MPI 入門 /LAM による開発

平成8年 10月 3日　第2版

本文書はオハイオ州立大学の許可を得て東海大学工学部通信工学科 清水尚彦監訳のもと清水研究室の下記卒業研究メンバーが翻訳を行った。訳者グループは英語に堪能とはお世辞にも言えず、並列処理の経験もほとんどないが、研究室輪講の一環として訳出することとした。監訳者は本文書が日本の並列処理を学ぶ学生や研究者にとって有用であることを願ってやまないが、限られた人員と時間の中、訳出の正確さや語彙の不統一あるいは訳者の誤解等問題点も多数あると思われる。今後の改良のため本文書の問題点、改良点、意見等は監訳者に連絡いただきたい。

訳者 船山正樹、青柳真佐樹、榎田裕一、黒子周作、佐川岳彦、林亮一、渡邉岳彦、河野賢一

東海大学工学部 清水研究室 1995 年 4 月に計算機関連技術の研究室としてスタートした。1995 年度の活動は主として PVM をベースにワークステーションクラスタでのアプリケーションの並列化を行った。1996 年度はこれに加えてコンピュータアーキテクチャ、MPI の実行環境の基本技術、OS の研究等を行う予定である。できたての研究室なので機材、予算、スタッフのいずれも常に不足しており、機材、予算の寄付、研究委託等いつでも歓迎している。

監訳者 清水尚彦 1985 年に上智大学理工学研究科博士前期過程を終了し、(株) 日立製作所に入社した。依頼 10 年間主として汎用コンピュータを中心にハードウェアの設計業務に従事していたが、1995 年度から東海大学に移りコンピュータ関連の講義と研究を担当している。1993 年度上智大学より博士 (工学) を授与。IEEE、ACM、情報処理学会、電子情報通信学会各会員。MPI メッセージ通信インターフェイス ドラフト (日本語訳) 訳出グループメンバ。

連絡先 〒 259-12 神奈川県平塚市北金目 1117
東海大学工学部通信工学科
清水尚彦
TEL:0463-58-1211(ex.4084)
FAX:0463-58-8320
email:nshimizu@et.u-tokai.ac.jp

LAM はネットワーク接続されたコンピュータのための並列処理環境および開発システムである。LAM は、拡張モニタリングおよびデバッグツールによってサポートされた MPI プログラミング標準である。

LAM / MPI の主な特徴

- MPI 標準の完全実装
- ランタイムおよびプロセス終了後における優れたモニタリングとデバッグツール
- 異機種間のコンピュータ・ネットワーク
- ノードの追加と削除
- ノード障害の検出と復旧
- MPI 拡張と LAM プログラミングの補足
- アプリケーション・プロセス間直接通信
- 堅牢な MPI リソースの管理

この文書の使い方

この文書は大きく４つの章から構成されている。それは、プログラミングと操作を簡単に紹介するチュートリアルから始まる。新規ユーザーはそのチュートリアルから始めるべきである。第２章は、よく使用されるルーチンを中心とした MPI プログラミング入門である。MPI の非標準拡張と、LAM 固有の拡張機能は、第３章に分けて記述される。最終章は、オペレーション・リファレンスであり、LAM マルチコンピュータの設定と起動方法および、プロセスとメッセージの監視方法を述べる。この文書はユーザー指向である。システムがどのように動いているかを詳しく述べてはいないし、すべてのオプションとコマンドとルーチンの機能を詳細に述べてはいない。この文書を補足するために、全てのコマンドと内部ルーチンについてのオンラインマニュアルがある。読者は、特にコードサンプルが C 言語に著しく偏っていることに気がつくだろう。この文書の Fortran 言語版はない。本文では言語による違いが生じないように注意しており、付録において FORTRAN コードサンプルおよびルーチンのプロトタイプを示した。$\langle symbol \rangle$ $\langle symbol \rangle$:これは、読者がタイプする文字であり、コマンドの説明において使う。「」は文書中の節の相互参照に使われる。また、イタリック体は LAM コマンドを識別するためにも使われる。

LAM アーキテクチャ

LAM は、１つのデーモン（サーバー）として各々のコンピュータで動き、ナノカーネルおよび人手でスレッド化された仮想プロセス群によって構成される。ナノカーネルは、簡単なメッセージ通信およびローカルプロセスに対する待ち合わせサービスを提供する。いくつかのデーモン中のプロセスが、ネットワーク通信サブシステムを形成し、それは他のマシン上の LAM デーモンとメッセージの送受信を行う。そのネットワーク・サブシステムは、基本同期機構に加えてパケット化およびバッファリングなどの特徴を追加する。デーモン中の他のプロセスは、プログラムの実行と並列ファイルアクセスのためのサーバーである。その階層化はたいへん明確である：ナノカーネルとネットワーク・サブシステムには何も接続関係はなく、またネットワークとサーバーの間にも何も接続関係はない。ユーザーは必要に応じてサービスの追加、削除ができる。LAM 独自のソフトウェア・エンジニアリングは、ユーザーとシステム管理者にとって不可視であり、単に一つの従来型デーモンに見える。システム開発者は、デーモンを分解し、デーモン中のナノカーネルといくつかの完全なクライアント・プロセスを含むデーモンを仮に生成できる。この開発者モードはユーザーに不可視であるが、個々のデバッギングを簡単にするために、LAM の高度なモジュール化コンポーネントを捨てることができる。それはまた、Trollius から LAM への発展を示し、スケーラブル・マルチコンピュータ上でネイティブに動き、そして統一したプログラミング・インターフェースを通じてホスト・ネットワークに結合それらを結合する。LAM のネットワーク

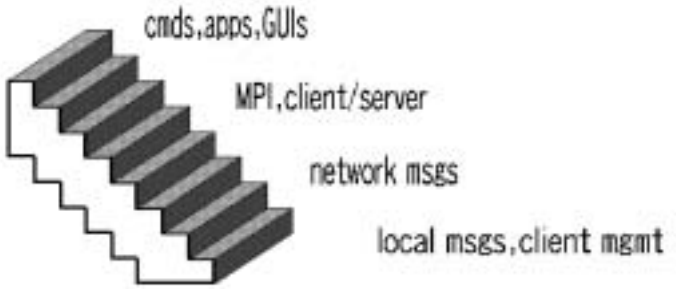

Figure 0.1: LAM の階層デザイン

層は文書化されており、プリミティブと抽象層の上において MPI のようなより強力な通信標準を実装するためのものである（PVM も実行されている）。

デバッギング

LAM の最も重要な特徴は、マルチコンピュータのコントロールである。実行時において見たり変更したりすることができないものはほとんどない。プログラムの実行、停止、再開、削除そして監視はいつでも可能である。メッセージは、マルチコンピュータ上のどこでも見ることが可能である。もしプロセスとメッセージの同期が簡単に表示できるのなら、バグを簡単に見つけることができるだろう。これらおよび他のサービスはプログラミング・ライブラリおよびどのようなシェルからでも動くユーティリティプログラムとして利用できる。

MPI の実行

MPI 同期は、４つの変数（コンテキスト、タグ、送信元ランク、そして送信先ランク）にまとめられるこれらはネットワーク層における LAM の抽象層の同期に写像される。MPI デバッギング・ツールは，LAM/MPI マッピングの情報による LAM 情報を解釈し、MPI プログラマーに詳細な情報を提供する。MPI 仕様の重要な部分は、ランタイムシステムに実装されており、実行環境とは独立している。他のすべての MPI 実装と同様に、ユーザーコードが手に入る前に全てのプロセスはお互いに位置づけを行なうので、LAM は MPI アプリケーションの発行に同期をとる必要がある。mpirun コマンドは、アプリケーションを構成するプログラムを検索およびロードした後、この同期操作を行う。簡単な SPMD アプリケーションは、mpirun コマンドラインに指定するが、より複雑な構成は、アプリケーション・スキーマと呼ばれるファイルに記述される。LAM 上で開発された MPI プログラムは、ソースコードの修正をせずに、MPI をサポートする他のどのようなプラットフォームでも動かせる。LAM はどこへインストールを行ってもよく、LAM および実行可能なアプリケーションを探すために、シェルのサーチパスを使用する。マルチコンピュータは、ファイル中にマシンネームの単純なリストとして指定され、LAM がアクセスの確認、環境の使用開始、そしてその削除を行うために使う。

LAM の入手方法

LAM は、ftp.osc.edu から anonymous ftp 経由で入手可能であり、GNU ライセンスに従い自由に使える。

目 次

Chapter 1

LAM / MPI チュートリアル序論

この章におけるプログラム例は、MPI でよく使用される操作を説明する。また、LAM によるプログラムの実行とデバッグの方法も示される。

1.1 プログラミングチュートリアル

基本アプリケーションの場合、MPI は他のメッセージ通信システムと同じくらい簡単に使える。最初のプログラムは、ちょうど 2 つのプロセスで動くように設計されている。プロセスはメッセージを他方に送り、その後両者は終了する。以下のコードを trivial.c に書き込むか、LAM のソース配布から得なさい (examples/trivial/trivial.c)。

```
/*
 * 2 プロセスシステムにおけるメッセージ伝送。
 */
#include <mpi.h>
#define BUFSIZE         64
int                     buf[64];
int main(argc, argv)
int                     argc;
char                    *argv[];
{
        int             size, rank;
        MPI_Status      status;
/*
 * MPI の初期化。
 */
     MPI_Init(&argc, &argv);
/*
 * プロセスのエラーチェック。
 * ワールドグループ中の自身のランクを決定する。


/*
 * 送り手がランク 0、受け手がランク 1。
 */
     MPI_Comm_size(MPI_COMM_WORLD, &size);
     if (2 != size) {
          MPI_Finalize();
          return(1);
     }
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
/*
 * ランクが 0 なので、ランク 1 にメッセージを送る。
 */
```

```
        if (0 == rank) {
              MPI_Send(buf, sizeof(buf), MPI_INT, 1, 11,
                          MPI_COMM_WORLD);
    }
    /*
     * ランクが 1 なので、ランク 0 からのメッセージを受信する。
     */
        else {
              MPI_Recv(buf, sizeof(buf), MPI_INT, 0, 11,
                          MPI_COMM_WORLD, &status);
        }
        MPI_Finalize();
        return(0);
    }
```

プログラムが、標準の C 言語プログラム構造、ステートメント、変数宣言と型、そして関数を使うことに注意すること。

1.1.1 MPI の世界

プロセスは、各々異なる 0, 1, 2, ..., N-1 の番号がつけられた固有の「ランク」（整数）によって表される。MPI_COMM_WORLD は、「MPI アプリケーションにおける全てのプロセス」を意味する。それはコミュニケータと呼ばれ、メッセージ通信に必要な全ての情報を提供する。ポータブルライブラリは，他のメッセージ通信システムが取り扱えないような同期保護を、コミュニケータを使って提供する。

1.1.2 MPI の起動と終了

他のシステムと同様に、初期設定と MPI プロセスの除去のために 2 つのルーチンが提供される。

```
MPI_Init(int *argc, char ***argv);
MPI_Finalize(void);
```

1.1.3 私はだれ？あなたたちはだれ？

一般的に、並列アプリケーションのプロセスは、自分が誰なのか（自身のランク）を知ることと、他のプロセスがいくつ存在するかを知ることが必要である。プロセスは自分自身のランクを、MPI_Comm_rank() を呼び出すことによって知る。

```
MPI_Comm_rank(MPI\_Comm comm, int *rank);
```

プロセスの総数は、関数 MPI_Comm_size() によって返される。

```
MPI_Comm_size(MPI_Comm comm, int *size);
```

1.1.4 メッセージ送信

メッセージは与えられたデータ型の要素の並びである。MPI は全ての基本的データ型をサポートし、より手の込んだアプリケーションに対しては、実行時において新しいデータ型を作る手段を提供する。メッセージは指定されたプロセスに送られ、ユーザーによって指定されたタグ（整

数）をつけられる。タグは、プロセスが送信・受信するであろう異なるメッセージ型を区別するために使われる。上記のプログラム例においては、タグによってさらに同期機構を付加する提供する必要はない。ゆえに、両側で一致するどんな値でも使用可能である。

```
MPI_Send(void *buf, int count, MPI_Datatype
     dtype, int dest, int tag, MPI_Comm comm);
```

1.1.5　メッセージ受信

受信プロセスは送信プロセスのタグとランクを指定する。MPI_ANY_TAG および MPI_ANY_SOURCE は、任意のタグおよび任意の送信プロセスのメッセージを受信するために使う。

```
MPI_Recv(void *buf, int count, MPI_Datatype
      dtype, int source, int tag, MPI_Comm comm,
      MPI_Status *status);
```

受信されたメッセージについての情報は、status 変数によって得られる。もしワイルドカードを使ったのなら、受信メッセージ・タグが status である。MPI_TAG および送信プロセスのランクが status.MPI_SOURCE である。プログラム例で使用されていない他のルーチンが、受信されたデータ型要素の総数を返す。受信要素数が、MPI_Recv() に指定された数よりも小さい可能性がある場合、それが使われる．受け入れられる受信プロセスよりも送る要素が多い場合は、エラーである。

```
MPI_Get_count(MPI_Status, &status,
     MPI_Datatype dtype, int *nelements);
```

1.1.6　マスター・スレーブの例

次のプログラム例は、動的に負荷分散されたマスター・スレーブアプリケーションのための通信スケルトンである。ソースは、LAM のソース配布から得ることができる (examples/trivial/ezstart.c). プログラムは、最小の 2 つのプロセスで動くように設計されている。すなわち、一つはマスター、もう 1 つはスレーブである。

```
#include <mpi.h>
#define WORKTAG         1
#define DIETAG          2
#define NUM_WORK_REQS   200
static void             master();
static void             slave();
/*
 * メイン
 * このプログラムは実際の MIMD であるが、
 * 簡単な SPMD として書かれている。
 */
int
main(argc, argv)
int                     argc;
char                    *argv[];
{
      int               myrank;
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, /* group of everybody */
            &myrank);               /* 0 thru N-1 */
```

```
        if (myrank == 0) {
                master();
        } else {
                slave();
        }
        MPI_Finalize();
        return(0);
}
/*
 * マスター
 * マスタープロセスは、スレーブに結果の収集をリクエストする。
 */
static void
master()
{
        int               ntasks, rank, work;
        double            result;
        MPI_Status        status;
        MPI_Comm_size(MPI_COMM_WORLD,
                &ntasks);              /* #processes in app */


/*
 *  スレーブを生成する
 */
        work = {NUM_WORK_REQS;}                  /* simulated work */
        for (rank = 1; rank < ntasks; ++rank) {
                MPI_Send(&work,                  /* message buffer */
                        1,                       /* one data item */
                        MPI_INT,                 /* of this type */
                        rank,                    /* to this rank */
                        WORKTAG,                 /* a work message */
                        MPI_COMM_WORLD);         /* always use this */
                work--;
        }
/*
 *  ワークリクエストが無くなるまで、いづれかのスレーブから
 *  結果を受信し、新しいワークリクエストを配布する
 */
        while (work > 0) {
                MPI_Recv(&result,               /* message buffer */
                        1,                      /* one data item */
                        MPI_DOUBLE,             /* of this type */
                        MPI_ANY_SOURCE,         /* from anybody */
                        MPI_ANY_TAG,            /* any message */
                        MPI_COMM_WORLD,         /* communicator */
                        &status);               /* recv'd msg info */
                MPI_Send(&work, 1, MPI_INT, status.MPI_SOURCE,
                        WORKTAG, MPI_COMM_WORLD);
                work--;                         /* simulated work */
        }
/*
 * 未解決のワークリクエストに対する結果を受信する
 */
        for (rank = 1; rank < ntasks; ++rank) {
                MPI_Recv(&result, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
                MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        }
/*
 * 全てのスレーブに終了を告げる
 */
        for (rank = 1; rank < ntasks; ++rank) {
                MPI_Send(0, 0, MPI_INT, rank, DIETAG,
                MPI_COMM_WORLD);
        }
```

```
}
/*
 * スレーブ
 *
 * スペシャルターミネイションリクエストを受信するまで、
 * それぞれのスレーブプロセスはワークリクエストを受信し
 * 結果を返す
 *
 */
static void
slave()
{
        double             result;
        int                work;
        MPI_Status         status;
        for (;;) {
                MPI_Recv(&work, 1, MPI_INT, 0, MPI_ANY_TAG,
                        MPI_COMM_WORLD, &status);
/*
 * Check the tag of the received message.
 * 受信メッセージのタグをチェックする
 */
        if (status.MPI_TAG == DIETAG) {
                return;
        }
        sleep(2);
        result = 6.0;           /* simulated result */
        MPI_Send(&result, 1, MPI_DOUBLE, 0, 0,
                MPI_COMM_WORLD);
        }
}
```

本格的な MPI アプリケーションを作成する前に、ランクとタグとメッセージ長の働きをマスターしておくべきである。

1.2　オペレーション チュートリアル

LAM を動かす前に、必要な環境変数と、シェルのパスを設定しなければならない。以下のコマンドまたは同等のものを、シェル起動ファイル (.cshrc) に追加する（C シェルと仮定する)。LAM の起動に rsh が使われている場合、リモートマシンでは .login の設定が有効でないので、ここに設定してはならない。

```
setenv LAMHOME<LAM のインストール先ディレクトリ>
set path = ($path $LAMHOME/bin)
```

LAM をインストールした人またはローカルシステムの管理者は、LAM のインストール先ディレクトリを知っているだろう。シェルの起動ファイル (.cshrc) を編集した後、その設定を有効にするため次のコマンドを入力する。これは、次回から UNIX システムに login する場合には必要とされない。

```
% source .cshrc
```

多くの LAM コマンドは 1 個以上のノード識別子を必要とする。ノード識別子は、コマンドラインにおいて、n< *list* > として指定される。ここで、< *list* >はコンマで区切られたノード識別子またはノード識別子の範囲である 。

```
n1
n1,3,5-10
```

ニーモニックの ‘h‘(here の h) は、コマンドが入力されたローカルノードを示す。

1.2.1　コンパイル

実行用の LAM プログラムのコンパイルには、どのようなネイティブの C 言語コンパイラでも利用可能である。すべての LAM ランタイムルーチンは、2 〜 3 のライブラリ中にある。LAM は、適当なヘッダーとライブラリディレクトリで cc を利用する hcc と呼ばれるコマンドを提供し、あたかもネイティブな cc のように利用することができる。

```
%hcc -o trivial trivial.c -lmpi
```

