［大規模科学計算システム］高速化推進研究活動報告第 5 号より転載

並　列　処　理

スーパーコンピューティング研究部　江川隆輔　岡部公起
情報部情報基盤課　伊藤英一　小野敏　山下毅
日本電気株式会社　撫佐昭裕　神山典　小久保達信　金野浩伸
NEC システムテクノロジー株式会社　曽我隆　塩田和永

6.1 並列処理の概要

並列処理とは，並列に実行可能な処理を複数のプロセッサ（あるいはコア）を用いて実行することである．単一のプロセッサでのみ実行可能なプログラムを逐次処理プログラムと呼び，逐次処理プログラムを並列処理プログラムに書き換えることを並列化と言う．並列化を行うことにより，プログラムの実行時間を短縮することができる．

並列処理には，コンピュータアーキテクチャに応じた以下の方式がある．

(1) 共有メモリ型並列処理

共有メモリ型並列処理とは，図 6.1-1 のように複数のプロセッサが単一のメモリ空間を共有しながら並列処理を行うものである．これは，コンパイラの自動並列化機能やコンパイル指示行の挿入により並列化を行うものである．東北大学サイバーサイエンスセンターにおける SX-9 では 16 並列，Express5800 では 32 並列まで実行することができる．

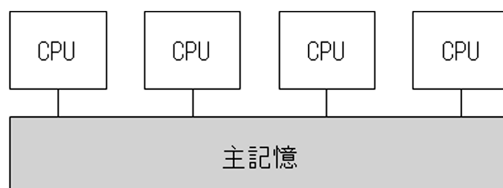

図 6.1-1 共有メモリマシンのアーキテクチャ

(2) 分散メモリ型並列処理

分散メモリ型並列処理とは，図 6.1-2 のようにネットワークを介して接続されている，複数の独立したメモリ空間を持つコンピュータで行われる並列処理のことである．この並列処理では，処理を分割し異なるプロセッサで別々に実行する．このとき，プロセッサ間で情報交換が必要な場合，MPI（Message Passing Interface）等の通信ライブラリを用いてデータのやり取りを行う．東北大学サイバーサイエンスセンターにおける SX-9，Express5800 では共に 64 並列まで実行することができる．

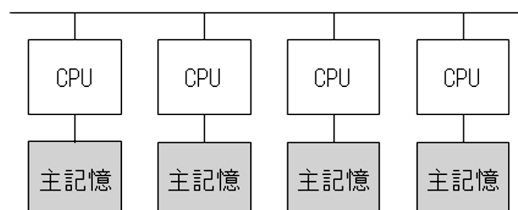

図 6.1-2 分散メモリマシンのアーキテクチャ

6.2 MPI プログラムの概要

6.2.1 MPI とは

MPI とは，MPI フォーラム(注 1)により開発・規格化された分散メモリ型並列処理におけるデータ通信のための標準規格であり，複数のプロセス(注 2)間でデータをやり取りするために用いるメッセージ通信操作のサブプログラム仕様の規格である.

主な特徴を以下に示す.

- FORTRAN，C から呼び出す通信操作のサブプログラムであり，ライブラリで提供される.
- 様々なコンピュータシステムの MPI 実装環境において同じソースコードで利用できる.
- 共有メモリ型並列処理に比べてプログラマの負担が大きいが，大規模計算が行える.

(注 1)MPI フォーラムの公式 Web ページは http://www.mpi-forum.org である.
(注 2)プロセス:コンピュータのプロセッサ上で自律的に動作するプログラムの実体

6.2.2 MPI のプログラム例

ここでは，総和計算を行う FORTRAN プログラムを例に MPI プログラムの説明を行う.

(1) 総和計算のプログラム

図 6.2-1 は，1 から 100 までの整数の和を求めるコードである.

```
program main
parameter(N=100,i1=1,i2=N)
isum = 0
do i = i1, i2
  isum = isum + i
enddo
write(6,*) "sum=",isum
stop
end program
```

図 6.2-1 総和計算のコード例

図 6.2-1 に示した総和計算の例では，1 から 100 まで順に和を求めている. この総和計算では，DO ループを複数に分割し，分割したループごとに部分和を取り，それぞれの部分和を集計して求めることができる. このように DO ループを分割した部分について，部分和を並列に実行することで並列処理することができる.

(2) 処理の分割

図 6.2-2 は，DO ループを均等に 4 分割し，4 プロセスで実行する場合の概念図である. 各プロセスがそれぞれ DO ループのどの部分を担当するか，プロセス識別番号(0～3)で表している. MPI ではプロセスの識別番号のことを rank 番号と呼ぶ. 各プロセスは担当する DO ループの部分和を求める計算を行う.

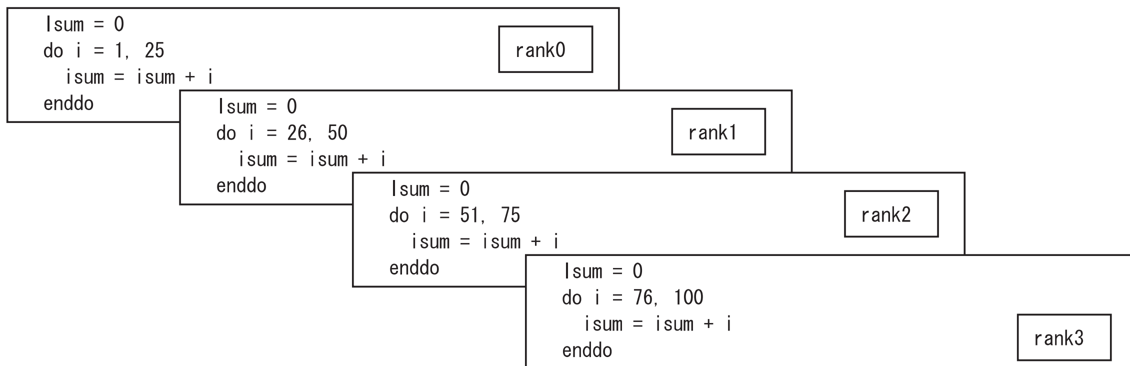

図 6.2-2　DO ループ分割の概念図

図 6.2-2 は並列数 4 の例であるが，プロセス数に応じて，各プロセスが分担する領域の始点と終点を下式①，②のように rank 番号から求める必要がある．ここで nprocs が全プロセス数を，myrank が各プロセスの rank 番号を表している．ただし，各プロセスでループを均等に分割できることを前提にしている．

①　始点　ist = ((i2-i1+1)/nprocs)*myrank+1
②　終点　ied = ((i2-i1+1)/nprocs)*(myrank+1)

この式によって求められる ist，ied は表 6.2-1 のとおりである．nprocs，myrank を使用することにより，図 6.2-3 のように各プロセスの処理を共通のコードで表すことができる．

表 6.2-1　rank 番号ごとの ist，ied の値

nprocs=4	ist	ied
myrank=0	1	25
myrank=1	26	50
myrank=2	51	75
myrank=3	76	100

```
ist = ((i2-i1+1)/nprocs)*myrank+1
ied = ((i2-i1+1)/nprocs)*(myrank+1)
isum = 0
do i = ist, ied
  isum = isum + i
enddo
```

図 6.2-3　部分和のコード例

(3)　プロセス間通信

全体の総和は，各プロセスで求められた部分和を代表プロセスで集計して求める．集計には MPI による通信ライブラリを使用する．図 6.2-4 は，プロセス間の通信と部分和を集計する例であり，データの送受信を行う MPI の一対一通信サブルーチン（MPI_RECV，MPI_SEND）を用いている．rank 番号が 0 以外のプロセスは，それぞれの部分和を rank 番号 0 に送信する．rank 番号 0 のプロセスは，自プロセス以外のすべてのプロセスから送られる部分和を（nprocs-1）回受信し，受信した部分和 itmp を isum に集計する．

