講義 2

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において，並列化インタフェースは大きく 2 つに分類される．一つは，**メッセージパッシング方式**であり他方が**データ並列方式**である．以下，MPI が用いているメッセージパッシング方式について説明する．

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リとして MPI と PVM がある．

□ MPI と PVM

現在，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，PVM が劣性に立たされており MPI の急速な延びが目立つ．以下では，MPI と PVM について概説し，今後の両者の動向についての観測について述べる．

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 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

PVM

MPI はインターフェイスの規定であるのに対して，PVM は実装パッケージそのものである．PVM は，TCP/IP ネットワークで接続された何台ものコンピュータを仮想的に 1 台のマシンととらえて，並列プログラムを走らせることのできるメッセージ・パッシングの環境とライブラリを提供する．その歴史は，1991 年にオークリッジ国立研究所とテネシー大学で開発に始まり，PVM プロジェクトとして発展をしてきた．PVM プロジェクトはその実験的性質から，副産物として科学者のコミュニティあるいはその他の分野の研究者に役立つようなソフトウェアをこれまで作り出してきている．元々は，ワークステーション・クラスタのための TCP/IP ベースの通信ライブラリであったが，現在では多くの並列計算機にも移植されている．

しかしながら，PVM では各並列計算機ベンダが独自にチューニングを施した独自の PVM を開発しており，移植性に乏しいという欠点がある．これは，MPI フォーラムのような第三者的なしっかりとした決定機関を保有していないことに起因している．

かつては分散メモリ環境では PVM が主流であったものの，現在は MPI に押されており Web 上における情報量も MPI と比較して圧倒的に少ないのが現状である．

PVM の本家サイトは `http://www.epm.ornl.gov/pvm/pvm_ home.html` である．ここで PVM の本体やチュートリアル，その他の様々な情報を得ることができる．PVM のソースコードは Netlib から取得可能である． **Netlib** は数値計算を中心とする科学技術計算に関するフリーソフトやドキュメントをアーカイブしているプロジェクトで，情報は `http://www.netlib.org/` から得ることができる．PVM のソースコードは `http://www.netlib.org/pvm3/index.html` から得ることができる．ここには，ソースコード以外にもドキュメントや仮想コンソールの GUI である XPVM，テストユーティリティ，論文，ユーザグループの活動報告など PVM に関するあらゆるアーカイブが置かれている．

MPI と PVM の比較

MPI と PVM に関する議論は，これまでにも幾つか行われている．最近の動向としては，MPI が新規格である MPI-2 の仕様において，それまでの PVM にしか無かった「動的なプロセス管理」の機能を取り入れるなど活発な動きを見せており，MPI がかなり優勢な状況となっている．

ここで両ライブラリの利点・欠点についてまとめる．

PVM は，ワークステーションをクラスタにすることで再利用するという世界で育ってきたものである．そのために不均質ないくつものマシンやオペレーティング・システムを直接に管理する．そしてダイナミックに仮想マシンを形成することができる．PVM のプロセス管理の機能では，アプリケーションの中から動的にプロセスを生成したり，停止したりすることができる．この機能は MPI-1 にはなく，MPI-2 で採り入れられている．さらに利用可能なノードのグループを管理する機能では，ノード数を動的に増減したり調べたりすることができる．これは，MPI ではまだ採り入れられていない．PVM の利用における最大の問題点は，先にも述べた移植性である．

それに対して MPI は，その実装が MPP（Massively Parallel Processors）やほとんど同一な特定のワークステーション・クラスタを対象としている．MPI は PVM の後に設計されたために，明らかに PVM における問題点を学んでいる．つまり MPI の方が PVM に比べて，高レベルのバッファ操作が可能であり，高速にメッセージを受け渡すことができる．そして，オープンなフォーラムによって達成された「標準」であるために，MPI で作成されたプログラムは非常に移植性が高い．ただし MPI の実装は数多くあるが，MPI-2 を完全にサポートしたものはまだない．また，Web や書籍などでの情報も MPI の方が入手しやすく便利であるといえる．

ここで，MPI のライブラリが集まる `http://www.mpi.nd.edu/lam/lam-info.php3` では，「MPI が PVM より優位である 10 の理由」と題して興味深いリストが載っていたので紹介する．ここでは，MPI が PVM に勝っている点として，

