第7章

並列プログラムの実装

本章では、***CReSS*** がどのように並列計算機での実行を実現しているか、その概要を説明する。

CReSS の並列プログラムは、基本的に分散メモリ型の並列計算機で実行することを目的に設計されている。その並列プログラムは MPI（Message Passing Interface）を用いて記述されている。MPI は、現在、デファクトスタンダードな並列プログラミングライブラリであるので、***CReSS*** をほとんどの並列計算機環境で実行することが可能である。また、ワークステーションが複数台あれば、クラスタ環境で ***CReSS*** を実行することもできる。

7.1 並列化手法

7.1.1 2 次元領域分割

CReSS は分散メモリ型の並列計算機で実行することを念頭に設計されている。これは、各ノードで全く同じプロセスを実行し、計算中に必要となる他のノードが担当する領域にある値は、それらのノード間で通信をすることにより全体として 1 つの領域の計算を実行するものである。このような並列プログラムがターゲットとする計算機は、大規模な分散メモリ型の並列計算機だけでなく、ワークステーションや PC-UNIX などのクラスタにも対応できるものである。

さて、***CReSS*** では、水平方向の空間差分を陽に扱っているので、下記で述べるような水平面の 2 次元領域分割を適用すれば、並列化を実現することは容易である。また、雲微物理過程等の各種物理過程は、上下セルの参照が生じることもあるが基本的には当該セルの物理量のみで計算が可能である。また、座標系が z^* 系（ζ 系）であることも効率の良い並列化を容易にする。

まず、この水平面の 2 次元領域分割による並列化を採用したときの利点を挙げる。

- 座標系が地上付近に欠損が生じない z^* 系（ζ 系）なので、水平方向の分割に対して、各ノードが担当する計算量がほぼ等しくなる。
- つまり、ロードバランスに優れる（他のノードの処理を待つことが少ない）。
- 通信量は、水平格子数が $ni \times nj$ のとき、$ni + nj$ となり、総格子数の増加に対する相対的な通信量の増加が少ない。
- つまり、並列度が同じ場合、総格子数が大きな計算ほど並列効率が増す。
- よって、非常に大規模な計算に適合する。

次に、2 次元領域分割の概念図を示す。

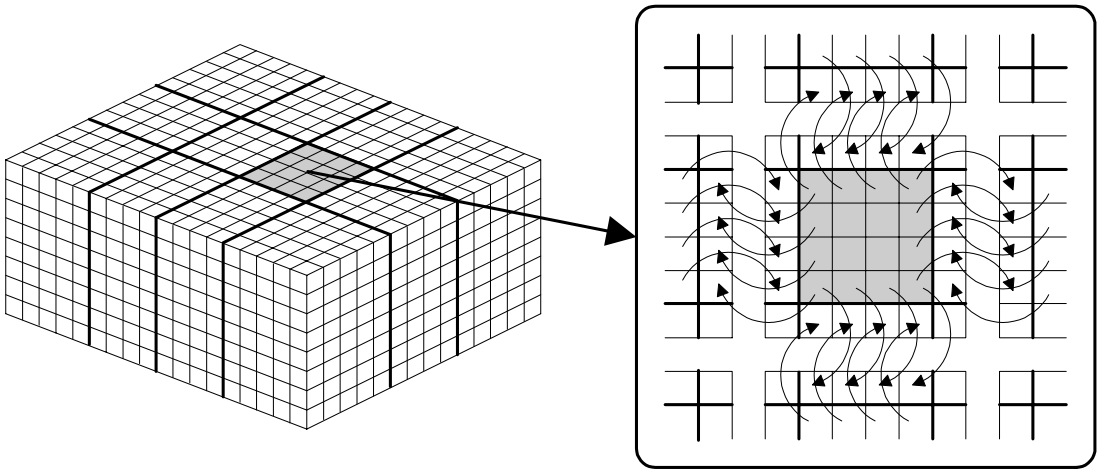

図 7.1. 2 次元領域分割と袖領域の値の交換の様子。

図 7.1 は、2 次精度で計算をする場合の並列化の模式図である。2 次精度で中心差分近似をする場合、ある 1 つの点が参照しなければならない格子点は各方向に ± 1 の 6 点である。ここで考えるのは水平方向の 2 次元領域分割であるので、図のとおり、鉛直方向には参照の問題は生じないが、水平方向には分割の境にある点の近似解を求める際に、参照すべき点の値が他のノードが担当する領域に存在することになるので問題が生じてしまう。そこで、右側の拡大図にあるように、それぞれの領域を囲むように取られた袖領域を重ねることにより、全体を 1 つのノードで実行したときと全く同様の結果を得ることができるようにする。そのためには、袖領域の値が必要となる度に隣り合う領域の該当する部分から値を得るように通信を実行すればよい（節 7.1.2 にて詳細は示す）。