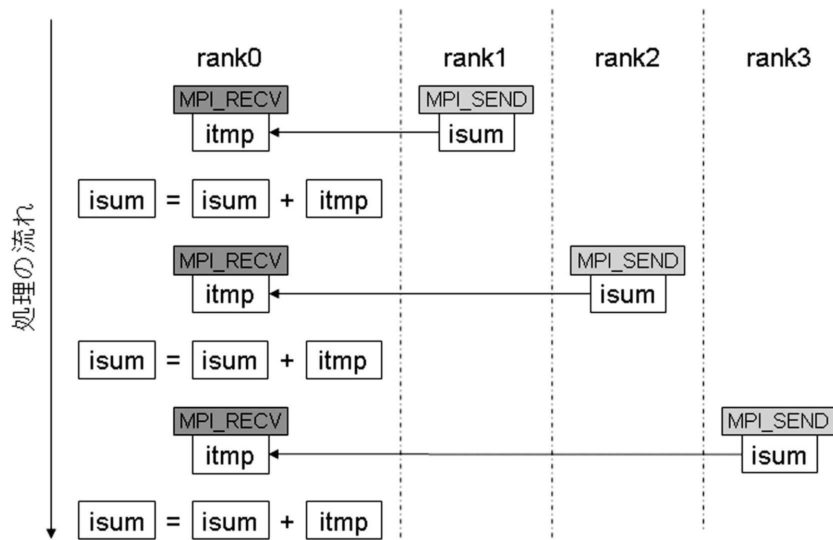

図 6.2-4　総和処理の流れ

(4)　総和計算の MPI コード例

以上を総合して，図 6.2-1 のプログラムを並列化したコードを図 6.2-5 に示す．

```
program example1
include 'mpif.h'                                                    !①
integer status(MPI_STATUS_SIZE)                                     !②
parameter(N=100, i1=1, i2=N)
integer nprocs,myrank,source,tag,ierr,ist,ied,i,isum,itmp
call MPI_INIT(ierr)                                                 !③
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)                      !④
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)                      !⑤

ist = ((i2-i1+1)/nprocs)*myrank+1
ied = ((i2-i1+1)/nprocs)*(myrank+1)
isum = 0
do i = ist, ied
  isum = isum + i
enddo

if(myrank .eq. 0) then
  do i = 1, nprocs-1
    source = i
    tag = i
    call MPI_RECV(itmp,1,MPI_INTEGER,source,tag,MPI_COMM_WORLD, status,ierr)   !⑥
    isum = isum + itmp
  enddo
else
  tag = myrank
  call MPI_SEND(isum,1,MPI_INTEGER,0,tag,MPI_COMM_WORLD,ierr)       !⑦
endif
if(myrank .eq. 0) write(6,*) "sum=",isum
call MPI_FINALIZE(ierr)                                             !⑧
stop
end
```

図 6.2-5　並列化コード例（総和計算）

この例で使用したMPIの宣言部分とサブルーチンについて以下に示す．

① mpif.h
MPIで予約されている定数が定義されている．MPIサブルーチンを利用するすべてのサブルーチン，関数においてinclude 文で引用する必要がある．

② status(MPI_STATUS_SIZE)
下記MPI_RECVの第七引数で使用されるメッセージ情報．

③ MPI_INIT
MPIの実行開始を宣言するサブルーチンであり，すべてのMPIサブルーチンに先立ち，呼び出される必要がある．引数(ierr)に実行結果の状態を受け取る(以下同様)．

④ MPI_COMM_SIZE
MPI のプロセス管理を行うサブルーチンであり，総プロセス数の問い合わせを行う．第一引数はMPI通信のためのコミュニケータ(注1)“MPI_COMM_WORLD”を指定する．第二引数(nprocs)に総プロセス数を受け取る．

⑤ MPI_COMM_RANK
MPIのプロセス管理を行うサブルーチンであり，自プロセスのrank番号の問い合わせを行う．第二引数(myrank)に自 rank 番号を受け取る．

⑥ MPI_RECV
MPIの通信処理を行うサブルーチンであり，データの受信をブロッキング(同期)モード(注2)で行う．このサブルーチンは，一対一通信と呼ばれる分類に属し，通信相手が必要になる．送信側が送ったデータを受信するために呼び出される．第一引数(itmp)は受信データの開始アドレス，第二引数(1)は受信データの要素数，第三引数(MPI_INTEGER)は受信データのタイプ，第四引数(source)は通信相手の rank 番号，第五引数(tag)はタグ(送受信は同じタグ間で行われる)，第六引数(MPI_COMM_WORLD)はコミュニケータ，第七引数(status)はメッセージ情報を受け取る．

⑦ MPI_SEND
MPI の通信処理を行うサブルーチンであり，データの送信をブロッキング(同期)モードで行う．このサブルーチンは，一対一通信と呼ばれる分類に属し，通信相手が必要になる．受信側にデータを送信するために呼び出される．第一引数(isum)は送信データの開始アドレス，第二引数(1)は送信データの要素数，第三引数(MPI_INTEGER)は送信データのタイプを示す．以降の引数はMPI_RECVと同じである．

⑧ MPI_FINALIZE
MPI の終了処理を行うものであり，すべての MPI サブルーチンが呼ばれた最後で呼び出す必要がある．

(注 1)コミュニケータ：通信に参加する MPI プロセスを集めてできたグループにつけられる一種の名前のようなものである．この識別は通信相手のプロセスを特定する場合に必要となる．利用者は，プログラムにおいて複数のコミュニケータを定義することもできる．

(注 2)ブロッキング(同期)モード：対応する送信手続きと受信手続きのどちらかを先に呼び出した場合に，両方の呼び出しが揃うまで先行した側の MPI プロセスは待ち合わせを行う処理モードである．

6.3 領域分割法

6.3.1 領域分割法の例

領域分割法とは，図 6.3-1 のようにシミュレーションの対象となる空間を分割し，複数のプロセスで処理を分担して実行するものである．分割した空間の間で，それぞれのプロセスがデータの参照を必要とする場合，MPI を用いてデータの転送を行う．

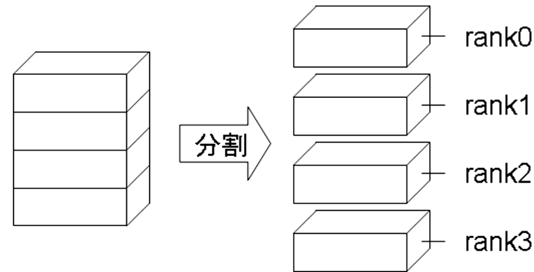

図 6.3-1 領域分割

図 6.3-2 に，差分コードの例を示す．この差分コードの領域を均等に 2 分割し，2 プロセスで実行する場合のコードを図 6.3-3 に示す．

```
 do i = 1, 10
   a(i) = i
enddo
 do i = 2, 10
   b(i) = a(i) - a(i-1)
enddo
```

図 6.3-2 差分コード

```
 do i = 1, 5                 rank0
   a(i) = i
enddo
 do i = 2, 5
   b(i) = a(i) - a(i-1)
enddo
```

```
 do i = 6, 10                rank1
   a(i) = i
enddo
 do i  = 6, 10
   b(i) = a(i) - a(i-1)
enddo
```

図 6.3-3 領域分割のコード

図 6.3-3 のように分割する場合，rank 番号 1 は i=6 のときの計算に必要な a(5)のデータが領域外となる．このとき，図 6.3-4 で示されるように a(5)のデータは rank 番号 0 の領域内にあり，rank 番号 1 は rank 番号 0 から a(5)のデータを受け取る必要がある．

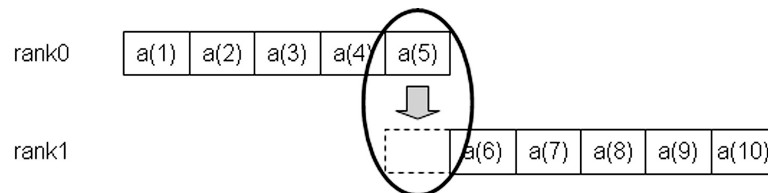

図 6.3-4 境界面で不足するデータのイメージ

6.3.2 領域分割法におけるデータ転送

領域分割法では，自プロセスで演算に必要となるデータが担当する領域の外にある場合，そのデータが含まれる領域を担当するプロセスから，データを受け取る必要がある．このデータ転送がオーバヘッドとなり，並列性能を低下させる要因となる場合がある．

図 6.3-5 はポアソン方程式をヤコビ法で解くコードである．ここでは i，j，k の三次元空間を扱っている．これを用いて領域分割の違いによるデータ転送のコストについて説明する．

```
30: +------>         do loop =1,n
31: |+----->           do k = 2,kmax-1
32: ||+---->             do j = 2,jmax-1
33: |||V--->               do i = 2,imax-1
34: ||||                     u(i,j,k)=s*(uu(i+1,j,k)+uu(i-1,j,k)
35: ||||           &                   +uu(i,j+1,k)+uu(i,j-1,k)
36: ||||           &                   +uu(i,j,k+1)+uu(i,j,k-1)
37: ||||           &                   +a(i,j,k))
38: |||V---                enddo
39: ||+----              enddo
40: |+-----            enddo
41: |                  err=0.0d0
42: |+----->           do k = 2,kmax-1
43: ||+---->             do j = 2,jmax-1
44: |||V--->               do i = 2,imax-1
45: ||||                     err=err+(uu(i,j,k)-u(i,j,k))*(uu(i,j,k)-u(i,j,k))
46: |||V---                enddo
47: ||+----              enddo
48: |+-----            enddo
49: |                  if(err .lt. eps) go to 100
50: +------            end do
51: +------>           do k = 2,kmax-1
52: |+----->             do j = 2,jmax-1
53: ||V---->               do i = 2,imax-1
54: |||                      uu(i,j,k)=u(i,j,k)
55: ||V----                enddo
56: |+-----              enddo
57: +------            enddo
58:             100  continue
```

図 6.3-5 ポアソン方程式をヤコビ法で解くコード

三次元空間を j 方向または k 方向で 4 分割することを考える．ただし，配列 uu はサイズ(imax,n,n)の倍精度実数とする．図 6.3-5 の 34-37 行目の配列 uu は 3 次元のそれぞれの方向に差分式となっている．どの方向に分割しても隣接しているデータ(境界面)を参照しているため，MPI を用いてデータ転送する必要

