第 2 回 並列ゼミ

ゼミ担当者 ： 輪湖純也，澤田淳二，降幡建太郎
指導院生 ： 児玉憲造，下坂久司
開催日 ： 2002 年 4 月 23 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 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．そのため，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．

2 メッセージパッシングの形態

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お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元になって行われる．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られない．

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．

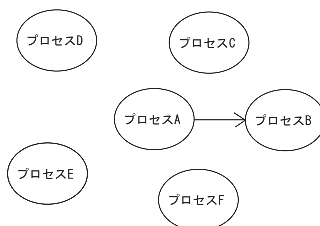


Fig. 1 1 対 1 通信の形態

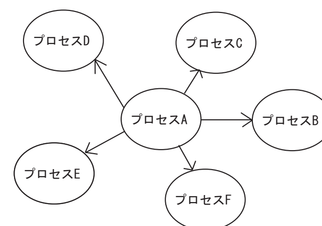


Fig. 2 グループ通信の形態

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 ブロッキング通信とは，操作が完了するまで手続きから戻ることがない場合のことを意味する．この場合，各作業はその手続き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のである．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とは，操作が完了する前に手続きから戻ることがあり得る場合のことを意味する．ノンブロッキング通信を用いることにより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計ることができる．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．

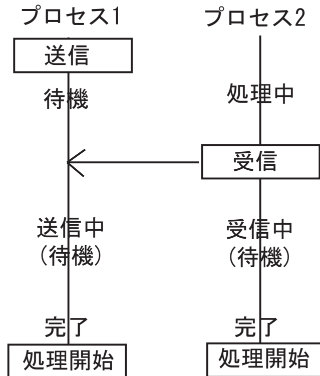


Fig. 3　ブロッキング通信

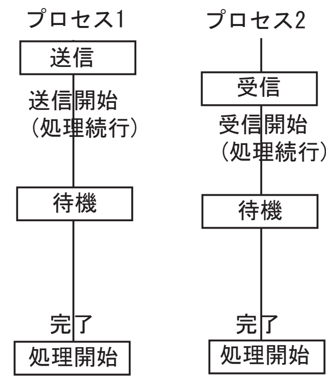


Fig. 4　ノンブロッキング通信

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が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 がある．

3　MPI について

MPI はメッセージパッシングを行うプログラムを記述するために広く使われる「標準」を目指して MPIF(MPI Forum) によって定められたメッセージパッシングのための API(Application Program Interface) 仕様である．

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．しかし，この MPI 規約を用いたプログラミングを行い，実行するための計算機に MPI ライブラリ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　何故，MPI を用いるのか

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，このような問題はあくまで通信の問題であって，並列プログラムを組もうと思っているプログラマにとって，このような部分は気をやむべきことではない．実際何が行われているかが重要なのではなく，どのようにすれば通信ができるかがわかってさえいればよい．つまり，MPI を用いることでプログラマはそのような低レベルの処理を隠蔽し，並列プログラムの作成に専念ことができる．

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. 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
2. 信頼できる通信インターフェースを提供する
3. 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
4. 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

MPI の実装ライブラリとしてはフリー，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は，MPICH があげられる．

3.3.1　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．

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-タ内のプロセス数を取得
MPI_Comm_rank()	コミュニケ-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　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**. 大変な作業と思われるかもしれないが，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・計算してランク 0 に, 計算後のデータを送信する一連の流れを示す．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

