J. Plasma Fusion Res. Vol.89, No.1 (2013) 49‐56

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

2．並列コードを開発するための基礎技法

坂上仁志
核融合科学研究所

（原稿受付：2012年11月12日）

本章では，まず，並列計算機のアーキテクチャ（内部構造）について簡単に紹介し，並列実行できるコードをプログラミングするために最低限必要な事柄を概説し，特に注意を払わなければならないプロセス間通信について述べる．そして，最後に並列コードを開発するための環境について，最新の状況を紹介する．

Keywords:
code development, parallelization, parallel language

2.1 はじめに

コンピュータを用いたシミュレーションは，理論，実験に次ぐ第三の重要な研究手段であり，コンピュータの性能向上に伴って，研究は，より詳細に，より大規模になってきた．しかし，コンピュータを構成する基本要素であるプロセッサの性能向上は，命令レベル並列性の壁，消費電力の壁，メモリの壁という，三つの壁に突きあたっており，さらなる高性能を実現することは困難と考えられている[1]．まず，命令レベル並列性の壁とは，1クロックあたりの演算性能向上をめざして，同時に複数の命令が並列実行できるハードウェアをプロセッサに導入したが，命令やデータに依存関係があるため，実際に並列実行できる場合は非常に限定的であり，そのハードウェアを有効に活用できなかったことである．その一方で，プロセッサの消費電力は，熱密度の観点から限界近くに達しており，電圧を上げることで動作周波数を高くし，性能向上を図るという手法は通用しなくなっている．これを消費電力の壁と呼ぶ．最後のメモリの壁とは，メモリのアクセス速度の向上はプロセッサの演算性能の向上よりずっと遅く，両者の乖離が著しくなっており，キャッシュだけでは，もはや，その溝を埋めることができないということである．そして，命令レベル並列性の壁は，必然的にスレッドレベルやプロセスレベルの並列性が重要であることを示唆している．また，一般に消費電力は，動作周波数のおおよそ自乗に比例するため，マルチコア化によりコア数を2倍にし，動作周波数を1/2にすれば，同一の演算性能で消費電力は半分になる．このため，動作周波数を下げてマルチコア化することは，消費電力の壁に対する一つの解決策であるが，マルチコアを効率良く使う並列プログラムの存在が前提である．最後のメモリの壁も，ローカライズされたメモリ上のデータとコアの組み合わせが重要となることを示しており，分散メモリ型の並列計算が必要となる．この流れは，コモディティ・コンピュータを安価なネットワークで結合した，いわゆるBeowulf型のPCクラスタ型並列計算機でも同様である．つまり，分散されたメモリを前提とした並列計算機と並列プログラミングが，これからの高性能コンピューティングには必須である．

2.2 並列計算機のアーキテクチャ

コンピュータにおいて一連の演算を行うハードウェアは，従来CPUと呼ばれることが多かったが，ここではコアと呼ぶことにする．コアはメモリと接続されており，メモリからデータを読み出して演算し，その結果をメモリに書き込むことが基本動作である．また，コアが，外部のハードウェアにアクセスできるようにするためのコントローラがある．このひとまとまりを計算ノードと呼ぶことにし，**図1**(a)に示す．

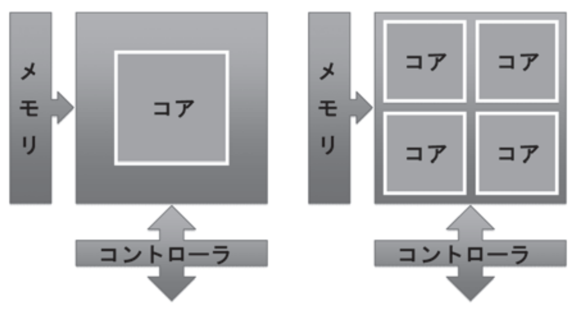
まず，以下のプログラムを考える．

```
do i = 1, N
   a(i) = b(i) + c(i)
end do
do i = 1, N
   a(i) = s * b(i) + t * c(i) * c(i)
end do
```

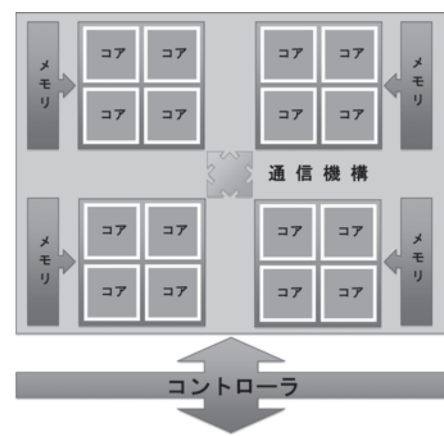
最初のループでは，演算一回毎に，データの読み出しが2つ，書き込みが1つ必要である．ここで，簡単のため次のデータを読み出しているときに，今計算したデータを同時に書き込めるとする．例えば，コアの演算性能が100 MFLOPSであるとすると，1秒間に1億回の演算ができることになり，この性能を活かすためには，1億分の一秒ごとに2つのデータをメモリからコアに転送しなければならない．単精度の実数変数は4バイトであるので，コアとメモリ間のデータ転送速度（メモリバンド幅）は，800 MB/s