がある．以下にｋ方向，ｊ方向に分割したときの通信量，通信回数について説明する．

(1)　k 方向に分割したとき

図 6.3-5 の 34-37 行目の配列 uu は分割した空間の境界面のデータを参照（3 次元目が k+1, k-1）しているため，MPI を用いて転送する必要がある．この例では，MPI_SEND，MPI_RECV を用いて転送する．k 方向に分割した場合，境界は ij 平面となりメモリ上に連続に配置されているので，1 つの境界面あたり 1 回の通信でデータを送受信することができる．図 6.3-6 に rank 番号 1 におけるデータ転送を示す．1 度の通信で（imax×n×8）Byte のデータを転送し，必要な転送は 2 回のみとなる．

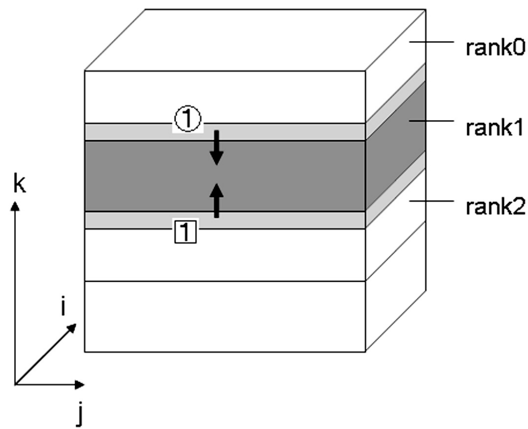

図 6.3-6　k 方向で分割したときのデータ転送

図 6.3-7 は上に示したデータ転送の通信例である（数字は図 6.3-6 と対応）．rank 番号 1 は，rank 番号 0，2 から境界面のデータを受信する必要がある．

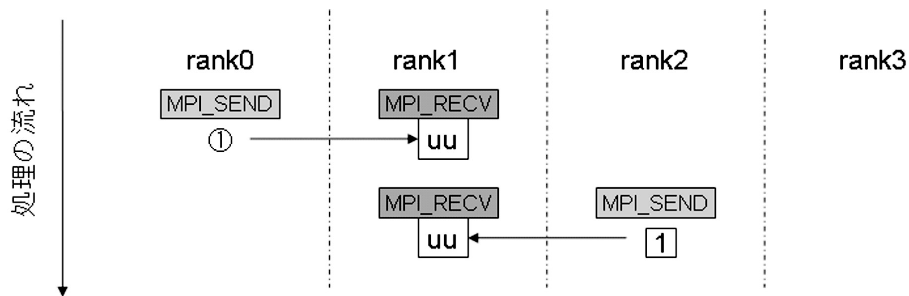

図 6.3-7　k 方向で分割したときの通信

(2)　j 方向に分割したとき

配列 uu は分割した空間の境界面のデータを参照（2 次元目が j+1, j-1）しているため，MPI を用いて転送する必要がある．図 6.3-8 に MPI_SEND，MPI_RECV を使用した場合の rank 番号 1 におけるデータ転送を示す．メモリ上に連続に配置されているデータは i 方向のみであるため，1 度の通信で転送できるデータは（imax×8）Byte であり，（n×2）回の転送が必要となる．

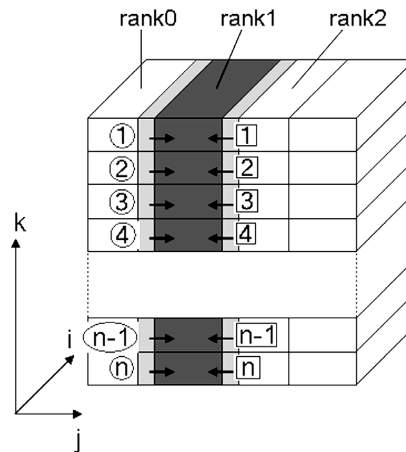

図 6.3-8　j 方向で分割したときのデータ転送

図6.3-9は上に示したデータ転送のプロセス間の通信例である(数字は図6.3-8と対応). rank番号1は, rank番号0, 2から境界面のデータを受信する必要がある.

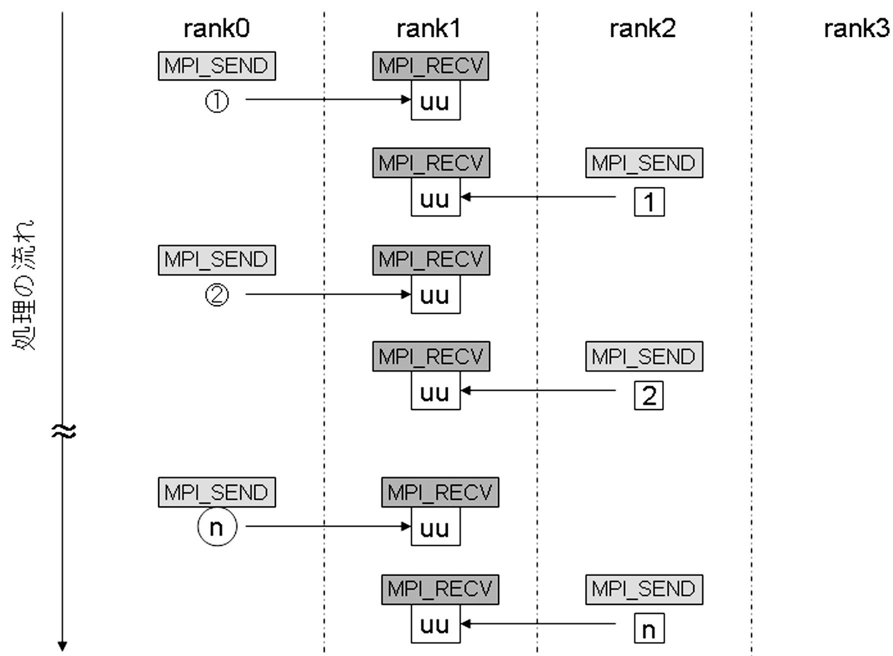

図 6.3-9　j 方向で分割したときの通信例

6.3.3 領域分割法による通信性能

図6.3-5にあるポアソン方程式をヤコビ法で解くコードを, 問題サイズ「imax=1280, jmax=128, kmax=128」として性能評価を行う. このコードをMPIで分散並列化し, 16プロセスで並列実行するときの通信量, 呼び出し回数, 総通信量を表6.3-1に示す. jmax=kmaxであるため, j, kの分割方向によらず総通信量は同じである.

表 6.3-1　通信量，呼び出し回数，総通信量

	1 回あたりの MPI 通信量	呼び出し回数	総通信量
k 方向に分割	約 1.25MB	20,000	約 25,000MB
i 方向に分割	約 10KB	2,560,000	約 25,000MB

上記条件で，SX-9とExpress5800それぞれで実行する．図6.3-10に分割方向の違いによる通信時間のグラフを示す．総通信量は同じであるが，SX-9，Express5800ともに，通信回数の少ないk方向の分割の方が通信時間は短いことがわかる．このように並列性能の向上のためには，通信回数が少なく，また 1 度の通信で多くのデータを送受信できる適切な分割方法を検討することが重要になる．

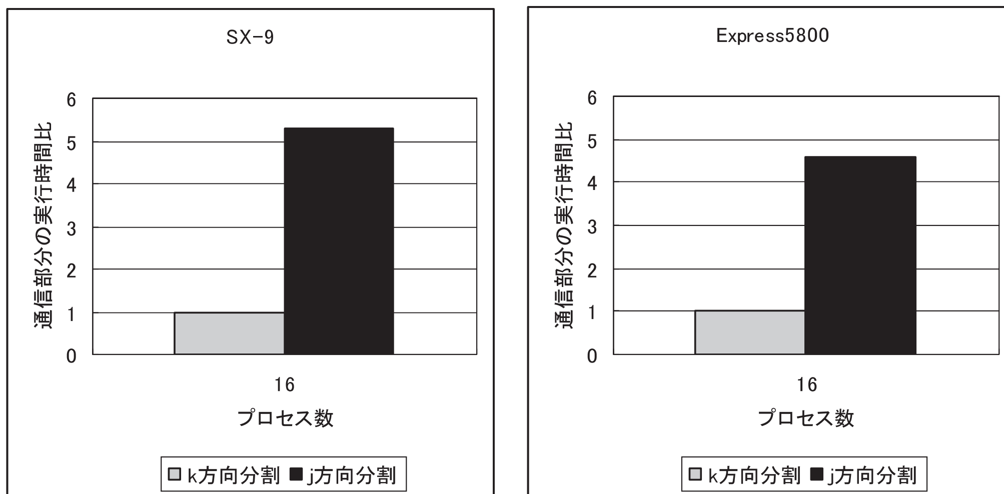

図 6.3-10　分割方向の違いによる通信時間

6.4　領域分割法における MPI コード例

本節では，6.3 節のポアソン方程式を領域分割法によって MPI 化した実際のコードについて説明する．

6.4.1　MPI による隣接領域間の転送

6.3 節でも述べたが，図 6.4-1 の 34-37 行目の配列 uu は分割した空間の境界面のデータを参照している差分式であるため，隣接領域間の転送が必要である．

```
30: +------>          do loop =1,n
31: |+----->            do k = 2,kmax-1
32: ||+---->              do j = 2,jmax-1
33: |||V--->                do i = 2,imax-1
34: ||||                      u(i,j,k)=s*(uu(i+1,j,k)+uu(i-1,j,k)
35: ||||             &                   +uu(i,j+1,k)+uu(i,j-1,k)
36: ||||             &                   +uu(i,j,k+1)+uu(i,j,k-1)
37: ||||             &                   +a(i,j,k))
38: |||V---                 enddo
39: ||+----               enddo
40: |+-----             enddo
 :
57: +------             enddo
```

