講義 3

MPIによる並列プログラミングの基礎

同志社大学 大学院　工学研究科

渡邉 真也

sin@mikilab.doshisha.ac.jp

■　はじめに

ここでは，MPIを用いた並列プログラミングについて初歩的な基礎事項を含めて解説する．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実装である．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，複数存在する．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，Mac 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，どのようなクラスタ環境においても MPI はフリーで使用することができる．以下では並列プログラミングを始めようとする初心者の方を対象として，MPI を用いた簡単なプログラム作成への手引き書として解説を行っていく．

■　並列プログラミングの基礎知識

具体的なプログラミングの説明に入る前に，MPI を中心とした幾つかの基礎事項ついて説明する．

□　メッセージパッシング方式とデータ並列方式

クラスタのような分散メモリシステムでは，並列化インタフェースとして **メッセージパッシング方式** を用いるのが一般的である．無論，MPI もメッセージパッシング方式の一つである．

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，プロセッサ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，複数のプロセッサ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ッサ間の通信であるが，メッセージパッシング方式ではプロセッサー間での通信をお互いのデータの送受信にて行う．

そのため，

・プロセッサー i 上でプロセッサ j にデータを送る．

・プロセッサー j 上でプロセッサ i からデータを受ける．

が必ずペアとして成り立たなければならない．

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が MPI である．

□ MPI

分散メモリ型のメッセージパッシングライブラリの代表としては MPI と PVM(Parallel Virtual Machine) があげられる．しかし近年，事実上の標準は MPI となっている．ここでは，本講の主題である MPI についてその歴史的背景を含めて簡単に説明する．

MPI

MPI は Message Passing Interface を意味し，メッセージ通信のプログラムを記述するために広く使われる「標準」を目指して作られた，メッセージ通信の API 仕様である．

MPI の経緯について説明する．MPI の標準化への取り組みは Supercomputing'92 会議において，後に **MPI フォーラム** として知られることになる委員会が結成され，メッセージ・パッシングの標準を作り始めたことで具体化した．これには主としてアメリカ、ヨーロッパの 40 の組織から 60 人の人間が関わっており，産官学の研究者，主要な並列計算機ベンダのほとんどが参加した．そして Supercomputing'93 会議において草案 MPI 標準が示され，1994 年に初めてリリースされた．MPI の当初より掲げていた目標を以下に示す.

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.

MPI の成功を受けて，MPI フォーラムはオリジナルの MPI 標準文書の改定と拡張を検討し始めた．この MPI-2 フォーラムにおける最初の成果物として，1995 年 6 月に MPI1.1 がリリースされた．1997 年 7 月には，MPI1.1 に対する追加訂正と説明がなされた MPI1.2 と，MPI-1 の機能の拡張を行った MPI-2 がリリースされた．MPI-2 の仕様は基本的に MPI-1 の改定ではなく，新たな機能の追加であるために，MPI-1 で書かれたプログラムが MPI-2 をサポートするプラットフォームで実行できなくなるということはない．これらの MPI 文書は，MPI フォーラムのサイト（ `http://www.mpi-forum.org/` ）から入手可能である．また MPI-1，MPI-2 の日本語訳が MPI-J プロジェクトにより公開されており， `http://www.ppc.nec.co.jp/mpi-j/` から入手可能である．

MPI-2 への主な取り組みとしては，

- 入出力（I/O）
- Fortrun90，C++言語向けの枠組み
- 動的プロセス制御
- 片側通信
- グラフィック
- 実時間対応

などが挙げられる．この中で特に注目すべきは動的プロセスの制御の導入である．これまで，PVM では実現されていた機能であるが MPI-1 では実現されていなかった．この機能実現は，単に MPI のプログラミングがより高度なことが実現できるようになったというだけでなく，PVM の優位性が一つ崩れたという意味でも重要である．

□ MPI ライブラリ

前述のとおり MPI は，仕様（規格）そのものであり実装では無い．MPI ライブラリとは,MPI を実装したライブラリのこと，すなわち実装そのものである．

MPI にはその仕様に準じた幾つかのライブラリが存在する．実装によっては一部の MPI の関数が使えないものもあるが，ほとんどの主要な MPI 関数は利用することができる．ただし、現在はまだ MPI-2 を完全にサポートした実装はない．MPI がサポートされているシステムは，専用の並列計算機からワークステーション，PC に至るまでと幅広い．下に示す表に，フリーに提供されているものとベンダによって提供されている主な実装を紹介する．

［Freeware MPI Implementation］

実装名	提供元
	ホームページ
	サポートされているシステム
CHIMP/MPI	Edinburgh Parallel Computing Centre(EPCC)
	ftp://ftp.epcc.ed.ac.uk/pub/chimp/release/
	Sun SPARC(SunOS 4,Solaris 5),SGI(IRIX 4,IRIX 5), DEC Alpha(Digital UNIX),HP PA-RISC(HP-UX),IBM RS/6000(AIX), Sequent Symmetry(DYNIX),Meiko T800,i860,SPARC,Meiko CS-2
MPICH	Argonne National Laboratory
	http://www-unix.mcs.anl.gov/mpi/mpich/
	MPP では IBM SP,Intel ParagonSGI Onyx, Challenge and Power Challenge,Convex(HP) Exemplar,NCUBE, Meiko CS-2,TMC CM-5,Cray T3D, TCP で接続されたネットワーク上では，SUN(SunOS,Solaris),SGI, HP,RS/6000,DEC Alpha,Cray C-90, その他多数のマシン
LAM	Laboratory for Scientific Computing,University of Notre Dame
	http://www.mpi.nd.edu/lam/
	Sun(Solaris 2.6.1,2.6),SGI(IRIX 6.2-6.5), IBM RS/6000(AIX 4.1.x-4.2.x),DEC Alpha(OSF/1 V4.0), HPPA-RISC(HP-UX B.10.20,B.11.00), Intel x86(LINUX v2.0,v2.2.x)
WMPI	Universidade de Coimbra - Portugal
	http://dsg.dei.uc.pt/wmpi/intro.html
	Windows 95,Windows NT

［Vendar MPI Implementation］

IBM Parallel Environment for AIX-MPI Library	IBM Corporation
	http://www.ibm.com/
	Risc System/6000,RS/6000 SP
MPI/PRO	MPI Software Technology
	http://www.mpi-softtech.com/
	Redhat Linux,Alpha アーキテクチャ,Yello Dog PPC(Machintosh G3 box), Windows NT,Mercury RACE
Sun MPI	Sun Microsystems,Inc.
	http://www.sun.com/software/hpc/
	すべての Solaris/UltraSPARC システム, またそのクラスタ