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	

```
/* (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

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**.

4.3　MPICH の動作原理

4.3.1　実行ファイルの共有

MPI の管理者でなくとも，その簡単な動作原理は知っておく必要がある. そこでここでは，MPICH の実行に関して的を絞って話をする.

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．その為に実行ファイルをスレーブノードへ転送する必要が生まれる．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，その技術を公開している 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. 6 のようにファイルの共有が実現される．これにより，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．

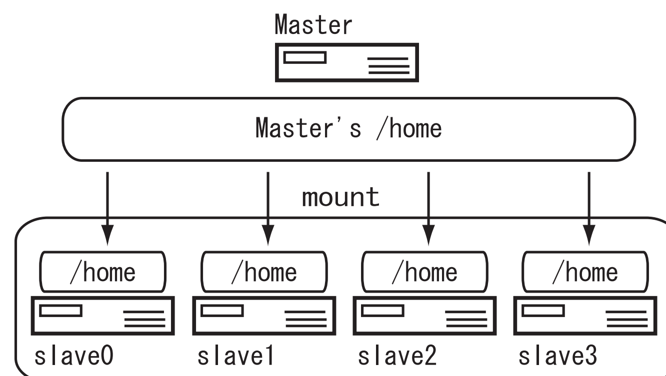


Fig. 6　NFS の仕組み

4.4　マスターからスレーブへの実行の依頼

MPICH では, 通信の枠組みを提供しているだけで特に, 裏でサービスが走っているわけではないことは先述した. そのかわり，UNIX のシステムで用いられている「rsh」を介して裏で同時に作成した一連の 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」が一切の処理を行っている. この際に, 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のかということは，通常は MPI 使用者の問題ではない. これは管理者の問題であり, 単純には「machines.LINUX」に記述されたマシン名の上から順にリモートプロセスが実行される. ここまでの流れを Fig. 7 に示す. 各ノードは並列にそのプログラムを

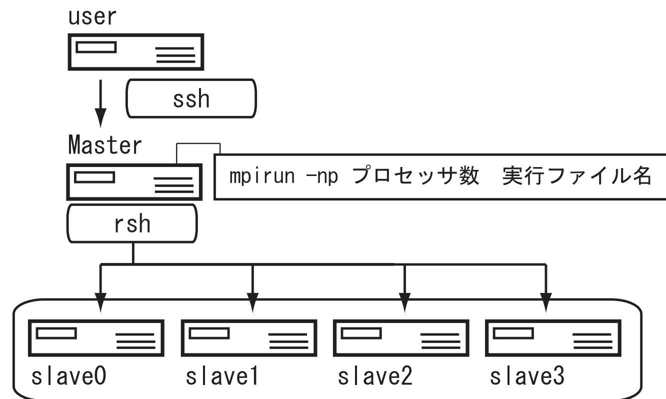


Fig. 7　MPI の実行法

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.

これは, 単純に,「IF 文」などによって場合分けをするだけであり, 特に難しいことはしていない. この ID の割り振り自体も 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　MPICH の使用法

5.1　C 言語を用いたコンパイル法

MPI は，C プログラムのコンパイルのフロントエンドとして「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パッケージであるので，C コンパイラに並列計算ライブラリである MPI のライブラリパッケージ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. 卓越した人になれば, 様々なオプションを付け足すことでさらにプログラムを高性能にすることもできる.

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

5.2　MPI プログラムの実行方法

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ればよい.

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

第一引数である「-np」に対して, 使用するプロセッサ数を指定している. その後に実行対象を指定する. この場合,「samplecpi」が実行対象である. あらかじめ使用するマシンなどの設定は管理者が別のファイルで設定しているので, 通常はこれで一連の計算プログラムが起動され, 各マシンで実行される. 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.

コマンドを見ると分かるように MPICH では, プログラム起動時にプロセス数について静的に宣言する必要がある. この点は，MPI-2 では，動的にプロセスを起動できるように改良される予定である.

6 MPICH のプログラム

6.1 MPI の枠組みと意味

MPI での各ランクでのプログラムは，SIMD 型の並列計算機として動作する. つまり, 全プロセッサに対して同じプログラムがロードされるが, おのおのが自分のなすべき部分をこなすことになる. 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. 8 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. 8 プロセッサへの仕事の割り振り

6.2 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 こ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.

6.3 送受信に必要な情報

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.1) 整数値の tag」でプログラマが値を指定する.2)communicator で指定する.

6.4 MPI の基本的な通信関数

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. ここでは, いくつかの通信パターンを示す.

6.4.1 1 対 1 の通信

先ほど述べたように，1 対 1 の通信にはブロッキング通信，ノンブロッキング通信の 2 通りの通信方法が存在する。まず，ブロッキング通信で主に使われる関数を紹介する。

■ **ブロッキング通信**

```
MPI_SEND(buffer,count,datatype, destination,tag,communicator)
```

基本的なブロッ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(buffer,count,datatype,source, tag,communicator,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, とにかく最初な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.
Fig. 9 に MPI_Send と MPI_Recv を用いた通信方法の例を示す.

