非 MPI のサンプルコード5.A とまったく同じでよい．現在の計算機の性能であれば np*nmpi*nmax の値がかなり大きくても，この事前準備にかかる時間は大したものではないであろう．メインのシミュレーションにおいては乱数生成にかかる時間は非並列の場合と変わらず，理想的な並列実行性能向上が見込める．もし同じパラメータのシミュレーションを（統計性を見るために）乱数だけ変えて複数回試行したい場合は，シミュレーションの最初に十分長い乱数発生の「から回し」を，試行ごとに長さを変えて行えばよい．

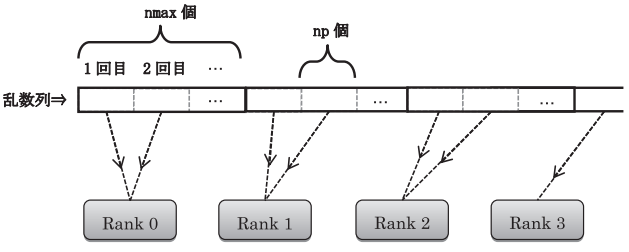

図10　サンプルコード 5.C における MPI 並列コードにおける乱数の並列化例．

疑似乱数ルーチンは漸化式を解いているため，$N \gg 1$ 回先の状態行列を求めるには基本的には 5.C のように N 回実際にから回ししてやる必要があるが，最近の疑似乱数生成法の中には1つずつ逐次実行せずともある整数 n 回分先の状態行列にジャンプできるものがある．そのような疑似乱数ルーチンを利用できる環境であれば，5.C のから回しループを置き換えることができるし，またその実行速度が十分速いようであれば 5.C に相当する部分をシミュレーションコードの先頭に組み込んで，計算開始時に各ランク毎に十分離れた乱数列の始点を決めればよい．

次に，並列化の方法 2．で挙げた，独立性の保証された並列乱数列の利用として，疑似乱数ルーチン Mersenne Twister（MT）[19] とその並列化のための Dynamic Ceartor（DC）を紹介する．これらは松本眞氏（東大数理科学研究科）によって開発されたもので，ソースコードがオープンになっているので利用したい方はぜひ参照してほしい [21]．Mersenne Twister は単体の疑似乱数としても良好な統計的性質と非常に長い周期長などの特長を持っている．Dynamic Creator は大まかに説明すると，この MT を規定する漸化式の係数を，互いの乱数列が相関をもたないように複数組生成してくれるものである．すなわち nmpi 並列でモンテカルロシミュレーションする際に，nmpi 個の「異なる乱数列表」を用意してくれるというわけである．公開されている DC のソースコードは C 言語で書かれているため FORTRAN ユーザーにはわかりにくいかもしれないが，事前準備として get_mt_parameter_id に異なる整数値 id を入力として与えて得られた構造体 mt_struct の3つ組（mts->aaa，mts->maskB，mts->maskC）の値をファイルに記録しておく．MPI コード内で MT を並列に利用する際にはそのファイルを読み込み，MPI ランク毎に異なる（aaa, maskB, maskC）の組を MT サブルーチン内のパラメータとして与えればよい（ネット上に公開されている作者の MT のソースコード genmtrand.c を参照せよ*2）．ただしこの DC は実行に時間がかかるので，数千並列を超える計算の準備にはそれなりの日数が必要となるだろう．なお，最近 MT の状態行列を 2^n 回分ジャンプさせる手法が発表されたので，これを用いて1．の方法で MT の並列化を実現するのもよい．それに関する情報やサンプルコードが公開されている [22]．

最後に，3．の手法については物理乱数ボードが組み込

＊2　[21] のホームページ中に，有志の方が作られた Fortran90の MT ソースコードも公開されている．ただし，DC で生成した3つ組変数を利用する際，Fortran90サンプルコード中にある TSHFTU(y)=ishft(y,-11) は DC のソースコードで使われている shift_00(=12) の値に合わせ，TSHFTU(y)=ishft(y,-12) に修正する必要がある．