1. MPI はフリーの実行可能かつ高品質なインプリメンションを 1 つ以上持っている．

2. MPI は第 3 者的な決定機関を持っている．

3. MPI グループは完全に非同期通信をサポートしている.
4. MPI グループはしっかりとした基盤と持ち能率的で決定論的な側面を持っている.
5. MPI は効果的なメッセージバッファを扱うことができる.
6. MPI における同時性は第 3 者的なソフトウェア機関により保護されている.
7. MPI は効果的な MPP，クラスタのプログラミングを行うことができる.
8. MPI は全体的にポータブルである.
9. MPI は正式に明示されている.
10. MPI が標準である.

を挙げている．上記の理由の主な論拠は，

- **複数の信頼性の高いライブラリの存在**
- **MPI フォーラムの存在**
- **MPI の移植性の良さ**

などである．このことより，MPI において，MPI フォーラムの存在が MPI への信頼性の向上に大きく関与していることがわかる．また，PVM における最大の問題点である移植性の良さも MPI の最大の利点の一つとなっていることも分かる.

□ 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．最新バージョン 6.3.2 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.

また，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.

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．LAM に関する詳細な情報は， `http://www.mpi.nd.edu/lam/` を参考にして頂きたい．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．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．また，MPICH 1.2.0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 1.2.0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．

■ 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．

□ 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．尚，以下の内容はソースコードレベルからインストールを前提としている．

コンフィグ

ここでは，コンフィグをライブラリのインストールのための設定という意味で用いる．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，コンフィグという作業を行い，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．

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[1]．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/local/lam-6.4-a3 --with-romio
```

// `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．

[1] 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．

```
$make
   [ ‥ lots of output ‥ ]
```

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/local/mpich/** を想定する．

- **アーカイブの入手**

  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 1.2.0 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-1.2.0
#./configure -prefix=/usr/local/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 ‥ ]
```

- **マシンの指定する**

  /usr/local/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 2.1 に示すモデルを参考にした場合について図 2.2 に例を示す．

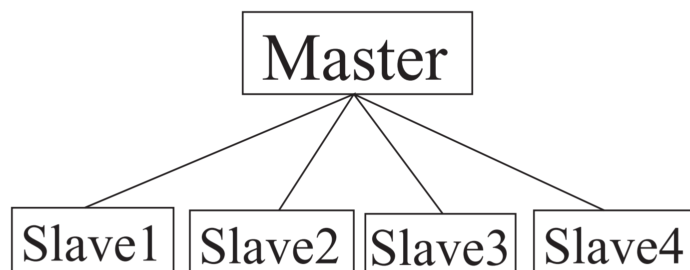

図 2.1: サンプルクラスタモデル

```
# 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
slabe01.opt.isl.doshisha.ac.jp
slabe02.opt.isl.doshisha.ac.jp
slabe03.opt.isl.doshisha.ac.jp
 .
```

図 2.2: `/usr/local/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 (無論，直接コマンドを打ち込んでも良い)．

[.bashrc（LAM の場合）]

```
export PATH="$PATH:/usr/local/lam-6.4-a3/bin"
export MANPATH="$MANPATH:/usr/local/lam-6.4-a3/man"
```

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

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

□ ノードの指定

LAM では，**作業ディレクトリ内**において `lamhosts` 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．また，LAM・MPICH ともに実行には，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．

[lamhosts]

```
master.opt.isl.doshisha.ac.jp
slabe01.opt.isl.doshisha.ac.jp
slabe02.opt.isl.doshisha.ac.jp
slabe03.opt.isl.doshisha.ac.jp
 .
 .
```

[.rhosts]

```
master.opt.isl.doshisha.ac.jp
slabe01.opt.isl.doshisha.ac.jp
slabe02.opt.isl.doshisha.ac.jp
slabe03.opt.isl.doshisha.ac.jp
 .
 .
```

□ SMP を含んだクラスタでは …

最新版の MPICH では，SMP クラスタのサポートも行われている．

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．

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

```
master:2
slabe01:2
slabe02:2
slabe03:2
 .
 .
```