```
MPI_SENDRECV(sendbuf,count,datatype,tag,
communicator,recvbuf,count,datatype,tag,communicator)
```

基本的なブロッ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)

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.
MPI_SENDRECV の応用であるシフトについて，Fig. 10 に示す。

■ ノンブロッキング通信

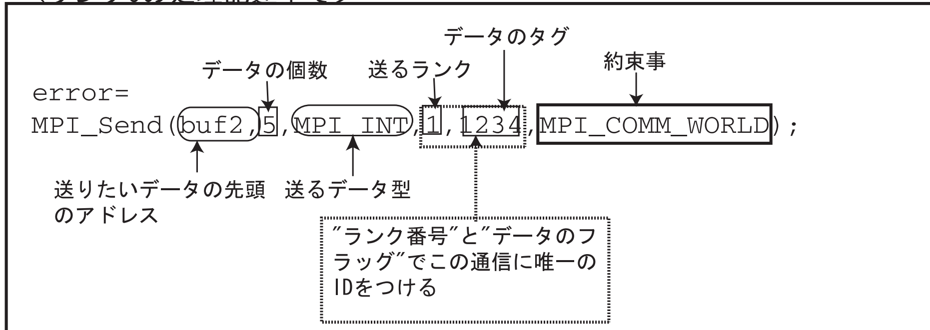
次に，ノンブロッキング通信で用いる関数を紹介する。ノンブロッキング通信では，送受信の手続きの完了を待たずに次の操作が行われるため，送受信の完了を明示しなければならない。これは，MPI_Wait() によって宣言する。

・一対一通信

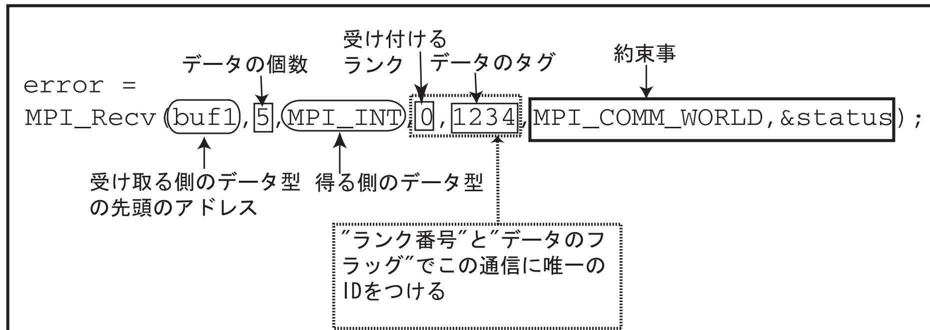
ex. ランク０からランク１に一個の配列のデータを先頭から5個送りたい場合

```
int buf2[100];
```

〈ランク0の処理記述中で〉



〈ランク1の処理記述〉



ex. 次の配列のデータを送りたい場合

```
int a[50];
```

の中のはじめから20データを送りたいとき

配列名は自動的にその配列の先頭アドレスに変換される

```
error=
MPI_Send(a,20,MPI_INT,1,5678,MPI_COMM_WORLD);
```

ex. 次のデータを送りたい場合

```
int n;
```

を単に送りたいとき

単に”n”ではデータのアドレスにならない
アドレス参照演算子”&”をつける

```
error=
MPI_Send(&n,1,MPI_INT,1,1111,MPI_COMM_WORLD);
```

Fig. 9　MPI プログラムでの通信法

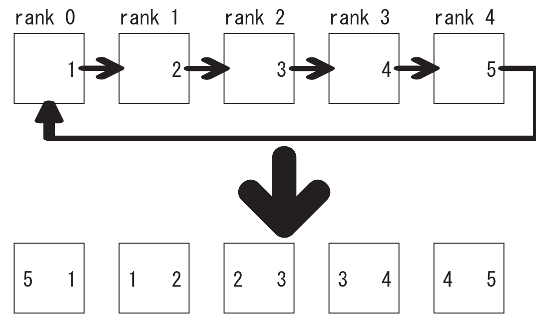


Fig. 10　シフトによるデータの移動.

```
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.

6.4.2　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(sendbuf,count,datatype,root,communicator)
```

Fig. 11 のように，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.
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 )
```

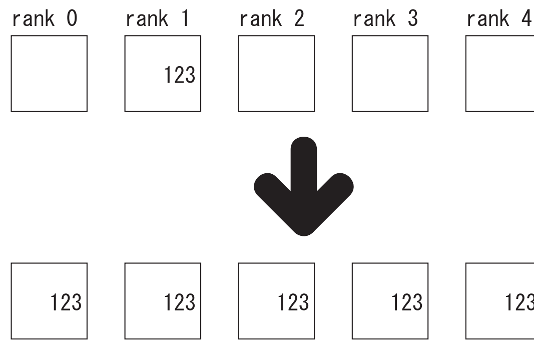


Fig. 11　ブロードキャストによるデータの移動.

