3. PC クラスタの原理と基礎

山口大学大学院理工学研究科　内藤裕志

本章では、粒子シミュレーションを例にして、PC クラスタや超並列コンピュータでのシミュレーションコードの並列化について解説する。現代の高度計算科学の一端を感じていただければ幸いである。3.1 節では、PC クラスタの構成例と基本的考え方について述べる。3.2 節では、2 次元の粒子コードを例にして、並列化の実際について解説する。超並列コンピュータへの適用については 3.3 節で述べる。参考文献は 3.4 にまとめた。

3.1 PC クラスタの構成例と基本的考え方

コンピュータの計算能力の発展はすばらしい。筆者が修士課程の学生の頃(３０年程前)、電子ビームとプラズマの相互作用の粒子シミュレーションを研究テーマとしていたが、その頃大学の大型計算センターにあった最先端のコンピュータでしか実行できなかった計算も、今や 1 台のノート PC で軽々と計算できる。一方、コンピュータの進化とともに科学技術計算のアルゴリズムの発展も著しいものがある。両者の相乗効果により従来は考えも及ばなかったような計算も可能になりつつある。例えば太陽と地球の相互作用の“まるごと”シミュレーションとか、トカマクの放電全体の“まるごと”シミュレーションとかがある。人体の“まるごと”シミュレーションというのも想定されているようである。

最近のコンピュータのハード的発展は主に並列化によるものである。１個のプロセッサの高速化には限界があるが、数個から数十万個のプロセッサを同時に働かせることにより格段の高速化が得られる。プロセッサ間の通信も十分高速化される必要があるのはいうまでもない。1 チップの上に複数の CPU を載せる技術も現実のものになった。デュアルコアやクワッドコアを始めとするマルチコア化が流れとしてある。また 1 個の物理コアを複数の論理コアとして働かせるマルチスレッド（MT: Multi-Threading、intel 社では HT: Hyper-Threading と呼んでいる ）技術も開発されている。以下では用語として CPU のかわりにコアを用いることにする。

並列コンピュータには分散メモリー型と共有メモリー型がある。Fig.1 に示すように分散メモリー型では、それぞれのコアがメモリーを持ち、ネットワークでつながれている。一方、共有メモリー型では、Fig.2 に示すように全てのコア

が共通のメモリーを持っている。どのコアも共有メモリーに対して対称な関係にあるときは対称型マルチプロセシング(SMP: Symmetric Multiprocessing)と呼ばれる。最近は、Fig.3 に示すような、ハイブリッド型が主流になりつつあるようである。ハイブリッド型では、いくつかのコアでメモリーを共有するものを 1 ノードとし、多数のノードをネットワークで分散メモリー的に結合している。