MPI ビルド時のオプションに SMP 用のものがある．SMP を含むクラスタにおいて実験を行ったところ，逆に計算時間が増加するという結果が得られたものの，完全な SMP 単体での MPI プログラミングを行う場合には有効であると思われる．その場合，インストール時の config のオプションに – comm=shared を加えれば良い．尚，この場合の – comm とはコミュニケーションタイプのことを意味する．

■ 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．さらに，1 対 1 通信，グループ通信のそれぞれに **ブロッキング通信** と **ノンブロッキング通信** が用意されておりそれぞれを適所で使用することができる．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 2.3，2.4 に示す．

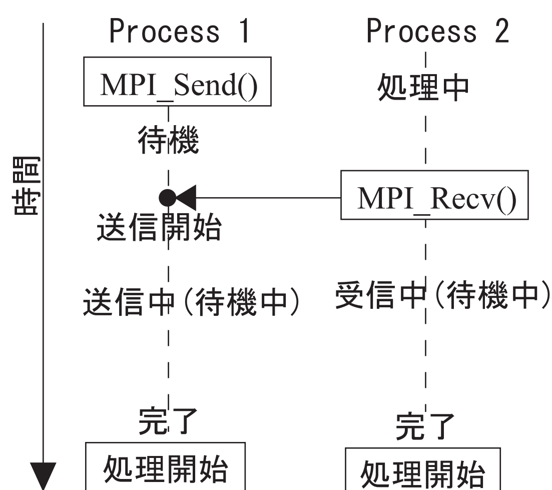

図 2.3: ブロッキング通信

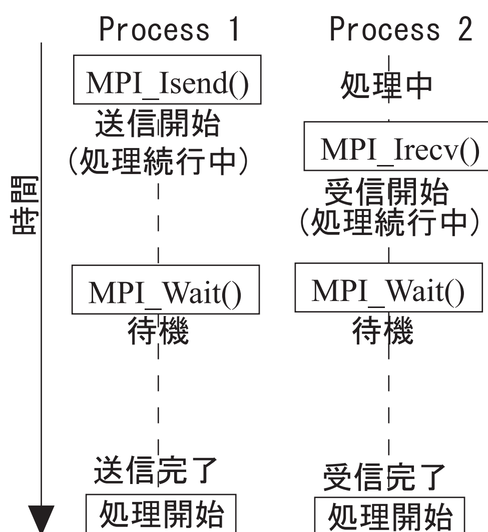

図 2.4: ノンブロッキング通信

□ 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．具体的には，図 2.3 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 2.4 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．

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．

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 2.5 に示す．

```
#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 2.5: プログラムの枠組み

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 2.1 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 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 2.1: 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 2.6 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 内の各変数に直接アクセスすることはできない．そのため，例えば受信されているエントリの数（データ数）など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 2.6: 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を多用しよう

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．

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,src,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 2.1 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 2.7 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．

$$\pi = \int_0^1 \frac{4}{1+x^2}dx \tag{2.1}$$

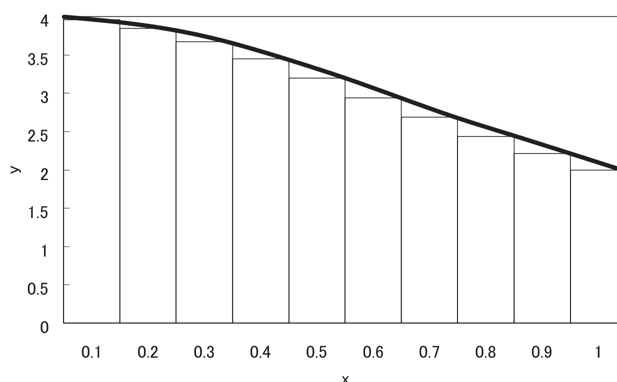

図 2.7: $y = \frac{4}{1+x^2}$

[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 2.8 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 2.9,2.10 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)

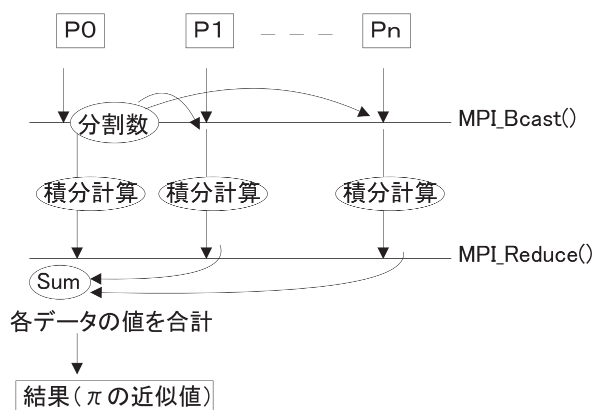

図 2.8: π の計算における通信

