MPIによる並列分散処理

商学研究科 丸田 寛之
社会情報学科 加地 太一

1 はじめに

近年，コンピュータの性能は驚くほど向上し一昔前では不可能だった計算も最近では短い時間で計算可能となった．とくに，物理シミュレーションやオペレーションズリサーチ等の分野はコンピュータの技術革新とともに成長してきたとも言える．

しかしながら，現在まではインテル社の Gordon Moore が唱えた有名な「ムーアの法則」の通り，プロセッサの速度は 18ヶ月ごとに倍増しているが，2005 年以降はその傾向を維持できないだろうという予測もある．これはその傾向を維持するために乗り越えなければならない障害には非常に深刻なものがいくつかあるからである．[1]

このような性能が頭打ちになる状況において，大規模な問題を処理する場合に利用されるのが並列計算機である．最近ではスーパーコンピュータでも汎用のプロセッサを複数搭載することで計算性能を確保する傾向にあり，複数のプロセッサを用いた並列計算は現代では計算性能向上の常套手段となっている．

しかしながら，一般的な並列計算機は数千万以上するもので非常に高価であるため，一個人や研究室単位での購入は難しい．そこで，最近では安価に並列計算を行うための環境としてワークステーションクラスタが多く利用されている．ワークステーションクラスタは一般に販売されているワークステーションやパーソナルコンピュータを何らかのネットワークで接続し，それぞれのコンピュータ上でプログラムを協調して動作させることで並列処理を行うための環境である．最近ではパーソナルコンピュータの価格が下落し，さらに Linux や FreeBSD といった無料の UNIX 互換 OS も出現したため，非常に低コストでワークステーションクラスタを構築可能である．実際に Beowulf と呼ばれるクラスタが世界中で数多く稼動しており，Los Alamos National Laboratory Center で稼動している Avalon と呼ばれるものでは 533MHz の Alpha プロセッサを搭載したコンピュータを 140 台接続し，parallel Linpack benchmark で 47.7Gflops という性能を確保した．Avalon は 70 台の構成で 195MHz/64 プロセッサの SGI Origin2000 とほぼ同じ性能であるが，これは 1998 年で約 100 万ドルもする一方 Avalon は 140 台のシステム全部で 31 万 3 千ドルで完成した．このシステムでは Linux 上で今回取り上げる MPICH がインストールされ，MPI 環境が構築されている．[2]

ワークステーションクラスタの特徴は低コストであるだけでなく，そのスケーラビリティである．性能が足りなくなればワークステーションを追加することで容易に性能を向上させることが可能である．また，使用するワークステーションは同一のものである必要はなく，様々な性能のワークステーションを混在させることも可能である．

以上のように，並列計算はコンピュータシステムの計算性能を向上させ，ワークステーションクラスタは低コストで並列計算を行うためのものである．今回はこれらを並列処理の概要とともに紹介する．

2　メモリモデル

並列処理にはプロセッサとメモリの接続などによって，いくつかのアーキテクチャが考えられている．

共有メモリは専用の並列計算機や，SMP(Symmetric Multiprocessor) で用いられるアーキテクチャであり，いくつかのプロセッサエレメント (処理要素) が同じメモリをアクセスする形態で，メモリを通して各プロセッサ間の通信を行うことができる．メモリは比較的高速にアクセスでき，強結合であるといえる．

一方，分散メモリは各プロセッサエレメントは独自のメモリを持ち，何らかの通信路を用いてプロセッサ間の通信を行うもので，ワークステーションクラスタなどに用いられる．一般的には通信路の速度は遅く，弱結合である．この分散メモリにおけるプロセッサ間のメッセージ通信を Message Passing と呼ぶ．

3　並列マシンモデル

並列処理は大きく 2 つに分類することができる．

- 並列システム (parallel system)
- 分散システム (distributed system)

並列システムは全てのプロセッサが協調しながら一つの目的のために協調して働く．また，プロセッサ同士が強く結合されており，高速にデータ交換を行うことができる．この場合，全てのプロセッサはほぼ同一となり，均一システムと呼ばれるものとなる．

それに対して分散システムは，必ずしも同一ではないプロセッサが資源を共有しながら弱く結合された状態で働く．各プロセッサは離れた場所にあってもよく，トランザクションの実行に際して，少量のデータのみを交換する．また，プロセッサ同士は互いにデータ交換を行うための通信プロトコルを持っていればよく，同じプロセッサである必要はない．[3]

2 つのシステムの最も大きな違いは協調するかしないかということと，結合度の強さである．逐次計算においては計算のみを考えれば十分であったが，並列計算では通信も考える必要がある．並列計算ではプロセッサ間のデータ転送の方法も様々であり，これが並列計算の最も難しい側面といえる．[3]

並列システムにはプロセッサとメモリの結合のしかたに多くの方式があり，どのような結合で，どのような通信が行われるかによって様々なモデルが定義されている．並列処理システムは以下のようなマシンモデルとして分類することができる．

- SISD (Single Instruction stream, Single Data stream)
  同時に 1 つのインストラクションと 1 つのデータセットを処理する．従来の単一の

パソコンやワークステーションなど．このモデルでは並列処理は行わない．

- SIMD (Single Instruction stream, Multiple Data stream)
  全てのプロセッサが同じインストラクションを実行し，各プロセッサはそれ自身のデータをオペレートする．データ (ベクトルや配列) の一部が規則的に他のプロセッサに送られる．このモデルは配列の全エレメントで同じオペレーションを実行するようなコンセプトに適応するもので，パイプラインコンピュータやコネクションマシンなどで用いられるモデルである．

- MISD (Multiple Instruction stream, Single Data stream)
  各プロセッサが 1 つのデータセットを同時に処理するもので，実用化はされていない．各プロセッサは独自の制御フローをもち，すべての同じデータが与えられる．プログラムは同じ対象データに対してほぼ独立の処理を行うモデルである．

- MIMD (Multiple Instruction stream, Multiple Data stream)
  各プロセッサが独立して動作する．これは SIMD よりもフレキシブルなモデルであり，ワークステーションクラスタなどが該当する．このモデルは同期によってさらに以下のように分類される．

  - Synchronous MIMD
    各プロセッサはいずれも同期され，連続したインストラクションを同時に実行する．
  - Asynchronous MIMD
    各プロセッサはすべて独立してインストラクションを実行する．

- SPMD (Single Program, Multiple data stream)
  全てのプロセッサが同じプログラムを実行する．SIMD と異なるところは，各プロセッサは異なるフローパスを通ることが許される点である．MIMD 上で同じプログラムが独立して動いているものと考えられる．