おもな内部 LAM ライブラリは自動的にリンクされる。MPI ライブラリは明示的にリンクされる。LAM は異機種のコンピュータ処理をサポートしているので、ユーザの各マシンの様々な CPU 用にソースコードをコンパイルすることはユーザにまかされている。

コンパイラにより報告されたエラーを訂正した後に、LAM セッションを始めることができる。

1.2.2　LAM を起動する

LAM を起動する前に、ユーザーは、マルチコンピュータを構成するマシンを指定する。1 行ずつマシン名をリスト化したホストファイルを作成する。マシン「ohio」と「osc」のための例を以下に示す。#文字で始まるラインは、コメントラインとみなされる。

```
# a 2-node LAM
ohio
osc
```

ホストファイルの最初のマシンは、ノード 0、2 番目のマシンはノード 1 に割り当てられる。以下同様である。ここで、マルチコンピュータが LAM を動かす準備ができていることを確認する。ユーザーがアクセス権をマルチコンピュータの個々のマシンに持ち、LAM がインストールされていて、アクセス可能かどうかを、recon ツールはチェックする。

```
%recon -v <ホストファイル>
```

もし recon コマンドが問題を報告しなければ、lamboot ツールによって LAM セッションを開始する。

```
%lamboot -v <ホストファイル>
```

-v（冗長）オプションを付けると、lamboot コマンドは、スタートアッププロセスの進捗を報告する。ここで、シェルのプロンプトに戻ってくる。LAM は特別なシェルやインタフェース環境を提供しない。起動後、たとえ全てうまくいっているように見えたとしても、個々のノードの通信を検証する。tping は、この目的のための簡単な信用形成コマンドである.

```
% tping n0
```

すべてのノードに対して、このコマンドを繰り返すか、または 1 回で済ますためにはブロードキャストのニーモニック N で、全てのノードを確認する。tping は、ローカルノード（ユーザーが tping を入力した所）と指定されたノードの間にメッセージを送って応答を確認する。tping の実行が成功することで、相手ノードおよび相手ノードとローカルノードの間のルート上にあるノード、そしてそれらのノード間の通信が適切に作動していることが証明される。もし tping が失敗したなら、Ctrl-Z を押し、wipe ツールでセッションを終わらせて、それから、システムを再スタートする。(‘セッションの終了 ‘を参照）

1.2.3　プログラムの実行

プログラムを実行するためには、mpirun コマンドを利用する。最初の例は、2 つのプロセスで動作するように設計されている。-c< # >オプションにより、ラウンドロビン方式で選ばれたノードで、与えられたプログラムの複製が実行される。

```
% mpirun -v -c 2 trivial
```

上記の起動例は、mpirun を動かすマシン上でプログラムが実行可能であると仮定している。mpirun は、実行前にプログラムを相手ノードに転送することもできる。このチュートリアル用に、均質なマルチコンピュータを仮定するならば、両方のプロセスを動かすために、-s h オプションを使用できる。

```
% mpirun -v -c 2 -s h trivial
```

もしプロセスが正しく実行したならば、プロセスは終了しトレースを残さない。もしより多くのフィードバックを望むなら、tprintf() 関数を使用することができる。また、他の UNIX プログラムと同様の方法で、UNIX シェルからローカルノードにプログラムを起動できる。その場合、与えられた全ノードにおいて、各々1 個の MPI プログラムが起動される。

```
% trivial
```

1.2.4　モニタリング

最初の例題プログラムはすぐに実行が完了するので実行状態をモニタすることはできないそこで、MPI_Recv() の call において、タグを 11 から 12 に変更し、プログラムを再コンパイルし、前回のように再度実行する。この状態ではイベントが異なっているので、受信プロセスは送信プロセスからのメッセージに同期をとることができない。mpitask コマンドによってすべての MPI プロセスの status を調べる。

```
% mpitask
TASK (G/L)      FUNCTION    PEER|ROOT   TAG    COMM    COUNT    DATATYPE
1/1  trivial    Recv        0/0         12     WORLD   64       INT
```

たった１つのプロセスしか動作しておらず、同期メッセージが未到着となっている MPI_Recv() の呼び出しによって、そのプロセスがブロックされていることがわかる。　プログラムを参照すると、これは、MPI アプリケーションのランク 1 のプロセスであることがわかる。それは、MPI タスク状態表示の１桁めで確認できる。最初の数はワールドグループ内のランクである。2 番目の数は、MPI_Recv() で使用されているコミュニケータ内のランクである。このケース（そして簡単な通信構造を持つ多くのアプリケーションの）の場合、world group のランクと一致する。MPI_Recv() の指定したメッセージの送信元も同じく示される。同期タグは 12 であり、受信バッファの長さはタイプ MPI_INT の 64 要素分である。何が送信側のプロセスランク 0 に起こったか？[1]まだメッセージが受け取られていないのに、送信側プロセスをさらにブロックしなくて良いのだろうか？その理由を説明する。LAM は一定量のシステムメッセージバッファを提供していて、メッセージはそのシステムバッファを経由し、ランク 1 のプロセスに伝えられる。したがって MPI_Send() はシステムバッファにメッセージを全て転送した時点で、制御を返すことができ、プロセスは MPI_Finalize() を呼び出した後、終了した。個々の MPI プロセスのためのメッセージキューと見なすことができるシステムバッファは、mpimsg コマンドによって確認できる。mpimsg の表示から、メッセージが MPI_COMM_WORLD でプロセスランク 0 から送られて、送信先であるプロセスランク 1 のメッセージキューで待っていることがわかる。タグは 11 であり、メッセージはタイプ MPI_INT の 64 個の要素を含んでいる。

```
% mpimsg
SRC(G/L)    DEST(G/L)    TAG    COMM    COUNT    DATATYPE    MSG
0/0         1/1          11     WORLD   64       INT         n1,#0
```

この情報は、MPI_Send に与えられた引数と一致している。アプリケーションが誤っており完了することがないので、lamclean コマンドでアプリケーションを削除することが出来る。

```
% lamclean -v
```

これによって LAM セッションは、lamboot 後と同じ状態になるはずである。セッションを終了させてから、lamboot によって再スタートすることもできるが、これはずっと時間のかかる操作である。この状態で、プログラムを訂正し、再コンパイルし、再計算することが可能である。

1.2.5　セッションの終了

LAM を終了するためには、wipe ツールを使用する。引数ホストファイルは,lamboot のものと同じでなければならない。

[1]訳注:mpitask によると、受信が完了していないのに、送信側プロセスがすでに消えている

```
% wipe -v <host file>
```

Chapter 2

MPI Programming Primer MPI プログラミング入門

2.1 基本的な概念

メッセージパッシングインタフェース（MPI）を通して、アプリケーションは、その並列環境をプロセスの静的グループとみなす。MPI プロセスは、未知の親から、０ないし１以上の兄弟プロセスとともに誕生する。プロセスのこの初期の集合は、 world group と呼ばれる。ランクと呼ばれる独特な数は、0 から N-1 まで順番に、個々のメンバープロセスに割り当てられる。ここで、N は world group のプロセスの全体の数である。メンバーは自分のランクおよび world group のサイズを調べることができる。各プロセスは、全て同一のプログラムを実行するか（SPMD）または別々のプログラム (MIMD) を実行する。world group のプロセス群はさらに分割され、サブグループを生成するかもしれない。個々のサブグループにおいてプロセスは異なるランクを与えられる。プロセスは、メッセージを要求されたグループ内の送信先ランクを持つプロセスに送る。プロセスは、メッセージを受け取る時に送信元ランクを指定するかもしれないし、しないかもしれない。メッセージは、ユーザが任意に指定するタグと呼ばれる同期用の整数でさらに選択される。ただし、タグを無視して受信することもできる。MPI の重要な特徴は、独立系ソフトウェア開発者 (ISV) にライブラリの設計におけるタグの選択において、他の独立系開発者や library 利用者の選択したタグが library 内で競合しないということを保障する能力である。コンテキストと呼ばれるさらに強力な同期用の整数は、MPI によって割り当てられて、すべてのメッセージに自動的に付属する。従って、MPI の 4 個の同期用の変数は、送信元ランク、送信先ランク、タグ、およびコンテキストである。コミュニケータは、1 グループについての情報を含み、1 個のコンテキストを含んでいる不透明な MPI データ構造である。コミュニケータは、すべての MPI 通信ルーチンに必要とされる引数である。プロセスが生成されて、MPI を初期設定した後に、2 つのあらかじめ決められたコミュニケータが利用可能である。

`MPI_COMM_WORLD`	`world group`
`MPI_COMM_SELF`	自分一人だけの `group`

ほとんどのアプリケーションは,world group 以外のコミュニケータを必要としない。もし新しいサブグループまたは新しいコンテキストが必要なら、コミュニケータを追加しなければならない。MPI 定数、テンプレート、およびプロトタイプは、MPI ヘッダーファイル、mpi.h にある。

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

2.2 初期化

MPI_Init	MPI の初期化。
MPI_Finalize	MPI の終了。
MPI_Abort	異常終了。
MPI_Comm_size	グループのプロセスを得る。
MPI_Comm_rank	グループ内での自分のランクを得る。
MPI_Initialized	MPI が初期化済みか確認する。

プログラムにより呼ばれる最初の MPI ルーチンは、MPI_Init() であるなければならない。コマンドライン引数は MPI_Init() に手渡される。

```
MPI_Init(int *argc, char **argv[]);
```

プロセスは MPI の操作を MPI_Finalize() により終了する。

```
MPI_Finalize(void);
```

エラー条件に応じる場合、プロセスは自分自身とコミュニケータ内の全メンバを MPI_Abort() で終了することができる。MPI_Abort() の実装は、オペレーションシステムに矛盾しない方法で、ユーザにエラーコードを報告することができる。

```
MPI_Abort (MPI_Comm comm, int errcode);
```

2.2.1 並列プロセスの基本情報

ほとんどの並列アプリケーションにとって並列プロセスの総数と自身のプロセス識別子は非常に有益な 2 個の数である。この情報は、MPI_Comm_size() と MPI_Comm_rank を使って、MPI_COMM_WORLD コミュニケータから得られる。

```
MPI_Comm_size (MPI_Comm comm, int *size);
MPI_Comm_rank (MPI_Comm comm, int *rank);
```

もちろん、どのようなコミュニケータでも使用できるけれども、通常 world コミュニテータ情報は、並列アプリケーション全体に関わるデータを切り分けるための鍵となる。

2.3　２点間ブロッキング通信

MPI_Send	標準モードでメッセージ送信する。
MPI_Recv	メッセージを受信する。
MPI_Get_count	受信した要素数を数える。
MPI_Probe	メッセージの到着を待つ。
MPI_Bsend	バッファモードでメッセージを送信する。
MPI_Ssend	同期モードでメッセージを送る。
MPI_Rsend	待機モードでメッセージを送信する。
MPI_Buffer_attach	バッファモード送信用のバッファを割り付ける。
MPI_Buffer_detach	現在のバッファを解放する。
MPI_Sendrecv	標準モードで送信し、次に受信する。
MPI_Sendrecv_replace	１個の領域から送信し、そして同じ領域に受信する。
MPI_Get_elements	受信した基本要素を数える。

このセクションでは、１対１ブロッキングメッセージパッシングルーチンについて述べる。MPI での「ブロッキング」という言葉の意味は、関連したデータバッファが再利用可能になるまで、そのルーチンから制御が戻ってこないことを意味する。１対１メッセージは、1 個のプロセスが送信し、1 個のプロセスが受信する。

2.3.1　送信モード

メッセージ通信の基本設計において, フロー制御とバッファリングについては異なった選択が可能である。MPI は単一な選択を強要せず、代わりに、ほとんどのアプリケーションの同期、データ転送、および性能に対する要求を満足する 4 個の転送モードを提供する。このモードは、おなじ引数を持つ４つの送信ルーチンによって送信側が選択する。受信ルーチンはたった１つしかない。４つのモードは以下の通りである。

標準モード (standard)

> システム側でメッセージのバッファリングが可能であるか（システムはバッファリングの義務はない)、メッセージが受信された時に送信が完了する。

バッファドモード (buffed)

> アプリケーション内に確保した領域にメッセージがバッファリングされるか、あるいはメッセージが受信された時送信が完了する。

同期モード (synchronous)

> メッセージが受信された時、送信が完了する。

待機モード (ready)

> 対応する受信側で受信が開始するまで、送信を開始してはいけないが、送信はすぐに完了する。

2.3.2 標準モード送信

標準のモードはほとんどのアプリケーションのニーズにかなう。標準モードのメッセージは、MPI_Send() で送信する。

```
MPI_Send (void *buf, int count, MPI_Datatype
     dtype, int dest, int tag, MPI_Comm comm);
```

MPI メッセージは単なる、生のバイト配列ではなく、型宣言された要素を単位として数えられる。要素タイプは簡単な生のバイトまたは複雑なデータ構造であるかもしれない。(「メッセージデータ型」参照) 4 個の MPI 同期用の変数は MPI_Send() のパラメーターである。送信元ランクは呼びだし側プロセスのランクのことである。送信先ランクとメッセージタグは明示的に与えられる。コンテキストはコミュニケータの特性である。ブロックキングモードのルーチンの場合、MPI_Send() から制御が戻った時に、バッファは上書き可能となる。ほとんどのシステムはいくつかのメッセージ、特に短いメッセージをバッファリングするが、受信が発行されない場合では、プログラマは１つのメッセージさえも MPI_Send() でバッファリングすることを期待してはならない。対応する受信が発行されるまで、そのルーチンから決して制御が戻らないと思うべきである。

2.3.3 受信

どのようなモードのメッセージでも MPI_Recv() によって受け取られる。

```
MPI_Recv (void *buf, int count, MPI_Datatype
     dtype, int source, int tag, MPI_Comm comm,
     MPI_Status *status);
```

ここでまた、(MPI_Send() に対して) 送信先ランクと送信元ランクを入れ換えた４つの同期用変数が指定される。送信元ランクとタグは、特別な値である MPI_ANY_SOURCE と MPI_ANY_TAG によって無視可能である。もしこれら両方のワイルドカードが利用された場合、与えられたコミュニケータに対して発行される直近のメッセージが受け取られる。

2.3.4 ステータス・オブジェクト

MPI_Send() に存在しない引数は、ステータスオブジェクトのポインタである。ステータスオブジェクトは、MPI_Recv() から制御が戻る時、有益な情報によって満たされる。もし送信元ランクまたはタグにワイルドカードが使用されたならば、ステータスオブジェクトから直接的に実際に受信された送信元ランクまたはタグを入手することが出来る。

```
status.MPI_SOURCE          送信元ランク
status.MPI_TAG             送信元で与えたタグ
```

2.3.5 メッセージ長

MPI プログラムが指定された受信バッファより大きなメッセージを受信することは誤りである。その場合、メッセージが切り詰められてしまうか、エラー条件が発生するか、その両方となるかである。指定された受信バッファより短いメッセージを受け取ることは全く問題ない。もし短いメッセージが到着したならば、アプリケーションは MPI_Get_count() によってメッセージの実際の長さを問うことができる。

```
MPI_Get_count (MPI_Status *status,
     MPI_Datatype dtype, int *count);
```

ステータスオブジェクトと MPI datatype は、MPI_Recv() の引数として与えたものを用いる。返された count は、与えられたデータ型で受信した要素の数である。(「メッセージデータ型」参照)

2.3.6 調査

受信バッファを事前に割り当てておくのは、実際的でない場合が時々ある。MPI_Probe() はメッセージを同期させて、実際にメッセージは受け取らずに、メッセージについての情報を返す。同期用の変数とステータスオブジェクトだけは、引数として提供される。MPI_Probe() は、メッセージが同期されるまで制御が返らない。

```
MPI_Probe (in source, int tag, MPI_Comm comm,
     MPI_Status *status);
```

適当なメッセージバッファが準備された後に、MPI_Probe() により報告された同じメッセージは、MPI_Recv() によって受信できる。

2.4　１対１ノンブロッキング通信

MPI_Isend	標準メッセージ送信開始。
MPI_Irecv	メッセージ受信開始。
MPI_Wait	保留中要求完了。
MPI_Test	保留中要求のチェック又は、完了。
MPI_Iprobe	チェックメッセージ到着。
MPI_Ibsend	蓄積メッセージ送信開始。
MPI_Issend	同期メッセージ送信開始。
MPI_Irsend	準備メッセージ送信開始。
MPI_Request_free	保留中要求開放。
MPI_Waitany	１要求完了。
MPI_Testany	１要求のチェック又は、完了。
MPI_Waitall	全要求完了。
MPI_Testall	全要求のチェック又は、完了。
MPI_Waitsome	１個以上の要求完了。
MPI_Testsome	１個以上の要求のチェック又は、完了。
MPI_Cancel	保留中要求のキャンセル。
MPI_Test_cancelled	保留中要求のキャンセルのチェック。

「ノンブロッキング」という用語は、起動されたメッセージ転送操作が完了していなくても、ルーチンがすぐに戻ってくるということを意味している。ノンブロッキングルーチンが戻って来きてもアプリケーションは、メッセージバッファを安全に再利用できないかもしれない。4 個のブロック送信ルーチンおよび 1 個のブロック受信ルーチンに対してノンブロッキングの対応ルーチンが存在する。ノンブロッキングルーチンは、リクエストオブジェクトと呼ぶもう一つの出力引数を持つ。リクエストは後で完了ルーチンの一つに渡される。いったん操作が完了したら、そのメッセージバッファは再利用できる。重要な計算を続行しつつメッセージ転送をできるだけ早く行い、できるだけ後で完了を宣言することが、ノンブロッキング・メッセージ通信の目的である。この最も速い時刻と最も遅い時刻が同一である場合、ノンブロッキングルーチンは有益ではない。また、ハードウェアによってはノンブロッキング操作によって、通信と計算のオーバーラップが可能であり結果として性能が改善される。

MPI_Isend() は標準のノンブロッキングメッセージ送付を開始する。

```
MPI_Isend (void *buf, int count, MPI_Datatype
     dtype, int dest, int tag, MPI_Comm comm,
     MPI_Request *req);
```

同様に、MPI_Irecv() はノンブロッキングメッセージ受信を開始する。

```
MPI_Irecv (void *buf, int count, MPI_Datatype
     dtype, int source, int tag, MPI_Comm comm,
     MPI_Request *req);
```