まれた計算機が利用できる環境がないと使えないので，詳しい説明は省く．物理乱数の利点は，どんな疑似乱数よりも乱数らしい乱数である，という点に尽きる．その一方で，再現性がないためコードのデバッグ用途には使えないこと，以前に比べると高速化されたとはいえ，乱数列の生成速度はまだ疑似乱数に及ばない点が挙げられる．疑似乱数の品質に不安や関心のあるユーザーは利用を試みるのもよいだろう．MPI 並列コードから利用する際は，各 MPI プロセスが走っているノードに乱数発生器が付いているか否かの対応関係が問題になり，例えば発生器を持つノードで実行されているランクが他のランクに乱数列を MPI_SCATTER するなどの工夫が必要になる．なお，統計数理研究所ではインターネットを通じていくつかの物理乱数器や上記の MT で発生させた乱数（30分毎に更新）をダウンロードできるサービスを提供している[23]．

5.3.2 粒子コードにおけるランダムアクセスの抑制

この節では一旦 MPI による分散並列の話は置いて，1ノード内の並列メモリ上でのスレッド並列化に目を向けよう．粒子コードの場合，2.3章で説明したように OpenMP 指示行を用いてマーカーのインデックスに関するループ（例：サンプルコード 5.A 中の do i＝1,np）をスレッド並列化するのが最も容易だ．しかし，マーカーは系の中をバラバラに走り回るので，その位置はインデックス i に関して不連続になる．モンテカルロ法や PIC では系を微小区間に切り，各区間にいるマーカーのアンサンブル平均量（密度や平均速度など）を計算することが多い．例えば，マーカーの位置 x が $0 \le x < xb$ の範囲にある場合，これを nx 個の区間に分割して平均速度 u を求める計算は次のようになる．

（サンプルコード 5.D）

```
  dx=xb/nx; u(:)=0; nb(:)=0
!$OMP PARALLEL DO reduction(+:u,nb)
  do i=1,np
    k=int(x(i)/dx)+1
    u(k)=u(k)+v(i)
    nb(k)=nb(k)+1
  end do
!$OMP END PARALLEL DO
  ! ここで各スレッドの部分和が u,nb に集められる．
  u(:)=u(:)/nb(:)
```

これで u と nb に関する総和演算がマーカーインデックス i に関してスレッド並列化される．各スレッドはそれぞれが一時的に u と nb の部分和を記憶するための作業配列を用意し，do ループの終了後に全スレッド分の総和を集計する．しかし多次元領域分割の場合や分割数が多いと，キャッシュに作業配列が乗り切らなくなったり，最後に各スレッドの部分和を u や nb にかき集める作業に時間がかかったりして並列実行性能が出ない場合がある．また，各マーカーがどの区間に入るかを示す k の値が，i に対して不連続に変化することも，このような区間ごとの統計量の計算を遅くする要因である．このような場合の対処法として，いくつかの方法が考えられる．なお，ここでは単純化のため 1 次元空間の分割を考えているが，5.3.3節のコード 5.H のように多次元空間における空間セルインデックスの 1 次元化を行えば，本節で示す考え方は5.2.1節で解説したような PIC 法における SOURCE プロセスのスレッド並列にも応用可能である．

まず一つ目は，各区間 k に入るマーカーのインデックスと総数を予め調べて記録しておく方法である．

（サンプルコード 5.E）

```
  nb(:)=0
  do i=1,np
    k=int(x(i)/dx)+1
    nb(k)=nb(k)+1
    ib(nb(k),k)=i
! 区間 k に入る nb(k) 番目のマーカーのインデックス
  end do
!$OMP PARALLEL DO
  do k=1,nx
    u(k)=0
    do j=1, nb(k)
      i= ib(j,k)
      u(k)=u(k)+v(i)
    end do
    u(k)=u(k)/nb(k)
  end do
!$OMP END PARALLEL DO
```

このようにすると 2 番目のループは区間 k に対するスレッド並列になり，コード 5.D より効率がよくなると期待できる．ただし，各区間に入るマーカー数 nb(k) がなるべく均等であることが望ましい．ところで，1 番目のループは順序に依存するので，このままではスレッド並列化できない．このようなループも並列化するには，以下のような明示的な並列化が必要である．

（サンプルコード 5.F）

```
  integer, allocatable :: kst(:),ked(:)
  nomp=OMP_GET_MAX_THREADS()
                  !スレッド並列数を取得する関数
  allocate(kst(0:nomp-1),ked(0:nomp-1))
  do ith=0,nomp-1
    kst(ith)=ith*nx/nomp+1
    ked(ith)=(ith+1)*nx/nomp
  end do
  nb(:)=0
!$OMP PARALLEL DO private(ith,i,k) &
&  shared(nb,ib)
  do ith=0,nomp-1
    do i=1,np
      k=int(x(i)/dx)+1
      if((k<kst(ith)).or.(k>ked(ith))&
      &cycle
```

```
        nb(k)=nb(k)+1
        ib(nb(k),k)=i
      end do
    end do
!$OMP END PARALLEL DO
```