□ LAMとMPICH

MPIはインターフェースの規定であり，実装パッケージそのものではない．MPIの実装ライブラリとしてはフリー，ベンダ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，LAMとMPICHがあげられる．ここでは，LAMとMPICHの2つのライブラリについて簡単に述べる．もし，これらのMPIのライブラリに関するより詳細な情報に興味があれば `http://www.mpi.nd.edu/MPI/` を参考にして頂きたい．

LAM

LAM(Local Area Multicomputer)は，ノートルダム大学の科学コンピュータ研究室(Laboratory for Scientific Computing, University of Notre Dame)が作成したフリーのMPIライブラリである．LAMは，標準的なMPI_APIだけでなく幾つかのデバッキングとモニタリングツールをユーザに提供している．MPI-1を完全にサポートしているだけでなくMPI-2の標準的な幾つかの要素についても機能を提供している．

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．そのため，LAMは世界中のほとんどのUNIX系ベンダの並列マシン上で利用することができる．ただし，WindowsといったUNIXベースでないベンダに関してはサポートされていない．また，LAMはXMPIとの相性が良くXMPIを使用したいのであれば非常に便利である[1]．

2001年6月現在の最新バージョン6.5.2ではMPI-2における1方向通信，動的プロセスの管理に関する機能がカバーされている．LAMはMPICHと比較してバージョンアップの期間が短くインストールするときには常に， `http://www.mpi.nd.edu/lam/` を参考にする必要がある．また，本URLは内容が充実しており，非常に多くの有益な情報を得ることができる．

MPICH

MPICHは，アメリカのゴードン国立研究所(Argonne National Laboratory)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，LAM同様，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．特に，MPICHではUNIX系に限らずWindows系へのサポートも充実している[2]．さらに，SMP，Myrinetなどのハード面にも対応している上，DQS，Globusといった様々なツールを使用できることも大きな特徴の一つである．

その中身は，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．例えば，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するといった具合である．ちなみに，末尾についている”CH”は”Chameleon”から取ったものである．

また，2001年6月現在における最新バージョンMPICH 1.2.1では，MPI-1.2の全ての機能をカバーしており，MPI-2に関しても幾つかの機能についてはサポートしている．MPICH 1.2.1におけるMPI-2への詳細なサポート情報については `http://www-unix.mcs.anl.gov/mpi/mpich/mpi2-status.html` に載せられている．また．MPICHに関する詳細な情報は，`http://www-unix.mcs.anl.gov/mpi/mpich/` を参考にして頂きたい．ここでのInstallation GuideではMPICH導入に関する詳細な情報を得ることができる．

□ MPIの機能

先ほど，MPI-2における拡張機能について紹介したが本講では，MPIの基礎的な使用方法のみを扱う．そのため，以下での話はMPI-1に焦点を当ててMPIの機能説明を行う．

[1]ただし，6.5.2ではXMPIをまだサポートしていない
[2]Windows NT/2000には対応している

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 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 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.

図 3.1: MPI-1 の機能

MPI-1 で実現されている主な機能を図 3.1 に示す.

図 3.1 において，特に重要なのは，1 対 1 通信，グループ間通信，MPI 環境管理である．本講でもこれらの 3 点について特に詳細に説明していく.

■ インストールに関して

ここでは，並列ジョブを実行するための最低限の準備 (環境設定) に関して説明する．尚，ここでのインストールを行う OS は，Linux を想定している．

□ インストール

他の講義と重なっている部分があるが，ここでは LAM と MPICH のインストール，特にコンフィグについて述べる．インストール方法として，パッケージを用いたインストールの方が非常に簡単かつ便利であるので，もしパッケージを用いたインストールが可能であるならばそちらでも構わない．しかし，ここでは MPI ライブラリインストールの基本的な仕組みの理解が目的であるため，あくまでソースコードレベルからインストールを前提として説明する．

また，インストールにおけるポリシーとして今回は，LAM と MPICH の共存を目指す．というのは，実際の利便性や特徴において両者は大きく異なる．例えば，自前のクラスタのベンチマークを測定しようとする場合，MPICH を用いた場合の性能と LAM を用いた場合の性能は異なる．そこで，LAM と MPICH を上手く共存させて使いこなすことを目標に以下説明する．

コンフィグ

LAM や MPICH は，様々な環境や目的に対応するため多くの機能を持っている．しかし，単純にそれらのライブラリの機能を全てインストールするのでは，無駄な機能が多くなり容量が大きくなるだけでなく，自分の使っている環境とマッチせずライブラリとしての本来の機能を発揮できない恐れがある．

それを避けるためインストールの前準備として，設定（コンフィグ）作業を行い，一体自分はどういった環境でのライブラリ使用を求めているのか，ライブラリのどういった機能を実現したいのかを明示的に指定してやる必要がある．

具体的には，コンフィグによって得られた情報に基づき make ファイルが変更され，その結果，求める形態のライブラリがインストールされるという仕組みになっている．

□ MPI ライブラリのディレクトリ構造

LAM や MPICH はライブラリであるため,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. ここでは，MPICH を例に `/usr/mpi/` にインストールした場合のディレクトリ構造を図 3.2 に示す.

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.

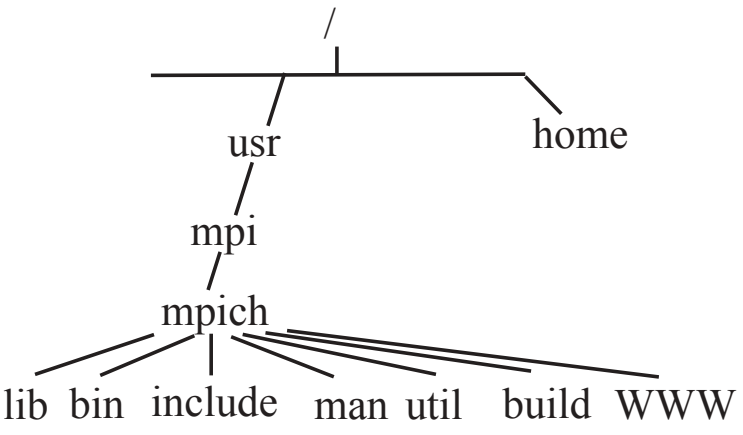

図 3.2: MPICH のディレクトリ構造

図 3.2 におけるディレクトリ構造の内，実際の MPI プログラムを走らせる上で重要なものは，include, bin, man である．LAM では，若干異なったディレクトリ構造になっているが，これらの「include, bin, man」に関しては MPICH と全く同名で存在する．

