第2回並列ゼミ

指導者 佐野正樹
チーフ 下坂久司
サブチーフ 田村隆一 松山靖彦

1 MPI とは

MPI とは「Message Passing Interface」のことをさしている. 並列処理には様々なモデルがあるが, MPI はその一形態ということができる.「Message Passing」とは, プロセッサ間で互いに通信してメッセージ (データ) 交換したり同期を取ったりすることをさしている. しかし, 通信といっても様々な種類が存在する. たとえば共有メモリモデルの並列処理マシンにおいては通信はメモリバスを通して直接行うが, ワークステーションクラスタにおいては TCP/IP などのネットワーク越しに通信を行っている. このように並列処理マシンのアーキテクチャによって通信方法が異なるためその実装も全く異なったものになる. このユーザにとってこの大変な部分を代行してくれるものこそが「MPI ライブラリ」である.

1.1 MPI の歴史

MPIF(Message Passing Interface Forum) は,40 を越える団体の参加を受けて, 1993 年 1 月から会合を繰り返し, メッセージ通信のためのライブラリインターフェースの標準に関する議論と定義を行った. その活動の中から,1994 年に標準規格の第一版が発表された. つまり,MPIF が MPI の生みの親である. そして, その後も改良が加えられ,1997 年に MPI2.0 が発表され各ベンダーが実装を進めている.

1.2 MPI の目標

1. ネットワーク上のプロセス間の効率の良い通信を可能とする
2. 異なるプラットホーム上への移植のしやすさの保証
3. 信頼できる通信インターフェースを提供する
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.

1.3 MPI ライブラリについて

MPI ライブラリとは,MPI を実装したライブラリのことである.MPI というのは MPIF が「MPI スタンダード」という標準規格を作っている. この MPI 標準に準じて実際に実装されたライブラリが「MPI ライブラリ」である.

MPI そのものは, インターフェイスの規格でしかない. つまり,「`MPI_Send` 関数はデータを送信する関数で, 引数はこれ」といったことを規定している. この関数の動作や引数さえ標準に準じていれば, 中身をどう実装するかは実装者に委ねられている. たとえば, 全員にメッセージを送る「ブロードキャスト」という関数がある. これをどのように実装するかは実装者 (実際に MPI 規格に則ったライブラリの作成者) によって異なる. たとえば,Ethernet や IP の Broadcast 機能を使って一斉に送信する関数を実装するかもしれない. また, 1 台づつに Send 関数で送信するかもしれない. もちろん, 前者のほうがパフォーマンスはよいかもしれないが, 並列処理はワークステーションクラスタに限った話ではなく, WAN 上では Broadcast は基本的に好ましい通信法ではないので, ポータビリティは落ちることになる. MPI ライブラリは並列処理に必要な様々な通信関数などが実装されている. 並列処理をしたい人は, 通信処理などの本質以外の雑多な部分にとらわれずに, 純粋に並列アルゴリズムに集中することができる. また,MPI は標準規格なので, すべてのプラットホーム上での MPI ライブラリはその規格を則ったものを作成する. そのため, ソースコードの可搬性が保証され, 目的とする機種に MPI ライブラリが実装されていれば, 面倒な通信コードの手渡しなしに, ソースコードをコンパイルするだけで移植を完了することができる.

1.4 MPI ライブラリの実装について

MPI ライブラリをどう実装するかは実装者に委ねられると述べたが, 実際にインプリメントされたライブラリをいくつか紹介する (詳細は必要に応じて各自で調べてください). LAM は, ワークステーションクラスタに特化して実装

されたライブラリである. そのため, ほかの MPI ライブラリよりもワークステーションクラスタにおいては, 優れた性能を示す傾向がある.

また, 私たちの研究室で使用している MPICH は, 様々なアーキテクチャで動作するように実装されたライブラリである. このライブラリはワークステーションクラスタや SMP 等の共有メモリ型マシンまで, 様々なアーキテクチャで動作する. 実際には, 送信を行うためのドライバを取り替えることで対応するようになっている. たとえば, ワークステーションクラスタならネットワークを使った通信ドライバを, 共有メモリならメモリバスを使った通信ドライバをリンクします. ちなみに, 末尾についている"CH"は"Chameleon"から取ったものです.