なお、***CReSS*** は 4 次精度の計算もできるようになっているが、そのためには 2 次精度の場合と全く同様にして、袖領域を二重にすることにより必要となる値を余分に通信すればよい。現実的ではないが、全く同様の実装方法で、差分の精度を 6 次、8 次 … と上げていくことができる。

7.1.2 並列化の具体例

ここでは、節 7.1.1 で述べた袖領域の値の交換が、具体的にどのように実装されているのかを示す。

実際のプログラミングは MPI（Message Passing Interface）を用いて記述されている。MPI は、このような分散メモリ型の計算機向けのプログラミングに必要な各種通信ルーチンの、C や Fortran の関数やサブルーチンの呼び出し形式を定めたインターフェース仕様である。あくまでもインターフェース仕様であり、具体的な実装方法は定めていない。

なお、MPI についての詳細は、以下のホームページを参照するとよい。

http://www.mpi-forum.org

さて、図 7.1 のような通信を MPI によって実現するには、境界面を除いてシフトを東西南北各方向に繰り返せばよい。図 7.2 は、境界面を除いた、西側の袖領域の東側の袖領域へのシフトの例である。この図で、*nipe* は東西方向のノード数、*ipe* はノード番号を表す。

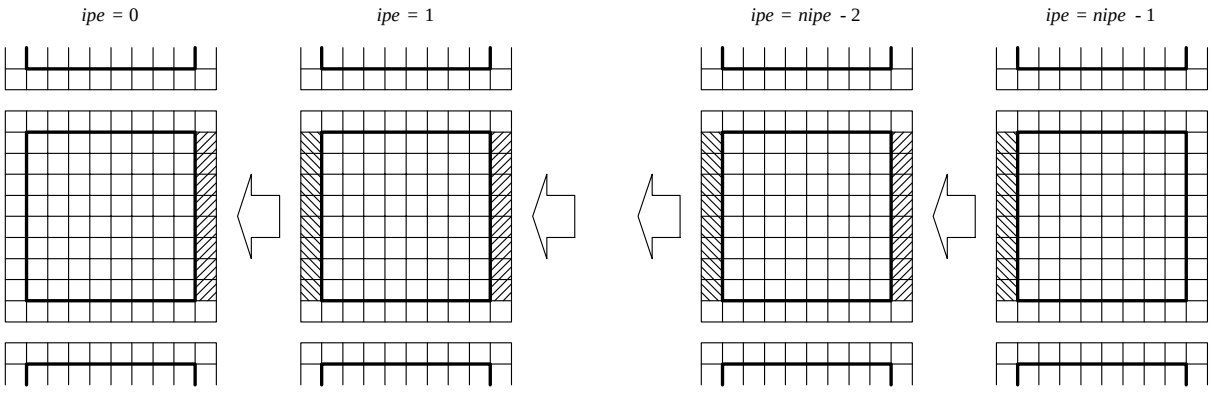

図 7.2. 西側の袖領域の東側の袖領域へのシフトの例。

また、***CReSS*** におけるこれを実現するソースコードは以下のとおりである。ここでは一部を示したに過ぎないので、より詳しくは Src ディレクトリの exchansn.f と exchanwe.f を参考にするとよい。

```
      ! 西端のノードは、送信バッファなし
      if(ipe.eq.0) then
        dst=mpi_proc_null
      end if

      ! 東端のノードは、受信バッファなし
      if(ipe.eq.nipe-1) then
        src=mpi_proc_null
      end if

      ! 西端のノード以外の送信バッファの設定
      if(ipe.ne.0) then

        do 140 k=kbmin,kbmax
        do 140 j=jbmin,jbmax
          ib=(k-kbmin)*jbmxn1-jbmin1+j

          sbuf(ib)=var(iwsend,j,k)

140     continue

      end if

      ! 送受信のための MPI ルーチンの呼び出し
      call mpi_sendrecv(sbuf,siz,mpi_real,dst,tag,rbuf,siz,mpi_real,src,
     .                  tag,mpi_comm_world,stat,ierr)

      ! 東端のノード以外の受信バッファの設定
      if(ipe.ne.nipe-1) then

        do 170 k=kbmin,kbmax
        do 170 j=jbmin,jbmax
          ib=(k-kbmin)*jbmxn1-jbmin1+j

          var(ierecv,j,k)=rbuf(ib)

170     continue

      end if
```

なお、ここでは mpi_sendrecv によってシフトを実装しているが、これを、mpi_send と mpi_recv によって実装しないほうがよい。できないことはないが、様々なプログラミングが考えられ、場合によってはデッドロックしてしまう可能性がある。また、MPI の公式仕様では、これによりデットロックするかしないかを定めていないようであるので、プログラムの動作の保証ができなくなってしまう。