LAM

LAM のインストール方法について示す．ドキュメントでの推奨では，**/usr/local/lam-6.***/へのインストールになっているが，特に指定は無い．LAM は MPICH と異なりプロセスの起動がマスターからの rsh を用いるといった方法では無い[3]．そのため MPICH では，ホストマシンさえインストールされていれば良いのに対して LAM では，各マシン毎にライブラリをインストールするか，もしくはライブラリの部分を NFS で共有させてやる必要がある．尚，バージョンは 6.4-a3 を用いた．

- **アーカイブの入手**

  まず，LAM のアーカイブを `http://www.mpi.nd.edu/lam/download/` より入手する．

- **展開，コンパイル**

  手に入れたソースを以下のようにコンパイル，コンフィグ，メイクを行う．

```
$ tar xvfz lam-6.4-a3.tar.gz
$ cd lam-6.4-a3
$ ./configure --prefix=/usr/mpi/lam --with-romio --with-rpi=usysv
   //  configure に関する詳細については，$./configure --help|less によって得られる．

        -prefix      ： インストールするディレクトリ
        --with-romio : ROMIO をサポート．ROMIO を組み込むことにより，MPI-2 から MPI-I/O
                       の機能を組み込むことができる．
        --with-rpi=usysv : ノード間通信に関する指定．LAM ではノード間通信の方法として，tcp，sysv，usysv の 3 種類が用意されておりそれぞれを指定することができる．rpi とは，Request progression Interface の略．デフォルトでは，tcp に設定されている．usysv とは，tcp と sysv のマルチファンクションである．詳細は，INSTALL ファイルなどを参照．
$make
   [ ·· lots of output ·· ]
```

[3]LAM と MPICH ライブラリは，MPI プロセス間での通信チャネルの起こし方が異なっている．クライアント to クライアントモデルにおいて，MPICH では要求によりコネクションが形成されるのに対して，LAM では初期状態において全てのネットワークがつながれる．そのためコネクションの立ち上がり時間において両者は若干の差がある．

MPICH

次に，MPICH におけるインストール方法について示す．前述したとおり MPICH では全てのマシンにインストールもしくはライブラリ部分を NFS で共有する必要はない．そのため，LAM に比べて多少インストールが楽と感じるかも知れない．ここでは，インストール先として，**/usr/mpi/mpich/** を想定する．

まず，MPICH のアーカイブを `http://www-unix.mcs.anl.gov/mpi/mpich/download.html` より入手する．尚，バージョンは 1.2.1 を用いた．

- **展開，コンパイル**

手に入れたソースを以下のようにコンパイル，コンフィグ，メイクを行う．

```
$su
#tar xvzf mpich.tar.Z
   [ ‥ lots of output ‥ ]
#cd mpich-1.2.1
#./configure -prefix=/usr/mpi/mpich --with-device=ch_p4 --with-arch=LINUX
--with-romio  -opt=-02 -fc=g77 -flinker=g77
   //  LAM と同様，詳細については，#./configure --help|less によって得られる
         --with-device:コミュニケーションデバイスの指定.
         --with-arch  :アーキテクチャの指定.
         -opt         :この最適化オプションは，MPICH を構築するに辺り使用されるもの

                       である．すなわち，mpicc,mpiCC,mpif77 といったスクリプトには
                       一切影響しない.
         -fc          ： fortran コンパイラの指定
         -flinker     ： fortran リンカ指定
   [ ‥ lots of output ‥ ]
#make
   [ ‥ lots of output ‥ ]
#make install
   [ ‥ lots of output ‥ ]
```

□ 各種設定

基本的なインストールが無事に終わると，次に MPI が利用できるように幾つかの設定を行ってやる必要がある．主な設定項目を以下に示す．

- 使用するマシンの登録
- パスの設定

以下では，上記の設定項目について，図 3.3 に示すモデルを参考にした場合を例に説明を行うものとする．

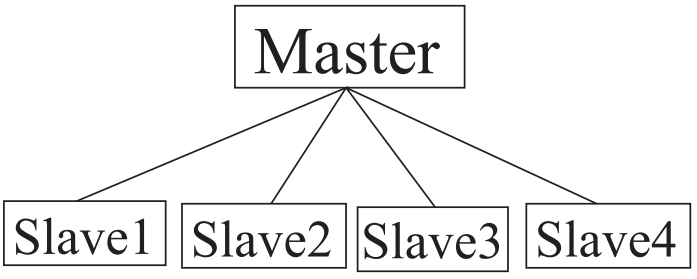

図 3.3: サンプルクラスタモデル

マシンの登録

LAM と MPICH では，起動するマシンファイルの扱いについて若干の異なりがある[4]．しかし，LAM・MPICH ともにプログラムの通信コマンドとして rsh を使用するため，**自分のホームディレクトリ**に `.rhosts` というファイルを作成し，使用する各ノードのホスト名を記入する必要がある．下記はその一例である．尚，`.rhosts` ファイルの属性（保護モード）が `-rwxr-xr-x` すなわち，755 になっているか注意する必要がある．特に，グループ，その他の権限において上書き可能になっていないようにしないと，MPI を実行できない場合がある． `.rhosts` ファイルの例を図 3.4 に示す[5]．

```
#This file is local rhosts file [.rhosts]
master.opt.isl.doshisha.ac.jp
slave01.opt.isl.doshisha.ac.jp
slave02.opt.isl.doshisha.ac.jp
slave03.opt.isl.doshisha.ac.jp
・
・
```

図 3.4: `.rhosts`

また，ユーザごとの設定が面倒な場合は /etc/hosts.equiv に記述すると良い．記述様式は，図 3.4 の場合と同様である（ただし root 権限が必要）．

さらにディストリビューションによっては標準状態において rsh が使用できないようになっている場合がある．その場合には適宜，/etc/inetd.conf を図 3.5 のように編集してやる必要がある．上記のように，in.rshd に関する行のコメント（#）を外せば良い．その後，inetd をハングアップさせるか，マシンの再起動が必要となる．

LAM

LAM では，MPI プログラムを走らせる前段階として lam デーモンの起動を行ってやる必要がある．そのデーモン起動のための起動マシンファイルとして lamhosts ファイル（ファイル名は何でも良い）が必要となる．記述の方法は，図 3.4 と同様である．尚，デーモンの起動に関しては MPI の実行の段階において詳述する．