図 6.4-1　ポアソン方程式における差分式（逐次処理コード）

図 6.4-2 に領域を分割したとき，差分式の演算で必要な境界面のデータ通信を示す．k 方向に領域を 4 分割すると，①～④の切り出した領域を各プロセスが分担する．分割した領域には隣接した空間の境界面のデータをそれぞれ冗長に持つ必要がある．具体的には，①は B，②は A と D，③は C と F，④は E の境界面のデータを保有する．ただし，これら境界面は隣接したプロセスが演算結果を更新するため，自プロセスでは更新前の古いデータを保持することなる．そのため，各プロセスは隣り合う rank 番号に対して更新されたデータの送受信を行う必要がある．

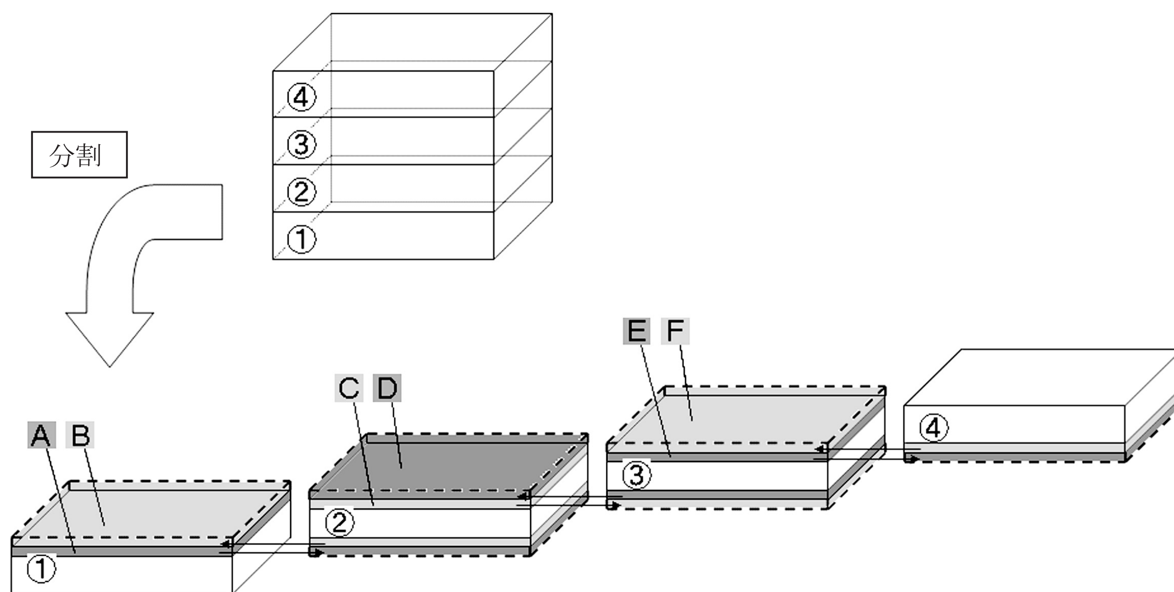

図 6.4-2　領域分割とデータ通信

図 6.4-3 に MPI_SEND，MPI_RECV を用いたコード例を示す．この例では分割する 3 次元目のサイズは実行するプロセス数で割り切れるものとする．21-59 行目では，k 方向に分割したときの各プロセスが必要とするデータ(作業配列，ループ長とループの始点，終点)を作成し，71-80 行目では，分割された領域内でポアソン方程式を解く．94-117 行目で更新された境界面のデータを転送する．

```
 : !各プロセスが担当する３次元目の大きさを計算
 21:                kmax = kmax/nprocs
 : !各プロセスにおける配列ｕの３次元目の始点を計算
 23:                  ukst =myrank*kmax
 : !境界面のデータを格納する領域を持つ配列 uu,各プロセスの配列ｕの値を保持する作業配列 wu の確保
 29:                allocate(uu(imax,jmax,0:kmax+1),wu(imax,jmax,kmax))
 : !各プロセス毎の始点，終点の定義
 50:                if(myrank .eq. 0)then
 51:                  kst=2
 52:                else
 53:                  kst=1
 54:                endif
 55:                if(myrank .eq. nprocs-1)then
 56:                  ked=kmax-1
 57:                else
 58:                  ked=kmax
 59:                endif
 :
 71: +------>       do loop =1,n
 72: |+----->         do k = kst,ked
 73: ||+---->           do j = 2,jmax-1
 74: |||V--->             do i = 2,imax-1
 75: ||||                   wu(i,j,k)=s*(uu(i+1,j,k)+uu(i-1,j,k)
 76: ||||          &                   +uu(i,j+1,k)+uu(i,j-1,k)
 77: ||||          &                   +uu(i,j,k+1)+uu(i,j,k-1)+a(i,j,k))
 78: |||V---               enddo
 79: ||+----             enddo
 80: |+-----           enddo
 : !隣接領域間の転送
 94: |              if(myrank.ne.0)then
 95: |                dest=myrank-1
 96: |                tag=myrank
 97: |                call MPI_SEND(wu(1,1,kst),imax*jmax,MPI_DOUBLE_PRECISION,
 98: |             &                 dest,tag,MPI_COMM_WORLD,IERR)
 99: |              endif
100: |              if(myrank.lt.nprocs-1)then
101: |                source=myrank+1
102: |                tag=myrank+1
103: |                call MPI_RECV(uu(1,1,ked+1),imax*jmax,MPI_DOUBLE_PRECISION,
104: |             &                 source,tag,MPI_COMM_WORLD,status,IERR)
105: |              endif
106: |              if(myrank.ne.0)then
107: |                source=myrank-1
108: |                tag=myrank-1
109: |                call MPI_RECV(uu(1,1,kst-1),imax*jmax,MPI_DOUBLE_PRECISION,
110: |             &                 source,tag,MPI_COMM_WORLD,status,IERR)
111: |              endif
112: |              if(myrank.lt.nprocs-1)then
113: |                dest=myrank+1
114: |                tag=myrank
115: |                call MPI_SEND(wu(1,1,ked),imax*jmax,MPI_DOUBLE_PRECISION,
116: |             &                 dest,tag,MPI_COMM_WORLD,IERR)
117: |              endif

130: +------        enddo
```

図 6.4-3　ポアソン方程式における差分式（MPI コード）

6.4.2 収束判定のための総和演算

図 6.4-4 は，ヤコビ法の収束判定を行っている箇所を抜き出したものである．49 行目の変数 err の値によって収束判定を行っている．

```
30: +------>        do loop =1,n
 :
41: |                 err=0.0d0
42: |+----->          do k = 2,kmax-1
43: ||+---->            do j = 2,jmax-1
44: |||V--->              do i = 2,imax-1
45: ||||                    err=err+(uu(i,j,k)-u(i,j,k))*(uu(i,j,k)-u(i,j,k))
46: |||V---               enddo
47: ||+----             enddo
48: |+-----           enddo
49: |                 if(err .lt. eps) go to 100
 :
57: +------           enddo
58:           100  continue
```

図 6.4-4　ヤコビ法の収束判定（逐次処理コード）

MPI コードでは図 6.4-4 の 42-48 行目のループは分割されているため，変数 err は部分和となる．各プロセスの部分和を集計し，総和を求める必要がある．6.2 節の総和計算では，一対一通信の MPI_RECV と MPI_SEND のサブルーチンを使用して集計したが，ここでは総和演算を MPI の集団通信のサブルーチンで集計する例を図 6.4-5 に示す．

```
 71: +------>        do loop =1,n
  :
 81: |                 err=0.0d0
 82: |+----->          do k = kst,ked
 83: ||+---->            do j = 2,jmax-1
 84: |||V--->              do i = 2,imax-1
 85: ||||                    err=err+(uu(i,j,k)-wu(i,j,k))
 86: ||||          &                 *(uu(i,j,k)-wu(i,j,k))
 87: |||V---               enddo
 88: ||+----             enddo
 89: |+-----           enddo
 90: |                   CALL MPI_ALLREDUCE(err,all_err,1,MPI_DOUBLE_PRECISION,    !①
 91: |             &                              MPI_SUM,MPI_COMM_WORLD,IERR)
 92: |                   if (myrank.eq.0) write(*,*) all_err,loop
 93: |                 if(all_err .lt. eps) go to 100
  :
130: +------          enddo
131:
132:            100 continue
```