2.4.1 要求完了

両ルーチンは，対応するブロッキングルーチンと同じ引数を用いる。アプリケーションがノンブロッキング送信か受信の完了を希望する時には、対応する request 引数を与えて完了ルーチンを呼ぶ。Test() ルーチンはノンブロッキングであり、Wait() ルーチンはブロッキングである。他の完了ルーチンは、複数のリクエストに対応した動作を行う。

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

MPI_Test() は、要求が完了したかどうかを出力引数 flag に返す。もし flag が真であれば、引数 status は対応する情報によって満たされる。もし要求が受信操作であったならば、status オブジェクトには MPI_Recv() の場合と同じ情報が与えられる。MPI_Wait() は、完了までブロックされるので，ステータスオブジェクト引数は，必ず満たされている。

2.4.2 調査

MPI_Iprobe() は MPI_Probe() のノンブロッキング版であるが、それはリクエストオブジェクトを返さない。なぜならこのルーチンは完了を待つ必要のある転送を何も開始しないからである。このルーチンは、引数を受信に使用するため一致したメッセージが到着したか否かを引数 flag に設定する。

```
MPI_Iprobe (int source, int tag, MPI_Comm comm,
     int *flag, MPI_Status *status);
```

プログラマーは、ノンブロッキング・ルーチンを単純にブロック呼び出しの早いバージョンとして、全てのアプリケーションにおいて好まれた選択であると考えるべきではない。いくつかの実装は、ノンブロッキング・ルーチンが提供するべき性能最適化がなされない。メッセージパッシングインタフェースに準拠するため実装では、ノンブロッキング転送の方が遅くなることもあり得る。プログラマーは、ノンブロッキング・ルーチンを考える前に計算と通信を明示的にかつ実質的にオーバーラップさせるべきである。

2.5　メッセージデータ型

MPI_Type_vector	等間隔に並んだデータ型の複製でデータ型を生成する。
MPI_Type_struct	異なるデータ型の複製を生成する。
MPI_Address	メモリ位置の絶対番地を取得する。
MPI_Type_commit	メッセージ転送のデータ型を使用する。
MPI_Pack	連続しているバッファをパックする。
MPI_Unpack	連続しているバッファをアンパックする。
MPI_Pack_size	必要なパッキングバッファサイズを取得する。
MPI_Type_continuous	連続データ型配置を生成する。
MPI_Type_hvector	バイト単位に等間隔に並んだデータ型の複製でデータ型を生成する。
MPI_Type_indexed	元のデータの複製でブロック並びを生成する。
MPI_Type_hindexed	バイト単位によってインデックスを生成する。
MPI_Type_extent	データ型のスペース範囲を取得する。
MPI_Type_size	データ型のスペースの総計を取得する。
MPI_Type_lb	データ型の下限の変位を取得する。
MPI_Type_ub	データ型の上限の変位を取得する。
MPI_Type_free	データ型を解放する。

異機種間接続の計算においては、そのコンピュータアーキテクチャ間で必要とするマシン表現に、何らかの方法で変換できるようメッセージデータは型付けもしくは、記述されなければならない。MPI はメッセージデータ型として簡単な原始マシン型から複雑な構造体、配列、インデックスまで記述可能である。すべてのメッセージパッシングルーチンは，データ型引数（C 言語での typedef は MPI_Datatype）を受け入れる。例えば、「MPI_Send()」を参照せよ。メッセージデータは、与えられた型の要素の一群として指定される。ほとんどのコンピュータアーキテクチャにおいて基本的な単位データ型であるいくつかの MPI_Datatype 値は、あらかじめ決められている。

MPI_CHAR	符号付 char 型
MPI_SHORT	符号付 short 型
MPI_INT	符号付 int 型
MPI_LONG	符号付 long 型
MPI_UNSIGNED_CHAR	符号無 char 型
MPI_UNSIGNED_SHORT	符号無 short 型
MPI_UNSIGNED	符号無 short 型
MPI_UNSIGNED_LONG	符号無 long 型
MPI_FLOAT	float 型
MPI_DOUBLE	double 型
MPI_LONG_DOUBLE	long double 型
MPI_BYTE	raw byte 型

これらの基本データ型により占められるバイト数は、C 言語の定義に対応している．従って、

MPI_INT は、あるマシンでは 4 バイト、別のマシンでは 8 バイトを占めるかもしれない。1 つの MPI_INT が、送受信の両方によって明記されたのであれば、メッセージはある方向対してパディングが必要とされることがあるが、常に正しく転送される。逆方向において、整数はより小さいバイト数であらわせないかもしれず、その場合通信は失敗する。

2.5.1 派生データ型

派生データ型は、基本データ型、または前もって定義された派生データ型を結合することによって定義される。派生データ型は、要素の複数の配列から成るメモリレイアウトを示す。この能力の一般化は、配列長、配列要素データ型、および配列変位に対する制御能力の異なる 3 種類のコンストラクタルーチンによって実現される。

`contiguous`	1 配列長,	変位なし,	単一データ型
`vector`	1 配列長,	単一変位,	単一データ型
`indexed`	複数配列長,	複数変位,	単一データ型
`structure`	複合全指定可		

2.5.2 ストライド・ベクトル・データタイプ

列めもりが順に蓄えられた R 行と C 列からなる 2 次元の行列において、アプリケーションは、1 個の列全体を通信することを望むとする。ベクトル型の派生データ型はこの要求に適合する。

```
MPI_Type_Vector (int count, int blocklength,
    int stride, MPI_Datatype oldtype,
    MPI_Datatype *newtype);
```

もし行列要素が MPI_INT であるならば、この要求のための引数は下記のようになる：

```
int              R, C;
MPI_Datatype newtype;
MPI_Type_vector(R, 1, C, MPI_INT, &newtype);
MPI_Type_commit(&newtype);
```

ブロック（配列）の数は列の要素の数 (R) と等しい。個々のブロックはちょうど 1 個の要素を含み、要素はお互いに列の要素の数 (C)[1]のストライド (変位) を持つ。

2.5.3 構造データ型

C 言語の構造体で例示されるような任意のレコードは、よく使用されるメッセージ形態である。最も柔軟な MPI の派生データ型である構造体を使用する為には，そのメモリーレイアウトを記述する必要がある。

```
MPI_Type_struct (int count, int blocklengths[],
     MPI_Aint displacements[], MPI_Datatype
     dtypes[], MPI_Datatype *newtype);
```

以下のコードにおいて、C 言語の構造体の多様なフィールドは、最もポータブルで安全な方法で、MPI_Type_struct() によって記述される。

[1] このデータ型を使用する場合、1 番目の列の最後の要素と 2 番目の列の最初の要素との間の最終的な位置関係を推定できないので，行列から複数の列を送信するには情報が不十分であることに注意が必要である。

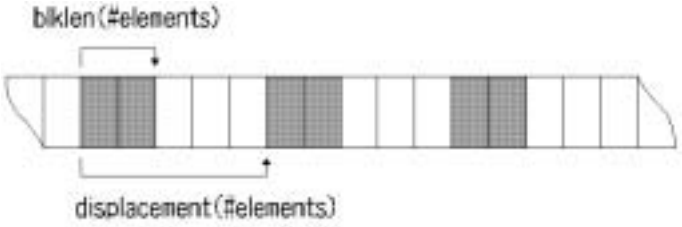

Figure 2.1: ストライド・ベクトル・データタイプ

```
/*
 * 簡単ではない構造体
 */
struct cell {
      double            energy;
      char              flags;
      float             coord[3];
};
/*
 * このデータ型のアライを送信することができる。
 */
struct cell             cloud[2];
/*
 * cell 構造のための新しいデータ型
 */
MPI_Datatype           celltype;
```

1 つの解決策は，MPI_Type_struct() と MPI_UB を使用することである。「構造体データ型」を参照のこと。

```
int                    blocklengths[4] = {1, 1, 3, 1};
MPI_Aint               base;
MPI_Aint               displacements[4];
MPI_Datatype           types[4] = {MPI_DOUBLE, MPI_CHAR,
                       MPI_FLOAT, MPI_UB};
MPI_Address(&cloud[0].energy, &displacement[0]);
MPI_Address(&cloud[0].flags, &displacement[1]);
MPI_Address(&cloud[0].coord, &displacement[2]);
MPI_Address(&cloud[1].energy, &displacement[3]);
base = displacement[0];
for (i = 0;
i < 4;
++i) displacement[i] -= base;
MPI_Type_struct(4, blocklengths, displacements, types,
            &celltype);
MPI_Type_commit(&celltype);
```

構造データ型の相対位置を示す displacements 引数は，C 言語の構造体の最初の記憶位置からの相対バイト数の配列である。C 言語の構造体のパックと配列のアライに対するコンパイラの方針を推測することなしに通信を行うためには，MPI_Address() ルーチンおよびポインタの計算は，正確な値を得るための最良の方法である。MPI_Address() はメモリー位置の絶対アドレスを単に返すルーチンである。構造体内の最初の要素の変位は 0 である。与えられたデータ型の配列 (例えば，MPI_Send() で 1 以上の転送数を指定する場合）を転送するとき，MPI は，配列要素が連続に蓄積されていると仮定する。派生データ型のメモリレイアウトの最後に、MPI_UB と言う人工

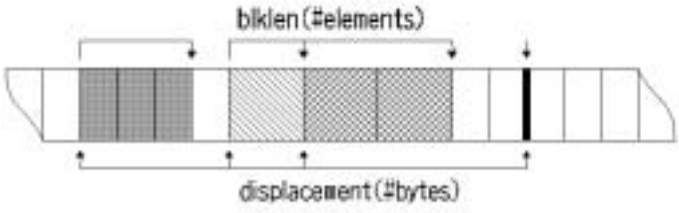

Figure 2.2: データ型の生成

的な型を持つ要素を追加して、配列の第 2 要素の始めのバイトまでの変位与えることで，データ型の記述においてギャップを指定できる。MPI_Type_Commit() によって，非常に複雑なデータ型の足がかりとしての中間データ型から、メッセージ転送に使用されるデータ型を分離する。派生データ型は通信で使われる前にコミットされなければならない。

2.5.4　パックされたデータ型

派生データ型の記述は，ランタイムにおいて生成後に固定される。そこで，構造内の特定フィールドのブロック長などのわずかな細部構造でも変わるならば新しいデータ型が必要となる。多くの派生データ型を生成しなければならない事に加え、受信側は，次のメッセージとしてほとんど同一の派生データ型のどの型が到着するのか前もって知らないかもしれない。この問題に対する MPI の解決策は、連続しているメッセージバッファを，パック／アンパックルーチンで組み立て／分解することである。パックされたメッセージは、特別の MPI データ型 MPI_PACKED を持ち、バイト長と等しい要素数を与えて転送される。

```
MPI_Pack_size (int incount, MPI_Datatype dtype,
     MPI_Comm comm, int *size);
```

MPI_Pack_size() は、与えられたデータ型を，パックする場合に必要とされるメッセージバッファサイズを返す。これは，タイプ記述から期待される値より大きいかもしれない。実装依存のパッキングオーバヘッドが隠されていることがあるからである。

```
MPI_Pack (void *inbuf, int incount, MPI_Datatype
     dtype, void *outbuf, int outsize,
     int *position, MPI_Comm comm);
```

複数の同種要素の連続ブロックからなるデータは、MPI_Pack() を用いて各 1 回ずつパックされる。個々の呼び出しの後に、パックされたメッセージバッファの現在位置が更新される。「入力」データは、パックされるべき要素であり、「出力」データは、パックされたメッセージバッファである。オーバフローを防ぐため出力サイズは，パックされたメッセージバッファの最大サイズとすること。

```
MPI_Unpack (void *inbuf, int insize,
     int *position, void *outbuf, int outcount,
     MPI_Datatype datatype, MPI_Comm comm);
```

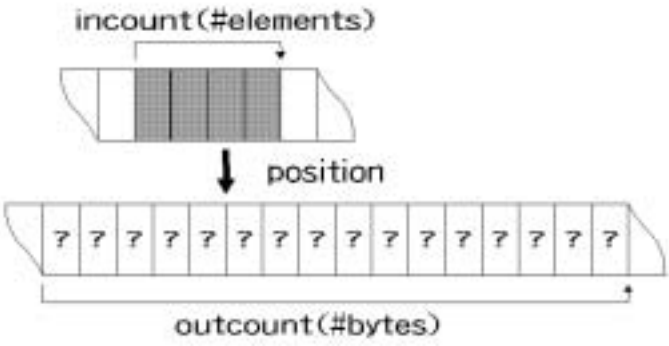

Figure 2.3: パックされたデータ型

MPI_Unpack() は、MPI_Pack() の逆を行うルーチンであり,「入力」データはパックされたメッセージバッファ、「出力」データはアンパックされた要素である。整数 count と４バイト文字列からなるいくつかの (count 数の) インターネットアドレスと，同じ数だけの short 型のポート番号からなる可変長のメッセージ転送を行うネットワークアプリケーションを考える。

```
#define MAXN                100
unsigned char               addrs[MAXN][4];
short                       ports[MAXN];
```

以下のコードでは、メッセージは，与えられた数に従ってパックされ送信される。

```
unsigned int                membersize, maxsize;
int                         position;
int                         nhosts;
int                         dest, tag;
char                        *buffer;
/*
 * Do this once.
*/
MPI_Pack_size(1, MPI_INT, MPI_COMM_WORLD, &membersize);
maxsize = membersize;
MPI_Pack_size(MAXN * 4, MPI_UNSIGNED_CHAR, MPI_COMM_WORLD,
            &membersize);
maxsize += membersize;
MPI_Pack_size(MAXN, MPI_SHORT, MPI_COMM_WORLD, &membersize);
maxsize += membersize;
buffer = malloc(maxsize);
/*
 * Do this for every new message.
*/
nhosts = /* some number less than MAXN */ 50;
position = 0;
MPI_Pack(nhosts, 1, MPI_INT, buffer, maxsize, &position,
            MPI_COMM_WORLD);
MPI_Pack(addrs, nhosts * 4, MPI_UNSIGNED_CHAR, buffer,
            maxsize,&position, MPI_COMM_WORLD);
MPI_Pack(ports, nhosts, MPI_SHORT, buffer, maxsize,
            &position,MPI_COMM_WORLD);
MPI_Send(buffer, position, MPI_PACKED, dest, tag,
            MPI_COMM_WORLD);
```

パックされたメッセージの最大のサイズを格納できるだけのバッファが割り当てられているとする。以下のコードでは、メッセージ受信した後、メッセージの先頭にパックされているカウント

数に基づきアンパックされる。

```
int             src;
int             msgsize;
MPI_Status status;
MPI_Recv(buffer, maxsize, MPI_PACKED, src, tag,
            MPI_COMM_WORLD,&status);
position = 0;
MPI_Get_count(&status, MPI_PACKED, &msgsize);
MPI_Unpack(buffer, msgsize, &position, &nhosts, 1, MPI_INT,
            MPI_COMM_WORLD);
MPI_Unpack(buffer, msgsize, &position, addrs, nhosts * 4,
            MPI_UNSIGNED_CHAR, MPI_COMM_WORLD);
MPI_Unpack(buffer, msgsize, &position, ports, nhosts,
            MPI_SHORT,MPI_COMM_WORLD);
```

2.6 集団メッセージパッシング

MPI_Bcast	すべてのグループのメンバーにメッセージを送る。
MPI_Gather	すべてのメンバーからメッセージを受け結合させる。
MPI_Scatter	分離し、そしてすべてのメンバーのデータを分散させる。
MPI_Reduce	すべてのメンバーからのメッセージを結合する。
MPI_Barrier	すべてのグループのメンバーがこの点に達するまで待つ。
MPI_Gatherv	カウントとバッファ変位を変える。
MPI_Scatterv	カウントとバッファ変位を変える。
MPI_Allgather	集めそれからブロードキャストする。
MPI_Allgatherv	変位を変えられた結果を集めそれからブロードキャストする。
MPI_Alltoall	集めそれから分散させる。
MPI_Alltoallv	変位を変えられた結果を集めそれから分散させる。
MPI_Op_create	リダクションオペレーションを引き起こす。
MPI_Allreduce	リデュースし、それからブロードキャストする。
MPI_Reduce_scatter	リデュースし、それから分散させる。
MPI_Scan	プレフィックスリダクションを実行する。

集団操作は、多くの１対１のメッセージからなっている。そしてそれは、オペレーションおよび内部のアルゴリズムに依存するが、多かれ少なかれ同時に起こり、与えられたコミュニケータのすべてのプロセスが関与する。すべてのプロセスは同じ MPI 集合的ルーチンを呼ばなければならない。大部分の集団操作は、broadcast,gather,scatter,reduce の４つの原始関数の変形及び (または) 結合である。

2.6.1 ブロードキャスト

```
MPI_Bcast (void *buf, int count, MPI_Datatype
     dtype, int root, MPI_Comm comm);
```

broadcast オペレーションにおいて、すべてのプロセスは同じルートプロセスを指定し、そのバッファの中身が送られる。ルート以外のプロセスは受信バッファを指定する。オペレーション後、すべてのバッファはルートプロセスからのメッセージを含んでいる。

2.6.2 スキャッタ

```
MPI_Scatter (void *sendbuf, int sendcount,
     MPI_Datatype sendtype, void *recvbuf,
     int recvcount, MPI_Datatype recvtype,
     int root, MPI_Comm comm);
```

MPI_Scatter() もまた 1 対多の集団操作である。すべてのプロセスは同一の受信数を指定する。send の引数は、ルートプロセスにだけ有効である。実際にそのバッファは、与えられたデータ型

の sendcount*N 個の要素を含んでいる。ここで、N は与えられたコミュニケータにおけるプロセスの数である。send バッファは均等に分割され、そして、自分自身を含むすべてのプロセスに分散される。このルーチンが呼ばれると、ランク順に各プロセスへ sendcount で示す数の要素を送信する。　ランク 0 は send バッファから最初の sendcount 数の要素を受け取る。ランク 1 は、send バッファから 2 番目の sendcount 要素を受け取る。ランク 2、ランク 3 なども同様にそれぞれの要素を受け取る。