```
#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 = 3.141592653589793238462643;
    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 2.9: 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 2.10: pi.c の続き

■ その他

□ プログラム内の一部分の計測時間を計りたい

次のような場合，プログラム内の一部分の計測時間を測定することがある．

(A) 主力計算を行っているタイムステップのループのみの経過時間を測定したい．

(B) 並列化した部分の経過時間をプロセス毎に測定し，ロードバランスを調べたい．

(C) 並列化した部分の計測時間と，それに伴う通信時間をプロセス毎に測定して比較したい．

並列プログラムでは，各プロセスでその進行が異なっているため，各プロセス毎で計算時間も異なる．ここでは，全プロセスの最大値を求める場合 (maxtime_ mpi.c) と各プロセス毎にそれぞれ経過時間を求める場合 (each_ proctime_ mpi.c) について示す．

maxtime_ mpi.c では，最大値の計算時間を，each_ proctime_ mpi.c では各プロセス毎にそれぞれ経過時間を求めている．ここで重要なのは，計測前に必ず MPI_ Barrier() により同期をとることである．また，最大時間を求める場合には，任意の測定部分が終了した時点で再度同期をとり全てのプロセスの足並みをそろえる必要がある．

int MPI_ Barrier (MPI_ Comm comm)

すべてのプロセスが呼出しを完了するまでブロックする．プロセスは，すべてのプロセス・メンバの入力が完了するまで操作を終了することはできない．

[maxtime_ mpi.c]

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

int main(int argc,char *argv[])
{
        int myid,procs,src,dest,tag=1000,count;
        double s_time,e_time;
        MPI_Status stat;

        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);

        MPI_Barrier(MPI_COMM_WORLD); //← 測定前に同期をとる
        s_time = MPI_Wtime(); //← この部分以降からの時間を計測
        //略（測定部分）
        MPI_Barrier(MPI_COMM_WORLD); //← 全プロセスの足並みをそろえる．
        e_time = MPI_Wtime();
        printf("time = %f \n",e_time-s_time);
        MPI_Finalize();
        return 0;
}
```

[each_ proctime_ mpi.c]

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

int main(int argc,char *argv[])
{
        int myid,procs,src,dest,tag=1000,count;
        double s_time,e_time;
        MPI_Status stat;

        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        MPI_Barrier(MPI_COMM_WORLD);
        s_time = MPI_Wtime(); //← この部分以降からの時間を計測
        //略（測定部分）
        e_time = MPI_Wtime();
        printf("time = %f \n",e_time-s_time);
        MPI_Finalize();
        return 0;
}
```

□ 受信データの要素数が分からない場合

サンプルプログラムの hello.c，cpi.c では，受信データの要素数が予め分かっていたためスムーズに受信を行うことができた．しかし，場合によっては予め受信するデータの要素数が分かっていない場合もある．

そのような場合，単純に実際の送信データを前もって受信先に教えてやるというのが最も単純な方法として考えられる．しかし，そのために余計な通信を行わなければならず無駄な通信のオーバーヘッドが生じてしまう．そのような場合，MPI_ Probe(), もしくは MPI_ Iprobe() などを用いて実際には受信操作を行わずに送信されたデータの要素数を把握するのが最も効率の良いやり方である．

以下にその例として，データタイプとして byte 型の要素が要素数不明の状態で送られてきた場合について示す．流れとしては，

1. MPI_ Probe() によって送信されてきたデータ情報を MPI_ status に格納する．
2. MPI_ Get_ count() によって MPI_ status 内から送信データの要素数を把握する．
3. MPI_ Recv() を用いて具体的な要素数が把握できた段階において，実際の受信を行う．

となる．
#ただし，以下のプログラムでは char 型が 1byte であると仮定のもとで行っている．

[getcount.c]

