第 2 回 並列ゼミ

ゼミ担当者　：　真武信和，山本啓二，坂田大輔
指導院生　　：　下坂久司，輪湖純也，冨岡弘志
開催日　　：　2003 年 4 月 30 日

ゼミ内容: 本ゼミでは，分散メモリ環境における並列プログラミングである MPI について学ぶ．まず，並列処理のプロセッサ間通信方式であるメッセージパッシング方式に触れ，MPI の実装ライブラリである MPICH を用いた実際のプログラミング方法を説明する．

1　はじめに

このゼミでは，MPI を用いた並列プログラミングの方法について説明する．MPI(Message Passing Interface) とは，分散メモリ環境における並列プログラミングの標準的な規格である．その名の通りメッセージパッシング方式に基づいた仕様であり，MPI の仕様に準じた実装ライブラリは，複数存在している．その中の幾つかはフリーで配布されており，UNIX 系を中心として Windows，Machintosh とほぼ全ての OS，アーキテクチャに対応しているため，どのような分散メモリ環境においても MPI をフリーで使用することができる．本ゼミでは，MPI の実装ライブラリとして MPICH を用いる．

2　メッセージパッシング方式

メッセージパッシング方式とは，プロセス間でメッセージを交信しながら並列処理を実現する方式である．並列処理では，複数のプロセスが通信を行いながら同時に処理を進めていく．そこで問題になるのがプロセス間の通信であるが，メッセージパッシング方式ではプロセス間での通信を互いのデータの送受信にて行う．

メッセージパッシングの方式には 1 対 1 通信とグループ通信の 2 種類が存在する．また，それぞれについて，ブロッキング通信とノンブロッキング通信という方式が存在する．

2.1　1 対 1 通信とグループ通信

■ 1 対 1 通信

1 対 1 通信とは，メッセージパッシングにおける最も基本的な通信機能であり，一つのプロセスが送信元，相手のもう一つのプロセスが受信先になって行われる．Fig. 1 の例で説明すると，プロセス A がプロセス B にメッセージを送っていることになる．この場合，ほかのプロセス C〜F にはメッセージは送れない．

■グループ通信

グループ通信とは，プロセスがグループ内での集団通信動作，例えば，一台のマシンが他のマシン全体に対してデータを送信することである．Fig. 2 の例では，プロセス A がグループ中のそのほかのプロセス，プロセス B〜F 全てにメッセージを送ることになる．

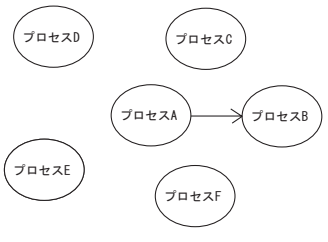

Fig. 1　1 対 1 通信の形態

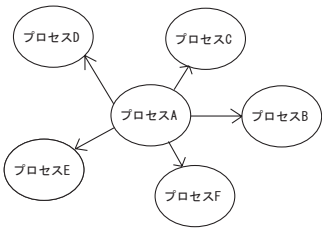

Fig. 2　グループ通信の形態

2.2 ブロッキング通信とノンブロッキング通信

■ブロッキング通信

ブロッキング通信では，送受信の各プロセスは，通信が完了するまでそのプロセス内の他の作業を実行することが出来ない．この場合，他の作業は通信が終了するまで待たされることになり効率が悪くなる場合がある．具体的には，Fig. 3 に示すように送信，受信が宣言されてから完了するまで処理は待機状態になるため非常に無駄が多くなってしまう．ただし，各操作の完結が保証されているためノンブロッキング通信に比べ処理が簡単になる．

■ノンブロッキング通信

ノンブロッキング通信では，送受信の各プロセスは，通信が完了しなくても，プロセス内のほかの作業を実行することができる．この場合，通信が終了するのを待つ必要が無いため，より効率の良いプログラムを作成することができる． 具体的には，Fig. 4 に示すように，送信，受信が宣言されてからも処理を継続することができるためブロッキング通信に比べ通信待ちの時間が少なくなり処理時間の軽減を図ることができる．特に，非同期通信などを行う場合にはノンブロッキング通信は必要不可欠である．しかし，操作の完了が保証されていないため，ノンブロッキング通信を完了するための手続きを呼び出す必要がある．