1.5 MPI の構成

MPICH 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ている. そして, ソースコードも無償で配布していることや, 移植しやすさを考慮したつくりになっているため盛んに移植が行われている.

mpi+"コンパイラ名" mpi+"コンパイラ名" とは,MPI ライブラリのリンクなどの面倒な設定からユーザを解放するためのコンパイルスクリプトである. 私たちが一般的に,MPI を使用したプログラムを作成する際にはこれを使用する. つまり, ユーザは, あらかじめ GCC や G++,G77 といったコンパイラをあらかじめ導入しておく必要がある. 例えば,C のコンパイルには「mpicc」,C++のコンパイルには「mpiCC」などである.

mpirun mpirun は,MPI の一連のプログラムを走らせるためのプログラム起動スクリプトである. 私たちは, この起動スクリプトに対して色々な引数を渡すことで MPI プログラムを起動することができるようになる.

2 MPI の機能

2.1 MPI-1 の機能

1 対 1 の通信 MPI における最も基本的な通信機能. 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. ユニキャスティングであるといえる.

集団通信 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. ブロードキャスティングである.

グループ, コミュニケータ, コンテキスト だれがだれに送っているのかを明確にする機能. グループはプロセスの集合, コミュニケータは通信するプロセスのグループ, そしてコンテキストは通信で転送されるデータが何に関するものかを表す. これらによって,MPI は安全な通信を提供する.

プロセストポロジ グループのプロセスを仮想的に整理する機能.

MPI 環境管理と問い合わせ MPI のマシンへの実装と実行環境についての情報を取得するための機能である. 具体的には, 各種エラー処理や, 初期化など環境の構築機能である.

2.2 MPI-2 の拡張部分 (参考程度)

・動的プロセス生成 MPI1.0 では,MPI が立ち上げるプロセスは静的にはじめに指定する必要があった. しかし, 処理の推移によってプロセスを新たに立ち上げることができると有効な場合が存在する. それが, 動的プロセスの目指すものである.

・one-side 通信

・パラレル I/O I/O の並列化

・Fortrun90,C++言語向けの枠組み

・グラフィック

・実時間対応

・動的プロセス制御

3 MPI 実行環境の例

MPI はその実装については一般化されていても, 実際にエンドユーザに対して配布される際には, その環境ごとに若干の違いが生じることがある. 例えば,OS によってファイルの格納される場所は変わる可能性がある. さらには, 同じ OS 例えば,LINUX でもディストリビューションによってその格納場所は大きく異なることがある. そこで, 一例としてマシン名 forte,Debian/GNULinux2.2 に MPICH(バージョン 1.2.1-6) をインストールした場合を紹介する. クラスタの構成としては, マスター 1 台, スレーブ 40 台の構成になっている.MPICH は, マスターであるゲートにさえインストールすれば, 他のスレーブノードに対しては, インストールの必要はない. これは,MPI が PVM のようにデーモンを介して通信を行うようなことをせず直接通信機能をプログラムにライブラリとして導入するという形をとっていることに由来するといえる. リモートのスレーブノードへの実行に関しても, 外部として「rsh」, リモートシェルを利用していることも一因である.

3.1 ディレクトリ構造

MPI はライブラリであるから,MPI を利用したプログラムを作成する際にはライブラリをリンクする必要がある. 具体的にはライブラリや, インクルードファイル, 先ほど説明した「mpirun」「mpicc」などのファイルは以下のようなところに存在している.

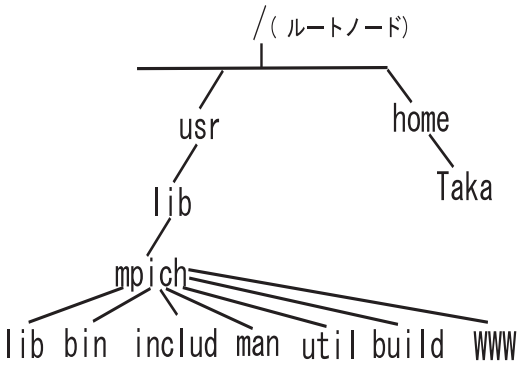


図 1: forte の MPI 関連のディレクトリ構造

lib この下には, MPICH に必要なリンクされるべきライブラリが収められている.

include この下には,「lib」と同じように,MPICH に必要なインクルードライブラリが収められている. 例えば,「mpi.h」である.

util この下には, 各種設定ファイルが収められている. 例えば, さらに下には,「machines.LINUX」という「mpirun」をしたときに, 起動されるマシン名, もしくは IP アドレスを記述することになる.

bin この下には, 先ほども説明した一連のスクリプト群, つまり「mpi ○○」が収められている.