今回のモデルは各ワークステーションが完全に独立しており，それぞれが独立してインストラクションを実行するものであり，Asynchronous MIMD に属するものである．

4 分割手法

逐次処理から並列処理へ計算処理を分割する場合，どのように分割するかによって以下のように分類される．

- データ分割手法
  扱うデータ量が膨大で，かつデータに対する処理が単純な場合，データを小さな単位に分割して各プロセッサに割り当てる．

- 機能分割手法
  扱うデータ量がそれほど多くなく，データに対する処理が複雑な場合，処理そのものををいくつかに分割してプロセッサに割り当てる．

データを分割するか，また機能を分割するかは並列マシンのアーキテクチャや問題に依存するものである．機能分割の場合，複雑な処理を複数のプロセッサで協調させて処理しなければならない．しかし，一般にプロセッサ同士を協調させるためには高速な通信路が必要となり，強結合である必要がある．ただし，分割した処理が十分に大きく，それぞれが独立して処理できる場合，つまりそれぞれの処理に依存関係が薄いか存在しない場合は弱結合でも良いことになる．一方，データ分割においては各データを独立したプロセッサで処理し，結果だけを交換すれば良い場合が多い．この場合，通信路は低速でも良く，弱結合に向く．しかしながら，分割されたデータ量が十分に小さく，データ処理の前後に依存関係が存在する場合はやはり頻繁に通信が必要となり，強結合である必要がある．

どちらにせよ，一つの問題を複数のプロセッサで処理する場合はプロセッサ間のデータ通信が必要となる．この通信の頻度を粒度と呼ぶ．頻繁に通信を行う粒度の細かいアルゴリズムでも高速な通信路を持つ強結合のシステムにおいては十分に実用的であるが，低速な通信路しか持たない弱結合のシステムにおいてはいかに粒度を粗くし，通信の頻度を下げるかが効率的なアルゴリズムのポイントとなる．また，一般的にネットワーク上の通信においては小さなデータを数多く頻繁に送受信するよりも，ある程度まとまったデータを一度に送受信するほうが通信パフォーマンスが向上することも効率的なアルゴリズム構成における重要な点である．

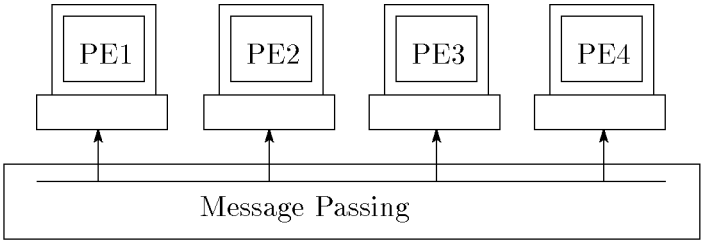
5 MPI

以上の並列処理を実現する手段として，MPI(Message Passing Interface) ライブラリがある．MPI はメッセージ通信のためのライブラリの標準であり, 様々な実装が存在し, 多くのプラットフォームで動作する．MPI の標準化は並列コンピュータの主要ベンダ，大学，国立研究所，企業などからの研究者が加わり，1993 年に MPI-1 と呼ばれる標準化が完成した．[4]

MPI はメッセージ通信を行うためのライブラリ標準であるが，その利点はポータビリティと使いやすさにある．MPI は分散メモリのマルチプロセッサ，ワークステーションのネットワーク，またこれらの組み合わせの環境でも動作するように設計され，さらに共有メモリでも実現可能である．

また，MPI は標準 (Standard) であり，実際にそれを実装したライブラリがいくつか存在する．例えば，LAM(Local Area Multicomputer) は MPI-1 の完全な実装の一つである．LAM はネットワーク上に混在するコンピュータ上で MPI 環境を提供するものである．このライブラリはネットワーク接続されたコンピュータに特化したライブラリで，ポータビリティは多少劣るものの，特定環境下でパフォーマンスを発揮するように設計されている．また，MPICH は ANL/MSU による実装のひとつであり，様々な機種で動作するように設計されている．MPICH は Abstract Device Interface というアーキテクチャを採用しており，ネットワーク上のコンピュータであれば TCP/IP を使うといったような，メッセージ通信のインターフェイスデバイスを交換することで，様々な並列アーキテクチャに対応する．

図.1 は MPI によるワークステーションクラスタの例である．メッセージの通信路には 100BASE-TX Ethernet を用い，各 PE(Processor Element) 間でネットワークを介したメッ



100BASE-TX Switching Hub

図 1: MPI Workstation Cluster

セージ通信を行うことで，PE 間の同期やデータ交換を行う．

MPI では C 言語や Fortran の関数として標準化されている．その中にはデータの送受信等の 1 対 1 通信や 1 対多通信であるブロードキャスト，また PE 間の同期をとるためのものや，システム全体からプロセスグループとよばれる PE 群を生成したりするものなどが用意されている．これらの関数は MPI 標準として標準化されたものであり，ソースレベルでは各種ライブラリ間でコンパチビリティが保証されている．つまり，ワークステーションクラスタ上で作成したアプリケーションは高価な並列コンピュータ上でも同様に動作させることができる．

6 MPIによるプログラミング

MPI の実装は MPI ライブラリの関数を用いて実装する．例えば，図.2 は 1〜100 までの総和を求める逐次アルゴリズムである．これを 2 プロセッサ上での並列アルゴリズムにすることを考えてみる．単純にデータ分割的手法で 1〜50 と 51〜100 までの総和をそれぞれのプロセッサで求めて最後に各プロセッサで求められた値を加算すれば良いことになる．それを実装したものが，図.3 である．

MPI では，MPI 関数を用いることでメッセージ通信等を実現する．まず，MPI を利用するために mpi.h ヘッダファイルをインクルードする．このファイルでは，MPI に必要なデータ型の typedef や構造体宣言等が書かれている．

次に，MPI 関数を利用する前に MPI を初期化する必要がある．これは，MPI_Init 関数によって行う．MPI_Init 関数は，これ以外の全ての MPI 関数を利用する前に呼ぶ必要がある．引数には main() 関数の引数である，コマンドライン引数の個数と文字列を受け渡す必要がある．これは，MPI 実行時に mpirun コマンドから各種パラメータを MPI ライブラリに受け渡す必要があるためである．

ソースでは，MPI_Comm_rank 関数を用いて自分のランクを取得している．ランクとは MPI 実行時に動的に各プロセスに割り当てられる動的な識別子である．mpich では基本的に実行開始順に 0 から割り当てられる．