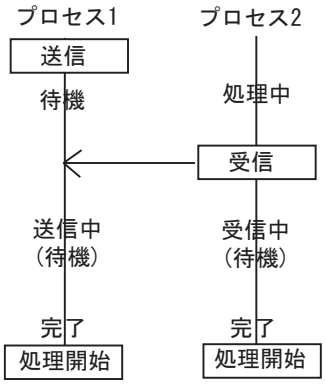

Fig. 3　ブロッキング通信

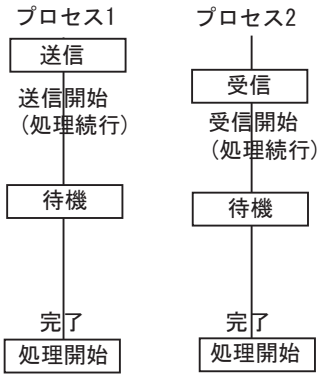

Fig. 4　ノンブロッキング通信

メッセージパッシング方式は，分散メモリ環境において現在最も主流の方式であり，かなり本格的な並列プログラミングを実現することができる．この方式をプログラムで実現するためのライブラリ[1]が， メッセージパッシングライブラリである． その最も代表的なライブラリとして MPI の仕様に基づいた MPICH がある．

3 MPI について

MPI とは，ある特定のライブラリを指すのではなく，メッセージ通信を行うためのインターフェースの仕様のことである．この仕様は MPI Forum によって定められている．MPI はあくまで仕様であるので,「`MPI_Send` 関数はデータを送信するための関数で，引数はこれ」ということが決められているだけであり，実際にその関数をどういう方法で実現するかは実装者に委ねられる．この MPI 規約を用いたプログラミングを行い，実行する計算機に MPI ライブラリ（MPICH など）が実装されていれば，プログラムのソースコードを修正することなく，プログラムをコンパイルするだけで移植が完了できるという利点がある．

3.1 MPI の存在意義

分散メモリ型の場合，メッセージを送るためにはネットワークを用いた通信が必要となる．もし，MPI のようなメッセージパッシングをサポートする API がない場合，プログラマはすべての通信手段を書かなければならず，TCP/IP を用いるか，そのほかのプロトコルを用いるかによってプログラムを変更する必要がある．しかし，MPI という共通のインターフェイスを用いることで，プログラマは通信といった低レベルの処理を気にすることなく，並列プログラムの作成に専念することができる．

3.2 MPI の目標

MPI は次に示すようなことを目標としている．

[1]Library：ある特定の機能を持ったプログラムを他のプログラムから利用できるように部品化し，複数のプログラム部品を一つのファイルにまとめたもの．ライブラリ自体は単独で実行することはできず，他のプログラムの一部として動作する．(e-Words より)

- ネットワーク上のプロセス間の効率のよい通信を可能とする
- 信頼できる通信インターフェースを提供する
- 異なるプラットフォーム上への移植のしやすさを保証する
- PVM などの既存のものと同等の使いやすさとより高い融通性を実現する

3.3 MPI の実装ライブラリ

MPI の実装ライブラリはフリー，ベンダを問わずに数多く存在する．その中でも代表的なフリーのライブラリとして，MPICH と LAM が挙げられる．

■ MPICH

MPICH とは，アメリカのアーゴン国立研究所が模範実装として開発し，無償でソースコードを配布したライブラリである．移植しやすさを重視した作りになっているため，盛んに移植が行われ，世界中のほとんどのベンダの並列マシン上で利用することができる．また，通信ドライバを切り替えることによって，プログラムのソースコードの変更をすることなく，分散メモリ環境だけでなく，共有メモリ環境のマシンでも動作させることが可能となる．