WWW この下には,MPICH に関連したサイトのアドレスが html ファイル形式で収められている.

man この下には,man コマンドでみるための mpich 関連マニュアルが収められている.

3.2 今回使用する MPI の環境とは？

今回使用する環境は, クラスタにおけるメッセージパッシングを用いた並列化である. ライブラリとしては,MPICH を利用している. クラスタの仕組みは, 図.2 のようである. マスターである「forte」のアドレスは「202.23.147.73」であり, ここへ SSH を用いて接続することでクラスタを利用する.

使用法 ssh でマスタ (forte) にログインし，そこからスレーブノード (forte01〜forte04) で実行する.
rsh [スレーブノード名]

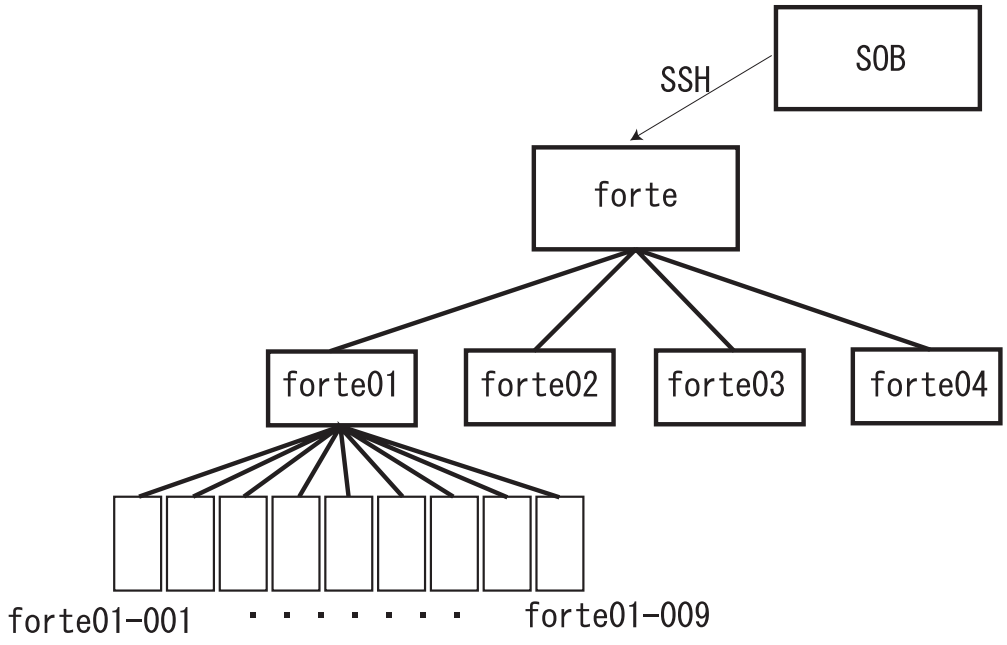


図 2: forte のクラスタ環境

4 MPI プログラミング

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

すべての MPI プログラムに共通する枠組みを下に示す.

MPI ヘッダファイルの読み込み
MPI の初期化
MPI を使った並列処理
MPI の終了処理

MPI ヘッダファイルの読み込み 最初に,MPI ライブラリを使うためのヘッダファイルを読み込む必要がある. ヘッダファイルには,MPI 独自の設定済み定数や, 変数の宣言が入っている.

MPI の初期化 MPI ライブラリを利用するために必要な準備 (初期化) を行う. 一連のこれらの処理のうち, `MPI_Init` は, 他のすべての MPI 関数の呼び出しに先立って呼び出す必要がある.