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

int main(int argc,char *argv[])
{
        int myid,procs,src,dest,msgt,count;
        unsigned char *rna_recv;
        double s_time,e_time;
        size_t n1;
        MPI_Status stat;

        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        //相手が何らかのデータを送信した．
        MPI_Probe(MPI_ANY_SOURCE,msgt,MPI_COMM_WORLD,&stat); //送信データの収集
        MPI_Get_count(&stat,MPI_BYTE,&size); //送信データサイズの把握
        n1 = size*sizeof(unsigned char);
        if(( rna_recv = (unsigned char *)malloc(n1) ) == NULL ){
            printf("Error \n");
            exit(1);
        }
        MPI_Recv(rna_recv,size,MPI_BYTE,MPI_ANY_SOURCE,msgt,MPI_COMM_WORLD,
                 &stat);        //実際の受け取り
        MPI_Finalize();
        return 0;
}
```

□ 構造体を送信受信したい

実際に並列のプログラムをくみ出すと厄介になってくるのが構造体の送受信である．構造体のメモリを連続的に確保している場合には，Byte 型を用いて先頭アドレスを基準に構造体の size 分を指定し送受信を行う，もしくは MPI_ Type_ contiguous(),MPI_ Type_ commit() を使い新たなデータタイプを作成し登録することにより，より簡単に送受信を行うことができる．しかし，実際問題として連続的に構造体のメモリを確保することは難しい場合が多い．

そのため，構造体の中身を何度かに分けて送受信しなければならない状況が生じる．そのような無駄を極力省くため，送信前に構造体の中身を 1 つの Byte 型の変数に全てパックし，受信側で再度アンパックするという方法を用いる．この方法は，パック，アンパックという操作が生じるため小規模 (2 変数しか含まないような) の構造体の場合にはあまり効果は無いが，大規模の構造体では通信回数がかなり減るため計算時間の削減が期待できる．以下では，ある構造体を想定しこの構造体を送受信する際のパック，アンパックの一例を pack_ unpack.c に示す．尚，本例は並列分散 GA ライブラリ GAPPA のソースを参考にしている http://mikilab.doshisha.ac.jp/dia/GAPPA/ .

[pack_ unpack.c]

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <math.h>
 4 #include "mpi.h"
 5
 6 typedef struct {
 7     int a;
 8     int *b;
 9     double *c;
10 } Sample;
11
12 void all_malloc(struct Sample *s_sample,struct Sample *r_sample)
13 {
14        s_sample->b = (int*)malloc(sizeof(int) * 3);
15        s_sample->c = (double*)malloc(sizeof(double) * 3);
16        r_sample->b = (int*)malloc(sizeof(int) * 3);
17        r_sample->c = (double*)malloc(sizeof(double) * 3);
18 }
19
20 void free_malloc(struct Sample *a,struct Sample *b)
21 {
22        free(s_sample->b);
23        free(s_sample->c);
24        free(r_sample->b);
25        free(r_sample->c);
26 }
```

```
28 unsigned char *transcription(struct Sample *sample)
29 {
30         int i,j,n,mant;
31         unsigned char *buffer;
32         unsigned int mant_n,exp2;
33
34         mant = htonl(sample->a);
35         memcpy(buffer,&mant,4);
36         j=4;
37         for(i=0;i<3;i++){
38             mant = htonl(sample->b[i]);
39             memcpy(buffer+j,&mant,4);
40             j +=4
41         }
42         for(i=0;i<3;i++){
43             mant = (int)frexp(sample->c[i],&n)(1 << 30));
44             mant_t = htonl(mant);
45             exp2 = htonl(n);
46             memcpy(buffer+j,&mant_t,4);
47             memcpy(buffer+j+4,&exp2,4);
48             j += 8;
49         }
50         return buffer;
51 }
52
53 void reverse_transcription(const unsigned char *date,struct Sample *)
54 {
55         int n,i,mant,j;
56         double dum;
57
58         mant =  ntohl(*(unsigned int *)(data));
59         sample->a = mant;
60         j=4;
61         for(i=0;i<3;i++){
62             mant =  ntohl(*(unsigned int *)(data+j));
63             sample->b[i]=mant;
64             j+=4;
65         }
66         for(i=0;i<3;i++){
67             mant =  ntohl(*(unsigned int *)(data+j));
68             n = ntohl(*(unsigned int *)(data + j + 4));
69             dum = ldexp((double)mant, n - 30);
70             sample->c[i]=dum;
71             j += 8;
72          }
73 }
```

```
 75 int main(int argc,char *argv[])
 76 {
 77         int myid,procs,src,dest,msgt,count,size,partner;
 78         unsigned char *rna_recv,*rna_send;
 79         double s_time,e_time;
 80         struct Sample s_sample,r_sample;
 81         size_t n1;
 82         MPI_Status stat;
 83
 84         MPI_Init(&argc,&argv);
 85         MPI_Comm_rank(MPI_COMM_WORLD,&myid);
 86
 87              all_malloc(&s_sample,&r_sample);
 88
 89                 ////略 (何らかのプログラム操作)
 90              size = sizeof(int) * 4+sizeof(double)*3;
 91              rna_send = (unsigned char*)malloc(size);
 92              rna_recv = (unsigned char*)malloc(size);
 93         rna_send = transcription(&s_sample);
 94         if(myid==0)
 95             partner=1;
 96         else
 97             partner=0;
 98
 99         MPI_Sendrecv(rna_send,size,MPI_BYTE,partner,
100                      msgt,rna_recv,size,MPI_BYTE,MPI_ANY_SOURCE,
101                      msgt,MPI_COMM_WORLD,&stat);
102         reverse_transcription(rna_recv,&r_sample);
103         ////略 (何らかのプログラム操作)
104         MPI_Finalize();
105         all_free(&s_sample,r_sample);
106         return 0;
107 }
```