7.2 並列プログラムの検査

7.2.1 計算結果の一致の検査

前節で述べたように、ここで採用した並列化は基本的にシフトを行うだけでよいので、当然のことながら、複数ノードを用いて実行した結果と 1 つのノードを用いて実行した結果は、完全に一致しなければならない（全体の平均値を求めるような計算の場合には、完全に一致するとは限らない。シフトを行う場合にはその値に演算を適用することがないので、不一致は起こりえない）。***CReSS*** では、結果が完全に一致することを次のように確認している。

複数ノードを用いて計算した場合、結果は各ノード毎に出力される。ここで、ファイル名の *exprim* は実験名である（節 9.1 を参照）。

例： *exprim*.dmp*xxxxxx*.pe0000.bin ～ *exprim*.dmp*xxxxxx*.pe*yyyy*.bin

これらのファイルを 1 つにまとめ、並びの書式つきテキストファイルで出力するようなプログラムを作成し（ポストプロセッサ *unite* でも各ノード毎に出力された結果のファイルを 1 つにまとめることができるが、直接探査の書式なしバイナリファイルを出力するようになっている）、1 つのノードで計算した結果を同じ形式のテキストファイルで出力した結果と比較して完全一致を確かめればよい。

CReSS では、様々な実験に対してこの検証を行い、並列化に問題がないことを確認している。

7.2.2 プログラムの並列化効率

最後に、全体で同じ格子数の計算を、様々なノード数で実行したときの並列化効率を示す。ここでは、$67 \times 67 \times 35$ の格子数で 50 ステップ実行した結果を示す。用いた計算機は日立 SR2201 である。

図 7.3 に見られるように、ノードの数が増えるに従って計算時間はほぼ線型に減少している。単純にみて、***CReSS*** は効率よく並列計算できることが分かる。図 7.4 の並列化効率に関しては、除々に効率が下がっていく様子が見られる。しかし、これは当然のことで、ここで用いた格子数が $67 \times 67 \times 35$ と比較的小さく、また、このように一定の格子数で計算する以上、ノード数が大きくなれば相対的な通信量が増えていくためである。この結果は、***CReSS*** の並列化効率が悪いことを示す結果ではなく、各ノードの担当する格子数が大きくなればより効率的に計算できることを示している。計算が大規模になればなっただけ、多くのノードを用いて計算する価値が出てくるのである。

なお、図 7.4 において多少グラフが波打っているのは、2 次元領域分割の際の分割方法の違いによるものである（例えば 4 ノードを用いて計算するときには、1×4、2×2、4×1 のいずれかの分割方法があり、通信量が変わる）。

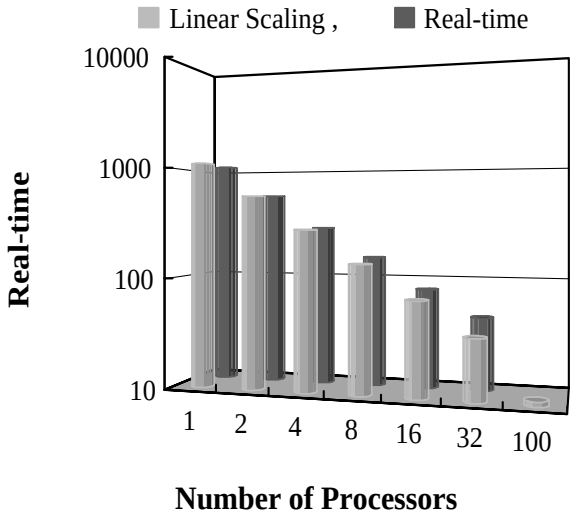

図 7.3. 並列テストで得られた実計算時間［s］。

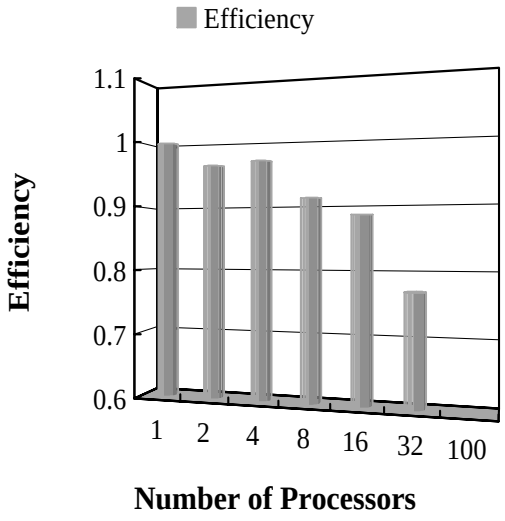

図 7.4. 並列テストで得られた並列化効率。

このように、アルゴリズムにおいても効率においても問題が無いように示したが、全く問題がないわけではない。雲微物理過程のプログラムにおいては、雲の有無によって計算量が変化する。例えば、あるノードが担当する領域にはほとんど雲が発生していないが、他のノードが担当する領域には雲が発生しているような場合、雲が発生している領域を担当するノードの処理を待たなければならないことになり、効率が下がることが考えられる。現状では、最初に設定した分割法を計算途中で効率が上がるように変更するというような高度な並列化は実装していない。