MPI を使った並列処理 上記の一連の処理をした後に実際に並列処理が行われる.

MPI の終了処理 何事も後始末が必要である. そして,MPI においても `MPI_Finalize` 関数を呼び出し, 後始末を行う.

4.2 メッセージ通信の実現

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. ランク 0 から送信された数をランク 1 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.

ランク 0	ランク 1
処理してほしいデータをランク 1 へ**送信**	ランク 0 からデータを**受信**
	受信データに対して計算
ランク 1 からデータ**受信**	ランク 0 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	

以上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, 集団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 MPI 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,MPI の実行に関して的を絞って話をする. コンパイルは, 通常マスタで行う. つまり SSH で接続した相手である forte において実行する. すると, 実行ファイル自体は,forte 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[1] 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図.3 のようにファイルの共有が実現される.

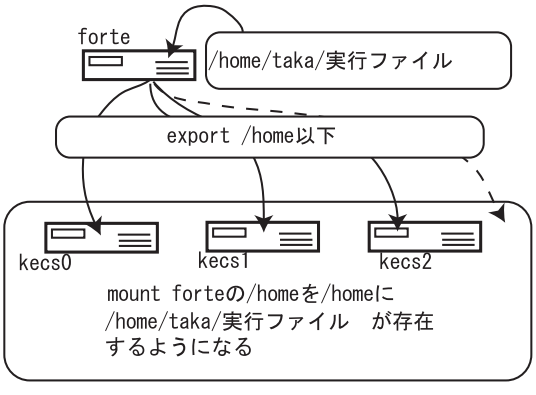


図 3: Sparcgate のファイル共有

4.4 マスターからスレーブへの実行の依頼

MPI 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「rlogin」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

[1]Network File System

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. ここまでの流れを図 4 に示す.

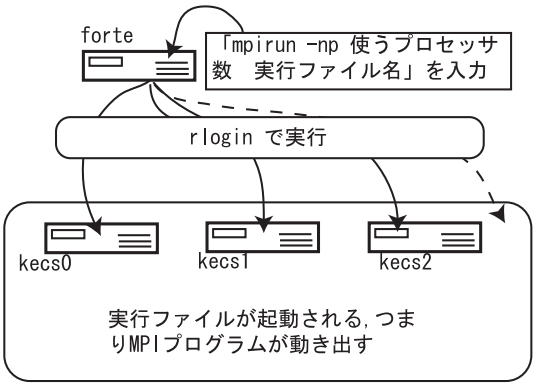


図 4: MPI の実行法

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 MPI-CH 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ます. MPI 自体は単なるライブラリパッケージですので,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す. 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す. 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ます.

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

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るだけです.

```
bash$ mpirun -np 8 cpi
```

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ます. その後に実行対象を指定します. この場合,「cpi」が実行対象です. 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ます. この際には, マスタからリモートシェル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ます. 以下の図.5 に使用法について簡単に説明します. これを見るとわかるように, 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ります. 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す.

6 MPI-CH のプログラム

6.1 MPI の枠組みと意味

ここで, 再び MPI プログラムの枠組みと意味について図.6 に示す.

次に, 各ランクでのプログラムの動きであるが, MPI では 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.

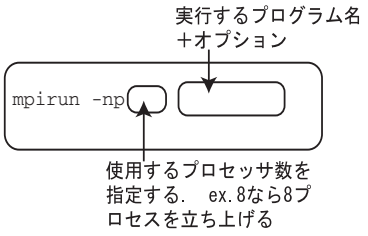


図 5: mpirun の使用法

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, 図 8 のようになる.

6.2 MPI の基本的な通信関数

MPI-1 には 127 個の通信関数が用意されている. 次に代表的なものを幾つか示す.

6.2.1 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.

6.2.2 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.

6.2.3 通信パターン

MPI には様々な通信パターンに対応する関数が用意されている. 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.