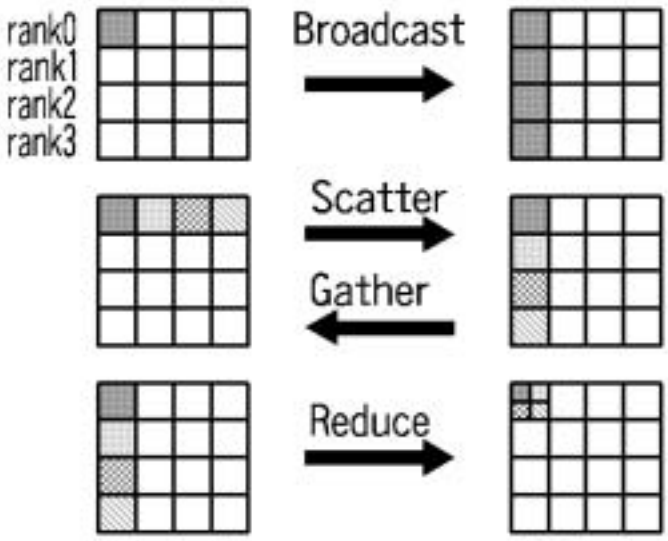

Figure 2.4: 単純な集団操作

2.6.3　ギャザ

```
MPI_Gather (void *sendbuf, int sendcount,
     MPI_Datatype sendtype, void *recvbuf,
     int recvcount, MPI_Datatype recvtype,
     int root, MPI_Comm comm);
```

MPI_Gather() は多対一の集団操作であり、そして、MPI_Scatter() の記述の完全に逆の動作を行う。

2.6.4　リデュース

```
MPI_Reduce (void *sendbuf, void *recvbuf,
     int count, MPI_Datatype dtype, MPI_Op op,
     int root, MPI_Comm comm);
```

MPI_Reduce() も、多対一の集団操作である。すべてのプロセスは同一の count 数とリダクション操作を指定する。リダクション操作後、すべてのプロセスは各々の send バッファから count 数の要素をルートプロセスに送る。対応する send バッファの場所から送られた要素は、2 つずつで組み合わせられルートプロセスのレシーブバッファにおいて単一の要素となる。すべてのプロセ

スに対する完全なリダクションは、常に結合的であるか、可換ではないかもしれない。特定用途向けのリダクション操作は runtime で定義されうる。MPI は、いくつかの事前に定義された操作を提供している。それらは全て可換である。それらは、適切な既定義のデータ型でのみ使用可能である。

MPI_MAX	最大値
MPI_MIN	最小値
MPI_SUM	合計
MPI_PROD	内積
MPI_LAND	論理積
MPI_BAND	ビット演算の積
MPI_LOR	論理和
MPI_BOR	ビット演算の和
MPI_LXOR	排他的論理和
MPI_BXOR	ビット演算の排他的論理和

以下のコードでは、静的に仕切られた規則的なデータ領域（例えば１次元配列）のコンテキストに対する基本的な集団操作の例を説明する。グローバル領域の情報は、最初、ルートプロセス（例えばランク 0）によって獲得され、すべての他のプロセスにブロードキャストされる。初期のデータセットも、ルートによって獲得され、すべてのプロセスに分散される。計算段階の後に、全体の中での最大値は、ルートプロセスに戻される。その後、新しいデータセットもルートに戻される。

```
/*
 * シングルコントロールプロセスによる並列プログラミング
 */
   int              root;
   int              rank, size;
   int              i;
   int              full_domain_length;
   int              sub_domain_length;
   double           *full_domain, *sub_domain;
   double           local_max, global_max;
   root = 0;
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
/*
 *ルートが全領域を得、その長さをブロードキャストする。
 */
      if (rank == root) {
            get_full_domain(&full_domain,
                  &full_domain_length);
      }
      MPI_Bcast(&full_domain_length, 1 MPI_INT, root,
            MPI_COMM_WORLD);
/*
 *初期のデータセットを配給する。
 */
      sub_domain_length = full_domain_length / size;
      sub_domain = (double *) malloc(sub_domain_length *
            sizeof(double));
      MPI_Scatter(full_domain, sub_domain_length,
            MPI_DOUBLE, sub_domain,sub_domain_length,
            MPI_DOUBLE, root, MPI_COMM_WORLD);
/*
 * 新しいデータセットを計算する。
 */
```

```
      compute(sub_domain, sub_domain_length, &local_max);
/*
 *ローカルの各最大値をグローバルの最大値に結合する。
 */
      MPI_Reduce(&local_max, &global_max, 1, MPI_DOUBLE,
            MPI_MAX,root, MPI_COMM_WORLD);
/*
 *新しいデータセットを収集する。
 */
      MPI_Gather(sub_domain, sub_domain_length, MPI_DOUBLE,
            full_domain, sub_domain_length, MPI_DOUBLE,
            root, MPI_COMM_WORLD);
```

2.7 コミュニケータの生成

MPI_Comm_dup	新しいコンテキスト用にコミュニケータをコピーする。
MPI_Comm_split	分類された sub_group に分割する。
MPI_Comm_compare	2 つのコミュニケータを比較する。
MPI_Comm_create	指定された group のプロセスにより新たなコミュニケータを生成する。
MPI_Comm_free	コミュニケータを解放する。
MPI_Comm_test_inter	グループ内、グループ間のコミュニケータをテストする。
MPI_Comm_remote_size	相互コミュニケータのリモートグループサイズを数える。
MPI_Intercomm_create	相互コミュニケータを生成する。
MPI_Intercomm_merge	相互コミュニケータから内部コミュニケータを生成する。
MPI_Group_size	グループ内のプロセス数を得る。
MPI_Group_rank	自プロセスのランクを得る。
MPI_Group_translate_ranks	グループＡでのランクをグループＢでのランクに変換する。
MPI_Group_compare	2 つのグループ間で、メンバー構成を比較する。
MPI_Comm_group	コミュニケータから、グループハンドルを得る。
MPI_Group_union	2 つのグループの全メンバーで、新規のグループを生成する。
MPI_Group_intersection	２つのグループの共通メンバーで、新規のグループを生成する。
MPI_Group_difference	2 つのグループ間で異なるメンバーから、新規のグループを生成する。
MPI_Group_incl	旧グループの指定されたメンバーからなるグループを生成する。
MPI_Group_excl	旧グループの指定されたメンバー以外からなるグループを生成する。
MPI_Group_range_incl	旧グループの指定された範囲のメンバーからなるグループを生成する。
MPI_Group_range_excl	range_incl の補群をつくる。
MPI_Group_free	グループオブジェクトを解放する。

コミュニケータは簡単にいえばプロセスのグループと考えることができる。その生成は同期的であり、そしてそのメンバ構成は静的である。コミュニケータが生成されたが、その全てのメンバがまだ参加していない期間は、ユーザコード上では存在しない。これらの特性により、コミュニケータは信頼できる並列プログラミングの基礎となる。２つのコミュニケータは、ユーザーコードが最初に呼ばれる前に組み立てられる。すなわち、MPI_COMM_WORLD と MPI_COMM_SELF である。参照「基本的概念」

コミュニケータは、コンテキストと呼ばれる隠された同期変数を持っている。もし、２つのプロセス間で、ソースランクとディスティネーションランク、メッセージタグが一致しても、コミュニケータが異なるなら、同期しない。この付加された同期によって、世界的なソフトウェア産業がタグの値を分配したり、割り当てたり、予約したりする必要はなくなる。[2]ライブラリやアプリケーションのモジュールを書く時、新しいコミュニケータを作って結果としてプライベート同期空間を作ることは良い方法である。この目的のための最も簡単な MPI ルーチンは MPI_Comm_dup() である。それは、コミュニケータ内のすべて、特に、グループのメンバ構成をコピーし新しいコンテキストの割り当てを行う。

```
MPI_Comm_dup (MPI_comm comm, MPI_comm *newcomm);
```

アプリケーションは、多くのサブグループに分割することが望ましい。それは時にはデータ並列の容易性のためであったり (すなわち、行列の列)、機能のグループ分けであったりする。(すなわち、データフローアーキテクチャにおける複数の異なるプログラム) group の構成メンバは、コミュニケータから引き出すことができる、そして、そのグループ全ての MPI ルーチンによって操作される。その新しいグループは、新しいコミュニケータを作るために用いられる。MPI はまた、強力なルーチン、 MPI_Comm_split() を提供している。それは、１つのコミュニケータから始めて、１つまたはそれ以上の新しいコミュニケータを生成する。それは、グループの分割とコミュニケータ生成を結合したルーチンであり、多くの一般的なアプリケーションの要求に対して十分な機能を有している。

```
MPI_Comm_split (MPI_comm comm, int color,
     int key, MPI_Comm *newcomm);
```

引数 color 及び key はグループ分けを誘導するために使われる。color の値ごとに１個の新しいコミュニケータが生成される。color に対して同じ値を供給するプロセスは、同じコミュニケータにグループ分けされるだろう。新しいコミュニケータにおけるそれらのランクは、引数 key の順序によって決められる。key の最も低い値は、ランク０になるであろう。同じ key の値を持つランク同志は古いコミュニケータにおけるランクによって順序付けされる。単に、全プロセスが同じ key を使う事で古いコミュニケータにおける相対順序を保存することができる。

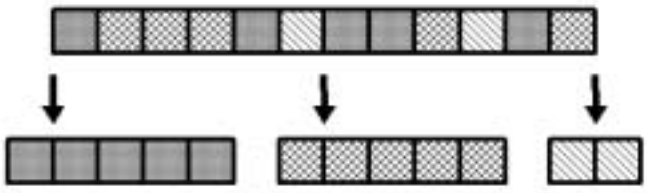

Figure 2.5: コミュニケータ分割

[2]訳注：IP アドレスや Ethernet の MAC アドレスのような分配のための組織がなくても、安全なソフトウエアが作れる。

2.8 プロセス・トポロジー

MPI_Cart_create	カルテシアントポロジー情報を含むコミュニケータを作成する。
MPI_Dims_create	バランスのとれた次元の範囲を示す。
MPI_Cart_rank	カルテシアン座標からランクを得る。
MPI_Cart_coords	ランクからカルテシアン座標を得る。
MPI_Cart_shift	カルテシアンシフトのためのランクを決定する。
MPI_Cart_sub	カルテシアンコミュニケータを低次元のサブグリッドに分割する。
MPI_Graph_create	グラフトポロジー情報を含むコミュニケータを生成する。
MPI_Topo_test	コミュニケータの仮想トポロジーの型を得る。
MPI_Graphdims_get	グラフトポロジーのエッジ数とノード数を得る。
MPI_Graph_get	グラフトポロジーのエッジとノードを得る。
MPI_Cartdim_get	カルテシアントポロジーの次元数を得る。
MPI_Cart_get	カルテシアントポロジーの配列と、周期性と、カルテシアン構造内のローカル座標を得る。
MPI_Graph_neighbors_count	グラフトポロジー内の隣接プロセス数を得る。
MPI_Graph_neighbors	グラフトポロジー内での隣接プロセスのランクを得る。
MPI_Cart_map	最適なカルテシアン配置での新しいランクを提案する。
MPI_Graph_map	最適なグラフ配置での新しいランクを提案する。

MPI は、実装される並列コンピュータのノードとは独立した プロセス志向のプログラミングモデルである。それにもかかわらず、パフォーマンスを向上するためには、並列アプリケーションにおいてデータの転送 (動き) のパターンとハードウェアのコミュニケーショントポロジーをできるだけ一致させるべきである。コンパイラーやメッセージパッシングシステムにとって、アプリケーションのデータの動きを推測する事は困難であるので、MPI の実装が、プロセスを特定する際に最適化の情報として使うであろうという事を期待して、MPI はコミュニケータにトポロジを提供する事をアプリケーションに許している。例えば、もしアプリケーションがカルテシアンコミュニケーションを指向してていてかつ、並列コンピュータがカルテシアントポロジーを持つならば、任意のデータ座標を任意のノード座標に盲目的に配置するよりも、マシンの特性に合わせてデータを整列させる方が好ましい。MPI は、偏在するカルテシアングリッド、および任意のグラフの２種類のトポロジーを提供している。トポロジーの情報は、新しいコミュニケータを作る時にコミュニケータにつけられる。MPI_Cart_creat() はカルテシアントポロジーのための、コミュニケータ生成ルーチンである。

```
MPI_Cart_create (MPI_Comm oldcomm, int ndims,
     int *dims, int *periods, int reorder,
     MPI_Comm *newcomm);
```

カルテシアン配列にとって不可欠な情報は、次元の数、それぞれの次元の長さおよび、ラップアラウンドするか否かを示すそれぞれの次元の周期性フラグである。引数 reorder は、アプリケーションが新しいトポロジーコミュニケータにおいて異なったランク付けを許しているかどうかを

示すフラグである。並び替えは、MPI の実装にとって座標計算をより簡単なものにするであろう。トポロジーによって拡張されたコミュニケータでは、アプリケーションはソースと宛先ランクを決めるため座標を使用するだろう。MPI コミュニケーションルーチンがランクをまだ使用しているので、座標はランクに、またランクは座標に変換されなければならない。MPI はこの変換を、MPI_Cart_rank() と MPI_Cart_coords() の提供により容易にしている。

```
MPI_Cart_rank (MPI_comm comm, int *coords,
     int *rank);
MPI_Cart_coords (MPI_Comm comm, int rank,
     int maxdims, int *coords);
```

カルテシアントポロジーのアプリケーションがさらにプロセスを識別しやすくするために、MPI_Cart_shift() は、一般的な隣接座標間シフト通信に対応するランクを返す。方向（次元）および相対的な距離は、入力引数であり、２つのランクは出力変数である。そして、出力されるランクは与えられた方向にそった呼出プロセスの両側にそれぞれ対応する。与えられたコミュニケータにおけるカルテシアントポロジーの周期性に依存するが、片方もしくは両方のランクの値としてグリッドのはじから落ちたことを示す、 MPI_PROC_NULL が返されるかもしれない。

```
MPI_Cart_shift (MPI_Comm comm, int direction,
     int distance, int *rank_source,
     *int rank_dest);
```

2 次元のカルテシアンデータセットを考える。以下のコードの概略は、任意のプロセス数に対応するプロセストポロジーを設定している。それから、集団操作のため第一列のプロセスからなる新しいコミュニケータを作る。最後に、それは、データ交換のために前、および次の列を保持するプロセスのランクを得る。

```
 int             mycoords[2];
 int             dims[2];
 int             periods[2] = {1, 0};
 int             rank_prev, rank_next;
 int             size;
 MPI_Comm        comm_cart;
 MPI_Comm        comm_col1;
/*
 * Create communicator with 2D grid topology.
 */
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      MPI_Dims_create(size, 2, dims);
      MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1,
            &comm_cart);
/*
 * Get local coordinates.
 */
      MPI_Comm_rank(comm_cart, &rank);
      MPI_Cart_coords(comm_cart, rank, 2, mycoords);
/*
 * Build new communicator on first column.
 */
      if (mycoords[1] == 0) {
            MPI_Comm_split(comm_cart, 0,mycoords[0],
                  &comm_col1);
      } else {
            MPI_Comm_split(comm_cart, MPI_UNDEFINED, 0,
                  &comm_col1);
}
```

```
/*
 * Get the ranks of the next and previous rows, same column.
 */
      MPI_Cart_shift(comm_cart, 0, 1, &rank_prev,
            &rank_next);
```

MPI_Dims_create() は、与えられたノード及び次元の数に対して最もバランスのとれた正方に近い次元領域を提供する。グリッドの部分集合上のコミュニケータを作る事の利点は (この場合、メッシュの最初の列)、集団操作の使用を可能にすることである。「集団メッセージパッシング」を参照 。

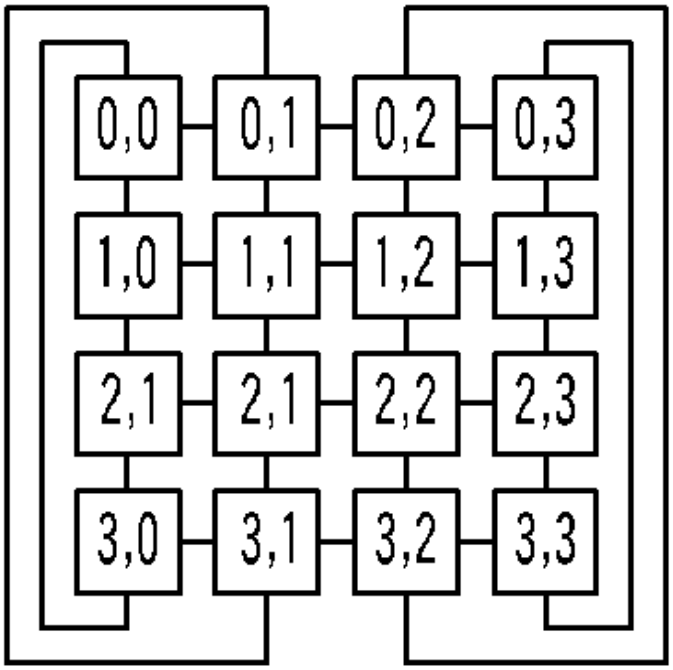

Figure 2.6: カルテシアントポロジー

2.9 その他の MPI の特徴

MPI_Errhandler_create	特製のエラーハンドラを生成する。
MPI_Errhandler_set	コミュニケータにエラーハンドラを設定する。
MPI_Error_string	エラーコードの種類を得る。
MPI_Error_class	エラーコードのクラスを得る。
MPI_Abort	アプリケーションを異常終了させる。
MPI_Wtime	wall clock time を得る。
MPI_Errhandler_get	コミュニケータがエラーハンドラを得る。
MPI_Errhandler_free	特製エラーハンドラを解放する。
MPI_Get_processor_name	プロセッサの名前を得る。
MPI_Wtick	wall clock timer の解像度を得る。
MPI_Keyval_create	属性キーを生成する。
MPI_Keyval_free	属性キーを解放する。
MPI_Attr_put	コミュニケータの属性をキャッシュする。
MPI_Attr_get	キャッシュした属性を得る。
MPI_Attr_delete	キャッシュした属性を破棄する。

2.9.1 エラー・ハンドリング

エラーハンドラーは、ある MPI オペレーションの間にエラーが起こる時に呼ばれるソフトウェアルーチンである。ハンドラーは、コミュニケータに付随するものであり、そのコミュニケータから作られたコミュニケータに受け継がれる。あるコミュニケータを使用する MPI ルーチンにおいてエラーが起きた時、そのコミュニケータのエラーハンドラーが呼ばれる。アプリケーションの初期のコミュニケータ、MPI_COMM_WORLD は既定の組み込みハンドラー MPI_ERRORS_ARE_FATAL を持つ。それは、そのコミュニケータのすべてのタスクを終了させる。アプリケーションはまずユーザールーチンから MPI エラーハンドラーオブジェクトをつくる事によってエラーハンドラーを供給できる。

```
MPI_Errhandler_create (void (*function)(),
     MPI_Errhandler *errhandler);
```