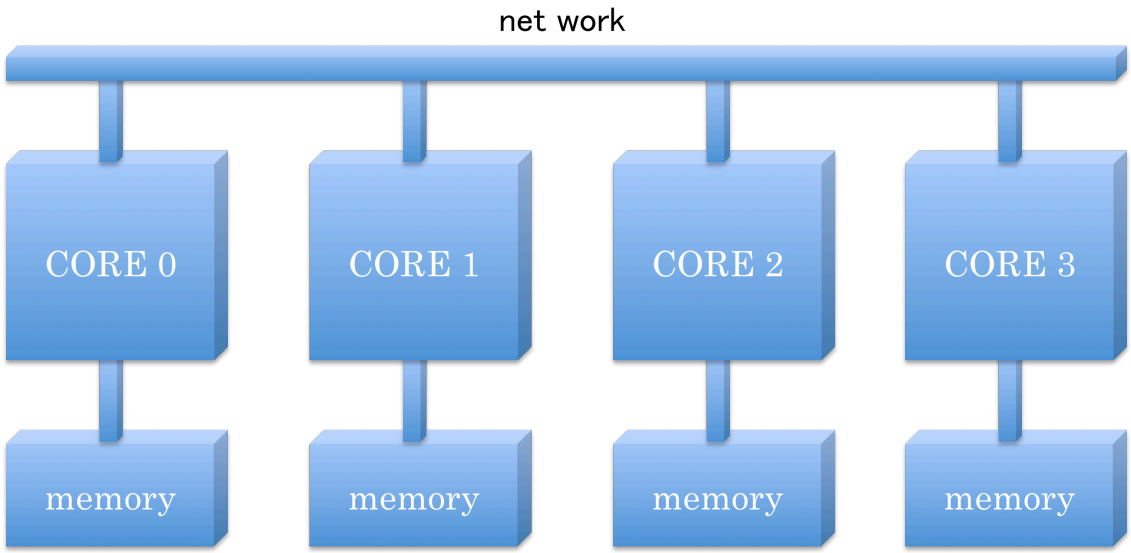

Fig.1 Distributed Memory

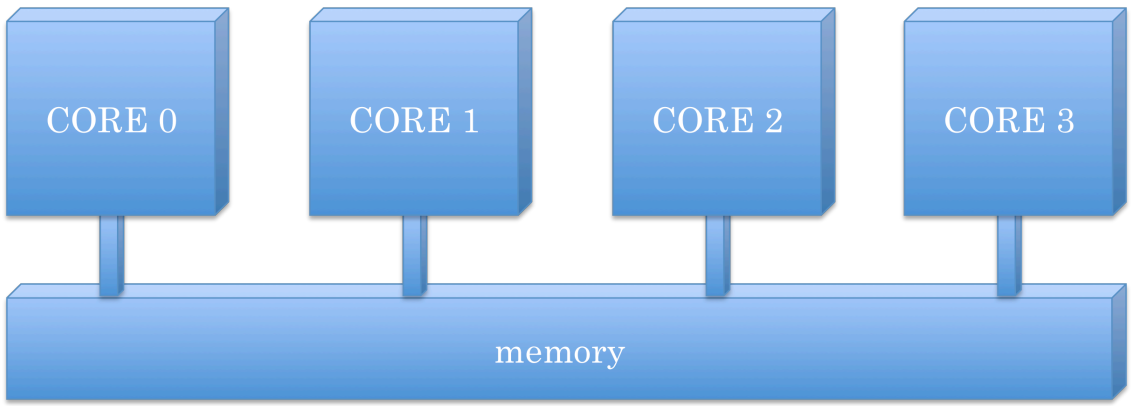

Fig.2 Shared Memory

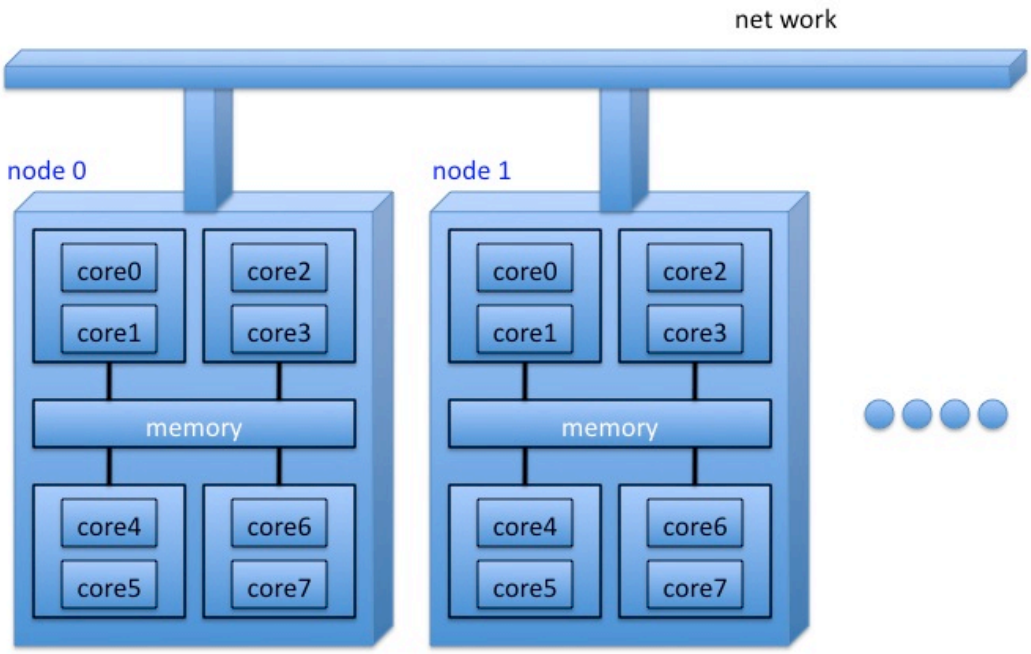

Fig.3 Hybrid System of Distributed Memory and Shared Memory

Fig.4 に分散メモリー型の PC クラスタの例を示す。著者の研究室で２００３年頃に制作したものである。２CPU の PC を４台接続した構成になっている。

Fig.4 Example of PC Cluster (4PC×2CPU)

インターネットを利用して多数の PC またはコンピュータシステムを結合し、巨大計算を実行する場合もある。これらはグリッドと呼ばれている。グリッドとは電力系統を意味するパワーグリッドを意識して作られた言葉で、電気器具をコンセントに差し込めば、いつでも電力系統から電気を取り出せるように、ネットワークにつなげばネットワーク上の計算資源を自由に取り出せるシステムをイメージしている。PC クラスタや超並列コンピュータもグリッドの一種と考えることができる。ここでは、インターネットを介さない孤立型のシステムのみを取り扱う。

並列コンピュータを用いて並列計算を行うためには並列プログラミングを行う必要がある。プログラマーが意識しないでもコンパイラが自動的に並列化してくれれば理想的であるが、現実はそうではない。逆にプログラマーの存在意義がある。プログラミングモデルとしては、分散メモリーを想定した MPI(Message Passing Interface)モデルと、共有メモリーを想定したモデルがある。

分散メモリーを想定すると、あるコアが処理したいデータが別のコアに所属する場合が一般的である。この場合データを異なったコア間でネットワークを介して交換する必要がある。MPI モデルとはコア間のメッセージのやり取り（パス）を基本としたプログラミングモデルである。言い換えるとデータの送信(send)と受信(receive)を基本としたプログラミングモデルになっている。実際には並列計算用の複数の関数のセットが供給されていて、通常の FORTRAN や C のプログラムと組み合わせて使用できる。MPI モデルによる並列化は、プロセス並列のプログラミングモデルの一つである。プロセス並列では、各プロセスが独立して動いていて、必要なときにプロセス間通信を行うと考えるとわかりやすい。

共有メモリーを想定したプログラミングモデルには自動並列化と OpenMP がある。これらはスレッド並列のプログラミングモデルに対応している。スレッド並列では、通常 1 スレッドで動いていて、並列化可能な演算があると複数のスレッドに演算を分割して実行することになる。複雑なループに対しての自動並列化は困難で、OpenMP でプログラムする必要がある。最新のコンパイラでは、簡単なループであれば、自動並列化で十分高速化が可能なようである。ハード的に共有メモリーのところを分散メモリー型のプログラミングモデルを使用してプログラムすることも可能である。逆にハード的に分散メモリー型のところを共有メモリー型のプログマミングモデルでプログラムをすることも可能

である（この場合は、計算速度が極端に遅くなって現実的ではない)。いずれの場合もコンパイラがハードの違いを吸収することになる。プログラマーは実際のハードの構成を頭の中に入れながらプログラミングをすることになる。最新のハイブリッド型のコンピュータでは、純粋な MPI より、自動並列化（または OpenMP）と MPI を組み合わせたハイブリッドプログラミングの方が良い結果を出すようである。

ここで、GPU を用いた高速化についても述べる。GPU は、Graphic Processing Unit の略で、従来は画像処理用のプロセッサであった。最近の GPU は演算ユニットを多数（１００程度）搭載しているため、この演算ユニットを利用した科学技術計算の高速化エンジンとして使用可能になっている。GPU を利用した計算は GPGPU(General Purpose computing on GPU)と呼ばれる。NVIDIA 社により GPU を使用するプログラミング環境として CUDA(Compute Unified Device Architecture)が提供されている。OS としては、Windows、Linux、Mac OS X が対応している。従来は C 言語のみの対応であったが、最近は FORTRAN も対応するようになっている。筆者の研究室でもテスト的に使用しているが、粒子コードで１０～２０倍の高速化が得られている。直近のニュースとして、長崎大学工学部の浜田剛助教を中心とするグループが開発した３８０GPU を並列化したコンピュータが日本最高速を示し、「ゴードン・ベル賞」（価格・性能部門）を受賞した（J-CAST ニュース：2009 年 11 月 27 日 20 時 45 分配信、毎日 JP:2009 年 11 月 27 日）というものがあった。

3.2 粒子コードを例とした並列化の実際

プラズマのコンピュータ・シミュレーションの手法は、運動論的手法と流体的手法に分類される。運動論的手法は運動論的方程式系を基礎とし、流体的手法は流体的方程式系を基礎としている。簡単に説明すると、運動論的方程式系は実空間に速度空間を加えた位相空間での荷電粒子の運動を取り扱うものである。一方流体的方程式系は運動論的方程式系を速度空間で積分して求められる。このことは、「流体的方程式系は運動論的方程式系の速度空間でのモーメントをとることにより求められる」と表現される場合もある。流体的方程式系は、実空間での密度・速度・温度等の平均量（モーメント量）の時間的変化を取り扱う。導出の過程から推察されるように、速度空間の情報の多くを捨て去って、身軽なものになっている。流体的手法を用いたシミュレーションでは、粒子的手法を用いたシミュレーションと比較して、必要とされる計算資源も格段に小さい。プラズマの物理現象には、速度空間の情報に密接に関係したものと、そうでないものに分類される。実際のシミュレーションを行う場合は、シミュレーションしたい物理の内容を考慮に入れて、最適なシミュレーション手法を選択する必要がある。

運動論的手法によるシミュレーションは第一原理シミュレーションと呼ばれる。原理的に古典力学で表されるプラズマの力学系の全ての物理現象をシミュレーション可能であるが、コンピュータ資源に対する要求は非常に高いものになる。運動論的手法は、粒子的手法とブラソフ的手法に分類される。粒子的手法では、個々の荷電粒子の情報はそのまま保持される。ブラソフ的手法では、個々の荷電粒子の情報を位相空間の各点の近傍で平均化し、プラズマを位相空間での流体として取り扱う方程式系を基本とする。すこし先進的手法になるが、粒子的手法を用いてブラソフ方程式系をシミュレーション場合もある。

粒子的手法ではプラズマを構成する荷電粒子（電子とイオン）の運動を、荷電粒子自身が作る電磁場と連動させて、直接時間積分することによりプラズマの物理を解析する。現実の荷電粒子をそのまま取り扱うのは最新鋭のコンピュータを用いても不可能であるので、通常 10^6〜10^{12} 個の「荷電粒子」を取り扱う。このためコンピュータで取り扱う「荷電粒子」を超粒子と呼ぶことがある。個々の超粒子は実際の荷電粒子を多数集めて 1 個にしたものに対応している。極めて多数の超粒子からなる超多体系を取り扱うため、粒子的手法によるシミュレーションは（ブラソフ的手法の場合と同様に）計算資源に対する要求が高く、