この例では，ith のループに対してスレッド並列化される．第 ith スレッドは kst(ith)<=k<=ked(ith) の区間に入るマーカーの数とインデックスだけをみて，その情報を共有メモリ上の配列 nb，ib の各スレッドが担当している区間に直接書き込む（shared 指定のため）．担当区間に重複がないので，配列の同じ要素に複数のスレッドが同時に書き込むといったエラーは起きない．

上のようにローカル区間ごとのマーカーのインデックスを調べる方法は，そのインデックスを何回も使いまわせる場合にはメリットが大きいが，もし知りたい統計量がコード5.Dのように１つだけなら，サンプル5.Fの$OMP PARALLEL DO 以下を次のように書いてもよい．

（サンプルコード　5.F'）

```
  nb(:)=0; u(:)=0
!$OMP PARALLEL DO private(ith,i,k) &
   & shared (u,nb)
  do ith=0,nomp-1
    do i=1,np
      k=int(x(i)/dx)+1
      if((k<kst(ith)).or.(k>ked(ith))&
      & cycle
      nb(k)=nb(k)+1
      u(k)=u(k)+v(i)
    end do
  end do
!$OMP END PARALLEL DO
  u(:)=u(:)/nb(:)
```

この他にも総和演算に対する明示的なスレッド並列化の方法として，複数の統計量（u の他に v の n 乗モーメントの和など）を求める必要がある場合，スレッドごとに担当する統計量を明示的に割り振ってしまうという方法もある（例：ith=0 なら v の和を u に，ith=1 なら v**2 の和を q に，…）．これはスレッド並列数をある固定値で実行することがプログラム作成時にわかっている場合にとられる戦略である．どのようにスレッド並列した場合に最も速いかは，np や nx の大きさ，スレッド数 ith，またコンピュータのキャッシュサイズやアーキテクチャにも依存するので，各自様々なパターンで試してみてほしい．

ここまでは配列への書き込みがあるランダムアクセスのループに関しての話であったが，モンテカルロコードでは各マーカー位置における場の量（磁場，電場，温度など）を軌道追跡しながら読み込む必要がある．バラバラに動いているマーカーのインデックス順にマーカー位置の場のデータにアクセスするとベクトル計算機の場合はバンク衝突，キャッシュの小さいスカラ計算機だとキャッシュミスを起こす恐れがある．このようなケースはどうチューニングすればよいであろうか．トーラスプラズマの新古典輸送コードを一例として取り上げる．

新古典輸送コードでは，磁気座標系 (r,θ,ζ) で与えられた三次元磁場中の案内中心軌道を解くために，i 番目のマーカー位置 $(r,\theta,\zeta)_i$ における磁場の値 B_i やその微分値が必要である．ところでトーラスプラズマ中の荷電粒子は磁力線に沿って速く動くが，磁気面を横切る r 方向のドリフト運動は比較的遅い．したがって，磁場データを (r,θ,ζ) の３次元メッシュデータではなく，r 方向にスプライン展開，(θ,ζ) 方向にはフーリエ級数展開した次のような形で与える（注：sin 成分をもつ場合もある）．

$$B(r,\theta,\zeta)_i=\sum_{m=1}^{Nm} B_m(r_i-r_l)\cos[cm(m)\times\theta_i-cn(m)\times\zeta_i]$$

ただし，r 方向のメッシュ点数を Nr，r の最大値を a として，dr = a/Nr，l = int(r_i/dr)，r_l = l×dr とする．したがって $0\le ds_i=r_i-r_l<dr$ であり，$B_m(ds_i)$ はスプライン補間係数によって

$$B_m(ds_i)=c0_{l,m}+c1_{l,m}*ds_i+c2_{l,m}*ds_i^{\,2}+c3_{l,m}*ds_i^{\,3}$$

の形で与えられる．この記述の利点の一つは，場の値だけでなくその (r,θ,ζ)-方向微分も容易に計算できることである．ところで，軌道を解く毎に各マーカーがその瞬間に滞在している区間 l に対応したスプライン補間テーブル $c(0-3)_{l,m}$ を参照していたのでは，$Nm\times4\times np$ 個のデータを毎回読み込むことになる．一般的に磁気面が複雑な形状をしていたり，プラズマ β 値が高くシャフラノフシフトが大きいほど磁場の (θ,ζ)-依存性を表現するのに必要なモード数 Nm を大きくとる必要がある．　データ参照を必要最低限に抑えるために，１ステップ前の各マーカー位置の l の値とそこでのスプライン係数を配列 li，ci に保存しておき，以下のようにして磁場データを参照する．