エラーハンドラールーチンは、ANSI C < stdargs.h > 構造を使用して、２つの定義済パラメータに続いて実装依存のパラメータを有する。最初のパラメータは ハンドラーのコミュニケータであり、２番目は、問題を記述しているエラーコードである。

```
void function (MPI_Comm *comm, int *code, ...);
```

エラーハンドラーオブジェクトは、その後 MPI_Errhandler_set() によってコミュニケータと関連付けられる。

```
MPI_Errhandler_set (MPI_Comm comm,
     MPI_Errhandler errhandler);
```

２番目の組み込みエラーハンドラーは、 MPI_ERRORS_RETURN で、何もせず、それがテストされて、実行するかもしれないまちがった、 MPI ルーチンによってエラーコードが返されること

を可能にしている。C で、エラーコードは MPI 関数の返り値である。Fortran 言語において、エラーコードはエラーパラメーターをによって MPI サブルーチンに戻される。

```
MPI_Error_string (int code, char *errstring,
     int *resultlen);
```

エラーコードは MPI_Error_string() により記述的な文字列に変えられる。ユーザーは、最小限の MPI_MAX_ERROR_STRING の長さの文字列のためのスペースを提供しなけらばならない。戻された文字列の実際の長さは、resultlen 引数によって返される MPI は、ポータブルなアプリケーションによりテスト、実行される一連の標準的なエラーコード (また、エラークラスと呼ばれる) を定義している。実装に特有なすべての特別なエラーコードは、標準エラーコードの一つに割当てられる。これは特別なエラーコードは、標準コードの１つの変形または、同じエラークラスのメンバーであるという考えによるものである。2 つの標準的なエラーコードが、この目的に合っていないすべてのエラーコードに対して用意されている。すなわち、MPI_ERR_OTHER と MPI_ERR_UNKNOWN である。再度強調するが、この設計の目的は移植可能でインテリジェントなアプリケーションを記述可能とすることである。標準のエラーコード（クラス）へのエラーコードのマッピングは、MPI_Error_class() によって行われる。

```
MPI_Error_class (int code, int class);
```

2.9.2 タイミング

過去のある時点からの、経過時間を返す MPI_Wtime() が性能測定のために用意される。

```
double MPI_Wtime (void);
```

Chapter 3

LAM ／ MPI 拡張

LAM はソフトウェアアプリケーションの開発段階においてプログラマにとって役立つ MPI 標準外のいくつかの機能を含んでいる。
それらは最終製品に使われることができるが、明らかに移植性が損なわれるだろう。実際に拡張部分の 1 つは、他の MPI 実装においても動作することができる MPI ポータブルライブラリ（集団 I/O を参照）である。
このライブラリは LAM とは別の物でであって、別途入手してコンパイルする必要がある。他の拡張部分はすべて LAM 固有のものである。
MPI と整合性の高い拡張ルーチンのいくつかは MPIL_から始まる名前を持っている。同様な機能は、MPI 標準の後のバージョンで見つけられることもあるだろう。MPI の概念やオブジェクトとは離れている他のルーチンは lam_から始まる名前を持つ。

3.1 リモートファイルアクセス

lam_rfopen	ファイルをオープンする。
lam_rfclose	ファイルをクローズする。
lam_rfread	ファイルから読む。
lam_rfwrite	ファイルに書き込む。
lam_rflseek	ファイルの中のポジションを変更する。
lam_rfaccess	ファイルの許可を検査する。
lam_rfmkdir	ディレクトリを作成する。
lam_rfchdir	作業用のディレクトリを変更する。
lam_rffstat	ファイル記述子でステータスを得る。
lam_rfstat	名前付きのファイルでステータスを得る。
lam_rfdup	ファイル記述子を複製する。
lam_rfdup2	ファイル記述子を複製＆置く。
lam_rfsystem	シェルコマンドを発行する。
lam_rfrmdir	ディレクトリを削除する。
lam_rfunlink	ファイルを削除する。
lam_rfgetwd	作業用のディレクトリを得る。
lam_rfftruncate	ファイル記述子の長さを設定する。
lam_rftruncate	名前付きのファイルの長さを設定する。

ノードのファイルシステムには POSIX ライクなインタフェースを持つリモートファイル関数を経由してアクセスできる。LAM はファイルシステムを用意するという訳ではなく、任意のノードからファイルシステムへのリモートアクセスを提供するだけである。

デフォルトでは、ファイルのパスネームはプログラム起動ノード上のファイルを参照する。しかしながら特定の ノード識別子は、次の構文によってパスネームに取り付けられ得る：

```
nodeid:path
```

各々の LAM プロセスは同時オープン数が制限されている LAM ファイル記述子を持つことができる。メッセージ通信を行う全ての LAM ファイル関数は一般のアプリケーションと同じリンク、バッファや他のリソースを使用する。

LAM は入力のための遅いデバイス（ターミナルなどの）を開くことを禁止する。

引数におけるリモートファイルアクセスのいくつかの LAM 特有の特徴は、lam_rfopen() ルーチンのフラグ引数における追加フラグよって制御される。これらのフラグは、下に列挙されている。

LAM_O_LOCK

リモートファイルサーバのオープンディスクリプタキャッシュにファイル記述子をロッ

クする。マニュアルのページ lam_rfposix() を参照。

LAM_O_REUSE

もし見つけられるならば、パスネームとオープンフラグが一致する現在開いているファイル記述子を再利用する。これは１つのファイルポインタによってオープン中ファイルへ非同期アクセスをする場合に有用である。

LAM_O_1WAY

完了またはリターンコードの待ち合わせをせずファイルに書き込みを行う。これは書込み性能を大いに増加させるが、充分デバッグされたアプリケーションだけで利用されるべきである。[1]

3.1.1　移植性と標準 I/O

LAM はネーティブオペレーティングシステムのファイルインタフェースに干渉しないし、リダイレクトもしない。従って、 open() は（LAM が関与しない）直接的な UNIX ルーチンであると言える。フォーマットされたストリームパッケージである stdio は、同じ論をたどる。(即ち LAM からは使用できない)

その例として、LAM はネーティブな利用または LAM ファイルアクセスのために、stdio の機能に拡張機能を加えた tstdio と呼ぶ別のパッケージを用意する。標準出力とエラーファイルは LAM モードの中にあるので、アウトプットがリモートノード上のユーザのターミナルに転送される。新しいファイルはデフォルトではネーティブモードが開かれるが、特殊なオプション文字‘ T ’を設定することで LAM モードを選ぶことができる。tstdio を使用するために、stdio 関数に文字‘ t ’を付加する。

特に複数のノードを渡るアウトプット文を用いてデバッグする場合には、lam_rfwrite() か tprintf() などを使用しなければならない。ネーティブの標準 I/O ディスクリプタは起点ノードを除く全てのノードにおいて /dev/null に連結される。

LAM 標準入力も/dev/null に接続される。アウトプットとエラーファイルは親プロセスから引き継がれる。これらのファイルは、 mpirun が呼び出されていたターミナルかウィンドウに接続する。

[1]訳注:本番のプログラムにおいて各種の安全装置をはずして高速化を行うことは一般的には薦められない行為であろう。

3.2 集団 I/O

CBX_Open	MPI Cubix アクセスのためにファイルをオープンする。
CBX_Close	MPI Cubix ファイルをクローズする。
CBX_Read	単一または複数のモードで読み取る。
CBX_Write	単一または複数のモードで書き込む。
CBX_Lseek	単一または複数のモードでシークする。
CBX_Order	マルチプルアクセスの命令を変更する。
CBX_Singl	単一のモードへファイルアクセスを変える。
CBX_Multi	複数のモードへファイルアクセスを変える。
CBX_Is_singl	単一のモードのファイルであるか？
CBX_Is_multi	複数のモードのファイルであるか？

緩く同期式な集団 I/O ライブラリ MPICubix は、カリフォルニア工科大学における同じ名前の研究開発に基づいている。この Cubix は MPI コミュニケータとデータ型との概念によって統合された。コミュニケータグループのプロセスが、入出力操作に集団的に参加する。データは、MPI メッセージ通信と同様にデータ型の要素の数に従って転送される。

全てのファイルアクセスルーチンは、最終的に単一ファイル上の POSIX オペレーションに変換される。コミュニケータグループの中の１つのプロセスだけが、実際の POSIX オペレーションを呼び出す。POSIX ファイルオペレーションの返り値は MPI オブジェクトで調節され MPI Cubix ルーチンの返り値に反映される。

データパラレルプログラミングにおいて、一般的な２つのファイルの読み／書き方法に対応する２つの異なる MPI Cubix アクセス方式がある。

シングル

> 全てのプロセスは、同一データ、同一転送量の同一ファイルルーチンを実行する。ただ１つのプロセスのデータだけが転送される。全てのプロセスがファイルから大域の値を読取る場合や、ファイルに大域の値を書き込みたい時にこれは役立つ。それはターミナルへの出力時に特に便利である。全てのノードがエラー・メッセージをプリントするが、それは１度だけターミナルに現れる。

マルチプル

> 全てのプロセスは、異なるデータの異なる量をもって同じファイルルーチンを実行する。全てのプロセスからの全てのデータが転送される。しかし転送の順序は、厳密に制御される。デフォルトでは、プロセスランク 0 は最初に転送する。そして最も高いランクを最後に転送するまで順次続けていく。これは正しいノードがデータの正しい部分集合を得るように、読み込みの間データ構造を分解すること、そしてデータ構造がごたまぜにされないように、書き込みの間にデータ構造を再構成することに役立つ。

Cubix ファイルアクセスがなければ、アプリケーションは並列のプロセスの管理とＩ／Ｏのフィルタリングのためにコントロールプログラムが必要となるだろう。Cubix によって制御プログラ

ムの必要性を除去することができる。同期制御がなければ、N ノードにより書かれたメッセージは、ターミナル上 N 回表われる。分解されたデータ構造は、ランダムな順序でファイルに書込まれる。

MPI Cubix ファイル記述子は LAM リモートのファイル記述子や純粋なオペレーティングシステムのファイル記述子とは区別される。MPI Cubix ファイル記述子は、CBX_Open() によって得られる。アクセス方式は、特殊なフラグ CBX_O_SINGL か CBX_O_MULTI のどちらかによって選ばれる。ファイルのオーナ（ POSIX レベルの操作を行う 1 つのプロセス）は CBX_Open() の他の引数によって選ばれる。

```
#include ＜ fcntl.h ＞
#include ＜ cbx.h ＞
int CBX_Open (const char *name, int flags,
     int mode, int owner, MPI_Comm comm);
int CBX_Close (int fd) ;}
```

開いている MPI Cubix ファイルにおいて利用されてるアクセス方式は、いつでも問い合わせ、変更もできる。変更ルーチンは集団操作である。問合せルーチンはそうではない。

```
int CBX_Multi (int fd);
int CBX_Singl (int fd);
int CBX_Is_multi (int fd);
int CBX_Is_singl (int fd);
```

CBX_Read() と CBX_Write() は開いている MPI Cubix ファイルとの間でデータを転送を行う。MPI データ型は引数の一つである。データバッファの長さは、与えられたデータ型の要素の総数である。連続データだけが転送される。もし MPI データ型にすき間があるならば、それらも合わせて転送される。

```
int CBX_Read (int fd, void *buffer, int count,
     MPI_Datatype dtype);
int CBX_Write (int fd, void *buffer. int count,
     MPI_Datatype dtype);
```

CBX_Order() ルーチンは、MPI Cubix マルチプルアクセス方法のプロセスデータ転送のデフォルト順序を変更する。各々のプロセスは 0 から N-1 までの重ならないシーケンス番号を指定する。ここで、N はコミュニケータのサイズである。

```
int CBX_Order (int fd, int newrank);
```

3.2.1 キュービックス例

```
/*
 * 実数の 1 次元配列を読取ってプロセスの 1 次元配列上
 * に分解する。
 * 最初にシングルモードで配列サイズを読み、それから
```

```
 * マルチプルモードで配列を読む。
 * ここで配列の長さが一様に分解されると仮定する。
 */
static float *data;
main (argc, argv)
int argc;
char *argv[];
{
       int fd;
       int glob_len, local_len;
       int nread;
       int size;
       MPI_Init(&argc, &argv) ;
/*
 * 最初に Cubix 単一方法でファイルをオープンする。
 * ファイルは、プロセスランク 0 によって所有される。
 * これはエラー処理のチュートリアルではないので、
 * オープンエラーは想定しない。
 */
       fd = CBX_Open("data", O_RDONLY | CBX_O_SINGL, 0, 0,
             MPI_COMM_WORLD) ;
/*
 * 配列の大域（合計）の長さを読む。
 */
     CBX_Read(fd, &glob_len, 1, MPI_INT);

/*
 * Cubix 複数方法へ切り替る。
 */
     CBX_Multi(fd) ;

/*
 * ローカルの長さを計算して、十分なスペースを割り当て
 * データのローカルの部分集合を読み込む。
 */
     MPI_Comm size(MPI_COMM_WORLD, &size) ;
     local_len = glob_len / size;
     data = (float *) malloc(local_len * sizeof(float));
     CBX_Read(fd, data, local_len, MPI_FLOAT) ;
     CBX_Close (fd) ;
     MPI_Finalize ();
}
```

3.3 プロセス生成

MPIL_Spawn	プロセスのグループを生成する。
MPIL_Comm_parent	親グループ間コミュニケータを得る。
MPIL_Universe_size	LAM ノードの数を得る。

プロセスグループの静的性質は MPI の長所の一つであるからプロセス生成は、慎重にされなければならない。プロセス生成は、与えられたコミュニケータの上の集団操作である。プロセスの 1 グループが、 MPIL_Spawn() への 1 回の呼出しによって作成される。子プロセスは従来の MPI の方法でスタートアップ、初期化、通信を行う。それらは MPI_Init() を呼び出すことによって始まらなければならない。子グループは、親グループの world コミュニケータとは別の独自の MPI_COMM_WORLD を持つ。

```
MPIL Spawn (MPI\_Comm comm, char *app,
     int root, MPI\_Comm *child\_comm);
```

親は、どのようにそれらの子と通信をするのだろう？ 2 グループ間の通信のための自然なメカニズムは、グループ間コミュニケータである。リモートの子プロセスのグループに対応するグループ間コミュニケータは、MPIL_Spawn() の 2 番目コミュニケータ引数で親に返される。子は、MPIL_Comm parent() を呼び出すことによってリモートの親プロセスのグループに対応するミラーコミュニケータを得る。もし子が生まれなかったならば、このルーチンはヌルコミュニケータを返す。したがって、プログラムはオリジナルのアプリケーションプロセスか生まれた子プロセスとして動作するようにコード化されることができる。

```
MPIL_Comm_parent (MPI_Comm *comm);
```

3.3.1 リソース仕様

生成されたプロセスの構成、それらの数、および割り当てられた計算リソースは、mpirun によって利用されたスキーマと同じアプリケーションスキーマによって指定される。「MPI プログラムの実行」を参照。ちょうど単純な SPMD アプリケーションが mpirun コマンドラインからスタートされるように、アプリケーション引数でスキーマファイルの代わりに単純なアプリケーションが MPIL_Spawn() に直接指定することができる。例えば：

”appfile”	単一のファイル名は、アプリケーションスキーマを示す。
”a.out -c 1”	プロセスカウント (-c 1) は”a.out”が実行可能なファイルネームであるということを示す。
”a.out n0-3”	ノード指定は”a.out”が実行可能なファイルネームであるということを示す。

MPIL_Spawn() は環境変数 LAMAPPLDIR によって指定されたディレクトリ、そして呼び出すプロセスの現在の作業用ディレクトリ、そして LAM インストレーションディレクトリのアプリケーションスキーマをサーチする。実行可能なファイルは、 PATH 環境変数で指定されたディレクトリ

をサーチすることによって見つけられる。mpirun から MPI アプリケーションに渡された実行時オプションが、生まれたプロセスに伝播される。もし使用可能であるならばこれはトレース収集のための特に重要な結果を与える。「デバッギングおよびトレーシング」、「トレースデータの収集」を参照。生成されたプロセスグループは、それぞれ新しい world コミュニケータであり、そしてそのことにより単一の world グループが選ばれなければならない時、トレースデータの取り出しを複雑にする。

3.3.2　耐故障性

もしプロセスが動いているノードが故障したり、LAM マルチコンピュータから取り去られるならば、コミュニケータは不正な状態になり得る。MPI ライブラリは死んだコミュニケータを使用する試みにおいてエラー状態を引き起こすだろう。これは進行中の要求が使用中のコミュニケータが突然無効になる場合を含む。これらの障害は、 MPI_ERRORS_RETURN （MPI のその他の特徴を参照）にコミュニケータのエラーハンドラを設定することによってアプリケーションレベルで検出することができる。
ある程度の耐故障性を有するマスタースレーブアプリケーションは、次の戦略によって構成できる：

- １つのグループでプロセスを生み出す。
- MPI_ERRORS_RETURN に親／子グループ間コミュニケータのためのエラーハンドラを設定する。MPI のその他の特徴を参照。
- もし子との通信がエラーを返すならば、それが死んだものと仮定してグループ間コミュニケータを解放する。
- もし死んだプロセスを取り替えることが要求されるならば、他のプロセスを生成する。

3.3.3　リソース仕様

アプリケーションによって生成されるプロセスの数は、利用できるプロセッサの数によってしばしば制限される。LAM においてはノードの数がこの情報の基礎であって、MPIL_Universe_size() によって返される。

```
MPIL_Universe_size (int *size);
```

3.4 シグナル・ハンドリング

lam_ksignal	シグナルハンドラをインストールする。
lam_ksigblock	シグナルを選択したブロック。
lam_ksigsetmask	全体のブロッキングマスクを設定する。
lam_ksigretry	シグナルを選択した後のリトライリクエスト。
lam_ksigsetretry	全体のリトライマスクを設定した。
lam_ksigmask	シグナルマスクを作成する。
MPIL_Signal	プロセスにシグナルを引き渡す