超並列コンピュータを用いた巨大ジョブになる場合が多い。もちろん、必要最小限のシステムサイズで物理現象の特徴をうまく引き出すシミュレーションはノート PC でも可能である。一方で実際の実験装置をそのままのサイズでシミュレーションしたい場合等は超並列コンピュータの使用が前提になる。

ここでは、粒子的手法を用いた粒子コードの並列化について解説する。通常粒子コードは PIC コードと呼ばれる。PIC とは Particle-In-Cell を省略したもので、Fig.5 に示すように、粒子とセル（空間グリッドまたは空間メッシュを指す）を用いる。電場や磁場は（図中に、黒丸で示した）グリッド点上でのみ計算される。グリッド点以外の電磁場は、近辺のグリッド点の電磁場の値を内挿したものを用いる。グリッドを使用しないで直接粒子間の相互作用を計算することも可能であるが、この場合は粒子数を N とすると N×N の計算が必要になり、N が非常に大きな値であることを考えると、実際的でない。

粒子シミュレーションの基礎方程式系は、荷電粒子の運動方程式（ニュートン・ローレンツの運動方程式）と場の量を決定する方程式（電磁気学のマクスウェル方程式）からなる。簡単のため、静電近似が成立する場合の基礎方程式系を次ページに示す。静磁場も存在しないと仮定する。この場合の基礎方程式系を以下に示す。第１と第２式はニュートンの運動方程式。第３式はポアッソンの方程式で電荷密度から静電ポテンシャルを決定する。第４式は静電ポテンシャルと静電場の関係を表している。式中で添字 s は電子（s=e）とイオン（s=i）を、j(1, 2, …., N)は j 番目の粒子を、m_s と q_s はそれぞれ粒子の質量と電荷を示す。

$$\frac{d\boldsymbol{r}_{sj}}{dt} = \boldsymbol{v}_{sj}$$

$$m_s \frac{d\boldsymbol{v}_{sj}}{dt} = q_s \boldsymbol{E}(\boldsymbol{r}_{sj})$$

$$\nabla^2 \phi = -\frac{\rho}{\varepsilon_0} = -\frac{1}{\varepsilon_0}\left(\sum_s \sum_j q_s \delta(\boldsymbol{r} - \boldsymbol{r}_{sj})\right)$$

$$\boldsymbol{E} = -\nabla\phi$$

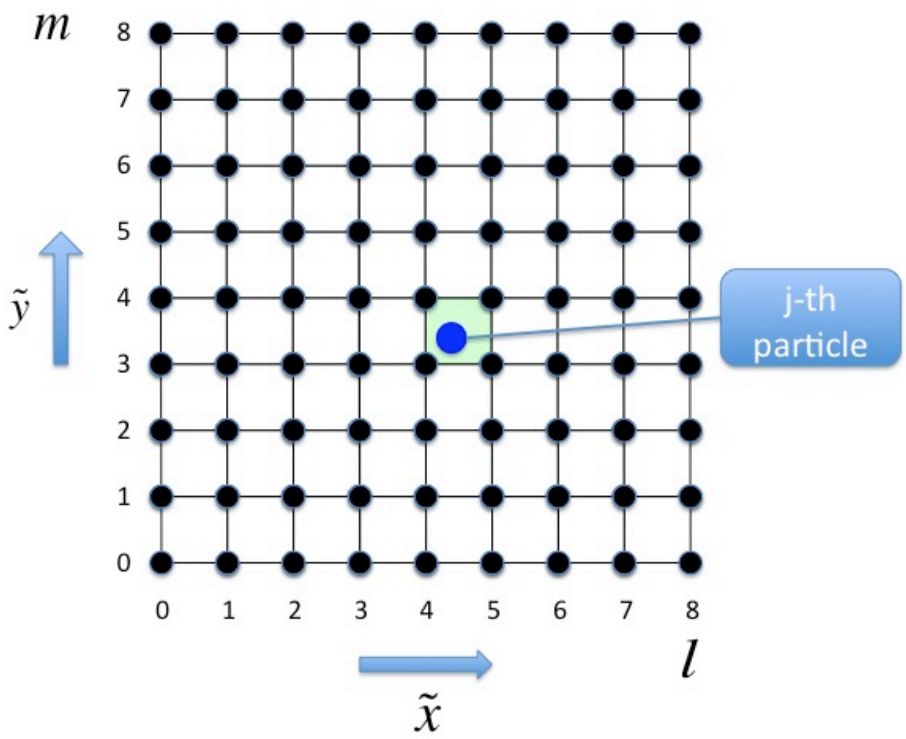

Fig.5 Spatial Grid (Mesh)

以下では、Fig.5 に示すような２次元のシステムを用いて説明する。直角座標系を考え、物理量は x、y 方向のみに変化するとする。２次元のシステムでは、物理量は z 方向には変化しないため、各粒子は、z 方向に無限の長さをもつ棒（ロッド）で表される。このため２次元の粒子モデルはロッドモデルと呼ばれる。（同様の理由で１次元の粒子モデルはシートモデルと呼ばれる。）

コンピュータ・シミュレーションでは、物理方程式系をそのままプログラムするよりは、物理方程式系を規格化したものを用いる場合が多い。規格化では、単位を持つ物理量を、無次元量で表した式に置き換える。規格化された方程式系は、単位系に依存しない、物理定数を直接表記する必要がない、方程式系のスケーリングが明白になる等のメリットがある。初期のコンピュータでは桁落ちや桁溢れの危険を回避できるというメリットもあったが、これは現在のコンピュータでは表示できる実数の範囲が広いため、特にメリットであるということはないと思われる。ここでは、空間はグリッドの幅 Δ で、時間はプラズマ振動数の逆数 ω_{pe}^{-1} $(\omega_{pe} = n_0 e^2 / \varepsilon_0 m_e)$ で規格化する。この規格化を基本に方程式系が簡単になるように静電ポテンシャルと電場の規格化を以下のように決める。

$$\tilde{x} = x/\Delta, \quad \tilde{y} = y/\Delta, \quad \tilde{t} = t\omega_{pe},$$
$$\tilde{v}_x = v_x/(\Delta\omega_{pe}), \quad \tilde{v}_y = v_y/(\Delta\omega_{pe}),$$
$$\tilde{\phi} = \phi/(m_e\Delta^2\omega_{pe}^2/e),$$
$$\tilde{E}_x = E_x/(m_e\Delta\omega_{pe}^2/e), \quad \tilde{E}_x = E_x/(m_e\Delta\omega_{pe}^2/e)$$

この規格化では、電荷と質量は以下のように表される。

$$\tilde{q}_e = -1, \quad \tilde{q}_i = q_i/e, \quad \tilde{m}_e = 1, \quad \tilde{m}_i = m_i/m_e$$

粒子の運動方程式は、例えば以下のように、時間微分を時間差分で近似すると、

$$\left.\frac{d\tilde{x}}{d\tilde{t}}\right|_{\tilde{t}+0.5\Delta\tilde{t}} \rightarrow \frac{\tilde{x}(\tilde{t}+\Delta\tilde{t})-\tilde{x}(\tilde{t})}{\Delta\tilde{t}},$$
$$\left.\frac{d\tilde{v}_x}{d\tilde{t}}\right|_{\tilde{t}} \rightarrow \frac{\tilde{v}_x(\tilde{t}+0.5\Delta\tilde{t})-\tilde{x}(\tilde{t}-0.5\Delta\tilde{t})}{\Delta\tilde{t}}$$