図 6.4-5　ヤコビ法の収束判定（MPI コード）

この例で使用した MPI サブルーチンについて以下に示す．

① MPI_ALLREDUCE

MPI の集団通信を行うサブルーチンであり，複数のプロセスからそれぞれのデータを受信し，リダクション演算を行う．集団通信は，同一コミュニケータの全プロセスが対象になる．第五引数に MPI_SUM を与えると，第一引数で与えられた各プロセスのデータの総和を第二引数で受け取る．

6.5　デッドロック

MPI プログラムの実行時には，プログラムの書き方によって各プロセスが送受信待ち状態となり次の処理へ進めなくなる場合がある．この状態をデッドロックと言う．図 6.5-1 はデッドロックが発生する場合のコードである．

```
do i = 0,nprocs-1
  if(myrank==i)then
    k=mod(i+1,nprocs)
    itag=k
    call MPI_SEND(b(1+i*it),it,MPI_INTEGER,k,itag,MPI_COMM_WORLD,ierr)
  endif
enddo

do i = 0,nprocs-1
  if(myrank==i)then
    k=mod(i-1+nprocs,nprocs)
    itag=i
    call MPI_RECV(b(1+i*it),it,MPI_INTEGER,k,itag,MPI_COMM_WORLD,status,ierr)
    endif
enddo
```

図 6.5-1　デッドロックのコード例

MPI_SEND は 6.2 節で述べたようにブロッキングモードで通信を行うため，データの受信が完了しないと次の処理に進めない．つまり，MPI_SEND に対して，受信側プロセスで MPI_RECV が実行されるまで処理は中断される．図 6.5-2 は図 6.5-1 を 2 並列で実行したときのイメージである．この例では，rank 番号 0 は rank 番号 1 が MPI_RECV を呼ぶまで MPI_SEND で待ち続け，rank 番号 1 は rank 番号 0 が MPI_RECV を呼ぶまで MPI_SEND で待ち続けることになり処理が進まなくなる．

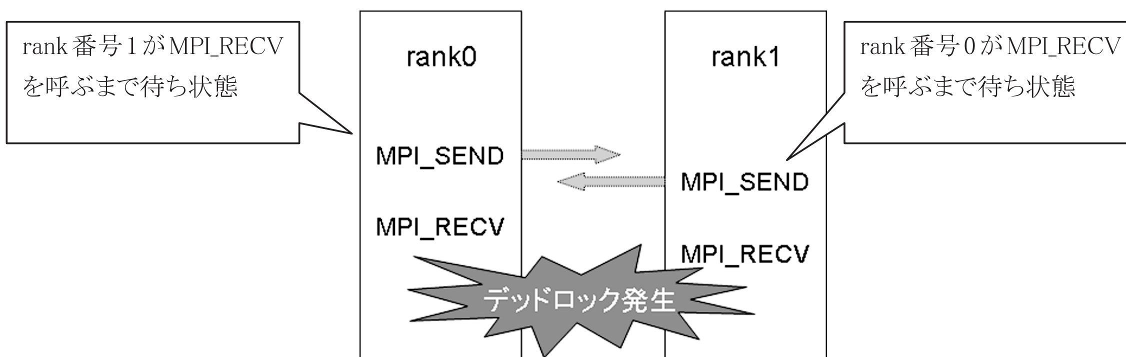

図 6.5-2　デッドロック発生のイメージ

```
do i = 0,nprocs-1
  if(myrank==i)then
    k=mod(i+1,nprocs)
    itag=k
    call MPI_ISEND(b(1+i*it),it,MPI_INTEGER,k,itag,MPI_COMM_WORLD,request,ierr)        !①
  endif
enddo

do i = 0,nprocs-1
  if(myrank==i)then
    k=mod(i-1+nprocs,nprocs)
    itag=i
    call MPI_RECV(b(1+i*it),it,MPI_INTEGER,k,itag,MPI_COMM_WORLD,status,ierr)
  endif
enddo

call MPI_WAIT(request,status,ierr)                                                      !②
```

図 6.5-3　正常動作(デッドロックが発生しない)例

図 6.5-3 にデッドロックを発生させずに実行するコードを示す．図 6.5-1 では MPI_SEND を使用していたのに対し，図 6.5-3 では MPI_ISEND を使用している．MPI_ISEND はノンブロッキングモード（注）で通信を行うため，送信バッファが再利用可能になるのを待たずに次の処理へ進むことができる．ただし，MPI_WAIT を指定し，通信が完了したことを待ち合わせる必要がある．図 6.5-4 は図 6.5-3 を 2 並列で実行したときのイメージである．

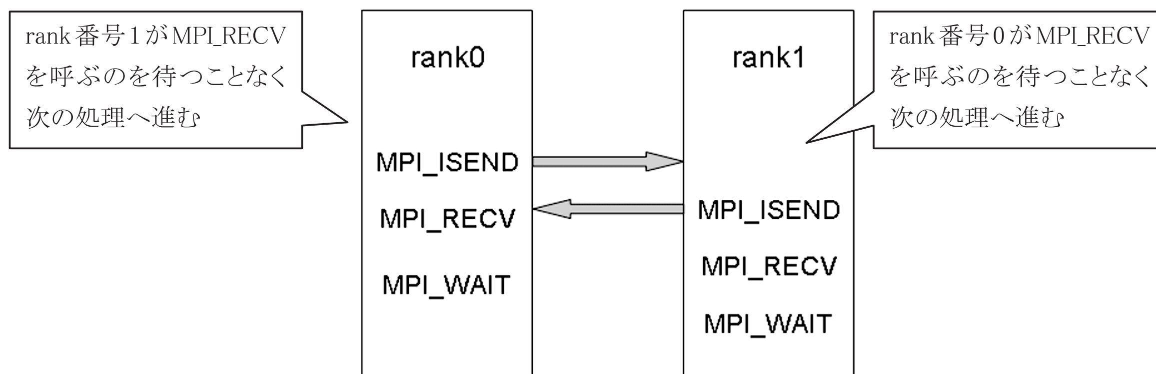

図 6.5-4　正常動作のイメージ

図 6.5-3 で使用した MPI サブルーチンについて以下に示す．

① MPI_ISEND

MPI の通信処理を行うサブルーチンであり，データの送信をノンブロッキング（非同期）モードで行う．このサブルーチンは一対一通信と呼ばれる分類に属し，通信相手が必要になる．受信側にデータを送信するために呼び出される．MPI_WAIT とペアで使用する．第一引数から第六引数は，MPI_SEND と同じである．第七引数（request）は通信識別子を示す．

② MPI_WAIT

ノンブロッキング通信の完了の待ち合わせを行う．一対一ノンブロッキング通信を使用した場合にペアで使用する．第一引数（request）に通信識別子，第二引数（status）にメッセージ情報を示す．

(注)ノンブロッキング(非同期)モード:送信または受信の操作が完了する前に手続きから戻る．ノンブロッキング通信を使用する場合は，送受信操作が完了する前に送信または受信領域の内容を変更してはならない．

6.6 SX-9 の最適化事例

6.6.1 通信方法の見直し

適切な通信サブルーチンを選択することにより，通信時間を短縮することができる．本節では，集団通信である MPI_SCATTERV と MPI_GATHERV を単方向通信である MPI_GET と MPI_PUT に置き換える例を示す．MPI_GET, MPI_PUT は，MPI プロセスが単独で他の MPI プロセスに対してデータの送受信を行うことができる．このため，すべてのプロセスが同時に通信を行う集団通信 MPI_SCATTERV, MPI_GATHERV に比べて実行時間を短縮することができる．

図 6.6-1 は rank 番号 0 のプロセスが持つデータを他の rank 番号のプロセスへ送り，処理を行った後にそれらのデータを rank 番号 0 のプロセスへ送り返すコードである．図 6.6-1 の通信のイメージを図 6.6-2 に示す．

```
  if(myrank.eq.0) then
    call MPI_SCATTERV(data,iscnt,idisp,MPI_REAL8,                      !①
&                  MPI_IN_PLACE,idummy,idummy,0,
&                  MPI_COMM_WORLD,ierr)
  else
    im=nz/nprocs*myrank+1
    call MPI_SCATTERV(data,iscnt,idisp,MPI_REAL8,                      !①
&                  data(1,1,im),isize,MPI_REAL8,0,
&                  MPI_COMM_WORLD,ierr)
  endif
   :
  if(myrank.eq.0) then
    call MPI_GATHERV(MPI_IN_PLACE,idummy,idummy,                       !②
&                  data,ircnt,idisp,MPI_REAL8,0,
&                  MPI_COMM_WORLD,ierr)
  else
    call MPI_GATHERV(data(1,1,im),iscnt,MPI_REAL8,                     !②
&                  data,ircnt,idisp,MPI_REAL8,0,
&                  MPI_COMM_WORLD,ierr)
  Endif
```

図 6.6-1　MPI_SCATTERV と MPI_GATHERV を用いた通信コード

この例で使用した MPI サブルーチンについて以下に示す．

① MPI_SCATTERV

MPI の集団通信を行うサブルーチンであり，同一コミュニケータ内の 1 つのプロセスがデータを送信し，全プロセスが受信を行う．第一引数は送信データの先頭アドレス，第二引数は送信するデータ要素数（プロセス毎），第三引数は送信データの先頭後レスからの変位（プロセス毎），第四引数（MPI_REAL8）は送信するデータの型，第四引数は受信データの先頭アドレス，第五引数(MPI_REAL8)は受信するデータの要素数，第六引数は受信するデータの型，第七引数(0)は送信元プロセスの rank 番号を示す．

② MPI_GATHERV

MPI の集団通信を行うサブルーチンのであり，同一コミュニケータ内のすべてのプロセスがデータを送信し，1 つのプロセスがそれらのデータの受信を行う．第一引数は送信データの先頭アドレス，第二引

数は送信するデータの要素数，第三引数(MPI_REAL8)は送信するデータの型，第四引数は受信データの先頭アドレス，第五引数は受信するデータの要素数(プロセス毎)，第六引数は受信データの先頭アドレスからの変位(プロセス毎)，第七引数(MPI_REAL8)は受信するデータの型，第七引数(0)は送信先プロセスの rank 番号を示す.