（サンプルコード　5.G）

```
integer :: li(np)
real:: r(np),dsi(np),ci(0:3,Nm,np),&
   & clmn(0:3,Nm,Nr)
do i=1,np
  l=r(i)/dr
  rl=dr*l
  dsi(i)=r(i)-rl
  if(l==li(i)) cycle
  ci(:,:,i)=clmn(:,:,l)
                    ! スプラインテーブルを参照
  li(i)=l  ! li の値を更新
end do
!
do i=1,np
  Bi(i)=0
  ds=dsi(i)
  do m=1,Nm
```

```
      bmn=ci(0,m,i)+(ci(1,m,i)+(ci(2,m,i)&
          &+ci(3,m,i)*ds)*ds)*ds
      cs=cos(cm(m)*th(i)-cn(m)*zt(i))
      Bi(i)=Bi(i)+bmn*cs
    end do
  end do
```

このようにすれば，ciの値はマーカーが1ステップ前にいたr方向メッシュliから他のメッシュに移動した場合だけ磁場データclmnを参照することになる．また，Biの計算ループにおいてもciはメモリに保存された順番に連続アクセスされる．なお実際にどの程度データ参照数が減るかは解いている問題設定に強く依存する．一般的にイオンに比べ電子の方がr方向のドリフト幅が小さく，その一方で磁力線方向への運動が速いため，より細かい時間刻みで軌道を計算する必要があるので，ここで紹介した方法は電子軌道計算においてより有益であるといえる．

余談であるが，計算機によってはcos，sin計算がベクトル型命令に置き換えられるため，三角関数だけをまとめたループとして回した方が速い場合と，そうでない場合とがある．前者の場合，csを配列cs(np)として宣言し，cs(i)=cos(cm(m)*th(i)- cn(m)*zt(i))の計算だけを独立したループとして先に回した方が速い．また，スレッド並列をiのループに関して掛ければよい並列実効性能を期待できるが，粒子数が非常に多くてnp*Nm*4の大きさのデータciがキャッシュに乗り切らない場合，上の書き方では性能を引き出せない場合がある．その場合，ciのiとmの次元を入れ替えて宣言しておき，後半のループを次のように並列実行させる．

（サンプルコード　5.G'）

```
  Bi(:)=0
!$OMP PARALLEL, private(i,m,ccm,ccn,&
   &ds,bmn,cs) shared(Bi)
  do m=1,Nm
    ccm=cm(m)
    ccn=cn(m)
!$OMP DO
    do i=1,np
      ds=dsi(i)
      bmn=ci(0,i,m)+(ci(1,i,m)&
       &+(ci(2,i,m)+ci(3,i,m)*ds)*ds)*ds
      cs=cos(ccm*th(i)-ccn*zt(i))
      Bi(i)=Bi(i)+bmn*cs
    end do
!$OMP END DO
  end do
!$OMP END PARALLEL
```

この例でもスレッド並列はiのループに対して掛けることに注意．内側ループでは，各スレッドは配列ciのうち長さ4*np/nompの部分区間を参照するだけなので，キャッシュに乗りやすい．なお!$OMP PARALLELを外側ループにつけたのは，内側ループが終わるたびに並列スレッドを閉じていてはオーバーヘッドが大きくなるので，その抑制のためである．