2. *Elementary Technique to Develop Parallelized Codes*
SAKAGAMI Hitoshi *author's e-mail: sakagami.hiroshi@nifs.ac.jp*

©2013 The Japan Society of Plasma Science and Nuclear Fusion Research



(a)単一コアの計算ノード　(b) マルチコアの計算ノード



(c) 複数モジュールから構成される計算ノード

図１：計算ノードのアーキテクチャ．

必要になる．後のループでは，同じデータ量について，演算回数が4回なので，メモリバンド幅は200 MB/sあれば，十分である．このメモリバンド幅と演算性能の比はB/Fと呼ばれ，コアの実効性能を知る上で非常に重要な指標である．B/Fが2のコンピュータでは，後のループは100%の性能で実行されるが，最初のループは25%の性能しか発揮できないことになり，むやみに演算性能だけを向上させても意味はなく，これが，メモリの壁の正体である．コモディティ・コンピュータでは，B/Fが0.1以下であることも珍しいことではなく，カタログに書かれている性能を実感できない主因である．しかし，メモリバンド幅を向上させるためには，非常に大きなコストが必要なため，大きなB/Fの実現は現実的には難しい．そこで，通常，コアにはキャッシュと呼ばれる内部機構が付加されている．キャッシュとメモリ間のデータ転送速度は，演算速度に比較して遅いメモリバンド幅であるが，コアとキャッシュ間は高速にデータ転送ができる．このため，演算に必要なすべてのデータを一旦キャッシュに格納できると，メモリにアクセスすることなく演算ループが実行できるので，プログラムのその部分だけはメモリバンド幅に縛られないで，コアの演算性能を発揮できることになる．

最近のマルチコア化の流れにより，計算ノードは，コモディティ・コンピュータであっても，複数のコアが一体となってメモリに接続されている場合が多く，これを**図1**(b)に示す．この場合，それぞれのコア間でメモリを共有しており，いわゆる共有メモリ型の並列処理が可能である．さらに，この計算ノードをモジュールとして，複数のモジュールをモジュール間の通信機構とともに１つの計算ノードとして実装している高性能なコンピュータもあり，これを**図1**(c)に示す．このアーキテクチャは，計算ノード内において，一見分散メモリ型の並列計算機を構成しているようであるが，すべてのコア間でモジュールを跨ってメモリを共有しており，より多くのコアを使った共有メモリ型の並列処理が可能である．しかし，メモリアクセス速度が均等ではないことに注意しなければならない．即ち，コアと同じモジュール内のメモリには，メモリバンド幅でアクセスできるが，他モジュールのメモリには，一般には，それより更に遅いモジュール間の通信機構を介しなければならず，このようなメモリ構成は，NUMA（NonUniform Memory Access）と呼ばれる．NUMAでは，モジュール間通信機構の性能が足枷となり，モジュール内のコア数以上のコアを用いて共有メモリ型並列処理を行っても，性能は向上しない可能性がある．また，モジュール内のコア数以下のコアを用いる場合でも，あるモジュール内だけのコアを使う場合と他モジュール内のコアも一緒に使う場合には，性能差が現れる．さらに，演算するコアを含んだモジュールとは別のモジュール内のメモリを使うこともできるため，同じモジュール内のメモリを使う場合とは，異なった性能を示す．このように，複数のモジュールから成る計算ノードを使う場合は，どのモジュールのコアとメモリを使うかを制御することが重要となり，一般にはアフィニティ制御と呼ばれる．例えば，Linuxでは，numactlコマンドで制御できるので，参照されたい．

さて，並列計算機では上記で述べた計算ノードを通信ネットワークで接続し，計算ノード間でデータ転送できるようにしている．高性能な並列計算機では，その通信ネットワークに高速な専用ハードウェアを用いていることが多く，PCクラスタ型並列計算機では，Giga Bit Ethernet等のLANを構築するためのハードウェアを用いることが多い．複数ノードを用いて並列処理をする場合，この計算ノード間の通信性能が，並列処理の効率に大きな影響を与える．このため，計算ノードの演算性能とノード間の通信速度のバランスを考えることが重要となる．