以下のようになる。

$$\tilde{x}_{sj}(\tilde{t}+\Delta t) = \tilde{x}_{sj}(\tilde{t}) + \tilde{v}_{xsj}(\tilde{t}+0.5\Delta\tilde{t})$$
$$\tilde{y}_{sj}(\tilde{t}+\Delta\tilde{t}) = \tilde{y}_{sj}(\tilde{t}) + \tilde{v}_{ysj}(\tilde{t}+0.5\Delta\tilde{t})$$
$$\tilde{v}_{xsj}(\tilde{t}+0.5\Delta\tilde{t}) = \tilde{v}_{xsj}(\tilde{t}-0.5\Delta\tilde{t}) + (\tilde{q}_s/\tilde{m}_s)\tilde{E}_x(\tilde{x}_{sj}(\tilde{t}),\tilde{y}_{sj}(\tilde{t}))$$
$$\tilde{v}_{ysj}(\tilde{t}+0.5\Delta\tilde{t}) = \tilde{v}_{ysj}(\tilde{t}-0.5\Delta\tilde{t}) + (\tilde{q}_s/\tilde{m}_s)\tilde{E}_y(\tilde{x}_{sj}(\tilde{t}),\tilde{y}_{sj}(\tilde{t}))$$

ここでは時間微分が中心差分になるように差分化をおこなっている。時間に関する中心差分を用いると、実際の運動法的式と同様に、差分化した運動方程式も時間反転に関して対象になる。この手法は、粒子の座標と速度は 0.5Δt だけ異なった時間で計算されるため蛙飛び法(leap-frog-method)と呼ばれている。

Fig.5 で示されるように、j 番目の粒子は特定のセルの中に存在する。このセルを拡大すると、Fig.6 のようになる。

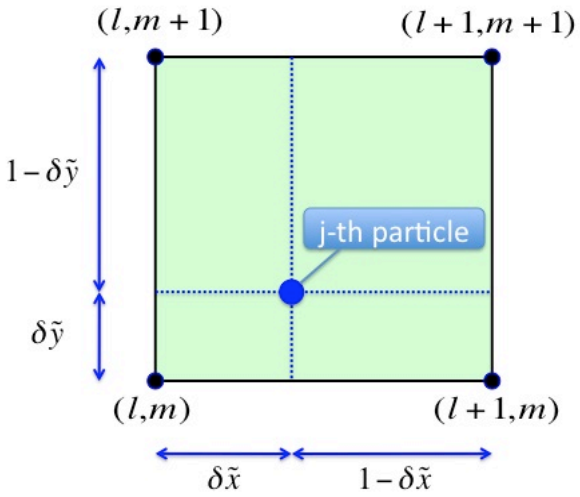

Fig.6 j-th particle in a cell

個々の粒子が感じる電場は、以下のように表記される。

$$\tilde{E}_x(x_{sj},y_{sj}) = (1-\delta\tilde{x})(1-\delta\tilde{y})\tilde{E}_x(l,m) + \delta\tilde{x}(1-\delta\tilde{y})\tilde{E}_x(l+1,m)$$
$$+(1-\delta\tilde{x})\delta\tilde{y}\ \tilde{E}_x(l,m+1) + \delta\tilde{x}\,\delta\tilde{y}\ \tilde{E}_x(l+1,m+1)$$
$$\tilde{E}_y(x_{sj},y_{sj}) = (1-\delta\tilde{x})(1-\delta\tilde{y})\tilde{E}_y(l,m) + \delta\tilde{x}(1-\delta\tilde{y})\tilde{E}_y(l+1,m)$$
$$+(1-\delta\tilde{x})\delta\tilde{y}\ \tilde{E}_y(l,m+1) + \delta\tilde{x}\,\delta\tilde{y}\ \tilde{E}_y(l+1,m+1)$$

ここで

$$\delta\tilde{x} = \tilde{x}_{sj} - l, \quad \delta\tilde{y} = \tilde{y}_{sj} - m$$
$$l \le \tilde{x}_{sj} < l+1, \quad m \le \tilde{y}_{sj} < m+1$$

である。これを Fig.7 に図示する。この方法は格子点上の電場の値を、粒子の感じる電場の値に、直線的（線形）に内挿することに対応している。

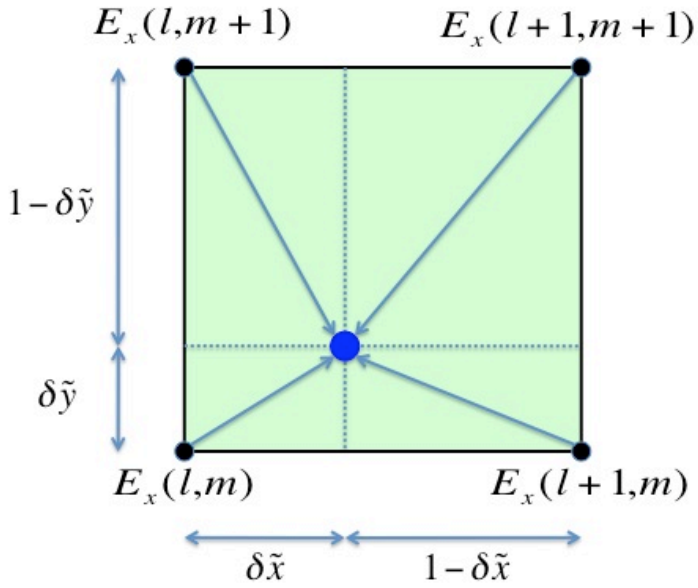

Fig.7 Interpolation of electric field in a cell

差分化された運動方程式をサブルーチン push とし、Program.1 に示す。サブルーチン push は、粒子の位置と速度を 1 時間ステップ $\Delta\tilde{t}$ だけ進める（時間積分する）ものである。このサブルーチンは、粒子シミュレーションコードの骨格の一つであり、非常に簡単な短いプログラムで表されることがわかる。ここで用いられる電場は別のサブルーチンで求める必要がある。電場は静電ポテンシャルより計算される。このサブルーチンを efield、電荷密度より静電ポテンシャルを求めるサブルーチンを poissn とする。ここでは、efield と poissn の作成は読者への宿題とする。poissn は、適当なパブリックドメインのソフトや、コンピュータ会社から供給されているサブルーチンで置き換えることも可能である。これらのサブルーチンに関して、差分化した方程式を、この節の最後の問題１・２の解答として示しているので参照されたい。

粒子コードでは格子点上の電荷密度を計算する必要がある。粒子の位置は特定のセル内にあるため、粒子の電荷を分割して格子点に与えることが必要になる。これは電荷分配（charge assignment）とよばれる作業であり、粒子シミュレーションの骨格の一つになっている。このサブルーチン名を source とし、Program.2 に示す。この場合も非常に簡単で短いサブルーチンになっていることが分かる。アルゴリズムの概要を Fig.8 に示す。