■ LAM

ノートルダム大学の科学コンピュータ研究室が作成したフリーの MPI ライブラリ．ライブラリだけでなく，いくつかのデバッキングとモニタリングツールも提供している． MPICH と違い，デーモンを介して通信を行うので，MPICH に比べて通信が高速になる．

4 MPI Programming

ここからは，MPI を用いたプログラムを書く際に必要な処理について説明していく．

4.1 プログラムの枠組み

MPI プログラムのための最低限の制御関数を Table 1 に示す．

Table 1　最低限の制御関数

MPI_Init()	MPI ライブラリを利用するための初期化
MPI_Comm_size()	コミュニケータ内のプロセス数を取得
MPI_Comm_rank()	コミュニケータ内の各プロセスが自分の rank を取得
MPI_Finalize()	MPI ライブラリの利用の終了処理

コミュニケータ ： お互いに通信を行うプロセスの集合である．ほとんどの MPI ルーチンは引数としてコミュニケータを取る．変数 MPI_COMM_WORLD は，あるアプリケーションをいっしょに実行している全プロセスからなるグループを表しており，これは最初から用意されている．また新しいコミュニケータを作成することも可能である．

rank ： コミュニケータ内のすべてのプロセスは，プロセスが初期化されたときにシステムによって示された ID を持っている．これは，0 から始まる連続した整数が割り当てられる．プログラマはこれを用いて，処理の分岐，あるいはメッセージの送信元や受信先を指定することができる．

さらに，MPI を用いたプログラムの枠組みを次の Fig. 5 に示す．

4.2 プロセスの認識

通信を行うには，送信元と送信先を明確に識別できなければならない．送信元や送信先はプロセスであるから，各プロセスが一意に識別できる必要がある．MPI では，プロセスを「ランク」によって識別する．ランクとは順位の意味で，ここでは，あるアプリケーションを一緒に実行している複数のプロセスの中での，各プロセスの順位のことである．一般には，n 個のプロセスからなるプロセスグループがあるとき，そのグループに所属するプロセスは，0，1，2，…… n-1 のいずれかの値をランクとして持つ．

```
/* (1) ヘッダファイルの読み込み */
#include "mpi. h"

int main(int argc,  char **argv)
{
    int numprocs,  myid;

    /* (2) MPI ライブラリを利用するための準備（初期化）*/
    MPI_Init(&argc, &argv);

    /* (3) コミュニケータ内のプロセスの数を取得.
           この関数により numprocs にはプロセス数が入力される. */
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

    /* (4) コミュニケータ内の各プロセスが自分の rank を取得.
           この関数により myid には自分の rank 番号が入力される. */
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    /* (5) 並列処理の記述*/

    /* (6) MPI ライブラリの利用の終了処理 */
    MPI_Finalize();
    return 0;
}
```

Fig. 5　MPI プログラムの枠組み

4.3　メッセージ・パッシングの実現

分散メモリ環境では，各プロセッサが自身の管理するメモリに対して排他的に管理している. つまり，他のクラスタ上のデータ，つまりメモリを参照するためには，**プログラマが明示的にデータを送ったり，その送られたデータを受け止める処理が必要になる**．大変な作業と思うかもしれないが，データの同期の必要がないため，タイミングに依存する嫌なバグが発生することが少ない.

一つ具体的な例を挙げてみる. rank0 から送信された数を rank1 が受信・計算して，rank0 に計算後のデータを送信する一連の流れを示す.

Table 2　メッセージ・パッシングの例

rank0	rank1
処理してほしい Data を rank1 へ**送信**	rank0 から Data を**受信**
	受信データに対して計算
rank1 からデータ**受信**	rank0 に対して得られたデータを**送信**
受信したデータを表示する	

Table 2 の例からもわかるように，**受信と送信**は，常にセットで実行される. この例は一対一通信になっているが，グループ通信であってもこの枠組みが崩れることはない．つまり，MPI における通信はどちらかが一方的に行うのではなく，送信・受信の両方が通信の準備ができた状態で行われる. プログラマが注意する点は，**送信側，受信側の両方が準備できた状態を効率よく作りださなければならない，つまりスケジューリングの必要が生まれる.**