2.3 並列コードのプログラミング

性能のいい並列コードを作成するためには，並列計算機のアーキテクチャを考慮したプログラミングが必要である．まず，前節で述べたキャッシュをうまく活用するため，配列変数へのアクセスは，できるだけ連続アクセスになるように，配列変数とループを定義する．Fortranにおける多次元配列では，より左側の添え字が優先されて小大順に変化するように配列要素がメモリ上に並ぶ．なお，C言語では，逆により右側の添え字が優先されて小大順に変化するので注意すること．さて，ループAおよびBを含む以下のプログラムを考えると，配列変数について，ループAでは10,000要素毎の飛び飛びのアクセスになっているが，ループBでは連続アクセスになっている．この２つのループでは，演算やメモリアクセスの総量は同じであるが，実行速度に大きな差があり，当然，後者の方がずっと速い．

```
parameter(N=10000)
dimension a(N,N), b(N,N)
! Loop A
do i = 1, N
 do ii = 1, N
    a(i,ii) = a(i,ii) + b(i,ii)
 end do
end do
! Loop B
do ii = 1, N
 do i = 1, N
    a(i,ii) = a(i,ii) + b(i,ii)
 end do
end do
```

キャッシュ容量は，メモリ量に比べると非常に小さいため，通常は配列変数の全要素をキャッシュに格納することはできない．この場合，連続アクセスを実現しただけでは，キャッシュを有効に活用できずに性能低下を招く．シミュレーションに現れる典型的なループとして以下のプログラムを考える．

```
parameter( N=10000 )
real x(N), y(N), vx(N), vy(N)
do i = 1, N
   x(i) = x(i) + vx(i)
   y(i) = y(i) + vy(i)
end do
```

まず，x(i)とvx(i)がアクセスされるので，それぞれのメモリ上のアドレスを先頭とする連続したデータブロックがキャッシュに一括ロードされ，ループ内の1行目が計算される．プログラムの実行順序に従うと，次にy(i)とvy(i)がアクセスされるが，これらはキャッシュには格納されていない．この状態をキャッシュミスと呼ぶ．このため，キャッシュ容量が少ない場合には，先程ロードしたx(i)とvx(i)の後続データを破棄し，新たにメモリからy(i)とvy(i)およびそれらの後続データがキャッシュ上に一括ロードされ，ループ内の2行目が計算される．その次にアクセスされるのは，x(i+1)とvx(i+1)であり，同様にキャッシュミスが起こり，メモリから再びデータをロードする．このように，キャッシュミスが頻発し，キャッシュに前もってロードされたデータを有効にアクセスできない場合には，演算性能が大幅に低下する．そこで，以下のようにプログラムを書き換えて，計算に必要なデータが連続アドレスに配置されるよう最適化する．この場合，最初のアクセスでpv(1,i)と後続のデータがキャッシュ上に一括ロードされると，それらのデータをすべて使い切るまで，キャッシュミスが発生しないため，キャッシュを有効に活用できる．

```
parameter( N=10000 )
real pv(4,N)
! x(i)=pv(1,i),y(i)=pv(3,i),
! vx(i)=pv(2,i),vy(i)=pv(4,i)
do i = 1, N
   pv(1,i) = pv(1,i) + pv(2,i)
   pv(3,i) = pv(3,i) + pv(4,i)
end do
```

上記の書き換えでは，配列変数pvの1次元目添え字と物理量の関係が不明瞭なため，プログラムの意味が理解しにくい．そこで，構造体を使って以下のように書くこともできる．

```
parameter( N=10000 )
type particle
   sequence
   real :: x, vx, y, vy
end type
type(particle) :: pv(N)
do i = 1, N
   pv(i)%x = pv(i)%x + pv(i)%vx
   pv(i)%y = pv(i)%y + pv(i)%vy
end do
```

さらに，書き換えなければならない箇所が多い場合，面倒な作業になり，かつ修正ミスを誘発してバグが混入しやすい．この場合，以下のようにコンパイラのプリプロセッサ機能を使うと簡略化できるので，参考にされたい．

```
parameter( N=10000 )
#define x(i) pv(i)%x
#define y(i) pv(i)%y
#define vx(i) pv(i)%vx
#define vy(i) pv(i)%vy
type particle
   sequence
   real :: x, vx, y, vy
end type
type(particle) :: pv(N)
do i = 1, N
   x(i) = x(i) + vx(i)
   y(i) = y(i) + vy(i)
end do
```

さて，これでようやくコードの並列プログラミングの準備ができたことになる．まず試すべきは，コンパイラの自動スレッド並列化機能の活用であろう．プログラムを修正することなく必要に応じてコンパイラ固有の指示行を挿入するだけで，コードの共有メモリ型のスレッド並列化ができる．一般に，並列化された一つのスレッドは，メモリを共有する一つのコア上で実行される．この方法は，非常にお手軽であるが，期待した性能を得られない場合も多い．また，良好な性能が得られている場合でも，それはコンパイラの能力に依存していることには注意されたい．コードの並列性を解析する能力はコンパイラによって異なるた