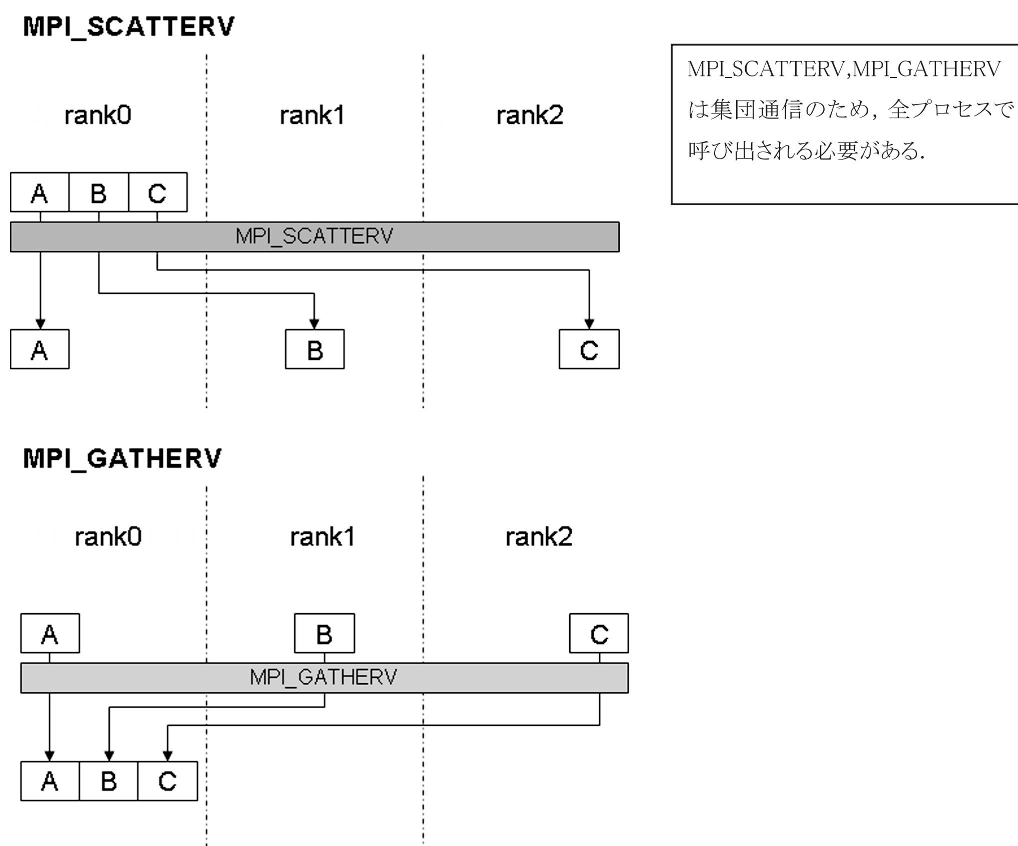

図 6.6-2　MPI_SCATTERV と MPI_GATHERV を用いた通信

次に MPI_SCATTERV，MPI_GATHERV を MPI_GET，MPI_PUT に置き換える. MPI_GET，MPI_PUT を用いた通信イメージを図 6.6-3 に，コードを図 6.6-4 に示す.

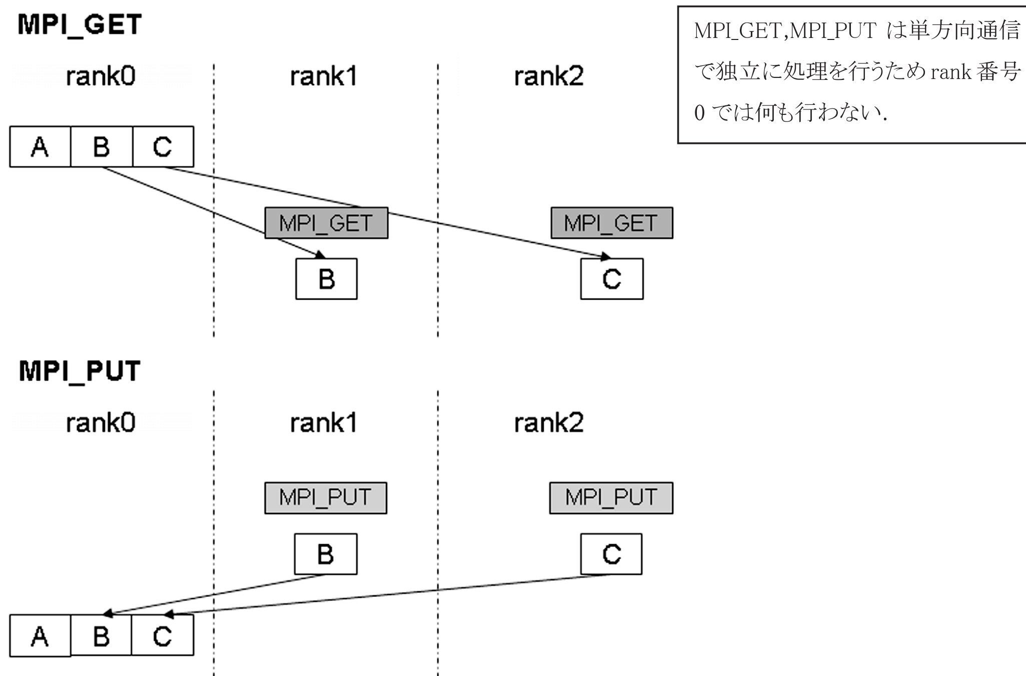

図 6.6-3　MPI_GET と MPI_PUT を用いた通信

```
    integer(kind=MPI_ADDRESS_KIND) idisp                        !①

    isize=8*nx*ny*nz
    call MPI_WIN_CREATE(data,isize,8,MPI_INFO_NULL,              !②
   &                    MPI_COMM_WORLD,win,ierr)
    call MPI_WIN_FENCE(0,win,ierr)                               !③

    idisp=nx*ny*nz/nprocs*myrank
    call MPI_WIN_FENCE(0,win,ierr)                               !③
    if(myrank.ne.0) then
      call MPI_GET(data(1,1,nz/nprocs*myrank+1),nx*ny*nz/nprocs, !④
   &               MPI_REAL8,0,idisp,nx*ny*nz/nprocs,
   &               MPI_REAL8,win,ierr)
    endif
    call MPI_WIN_FENCE(0,win,ierr)                               !③
     :
    call MPI_WIN_FENCE(0,win,ierr)                               !③
    if(myrank.ne.0) then
      call MPI_PUT(data(1,1,nz/nprocs*myrank+1),nx*ny*nz/nprocs, !⑤
   &               MPI_REAL8,0,idisp,nx*ny*nz/nprocs,
   &               MPI_REAL8,win,ierr)
    endif
    call MPI_WIN_FENCE(0,win,ierr)                               !③

    call MPI_WIN_FREE(win,ierr)                                  !⑥
```

図 6.6-4　MPI_PUT と MPI_GET を用いた通信コード

この例で使用した MPI の宣言部分とサブルーチンについて以下に示す.

①　integer(kind＝MPI_ADDRESS_KIND)

下記, MPI_GET, MPI_PUT の第五引数で使用する変数は, この型を宣言する.