LAM は UNIX ライクなグナルパッケージを提供する。シグナルは実装されるオペレーティングシステムのものとは異なっており、お互いにに干渉しない。

いくつかのシグナルは、システムによって内部で利用される。また、いくつかのシグナルは役立つデフォルトオプションを持ち、他はユーザに完全に任せられる。最も便利なシグナルはプロセスにそれ自身を強制的に終了させるシグナルである。シグナルは< lam_ksignal.h >で定義される。

LAM_SIGTRACE	トレースデータをアンロードする
LAM_SIGUDIE	終了する
LAM_SIGARREST	実行を延期する
LAM_SIGRELEASE	実行を再開する
LAM_SIGA	ユーザ定義（デフォルトは無視する）
LAM_SIGB	ユーザ定義（デフォルトは無視する）
LAM_SIGFUSE	ノードが死にそうである
LAM_SIGSHRlNK	他の一つのノードが死んだ

lam_ksignal() と Iam_ksigblock() と lam_ksigsetmask() 関数はそれら UNIX において相当する関数と同じ様に動作する。完了前に、シグナルによって中断された LAM や MPI ルーチンは自動的に再実行される。それぞれ lam_ksigblock() と lam_ksigsetmask() と同様に動作する lam_ksigretry() と lam_ksigsetretry() 関数を使用するとユーザは、自動的なシステム・コールの再実行を禁止し、その代わりにエラーコードを受け取ることができる。

3.4.1 シグナル転送

MPIL_Signal() はコミュニケータとランクによって識別されたプロセスにシグナルを送る。シグナル番号引数は、上で定義されたリストの値をとる。

```
MPIL Signal (MPI Comm comm, int rank, int signo);
```

3.5　デバッギングおよびトレーシング

MPIL_Comm_id	コミュニケータ識別子を得る。
MPIL_Comm_gps	MPI プロセスのために LAM 座標を得る。
MPIL_Type_id	データ型識別子を得る。
MPIL_Trace_on	トレース収集を可能にする。
MPIL_Trace_off	トレース収集を不能にする。

LAM は強力な監視能力を用意しており、デバッギングを非常に重視している。MPI の中の不透明オブジェクトは、LAM デバッギングツールによって得られる情報と走行中のプロセス中の値の相互参照を難しくさせる。もし mpitask（「プロセスの監視と制御」を参照）がコミュニケータでブロックされたプロセスを示すならば、それはそのコミュニケータの識別番号を出力する。その数は MPI 標準によって定義されたものではない。それは標準の API を使用して、プログラムがアクセスすることができない不透明なコミュニケータへの内部の実装依存情報である。

MPIL_Comm_id() と MPIL_Type_id() は、それぞれコミュニケータとデータ型の内部識別子を返す。

```
MPIL_Comm_id (MPI_Comm comm, int *id);
MPIL_Type_id (MPI_Comm comm, int *id);
```

lam_ 接頭部をもって始まっている LAM / MPI 拡張は、LAM 特有のものである。それらは MPI コミュニケータとランクではなく、LAM のノードとプロセス識別子によって動作する。MPIL_Comm_gps() は MPI 情報から LAM 座標を得る。

```
MPIL_Comm_gps (MPI_Comm comm, int rank, int *nid,
      int *pid);
```

性能視覚化とデバッギング目的のための実行トレース収集は、mpirun によって有効にされる。「MPI プログラムの実行」を参照。情報オーバーロードと巨大なトレースファイルを避け、計算の興味深い部分だけがモニタできるように、アプリケーションは実際のトレース収集のオンとオフが出来る。

```
MPIL_Trace_On (void) ;
MPIL_Trace_Off (void) ;
```

Chapter 4

LAM コマンドリファレンス

4.1 始める準備

4.1.1 UNIX 環境の設定

LAM を動かす前に、マルチコンピュータの個々のマシンに適当な環境変数とシェルのサーチパスを設定しなければならない。以下のコマンドまたは相当するものをシェル起動ファイル（.cshrc, C シェルという）に追加する。LAM の起動に rsh が使われる場合、.login はリモートマシン上では読み込まれないので、.login に設定してはならない。

```
setenv  LAMHOME ＜LAM インストールディレクトリ＞
set path = ($path $LAMHOME/bin)
```

ローカルなシステム管理者または LAM をインストールした本人は、LAM インストールディレクトリの位置を知っているだろう。シェル起動ファイルを編集した後、新しい値を有効にするために、以下のコマンドを呼び出す。これは UNIX システムへの次回のログインからは必要ではない。

```
% source .cshrc
```

マルチコンピュータの個々のリモートのマシンは、UNIX rsh コマンドを使えなければならない。rsh はパスワードのプロンプトではなく、リモートのマシン（／etc／hosts.equiv、および ／.rhosts）のスペシャルファイルによってアクセスを得ること。これらのファイルのうちの 1 個は、リモートのマシンにおいて選ばれたユーザアカウントを承認するために用意する必要がある。これらのファイルを準備する方法は rsh のための UNIX マニュアルページを参照。

4.1.2 ノードニーモニック

多くの LAM コマンドは少なくとも 1 個以上のノード識別子を必要とする。ノード識別子は、コマンドラインにおいて、n ＜リスト＞として指定される。ここで＜リスト＞はコンマで分離されたノード識別子または範囲表記されたノード識別子のリストである

```
n1
n1,3,5-10
```

明示的なノード識別だけでなく、LAM は、（特別なノードまたはグループのノードを参照する）特別なニーモニックを持っている。

```
h  コマンドがタイプされた（『ここ』で）ローカルなノード
o  LAM が lamboot コマンドによって始められた時の初期のノード
N  すべてのノード
C  アプリケーション計算のために意図されているすべてのノードを表す。
```

ノード識別子はLAM マルチコンピュータのブートスキーマ（「LAM ブート・スキーマ」書くを参照）と呼ばれるプランによって決定される。LAM ノード識別子は、システムが lam boot で始められる時に、いつも 0 から連続的に開始する数を割り当てられる。従って、ブートスキーマのノードの数はノード識別子の初期の集合を定義する。もしノードが付加されるか、または取り去られるならば、ノード識別子の連続している特性はそこなわれるかもしれない。「LAM ノードの追加と削除」を参照。

4.1.3 プロセスの識別

LAM プロセスは 2 つの方法で指定できる：プロセス識別子（下層のオペレーティングシステムからの）による方法または LAM プロセスインデックスによる方法。

PID は、コマンドラインにおいて、p ＜リスト＞として指定される。ここで＜リスト＞はコンマで分離または範囲表記された PID のリストである。

```
p5158
p5158,5160,5200-5210
```

プロセスインデックスは、コマンドラインにおいて、i ＜リスト＞として指定される。＜リスト＞はコンマで分離されたインデックスまたは範囲表記されたインデックスのリストである。

```
i8
i8-12,14
```

*MPI*プロセスは、通常、*MPI_COMM_WORLD* コミュニケーターにおけるグローバルなランク値を識別子として LAM ／ MPI ステータス報告コマンド、mpitask、および mpimsg によって、報告される。並列アプリケーションの実現性により、プロセスが生み出され、LAM プロセス制御コマンドとともに LAM ノード／プロセス識別子を使用する必要がある。そして補助的なラベリングスキーマが利用できる。それは GPS(Global Positioning System) として知られている。なぜなら、LAM システムにおいて他のすべてからプロセスを完全に特徴づけているからである。GPS はプロセスインデックスとノード識別子を含んでいる。

4.1.4 オンラインヘルプ

どのような LAM コマンドにおいても構文とオプションの短い要約を出力するには、-h オプションによってコマンドを実行する。

```
% recon -h
```

個々のコマンド（およびほとんどのプログラミング機能）についての詳細な情報が、オンラインマニュアルページに示される。それらはこの文書への重要な補助的リファレンスである。

```
% man recon
```

4.2 MPI プログラムのコンパイル

hcc	wrapper はローカルな C コンパイラのためのラッパ
hcp	wrapper はローカルな C++コンパイラのためのラッパ
hf77	wrapper はローカルな Fortran 言語コンパイラのためのラッパ

オブジェクトと実行形式はネイティブなコンパイラとリンカにより作られる。できれば，すべての LAM ノード、もしくは各アーキテクチャとオペレーティングシステムタイプで少なくとも１つのノードにおいて、これらのツールが使用可能である必要がある。hcc ツールは、（例えば cc のような）ネイティブの C 言語コンパイラドライバーを起動する上位コマンドであり、LAM ヘッダーファイルとライブラリのパスを提供し、すべての LAM ライブラリを暗黙にリンクする。MPI ライブラリは明示的にリンクされる。hcc に与えられたオプションは、ネイティブのコンパイラドライバーにすべて渡される。

```
% hcc -o appl appl.c -lmpi
% hcp -o appl appl.c -lmpi
```

デフォルトでは、hcc は、LAM を構築するために使用されて、LAM の構成ファイルに指定されているコンパイラドライバーを使用する。LAMHCC 環境変数に指定することによって異なる C 言語コンパイラも使用できる。C と C++コンパイラが異なる場合、C++のためのラッパ hcp が、別途与えられる。C と C++のラッパと違って、Fortran ラッパ〈hf77〉は、LAM のヘッダーフイルィのサーチパスのオプションを挿入しない。これは、すべての Fortran コンパイラドライバーがこのオプションをサポートしているわけではなく、MPI ヘッダーファイル、mpif.h を採り入れるために Fortran 言語のインクルード文を必要とするかも知れないからである。mpif.h は Fortran ソースファイルであることに注意すること。しかし、Fortran 言語用のすべての他の LAM ヘッダーファイルは、C 言語プリプロセッサーコードを含んでいる。Fortran ドライバーが自動的にプリプロセッサの処理をしないならば、C 言語プリプロセッサを、明示的に動かす必要がある。マルチコンピュータの異種のノードにおいて生み出されたオブジェクトファイルとバイナリを混同しないように気を付ける必要がある。この場合、それらを区別するためオブジェクトおよび実行ファイル名に、マシンや CPU 名を付加することは良い方法である。

```
{sparc}% hcc -o appl.sparc appl.c
{sgi}% hcc -o appl.sgi appl.c
```

4.3 LAM の起動

recon	マルチコンピュータの検証は LAM を実行するための準備。
lamboot	ＬＡＭマルチコンピュータセッションを開始する。
tping	与えられた node への通信をチェックする。
wipe	ＬＡＭセッションを終了させる。

4.3.1 recon

マルチコンピュータが LAM を実行可能か否かを検証する。LAM を動かしているマルチコンピュータのトポロジーは、完全グラフであるものとする。従って、LAM をスタートするにあたりマルチコンピュータに含めるべきマシンを指定するだけでよい。ASCII ファイルブートスキーマはこの目的にかなう。「LAM ブート・スキーマを書く」を参照。マシンがソフトウェアをリモートで実行するにあたり、いくつかの条件が必要である。

- マシンのアドレスはネットワークを経由で到達可能であること。
- ユーザーはそのマシンにおいて rsh によるリモート実行ができること。
- リモートのホスト許可は、/etc/hosts.equiv、またはリモートのユーザーの.rhosts ファイルに提供されなければならない。
- リモートのユーザのシェルは、LAM が動作可能なサーチパスを持っていなければならない。
- リモートのシェルの起動ファイルは、非対話的に起動された時に、標準エラー出力に何も出力してはならない。

4.3.2 lamboot

lamboot ツールは個々のユーザーのための LAM セッションを始める。ホストファイルの形式で書かれたブートスキーマファイルが lamboot の第一の引数である。

```
% lamboot -v <boot schema>
```

4.3.3 耐故障性

-x　障害検出と復旧の設定。すべてのノードの間で周期的な「心拍」メッセージを交換する。

LAM は、メッセージパケットの再送を何回か繰り返しても応答が無い場合、そのノードを停止したと見なす。デフォルトでは、停止したノードに対し対応せず、無限に再送信を続ける。lamboot への-x オプションによって、LAM は、停止しているノードをマルチコンピュータから取り去る手続を起動する。停止しているノード識別子は無効になる。すべての他のノード識別子は不変でありノード識別子リストには穴が生じる。最後に、すべてのノードのすべてのアプリケーション

プロセスにシグナルが送られ、故障を通知する。個々のプロセスのランタイムシステムは、LAM デーモンからの更新された情報を読むことができるように、最低限、ノード識別子のキャッシュテーブルを消去する必要がある。ユーザーはこのシグナル (LAM_SIGSHRINK) を捕まえることができる。「シグナル・ハンドリング」を参照。

4.3.4 tping

制御と監視は、LAM の特徴である。最も簡単なコマンド，tping は、並列環境のブラックボックス性を追い払うことと、信頼性のチェックを行う。tping は宛先ノード、またはマルチキャストとメッセージの応答を行う。もしこのコマンドがハングしたときは，セッションを再スタートするべきである。

```
% tping n0
% tping N
```

4.3.5 wipe

LAM セッションを終了するには、wipe ツールを使用する。システム故障の後に LAM を再スタートするには、wipe に続けて lambot を実行する。

```
% wipe -v <boot schema>
```

4.4 MPI プログラムの実行

4.4.1 mpirun

mpirun　MPI アプリケーションを実行する。
lamclean　すべての LAM プロセスの終了と後始末を行う。

MPI アプリケーションは１つの mpirun の呼び出しによって始められる。mpirun の引数として与える別ファイル、アプリケーションスキーマによってプロセス数と計算資源及びプログラムが指定される。簡単な SPMD アプリケーションは mpirun のコマンドラインから起動できる。MPI プロセスは、抽象的な概念である。コミュニケーターとプロセスランクによって互いの位置関係を決定する。mpirun はノード識別子のハードウェア情報と前もって定義される MPI_COMM_WORLD コミュニケーターを形成するプロセス ID を提供する。

```
% mpirun -v my_app_schema
```

4.4.2 アプリケーション・スキーマ

アプリケーションスキーマは、アプリケーションを構成しているそれぞれのプログラムごとに１行ずつ記述される。個々のプログラムのために、mpirun コマンドラインと同じ構文のオプションによって示される 3 個の重要な指示が与えられる。

-s ＜ノード識別子＞　実行プログラムを格納するファイルを有するノードを指定する。このオプションが指定されない場合、LAM は、プログラムの実行ノードにおいてプログラムを捜す。

＜ノード識別子＞　プログラムが実行するノードを指定、このオプションが指定されない場合すべてのノードを利用する。

-c ＜＃＞　与えられたノードで生成するプロセスの数を指定する。このオプションが指定されない場合、LAM は与えられたノードに一つずつプロセスを生成する。

もしアプリケーションがただ一つのプログラムから成っているならば、mpirun コマンドラインにおいてこれらと同じオプションが使用される。mpirun にこれらのオプションの１個以上が指定された場合、コマンドラインで指定したファイル名が実行プログラムであることを示す。さもなければ、ファイル名前は、アプリケーションスキーマとして、それぞれ解析される。

```
#
# sample application schema
#
master h
slave N -s h
```

上記の例は、(mpirun が呼び出された) ローカルノードにおいて ”マスタ ”を実行し、実行ファイルをローカルなノードからすべてのノードに転送した後に、すべてのノードにおいて ”スレーブ ”を実行させる。輸送されたプログラムは、／ tmp ディレクトリに格納され、プロセスが停止した時には削除される。

4.4.3 実行可能ファイルの配置

LAM は、-s オプションにより指定されたソースノードにおいて、PATH 環境変数により定義されたディレクトリのリストを利用して実行可能なファイルを探す。『.』パスの取り扱いは特別である。mpirun を呼び出したローカルなノードでは、『.』は mpirun の作業ディレクトリである。リモートのノードでは、それはユーザーのホームディレクトリである。他の mpirun オプションによって LAM の MPI ライブラリの強力な機能が利用できる。

4.4.4 直接通信

-c2c　　MPI 通信ににおいて LAM デーモンをバイパスする。

MPI プロセスの直接通信を行う、最適なプロトコルを使用する。MPI ライブラリの「クライアント間直接通信」の機能は、監視と制御の犠牲のもとにハードウェアから、最も速いスピード[1]を引き出す。送られてきても受信できないメッセージは受信側のバッファに置かれる。アプリケーションは最初デーモンを使う通信によってデバッグされ，その後、製品において直接通信を利用することが意図されている。[2]

4.4.5 保証されているエンベロープリソース

-nger　　メッセージエンベロープキューを保護する GER プロトコルを無効化する。資源オーバーフローエラーの検出、報告をしない。

「保証されたエンベロープリソース」プロトコルは、最も丈夫な MPI メッセージを供給する。

それは、他のプロセスの干渉からすべてのプロセスペア間の通信を保護する。それは、システム資源（エンベロープリソース）の不足のため受信側に引き渡されない可能性のある送信操作の発行を抑止し、それゆえ MPI のメッセージ伝達の保証という思想を尊重し、結果として、無作法なアプリケーションのデバッグにおいて混乱を減らす。プロセスペアのエンベロープキューの保護に加えて、GER は、プログラマが、デッドロックまたは故障になる前にこの資源をどれほど圧迫できるかを知ることができるように、キューのサイズを発行する。GER は、MPI における重大な移植性の欠陥を、メッセージ送信と直接関連したリソースの制限を提供することにより補う。LAM がインストールされた時には、LAM のための最小の GER は配置されている。さらに多くの詳細な議論のため、論文「保証された資源による Robust MPI メッセージ配達」および MPI のマニュアルページを参照のこと。

4.4.6 トレース・コレクション

-ton, -toff　　トレース収集を可能にする。トレース収集は、-ton が指示されると、$MPI\_Init()$ 後に開始される。一方、-toff が指示された場合 $MPI\_L\_Trace\_on()$ の発行まで延期される。

[1] スピードは MPI ライブラリ内の c2c モジュールの品質のため制約されている。カスタマイズされた通信方法を採用すればすべてのマシンで性能が向上するだろう。

[2] （ソケット実装のためのファイル記述子のような）下層システム資源の限界は、アプリケーションが-c2c を利用する際のスケーラビリティ（scalability）を制約する。

MPI ライブラリは、メッセージパッシングの活性度を詳説する実行トレースを生成できる。データは、高度なデバッグまたは性能チューニングに利用できる。実際のトレースの発生は２つのスイッチにより制御され、トレースの生成を可能にする為両方のスイッチが on のポジションにある必要がある。-ton および-toff オプションはどちらも、アプリケーション全体の実行に対して、第１のスイッチをオンとする。-ton が指定された場合、2 番目のスイッチは、アプリケーションが $MPI\_Init()$ 呼び出しの後に on で始まる。-toff が指定された場合、2 番目のスイッチは、off から始まり、トレースは全く生成されない。2 番目のスイッチは実行時の関数によって制御される。「デバッギングおよびトレーシング」を参照。2 番目のスイッチを off で開始する目的は、必要な部分のみにトレースを制限することである。第１のスイッチの目的は、アプリケーションを再コンパイルせずトレースを可能とすること、およびトレースを制御する関数をバラまいたアプリケーションに対してすべてのトレースを無効にし、オーバヘッドを最小にすることを可能とすることである。トレースを on にしたアプリケーションの実行後、どうやって単一のトレースファイルをつくるのかは、「トレースデータの収集」を参照すること。