め，同じコードを別のコンパイラでコンパイルすると，まったく自動並列化が働かないことも考えられる．また，このような指示行は一般にコンパイラ固有であり，別のコンパイラでは無視されるため，新しいコンパイラ用の指示行を再度挿入する羽目になってしまう．このため，ほんの僅かな手間で自動並列化できない場合は，そのコンパイラの指示行を用いて並列プログラミングをするのではなく，OpenMP[2]を使うべきである．

OpenMP は，従来のプログラムに指示行を挿入するだけで共有メモリ型のスレッド並列を記述できる標準化されたインターフェイスであり，ほとんどのコンパイラで利用することができる．上述のループでは，以下のように書くだけで並列化されるが，自動並列化とは異なり，指示行で並列実行を指定しないループは，一切並列化されない．なお，一つのスレッドを一つのコア上で実行することは，自動スレッド並列化と同様である．

```
!$OMP PARALLEL DO
do i = 1, N
   x(i) = x(i) + vx(i)
   y(i) = y(i) + vy(i)
end do
```

OpenMP を用いるとコンパイラ環境に依存しないで，比較的容易に並列プログラミングできるが，複数のコア（＝スレッド）から一斉に，近接したアドレスのメモリにアクセスが集中するので，メモリバンド幅の制限が大きく影響し，たとえキャッシュがあったとしても，コア数が多くなると並列処理による性能向上は得にくくなる．このため，数十コア以上を用いて良好な並列性能を得るためには，MPI（Message Passing Interface）[3]と呼ばれるプログラミングインターフェイスを用いて並列プログラミングし，複数の計算ノードから構成される分散メモリ型の並列計算機上で実行しなければならない．

MPI とは，分散メモリ型の並列計算をサポートするための標準化されたライブラリ群であり，自分のコードからサブルーチンを呼び出す形で利用する．MPI では，計算を行う単位をプロセスと呼び，各プロセスはそれぞれ独立した固有のメモリ空間を持つ．並列計算に必要なプロセス間のデータ転送は，プログラム実行のフローを意識した上で，ユーザが明示的に記述しなければならず，デッドロックの危険が常にある．この場合も，一般には，並列化された一つのプロセスは，一つのコア上でメモリを共有することなく実行される．なお，計算ノードのアーキテクチャを考慮し，ハイブリッド並列と呼ばれる複数のコア上で一つのプロセスを実行する場合もあるが，これについては次節で述べる．さて，例えば，２つのベクトルを読み込んで内積を求める以下のプログラムを考える．

```
parameter(n=100)
real a(n), b(n)
read(*,*) a, b
aipd = 0.0
do i = 1, n
   aipd = aipd + a(i) * b(i)
end do
write(*,*) 'aipd = ', aipd
stop
end
```

これを MPI で並列プログラミングし，内積計算を並列化するためには，以下のように書かなければならない．

```
parameter(n=100)
real a(n), b(n)
integer istat(MPI_STATUS_SIZE)
call MPI_INIT ( ierr )
call MPI_COMM_SIZE ( MPI_COMM_WORLD, np, ierr )
call MPI_COMM_RANK ( MPI_COMM_WORLD, id, ierr )
if( id .eq. 0 ) then
  read(*,*) a, b
  do i = 1, np-1
    is = ( n / np ) * i + 1
    call MPI_SEND ( a(is), n/np, MPI_REAL, i, &
                       1, MPI_COMM_WORLD, ierr )
    call MPI_SEND ( b(is), n/np, MPI_REAL, i, &
                       1, MPI_COMM_WORLD, ierr )
  end do
else
  call MPI_RECV ( a, n, MPI_REAL, 0, 1, &
                   MPI_COMM_WORLD, istat, ierr )
  call MPI_RECV ( b, n, MPI_REAL, 0, 1, &
                   MPI_COMM_WORLD, istat, ierr )
end if
aipdt = 0.0
do i = 1, n / np
  aipdt = aipdt + a(i) * b(i)
end do
call MPI_REDUCE ( aipdt, aipd, 1,MPI_REAL, &
              MPI_SUM,0,MPI_COMM_WORLD,ierr)
if( id .eq. 0 ) write(*,*) 'aipd = ', aipd
call MPI_FINALIZE ( ierr )
stop
end
```

ここでは，np 個のプロセス（各プロセスは id=0, 1, ..., np-1 の番号で識別される）で並列処理を行っている．まず，id=0 のプロセスがデータを配列 a, b に読み込み，配列 a, b の一部分のデータを id=1, 2, ..., np-1 のプロセスに送信（MPI_SEND）している．一方，id=1, 2, ..., np-1 のプロセスは，id=0 のプロセスからのデータを受信（MPI_RECV）する．そして，各プロセスは，自身が担当する部分の和 aipdt を計算し，最後にリダクション処理（MPI_REDUCE）により全プロセスと通信／演算し，全体和である内積 aipd を求めている．このように，MPI を用いた並列プログラミングは，計算機科学（Com-

puter Science）の研究者ではない計算物理（Computational Physics）の研究者や一般ユーザには，やや難しいものとなっている．しかし，MPI は，ほとんどすべての並列計算機上に実装されており，デファクト標準である．