②　MPI_WIN_CREATE

MPI の単方向通信は，許可したメモリ領域以外，他プロセスからのアクセスを禁止している．この許可したメモリ領域のことをウィンドウと言い，MPI_WIN_CREATE はこのウィンドウの登録を行うサブルーチンである．第一引数はウィンドウに登録するデータの先頭アドレス，第二引数はウィンドウのサイズ（単位は byte），第三引数はウィンドウ内の 1 つのデータのサイズ（単位は byte），第四引数(MPI_INFO_NULL)は info 引数，第六引数は登録されるウィンドウを示す．

③ MPI_WIN_FENCE

単方向通信 MPI の通信処理の同期をとるためのサブルーチンである．ウィンドウをアクセスする可能性のあるすべてのプロセスで呼び出される必要がある．

④ MPI_GET

MPI の単方向通信を行うサブルーチンであり，転送対象プロセスのウィンドウ領域からデータの読込みを行う．第一引数は受信先の先頭アドレス，第二引数は受信するデータ数，第三引数は受信するデータの型，第四引数は送信元の rank 番号，第五引数は登録したウィンドウの先頭からの変位，第六引数は送信するデータ数，第七引数は送信するデータの型を示す．

⑤ MPI_PUT

MPI の単方向通信を行うサブルーチンであり，転送対象プロセスのウィンドウ領域に対してデータの書込みを行う．第一引数は送信するデータの先頭アドレス，第二引数は送信するデータ数，第三引数は送信するデータの型，第四引数は受信先の rank 番号，第五引数は登録したウィンドウの先頭からの変位，第六引数は受信するデータ数，第七引数は受信するデータの型を示す．

⑥ MPI_WIN_FREE

ウィンドウの登録を抹消するためのサブルーチンである．第一引数は登録を抹消するウィンドウを示す．

SX-9 におけるそれぞれの通信時間は図 6.6-5 のとおりである．なお通信に使用する配列 data のサイズは約 8GB，プロセス数は 4 プロセスである．MPI の集団通信から単方向通信へ変更することで 1.27 倍の性能向上が得られる．

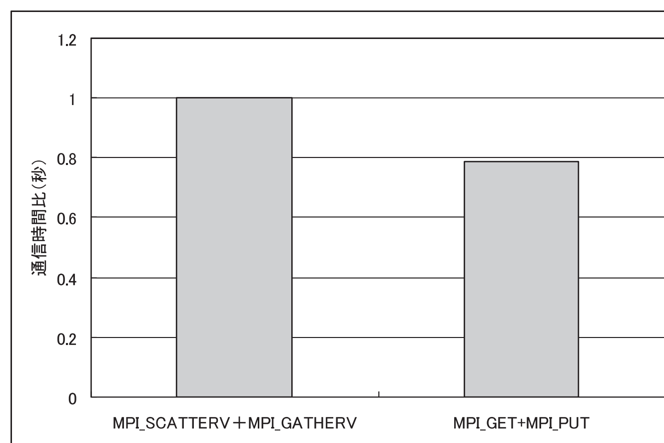

図 6.6-5 通信の違いによる性能比

6.6.2 SX-9 のグローバルメモリ機能

SX-9 では通常のユーザプログラムが使用するメモリ空間はローカルメモリにある．MPI がデータ通信をする際には，図 6.6-6 に示すように，ローカルメモリ上にある転送データをグローバルメモリの通信バッファへコピーし，MPI システム通信バッファ間で転送が行われる．受信側で MPI システム通信バッファからローカルメモリのユーザ領域へコピーが完了した時点でデータ通信が終了する．SX-9 では，グローバルメモリ機能を提供しており，図 6.6-7 に示すようにユーザプログラムが使用するメモリ空間をグローバルメモリ上へ割り付けておくことができる．このことにより，ローカルメモリとグローバルメモリ間のメモリコピーを行うことなく，グローバルメモリ空間にあるユーザ領域のデータを直接転送することができる．

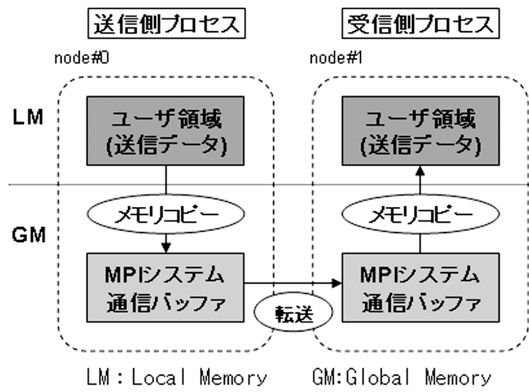

図 6.6-6　通常の MPI データ転送手順

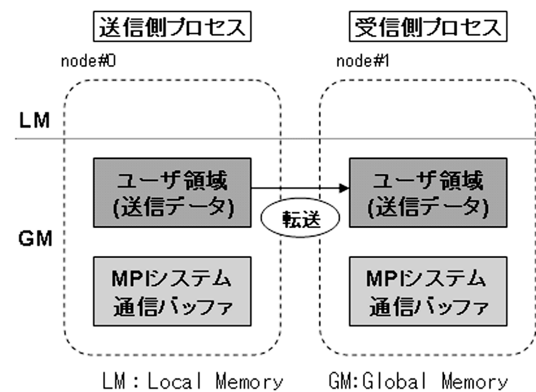

図 6.6-7　グローバルメモリ領域を使用した MPI データ転送手順

グローバルメモリ機能を使用するためには，通信するデータを格納する配列を動的(allocatable 属性)に確保し，コンパイルオプション「-gmalloc」を指定する．ただし，配列を通信するごとに動的に確保するような場合，配列の割当/解放の処理がオーバヘッドとなりグローバルメモリを用いた効果を打ち消す場合がある．このためグローバルメモリの動的な割当/解放は最小限に止めることが必要である． 表 6.6-1 に，SX-9 においてグローバルメモリ機能を使用した場合としない場合のデータ転送時間を示す．これは，約 100MB のデータを MPI_ALLGATHER でデータ転送した例である．グローバルメモリ機能を使用することで，2 ノード(32 プロセス)使用時に 2.77 倍，4 ノード(64 プロセス)使用時に 2.67 倍の性能向上が得られることがわかる．

表 6.6-1　グローバルメモリによる実行時間の比較

	2 ノード(32 プロセス)	4 ノード(64 プロセス)
グローバルメモリ機能なし	162.031sec	327.985sec
グローバルメモリ機能あり	58.441sec	122.542sec

6.7 SX-9 における MPI 性能情報採取方法

本節では SX-9 における MPI 性能情報の採取方法について説明する.

(1) MPIPROGINF

MPI/SX では各 MPI プロセスの性能情報を採取する機能がある. この機能によって MPI プロセス毎, または全 MPI プロセスの情報を集計編集して表示させることができる. 性能情報の表示は, MPI_COMM_WORLD(MPI_UNIVERSE=0)の rank 番号 0 における MPI プロセスの標準エラー出力に対して行われる. 使用するにはプログラム実行時に環境変数 MPIPROGINF を指定する. MPIPROGINF の値とその動作は以下, 表 6.7-1 のとおりである.

表 6.7-1 MPIPROGINF の値

NO	性能情報を出力しない(既定値).
YES	基本情報を集約形式で出力する.
DETAIL	詳細情報を集約形式で出力する.
ALL	基本情報を拡張形式で出力する.
ALL_DETAIL	詳細情報を拡張形式で出力する.

```
MPI Program Information:
========================
Note: It is measured from MPI_Init till MPI_Finalize.
      [U,R] specifies the Universe and the Process Rank in the Universe.

                                          (a)                  (b)              (c)
Global Data of 16 processes :          Min [U,R]            Max [U,R]        Average
============================

Real   Time (sec)           :        1.350 [0,15]         1.577 [0,0]          1.464
User   Time (sec)           :        1.267 [0,13]         1.324 [0,8]          1.308
System Time (sec)           :        0.008 [0,2]          0.012 [0,13]         0.008
Vector Time (sec)           :        0.735 [0,0]          0.766 [0,8]          0.752
Instruction Count           :    192390742 [0,0]      200269828 [0,1]      197702659
Vector Instruction Count    :      4788036 [0,0]        5041807 [0,1]        4979275
Vector Element Count        :     75304093 [0,8]       78786404 [0,1]       77034023
FLOP Count                  :       200402 [0,2]         200411 [0,0]         200403
MOPS                        :      199.609 [0,0]        210.707 [0,15]       206.323
MFLOPS                      :        0.151 [0,8]          0.158 [0,13]         0.153
Average Vector Length       :       14.988 [0,8]         15.854 [0,15]        15.472
Vector Operation Ratio (%)  :       28.193 [0,8]         28.841 [0,15]        28.555
Memory size used (MB)       :     1095.745 [0,0]       1095.745 [0,0]       1095.745
Global Memory size used (MB):       64.000 [0,0]         64.000 [0,0]         64.000
MIPS                        :      146.014 [0,0]        153.770 [0,15]       151.210
Instruction Cache miss (sec):        0.080 [0,3]          0.093 [0,0]          0.085
Operand     Cache miss (sec):        0.012 [0,8]          0.030 [0,0]          0.013
Bank Conflict Time
  CPU Port Conf.       (sec):        0.001 [0,7]          0.001 [0,13]         0.001
  Memory Net. Conf.    (sec):        0.578 [0,0]          0.608 [0,8]          0.595
```

図 6.7-1 MPIPROGINF の出力例(DETAIL 指定時)

表示される項目の意味は以下のとおりである. その他の項目はプログラム実行解析情報(PROGINF)と同じとなる.