4.4.7　lamclean

失敗したアプリケーションは多くのプロセスが動いたままもしくはブロックされたまま、または多くのメッセージが未消費のまま、あるいはマルチコンピュータに渡って多くの資源を割り当てたまま放置するかも知れない。放置されたユーザーの幽霊をすべての個々のサブシステムから取り去るユーザーインタフェースコマンドがあるけれども、それらをすべて呼び出すのは退屈な作業である。マルチコンピュータにおいてユーザーの幽霊（プロセス、メッセージ、割り当て、登録）が全く必要無い場合、lamclean コマンドが利用できる。ユーザーは、長い時間を要する LAM セッションの再スタートをせずにアプリケーションを起動したいはずである。

```
% lamclean
```

いくつかまたはすべてのノードが壊滅的な失敗、またはリンク故障を起こす完全なバファオーバーフローによって到達可能でないときには、lamclean は利用できない。lamclean が、戻ることに失敗するなら、wipe tool を使用するべきである。「LAM の起動」を参照。lamclean の仕事を確認するために mpitask と mpimsg コマンドを使用できる。

Chapter 5

LAM コマンドリファレンス

5.1 プロセスの監視と制御

mpitask	MPI プロセスの状態を出力する
doom	プロセスへ信号を送る

5.1.1 mpitask

プロセスの実行状態の監視は、マルチコンピュータアプリケーションのデバッキングにおいて主な手段となる。この特徴は、並列プログラミングにおいて新たに必要となるプロセスの同期とデータの転送のデバッグに役立つ。mpitask コマンドは MPI プロセスに関する情報を出力する。引数がない場合、すべてのノード上のすべての MPI プロセスが報告される。報告は、ノードと LAM プロセスを指定することにより限定することができる。LAM の監視能力をデモンストレートすることだけを目的とした以下例題を検討する。

```
/*
 * mpitask と mpimsg のレポートを生成する。
 */
#include <mpi.h>
#define ROWS            10
#define COLS            20
struct cell {
      int               code;
      double            coords[3];
};
static struct cell      mat[ROWS][COLS];
static int              blocklengths[3] = {1, 3, 1};
static MPI_Datatype     types[3] =
           {MPI_INT, MPI_DOUBLE, MPI_UB};
main(argc, argv)
int argc;
char *argv[];
{
      int               rank, size;
      MPI_Comm          newcomm;
      MPI_Datatype      dt_cell, dt_mat;
      MPI_Status        status;
      MPI_Aint          base;
      MPI_Aint          displacements[3];
```

```
      int               i, j;
% 1461
/*
 * 行列を初期化する。
 */
   for (i = 0;i < ROWS;++i){
        for (j = 0;j < COLS;++j){
             mat[i][j].code = i;
             mat[i][j].coords[0] = (double) i;
             mat[i][j].coords[1] = (double) j;
              mat[i][j].coords[2] = (double) i * j
        }
   }
   MPI_Init(&argc, &argv);
% 1474
/*
 *  sub-groups のためにコミュニケータを生成する。
 */
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_split(MPI_COMM_WORLD, rank % 3, 0, &newcomm);
% 1477
/*
 * 行列の 2 つの列に派生データ型を作る。
 */
      MPI_Address(&mat[0][0].code, &displacements[0]);
      MPI_Address(mat[0][0].coords, &displacements[1]);
      MPI_Address(&mat[0][1].code, &displacements[2]);
      base = displacements[0];
      for (i = 0;i < 3;++i) displacements[i] -= base;
      MPI_Type_struct(3, blocklengths, displacements,
                  types, &dt_cell);
      MPI_Type_vector(ROWS, 2, COLS, dt_cell, &dt_mat);
      MPI_Type_commit(&dt_mat);
% 1488
/*
 * 条件が満たされない送信と受信を実行する。
 */
      MPI_Comm_size(newcomm, &size);
      MPI_Comm_rank(newcomm, &rank);
      MPI_Send(&mat[0][0], 1, dt_mat, (rank + 1) % size,
                  0, newcomm);
      MPI_Recv(&mat[0][0], 1, dt_mat, (rank + 1) % size,
                  MPI_ANY_TAG, newcomm, &status);
      MPI_Finalize();
      return(0);
}
```

プログラムを十分なプロセス数で実行する。それから mpitask でアプリケーションプロセスの状態を調べる。

```
% mpirun -v -c 10 demo
```

デフォルトのディスプレイモードでは、mpitask は下記の見出しの下に情報を出力する。

TASK (G/L)　　プロセスの識別。MPI プロセスは通常、MPI_COMM_WORLD におけるランクによって識別される。また ”グローバル ”（G）ランクとしても参照される。もしそのプロセスがあるコミュニケータでブロックされたら、”／ ” 以降に、そのコミュニケータにおけるランクが示される。

```
% mpitask
TASK (G/L) FUNCTION PEER|ROOT TAG COMM COUNT DATATYPE
0/0   demo   Recv     3/1       ANY <2>   1     <30>
2/0   demo   Recv     5/1       ANY <2>   1     <30>
4/1   demo   Recv     7/2       ANY <2>   1     <30>
6/2   demo   Recv     9/3       ANY <2>   1     <30>
8/2   demo   Recv     2/0       ANY <2>   1     <30>
1/0   demo   Recv     4/1       ANY <2>   1     <30>
3/1   demo   Recv     6/2       ANY <2>   1     <30>
5/1   demo   Recv     8/2       ANY <2>   1     <30>
7/2   demo   Recv     1/0       ANY <2>   1     <30>
9/3   demo   Recv     0/0       ANY <2>   1     <30>
```

これは ”ローカル ”（L）ランクとして参照される。プログラム名もまた出力される。

FUNCTION　現在実行中の MPI ルーチン

PEER/ROOT　もし１対１通信操作が FUNCTION の下に記されていたら、通信操作のプロセスの送信元、または送信先、そうでなければ集団操作のルートプロセスが示される。

TAG　１対１通信のメッセージタグ

COMM　使用されたコミュニケータの識別子。（プログラムのデータとこの数字の相互参照方法は「デバッグとトレース」を参照。）

COUNT　転送された要素の数

DATATYPE　転送されたそれぞれの要素のデータ型識別子。

MPI ルーチンによっては、いくつかのフィールドは必要ないので、空白のままとなる。もしプロセスが現在実行している MPI ルーチンがないのであれば、以下に示す実行状態の内の一つが報告される。

⟨*running*⟩　OS において実行中。

⟨*paused*⟩　lam_kpause() でブロック中。

⟨*stopped*⟩　LAM 信号（LAM_SIGARREST）による停止。

⟨*blocked*⟩　LAM ルーチン内でブロック。一般にこれは一時的な状態であるべきである。

5.1.2　GPS 識別

プロセスが生成された場合や、そうでなくても、ある LAM セッションにおいて複数の MPI アプリケーションが実行している場合、MPI_COMM_WORLD のランクによって、必ずしも MPI プロセスの明確な識別ができるわけではない。LAM はグローバルランクに代わる、GPS（Global Pasitioning System) と呼ばれるものを持つ。

-gps　MPI_COMM_WORLD のランクであるグローバルランクの代わりに GPS によって MPI プロセスの識別を行なう。

GPS はプロセスで実行しているノード識別子とノード中の LAM プロセスはインデックスからなる。[1]

```
% mpitask -gps n0 i8
TASK (GPS/L)  FUNCTION  PEER|ROOT  TAG  COMM   COUNT   DATATYPE
n0,i8/0 demo  Recv      n1,i9/1    ANY  <2>    1       <30>
```

MPI のコミュニケータとデータ型は２つの不透明なオブジェクトであり、＜＃＞とい変わった形式で示される。拡張されたライブラリ実装は、実行中アプリケーションの中で同じ値を報告することができる。「デバッグとトレース」を参照。コミュニケータとデータ型の情報は mpitask により報告される。

5.1.3　コミュニケータの監視

-c　　デフォルトのレポートではなく、選択された全てのプロセスにおけるコミュニケータの情報を出力する。

コミュニケータレポートは、デフォルトのレポートの TASK という見出しの下にプロセスの識別子を表示する。また、コミュニケータのサイズとコミュニケータのプロセスグループ内の全てのプロセスのグローバルランク（-gps オプションをつけた時は GPS）も報告される。もしそれがグループ間コミュニケータであれば両方のグループのメンバが報告される。

```
% mpitask -c n0 i8
TASK (G/L):   0/0 demo
INTRACOMM:   <2>
 SIZE:       4
 GROUP:      0 3 6 9
```

5.1.4　データ型の監視

-d　　デフォルトのレポートの代わりに、選択された全てのプロセスにおけるデータ型の情報が出力される。

そのデータ型の報告は、デフォルトのレポートにおいて TASK 見出しの下に示されるプロセスの識別子を含んでいる。また、ASCll キャラクタだけで表現するのが困難なデータ型のタイプマップを図式表現で報告する。派生データ型のレベルを字下げによって示した階層的な形式をとる。基本データ型はコード中の名前で報告する。派生データ型では、型の構成子は変位長、ブロック長およびブロック数の情報によって示される。サンプルコードと mpitask　-d　の出力を比較せよ。

```
% mpitask -d n0 i8
TASK (G/L):  0/0 demo
DATATYPE:   <30>
      MPI_VECTOR (10 x 2, 20)
            MPI_STRUCT (3)
                (1, 0) MPI_INT
                (3, 8) MPI_DOUBLE
                 (1, 32) MPI_UB
```

[1]アプリケーションプロセスのインデックスは０でも１でも始まらない、それは LAM システムプロセスが初めのいくつかのポジションを占めているからである。

mpitask により報告される MPI プロセスの数は、mpitask コマンドラインにおいて示したノード識別子とプロセスインデックスによって制限することができる。GPS レポートを利用することで、正しいノード識別子とプロセスインデックスの選択が容易となる。多くのあるいは全てのプロセスが同じコミュニケータやデータ型を報告する場合、コミュニケーションやデータ型の報告において単一のプロセスを選択することは非常に有用である。

5.1.5 doom

doom は信号伝達のためのコマンドレベルのインタフェースである。一つまたは複数のノードをコマンドラインで指定しなければならない。もしプロセスを指定しなかった場合、選択されたノードの全てのアプリケーションプロセスに信号が送られる。他のオプションを付けない場合、doom は LAM_SIGUDIE 信号を送る。あいにく、ユーザは信号ニーモニックを指定できない。そして代わりに実際の信号の番号を引数として与えなければならない。それらを下のリストに示す。

-1 (LAM_SIGTRACE) トレースデータをアンロード

-4 (LAM_SIGUDIE) 終了させる

-5 (LAM_SIGARREST) 実行を延期する

-6 (LAM_SIGRELEASE) 実行を再開

-7 (LAM_SIGA) ユーザのための予備

-8 (LAM_SIGB) ユーザのための予備

-9 (LAM_SIGFUSE) ノードが死にそうである

-10 (LAM_SIGSHRINK) 他のノードが死んだ

例えばノード１のプロセスインデックス８を延期させるためには、以下の形式を使用する。

```
% doom n1 i8 -5
```

その延期したプロセスの実行を再開する場合は

```
% doom n1 i8 -6
```

5.2　メッセージの監視と制御

mpimsg	メッセージバッファの監視
bfctl	メッセージバッファの制御

5.2.1　mpimsg

受信プロセスは通常、mpitask コマンドでデバッグされる。しかし送信プロセスはメッセージを転送し、バッファがあるので、すぐにレディ状態になってしまう。mpimsg コマンドはバッファにためられたメッセージを調べるためにある。引数がある場合、全てのノードの全ての MPI メッセージが報告される。その報告は、ノードとプロセスを指定するによって制限できる。「プロセスの監視と制御」には、受信されないメッセージを送るプログラムの例がある。それらのメッセージは mpimsg によって調べることができる。

```
% mpimsg
 SRC (G/L) DEST (G/L) TAG COMM COUNT DATATYPE MSG
 9/3       0/0        0   <2>  1     <30>     n0,#0
 8/2       2/0        0   <2>  1     <30>     n0,#1
 1/0       4/1        0   <2>  1     <30>     n0,#2
 3/1       6/2        0   <2>  1     <30>     n0,#3
 5/1       8/2        0   <2>  1     <30>     n0,#4
 7/2       1/0        0   <2>  1     <30>     n1,#0
 0/0       3/1        0   <2>  1     <30>     n1,#1
 2/0       5/1        0   <2>  1     <30>     n1,#2
 4/1       7/2        0   <2>  1     <30>     n1,#3
 6/2       9/3        0   <2>  1     <30>     n1,#4
```

デフォルトの表示モードでは、mpimsg は下記の見出しの下に情報を出力する。

SRC(G/L)　送信プロセスの識別は ”／ ”に続いてプロセスのコミュニケータランク（ローカルランク）を示す。

DEST (G/L)　受信プロセスの識別。”／ ”に続いてプロセスのコミュニケータランクにを示す。

TAG　メッセージタグ

COMM　コミュニケータ ID

COUNT　メッセージ中の要素数

DATATYPE　各要素のデータ型識別子

MSG　内容の問い合わせに使用されるメッセージ識別子

mpitask によりプロセスから手に入れられるものと同じコミュニケータとデータ型の情報は、メッセージからも手に入る。その違いは、メッセージを指定するためにより正確さが必要とされることである。なぜなら１つのプロセスは複数のメッセージを生成することができるからである。プロセスインデックスの代わりに mpimsg は-c（コミュニケータ）または-d（データ型）をパラメータとしたメッセージナンバーを要求する。実際、mpimsg に必要な情報は、デフォルトの報告の MSG の見出しの下に出力されるものである。

5.2.2 メッセージ内容

-m ⟨#⟩　　指定されたノード上の指定されたメッセージナンバーの内容を表示する。

メッセージ報告における特色ある能力としてメッセージ内容の表示をあげられる。データ型の型マップはデータを整形するために使用される。それぞれのラインの先頭に示される数値は、アンパックされたメッセージの最初からの先頭からのオフセットである。基本データ型の連続ブロックは、連続して出力され、ブロックとブロックの間には改行が挿入される。

```
% mpimsg -gps n0 -m 4
MESSAGE:    n0,i12/2 #4
00000000:   0
00000008:   0 0 0
00000020:   0
00000028:   0 1 0
00000280:   1
00000288:   1 0 0
000002a0:   1
000002a8:   1 1 1
00000500:   2
...
```

5.2.3 bfctl

LAM デーモンは、オペレーティングシステムがメモリを使い尽くすまでバッファのスペースの割り当ての拡張をしたりはしない。メッセージがいくつか消費されるまで、新たなメッセージを受け入れない限界設定されている。必要とされるバッファのスペースが確保できないなら送信においてプロセスは送信作業をブロックされる。デフォルト GER プロトコル（「MPI プログラムの実行」を参照）を使用する場合、mpirun は保障されているエンベロープリソースに応じてバッファの限度を調整する。もしこのプロトコルが無視されている場合、手動でバッファの限界を調整する必要があるだろう。ユーザは、bfctl コマンドで LAM デーモンのバッファプールの最大サイズを制御できる。

-s　　Adjust the upper limit on buffered messages for the selected nodes. 選択されたノードに対する、バファに蓄積されるメッセージの上限調整。

```
% bfctl N -s 0x100000
```

5.3 トレースデータの収集

lamtrace　　トレースデータを収集し、ファイルに蓄積する。

トレースされたアプリケーションが実行を終えた後、通信の活動を記録しているトレースデータはアプリケーションを実行したすべてのノード中の LAM デーモンに蓄積される。１つの LAM デーモンが、どの程度トレースデータを保持するかには限度がある。その限度に達した時、一番古いトレースは、一番新しいトレースのために捨てられる。トレースデータ量を制限するランタイムルーチンについては「デバッグとトレース」を参照のこと。

5.3.1 lamtrace

lamtrace コマンドはトレースデータを集めて.lamtr というサフィックスを持つ１つのファイルの中に格納する。

```
% lamtrace -v -mpi
```

-mpi　　指定したプロセスが生成した MPI ワールドのトレースを検索する。

たいていは、lamtrace と LAM デーモンは個別のトレースフォーマットに関知しない。プロセスが生成されたり、複数のアプリケーションが存在することによって複数のワールドグループがある状態において、特定の一グループの MPI トレースデータを取り出すために lamtrace は LAM の MPI ライブラリにより生成された管理トレースレコードのフォーマットを理解する。１つのアプリケーションしかなくプロセスが生成されていない単純な状況ではノードやプロセスの特定は必要ない。lamtrace は全てのノードを調べ、MPI_COMM_WORLD のランクが０のプロセスによって生成された一つの MPI ワールドのトレースを割り当てる。しかしながら、もし複数のワールドグループのトレースデータがノード中にある場合、必要なワールドグループのデータを得るために、ノードとおそらくプロセスの指定がコマンドラインに与えられなければならない。正しいノード識別子とプロセスインデックスは、mpitask かアプリケーションスキーマからの推定によって知ることができる。例えば：

```
% lamtrace -v -mpi n0 i8
```

アプリケーションが終了する前であってもトレースデータを引き出すことは問題なく可能であるが、その瞬間の不完全な通信がトレースデータに反映されることに注意が必要である。

application terminates.　アプリケーションが終了した後、トレースデータは LAM デーモンに残り、引き出されるのを待っている。もしアンロードされていなかったなら、次のアプリケーションを実行する前に削除するべきである。これは、lamclean が行なう動作の１つである。

5.4 LAM ノードの追加と削除

lamgrow	カレント LAM セッションへノードを追加する。
lamshrink	ノードを削除する。

外部のリソースマネージャ等によって資源の割り当てが動的に行われる環境において LAM は動作する。LAM が開始される初期のノードの集合は lamboot により設定される。もしその後、（ソフトウエアまたは人間の）リソースマネージャが LAM セッションに属しているノードの集合を変更する場合、その変更は lamgraw と lamshrink の２つのコマンドによってなされる。どちらのコマンドも現在の LAM ノードの一つから実行されなければならない。