なお，MPI を用いたプログラムは，共有メモリ型の並列計算機上でも実行することが可能であり，各プロセスはメモリ空間上には独立に割り付けられるため，近接したアドレスのメモリにアクセスが集中しない．このため，共有メモリ型並列計算機であっても，スレッド並列のプログラムより，プロセス並列の MPI プログラムの方が高速に並列実行できる場合がある．

最後にスーパースカラについて述べる．問題サイズを固定して並列計算を考える場合，並列処理するプロセス数が増える程，一般に一つのプロセスに割り振られる計算量が減少し，必要とするメモリ量も減少する．このため，あるプロセス数を越えたところで，計算に必要なすべてのデータがキャッシュに格納できるか，キャッシュを非常に効率よく利用できる程度に減少する場合がある．横軸をプロセス数に縦軸に並列性能をプロットすると，通常はプロセス数が増加すると，並列オーバーヘッドの増加により性能が暫時低下するが，この場合，そのプロセス数のところで並列性能が急によくなる．この現象をスーパースカラと呼び，並列計算機の性能評価を難しくしている要因の一つである．

2.4　並列コードにおける通信

分散メモリ型の並列計算では，メモリアクセスより更に遅い計算ノード間の通信ネットワークを介して，データ交換が行われる．このため，たとえ演算処理が100%並列化できても，通信に要する時間は並列オーバーヘッドとなり，高い並列性能を得るときの足枷になる．これは，並列処理のプロセス数が増えると顕著になる．プロセス数の増加により，同時に通信すべきコネクションの本数も増えるが，通信ネットワークのハードウェアによっては，通信の衝突が起こり，片方のコネクションが待機状態になる場合がある．この場合，実効的な通信時間は長くなり，並列オーバーヘッドの増加と並列性能の低下に直結している．通信ネットワークに専用の高価なハードウェアを用いている並列計算機では，この性能低下は起こりにくい．しかし，PC クラスタ型並列計算機での通信ネットワークとしてよく用いられている LAN 用のハードウェアでは，このような通信の衝突は避けられないため，注意を要する．さらに，PC クラスタ型並列計算機では，ファイルシステムを複数の計算ノードでネットワーク共有するために NFS を用いていることが多く，NFS のための通信と並列処理のための通信が衝突し，思わぬ性能低下に直面する．この場合，NFS 用と並列処理用で別々の LAN を用意すべきであろう．つまり，並列計算機では，演算速度と通信速度のバランスが大事であり，通信ネットワークをなおざりにして，むやみに演算速度だけ速くしても，高い並列性能は得られない．

いずれにせよ，高い並列性能を得るための基本は，他プロセスが持っているデータを使わないで，なるべく自プロセスのデータだけで計算できるように，データの配置等を工夫することであり，場合によってはアルゴリズムを変えることも検討する．例えば，FFT はスペクトル法でよく使われているが，FFT そのものは全プロセスのデータが必要であり，通信負荷は大きい．このため，通信負荷が小さい FFT を使わないアルゴリズムが有効な場合もある．例えば，100台の並列計算では，通信負荷が大きいため10倍しか速くならないアルゴリズムより，一台で計算する場合は 5 倍遅くても，通信負荷が小さくて100台で80倍速くなるアルゴリズムの方が，前者より最終的には 1.6 倍速く計算できる．また，全データにおける総和や最大値等を求めるいわゆるリダクション処理も可能な限り少なくすることがポイントである．リダクション処理における通信量は一般に少ないが，必ず計算に参加する全プロセスを対象とした同期が必要である．この同期のための通信コストは，プロセス数が増えると急激に大きくなるからである．