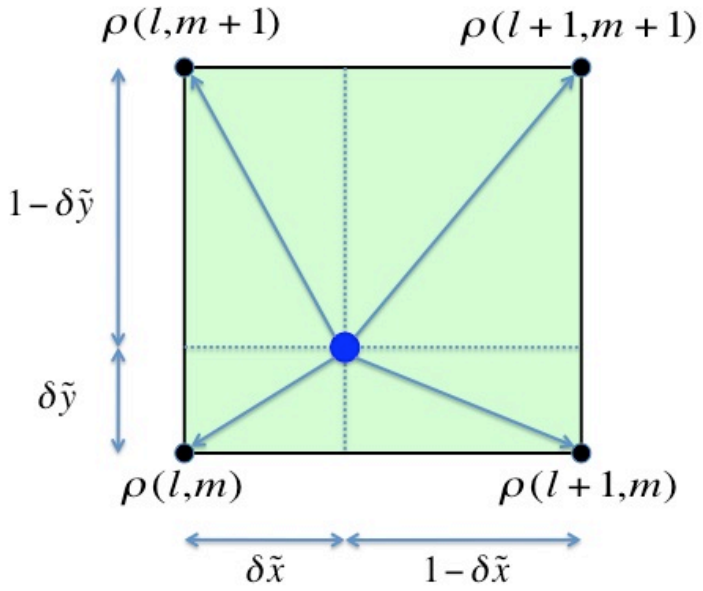

Fig.8 Assign charge from a particle to grid points

電荷の分割と格子点への割当の方法は次式で表される。

$$
\begin{aligned}
\tilde{\rho}(l,m) &\leftarrow \tilde{\rho}(l,m) + (1-\delta x)(1-\delta y)\tilde{q}_s \\
\tilde{\rho}(l+1,m) &\leftarrow \tilde{\rho}(l+1,m) + \delta x(1-\delta y)\tilde{q}_s \\
\tilde{\rho}(l,m+1) &\leftarrow \tilde{\rho}(l,m+1) + (1-\delta x)\delta y\,\tilde{q}_s \\
\tilde{\rho}(l+1,m+1) &\leftarrow \tilde{\rho}(l+1,m+1) + \delta x\,\delta y\,q_s
\end{aligned}
$$

ここで、右辺の電荷密度は、特定の粒子の電荷を分割・割当する前のもの、左辺の電荷密度は　分割・割当した後のものを示している。FORTRAN のプログラムでは上式の矢印は、＝で表される。

割当の方法の係数は、粒子の位置での電場を計算する時の内挿の係数と同じになっている。これらの係数が異なっていると、荷電粒子は、自分自身の作る電場で加速されることになり、数値的に不安定な系になるので注意する必要がある。この分割・割当方法は、それぞれの格子点から粒子を見て、遠い方の面積に比例して電荷を分割し格子点に割当てているため、area weighting と呼ばれる。

```
subroutine push(np,nx,ny,x,y,vx,vy,ex,ey,ctom)

     implicit none
     real(8), dimension(np)        :: x, y, vx, vy
     real(8), dimension(0:nx,0:ny) :: elx, ely
     real(8) :: ctom, dx, dy, dx1, dy1
     integer :: np, i, ip, jp

     do i = 1, np
! calcurate the electric field at the particle position
        ip = x(i)
        jp = y(i)

        dx = x(i) - dble(ip)
        dy = y(i) - dble(jp)
        dx1 = 1.0d0 - dx
        dy1 = 1.0d0 - dy
! electric field
        exx = ex(ip  ,jp  )*dx1*dy1 + ex(ip+1,jp  )*dx*dy1  &
            + ex(ip  ,jp+1)*dx1*dy  + ex(ip+1,jp+1)*dx *dy
        eyy = ey(ip  ,jp  )*dx1*dy1 + ey(ip+1,jp  )*dx *dy1 &
            + ey(ip  ,jp+1)*dx1*dy  + ey(ip+1,jp+1)*dx *dy
! push particles by dt
        vx(i) = vx(i) + ctom * exx * dt
        vy(i) = vy(i) + ctom * eyy * dt
        x(i) = x(i) + vx(i) * dt
        y(i) = y(i) + vy(i) * dt
     end do

end subroutine
```

Program.1 Particle pushing

```
 subroutine source(np,nx,ny,x,y,rho,chrg)

     implicit none
     real(8), dimension(np)        :: x, y
     real(8), dimension(0:nx,0:ny) :: rho
     real(8) :: chrg, dx, dy, dx1, dy1
     integer :: np, i, ip, jp

    do i = 1, np

        ip = x(i)
        jp = y(i)

        dx = x(i) - dble(ip)
        dy = y(i) - dble(jp)
        dx1 = 1.0d0 - dx
        dy1 = 1.0d0 - dy

        rho(ip  ,jp  ) = rho(ip  ,jp  ) + dx1 * dy1 * chrg
        rho(ip+1,jp  ) = rho(ip+1,jp  ) + dx  * dy1 * chrg
        rho(ip  ,jp+1) = rho(ip  ,jp+1) + dx1 * dy  * chrg
        rho(ip+1,jp+1) = rho(ip+1,jp+1) + dx  * dy  * chrg

     end do

end subroutine
```

Program.2 Charge assignment

以上より、PIC コードで Δt だけ時間ステップを進めるためには、Program.3 のようなプロセスを繰り返せば良いことが分かる。ここで下線を引いた部分は並列計算に関係した部分であるので、ここでは無視する。

```
! assign charges to grid points
     rho(:) = 0.0
     call source(np,nx,ny,xe,ye,rho,chrge)
     call source(np,nx,ny,xi,yi,rho,chrgi)
     call sumdim(nodes,myid,rho,phi,nxy)

! calculate phi and electric field
     call poissn(nx,ny,rho,phi)    ! poisson solver
     call efield(nx,ny,phi,ex,ey)  ! phi to ex and ey

! push particles by a time step size of dt
     call push(np,nx,ny,xe,ye,vxe,vye,ex,ey,ctome)
     call push(np,nx,ny,xi,yi,vxi,vyi,ex,ey,ctomi)
         ! treat partcles outside of the system
     call bound(np,xe,ye,vxe,vye,・・・・・)
     call bound(np,xi,yi,vxi,vyi,・・・・・)
```

Program 3. Advancement by one time step

荷電粒子の位置と速度を Δt だけ進める（時間積分する）と、粒子の位置がシステムの外に出てしまう場合がある。この場合の処理は、サブルーチン bound で行う。bound 内では、例えば粒子の境界条件を周期境界条件とすると、周期境界条件に従って、システム外に出た粒子の位置をシステム内に戻す。反射境界条件や吸収境界条件を用いる場合もある。bound のプログラミングも読者の宿題とする。

以上を用いると、２次元粒子コードの骨格は、Program.4 の様に表すことができる。ここでも下線部は並列計算に関係した部分であるので無視する。Program.4 の do loop は、1 時間ステップ毎の計算に対応していて、ループ内の作業は、Program.3 の内容になっている。

```
program pic2des
     include 'mpif.h'
     impicit none
     integer,parameter :: np = 1000000, &
          nx = 64, ny = 64, iend = 1000, nxy=(nx+1)*(ny+1)
     real(8), dimension(0:nx,0:ny) :: ex, ey, rho, phi
     real(8), dimension(np) :: xe, ye, vxe, vye &
                             xi, yi, vxi, vyi
     real(8) :: me, mi, chrge, chrgi, ctome, ctomi
     integer :: iloop
     call mpi_init(ierr)
     call mpi_comm_rank(mpi_com_world,myid,ierr)
     call mpi_comm_size(mpi_comm_world,nodes,ierr)
! set some parameters
            me    = 1.d0     ; mi    = 1836.d0
            chrge = -1.do    ; chrgi = 1.d0
            ctome = chrge/me ; ctomi = chrgi/mi
```

define some other parameters

```
     call iniset(np,nx,ny,xe,ye,vxe,vye,・・・・)
     call iniset(np,nx,ny,xi,yi,vxi,vyi,・・・・)
!-----------------------------------------------------
     do iloop = 1, iend
```

Program.3

```
     end do
!-----------------------------------------------------
     call mpi_finalize(ierr)
end program
```