■ 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 LAM の場合 hcc コマンド，MPICH の場合， mpicc コマンドを用いる（ここでは pi.c を想定）．

LAM

```
$hcc -O -o lampi test.c -lmpi
```

MPICH

MPICH では，mpicc を利用したコンパイルを行う．

```
$ mpicc -O -o chpi 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，実行の前にデーモンの起動を行わなければならない．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．

LAM

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．

```
$ 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
```

実行コマンドである mpirun は以下のように実行する．

[LAM の場合]

```
mpirun -v [マシン数 ex.n0-5 ] [プログラム名]
```

上記におけるマシン数の指定は， `n0-*` (*整数) のように行う．例えば 4 台使用したい場合には， `n0-3` となる．

[MPICH の場合]

```
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．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 -v n0-8 lampi
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 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
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 ‥ ]
```

■ LAMとMPICHの通信速度

最後に，簡単な通信速度を測定するプログラムを用いて，LAMとMPICHの速度比較を行う．ここまで見てきたように，LAMとMPICHはコンパイル，実行において同じMPIという範疇でありながら若干異なっている．勿論，対応しているアーキテクチャ，DQSなどのジョブ管理システム(JMS)への対応など異なっている点も多く各々の目的に適したライブラリを選択する必要がある．

しかし，単純にIntel系のマシンを安価なハブでつなぎ合わしたクラスタにおいて最も重要になってくるのは，速度の違いであろう．並列化のその最大のメリットである速度について「通信速度」という観点から下記に示すような簡単なプログラムを用いて比較を行った．

プログラムについて，若干の解説を行う．本プログラムは，MPI_Sendrecv(),MPI_Bcast()をそれぞれ1000回行い，各通信関数にかかった時間を表示するというものである．LAMとMPICHではその仕組みの違いにより，立ち上がり時間に差が生じるが今回は，それを省く形で計測を行った．プログラム中の関数などはできる限り上記で説明したものだけを用いた．尚，MPI_Sendrecv()ではできる限り異なる送受信相手となるような仕組みを用いた．ちなみに，この送受信相手の選択方法は，ソートなどで良く用いられる方法である．

[test_mpi.c]

```
 1 #include <stdio.h>
 2 #include "mpi.h"
 3 #include <stdlib.h>
 4
 5 int main(int argc,char *argv[])
 6 {
 7         int myid,procs,src,dest,tag=1000,count,nemelen,proc,msgt;
 8         int i,j,avg_rank,range,dum,namelen;
 9         char flag;
10         MPI_Status stat;
11         char processor_name[MPI_MAX_PROCESSOR_NAME];
12         double startwtime, endwtime,midwtime;
13
14         MPI_Init(&argc,&argv);
15         MPI_Comm_rank(MPI_COMM_WORLD,&myid);
16         MPI_Comm_size(MPI_COMM_WORLD, &procs);
17         MPI_Get_processor_name(processor_name,&namelen);
18         printf("proces_name %s rank %d \n",processor_name,myid);
19         MPI_Barrier(MPI_COMM_WORLD);
20         startwtime = MPI_Wtime();
21         i=0;
22         flag=0;
```

[test_ mpi.c の続き]