また、/usr/mpi/lam/boot/bhost.def に lamhosts と同様の記述をすれば，lamhosts を指定する必要は無くなるため非常に便利である．

MPICH

mpich では，LAM と違い通信インタフェースとしてデーモンを用いることは無い．[6]．しかし，

[4]設定ファイルの中身自体はほぼ同じ

[5]ちなみに 行の先頭に#を記述するとその行はコメントとなる

[6]その代わり，ch_p4 という実装を用いている．p4 とは，MPI 以前より存在していたメッセージパッシングライブラリの一つである．mpich は p4 のコードをそのまま含有しており，ch_p4 は p4 の上に被さって構築されている

```
.
.
# Shell, login, exec, comsat and talk are BSD protocols.
#
shell stream tcp nowait root /usr/sbin/tcpd in.rshd
login stream tcp nowait root /usr/sbin/tcpd in.rlogind
#exec stream tcp nowait root /usr/sbin/tcpd in.rexecd
.
.
```

図 3.5: /etc/inetd.conf

MPI を実行する段階ではやはり明示的に使用するノードを指定してやる必要がある．具体的には，/usr/mpi/mpich/share/machines.LINUX を編集し，MPIで利用するマシン名の一覧を作成する．図 3.3 に示すモデルを参考にした場合について図 3.6 に例を示す．図 3.6 を見てもらえれば分かるように，図 3.4 とほぼ同様の書き方であることが分かる．

```
# Change this file to contain the machines that you want to use
# to run MPI jobs on.  The format is one host name per line, with either
#    hostname
# or
#    hostname:n
# where n is the number of processors in an SMP.  The hostname should
# be the same as the result from the command "hostname"
master.opt.isl.doshisha.ac.jp
slave01.opt.isl.doshisha.ac.jp
slave02.opt.isl.doshisha.ac.jp
slave03.opt.isl.doshisha.ac.jp
.
```

図 3.6: /usr/mpi/mpich/share/machines.LINUX

□ パスの設定

各 Linux パッケージに対応したバイナリファイルからインストールした場合[7]には，自動的に mpi 関連のコマンド (mpicc,mpiCC,mpirun など) のパスが通るため特に設定の必要ない．しかしソースファイルからコンパイルを行いインストールした場合には，各種パスの設定を行う必要がある．ここでは，bash の場合を例にコマンドラインへのパスの例を示す．

自分のホームディレクトリ内にある**.bashrc**,（MPICH ならば .bash_ profile でも構わない）に以下の内容を書き加える必要がある．

[.bashrc（LAM の場合）]

```
export MPI_CH=lam
export PATH=/usr/mpi/$MPI_CH/bin:$PATH
export MANPATH="$MANPATH:/usr/mpi/$MPI_CH/man"
```

[.bashrc もしくは，.bash_ profile（MPICH の場合）]

```
export MPI_CH=mpich
export PATH=/usr/mpi/$MPI_CH/bin:$PATH
export MANPATH="$MANPATH:/usr/mpi/$MPI_CH/man"
```

ただし上記の設定では，子プロセスとして bash が起動されるたびに， /usr/mpi/$MPI_CH/bin がパスに追加され続けてしまう．これを避けるためには，

[.bashrc（LAM の場合）: 環境変数べた書きの場合]

```
export MPI_CH=lam
export PATH=/usr/bin:/usr/local/bin:/usr/bin/X11 ··· :/usr/mpi/$MPI_CH/bin
export MANPATH=/usr/mpi/$MPI_CH/man
```

とパスをべた書きで固定すれば良い．尚，わざわざ MPI_CH 環境変数を設定してからそれを使用してパスの設定を行っているのは、mpich と lam との切り替え（共存）を簡単にするためである．

□ SMP を含んだクラスタでは ···

最新版の LAM・MPICH では，SMP クラスタのサポートも行われている．LAM では，設定ファイルに特に変更を行う必要は無いが MPICH では machies.LINUX に若干の変更を加えてやる必要がある．

MPICH

MPICH において SMP クラスタとしてサポートさせるには，システムの machies.LINUX (/usr/local /mpich/share/machies.LINUX) において，下記の様に記述すれば良い．ただしこれは，mpich のバージョンが 1.2.0 以降からの機能である．

[7] deb パッケージや rpm パッケージなど

[/usr/local/mpich/share/machies.LINUX]

```
master:2
slave01:2
slave02:2
slave03:2
.
.
```

■ 並列プログラミング

以降より実際に MPI を用いて並列プログラムを組み，最低限必要であろうと思われる通信関数について解説を行う．MPI においてデータを交換するための通信関数には，任意の 2 つのプロセス同士だけが関わる **1対1通信**（**2 地点間**）通信と，任意のグループに属するプロセス（プロセス全体も含む）が全て関わる **グループ通信**（集団通信）が用意されている．

並列プログラムの実行方法についてはさらに後の節で説明するが，プログラムを書くにあたって並列プログラムの実行の様子について概念的に理解しておく必要がある．手順を述べると，まずユーザはクラスタの 1 つのマシン，あるいは専用の並列計算機のホストとなっているマシンにログインを行う．そこでユーザはクラスタ，あるいは専用の並列計算機に対してジョブの投入を行う．ジョブの投入のために MPI ではスクリプト，専用の並列計算機では専用のコマンドが用意されている．プログラムが無事終了すると，ファイル，あるいはユーザのモニタに結果が出力される．このように実際に並列に走る個々のプロセスの起動は，実行環境に任されている．プログラマはプログラムの中で，最初に通信ライブラリを利用するための簡単な手続きを記述し，その後で実際の並列処理の部分を書いていけばよいことになる．

□ MPI の初歩

MPI-1 には 127 個の通信関数が用意されている．しかし，20 程度の関数を知っていれば，通信をかなり細かく制御することができる．さらにいうなれば，最低限の制御関数 (MPI_ Init()，MPI_ Comm_ size()，MPI_ Comm_ rank()，MPI_ Finalize()) と 2 つの通信関数 (MPI_ Send()，MPI_ Recv) さえ使いこなせることができれば，単純な並列プログラムは組むことができる．大事なことは，**最低限，必要な機能を覚えプログラムを組んでみること**である．

MPI では全てのプロセスを一斉に起動する．そして各プロセスは，他のほとんどの MPI 関数よりも先に MPI_ Init() を呼ばなければならない．MPI を用いた並列プログラムの枠組みを図 3.7 に示す．

rank ：コミュニケータ内の全てのプロセスは，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 をもっている．これは 0 から始まる連続した正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．

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一緒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．