4.4　MPI プログラムによる並列化の原理

MPI には，あるアプリケーションを一緒に実行している全プロセスからなるグループを表す変数 MPI_COMM_WORLD が最初から用意されている．また，プロセスは自分のランクを関数 MPI_Comm_rank で知ることができる．プロセスが所属しているグループは 1 つとは限らないから，グループを決めないとランクは決まらない． 自分のなす仕事を判別するには， MPI_Comm_rank 関数で得た自己のランクから， プログラム中で,「if(rank == 0){ /処理/ }」のように仕事をプログラム中で割り振られた仕事を判別する． つまり，Fig. 6 のようになる.

・自己の担当部分の認識はランクを使う

```
if(myrank==0){
    ランク0に位置付けされたものへの処理の記述
    }
if(myrank==1){
    ランク1に位置付けされたものへの処理の記述
    }
```

Fig. 6　プロセッサへの仕事の割り振り

4.5　MPI の通信関数

MPI には様々な通信パターンに対応する関数が 127 個用意されている．それらは 1 対 1 通信の組み合わせでも実現できるが，専用の関数を使えば，プログラムの意図がわかりやすくなり，また，通常，通信効率も良くなる．

データの送受信には，大きく分けて三つの事項を指定する必要がある．

1. メッセージバッファ：送信の際には送るべきデータの所在を示し，受信の際には受け取ったデータをどこにおけばよいかを示す．バッファの先頭アドレス，データの個数，データの型の三つの値で指定する．
2. 通信相手：通信相手にはプロセスを指定できれば良いから，プロセスグループとランクの対を指定する．
3. 通信コンテクスト：通信で転送されるデータが何に関するものなのかを示すもので，指定するには二つの方法がある．
   (a) 整数値の「tag」でプログラマが値を指定する．
   (b) communicator で指定する．

基本的な関数については，添付資料に示す．これらの関数に関しては，後述する．

5　MPICH の使用法

5.1　実行環境

5.1.1　実行ファイルの共有

コンパイルは，通常 SSH（Secure SHell）で接続した相手であるマスタで行う．すると，実行ファイル自体はマスタの中に作成される．しかし，MPI において並列処理を実現する際には，実行する全マシンに実行ファイルが存在している必要がある．そのために実行ファイルをスレーブへ転送する必要が生じる．毎回 FTP などでファイルの転送をしていたのでは面倒である．そこで，Sun が開発した NFS（Network File System）というファイル共有システムを利用する．これによって以下の Fig. 7 のようにファイルの共有が実現され，スレーブの/home にマスタの実行ファイルが存在していることになる．

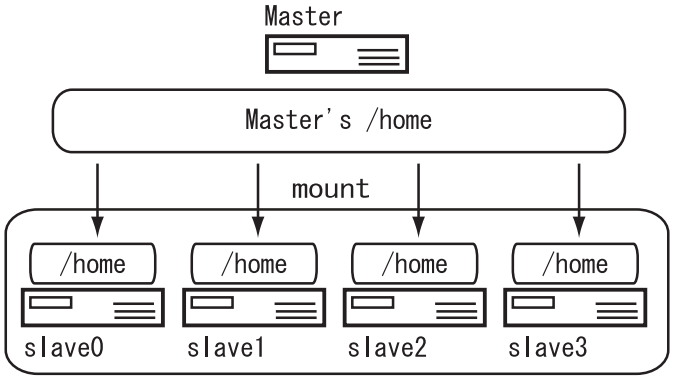

Fig. 7　NFS の仕組み

5.1.2 マスタからスレーブへの実行の依頼

MPICH では，UNIX のシステムで用いられている「rsh」を使用してマスタからスレーブにログインし，一連の MPI プログラムのプロセスを実行している．これらの一連の作業は MPI に用意されているスクリプトである「mpirun」が行ってくれる．走らせるプロセスをいったいどのマシンにリモートログインして実行させるかは，利用者が「machinefile」を作ることで指定することができる．machinefile 内に使用したいマシン名を記述し，実行時に参照することで，指定したマシンによる並列処理が実行される．指定しなかった場合は，管理者が「machines.LINUX」に記述したマシン名の順にリモートプロセスが実行される．マスタからスレーブへ実行を依頼する際の流れを Fig. 10 に示す．