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
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(sendbuf,count,datatype,recvbuf,datatype,communicator)
```

各ノードが持っている値を一つのノードに集める (Fig. 12).

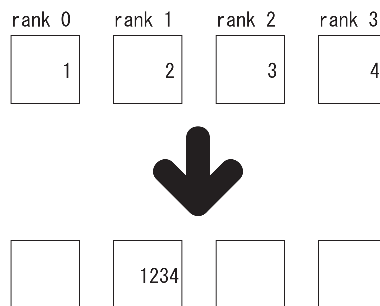


Fig. 12　MPI_GATHER によるデータの移動.

MPI_SCATTER

１つのノードが持っているデータを書くのどに配る. 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. 13).

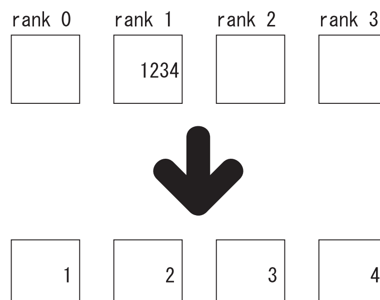


Fig. 13　MPI_SCATTER によるデータの移動.

MPI_ALLGATHER

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 (Fig. 14).

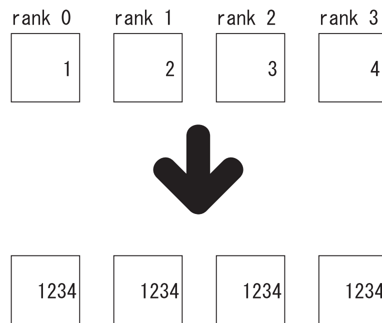


Fig. 14 MPI_ALLGATHER によるデータの移動.

MPI_ALLTOALL

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 (Fig. 15).

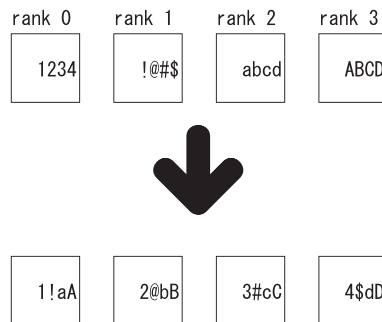


Fig. 15 MPI_ALLTOALL によるデータの移動.

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/

6.5 MPI を用いたプログラムの例

6.5.1 例. 1

[hello メッセージ]

```
--------------------------------------------------------------------------------
#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_Init(&argc,&argv);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    count=sizeof(outmsg)/sizeof(char);

    if(myid == 0){
        src =1;
        dest =1;
        /*"hello"という文字列データをランク１に*/
        MPI_Send(outmsg,count,MPI_CHAR,dest,tag,MPI_COMM_WORLD);
        MPI_Recv(inmsg,count,MPI_CHAR,src,tag,MPI_COMM_WORLD,&stat);
        printf("%s from rank %d\n",inmsg,src);
    }else{
        src =0;
        dest =0;
        MPI_Recv(inmsg,count,MPI_CHAR,src,tag,MPI_COMM_WORLD,&stat);
        MPI_Send(outmsg,count,MPI_CHAR,dest,tag,MPI_COMM_WORLD);
        printf("%s from rank %d\n",inmsg,src);
    }

    MPI_Finalize();
    return 1;

}
-----------------------------------------------------------------------------
```

6.5.2　例．2

[配列の総和]

```
-----------------------------------------------------------------------------
#include <mpi.h>
#include <stdio.h>
#define M 1000000
int main(int argc,char *argv[]){
    int i,j,rank,pnum,data[M],local_sum=0,global_sum,N;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&pnum);
    for(i=0;i<pnum;i++){
       if(rank==i){
           N=(M+i)/pnum;
           for(j=0;j<N;j++)data[j]=1;
           for(j=0;j<N;j++)local_sum+=data[j];
       }
   }
    MPI_Reduce(&local_sum,&global_sum,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
    if(rank==0)printf("Sum = %d\n",global_sum);
    MPI_Finalize();
    return(0);
```

```
}
-------------------------------------------------------------------------------
```

関数 MPI_Reduce は Fig. 16 のように順送り方式で，各プロセスの値を総和する関数である．

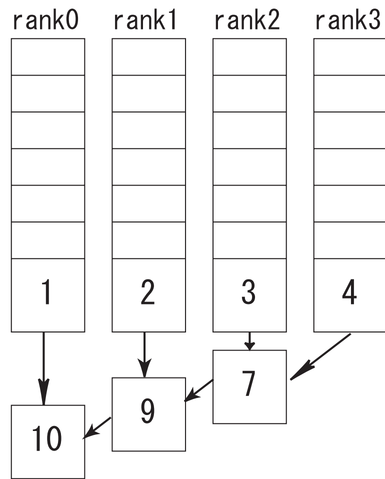


Fig. 16　順送り方式の総和

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/.