5.3.3 MPI_REDUCE_SCATTERを使った多次元領域分割の並列化

粒子コードにおける領域分割については5.2.2節でも取り扱ったが，ここでは少し変わった方法として，MPI_REDUCE_SCATTERを使った多次元領域分割における並列化の例を紹介しよう．まず問題の背景として，新古典輸送や微視的乱流シミュレーションは5次元位相空間$(r,\theta,\zeta,v_{\parallel},v_{\perp})$におけるプラズマ分布関数を扱う必要があることを指摘しておく．4章のVlasovシミュレーションでは，この位相空間を領域分割し，差分方程式の形で分布関数の時間発展を解いた．一方PIC法やモンテカルロ法など，粒子コードを使ったプラズマシミュレーションでは，並列化はマーカーに関して行うのがまず基本で，5次元位相空間における場の量に関してはすべてのMPIプロセスが同じレプリカをもっていることになる．今，5次元位相空間を単純化して(x，v)と書く．それぞれの次元をNx，Nv個に分割したセルに関して，そのセルに入るマーカーが持つ情報から決まる場の量を求めるには，まず各MPIプロセス内で集計をとり，次に全MPIプロセスで総和を取る，という過程が必要になる．

（サンプルコード　5.H）

```
a_tmp(:,:)=0
dx=(xmax-xmin)/Nx ; dv=(vmax-vmin)/Nv
do i=1,np
  j=int(x(i)/dx)+1
  k=int(v(i)/dv)+1
  jk=(k-1)*Nx+j   ! (j,k)2次元を1次元にする
  a_tmp(1,jk)=a_tmp(1,jk)+w(i)
  a_tmp(2,jk)=a_tmp(2,jk)+w(i)*x(i)
  a_tmp(3,jk)=a_tmp(3,jk)+w(i)*v(i)
          ……
  a_tmp(n,jk)=a_tmp(n,jk)+w(i)*……
end do
icnt=n*Nx*Nv
call MPI_ALLREDUCE(a_tmp,a,icnt, &
          &          mpi_real8,mpi_sum, &
          &          mpi_comm_world,ierr)
```

ここで，w(i)は一般にマーカーのウェイト（重み）と呼ばれるものであり，上の例では各セルに入ったマーカーのx，v，等に対するwの重み付きモーメントを計算している．必要に応じて5.3.2節で説明したスレッド並列における総和演算のチューニングを施すものとする．また，2次元位相空間のセル位置(j,k)を，変数jkを用いて1次元化している（より多次元の場合も同様にして次元を減らせる）．さて，このようにして求められたn種類のモーメント量a(1:n,jk)から何かしらの演算を行ってm種類の新しい場の量b(1:m,jk)を位相空間の各セル(j,k)に対して求める必要があるとする（例えば各セルの中で離散データ

w(i)を(x,v)に関して滑らかに補完した場 $\overline{w_{j,k}}(x,v)$ を求めるなど）．この時，aからbを求める計算量がそれなりに大きいなら，bの計算を領域分割してMPI並列実行したいと当然思うだろう．サンプルコード5.Hではすべての MPI プロセスにaの全区間分の集計結果のコピーが配信されるが，bを計算するにはaのうち各プロセスが担当する部分区間の情報だけあればよい．このようなケースに使えるのが MPI_REDUCE_SCATTER である．

まず，MPI プロセス毎に担当する位相空間の区間の始点，終点を決める．ここで総 MPI プロセス数を nproc，各 MPI プロセスの rank を myrank とする．

（サンプルコード 5.I）

```
integer, dimension(0:nproc-1) :: idispb,&
  & jkst,jked,icnta,icntb
Nxv=Nx*Nv
do mp=0,nproc-1
  if(nproc<Nxv) then
    jkst(mp)=mp*Nxv/nproc+1
    jked(mp)=(mp+1)*Nxv/nproc
    icnta(mp)=(jked(mp)-jkst(mp)+1)*n
    icntb(mp)=(jked(mp)-jkst(mp)+1)*m
  else
    if(mp<Nxv) then
      jkst(mp)=mp+1 ; jked(mp)=mp+1
      icnta(mp)=n ;icntb(mp)=m
    else
      jkst(mp)=0; jked(mp)=0; &
          & icnta(mp)=0; icntb(mp)=0
    end if
    end if
  end if
!
  idispb(0)=0
  do mp=1,nproc-1
    idispb(mp)=idispb(mp-1)+icntb(mp-1)
  end do
```

（このサンプルは Nxv<nproc で担当する区間がない rank が生じる場合にも対応している）

また，aのうち担当区間の総和を受け取る作業配列 a_loc と，担当区間におけるbの値を格納する b_loc を各 rank 毎に次のように宣言しておく．

```
real, allocatable :: a_loc(:,:),b_loc(:,:)
if(icnta(myrank).ne.0) then
  mjk=jked(myrank) - jkst(myrank)+1
  allocate(a_loc(n,mjk),b_loc(m,mjk))
end if
```