Program.4 Main Program

実際のプログラムでは各種パラメータの設定も必要になる。これもプログラム中に「define some other parameters」とした場所に書き込めばよい。Program.4 では、粒子の位置と速度を決めるサブルーチン iniset が呼ばれている。たとえば、粒子の位置は空間的に一様に、速度は乱数を用いて熱平衡の速度分布関数（マクスウェル分布）に従うように設定する。iniset の作成も読者の宿題とする。

ここから、粒子コードの並列化について解説する。並列化のためには、mpi 関数を呼べば良い。Program.4 ではまず、全ての実行文の前に、

include ‘mpif.h’

という一行を入れる。mpif.h はヘッダーファイルで、mpi の関数を呼ぶためのパラメータファイルを定義している。mpi の開始と終了は

call mpi_init(ierr)

call mpi_finalize(ierr)

の関数を呼ぶことで実行される。個々のコア（CPU）は、システム全体の情報は必要としない。ただ、全体のシステムが何コアで動いていて、自分が何番目のコアであるかを知れば良い。他のコアの情報が必要な場合は、メッセージを受信することにより受け取る。逆にメッセージを送信する側になることもある。自分の番号 myid は以下の様に mpi の関数を呼ぶことにより得られる。

call mpi_comm_rank(mpi_comm_world,myid,ierr)

全体のコア数 nodes（歴史的に nodes と表記される場合が多い。初期の並列コンピュータでは 1 CPU=1 node であったことによると思われる）は、

call mpi_comm_size(mpi_comm_world,nodes,ierr)

で求めることができる。以上がどのような並列コードを組む場合も必ず呼ばなければならない関数である。

粒子コードの並列化は比較的簡単である。これは、粒子コードの根幹であり、計算時間の大部分を占める粒子の時間積分（particle pushing または particle acceleration）と電荷の分割・分配(charge assignment)が基本的に各コアで独立に実行できることによる。このため最も簡単な並列化では、Fig.9 に示すように場の量のコピー（レプリカ）を用いる。例えば各コアで、サブルーチン source により電荷密度を計算する。各コアで計算する電荷密度は、そのコアに所属する荷電粒子の寄与のみを含む。このため各コアに属する電荷密度に対応する配列の要素を全てのコアで合計し各コアに分配すれば良い。このために、mpi_allreduce という関数が用意されている。配列 rho に加えて、配列の和を蓄

える rho_sum という配列を定義し、そのサイズを nxy(=(nx+1)*(ny+1))とすると、以下のように関数を呼べば良い。

```
call mpi_allredce(rho, rho_sum, nxy, mpi_double_precision,  &
                  mpi_sum, mpi_comm_world, ierr)
```

この場合、上記関数を呼んだ後、rho_sum の値を rho に代入する必要がある。

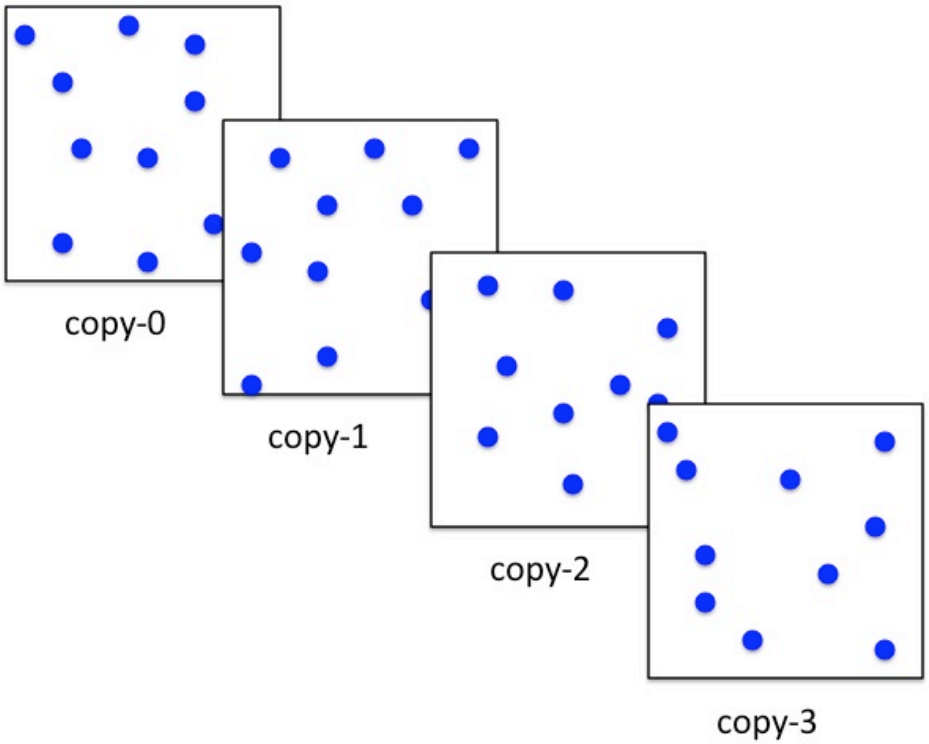

Fig.9 Parallelization using copies for field quantities

ここでは、並列化の実際を示す例として Program.5 にサブルーチン sumdim を示す。sumdim は２のベキ乗のコア数に対応している。総和計算と分配のアルゴリズムをコピーの数（コアの数）が８個の場合について Fig.10 に示す。第１ステップとして２個毎にグループを作り、それぞれ隣りどうしのペアで相手側にデータを送り、和を求める。第２ステップでは４個毎にグループを作り、その中で２個隣りのペア毎で、それぞれ相手側にデータを送り、和を求める。第３ステップでは８個毎にグループを作り、４個隣りのペア毎に、それぞれ相手側にデータを送り、和を求める。８個のコアの場合はこの段階で全てのコアが総和のデータを持っていることになる。コアの数が８以上の場合はこのプロセスを繰り返せばよい。以上の通信方法はバタフライと呼ばれている。

Program.5 では、非同期の send と receive 関数 mpi_isend と mpi_irecv を使っている。この場合はそれぞれの関数は mpi_wait とペアで用いる必要がある。同期の send と receive の場合は、データ数がコンピュータのバッファの制限を超えると正しい結果が得られない場合があるようである。また、プログラム中で mpi_barrier は、全てのコア間の同期を取る関数である。ここでは安全のため、mpi_barrier を呼んでいるが、原理的には省略してもよい。