よく用いられていると思われる MPI 関数の主なものを表 3.1 に示しておく．それ以外にも有用なものは存在するが，それは各実装に付属のドキュメントや MPI の仕様書などを参照されたい．特に，Web では

```
#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．この関数により numprocs
にはプロセス数が入力される．
        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．この関数によ
り myid には自分の rank 番号が入力される．

        /*  並列処理の記述 */

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

図 3.7: プログラムの枠組み

IBM のオンラインマニュアル（Web Book）の AIX に関するマニュアル http://www.jp.ibm.com/manuals/webbook/gc23-3894-01/index.html が各 MPI 関連のルーチンに関して詳細に載っているので参考にして頂きたい．

表 3.1: 主な MPI 関数

サブルーチン	タイプ	内容
MPI_ Init()	環境管理	MPI の実行環境の初期化
MPI_ Comm_ rank()	連絡機構	コミュニケータ内のランクを取得
MPI_ Comm_ size()	連絡機構	コミュニケータ内のプロセス数を取得
MPI_ Finalize()	環境管理	MPI の実行環境の終了 (全ての MPI 処理を終了)
MPI_ Send()	2 地点間	ブロッキング送信．最も標準的な送信関数
MPI_ Recv()	2 地点間	ブロッキング受信．最も標準的な受信関数
MPI_ Sendrecv()	2 地点間	ブロッキング送受信
MPI_ Isend()	2 地点間	ノンブロッキング送信.
MPI_ Irecv()	2 地点間	ノンブロッキング受信.
MPI_ Iprobe()	2 地点間	source、tag、および comm と一致するメッセージが着信しているか検査する.
MPI_ Probe()	2 地点間	source,tag, および comm と一致するメッセージが着信するまで待機する．MPI_ Iprobe と非常に似ているが，常に一致するメッセージが検出されてから戻るブロッキング呼出しであるという点が異なる.
MPI_ TEST()	2 地点間	ノンブロッキング送受信操作が完了しているか検査. MPI_ Iprobe にある種近い働きをする.
MPI_ Wait()	2 地点間	特定の非ブロック送受信の完了を待つ
MPI_ Waitall()	2 地点間	全ての非ブロック送受信の完了を待つ
MPI_Get_count()	2 地点間	メッセージの要素数を返す.
MPI_ Barrier()	集合通信	コミュニケータ内でバリア同期をとる
MPI_ Bcast()	集合通信	メッセージのブロードキャスト
MPI_ Reduce()	集合通信	コミュニケータ内で通信と同時に指定された演算を行う
MPI_ Type_ extent()	派生データタイプ	特定のデータタイプのサイズを取得
MPI_ Type_ struct()	派生データタイプ	新しいデータタイプを作成
MPI_ Type_ commit()	派生データタイプ	システムに新しいデータタイプを委ねる
MPI_ Type_ free()	派生データタイプ	データタイプの削除

■　１対１通信

MPI の１対１通信には，数多くの関数が用意されている．これにより，より細かく通信を制御することが可能である．しかしここでの説明は，基本的な送受信関数の紹介にとどめておく．ほとんどの MPI 関数のプロトタイプでは，成功した場合の戻り値は MPI_ SUCCSESS になる．失敗した場合の戻り値は実装によって異なる．

[１対１通信関数]

図 3.8 に示すプログラムは，rank0 と rank1 の 2 つプロセスがお互いに ”hello ”というメッセージを送り合うプログラムである．用いる関数は，ブロッキング通信である送信関数 MPI_ Send() と受信関数 MPI_ Recv() の２つである．

int 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)

int MPI_ Recv(void *buf, int count, MPI_ Datatype datatype, int source, int tag, MPI_ Comm comm, MPI_ Statusstatus)

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)

データタイプ

ポータビリティを高めるために，MPI によって前もって定義されたデータ型である．例えば，int 型であれば MPI_ INT というハンドルを用いることになる．プログラマは，新たなデータ型を定義することが可能である．基本データタイプを以下の表に示す．

タイプ	有効なデータタイプ引数
整数	MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT,MPI_UNSIGNED, MPI_UNSIGNED_LONG
浮動小数点	MPI_FLOAT, MPI_DOUBLE, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_LONG_DOUBLE
複素数	MPI_COMPLEX
バイト	MPI_BYTE

メッセージ・タグ ：メッセージを識別するためにプログラマによって割り当てられた任意の整数である．ワイルドカードとして MPI_ANY_TAG が存在するが，より安全に送受信を行うためには任意の整数を指定するのが望ましい．

MPI_Status :送信元のランクや送信時に指定されたタグの値を格納する構造体である．mpi.h では以下のように定義されている．

```
typedef struct {
    int count; //受信されるエントリの数 (整数型)
    int MPI_SOURCE; //送信先の情報
    int MPI_TAG;  //タグに関する情報
    int MPI_ERROR; //エラーコード
    int private_count;
} MPI_Status;
```

なお，返却ステータスである MPI_Status 内の変数 count にのみ直接アクセスすることができない．そのため，例えば受信されているエントリの数（データ数）を知りたい場合には，MPI_Get_count を用いて値を返してもらう．

```
#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;
                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 0;
}
```

図 3.8: Hello.c

□ もっとスマートに

先ほどのプログラム hello.c を，MPI_Sendrecv() を使うことによりもっと通信関数を減らすことができる．

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)

この MPI_ Sendrecv() 関数は，先ほどの MPI_Send() ，MPI_Recv() 関数の代わりに次のように用いることができる．

```
MPI_Recv(&inmsg,count,MPI_CHAR,src,tag,MPI_COMM_WORLD,&stat);
MPI_Send(&outmsg,count,MPI_CHAR,dest,tag,MPI_COMM_WORLD);
```

↓