次に出てくる，for ループは実際にデータを分割している部分である．$nRank$ は自分のランクであり，2 プロセスなので 0 か 1 のいずれかがセットされている．ランク 0 であれば，i=0 から 49 までのループであり，ランク 1 であれば，i=50 から 99 のループである．2 つあわせて 0 から 99 までのループとなる．

最後に，結果を通信する．結果はランク 1 のプロセスが得た答えをランク 0 に送信することで，ランク 0 上で最終的な合計値が得られる．

もし，自分がランク 0 であれば，ランク 1 からデータを受け取ることになるので，MPI_Recv 関数でデータを nGsum に受信する．また，自分がランク 1 であれば，ランク 0 へ結果を送信しなければならない．MPI_Send 関数でデータ nSum を送信する．

送受信は，送信関数と受信関数を双方で呼び出す必要がある．以上の関数の場合，もし一方のプロセスだけが送受信関数を呼び出した場合，相手のプロセスが送受信状態になるまで送受信関数内でブロックされる．

MPI に関する全ての処理が完了したら，最後に MPI_Finalize 関数で MPI の後処理を行う．

ただし，線形に計算時間が短縮されないことが重要である．上記の並列化の場合，データ分割は確かに 2 分割であり，この部分においては線形に 2 倍になっているが，この 2 つのデータを通信を行って統合する必要がある．この通信というのは逐次処理には存在しなかった部分であり，並列化に伴って新たに増加した処理である．データ分割でも機能分割でも，処理を分割する以上，各プロセッサ間で通信を行い同期や統合を行う必要が生じる．このときの通信時間は並列処理特有のものであり，これが時間短縮の妨げとなる．とくにワークステーションクラスタ型の並列計算環境においては，通信路のバンド幅が比較的細いため，通信時間の削減が効率的な並列化のポイントとなる．

一般的に通信を行う場合，少量のデータを数多く通信するよりも，ある程度のデータをまとめて通信したほうがパフォーマンスは向上する．つまり，並列化においては各プロセスをできるだけ独立させることによって，通信間隔をできるだけ長くすることで粒度を粗くし，パフォーマンスを向上させることが求められる．

```
#include <stdio.h>
main(){
  int i;
  int nSum=0;  /* sum */
  for(i=1; i <= 100; i++){
    nSum += i;
  }
}
```

図 2: Sequencial Algorithm

7 並列計算環境

今回構築した環境は，いくつかの独立したコンピュータをネットワークに接続し，それをプロセッサ間の通信路として利用するもので，ワークステーションクラスタの一形態である．(図.6)

また，今回採用したネットワークは 100BASE-TX Ethernet であり，各コンピュータ

```
#include <stdio.h>
#include <mpi.h>
main(int argc, char* argv[]){
  int i;
  int nSum=0;  /* sum */
  int nGsum=0; /* grand sum */
  int nRank;  /* my rank */
  MPI_Status mpistatus;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &nRank); /* Get my rank */
  for(i= nRank*50 ; i <= (nRank*50+49); i++){
    nSum += i+1;
  }
  if (nRank == 0){
    /* Receive from Rank 1 processor */
    MPI_Recv(&nGsum, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &mpistatus);
    nGsum += nSum;
    printf("sum = %d \n",nGsum);
  }
  else {
    /* Send to Rank 0 processor */
    MPI_Send(&nSum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
  }
  MPI_Finalize();
}
```

図 3: Parallel Algorithm with MPI

を FastEthernet Switching HUB (図.5) へ接続した．100BASE-TX Ethernet は従来の 10BASE-T を拡張したもので，Network Interface Card や HUB も数多く出され，様々なプラットフォームで動作可能であり，コストパフォーマンスも高い．Switching HUB は従来の Repeater HUB(Dumb HUB) と比較して通信パフォーマンスが高い．従来の Repeater HUB は HUB 上に Ethernet バスを構成したもので，あるノードが送受信中のパケットは全てのノードに送信され，バスが占有されることになる．これによってバス上でパケットの衝突が起こりやすくなり，スループットが低下する．Switching HUB は各ノード間をスイッチで結ぶ．ノード間の通信はスイッチにより独立したチャンネルとして構成される．このため，送受信パケットは特定のノードのみに送られ，それ以外のノードにはバスが開放されている．このため，他のノードが通信中であっても送受信可能となる．

コンピュータは Intel CeleronProcessor/400MHz を搭載した同性能のコンピュータを 4 台使用した．この 4 台は 1 組のモニタとキーボードをコンピュータスイッチ (図.4) を用い

図 4: Computer Switch

図 5: Fast Ethernet Switching Hub

て共有した．Operating System には Linux を採用した．Linux は UNIX 互換 OS の一つであり，基本的には Linux カーネルと GNU ソフトウェアのセットを各種ディストリビューションとして提供されている．

この環境において MPI の通信パフォーマンステストを行ったところ，1024 バイトブロックの実効転送速度は 1.3Mbps 程度であった．また，ブロックサイズを大きくしたピーク性能でも 23Mbps 程度であった．この通信路のバンド幅の狭さは弱結合であるワークステーションクラスタの欠点であり，専用の並列計算機等と比べて大きく劣る点である．

8 MPI 環境の構成

MPI 環境は http://www.mcs.anl.gov/mpi/mpich からダウンロードした mpich を利用した．このファイルを

```
tar -zxvf mpich.tar.gz
```

または

```
gunzip -c mpich.tar.gz | tar -xvf -
```

で展開した後，作成された mpich ディレクトリにおいて，`./configure` スクリプトを実行する．これによって，スクリプトがシステム環境を調査した後，自動的に Makefile が生成される．以後，コンパイルは `make` だけで完了し，次に root ユーザで

```
make -PREFIX=/usr/local/mpich install
```

図 6: Parallel Processing Environment

すれば完了である.

なお, `/usr/local/mpich` はインストール先ディレクトリを示す.

各端末へのインストールが完了した後に, 環境設定を行う. mpich は r-command (rsh など) を利用して他の端末のプロセスを起動するため, これらの r-command を適切に実行できる環境でなければならない. 各端末の`/etc/hosts.equiv` ファイルに r-command の実行を許可するリモートホスト名を自分自身も含めて全て記述する. 例えば,

```
node1.otaru-uc.ac.jp
node2.otaru-uc.ac.jp
```

などのように MPI で利用する全端末を記述する.

次に, MPI で利用するホストの設定を行う. インストール先ディレクトリが `/usr/local/mpich` であれば
`/usr/local/mpich/util/machines/` または`/usr/local/mpich/share/`
ディレクトリに設定ファイルが存在する.