次に，並列処理におけるデータ交換のためにポイント・ツー・ポイントの双方向通信を用いる場合の注意点について述べる．ここでは，自プロセスのデータを右隣のプロセスに送り出す処理を send / receive のペアで実装する場合を考える．まず，send / receive のペアがきちんとお互いに協調しながら動作する基本的な同期通信を考える．これを一斉に右隣のプロセスにsendした後で，一斉に左隣のプロセスから receive するというナイーブに実装した様子を**図 2**（a）に示す．一般に，send 処理は，そのデータがきちんと receive されなければ完了しない．このため，この実装では，プロセス間でデッドロックが発生してしまう．ただし，通信するデータ量が少なくて，全データが send 用のバッファにたまたま収まった場合，send 処理が完了する並列計算機システムもあり，この場合にはうまく動く．しかし，このような正常動作が並列計算機システムの実装や問題サイズに依存するプログラムは，可搬性が悪いので注意が必要である．デッドロックを防ぐためには，**図 2**（b）に示すように，例えば，まず，奇数のプロセスは右隣のプロセスに send し，偶数のプロセスは左隣のプロセスから receive する．次に，偶数のプロセスが右隣のプロセスに send し，奇数のプロセスは左隣のプロセスから receive するよう，常にハンドシェイクしなければならない．同期通信では，各プロセス間に独立した通信チャネルが用意されている場合でも，全体の半分の通信チャネルしか同時に利用しないので，通信ネットワークのハードウェアを有効に活用していない．そこで，すべての通信が同時に行えるよう**図 2**（c）に示す非同期通信を考える．非同期通信では，send / receive を呼び出すと，その処理の完了を待たずに呼び出し側に戻る．このため，すべてのプロセスにおいて，send / receive 処理が起動され，全チャネルを使って通信が開始される．そして，wait により，通信の完了を待ち合わせる．waitで待ち合わせる前に，send されている変数を書き換えた場合には，send を呼び出したときの変数の内容ではなく，書き換えた後の内容が通信される可能性がある．同様に，receive されている変数を wait 前に読み出した場合には，本来の通信で受け取る内容ではなく，通信する前の古

い内容を読み出す可能性がある．このため，非同期通信を行う場合は，wait の前にアクセスできる変数とそうでない変数を明確に区別しなければならないので，注意が必要である．

最後に特に大規模の並列計算で有用なハイブリッド並列化について述べる．ハイブリッド並列化とは，メモリを共有する計算ノード内ではスレッド並列計算を行い，計算ノード間は分散メモリ型のプロセス並列計算を行う並列化の手法である．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化だけで行う場合に比べて，ハイブリッド並列化の方が通信チャネルの数が減るので，通信オーバーヘッドの減少が期待できる．特に，リダクション処理に伴う同期通信を行うプロセス数が減る効果は大きい．並列プログラミングは，複雑になるが，大規模の場合，検討する価値は十分にある．ただし，スレッド並列計算はスレッド（＝コア）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できるだけ増やして，分散メモリ型並列計算を行うプロセス数をできるだけ減らすことが，最適な並列性能を与えないことには，注意が必要である．

2.5 並列コードの開発環境

現在，規模の大小を問わず並列計算機上で並列に動作するプログラムを開発する場合，MPI を用いることがデファクト標準であるが，並列プログラミングは非常に煩雑なものである．これを象徴する言葉として"Life is too short for MPI"が有名である[4]．一方で，計算ノードの最近のトレンドは，マルチコア化である．今までは，コアのクロック周波数の向上等によって，プログラマは，黙っていても技術革新の恩恵を受けられた．しかし，計算ノード内のマルチコアを活用して並列計算機の性能を享受するためには，根本的にプログラムを書き直して，並列化しなければならない．この状況を揶揄する言葉として"The free lunch is over"がある[5]．

そこで，従来のプログラムに最小限の付加的な指示行を追加するだけで分散メモリ型の並列計算機上でプログラムの並列実行を可能とするデータ並列言語 HPF（High Performance Fortran）が提案された［6］．基本的にHPFでは，データを各プロセッサ上にどのように配置するかのみをユーザが明示的に指示し，データ転送等のそれ以外のことはすべて処理系に任せる．このため，ユーザは煩わしいプロセッサ間の通信管理や実行制御を記述するプログラミングから解放され，比較的容易に並列計算機の高性能が享受できる．例えば，前節の 2 つのベクトルを読み込んで内積を求めるプログラムをHPFで並列化すると，以下のようになる．

```
      parameter(n=100)
      real a(n), b(n)
!HPF$ PROCESSORS proc(number_of_processors())
!HPF$ DISTRIBUTE (BLOCK) ONTO proc :: a,b
      read(*,*) a, b
      aipd = 0.0
!HPF$ INDEPENDENT, REDUCTION(+:aipd)
      do i = 1, n
         aipd = aipd + a(i) * b(i)
      end do
      write(*,*) 'aipd = ', aipd
      stop
      end
```

MPI に比べると，プログラミングの容易さは圧倒的であり，HPF には大きな期待が寄せられていた．しかし，HPF を使うためには Fortran90クリーンなプログラムにしなければならず，その手間が大きかったことや，当初の HPF コンパイラの性能があまりにも低く，並列化されていても並列性能が悪かったりしたため，期待が大きかった反動もあり，大きな失望が急速に広がってしまった．このため，残念ながら，この段階からHPFが今後広く普及する可能性は低いと言わざるを得ない．しかし，現在，HPF 推進協議会［7］からフリーのHPFコンパイラが配布されており，規則正しい差分系スキームのプログラムなら，簡単に並列化ができ，見劣りしない並列性能も得られるため，分散メモ