```
MPI_ Sendrecv(&outmsg,count,MPI_ CHAR,dest,tag,&inmsg,count, MPI_ CHAR,src,tag,MPI_
COMM_ WORLD,&stat);
```

□ ノンブロッキング通信

1 対 1 通信では，**ブロッキング通信** と **ノンブロッキング通信** の 2 種類がある．先述の MPI_Send(),MPI_Recv() はブロッキング通信に当たる．ブロッキング通信とノンブロッキング通信に関して概念図を図 3.9，3.10 に示す．

しかしながら，実際の仕組みとしては，ブロッキング通信において相手側の MPI_Recv() が呼ばれる前に送信側（MPI_Send()）のブロッキングは解除される．このブロッキングという作業は，相手側のバッファに内容のコピーが完全に行われた時点で解除される．つまりブロッキングとは，送信内容を相手側のバッファへのコピー中を完全にブロックすることを意味する．

□ ブロッキング

ブロッキングとは，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．実際には，図 3.9 に示すように受信側が受信命令（MPI_Recv()）を呼び出すまでブロッキングが解除されないということは無いが，相手側のマシンへのコピー（バッファへのコピー）が完了するまでブロッキングが解除されないため，送信側に待機状態が生まれやすくなり，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に比べ簡単である．

□ ノンブロッキング（非ブロッキング）

ノンブロッキングとは，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，図 3.10 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

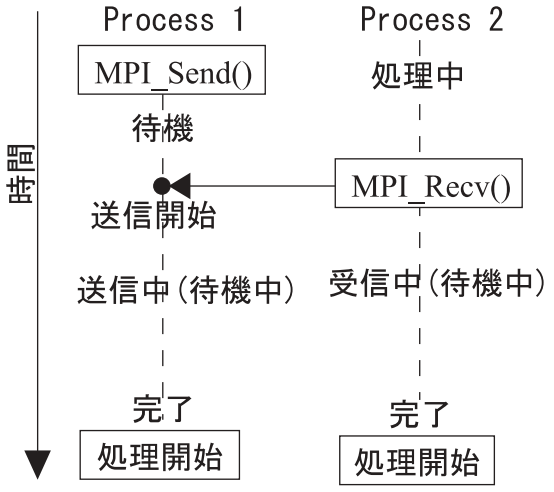

図 3.9: ブロッキング通信

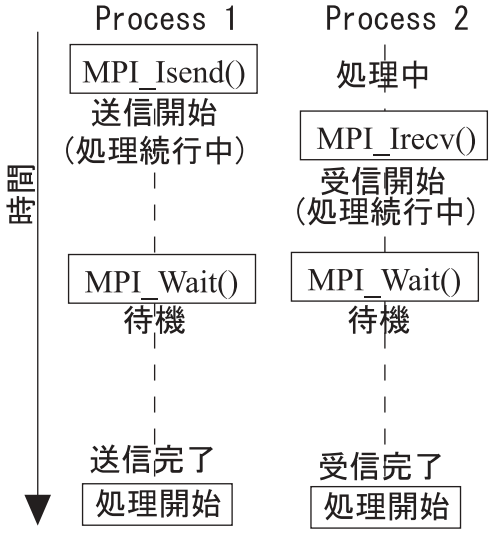

図 3.10: ノンブロッキング通信

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関数 (MPI_ Wait()) を呼び出す必要がある．

尚，ノンブロッキング関数には接頭語として I(即座 immediate) が必ずついている．また，基本的には 1 対 1 通信のみにしかノンブロッキング通信は存在しない．

1 対 1 通信においては，ノンブロッキング通信を用いる方が望ましい．というのも，ブロッキング通信では送受信の手続きが完了してから次の操作を開始するため通信での遅延が直接プログラムの計算時間に跳ね返ってくるためである．逆にノンブロッキング通信では，送受信の手続きを行いながら次の操作も平行して行うことができるため，通信での遅延を軽減することができる．

□ ノンブロッキン通信の使用方法

ここでは，ノンブロッキング通信での基本的な送受信方法について例を示す．ノンブロッキング通信における 1 対 1 通信では，送信ルーチンとして MPI_ Isend(), 受信ルーチンとして MPI_ Irecv() を用いる．また，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明示的に送受信の完了を宣言する必要がある．具体的には，MPI_ Wait() により明示的な完了を宣言する．

以下では．rank 0 と rank 1 の 2 つのプロセス間での簡単なノンブロッキングの送信，受信プログラムの例を示す．

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．

[isend_irecv.c]

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

int main(int argc,char *argv[])
{
        int myid,procs,src,dest,tag=1000,count;
        int date[100];
        MPI_Status stat;
        MPI_Request request;

        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        //略 (何らかのプログラム操作)

        if(myid == 0){
                src  = 1;
                dest = 1;
                count = 100;
                MPI_Irecv(&date,count,MPI_INT,tag,MPI_COMM_WORLD,&request);
                //略 (何らかのプログラム操作)
                MPI\_ Wait(&request,&stat);
        }else{
                src  = 0;
                dest = 0;
                count = 100;
                MPI_Isend(&date,count,MPI_INT,src,tag,MPI_COMM_WORLD,&request);
                //略 (何らかのプログラム操作)
                MPI\_ Wait(&request,&stat);
        }
        //略 (何らかのプログラム操作)
        MPI_Finalize();
        return 0;
}
```

■ グループ通信

□ π 計算のアルゴリズム

次にグループ通信について説明する．並列化を行う対象は π を近似的に求めるプログラムである．π は式 3.1 に示す計算式で求めることができる．またこれを逐次的に計算するプログラムは，積分計算が図 3.11 に示すように近似的に行われるので，0 から loop-1 個の区間の積分計算に置き換えられる．つまり．この積分計算のループ部分を並列化する．プログラム例では，複数のプロセッサで loop 個の区間の計算を分担するようなアルゴ リズムを採用する．

$$\pi = \int_0^1 \frac{4}{1+x^2} dx \tag{3.1}$$

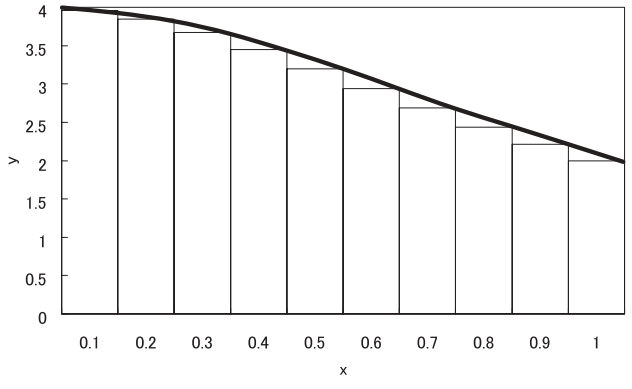

図 3.11: $y = \frac{4}{1+x^2}$

[逐次プログラム]

```
#include <stdio.h>

int main(int argc, char **argv)
{
        int i,loop;
        double width,x,pai=0.0;

        loop  = atoi(argv[1]);
        width = 1.0 / loop;

        for(i=0;i<loop;i++){
                x = (i + 0.5) * width;
                pai += 4.0 / (1.0 + x * x);
        }
        pai = pai / loop;
        printf("PAI = %f\n",pai);
        return 0;
}
```

□ MPI のグループ通信

[グループ通信関数]

MPI には１対１通信の他にコミュニケータ内の全てのプロセスが参加するグループ通信が用意されている．MPI では 16 個のグループ通信のための通信関数が用意されているが，ここでは π 計算のプログラムで用いられている MPI_ Bcast() と MPI_ Reduce() について説明する．π 計算のプログラムはこの２つの関数を用いて，図 3.12 に示されるような通信が行われる．まず最初に rank0 のプロセスにおいて，あらかじめ定義された，あるいはユーザによって入力された分割数を他の全プロセスに送信する．次に，それぞれのプロセスは自分の rank と分割数を照らし合わせて，各自が積分計算を行う区間を求め積分計算を行う．最後に計算結果を rank0 に送る．この通信においては，rank0 に各プロセスの計算結果を集めると同時にその総和をとり，π の近似値を求めている．サンプルプログラムを図 3.13,3.14 に示す.

int 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.