このディレクトリの machines.*arch* ファイルを編集する. *arch* はアーキテクチャ名 (Linux なら machines.LINUX) が入り, インストール時に生成される. このファイルに

```
node1.otaru-uc.ac.jp
node2.otaru-uc.ac.jp
```

のようにクラスタに含めるホスト名を記述する. 自分自身のホストも含まれる. mpich ではプロセス数に従って, この記述の順番でプロセスを起動していく. 上の例では, node1 がランク 0, node2 がランク 1 である.

さらに, インストールされた mpich のコンパイラ等の実行ファイル群は標準で `/usr/local/mpich/bin` にインストールされるので, ここに path を通しておく. MPI プログラムをコンパイルする場合, C 言語であれば mpicc を, C++であれば mipCC をコンパイラとして使用する. これらのコンパイラはシステムにインストールされているネイ

ティブなコンパイラ (cc や gcc など) を適切に呼び出すための wrapper であり，コンパイル時に MPI に必要なライブラリ等が設定される．また，mpicc/mpiCC のオプションはネイティブなコンパイラに受け渡されるため，同じオプション (例えば -O 等) を使用できる．

また，4 台のうちの 1 台を NFS サーバとして稼動させた．これは，MPI 実行時に同じパスに同じ実行オブジェクトが存在しなければならず，また同じデータファイルを扱うためである．NFS サーバは NFS サービス (NFS デーモン等) が開始されていれば，exports ファイルに公開するディレクトリを接続を許可する相手を記述するだけである．NFS サービスは，rpc.nfsd(nfsd) rpc.mountd 等を指す．NFS サーバでは，公開する /home ディレクトリを /etc/exports ファイルに設定した．

```
/home node*.otaru-uc.ac.jp (rw,no_root_squash)
```

この設定は，指定したホストに対して /home ディレクトリを公開するためのものである．
`node*` は node から始まるホスト名のワイルドカードであり，
`node1, node2, node3` などが当てはまる．

これに他の 3 台を NFS クライアントとして NFS サーバをマウントした．各端末にはすでに /home ディレクトリが存在するため，新たに作成した /ehome ディレクトリに以下のコマンドでマウントした．

```
mount -t nodeX.otaru-uc.ac.jp:/home /ehome
```

また，NFS サーバにおいては，

```
ln -s /home /ehome
```

でシンボリックリンクを作成することで，全ての端末上で/ehome 以下の同じパスに同じファイルが存在することを実現した．

9 MPICH の利用

MPICH を用いて並列処理を行う場合，MPI 関数を使うソースファイルに mpi.h ヘッダファイルをインクルードしなければならない．

また，コンパイルは C 言語であれば mpicc を，C++であれば mpiCC を利用する．このコンパイラは子プロセスとしてネイティブなコンパイラ (cc や gcc 等) を呼び出すが，その際に必要なオプションをいくつか指定してくれる．例えば，*para.cc* をコンパイルする場合，

```
mpiCC -O para.cc -o para
```

のように行う．`-O` や`-o` オプションはネイティブなコンパイラへ対するオプションであり，mpicc/mpiCC はこれらのオプションを透過的にネイティブなコンパイラへ受け渡すことができる．

X-Window を用いた MPI(MPE) のプロファイリングを利用する場合は，

```
-llmpi -lmpe -lpmpich -lmpich -L/usr/local/mpich/build/LINUX/ch_p4/lib
```

等のオプションで必要なライブラリを明示的にリンクする必要がある．これらのライブラリはX-Windowで計算状態などをリアルタイムに表示する場合に用いるもので，一般的なMPEプロファイルの場合は必要ないと思われる．プロファイリングが必要な場合，gprof/prof等の一般的なプロファイラを用いる方法とMPEを用いる方法がある．gprofやprofの場合，コンパイル時にmpicc/mpiCCに対して

```
mpiCC -pg -O para.cc -o para
```

のように`-pg`または`-p`オプションを渡すことで，ネイティブなコンパイラがプロファイルライブラリを自動的にリンクする．実行後にgmon.outまたはmon.outファイルが生成されるので，gprofはまたprofでプロファイルを取ることが可能となる．また，これらのプロファイルは各プロセス毎に生成されることになる．ここで得られる時間は各プロセス単体での計算時間であり，各プロセスが行う通信処理などの時間も含まれる．gprof/profを用いる場合はソース上では特に特別な処理を行う必要はないが，MPEを用いるプロファイルの場合はソースに関数を埋め込まなければならない．MPEではプロファイルを取りたい部分の前後に関数を埋め込むことで，関数に囲まれた部分だけの計算時間を得ることができる．通信処理部分を囲むことで通信処理時間のみを取得することもできる．また，この結果はclog形式等によって保存され，専用のViewerでこれを参照することで，グラフ化することもできる．

実行は`mpirun`を利用する．オプションに起動するプロセス数，実行ファイル名を指定する．例えば，4プロセスで*para*というプログラムを起動する場合は，

```
mpirun -np 4 para
```

等のように起動する．

また，プログラム中でprintf関数等を用いて標準出力に何らかの出力を行った場合，MPICHでは起動した端末に全て表示されるようになっている．例えば，あるプログラムの中に`printf("check point\n");`のような呼び出しをした場合，MPICHではプロセス分の表示を行う．4プロセスで起動していた場合，このメッセージが4回表示されることになる．このようなメッセージの場合，

```
printf("Rank %d: check point\n",myrank);
```

のようにランクと共に表示しなければどのランクのメッセージか識別できないことに注意しなければならない．また，表示されるタイミングもprintf関数が実行された時点ではなく，バッファがフラッシュされたときにまとめて表示されるため，他のプロセッサと必ずしも同じタイミングで表示されないことに注意する．

10 MPI関数リファレンス

ここでよく利用されると思われるMPI関数群を紹介する．MPIは一般的にC，C++，Fortran等の言語処理系で動作するが，ここではC言語に関する解説を行う．なお，文献[4]に詳細が記述してある．

C言語のコンパイラは通常，mpiccを利用する．また，プログラム本体がクラス等を用いたC++の場合はmpiCCを利用する．このとき，C言語のインターフェイスをそのまま

利用することが出来る．また，C++用のクラス化されたMPIインターフェイスも利用することが出来るが，それについてはここでは述べない．

10.1 初期化と終了

MPIではMPI関数を利用する前の準備と利用した後の後処理のために，
`MPI_Init` 及び `MPI_Finalize`
を呼び出す必要がある．

```
MPI_Init(&argc, &argv);
```

MPIを初期化する．他の全てのMPI関数を使用する前に呼び出す必要がある．この引数はプログラム本体の `main(int argc, char* argv[])` といったmain()関数の引数である．

```
MPI_Finalize()
```