準備は以上である．各 MPI プロセスで求めた a_tmp から a_loc へ振り分けながら総和通信を行うには，コード5.Hの MPI_ALLREDUCE を次のように置き換える．

```
call MPI_REDUCE_SCATTER(a_tmp,a_loc, &
      &     icnta,mpi_real8, &
      &     mpi_sum,mpi_comm_world,ierr)
```

これで a_tmp の全 MPI プロセスの総和のうち，先頭から icnta(0)個の要素が rank=0 に，次の incta(1)個の要素が rank=1 に，…という形に振り分けられる．位相空間についてインデックス jk を用いて1次元化してあるため，どう振り分けされたかイメージが付きやすいだろう（**図11**）．次に各プロセスの担当区間で a_loc から b_loc を求めてから，すべてのプロセスに b_loc の結果を送信する．それは以下のように行われる．

```
call solve_a_to_b(a_loc,l,b_loc,m,mjk)
call MPI_ALLGATHERV(b_loc,icntb(myrank),&
      &     mpi_real8, b,icntb,idispb, &
      &     mpi_real8,mpi_comm_world,ierr)
```

これで各プロセスの b_loc の icntb（myrank）個の計算結果が，mpi_comm_world に含まれる全てのプロセス上の配列bの，先頭から idispb(myrank)+1番目～idispb(myrank)+ icntb(myrank)に収められる（**図12**）．なお，icnt[a,b](myrank)=0 のプロセスも MPI による送受信に参加しなければならないことに注意．こうしてaからbの値を領域分割して分散計算し，結果を全プロセスで共有するという目的は達せられた．

ここで取り上げた2つの集団通信には，事前にデータの分配方法が記述された配列を与えておく必要がある（サンプル中の icnta，icntb，idispb）．3章，4章で取り上げた新しい communicator を定義して通信先を指定するものに比べるとやや煩雑であるが，icnta などの計算方法を理解さえすれば様々な応用ができるだろう．なお，筆者

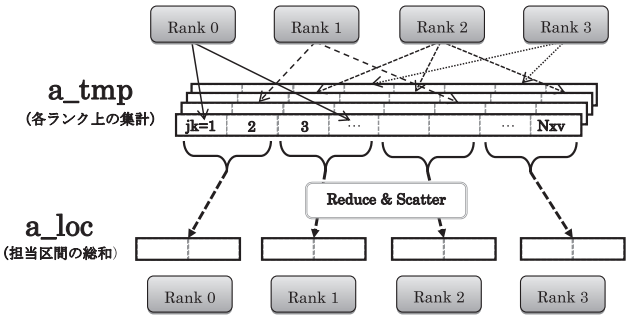

図11 MPI_REDUCE_SCATTER による総和演算データの振り分け．

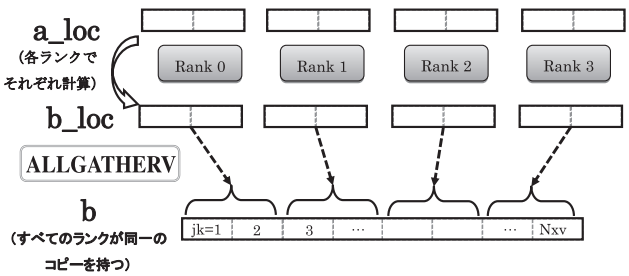

図12 MPI_ALLGATHERV による各担当区間の計算結果の通信．

がこの並列化法を取り入れている新古典輸送コードFORTEC-3D[24]では，5次元位相空間の領域分割数は10^6程度のオーダーであり，それに対してnproc＝512程度までならこの通信を使ったMPI並列での計算時間の短縮が確認できた（Plasma SimulatorおよびHELIOS上）．それ以上の並列数にしてもほとんど実行性能は向上しなかったが，多ノード間でのMPI集団通信に付随するオーバーヘッドが増える事と，一方で各プロセス上のb_locの担当区間が小さくなり，通信時間に比べ無視できるほどa_locからb_locを求める計算時間が短くなることの両方がその原因となっている．nproc>>1000のような次世代スパコンの環境でもスケーラビリティを得るためには，5.2.2節の「(4)プロセス並列化2」で紹介された領域分割とレプリカによる粒子分割を組み合わせたより高度な並列化が必要となるかもしれない．