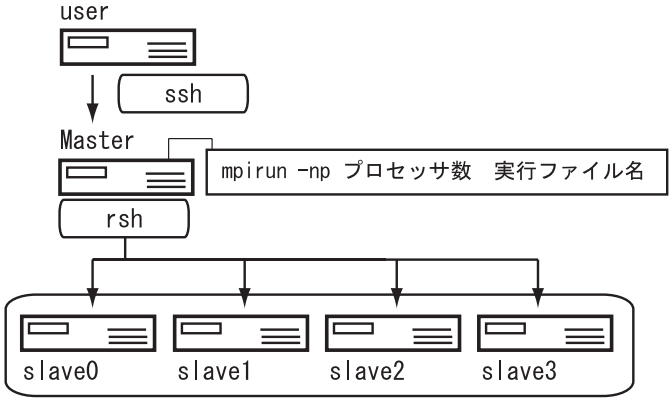

Fig. 8　MPI の実行法

各ノードは並列にそのプログラムを実行するが，MPI 関数によって作られたプロセスにはランクが割り振られ，各プロセスは自己の ID のようなものを認識しており，プログラムに記述された担当部分について実行することで自己のタスクを行う．この ID の割り振りは，MPI の実装，つまり 一連の初期化作業によって半自動的に行われる．

5.2 C 言語を用いたコンパイル法

C 言語で書かれた MPI のプログラムをコンパイルするには “mpicc” というスクリプトを用いる．MPI 自体は単なるライブラリであるので，C コンパイラに並列計算ライブラリである MPI のライブラリを必要に応じてリンクするだけで良い．つまり，毎回の面倒なライブラリのリンクをこのスクリプトが代行してくれるのである．

```
bash$ mpicc samplecpi.c -o samplecpi
```

5.3 MPICH の実行方法

並列処理を行うマシンを指定する場合，マスタにてプログラムのあるディレクトリにおいて以下のコマンドを入力する．

```
bash$ mpirun -np 8 -machinefile machines samplecpi
```

「-np」の後ろにプロセッサ数を指定し，その後，使用するマシン名が記述された machinefile（この場合 machines）を指定する．最後に実行対象をなるプログラム（この場合 samplecpi）を指定することで，並列処理が実行される．この際，マスタから rsh（Remote SHell）を各マシンに発行し，処理をしてもらう仕組みになっている．

また，使用するマシンを指定しない場合は以下のように記述する．

```
bash$ mpirun -np 8 samplecpi
```

6　プログラム例

■文字列を交換するプログラム

マスタはスレーブに hello という文字列を送る．スレーブはマスタから hello を受け取り，マスタに hello を返す．マスタはスレーブから hello という文字列を受け取り，それを表示する．

hello.c　～文字列を交換するプログラム～

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
    int myid,procs,src,dest,tag=1000,count;
    char inmsg[10],outmsg[]="hello";
    MPI_Status stat;

//  MPI ライブラリを利用するための初期化
    MPI_Init(&argc,&argv);
//  コミュニケータ内の各プロセスが自分のランクを取得
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    count=sizeof(outmsg)/sizeof(char);
    if(myid == 0){

//  マスタの処理

        src =1;
        dest =1;
//  "hello"という文字列をランク 1 に送る
        MPI_Send(outmsg,count,MPI_CHAR,dest,tag,MPI_COMM_WORLD);
//  ランク 1 から文字列を受け取る
        MPI_Recv(inmsg,count,MPI_CHAR,src,tag,MPI_COMM_WORLD,&stat);

        printf("%s from rank %d\n",inmsg,src);