MPIに関する処理を完了する．この関数はMPIの後処理を行うもので，MPIライブラリが使用したリソースの解放などを行う．MPI関数はMPI_Init関数とこの関数の中で実行しなければならない．

10.2 コミュニケータ

MPIでは「コミュニケータ」を用いて通信を行う．コミュニケータはグループの識別子と言える．

MPIでは,「プロセスグループ」を生成することができる．例えば，16プロセスのシステムを仮定する．このシステム上で8プロセスは処理Aを，残り8プロセスは処理Bを個別に行いたい場合，2つのプロセスグループに分割することができる．このとき，新しいコミュニケータが生成され，以後そのコミュニケータで通信を行うとグループ内に対する通信となる．ただし，MPIではまったく新規にグループを生成する関数はなく，既存のグループから子グループを生成する．MPIでは初期化時にMPI_COMM_WORLDというグループを生成する．このグループは全プロセス，上記の例であれば16プロセス全てを含むグループである．全てのグループはこの親グループの子グループとして生成される．上記の例であれば，MPI_COMM_WORLDを2つに分割し仮にGROUP_AとGROUP_Bとする．もし，処理Aをさらに処理A1と処理A2として独立させたい場合，GROUP_Aの子グループとしてGROUP_A1とGROUP_A2を生成することになる．以上のようにプロセスをグループ単位に分割して処理を行い，グローバルな処理はグループ間通信を用いて行うことができる．

図.7 はその一例である．この例では4プロセスからなるMPI環境であり，Rank 1から4までのプロセスが起動している．起動時はMPIによって4プロセス全てを含むMPI_COMM_WORLDというグループが生成されている．このMPI_COMM_WORLDから，Rank 1,2をGROUP_A，Rank 3,4をGROUP_Bとして分割でき，各プロセスグループはそれぞれ独立した環境として独立した処理や通信を行うことができる．各通信関数に

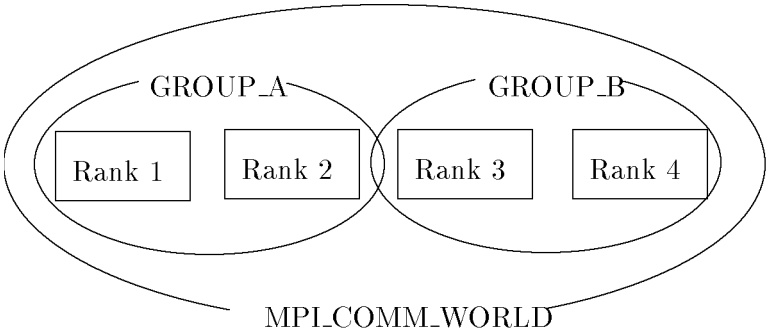


図 7: コミュニケータ

は，どのプロセスグループに関する通信であるかを示すためにコミュニケータを引数として受け渡す必要がある．

今回はプロセスグループの生成に関しては解説していない．全ての通信はMPI_ COMM_ WORLD に対するもの，つまり MPI 環境下の全プロセスに対して行われるものとする．

10.3 情報の取得

MPI 環境下において実行されるプロセスの情報を得るための関数群である．

```
MPI_Comm_rank(MPI_COMM_WORLD, int* nRank);
```

現在の自分のランクを $nRank$ に得る．ランクは MPI 起動時に動的にプロセスに割り当てられるもので，プロセスを識別するためのものである．MPICH の場合，0 から順に割り当てられるようである．MPI_COMM_WORLD は，コミュニケータである．

```
MPI_Comm_size(MPI_COMM_WORLD, int* nPnum);
```

MPI 環境で実行中の総プロセス数を $nPnum$ に得る．一般的には mpirun コマンドで指定したプロセス数が得られる．MPI_COMM_WORLD は，コミュニケータである．

```
MPI_Get_processor_name(char* chProcName, int* nNameLen);
```

自分のプロセッサ名を $chProcName$ 配列に得る．$nNameLen$ は実際の文字列の長さが選られる．このプロセッサ名は，TCP/IP ネットワークにおける MPICH では FQDN なホスト名 (xxx.otaru-uc.ac.jp 等) が得られる．ランクと共にプロセッサ名をログに記録しておくとログを見分けやすくなる．

10.4 1 対 1 通信

特定のプロセス間で行う 1 対 1 通信関数群である．これらの送信関数のそれぞれには，いくつかの通信モードが存在する．

バッファモード バッファモード送信関数は，送信先で受信関数が実行されているかどうかに関わらず実行することができる．送信メッセージは相手に受信されるまでバッファリングされる．この関数は相手が実際に受信する前に終了する場合があることに注意しなければならない．

同期モード 同期モードは，送信先で受信関数が実行されているかどうかに関わらず実行することができる．ただし，この関数は相手に受信されるまで終了しない．つまり，この関数が終了するときは相手が実際に受信したときとなる．

レディモード レディモードは，送信先で受信関数が実行されているときのみ実行することができる．それ以外の場合はエラーとなり結果は保証されない．この関数を実行できるということは相手が実際に受信されることを意味する．

標準モード 標準モードでは，送信先で受信関数が実行されているかどうかに関わらず実行することができる．バッファリングされるかどうかはMPIに任せられる．もしバッファリングされた場合，相手が受信する前に終了する場合がある．

```
MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,
         int tag, MPI_COMM_WORLD, MPI_Status* status);
```

データを*buf*に標準モードで受信する．*count*には受信データ数を設定する．これは対となるMPI_Send関数の送信データ数と同じ数にする必要がある．単一の変数であれば1を，配列を受信するのであればその数を設定する．*datatype*は変数型である．ここにはMPIで定義されている変数型を指定する．int型であればMPI_INTを，float型(浮動小数点型)であればMPI_FLOATを，double型(倍精度実数)であればMPI_DOUBLEを指定する．これ以外にもいくつかのデータ型が存在する．*source*には送信元ランクを指定する．このランクからのデータ以外は受信しない．ただし，ここにMPI_ANY_SOURCEを指定ことで，どのランクからのデータでも受信することができる．*tag*はデータの識別子である．これは送信側の*tag*と同一のものを指定する．このTagでデータの内容などを識別することがでいる．一致しないTagのデータは受信されない．ただし，ここにMPI_ANY_TAGを指定することで，どのタグのデータでも受信することができる．MPI_COMM_WORLDは，コミュニケータである．*status*は受信状態が返される．ここにはMPI_Status構造体の変数アドレスを指定する．エラーや，MPI_ANY_SOUCEまたはMPI_ANY_TAGを使用したときに，実際にどのランクから，またどのタグのデータを受信したのかを知ることができるように，受信完了時にこの構造体に値がセットされる．