1 対 1 の通信

1. 送信関数：

```
MPI __ SEND(buffer,count,datatype,destination,tag,communicator)
```

   buffer:送るデータの先頭アドレス,count:送るデータ数,datatype:送るデータの型,destination:あて先プロセスのランク,tag と communicator:先述

2. 受信関数：

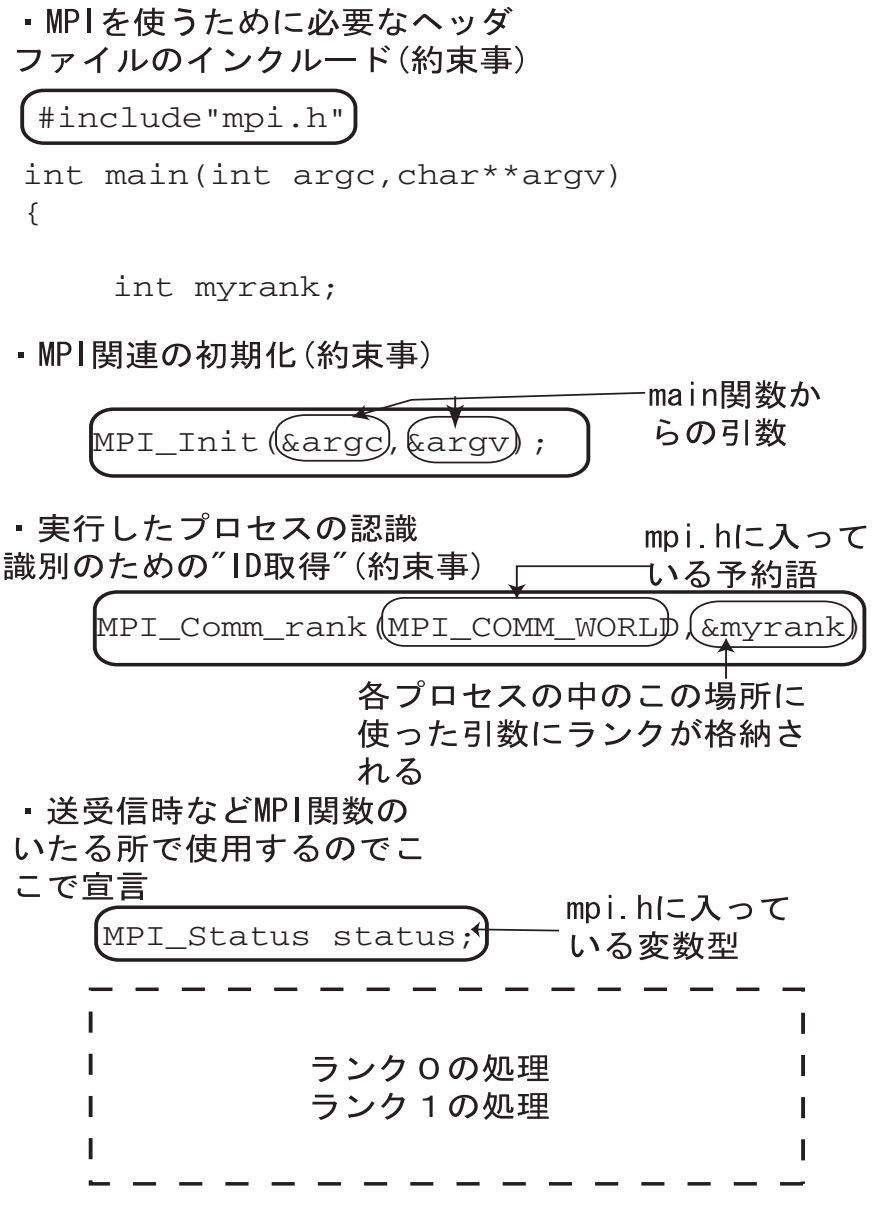


図 6: MPI プログラムの枠組み

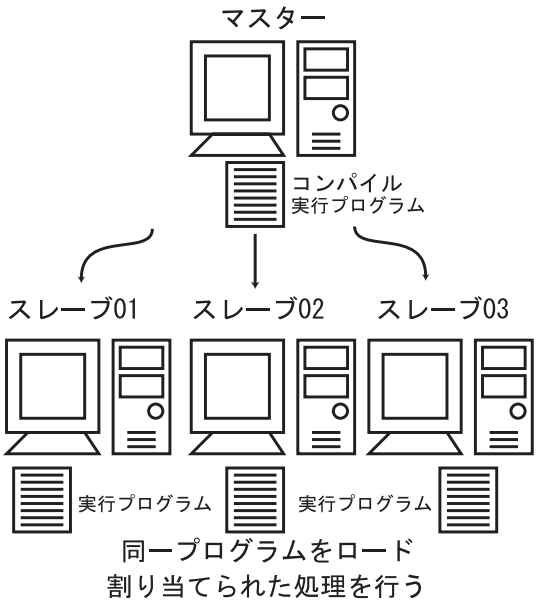


図 7: プログラムのロード

・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
    }
```

図 8: プロセッサへの仕事の割り振り

```
MPI__RECV(buffer,count,datatype,source,tag,communicator,status)
```

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__ANYSOURCE 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, 受信された目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.

3. 交換関数：

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

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.