    }else{

//  ランク 1 の処理

        src =0;
        dest =0;
//  ランク 0 から文字列を受け取る
        MPI_Recv(inmsg,count,MPI_CHAR,src,tag,MPI_COMM_WORLD,&stat);
//  "hello"という文字列をランク 0 に送る
        MPI_Send(outmsg,count,MPI_CHAR,dest,tag,MPI_COMM_WORLD);

        printf("%s from rank %d\n",inmsg,src);
    }

//   MPI ライブラリの利用終了処理
    MPI_Finalize();
    return 1;

}
```

■積分計算

並列処理による積分計算プログラムを以下に示す．このプログラムでは，各プロセスにそれぞれの積分区間を割り振り，最後にその積分区間ごとの積分値を合計することにより全積分区間の積分値を求める．1 プロセスでの実行（逐次処理）の場合よりも，1 プロセス当たりの積分区間が短くなるため，その分だけ実行速度が上がる．

integral.c　～並列積分計算プログラム～

```
#include <stdio.h>
//   MPI ライブラリを include
#include "mpi.h"
```

```
//  積分区間の終わり（0.0〜N までを積分）
#define N 7.0
#define dx 0.0000001

double f(double x)
{
        return (x*x*x);
}

//   argc：プロセス数
//  *argv[]：machinefile に示された各プロセッサ名
int main(int argc, char *argv[])
{
        int i, rank, pnum;
        double x, j, area, local_F = 0.0, global_F = 0.0;
        MPI_Status stat;
        double data[4];

//  前述
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &pnum);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

//   1 プロセス当たりの積分区間の幅
        area = N / (pnum-1);

//   rank0 のプロセスは実行過程を表示する
        if(rank==0){
                printf("nProcess 0 UPn");
                for(i=1;i<pnum;i++){
                        MPI_Recv(data,4,MPI_DOUBLE,i,123,MPI_COMM_WORLD,&stat);
                        printf("nnProcess %.0lf UPn",data[0]);
                        printf("t%lf~%lftF(x)=%lf",data[1],data[2],data[3]);
                }
        }
//  その他のプロセスは割り振られる積分区間を積分
        else{
                x = area * (rank-1);
                for(j=x;j<area*rank;j+=dx){
                        local_F += f(j)*dx;
                }
                data[0]=(double)rank;
                data[1]=x;
                data[2]=j;
                data[3]=local_F;
                MPI_Send(data,4,MPI_DOUBLE,0,123,MPI_COMM_WORLD);
        }

//  各プロセスが rank0 のプロセスに積分値を渡し，rank0 のプロセスはそれらを足しあわせる
        MPI_Reduce(&local_F, &global_F, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
        if(rank==0){
                printf("nnFunctionntf(X) = x^3nn");
                printf("Areant0~%.1lfnn",N);
                printf("AnswerntF(x) = %lfnn", global_F);
        }

        MPI_Finalize();
        return(0);
}
```

参考文献

1) 渡邊真也：MPI による並列プログラミングの基礎，http://mikilab.doshisha.ac.jp/dia/smpp/cluster2000/

2) 三木光範：並列処理入門，http://mikilab.doshisha.ac.jp/dia/smpp/cluster2000/

7　添付資料～MPI の基本的な関数～

7.1　1 対 1 の通信

■ ブロッキング通信

```
MPI_Send(void *buf,int count,MPI Datatype datatype,int dest,int tag,MPI Comm comm)
```

基本的なブロックキング送信の操作を行う．送信バッファのデータを特定の受信先に送信する．
void *buf：送信バッファの開始アドレス（IN）
int count：データの要素数 (IN)
MPI Datatype datatype：データタイプ (IN)
int dest：受信先 (IN)
int tag：メッセージ・タグ (IN)
MPI Comm comm：コミュニケータ (IN)