この関数はブロック型関数であり，指定したデータが受信されるまで関数内でブロックされ，処理が中断する．つまり，相手のMPI_Sendと同じタイミングでこの関数を実行しなければ無駄な待ち時間が生じることになる．ただし，結果的にここで相手ランクとの同期を取ることができる．

この関数を用いたサンプルを図.10.4に示す．このサンプルは，ランク1のsendbufの値をランク0のrecvbufへ送信する例である．

```
MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest,
         int tag, MPI_COMM_WORLD);
```

```
MPI_Status mpistatus;
int recvbuf;
int sendbuf = 100;

if (nRank == 0){
  MPI_Recv(&recvbuf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
                                              &mpistatus);
 }
 else{
  MPI_Send(&sendbuf, 1, MPI_INT, 0, 0, MIP_COMM_WORLD);
 }
```

図 8: MPI Send/Recv

データ *buf* を送信する．*count*, *datatype* は MPI_Recv と同一である．*dest* には送信先ランクを受信する．このほかのデータは MPI_Recv と同一である．この関数もブロック型であり，相手にデータが受信されるまでは関数内でブロックされることになる．

この関数を用いたサンプルを図.10.4 に示す．このサンプルは，ランク 1 の sendbuf の値をランク 0 の recvbuf へ送信する例である．

```
MPI_Ssend(void* buf, int count, MPI_Datatype datatype, int dest,
         int tag, MPI_COMM_WORLD);
```

データ *buf* を同期モードで送信する．それ以外の引数は MPI_Send と同一である．同期モードの送信は，送信対象のプロセスが受信を開始するまで関数内でブロックされる．この関数が終了するということは送信対象のプロセスが受信を開始したことを意味し，同時にこの関数で両プロセスの同期が得られることを意味する．

```
MPI_Rsend(void* buf, int count, MPI_Datatype datatype, int dest,
         int tag, MPI_COMM_WORLD);
```

データ *buf* をレディモードで送信する．それ以外の引数は MPI_Send と同一である．レディモード送信は送信対象ランクが既に受信操作を開始していなければ実行できない関数である．もし受信操作が開始されていなければこの関数は送信操作を起動せずに終了し，その結果は保証されない．

```
MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,
         int tag, MPI_COMM_WORLD, MPI_Request* request);
```

この関数は MPI_Recv のノンブロッキング関数である．MPI_Recv はブロッキング関数であり，その処理が完了するまで関数から戻らない．しかし,「I」という immediate という意味のプリフィクスがつくこの関数はノンブロッキング関数として処理され，その処理が完

了する前に関数から戻る．パラメータの最後にある *request* は要求ハンドルと呼ばれ，現在処理中の要求の識別を示す．この *request* を MPI_Wait や MPI_Test 関数に受け渡すことで，処理が完了したかどうかを知ることができると同時に，完了していればそのステータスを受け取ることができる．

ノンブロッキング関数は処理が完了する前に関数から戻ってくるため，送受信待ちの間なども別の処理を行うことができ，効率的に資源を利用することができることになる．

```
MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest,
        int tag, MPI_COMM_WORLD, MPI_Request* request);
```

この関数は MPI_Send のノンブロッキング関数である．MPI_Irecv と同様に *request* を引数に必要とする以外は MPI_Send と同様である．

```
MPI_Wait(MPI_Request* request, MPI_Status* status);
```

この関数はノンブロッキング関数の完了を待つ．現在処理中のノンブロッキング関数の送受信処理が完了しているかどうかを調査し，完了していなければこの関数内でブロックされ，完了していればステータスをセットしてこの関数から戻る．*request* には完了を待ちたい要求ハンドルを受け渡す．

```
MPI_Test(MPI_Request* request, int* flag, MPI_Status* status);
```

この関数はノンブロッキング関数が完了しているかどうかを調査する．現在処理中のノンブロッキング関数の送受信処理が完了しているかどうかを調査し，結果を *flag* に返す．要求が完了していてもしていなくてもこの関数からすぐに戻ってくる．完了していれば *flag* には true が返り，ステータスが同時にセットされこの要求は解放される．それ以外は false が返る．

この関数はノンブロッキング関数の要求を発行した後，処理が完了したかどうかをポーリングしたりする場合に利用される．MPI_Wait は完了するまでブロックされるが，この関数は完了していなくても別の処理を続行することができる．

図.9 はこれらの非同期通信関数の一例である．Rank 2 は MPI_Irecv で受信操作を起動するが，まだ受信はしない．非同期通信関数であるため，起動後すぐに関数から戻る．この間，別の処理を行い，定期的に MPI_Test 関数で受信操作が完了したかどうかをポーリングする．完了していなければ (MPI_Test 関数の flag が false であれば) また別の処理を行う．再度ポーリングを行い，受信が完了していれば受信したデータを適切に処理する．このとき，受信待ちデータが到着するまでこれ以上処理すべきことがなければ MPI_Wait 関数でブロックできる．

このように非同期通信関数は受信待ちの本来ブロックされる時間に別の処理を行うことで効率的な処理を行うことが可能である．

10.5 集団通信

これまで記述した送受信関数は「1 対 1 通信」と呼ばれる種類のものである．つまり，特定のプロセス間でのみメッセージ通信が行われるものである．しかし 4 プロセスや 16 プ

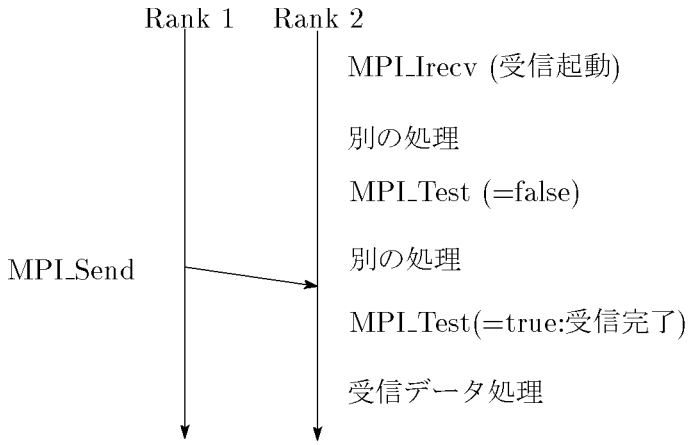


図 9: 非同期通信フロー

ロセスなどの多プロセスにおいては 1 対多の通信を行う必要がある場合がある．この場合，MPI では「集団通信」と呼ばれる関数が用意されている．

集団通信はグループの全プロセスが同一の引数を持って通信ルーチンを呼び出すことで実行される．また，あるプロセスが他のすべてのプロセスにメッセージを送信するなど，1 対多通信や多対 1 通信の場合，全プロセスへメッセージを発信するプロセス，または全プロセスからメッセージを受信するプロセスを「ルート」と呼ぶ．

```
MPI_Barrier(MPI_COMM_WORLD);
```