5.4 おわりに

4回に渡って連載してきました本講座は今回で終了です．計算機の進化は日進月歩であり，大規模なシミュレーションを効率よく実行し，今までに扱えなかった現象をシミュレーションするためには計算機の進化に合わせたプログラミングの知識，技法が必要とされますが，そのような知識は市販されている教科書やシミュレーション結果に関する論文を読んだだけでは得ることができず，そのことが新しくシミュレーション研究を始めようという初心者に対する障壁になっているのではないかという危惧がありました．本講座の一番の目標はそのような方々のために，実際の大規模並列化コードがどのようなことに配慮してどう書かれているのかという実例を紹介しながら，並列化コードの最適化・高速化を図るための基礎知識を学んでいただくことにありました．本講座の内容が今後ますます盛んになっていくと予想されるスーパーコンピュータを活用したプラズマ・核融合分野のシミュレーション研究を志す方々の一助になれば幸いです．

参考文献

[1] C.K. Birdsall and C.K. Langdon, *Plasma Physics via Computer Simulation* (MacGraw-Hill, New York, 1985).
[2] 内藤裕志：プラズマ・核融合学会誌 **74**, 470 (1998).
[3] S. Briguglio *et al.,* in Recent Advances in Parallel Virtual Machine and Message Passing Interface, Lecture Notes in Computer Science 2840, 180 (Springer, 2003).
[4] V.K. Decyk and T.V. Singh, Computer Physics Communications **182**, 641 (2011).
[5] H. Naitou *et al.,* Phys. Plasmas **2**, 4257 (1995).
[6] H. Naitou *et al.,* J. Plasma Fusion Res. **72**, 259 (1996).
[7] 内藤裕志，徳田伸二：プラズマ・核融合学会誌 **72**, 737 (1996).
[8] H. Naitou *et al.,* J. Plasma Fusion Res. SERIES **8**, 1158 (2009).
[9] H. Naitou *et al.,* Progress in Nuclear Science and Technology **2**, 657 (2011).
[10] H. Naitou *et al.,* Plasma Fusion Res. **6**, 2401084 (2011).
[11] H. Naitou *et al.,* Plasma Sci. Technol. **13**, 528 (2011).
[12] http://www.jspf.or.jp/conference/specialist/22prog.html
[13] http://www.jspf.or.jp/journal/8904koza.html
[14] X.Q. Xu and M.N. Rosenbluth, Phys. Fluids B **3**, 627 (1991).
[15] Z. Lin *et al.,* Phys. Plasmas **2**, 2975 (1995).
[16] A.H. Boozer and G. Kuo-Petravic, Phys. Fluids **24**, 851 (1981).
[17] 津田孝夫：モンテカルロ法とシミュレーション（三訂版）（培風館，1995）．
[18] 田村義保 他：日本統計学会誌 **35**, 201 (2006).
[19] 松本 眞：日本統計学会誌 **35**, 165 (2006).
[20] D.E. Knuth，有沢 誠 他訳：The Art of Computer Programming (2) Seminumerical algorithms 日本語版（アスキー，2004）．
[21] http://www.math.sci.hiroshima-u.ac.jp/m-mat/MT/mt.html
[22] http://theo.phys.sci.hiroshima-u.ac.jp/ishikawa/PRNG/mt_stream.html
[23] http://random.ism.ac.jp/random/index.php
[24] S. Satake *et al.,* Plasma Fusion Res. **3**, S1062 (2008).

内藤裕志（ないとうひろし）

山口大学大学院理工学研究科・教授．気がつけば，還暦を過ぎてしまいました．「少年老い易く学なりがたし」．年齢に抵抗して頑張っています．趣味はハングルの勉強と韓国ドラマの鑑賞です．ジャイロ運動論的粒子シミュレーションとジャイロ簡約MHDシミュレーションの両方からプラズマのMHD現象を解明するのが研究テーマです．

佐竹真介（さたけしんすけ）

核融合科学研究所核融合理論シミュレーション研究系准教授．専門は磁場閉じ込めプラズマ中の新古典輸送や両極性径電場，新古典粘性等のシミュレーション研究．趣味はドライブとサッカー観戦，ギター演奏．最近の海外での日本人サッカー選手の活躍を見て，研究者も世界を相手に挑戦し，渡り合わなければと励まされる気持ちです．