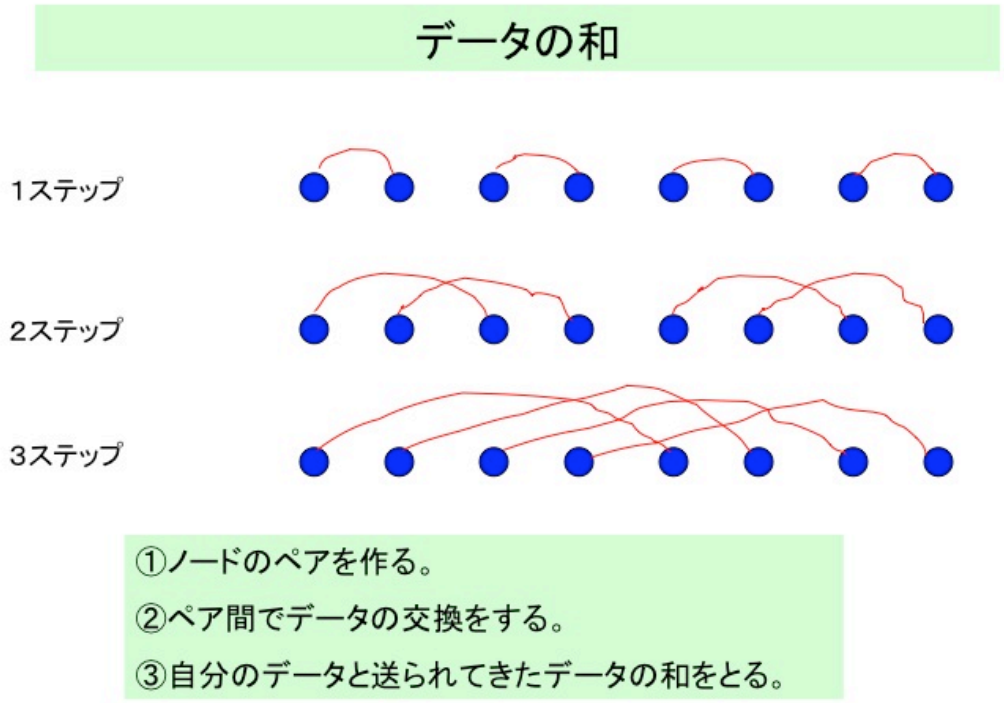

Fig.10 Algorithm for summation

電荷密度から静電ポテンシャルと電場を計算するのは各コアで独立に計算する。これは、各コアで同一の計算をするため冗長計算になるが、一般に粒子シミュレーションではメッシュ当たりの粒子数が数百個以上になり粒子に関する計算が支配的になるため、場の量の計算は問題にならない場合が多い。大規模シミュレーションで場の量の配列の大きさが大きくなると、場の量の計算が支配的になる場合がある。この場合については領域分割の手法等を用いる必要があるが、詳細は次節で述べる。

```
      subroutine sumdim(nodes,myid,a,b,ndim)

      implicit real*8(a-h,o-z)
      include'mpif.h'
      integer stat1(mpi_status_size)
      integer stat2(mpi_status_size)
      integer kmod
      dimension a(ndim), b(ndim)

      kmod = 1
!.................................................
      if(nodes.gt.1) then
      call mpi_barrier(mpi_comm_world,ierr)
    1 continue

      kmod1 = kmod
      kmod = kmod*2

      if (mod(myid,kmod) .lt. kmod1) then
        idiff = kmod1
      else
        idiff = -kmod1
      endif

      call mpi_isend(a,ndim,mpi_real8,myid+idiff,300,   &
                             mpi_comm_world,ireq1,ierr)
      call mpi_irecv(b,ndim,mpi_real8,myid+idiff,300,   &
                             mpi_comm_world,ireq2,ierr)
      call mpi_wait(ireq1,stat1,ierr)
      call mpi_wait(ireq2,stat2,ierr)

       do 20 j = 1, ndim
       a(j) = a(j) + b(j)
   20 continue

      if (kmod .lt. nodes) goto 1

      endif
!.................................................

      end subroutine
```

Program.5 Program for summation

問題１：規格化されたポアッソンの式を求め、差分で表せ。
[答え]

$$\frac{\partial^2\tilde{\phi}}{\partial\tilde{x}^2}+\frac{\partial^2\tilde{\phi}}{\partial\tilde{y}^2}=-\frac{\tilde{L}_x\tilde{L}_y}{N_e}\tilde{\rho}$$

差分化した式：

$$\begin{aligned}&[\,\phi(l+1,m)-2\phi(l,m)+\phi(l-1,m)\,]\\&+[\,\phi(l,m+1)-2\phi(l,m)+\phi(l,m-1)\,]\\&=-\frac{\tilde{L}_x\tilde{L}_y}{N_e}\sum_{s=e.i}\tilde{q}_s N_{sl,m}\end{aligned}$$

ここで$\tilde{L}_x$と$\tilde{L}_y$は、x 方向と y 方向のシステムサイズ、N_eはシミュレーションで使用する電子の数を示す。ただし、N_eは[コア当たりの粒子数]×[全コア数]であうことに注意する。また、$N_{sl,m}$は、s 種の粒子に対する（l,m）格子点に分配された粒子数（＝粒子密度）を示している。場の量に周期境界条件を仮定した場合は、高速フーリエ変換を利用して静電ポテンシャルを計算することもできる。

問題２：静電ポテンシャルと静電場の関係を規格化し、差分で表せ。
[答え]

$$\tilde{E}_x=-\frac{\partial\tilde{\phi}}{\partial\tilde{x}},\quad \tilde{E}_y=-\frac{\partial\tilde{\phi}}{\partial\tilde{y}}$$

差分化した式：

$$\begin{aligned}\tilde{E}_x(l,m)&=-0.5[\,\tilde{\phi}(l+1,m)-\tilde{\phi}(l-1,m)\,],\\\tilde{E}_y(l,m)&=-0.5[\,\tilde{\phi}(l,m+1)-\tilde{\phi}(l,m-1)\,]\end{aligned}$$

電場も周期境界条件を仮定した場合は、高速フーリエ変換により計算することもできる。

3.3 超並列処理に向けて

前説で解説した場の量にコピーを用いる粒子分割の方法は簡単で応用範囲が広い。慣れてくれば、１時間もあれば非並列コードを並列コードに書き換えることができる。特に１次元や２次元コードで十分多くの粒子を使用する場合は粒子分割による並列化で十分である。また、全てのコアで、ほぼ同一の計算がなされるため、コア間の負荷のバランスは非常に良い。研究室で製作する PC クラスタを利用する程度であれば特に高度の並列化プログラムに書き換える必然性は感じられない。コードの並列化に時間を取られるよりは、簡単な粒子分割のコードで早く結果を出して、論文を書いてしまうのが効率的である。

一方、超並列コンピュータを利用した超大型コンピューティングを目指すなら高度の並列化が必要となる。現在のコンピュータでも数千から数万のコアをもっている。例えば核融合科学研究所の SR16000 というコンピュータを例にとると、4096 物理コアをもっている。１物理コアで２スレッドのジョブが動作できるので、8192 論理コアに対応している。次世代のコンピュータでは数十万個以上のコアが予想される。非常に多くのコアに対応したプログラム技法の開発が必須である。

粒子分割では、場の量のコピーに対する総和の計算と、総和のコアへの分配の計算が、コア数が増えるに従って無視できなくなる。コピーの数が一定量を超えると通信時間が増加して、コピーの数を増やしても、高速化がなされない場合や、より遅くなる場合が生じる。コピーを使用しない方法としては領域分割という手法がある。１次元方向の領域分割の考え方を Fig.11 に示す。Fig.11 では２次元のシステムを１方向に領域分割している。領域分割した場合は、粒子の位置と速度を１ステップ進めて、他の領域に入る場合は、粒子のデータを他の領域に渡す（パスする）必要がある。また場の量も領域分割されている。このため、場の量の計算も分割数によって高速化される。他の領域と接するメッシュでは、格子点上の物理量は接する領域で共有されることになる。このため境界のメッシュ（ガードメッシュと呼ぶことがある）の領域間のデータ通信が必要になる。また高速フーリエ変換を使用する場合は、x 方向のフーリエ変換は、コア内に全てのデータがあるため問題ないが、y 方向のフーリエ変換をするためには、データが異なる領域に分散していることを考慮する必要がある。この場合 y 方向の領域分割を、x 方向の領域分割に変更する転置のサブルーチンを付加することにより問題は解決する。

３次元のシミュレーションを考えると、１方向のメッシュ数はコア数と比較してかなり少ない。従って、１方向の領域分割とコピー（領域分割）を併用する必要がある。この場合もコピーの数が増えると総和とその分配の計算が無視できなくなる。