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
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

演算のハンドル：MPI によって前もって定義された演算を指定することで，通信と同時にどのような演算を行うかを指定する．例えば，各データの合計をとる場合には MPI_ SUM と指定する．またプログラマは，MPI_Op_create() 関数によって独自の演算を定義することができる．代表的な縮約演算子を以下に示す.

縮約演算子	内容
MPI_ MAX	最大
MPI_ MIN	最小
MPI_ SUM	合計
MPI_ PROD	積
MPI_ LAND	論理 AND
MPI_ BAND	ビット単位 AND
MPI_ LOR	論理 OR
MPI_ BOR	ビット単位 OR
MPI_ LXOR	論理 XOR
MPI_ BXOR	ビット単位 XOR

double MPI_Wtime ()

このルーチンは、time の現行値を秒数の倍精度浮動小数点数として返す．この値は過去のある時点からの経過時間を表す．この時点は、プロセスが存在している間は変更されない．

MPI_Get_processor_name(char *name,int *resultlen)

呼出し時にローカル・プロセッサの名前を戻す. この名前は文字ストリングで、これによって特定のハードウェアを識別する．name は少なくとも MPI_ MAX_ PROCESSOR_ NAME 文字の長さの記憶域を表し、MPI_ GET_ PROCESSOR_ NAME はこれと同じ長さまでの文字を name に書き込むことができる．また，実際に書き込まれる文字数は resultlen に戻される．

char *name：実際のノードに固有の指定子を返す.(OUT)
int *resultlen：name に戻される結果の印刷可能文字の長さを返す．(OUT)

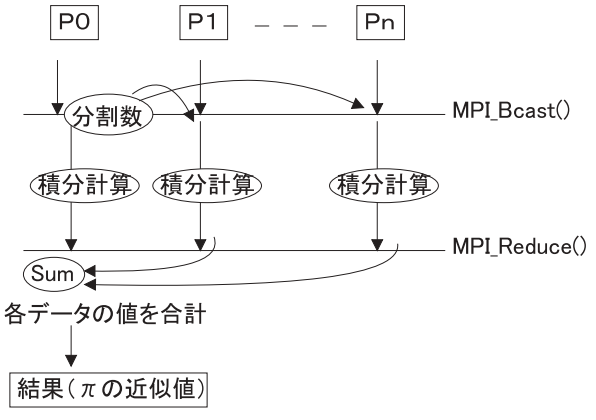

図 3.12: π の計算における通信

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

double f( double a ){ return (4.0 / (1.0 + a * a)); }

int main( int argc, char *argv[])
{
    int done = 0, n, myid, numprocs, i;
    double PI25DT = 3.14159265358979323846264 3;
    double mypi, pi, h, sum, x;
    double startwtime, endwtime;
    int  namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);

    fprintf(stderr,"Process %d on %s\n",myid, processor_name);
    n = 0;
    while (!done){
        if (myid == 0){
/*
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d",&n);
*/
            if (n==0) n=100; else n=0;
            startwtime = MPI_Wtime();
        }

        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0)
            done = 1;
        else{
            h   = 1.0 / (double) n;
            sum = 0.0;

            for (i = myid + 1; i <= n; i += numprocs){
                x = h * ((double)i - 0.5);
                sum += f(x);
            }
```

図 3.13: pi.c

[pi.c の続き]

```
            mypi = h * sum;
            MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

            if (myid == 0){
                printf("pi is approximately %.16f, Error is %.16f\n",
                       pi, fabs(pi - PI25DT));
                endwtime = MPI_Wtime();
                printf("wall clock time = %f\n",endwtime-startwtime);
            }
        }
    }
    MPI_Finalize();
    return 0;
}
```

図 3.14: pi.c の続き

■ MPIのプログラムの実行

□コンパイル

初めにMPIを用いたプログラムのコンパイル方法について説明する．プログラムのコンパイルには mpicc コマンドを用いる（ここでは pi.c を想定）[8].

```
\vspace{2mm}
$ mpicc -O -o pi pi.c
```

□ プログラムの実行

プログラムの実行の前にLAMの場合，実行の前にlam管理デーモン (lamd) を使用するマシン上で起動させてやらなければならない．何故，LAMにはデーモンの起動が必要でMPICHの場合には必要ないかというと，MPICHはプログラムの実行によりrshを用いてrootから各プロセスを起動しているのに対して，LAMは初期段階から各プロセスを起動した状態でプロセスを走らせるというスタイルをとっているためである．lamhostsファイルは，図3.4を参照のこと．

実行の前準備（LAM のみ）

前述の通り，LAMでは前段階としてデーモンの起動を行う必要がある．ここで間違えやすいのは，設定ファイルであるlamhostsの中にlocalhostが入っていなければならないことである．もし，localhostが設定ファイル内に含まれていなければエラーが出力される．

```
$ recon -v lamhosts  //lamhostsに記されているノードの確認
recon: -- testing n0 (duke.work.isl.doshisha.ac.jp)
recon: -- testing n1 (cohort1.mpara.work.isl.doshisha.ac.jp)
recon: -- testing n2 (cohort2.mpara.work.isl.doshisha.ac.jp)
recon: -- testing n3 (cohort3.mpara.work.isl.doshisha.ac.jp)
recon: -- testing n4 (cohort4.mpara.work.isl.doshisha.ac.jp)
recon: -- testing n5 (cohort5.mpara.work.isl.doshisha.ac.jp)
[ ‥ lots of output ‥ ]
$ lamboot -v lamhosts //lamhostsに記されているノード内からデーモン起動
LAM 6.3.2/MPI 2 C++ - University of Notre Dame

Executing hboot on n0 (duke.work.isl.doshisha.ac.jp)...
Executing hboot on n1 (cohort1.mpara.work.isl.doshisha.ac.jp)...
Executing hboot on n2 (cohort2.mpara.work.isl.doshisha.ac.jp)...
Executing hboot on n3 (cohort3.mpara.work.isl.doshisha.ac.jp)...
Executing hboot on n4 (cohort4.mpara.work.isl.doshisha.ac.jp)...
Executing hboot on n5 (cohort5.mpara.work.isl.doshisha.ac.jp)...
topology done
```

[8]LAMの場合， hcc コマンドでも同様のことを行える

無論,、/usr/sse/lam/etc/bhost.def に設定されたのであれば lamhosts の設定は必要なく下記のコマンドだけで良い.