```
MPI_Recv(void *buf,int count,MPI Datatype datatype,int source,int tag,MPI Comm comm,MPI
                                    Status *status)
```

要求されたデータを受信バッファから取り出す．またそれが可能になるまで待つ．
void *buf：受信バッファの開始アドレス（受け取ったデータの格納場所）(OUT)
int source：送信元（MPI_ANY_SOURCE で送信元を特定しない）(IN)
int tag：メッセージ・タグ（MPI_ANY_TAG でメッセージ・タグを特定しない）(IN)
MPI Status *status：ステータス (OUT)

発信者を問わずメッセージを受信したい場合には，発信者のランクの変わりに MPI_ANY_SOURCE を指定する．タグを問わずに受信したい場合には MPI_ANY_TAG をタグに指定する．

source：送信元のプロセスのランク
status：受信は通常，特定の発信者が特定のタグで送信したメッセージだけを受信するが，それらを問わずに，とにかく最初にやってきたメッセージを受信したい場合もある．そうした受信を行ったときに，受信されたメッセージの発信者のランクや，送信時に指定されたタグの値を status に書き込む．

```
int MPI_Sendrecv( void *sendbuf, int sendcount, MPI_Datatype sendtype,int dest, int
sendtag,void *recvbuf, int recvcount, MPI_Datatype recvtype,int source, int recvtag,
                        MPI_Comm comm, MPI_Status *status )
```

基本的なブロックキング送信の操作を行う．送信バッファのデータを特定の受信先に送信する．
void *sendbuf：送信バッファの開始アドレス（IN）
int sendcount：送信データの要素数 (IN)
MPI Datatype sendtype：送信データタイプ (IN)
int dest：受信先 (IN)
int sendtag：送信メッセージ・タグ (IN)
void *recvbuf：受信バッファの開始アドレス（受け取ったデータの格納場所）(OUT)
int recvcount：受信データの要素数 (IN)
MPI Datatype recvtype：受信データタイプ (IN)
int source：送信元（MPI ANY SOURCE で送信元を特定しない）(IN)
int recvtag：受信メッセージ・タグ (IN)
MPI Comm comm：コミュニケータ (IN)
MPI Status *status：ステータス (OUT)

この関数は，MPI_Send と MPI_Recv の組み合わせでもできるが，そうするとデッドロックが起こる可能性がある．

■ ノンブロッキング通信

次に，ノンブロッキング通信で用いる関数を紹介する。ノンブロッキング通信では，送受信の手続きの完了を待たずに次の操作が行われるため，送受信の完了を明示しなければならない。これは，MPI_Wait() によって宣言する。

```
int MPI_Isend(void* sendbuf,int sendcount,MPI Datatype sendtype, int dest,MPI Comm comm,MPI
                                    Request *request)
```

基本的なノンブロックキング送信の操作を行う．
MPI request ：通信要求 (ハンドル) を返す．(OUT)
MPI Request : ノンブロッキング通信における送信もしくは受信を要求したメッセージに付けられる識別子．整数である．

```
int MPI_Irecv(void *recvbuf,int recvcount, MPI Datatype recvtype, int source,int recvtag,
                            MPI Comm comm,MPI request *request)
```

基本的なノンブロックキング受信の操作を行う．

```
int MPI_Wait(MPI request *request,MPI Status *status)
```

MPI_Wait は，request によって識別された操作が完了した時点で手続きが終了する．このルーチンによりノンブロッキング通信は完了する．具体的には，ノンブロッキングでの送信，受信呼出によって作成された request が解除され，MPI REQUEST NULL が設定される．また，完了した操作に関する情報は status 内に設定される．

7.2 グループ通信

グループ通信では，通信に参加するすべてのプロセスが，同じ関数を呼び出し，引数として同じコミュニケータを与える．

```
MPI_Bcast(void *buf,int count,MPI Datatype datatype,int root,MPI Comm comm)
```

1 つのプロセスからコミュニケータ内の他の全プロセスにメッセージを一斉に送信する．

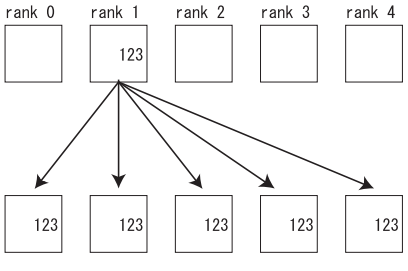

Fig. 9　MPI_Bcast によるデータの移動.

void *buf：送信元では送信バッファの開始アドレス，受信先では受信バッファの開始アドレス
int count：データの要素数
MPI Datatype datatype：データタイプ
int root：送信元の rank

MPI Comm comm：コミュニケータ