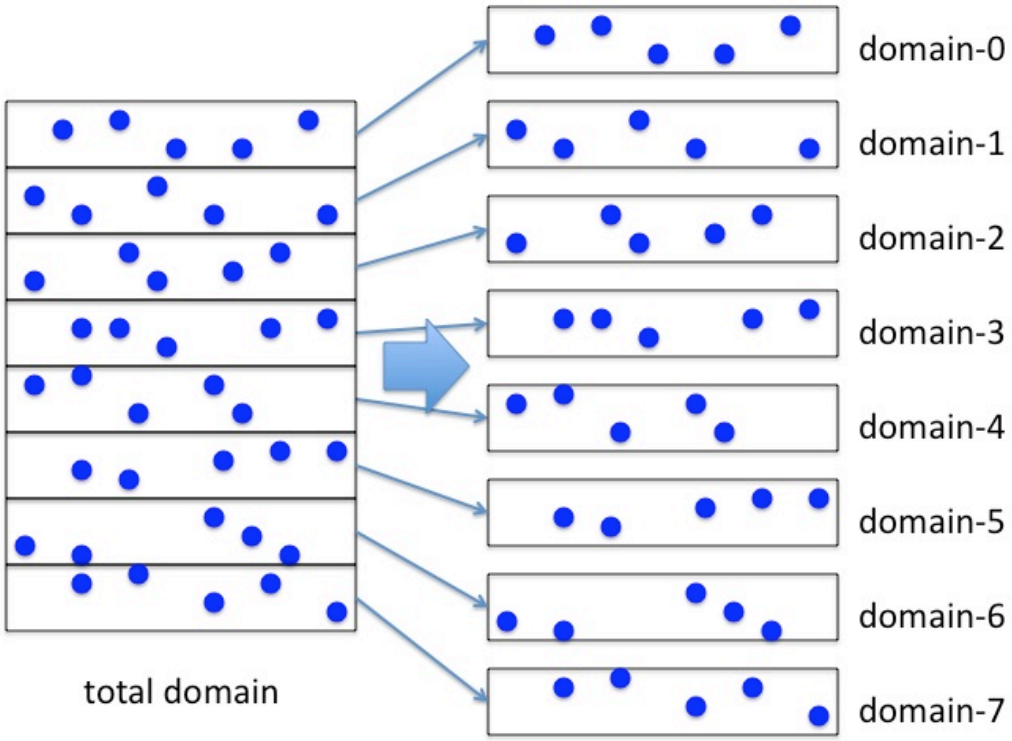

Fig.11 Domain decomposition in one-dimensional direction

コア数が増すに従って、２次元や３次元の領域分割をする必要が生じる。その場合、コア間でのロードがバランスするように領域分割の方法を選択する必要が生じる。直角座標系の場合は、問題は比較的簡単だが、円柱座標系や、一般化座標系の場合は、負荷がバランスするように領域を分割する方法を工夫する必要がある。たとえば円柱座標形で、半径方向に領域分割する場合には、各コアがほぼ同じ数の粒子を持つためには、半径方向のメッシュの分割は一様でなくなる。この場合、コアによって場の量の計算にアンバランスが生じるので注意が必要である。将来的には、粒子を取り扱う場合のメッシュ分割と、場の量を取り扱う場合のメッシュ分割を独立に取る必要が生じるかもしれない。

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、高度の並列化は極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。外国の研究所では、物理モデルのプログラミングと並列プログラミングも、それぞれ別の研究者が分担して行う場合が多いようである。

個々のプログラムが与えられた場合、異なる並列コンピュータでの並列化性能を比較する必要がある。計算性能（パフォーマンス）のコア数に対するスケーリングを出す場合、弱いスケーリングと強いスケーリングという二つのスケーリングが存在する。

強いスケーリングでは、プログラムサイズを固定して、コア数を増やした場合の計算時間の逆数の変化を調べる。コア数を N とすると、対応する計算時間が 1/N に、高速化性能は N 倍になるのが理想である。

弱いスケーリングでは、1 コア当たりのプログラムサイズを一定とする。この場合 N コアでの計算は、N 倍大きなプログラムを実行することになる。例えば、粒子コードを例に取ると、場の量は各コアで同じである。各コアでの粒子数を一定とすると、N 個のコアを用いた計算では、N 倍の粒子数を用いた計算になる。この時、理想は、コア数を増やしても計算時間が変化しないことである。この場合に、計算性能は N 倍になる。一般に粒子コードでは、粒子数は統計性に関連するため、物理結果の粒子数に対する依存性を調べることも多い。この場合は、弱いスケーリングの測定と同時に物理結果の粒子数依存性を調べることができる。

一般に粒子コードでは、強いスケーリングを測定すると、コア数に反比例して、コア当たりの粒子数が減少し、場の量の計算はコア数依存性がないため、スケーリングが飽和しやすい。逆にスケーリングが飽和しないように十分な数の粒子を使用すると、コア数の少ない場合の計算で、1 コア当たりの粒子数が主記憶の制限を超えてしますのでスケーリングを測定するのが困難である。以上の理由により、粒子シミュレーションの並列化性能は弱いスケーリングで表される場合が多い。

3.4 終わりに

本文中では、省略したが、最後に参考文献をまとめて示す。粒子シミュレー

ションの教科書としては文献[1,2]がある。PC クラスタで、並列化プログラムを動かすための環境である Score の解説として文献[3]がある。MPI プログラミングは文献[4]に、OpenMP プログラミングは文献[5,6]に解説されている。粒子シミュレーションの解説としてはプラズマ・核融合学会誌の講座[7]がある。また、プラズマ・核融合学会誌の PC クラスタの小特集[8]も参照されたい。

謝辞

PC クラスタの作成、プログラミングに関する相談等で大変お世話になっています山口大学の技術専門職員の田内康氏に感謝します。長年の共同研究者である、高度情報科学技術研究機構(RIST)の徳田伸二博士（2009 年 9 月までは日本原子力研究開発機構所属）、九州大学応用力学研究所の矢木雅俊教授に感謝します。並列コンピューティングに関するご助言をいただいている核融合科学研究所の中島徳嘉教授に感謝します。また、種々の計算を実行していただいている山口大学大学院理工学研究科、先端エネルギー研究室の学生さんに感謝します。

参考文献

1) C. K. Birdsall and A. B. Langdon: *Plasma Physics via Computer Simulation*, (Institute of Physics Publishing, Bistrol and Phyladelphea, 1995).
2) T. Tajima, *Computational Plasma Physics: With Application to Fusion and Astrophysics*, (Addison-Wesley Publishing Company, Inc., 1989).
3) 石川裕、佐藤三久、堀敦史、住元真司、原田浩、高橋俊行、「Linux で並列処理をしよう−Score で作るスーパーコンピューター、(共立出版、2002).
4) P.パチェコ、秋葉博約、「MPI 並列プログラミング」、(培風館、2001).
5) 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)、2006).
6) 北山裕幸、「OpenMP 入門　マルチコア CPU 時代の並列プログラミング」、（(株）秀和システム、2009）.
7) 内藤裕志: 講座　プラズマ計算機シミュレーション入門 I「２．粒子シミュレーションの基礎」、J. Plasma & Fusion Res., 74, pp.470～478, (1998).
8) 内藤裕志、矢木雅俊、姫野龍太郎、重谷隆之、黒川原佳、石川裕、南里豪志、Decyk Viktor K., Dauger Dean E., 福山淳、小特集「PC クラスタを作ってみませんか？」、J. Plasma & Fusion Res., 79, pp.750～789, (2003).