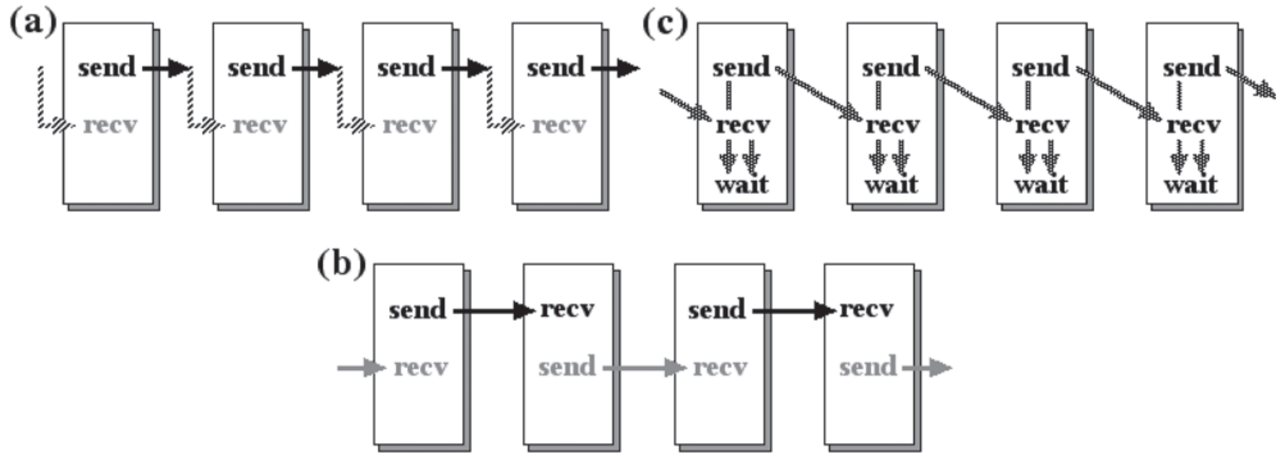


図 2 ：プロセス間のポイント・ツー・ポイント双方向通信．
（a）ナイーブな同期通信，（b）ハンドシェイク同期通信，（c）非同期通信

リ型並列処理の入門としては，非常に有用である[8]．

一方，並列処理を記述するため Coarray という概念が，Fortran2008の国際規格に導入された[9]．Coarray とは，各プロセスが自分のメモリ空間を供出し，全体としてグローバルな大域アドレス空間を作り上げ，他プロセスのメモリ空間を容易にアクセスできるように，いわば共有メモリ風に見せる仕組みである．逆に見れば，グローバルアドレス空間は，実際には複数の計算ノードに分割されて保持されているため，これを強調して Coarray は Partitioned Global Address Space（PGAS）と呼ばれることが多い．HPF では，元々の計算したい大きなサイズの問題を小さく分割して，それぞれを複数のプロセスに分担させるイメージであるが，Coarray では，それぞれのプロセスが計算する小さなサイズの問題を寄せ集めて大きなサイズにするイメージである．実際の他プロセスへのメモリアクセスは，put / get と呼ばれる単方向通信で実装されることが多く，他プロセスのメモリ空間上に直接書き込んだり，読み出したりできる．双方向通信では，送り手と受け手が協調して動作する必要があるため，効率が悪くなりがちである．しかし，一方的に動作する単方向通信は，メモリを操作される側の都合を考える必要がなく，効率のよい通信が期待できる．ただし，これはメモリのコンシステンシをプロセス間で伝達する特別なハードウェアが実装されている場合で，そのようなハードウェアを持たない並列計算機で単方向通信を行うと，双方向通信よりかえって性能が劣化するので注意を要する．

さて，Coarray を使ったプログラムを以下に示す．

```
   real :: r[*]     ! Scalar co-array
   real :: x(n)[*] ! Array  co-array
   real :: t        ! Local scalar
   integer :: p     ! Local scalar
 ! GET communication
   t = r[p]
   x(:) = x(:)[p]
 ! PUT communication
   x(:)[p] = r
```