4. シフト：MPI__SENDRECV の応用 (図 10).

集団通信

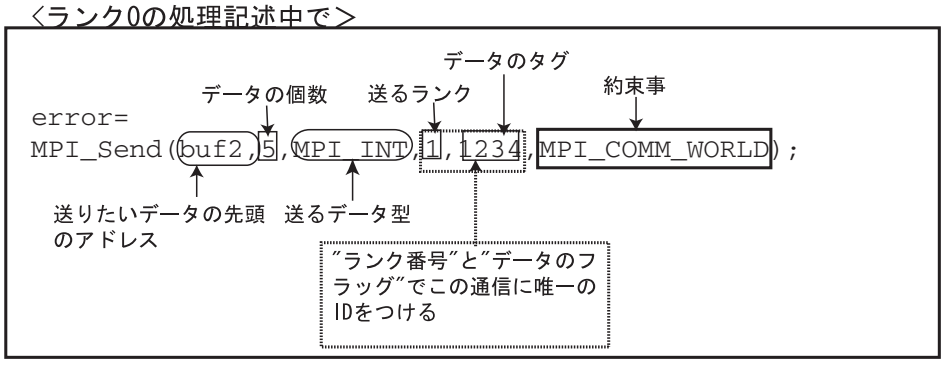
集団通信では, 通信に参加するすべてのプロセスが, 同じ関数を呼び出し, 引数として同じコミュニケータを与える.

・一対一通信

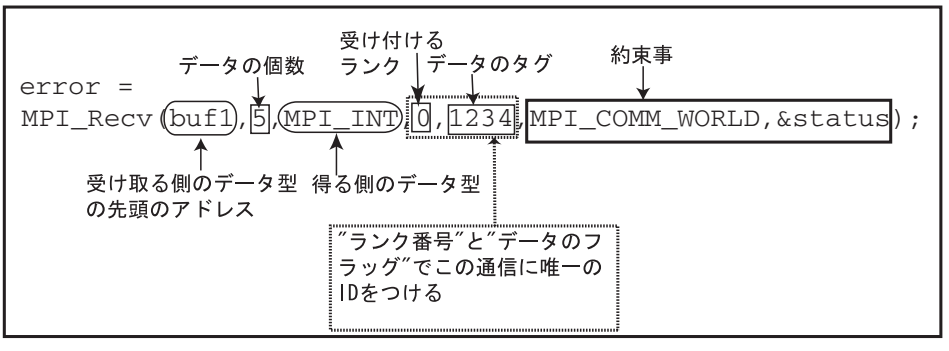
ex. ランク０からランク１に一個の配列のデータを先頭から5個送りたい場合

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

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

ex. 次のデータを送りたい場合

```
int n;
を単に送りたいとき
error=
MPI_Send(&n,1,MPI_INT,1,1111,MPI_COMM_WORLD);
```

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

図 9: MPI プログラムでの通信法

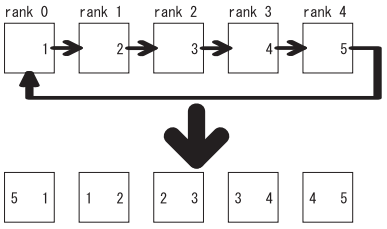


図 10: シフトによるデータの移動.

1. ブロードキャスト：

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

一つのプロセスから, 複数のプロセスのすべてに対して同じメッセージを送る (図 11).

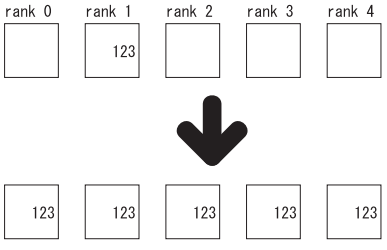


図 11: ブロードキャストによるデータの移動.