5.4.1 lamgrow

新しいマシンは新しいノード識別子を割り当てられ、lamgrow により LAM セッションへ追加される。使用法は標準的な LAM コマンドよりももっと制限されている。

- ノード識別子は現ノード集合の識別子と重複する指定をしてはいけない。
- lamgrow の発行につき１つのノードのみ追加できる。
- マシンの名前を与えなければならない。LAM は自動的に追加ノードを選択したりはしない。
- LAM 並列コンピュータの中で lamgrow はただ１つのみ実行していなければならない。

```
% lamgrow -v n8 buckeye.osc.edu
```

もしノード識別子が指定されていなければ、次の最も大きな LAM ノード識別子が使用される。ノード識別子を指定する能力によって lamgrow は lamboot が初期化時に保障がゼロから連続的に割り振られるという性質を変えることができる。

-x	欠陥の検出と復旧の耐故障性を有効にする。一般に lamboot の発行に引き続いてこのオプションの使用が決定される。
-c ⟨*bhost*⟩	ホストリストに新しいマシン名を追加することによりブートスキーマを更新する。これは、wipe で使用するためのブートスキーマを更新するシンプルで便利な機構である。

5.4.2 lamshrink

lamshrink の発行のたびに一つのノードが削除される。発行に際してノード識別子とマシン名を指定しなければならない。

```
% lamshrink -v n8 buckeye.osc.edu
```

-w ⟨*#secs*⟩　　停止させたノードの全てのアプリケーションプロセスへ信号 (LAM_SIGFUSE) が送信され、続行する前に一時的に中断する。
Signal　Handling　参照

5.5　ファイルの監視と制御

fstate　　リモートファイルシステムのステータスを得る。
fctl　　リモートファイルシステムを制御する。

5.5.1　fstate

リモートファイルアクセスの制御と監視をするためのコマンドがいくつかある。(「リモートファイルアクセス」を参照) fstate はオープンしているファイル記述子ごとにそれぞれ一行づつステータス情報を出力する。

FD	（クライアントハンドルではなく）グローバルファイルディスクリプタハンドル。
FLAGS	オープンフラグとステータスフラグ（下記を参照)。
FLOW	オープン以来の I/O の総バイト数。
CLIENT	最後のクライアントプロセスのノード識別子とプロセス ID。
NAME	ファイル名

オープン／ステータスフラグは一文字のニーモニックで示される。

R	読出し用オープン。
W	書込み用オープン。
L	アクティブでロック中。
A	アクティブ、ファイルシステムにおいてオープンしている。
I	インアクティブ、ファイルシステムにおいてクローズしている。

```
% fstate N
NODE    FD    FLAGS     FLOW    CLIENT     NAME
 n0 (o) 0     R|L       0       none       /dev/null
 n0 (o) 1     W|L       0       n0/p25825  /dev/ttya
 n0 (o) 2     R|W|L     0       none       /dev/ttya
```

5.5.2　fctl

fctl コマンドは２つの特徴を持つ。-s オプションは指定したファイル記述子の後始末をした後クローズする。一方、-S オプションはすべてのファイル記述子に対して同じことを行なう。オプションなしで起動されると、fctl はリモートファイルシステムのカレントワーキングディレクトリを出力する。ワーキングディレクトリは新しいパス名を fctl に与えることにより変更できる。LAM の現行リリースでは、ワーキングディレクトリはプロセスごとではなくノードごとに保持される。

```
% fctl -s 4
```

5.6　LAM ブート・スキーマを書く

bhost.my3suns　　ホストファイルの例

マルチコンピュータのトポロジーはブートスキーマによって設定される。ブートスキーマには識別子とノードの型、および使用される実マシンが記述される。それはローカルマシンのユーザ名とあるマシン上のユーザアカウント名が異なる場合にはそのアカウント名を含むだろう。ブートスキーマを LMA セッションが始まる時 lamboot に使用され、セッションを終了する時に wipe に使用される。「LAM の起動」を参照。
現実装に対応した何通りかの異なったマルチコンピュータを記述するブートスキーマがすでに利用可能であるだろう。これらのファイルは、一般に$LAMHOME/boot ディレクトリに見つけられる。ネットワークの構成にはしばしば多くの選択の余地があるので、LAM ユーザは、彼ら自身のブートスキーマを書く必要があるだろう。この節では、ホストファイル構文を使用した LAM のブートスキーマの書き方を示す。例となる並列コンピュータは３つのノードを持ち、それらの１つは違ったユーザアカウント名を持つ。

5.6.1　ホストファイル構文

ホストファイル構文は、LAM ブートスキーマで必要とされる情報を表す非常に簡単な方法である。各行に各１台のマシンが記述され、マシン名に続いてオプションのユーザアカウント名が示される。もし lamboot が用いられるローカルマシン上とそのマシンのアカウント名が異なっている場合、ユーザ名が必要とされる。もしユーザ名が与えられなかったら、そのローカルマシンと同一の名前が使用される。そのノード識別子は、マシンがそのファイルに現れた順番により決定される。それはノード番号０で始まり、連続したノード番号が割り当てられる。行内の#文字以降の部分は、コメントを示す。
３つのノード例においては、”ohio”、”osc”そして”faraway.for.edu” と名づけられ、０、１、２とそれぞれ番号が付けられたと仮定する。それはまた、ユーザがノード０にログオンし、ノード１で同じユーザ名を持ち、しかしノード２では異なった名前（guest）を持つと仮定する。ノード１はローカルノードとして同じユーザネームを持つのでそれを指定する必要はない。ホストファイル構文を使用するブートスキーマの例は以下に示される。

```
# a 3 node example
ohio
osc
faraway.far.edu guest
```

5.7　低レベルの LAM の起動

hboot　一つのノード上での LAM の起動

lamboot コマンドは特定のノード上の LAM をスタートさせる低レベルのプログラムを実行する。通常、ユーザは lamboot を使用するだけでよい。それが lamboot のオプションによって制御できない通常のスタートアッププロシージャのバリエーション要求されるような特別な場合には、ユーザはシステムを手動で起動できる。低レベルの hboot ツールを実行することによって、ユーザは起動を自分の要求に合うように調整し、そして (または)lamboot の複雑な部分を迂回するようなオプションを選択できる。

5.7.1　プロセススキーマ

hboot ツールはプロセススキーマと呼ばれる各々のノードに対応するコンフィギュレーションファイルを読む。プロセススキーマはノード上における LAM を構成するプログラムのリストと実行時引数を記述する。hboot のデフォルトのプロセススキーマのファイル名は conf.obt である。lamboot は conf.lam プロセススキーマを使用して hboot を呼び出す。ユーザがカスタムブートスキーマを作成できるように、ユーザはカスタムプロセススキーマを作成することができる。それによってプロセスレベルで LAM を再構成する事が容易になる。プロセススキーマの文法の完全な記述についてはプロセススキーマのオンラインマニュアルを参照すること。LAM セッションを手動で起動するには、まずブートスキーマを参照する。このファイルはノード識別子と実際のマシンとの対応を記述するだけでなくノード識別子を指定する。下に示すブートスキーマの例はホストのファイル構文によって書かれ、3 ノードのマルチコンピュータを記述している。

```
#a 3 node example
ohio
osc
faraway.far.edu guest
```

5.7.2　hboot

各ノードは hboot ツールによって起動される。そしてそれはマルチコンピュータにおいて完全（結合の）LAM トポロジを形成するため、各ノードに他ノードの情報を与える。ユーザが「ohio」というマシンにログオンされたと仮定し、まず LAM をローカルで起動する。

```
{ohio}%    hboot -vc conf.lam -I "-n0 -o0
           osc 1 faraway.far.edu 2"
```

次に「osc」というマシンにログインし、そのマシン上で LAM を起動する。

```
{osc}%   hboot -vc conf.lam -I "-n1 -o0
         ohio 0 faraway.far.edu 2"
```

そして「faraway.far.edu」というマシンに「guest」というアカウントでログインし、そのマシン上で LAM を起動する。

```
{faraway}%   hboot -vc conf.lam -I "-n2 -o0
             ohio.here.edu 0 osc.here.edu 1"
```

この最後のケースにおいて、「ohio」と「osc」は「faraway」と違うドメインにあるので、完全なインターネットアドレスでマシン名が与えられていることに注意すること。-I オプションのパラメータはプロセススキーマにおいては`$inet_topo`という変数の値になる。この変数はネットワーク情報を確かめるために LAM によって使用される。

- -o

  起源のノードのノード識別子ー起源のノードはユーザが lamboot を発動ノードであると仮定される。多くの LAM の機能において起源のノードをデフォルトのノード識別子として使用する。

- -n

  ローカルのノード識別子

ローカル及びリモートのノード識別子を設定する以外に、ネットワーク情報は他のすべてのノードに対するマシン名とリンク番号のペアを含んでいる。リンク番号は LAM のノード識別子に等しい。同様の手続きは、それぞれのマシンにログインせずに UNIX の rsh を使用して行うこともできる。この場合、hboot が行われたときに rsh がリターンすることを許可するために hboot の-s オプションを使用すること。

5.8　付録 A：Fortran 言語からの呼び出し形式

この付録は本文書で書かれているライブラリルーチンのための Fortran 言語の呼び出し形式を含んでいる。特に注意しない限り、すべての呼び出し形式はサブルーチンである。

- 初期設定から

```
MPI_INIT (ierror)
      integer ierror
MPI_FINALIZE (ierror)
MPI_ABORT (comm, errcode, ierror)
      integer comm, errcode
MPI_COMM_SIZE (comm, size, ierror)
      integer comm, size
MPI_COMM_RANK (comm, rank, ierror)
      integer comm, rank
```

- ２点間ブロッキングから

```
MPI_SEND (buf, count, dtype, dest, tag, comm,
          ierror)
      <type> buf(*)
      integer count, dtype, dest, tag, comm
MPI_RECV (buf, count, dtype, source, tag, comm,
          status, ierror)
      <type> buf(*)
      integer count, dtype, source, tag, comm
      integer status(MPI_STATUS_SIZE)
MPI_GET_COUNT (status, dtype, count, ierror)
      integer status(MPI_STATUS_SIZE), dtype, count
MPI_PROBE (source, tag, comm, status, ierror)
      integer source, tag, comm
      integer status(MPI_STATUS_SIZE)
```

- ２点間ノンブロッキングから

```
MPI_ISEND (buf, count, dtype, dest, tag, comm,
           request, ierror)
      <type> buf(*)
      integer count, dtype, dest, tag
      integer comm, request
MPI_IRECV (buf, count, dtype, source, tag, comm,
           request,ierror)
      <type> buf(*)
      integer count, dtype, source, tag
      integer comm, request
MPI_TEST (request, flag, status, ierror)
      logical flag
      integer request, status(MPI_STATUS_SIZE)
MPI_WAIT (request, status, ierror)
      integer request, status(MPI_STATUS_SIZE)
MPI_IPROBE (source, tag, comm, flag, status,
           ierror)
      logical flag
      integer source, tag, comm
      integer status(MPI_STATUS_SIZE)
```

- メッセージのデータ型から

```
MPI_TYPE_VECTOR (count, blocklength, stride,
          oldtype, newtype, ierror)
      integer count, blocklength, stride
      integer oldtype, newtype
MPI_TYPE_STRUCT (count, blocklengths,
          displacements, dtypes, newtype, ierror)
      integer count, blocklengths(*)
      integer displacements(*), dtypes(*), newtype
MPI_ADDRESS (location, address, ierror)
      <type> location(*)
      integer address
MPI_TYPE_COMMIT (dtype, ierror)
      integer dtype
MPI_PACK_SIZE (incount, dtype, comm size, ierror)
      integer incount, dtype, comm, size
MPI_PACK (inbuf, incount, dtype, outbuf, outsize,
          position, comm, ierror)
      <type> inbuf(*), outbuf(*)
      integer incount, dtype, outsize
      integer position, comm
MPI_UNPACK (inbuf, insize, position, outbuf,
           outcount, dtype, comm, ierror)
      <type> inbuf(*), outbuf(*)
      integer insize, position, outcount
      integer dtype, comm
```

- 集団メッセージ通信から

```
MPI_BCAST (buf, count, dtype, root, comm, ierror)
      <type> buf(*)
      integer count, dtype, root, comm
MPI_SCATTER (sendbuf, sendcount, sendtype,
          recvbuf, recvcount, recvtype, root,
          comm, ierror)
      <type> sendbuf(*), recvbuf(*)
      integer sendcount, sendtype, recvcount
      integer recvtype, root, comm
MPI_GATHER (sendbuf, sendcount, sendtype,
          recvbuf, recvcount, recvtype, root
          comm, ierror)
      integer sendcount, sendtype, recvcount
      integer recvtype, root, comm
MPI_REDUCE (sendbuf, recvbuf, count, dtype, op,
          root, comm, ierror)
      <type> sendbuf(*), recvbuf(*)
      integer count, dtype, op, root, comm
```

- コミュニケータの生成から

```
MPI_COMM_DUP (comm, newcomm, ierror)
      integer comm, newcomm
MPI_COMM_SPLIT (comm, color, key, newcomm, ierror)
      integer comm, color, key, newcomm
```

- プロセストポロジから

```
MPI_CART_CREATE (oldcomm, ndims, dims, periods,
          reorder, newcomm, ierror)
      integer oldcomm, ndims, dims(*), newcomm
      logical periods(*), reorder
MPI_CART_RANK (comm, coords, rank, ierror)
```

```
      integer comm, coords(*), rank
MPI_CART_COORDS (comm, rank, maxdims, coords,
          ierror)
      integer comm, rank, maxdims, coords(*)
MPI_CART_SHIFT (comm, direction, distance, rank_source,
          rank_dest, ierror)
      integer comm, direction, distance
      integer rank_source, rank_dest
```

- 様々な MPI の特徴から

```
MPI_ERRHANDLER_CREATE (errfunc, handler, ierror)
      external errfunc
      integer handler
MPI_ERRHANDLER_SET (comm, handler, ierror)
      integer comm, handler
MPI_ERROR_STRING (code, errstring, resultlen,
          ierror)
      integer code, resultlen character*(*) errstring
MPI_ERROR_CLASS (code, class, ierror)
      integer code, class
double precision MPI_WTIME()
```

- リモートファイルアクセスから

```
lamf_rfopen (lamfd, file, flags, modes,ierror)
      integer lamfd, flags, modes
      character*(*) file
lamf_rfclose (lamfd, ierror)
      integer lamfd
lamf_rfread (lamfd, buf, length, nread, ierror)
      integer lamfd, length, nread
      <type> buf(*)
lamf_rfwrite (lamfd, buf, length, nwritten, ierror)
      integer lamfd, length, nwritten
      <type> buf(*)
```

- 集団 I/O から

```
CBX_OPEN (file, flags, mode, owner, comm, cbxfd, ierror)
      character*(*) file
      integer flags, mode, owner, comm, cbxfd
CBX_CLOSE (cbxfd, ierror)
      integer cbxfd
CBX_READ (cbxfd, buf, count, dtype, nread, ierror)
      integer cbxfd, count, dtype, nread
      <type> buf(*)
CBX_WRITE (cbxfd, buf, count, dtype, nwritten, ierror)
      integer cbxfd, count, dtype, nwritten
      <type> buf(*)
CBX_LSEEK (cbxfd, offset, whence, ierror)
      integer cbxfd, offset, whence
CBX_MULTI (cbxfd, ierror)
      integer cbxfd
CBX_SINGL (cbxfd, ierror)
      integer cbxfd
CBX_IS_MULTI (cbxfd, result, ierror)
      integer cbxfd
      logical result
CBX_IS_SINGL (cbxfd, result, ierror)
      integer cbxfd
      logical result
```

```
CBX_ORDER (cbxfd, newrank, ierror)
      integer cbxfd, newrank
```

- プロセスの生成から

```
MPIL_SPAWN (comm, app, root, childcomm, ierror)
      integer comm, root, childcomm
      character*(*) app
MPIL_COMM_PARENT (comm, ierror)
      integer comm
MPIL_UNIVERSE_SIZE (size, ierror)
      integer size
```

- シグナルのハンドリングから

```
MPIL_SIGNAL (comm, rank, signo, ierror)
      integer comm, rank, signo
```

- デバッギングとトレーシングから

```
MPIL_COMM_ID (comm, id, ierror)
      integer comm, id
MPIL_COMM_GPS (comm, rank, nodeid, pid, ierror)
      integer comm, rank, nodeid, pid
MPIL_TYPE_ID (dtype, id, ierror)
      integer dtype, id
MPIL_TRACE_ON (ierror)
MPIL_TRACE_OFF (ierror)
```

5.9　付録 B：Fortran 言語のプログラム例

プログラミングチュートリアルからの Fortran 言語での一般的なプログラム例をここに示す。

```
c
c Transmit a message in a two process system.
c
      program trivial
#include <mpif.h>
      integer*4 BUFSIZE
      parameter (BUFSIZE = 64)
      integer*4 buffer(BUFSIZE)
      integer rank, size
      integer status(MPI_STATUS_SIZE)
c
c Initialize MPI.
c
      call MPI_INIT(ierror)
c
c Error check the number of processes.
c Determine my rank in the world group.
c The sender will be rank 0 and the receiver, rank 1.
c
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
      if (size .ne. 2) then
            call MPI_FINALIZE(ierror)
            stop
      endif
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
c
c As rank 0, send a message to rank 1.
c
      if (rank .eq. 0) then
            call MPI_SEND(buffer(1), BUFSIZE, MPI_INTEGER,
     +                  1, 11, MPI_COMM_WORLD, ierror)
c
c As rank 1, receive a message from rank 0.
c
      else
            call MPI_RECV(buffer(1), BUFSIZE, MPI_INTEGER,
     +                  0, 11, MPI_COMM_WORLD, status,
     +                  ierror)
      endif
      call MPI_FINALIZE(ierror)
      stop
      end
```

連絡　オハイオスーパーコンピュータセンタ
1224 Kinnear Road Columbus, OH 43212(lam@tbag.osc.edu)

詳細な情報　http://www.osc.edu/lam.html
ftp://ftp.osc.edu/pub/lam

著作権　この文書は著作権によって保護される。(著者 GBD/RDB)
©Copyright 1995 The Ohio State University

承認　LAM 文書は CCR-9510016 の承諾の元に
The National Science Foundation
によって一部サポートされる。