```
     23         while(!flag){
     24                 avg_rank = procs/2;
     25                 range=procs;
     26                 for(j=0;;j++){
     27
     28                         if(myid<avg_rank){
     29                                 dest=myid+range/2;
     30                         }
     31                         else{
     32                                 dest=myid-range/2;
     33                         }
     34                         msgt=100;
     35                         MPI_Sendrecv(&j,1,MPI_INT,dest,msgt,&dum,1,MPI_I
NT,MPI_ANY_SOURCE,msgt,MPI_COMM_WORLD,&stat);
     36                         range /= 2;
     37                         if(range==1)
     38                                 break;
     39                         if(myid<avg_rank)
     40                                 avg_rank = avg_rank - range/2;
     41                         else
     42                                 avg_rank += range/2;
     43                 }
     44                 i++;
     45                 if(i>1000)
     46                         flag=1;
     47         }
     48         MPI_Barrier(MPI_COMM_WORLD);
     49         midwtime=MPI_Wtime();
     50         if(myid==0)
     51                 printf("Sendrecv 1000 clock time = %f\n",midwtime-startwt
ime);
     52         for(i=0;i<1000;i++)
     53                 MPI_Bcast(&dum,1,MPI_INT,0,MPI_COMM_WORLD);
     54
     55         MPI_Barrier(MPI_COMM_WORLD);
     56         endwtime=MPI_Wtime();
     57         if(myid==0)
     58                 printf("Bcast 1000 clock time = %f\n",endwtime-midwtime);
     59         MPI_Finalize();
     60         return 0;
     61 }
```

□ 結果

前ページのプログラムを実行した結果，下記のような値が得られた．

```
[LAM]
Sendrecv 1000 clock time = 1.242967
Bcast 1000 clock time = 0.282649

[MPICH]
Sendrecv 1000 clock time = 3.580716
Bcast 1000 clock time = 4.023609
```

上記の結果より，予想以上にLAMが高速であることが分かる．MPI_Sendrecv()，MPI_Bcast()ともに倍以上に差が開いおり，特にMPI_Sendrecv()に至っては5倍近くの差が生じている．まだ私自身の勉強不足により，この原因はハッキリとは分かっていない．この点については，至急調査を行う予定である．

■ 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．

`http://www.jp.ibm.com/manuals/webbook/gc23-3894-01/index.html`
: 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．各MPI関連のルーチンに関して詳細に載っている．MPI関数に関する使用方法，機能などを調べたいときは大変重宝する．

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．

■ 参考文献

1) 1999 年超並列講習会「PC クラスタ超入門」テキスト，「PC クラスタ超入門」，(http://is.doshisha.ac.jp/SMPP/report/1999/990910/index.html)

2) 1998 年並列処理講習会テキスト,「並列処理の基礎と実習」,（日本機械学会，1998.8）

3) 湯淺太一，安村通晃，中田登志之,「はじめての並列プログラミング」,（共立出版，1998.6）

4) Linux Japan 特集２,「並列処理とクラスタリング」,（レーザー５出版局，1998.7）

5) Rajikumar Buyya,「High Performance Cluster Computing」,（Prentice Hall PTR，1999）

6) 「MPI Standard（日本語訳ドラフト）,（MPI-J ML，1996.5）

7) 「MPI-2：Extensions to the Message-Passing Interface（日本語訳ドラフト）」,（MPI-J ML, 1999.7）

8) 青山幸也,「虎の巻シリーズ」,（日本アイ・ビー・エム株式会社，1998）

9) Blaise M.Barney,「The Message Passing Interface，ISHPC 97 Tutorial」,（Maui High Performance Computing Center，1997.11）

10) G.A.Geist，J.A.Kohl，P.M.Papadopoulos,「PVM and MPI：a Comparison of Features」,（1996.3）