(a):すべての MPI プロセスを対象に, 基本情報または詳細情報について集計した最小値と rank 番号

（b）：すべての MPI プロセスを対象に，基本情報または詳細情報について集計した最大値と rank 番号

（c）：すべての MPI プロセスを対象に，基本情報または詳細情報について集計した平均値

MPIPROGINF の各項目で最大値と最小値の差が大きい場合，処理のインバランス（注）が発生している可能性がある．領域が均等に分割されていてもプロセスごとの処理量が異なるとインバランスが発生する．領域の分割方法の変更などの検討が必要である．

（注）インバランス：各プロセスに割り当てる処理が不均衡な状態を言う．インバランスの場合，プロセス間で同期を取るタイミングで，処理の長いプロセスを待ち合わせする必要があり，先に処理を終えたプロセスに待ち時間が発生する．

(2) MPICOMMINF

MPI/SX では各 MPI プロセスの MPI 通信情報を採取する機能がある．この機能によって全 MPI 手続き実行所要時間，MPI 通信待ち合わせ時間，送受信データ総量，および主要 MPI 手続き呼び出し回数を表示することができる．表示は，MPI_COMM_WORLD （ MPI_UNIVERSE=0 ）の rank 番号 0 における MPI プロセスの標準エラー出力に対して行われる．使用するにはプログラム実行時に環境変数 MPICOMMINF を指定する．MPICOMMINF の値とその動作は以下，表 6.7-2 のとおりである．

表 6.7-2　MPICOMMINF の値

NO	通信情報を出力しない(既定値).
YES	最小値，最大値，および平均値を表示する.
ALL	最小値，最大値，平均値，および各プロセス毎の値を表示する.

```
MPI Communication Information:
------------------------------

Real MPI Idle Time (sec)    :       0.092 [0,8]         0.098 [0,1]         0.095   (a)
User MPI Idle Time (sec)    :       0.089 [0,13]        0.098 [0,1]         0.095   (b)
Total real MPI Time (sec)   :       1.302 [0,15]        1.532 [0,0]         1.419   (c)
Send        count           :           0 [0,0]             0 [0,0]             0 ┐
Recv        count           :           0 [0,0]             0 [0,0]             0 │
Barrier     count           :           0 [0,0]             0 [0,0]             0 │
Bcast       count           :           0 [0,0]             0 [0,0]             0 │
Reduce      count           :           0 [0,0]             0 [0,0]             0 │
Allreduce   count           :           0 [0,0]             0 [0,0]             0 │
Scan        count           :           0 [0,0]             0 [0,0]             0 │
Exscan      count           :           0 [0,0]             0 [0,0]             0 │
Redscat     count           :           0 [0,0]             0 [0,0]             0 │
Gather      count           :           0 [0,0]             0 [0,0]             0 │
Gatherv     count           :           0 [0,0]             0 [0,0]             0 │
Allgather   count           :      100000 [0,0]        100000 [0,0]        100000 │
Allgatherv count            :           0 [0,0]             0 [0,0]             0 ├(d)
Scatter     count           :           0 [0,0]             0 [0,0]             0 │
Scatterv    count           :           0 [0,0]             0 [0,0]             0 │
Alltoall    count           :           0 [0,0]             0 [0,0]             0 │
Alltoallv   count           :           0 [0,0]             0 [0,0]             0 │
Alltoallw   count           :           0 [0,0]             0 [0,0]             0 │
Number of bytes sent        :           0 [0,0]             0 [0,0]             0 │
Number of bytes recv        :           0 [0,0]             0 [0,0]             0 │
Put         count           :           0 [0,0]             0 [0,0]             0 │
Get         count           :           0 [0,0]             0 [0,0]             0 │
Accumulate count            :           0 [0,0]             0 [0,0]             0 │
Number of bytes put         :           0 [0,0]             0 [0,0]             0 │
Number of bytes got         :           0 [0,0]             0 [0,0]             0 │
Number of bytes accum       :           0 [0,0]             0 [0,0]             0 ┘
```

図 6.7-2　MPICOMMINF の出力例（YES 指定時）

表示される各項目の意味は以下のとおりである．

(a)：プログラム実行に要した実 MPI 通信待ち合わせ時間

(b)：プログラム実行に要した CPU 時間の内，ユーザルーチン実行に要した MPI 通信待ち合わせ時間

(c)：全 MPI サブルーチン実行時間

(d)：主要 MPI サブルーチン呼び出し回数

ただし，本機能を利用する場合には，プロファイル版 MPI ライブラリをリンクする必要がある．プロファイル版 MPI ライブラリは，MPI プログラムのコンパイル/リンク用シェルスクリプト（mpif90）の-mpitrace，-mpiprof，-ftrace いずれかのオプション指定によりリンクされる．

主要 MPI サブルーチン呼び出し回数と次節 6.7(3)の FTRACE とを合わせて，通信性能を分析することができる．詳しくは 6.7(3)を参照のこと．

(3)　FTRACE

4.3.1(4)にて，逐次処理プログラムにおける FTRACE について説明した．MPI プログラムで本機能を使用すると，逐次処理プログラムと同様の情報に加え，手続きごと，あるいはユーザ指定リージョンの MPI 通信性能情報を採取することができる．使用するには，測定対象のソースプログラムをオプション -ftrace を指定してコンパイルする．環境変数 F_FTRACE の値とその動作は以下，表 6.7-3 のとおりである．

表 6.7-3 F_FTRACE の値

YES	プログラム終了直後に解析リストを標準エラー出力ファイルに形式2で出力することを指定する(FMT2 と同じ).
FMT0	プログラム終了直後に解析リストを標準エラー出力ファイルに形式0（注1）で出力することを指定する.
FMT1	プログラム終了直後に解析リストを標準エラー出力ファイルに形式1（注2）で出力することを指定する.
FMT2	プログラム終了直後に解析リストを標準エラー出力ファイルに形式2（注3）で出力することを指定する.
NO	プログラム終了直後に解析リストを標準エラー出力ファイルに出力しないことを指定する(既定値).

```
PROC.NAME  FREQUENCY  EXCLUSIVE        AVER.TIME     MOPS   MFLOPS V.OP  AVER.     VECTOR I-CACHE O-CACHE    BANK CONFLICT
                      TIME[sec]( % )    [msec]                     RATIO V.LEN      TIME    MISS    MISS CPU PORT  NETWORK

test               1     1.320(100.0)  1320.361    199.8      0.2 28.75  15.8     0.735   0.094   0.031    0.001    0.578
--------------------------------------------------------------------------------------------------------------------------
total              1     1.320(100.0)  1320.361    199.8      0.2 28.75  15.8     0.735   0.094   0.031    0.001    0.578

               (a)           (b)        (c)         (d)        (e)       (f)         (g)       (h)

PROC.NAME      ELAPSED       COMM.TIME  COMM.TIME   IDLE TIME  IDLE TIME  AVER.LEN      COUNT  TOTAL LEN
               TIME[sec]         [sec]  / ELAPSED       [sec]  / ELAPSED    [byte]                [byte]

test               1.615         1.572      0.973       0.093      0.057      68.0     100000       6.5M
-------------------------------------------------------------------------------------------------------
```

図 6.7-3 FTRACE の出力例（YES 指定時）

表示される各項目の意味は以下のとおりである.
(a)～(h)以外の項目は逐次処理プログラムの FTRACE と同じである.

(ａ):経過時間
(ｂ):MPI 通信の送受信に要した経過時間（MPI 手続きの実行に要した時間を含む）
(ｃ):関数の MPI 通信の送受信に要した経過時間（MPI 手続きの実行に要した時間を含む）と, 経過時間に対する比率
(ｄ):MPI 通信を行うまでの待ち時間, および同期待ちに要した経過時間
(ｅ):関数の MPI 通信を行うまでの待ち時間, および同期待ちに要した経過時間と, 経過時間に対する比率
(ｆ):MPI 通信 1 回当たりの平均通信量
(ｇ):MPI 通信回数
(ｈ):MPI 通信の通信量

FTRACE を用いることで, 手続き（サブルーチンや関数）単位の通信性能を知ることができる. しかしながら FTRACE では, どの MPI サブルーチンが何回呼び出されたかを知ることができない. 6.7(2)節の MPICOMMINF で得た情報と組み合わせることで, 使用された MPI サブルーチンと通信性能(通信時間, 通信量など)を分析することが可能になる. また, FTRACE の region 機能を使用することで, 特定の MPI ライブラリの通信性能情報を得ることができる. この場合, MPI_BARRIER（注 4）で同期を取ることで, 演算処理の待ち時間を含まない通信に要する時間のみを採取することができる.

（注　1）形式 0. プログラム単位名を左端に表示する.

（注　2）形式 1. プログラム単位名を右端に表示する.

（注　3）形式 2. プログラム単位名を左端に表示する. ただし, プログラム単位名の長さが 10 文字を超えるものについては, 名前の表示の直後で改行して表示する.

（注　4）MPI_BARRIER：コミュニケータ内の全プロセスで同期を取る場合に使用する. 全プロセスで MPI_BARRIER が呼ばれるまで, どのプロセスも次の処理へ進まない.