```
2. 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 (図 12).

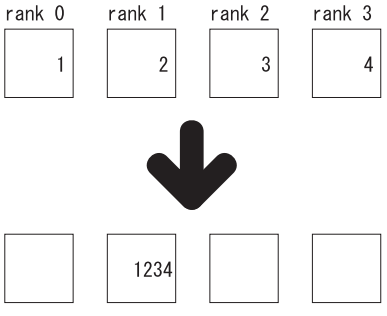


図 12: MPI__GATHER によるデータの移動.

3. 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 (図 13).

4. 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 (図 14).

5. 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 (図 15).

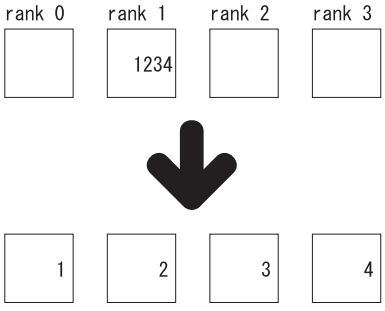


図 13: MPI __ SCATTER によるデータの移動.

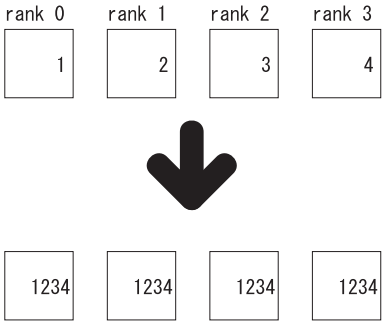


図 14: MPI __ ALLGATHER によるデータの移動.

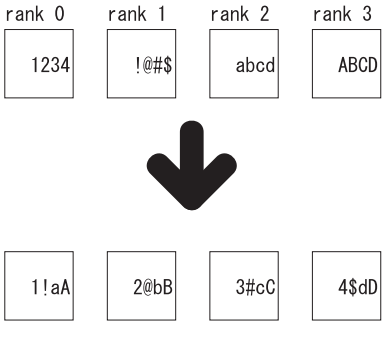


図 15: MPI __ ALLTOALL によるデータの移動.

その他の通信関数に関しては以下の URL などを参考にしてください.

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

6.3 MPI を用いたプログラムの例

6.3.1 例．1

[プログラムの枠組み]

```
-------------------------------------------------------------------------------
#include "mpi.h" // ヘッダファイルの読み込み

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

    MPI_Init(&argc,&argv); // MPI ライブラリを利用するための準備 (初期化)
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
                        //  コミュニケータ内のプロセスの数を取得
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
                       //  コミュニケータ内の各プロセスが自分の rank を取得
      /* 並列処理の記述 */

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

6.3.2 例．2

[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;

}
```

6.3.3 例．3

[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 は順送り方式で，各プロセスの値を総和する関数である．

7 課題

クラスタ forte を使用し，例２３もしくは他の並列プログラムを実行してください．

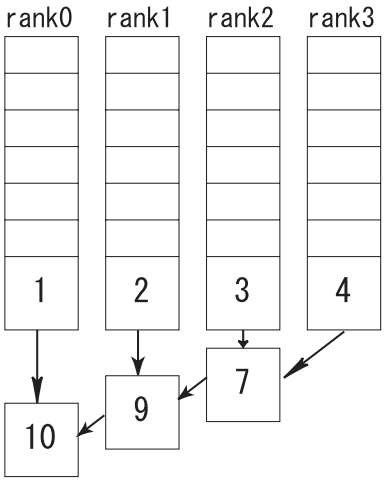


図 16: 順送り方式の総和

参考文献

[1] 谷村勇輔『Message Passing Library を用いた並列プログラミング』

[2] 湯銭太一・安村通晃・中田登志之『初めての並列プログラミング』