この関数は指定コミュニケータに属するプロセスの全てが MPI_Barrier 関数を実行するまで，この関数内でブロックするものである．つまり，処理が先に終わったプロセスは最後に処理を終えたプロセスがこの関数を実行するまで関数内でブロックされる．この関数はプロセス間で同期を取るときなどに利用する．

```
MPI_Bcast(void* buffer, int count, MPI_Datatype datatype,
          int root, MPI_COMM_WORLD);
```

ブロードキャスト関数である．この関数はルートであるプロセスから他のすべてのプロセスへメッセージを送信し，この関数から戻った時点でルートのバッファが全プロセスのバッファへコピーされている．この関数は全てのプロセスで実行しなければならない．

```
MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, int recvcount, MPI_Datatype recvtype,
              MPI_COMM_WORLD);
```

この関数は全プロセス間でデータの交換を行う関数である．図.10 のように，各プロセスの sendbuf のデータが全プロセスの recvbuf にコピーされる．このとき，recvbuf の大きさは sendcount のサイズのプロセス倍であることに注意する．ただし，関数に指定する recvcount は 1 エレメントのサイズであるため，sendcount のサイズと同一であるが，受

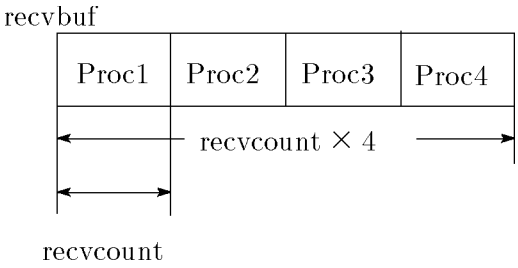


図 10: Allgathger

信時にプロセス数だけのエレメントを配列にランク順にコピーする．コピーは自分自身も含まれることに注意する．つまり，最終的には全てのランクは同じ構造の recvbuf が得られることになる．

この関数を用いたサンプルを図.10.5 に示す．このサンプルでは各プロセスが相互にデータを交換する．

```
int i;
int rank;
int pnum;
int* recvbuf;
int sendbuf;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPIP_COMM_WORLD, &pnum);
recvbuf = new int[pnum+1];
sendbuf = rank*100;

MPI_Allgather(&sendbuf, 1, MPI_INT, &recvbuf, 1, MPI_INT,
                                                MPI_COMM_WORLD);

for(i=0; i<pnum; i++)
 printf("Rank %d received: %d from Rank %d\n",rank,recvbuf[i],i);

delete recvbuf;
```

図 11: MPI Allgather

以上の関数でメッセージ通信を行う場合，デッドロックに注意しなければならない．例えば，全プロセスが MPI_Recv 関数を同時に実行してしまうと各プロセスは MPI_Recv 関数でブロックされプロセスは受信完了まで一時停止状態となる．この状態はデータ受信ま

で継続するが，全プロセスが受信待ちであるため送信プロセスが存在せず，結果的に全プロセスが一時停止し処理全体が停止する．このような状態を避けるため，送受信のタイミングに注意する必要がある．もし途中で各プロセスの同期を得たい場合，MPI_Barrier 関数などを用いて同期を取る必要がある．

10.6 プロファイリング

10.6.1 MPE を用いたプロファイル

MPICH ではプロファイリングのための関数をいくつか用意している．これらの関数群は MPE_ というプリフィクスから始まる．ただし，これらの関数は MPICH インストール時に MPE をインストールしている必要があるが，一般的な方法でインストールした場合はすでにインストールされている．

プロファイル結果はログファイルとして記録される．ログファイルは clog 形式であり，MPICH に付属される jumpshot という Java アプリケーションで参照することができる．

```
MPE_Init_log();
```

MPE を利用する前にこの関数で初期化しなければならない．

```
MPE_Describe_state(x, y, "comment", "color");
```

MPE で記録するログで利用する各種パラメータを設定する．x と y はログの記録に利用するイベント識別子である．x から y までの識別子のイベントは，*comment* のイベントである，ということになる．*comment* にはコメントとして何の処理を行っていたか，等を記述する．*color* はログ参照時に利用する色である．一般的には X-WindowSystem のカラーマップ (red:vlines3 等) を指定する．例えば，

```
MPE_Describe_state(1, 2, "Communicate", "red:vlines3");
MPE_Describe_state(3, 4, "Calculate","blue:gray3");
```

のように記述する．ここではイベント識別子 1 から 2 は，通信に関するイベントであることを指定している．また，3 から 4 は計算に関するイベントであることを指定している．必要なイベントの数だけこの関数を実行する．

```
MPE_Start_log();
```

実際にログの記録を開始する．この関数を実行する前に，MPE_Init_log 関数，及び MPE_Describe_state 関数で初期化と設定を行っておかなければならない．

```
MPE_Log_event(x,z, "comment");
```

イベントを記録する．x は MPE_Describe_state 関数で設定したイベント識別子である．*comment* はイベントのコメントである．

例えば，

```
MPE_Log_event(3, 0, "Start Calculation");
 for(i=0; i<1000; i++){ x += i; }
MPE_Log_event(4, 0, "End Calculation");
```

といったように，プロファイルを取りたい部分の処理をイベントで囲む．

```
MPE_Finish_log("filename");
```

ログ記録を完了する．この関数を実行した時点でログファイルが close される．*filename* にはログファイル名を記録する．実際には，`filename.clog` というように，.clog というサフィックスが付加される．

10.6.2 経過時間を用いたプロファイル

MPI では経過時間を取得することで計算時間を取得することもできる．

MPI_Wtime() 関数はプロセスの経過時間を返す関数である．この関数が返す値は常に変化しており，呼び出した時点での経過時間を返すものである．この関数を利用することで，計算時間等を計測することができる．たとえば，

```
double starttime, endtime;
starttime = MPI_Wtime(); // get start time
 for(i=0; i<10000; i++){ sum += i;} // Calculations
endtime = MPI_Wtime(); // get end time
printf("Calc time = %f\n",endtime-starttime);
```

のように，開始時間と終了時間を取得することでそれに挟まれた部分の経過時間を取得することができる．単位は秒である．なお，MPI_Wtime() 関数の解像度 (resolution) は MPI_Wtick 関数で取得することができる．これは恐らく OS やハードウェア等のプラットフォームで異なると思われる．ちなみに，ix86 上の Linux においては 0.000003 秒であった．