GET では，id=p のプロセス（Coarray では，プロセスを image，ID を image 番号と呼ぶ）から，スカラ変数 r の値を自身の t に，配列 x の全要素を自身の x に読み込んでいる．また，PUT では，自身のスカラ変数 r の値を id=p のプロセスの配列 x に書き出している．Coarray では，並列計算のためのデータ交換を Fortran 言語レベルで記述はできるが，MPI と同様に，ユーザが明示的に必要な通信をすべてコーディングしなければならないことからは免れることはできず，やはり，ユーザに大きな負担を要求する．このような状況を鑑み，逐次プログラムからシームレスに並列化および高性能化を支援する並列実行モデルの確立とそれに基づく超並列プログラミング言語 XcalableMP（XMP）の仕様策定と処理系の開発が行われている[10]．すでに MPI で書かれている膨大な並列プログラムをすべて新言語に書き換えることは非現実的である．このため，XMP は，MPI と対立・競合するものではなく，共存できるものとするため，MPI ライブラリと混在できる仕様としている．この機能により，並列化の大枠は，XMP を用いて容易に並列化し，特に細かなチューニングが必要な場所だけ MPI ライブラリを用いて性能向上を図ることで，高並列・超並列でも実用的な性能が期待できる．一方で，コンパイラによる自動並列化は，並列プログラミングに不慣れなユーザには有効であり，自動化だけで十分な性能を達成できるアプリケーションも，わずかではあるが存在する．しかし，高度な自動化が行われるほどプログラムとその動作に対する理解が困難となり，期待している効果を得るためには，どのようにプログラミングすればいいのかが不明瞭なため，ユーザによる性能チューニングは難しくなる．また，処理系のバージョンや動作条件により，並列性能が大きく異なる可能性が高くなり，同一プログラムにおける並列性能の互換性が悪くなる．そこで，XMP では，並列性能の互換性を重視し，プログラムからその動作が容易に想像できるレベルまで，利用者が明示的に書き下すことができる仕様としている．以上の基本方針の下に，現在，XMP では，グローバルな名前空間と変数を計算ノード間に分散配置するイメージをプログラマに提供するグローバルビューと MPI ライブラリ呼び出しの引数として記述できるローカル変数のビューであるローカルビューを提供し，これらの変数や関連する仕様が矛盾なく混在できる言語仕様とした．グローバルビューでは，通常の逐次コンパイラを使えば同じ結果を出す逐次実行が可能になるよう指示文によりベース言語（Fortran および C）を拡張している．そして，データ並列プログラミングモデルとワークシェアによって，典型的な並列化をサポートする．個別のプロセスを意識してプログラミングするローカルビューとしては，Coarray をベースにした PGAS 機能を提供し，メッセージ通信とリモート・メモリ操作を可能としている．そして，データの通信やプロセス間の同期等は，すべて明示的に記述し，並列性能のチューニングについてユーザがわかりやすく工夫できるようにする．また，柔軟性と拡張性を実現するために，基本的な実行モデルは MPI と矛盾なく共存できるようにし，より複雑に並列性能をチューニングされた MPI サブルーチンや MPI プログラムの一部分を直接利用可能にしている．また，マルチコアあるいは SMP クラスタでは，OpenMP を用いてスレッドプログラミングができるようにし，シームレスなハイブリッド並列化のためのプログラミングを可能としている．

2.6 まとめ

ノード間の通信に専用のハードウェアを実装している高性能な並列計算機だけではなく，コモディティ・コンピュータを安価なネットワークで結合した PC クラスタ型並列計算機でも，分散メモリ型のアーキテクチャが主流である．このため，並列コードを並列計算機上で実行し，その性能を十分に享受するためには，分散されたメモリを前提として並列コードを開発しなければならない．この並列プログラミングでは，並列計算機の内部アーキテクチャま

でも考慮しなければならず，非常に煩雑であり，開発者に大きな努力を要求する．しかし，それが悲しい現実であり，並列コードの開発者が並列プログラミングを始めるとき，本章がその理解の一助になれば幸いである．そして，この状況を打破すべく，HPF が普及しなかった原因を分析し，その失敗の経験に学んで言語仕様が策定され，処理系の開発が進んでいる超並列プログラミング言語 XcalableMP の今後の発展に期待していただきたい．

参考文献

[1] B. J. Smith, Int. Supercomputing Conf. 07, keynote speech, Dresden, Germany, June 26-29 (2007).
[2] http://openmp.org/
[3] http://www.mpi-forum.org/
[4] T-shirts message, Workshop on OpenMP Applications and Tools, Purdue University, West Lafayette, Indiana, US, July 30-31 (2001).
[5] H. Sutter, Dr. Dobb's Journal **30**, (2005).
[6] High Performance Fortran 2.0 公式マニュアル（シュプリンガー･フェアラーク東京，ISBN4-431-70822-7, 1999）.
[7] http://www.hpfpc.org/
[8] PC クラスタで並列プログラミング―High Performance Fortran で楽々並列化（培風館，2011），ISBN978-4-563-01586-2 (2011).
[9] http://www.co-array.org/
[10] http://www.xcalablemp.org/

坂上仁志（さかがみ ひとし）

核融合科学研究所基礎物理シミュレーション研究系，教授．仕事では，レーザー核融合に関連する粒子および流体シミュレーション，大規模並列計算の研究に従事しています．趣味では，「日本の秘湯を守る会」加盟の温泉旅館を訪ね歩いたり，未だにスキーをしたり，海外旅行も楽しんでます．