```
int MPI_Reduce ( void *sendbuf, void *recvbuf, int count, MPI Datatype datatype, MPI op op, int dest, MPI Comm comm )
```

コミュニケータ内の全プロセスのデータをある１つのプロセスに集める．また同時に各データを足し合わせるなどの演算を行う．

void *sendbuf：送信先（コミュニケータ内の全プロセス）の送信バッファの開始アドレス

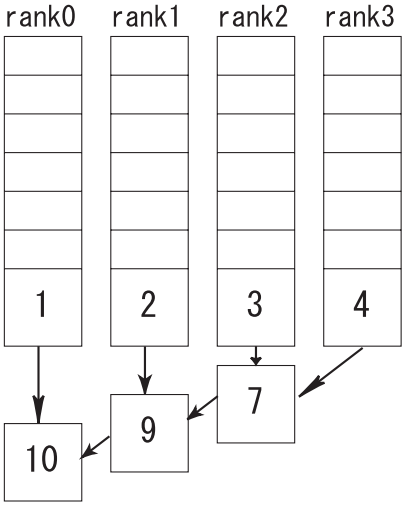

Fig. 10　MPI_Reduce による演算

void *recvbuf：受信先（dest で指定された rank）の受信バッファの開始アドレス
MPI Op op：演算のハンドル

```
MPI_Gather(void *sendbuf,int count,MPI Datatype datatype,void *recvbuf,MPI Datatype datatype,MPI Comm comm)
```

各ノードが持っている値を一つのノードに集める．

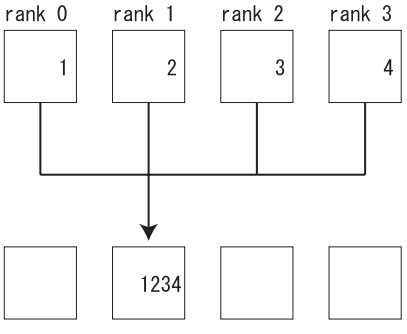

Fig. 11　MPI_Gather による値の収集

```
int MPI_Scatter (void *sendbuf,int sendcount,MPI_Datatype sendtype,void *recvbuf,int recvcount, MPI_Datatype recvtype,int root,MPI_Comm comm)
```

1つのノードが持っているデータを各ノードに配る．ブロードキャストとの違いは，ノードごとに配るデータが異なる点である．

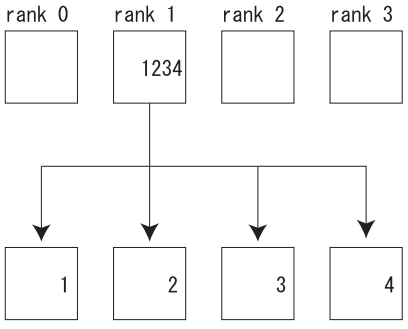

Fig. 12　MPI_Scatter による値の配布

```
int MPI_Allgather (void *sendbuf,int sendcount,MPI_Datatype sendtype,void *recvbuf,int
                   recvcount,MPI_Datatype recvtype,MPI_Comm comm )
```

最初にノードが持っている値を集めたものを全ノードに配る．Gather の直後にルートからブロードキャストを行ったのと同じ結果になる．ただし，Allgather のほうが効率が良い．

```
int MPI_Alltoall(void *sendbuf,int sendcount,MPI_Datatype sendtype,void *recvbuf,int
                 recvcount,MPI_Datatype recvtype,MPI_Comm comm)
```

すべてのノードからすべてのノードに対して，あて先ノードごとに違うデータを配る．各ノードをルートとして Scatter を繰り返したのと同じ結果になる．ただし，Alltoall のほうが効率が良い．

```
int MPI_BARRIER(MPI_Comm comm,int ierror)
```

全プロセス間で同期をとる．

その他の通信関数に関しては以下の URL などを参考にすること．

http://hamic6.ee.ous.ac.jp/software/mpich-1.1.2/