10.7 Example

ここで通信処理の実例を紹介する．

図.12 は隣接する片方のプロセスにデータを送ることで，データを一周させるものである．この場合，まず起点となるプロセスでデータを送信する．次に各プロセスでは受信操作を起動し，データが受信されたら隣のランクへ送信する．これを繰り返すことで巡回させ，最後に起点となるランクがデータを受信したら完了である．

次に示すソースは上のランクへデータを順次送信するものである．また，起点ランクは 0 とし，ランク 1 は必ず存在しなければならない．

```
int nPnum;  // Number of Processor (max. rank+1)
int nRank;  // My Rank
int data = 1;  // data
int recv;      // receive buffer
```

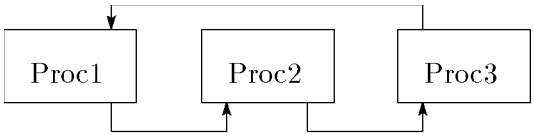


図 12: Send Loop

```
int sendto; // rank to send
int recvfrom; // rank to receive
MPI_Status mpist;

nPnum = MPI_Comm_size(MPI_COMM_WORLD, &nPnum);
nRank = MPI_Comm_rank(MPI_COMM_WOLRD, &nRank);

sendto = nRank + 1;
recvfrom = nRank - 1;
if (sendto >= nPnum) sendto = 0;
if (recvfrom < 0) recvfrom = nPnum-1;

if (nRank == 0){
  MPI_Send(&data, 1, MPI_INT, sendto, 0, MPI_COMM_WORLD);
}
MPI_Recv(&data, 1, MPI_INT, recvfrom, 0, MPI_COMM_WORLD, &mpist);
if (nRank != 0){
 MPI_Send(&data, 1, MPI_INT, sendto, 0, MPI_COMM_WORLD);
}
```

また，図.13 はランダム選択したプロセスにデータを送信する例である．この図ではランダムに選択した 2 つのプロセスへ送信している．しかしながら，MPI では送受信処理は 1 対で起動する必要がある．つまり，送信を起動したときに相手が受信処理を起動しなかった場合，送信側はそこでブロックされてしまう．そこで，この場合は MPI_Bcast による Broadcast 関数によって全員に送信し，受け取り側で判断する．つまり，ランダムに送信するのではなく，ランダムに受信するものである．その例を以下のソースに示す．このソースでは，送信元ランク (ルート) を 3 としている．

```
int nRank;  // My Rank
int data=0;  // data
int root=3;

nRank = MPI_Comm_rank(MPI_COMM_WOLRD, &nRank);

if (nRank == root) data = 1;
```

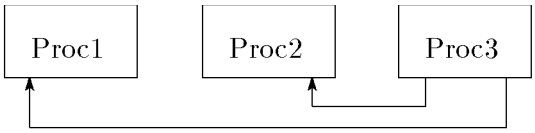


図 13: Send Random Node

```
MPI_Bcast(&data, 1, MPI_INT, root, MPI_COMM_WORLD);

if (nRank != root){
  if ((rand()/RAND_MAX) < 0.5){
    printf("Rank %d: Data=%d\n",nRank,data);
  }
  else{
    printf("Rank %d: data is rejected.\n",nRank);
  }
}
```

このソースでは MPI_Bcast 関数によってルートの data を他の全てのプロセスの data にコピーする．受信したプロセスはある確率 (0.5) でそのデータを受理するかどうかを決定する．

次に，送信時にデータの個数が異なる場合の例であるが，同期，非同期どちらも MPI では受信操作を起動する時点で個数を確定する必要がある．よって，受信操作起動前に個数を確定するためにはあらかじめ個数をネゴシエートする必要がある．以下のソースでは，ランク 0 からランク 1 へ不定個数のデータを送信する例である．

```
int nRank;  // My Rank
int data[10];  // data
int recv[10];      // receive buffer
int num;     // number of datas
MPI_Status mpist;

nRank = MPI_Comm_rank(MPI_COMM_WOLRD, &nRank);
num = (int)((rand()/RAND_MAX)*10)

if (nRank == 0){
 MPI_Send(&num, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
 MPI_Send(&data, num, MPI_INT, 1, 1, MPI_COMM_WORLD);
}
else{
```

```
  MPI_Recv(&num, 1, MPI_INT, 0, 0, MPI_COMM_WOLRD, &mpist);
  MPI_Recv(&recv, num, MPI_INT, 0, 1, MPI_COMM_WOLRD, &mpist);
}
```

このソースでは，まず送信個数を決定しランク 0 がその個数をランク 1 へ送信する．ランク 1 はこれを受信し，その個数で受信操作を起動する．ランク 0 は決定した個数のデータをランク 1 へ送信し，ランク 1 が実際にこれを受信するものである．

11 おわりに

今回は MPI を用いた並列処理環境について簡単に説明した．ワークステーションクラスタは低コストで並列環境を構築できるが，MPI は Cray や SMP マシン等多様なアーキテクチャに対応している．さらに，MPI スタンダードに準拠したソースはどのアーキテクチャでもそのままコンパイルすることができる．つまり，プログラムを身近なワークステーションクラスタで構築し，実際の計算を計算センター等の並列計算機で計算することも可能である．今回は詳しく触れなかったが，MPI には非同期通信機構を備えており，これを用いることで Internet 等の WAN を介したクラスタリングも可能である．このように，ワークステーションクラスタと MPI を用いた並列処理はほんの少しの投資でスーパーコンピュータをも凌駕する性能を出しうる大きなメリットがある．

ただし，並列処理の効率性はアルゴリズムに依存する部分が多く，場合によっては逐次的に実行する場合よりも劣る場合がある．並列アルゴリズムを構築する際，アルゴリズムの並行性，つまり独立して計算可能な部分をうまく並列化して，通信処理を出来るだけ減らす努力をしなければならないのはいうまでもない．

参考文献

[1] Jacek Radajewski, Douglas Eadline: Beowulf HOWTO (日本語訳:Hisakuni NOGAMI), http://jf.linux.or.jp/JFdocs/Beowulf-HOWTO.html, 1998.

[2] Phil Merkey: Beowulf Introduction & Overview, http://www.beowulf.org/intro.html, 1998.

[3] 上林弥彦, 岡部寿男, 浜口清治, 武永康彦 編・訳: プレパラータ先生の超並列計算講義, 共立出版, 1996.

[4] MPI:メッセージ通信インターフェイス標準 (日本語訳ドラフト), http://www.ppc.nec.co.jp/mpi-j/index.html, 1996.