```
$ recon -v
$ lamboot
```

実行

実行コマンドである mpirun は以下のように実行する.

```
mpirun -np [マシン数 ex. 5 ] [プログラム名]
```

上記におけるマシン数の指定は，* (*整数) のように行う．例えば 4 台使用したい場合には，4 となる．

mpirun はいくつかのオプションを指定することができる．これらについては,「mpirun -h」や「man mpirun」を参照されたい．例えば，プログラムを実行するノードをこの段階で明示的に指定した場合には，下記のようにオプションを加えてやれば良い.

```
$ mpirun -machinefile <machine-file name> -np [マシン数 ex. 5 ] [プログラム名]
```

尚，上記における¡machine-file name¿の書式は，図 3.4 と同様である.

以下に，π 計算のプログラムを実行した結果を示す.

π 計算の実行

[LAM の場合]

```
$ mpirun -np 8 chpi
22953 lampi running on n0 (o)
493 lampi running on n1
207 lampi running on n2
317 lampi running on n3
215 lampi running on n4
210 lampi running on n5
215 lampi running on n6
239 lampi running on n7
277 lampi running on n8
Process 0 on duke
Process 8 on cohort8
Process 1 on cohort1
Process 3 on cohort3
Process 5 on cohort5
Process 7 on cohort7
Process 2 on cohort2
Process 6 on cohort6
Process 4 on cohort4
pi is approximately 3.1416009869231245, Error is 0.0000083333333314
wall clock time = 0.006149
```

π 計算の実行

[MPICH の場合]

```
$ mpirun -np 8 chpi
Process 0 on duke.work.isl.doshisha.ac.jp
Process 1 on cohort10.mpara.work.isl.doshisha.ac.jp
Process 2 on cohort9.mpara.work.isl.doshisha.ac.jp
Process 5 on cohort6.mpara.work.isl.doshisha.ac.jp
Process 6 on cohort5.mpara.work.isl.doshisha.ac.jp
Process 7 on cohort4.mpara.work.isl.doshisha.ac.jp
Process 4 on cohort7.mpara.work.isl.doshisha.ac.jp
Process 3 on cohort8.mpara.work.isl.doshisha.ac.jp
pi is approximately 3.1416009869231245, Error is 0.0000083333333314
wall clock time = 0.016509
```

□ 計算後の処理 (LAM のみ)

LAM では，起動したデーモンを明示的に終了させてやる必要がある．具体的には，計算ジョブの終了を宣言して，並列計算用のデーモンプロセスを終了させるという作業を行う．

```
$ lamclean -v  //計算ジョブの終了
killing processes, done
closing files, done
sweeping traces, done
cleaning up registered objects, done
sweeping messages, done
$ wipe -v lamhosts  //   [ /usr/sse/lam/etc/bhost.def ]
                    // に設定されたのであれば lamhosts の設定は必要ない
LAM 6.3.2 - University of Notre Dame
Executing tkill on n0 (duke.work.isl.doshisha.ac.jp)...
Executing tkill on n1 (cohort1.mpara.work.isl.doshisha.ac.jp)...
Executing tkill on n2 (cohort2.mpara.work.isl.doshisha.ac.jp)...
Executing tkill on n3 (cohort3.mpara.work.isl.doshisha.ac.jp)...
Executing tkill on n4 (cohort4.mpara.work.isl.doshisha.ac.jp)...
[ ‥ lots of output ‥ ]
```

■ おわりに

MPIによる並列プログラミングの基礎と題して進めてきたが，まだまだ私自身が知識不足で満足のいく内容にはほど遠かったものの，MPIを始める初心者にとって一つの手がかりにはなったのではないだろうか．

今回のドキュメントはあくまでも初歩的な分野を広く浅く触れているだけに過ぎない．ここで，今回の講習を受講された皆さんに，知識の確認と向上のために下記に示す文献，URLを一読されることを強く勧める．紹介する文献，URLに簡単な紹介文も併記しておくので参考にしていただきたい．

`http://www.mpi.nd.edu/MPI/`

MPIの各種フリー，ベンダのライブラリの情報が載っている．そこから各ライブラリのメインのページへのリンクが張ってるためリンク集としても非常に価値のあるページである．

`http://www.mpi.nd.edu/lam/`

LAMのメインのページである．非常に良く整備されている上，LAMさらにはMPIに関する様々な情報を得ることができる．LAMを使用しないユーザーでも一読する価値はある．

青山幸也,「虎の巻シリーズ」,(日本アイ・ビー・エム株式会社，1999)

IBMの青山さんが執筆されたドキュメント．日本語の中では，恐らく最も充実したMPIに関するドキュメントである．基本的な並列プログラミングの考え方から，MPIを用いた数値計算プログラミング例，MPIの基本的な関数の使用方法までを網羅しておりかなり豊富な知識を得ることができる．MPIを本格的に学ぶ上では必読と言える．

`http://www.ppc.nec.co.jp/mpi-j/`

上記よりMPI-1，MPI-2の日本語訳ドラフトを手に入れることができる．ドラフト自体を読んでも量が膨大な上，使用方法に関して最低限の記述してしていないため少々読みづらい面は否めない．ただし，このドラフトが全ての基本となっているため手元にあると心強い．

`http://www.sse.co.jp/comid/engin/myrinet/no_frame/lets-cluster/tec/index.html`

SSE（住商エレクトロニクス株式会社）の技術資料．LAMを中心に書かれておりLAM初心者がMPIプログラミングをするのには，非常に良い資料が載っている．

`http://hamic6.ee.ous.ac.jp/software/mpich-1.2.1/`

岡山理科大学 工学部 電子工学科 橋本研究室のMPI・MPE（MPICH）に関するページ．MPICHにおけるMPI関数の一覧とその解説が載っている．MPI関数の使用方法，機能などを調べたいときには重宝する．