|  |
| --- |
| 博士課程教育リーディングプログラム  実世界データ循環学リーダー人材育成プログラム |
| データツールファースト |
| MPI入門 |

|  |
| --- |
| 名古屋大学  2014/03/03 |

* Linuxは、Linux Torvaldsの米国およびその他の国における登録商標あるいは商標です。
* Visual C++ は、米国およびその他の国における米国 Microsoft Corp.の 登録商標です。
* 記載された社名、製品名、機種名は各社の商標または登録商標です。

目次

[1. はじめに 1](#_Toc381439036)

[1.1 背景 1](#_Toc381439037)

[1.2 本書について 1](#_Toc381439038)

[2. 並列プログラミング 3](#_Toc381439039)

[2.1 プログラミング言語 3](#_Toc381439040)

[2.2 言語拡張 9](#_Toc381439041)

[2.3 ライブラリ 12](#_Toc381439042)

[2.4 まとめ 15](#_Toc381439043)

[3. MPI入門 16](#_Toc381439044)

[3.1 Hello World 16](#_Toc381439045)

[3.2 メッセージ通信 19](#_Toc381439046)

[3.3 マルチホスト 34](#_Toc381439047)

[3.4 C言語バインディングリファレンス 37](#_Toc381439048)

[3.4.1 基本バインディング 37](#_Toc381439049)

[3.4.2 同期通信バインディング 39](#_Toc381439050)

[3.4.3 非同期通信バインディング 41](#_Toc381439051)

[3.4.4 集団通信バインディング 44](#_Toc381439052)

[3.4.5 データ型 49](#_Toc381439053)

[3.4.6 エラーコード 50](#_Toc381439054)

[3.4.7 その他のバインディング 50](#_Toc381439055)

[3.5 章末問題 51](#_Toc381439056)

[付録A. 52](#_Toc381439057)

[A.1 mpicc 52](#_Toc381439058)

[A.2 mpirun 52](#_Toc381439059)

[参考文献 54](#_Toc381439060)

[変更履歴 55](#_Toc381439061)

# はじめに

## 背景

**並列計算の必要性**

スーパーコンピュータは膨大な演算処理を目的とします。昨今では価格性能比の観点から、汎用プロセッサによるノードを多数結合して並列に動作させ、全体として演算処理性能のスループットを得る構築が主流となっています。この構成は「High Performance Computing(HPC)クラスタ」と呼ばれます。

また、汎用プロセッサもクロック周波数の上昇が鈍化し、多数のコアを並列に動作させることにより全体として演算処理性能のスループットを得るメニーコアに移行しています。

多数のノードやプロセッサの性能を効率よく引き出すためには、ソフトウェアで処理を分散させ、並列に計算をさせる必要があります。

**MPI**

Message Passing Interface(MPI)とは、並列計算を利用するための通信ライブラリのデファクトスタンダードです。Message Passingとは、メッセージによってデータをやりとりする通信方式のことで、この通信によりノード間で並列計算に必要なデータの送受信を行います。

MPIは個人や研究室で使用する小規模のクラスタから、大学の計算機センターが提供する大規模なクラスタまで、同じソースコードで実行することができ、スケーラビリティが高いです。

HPCクラスタではMPIがデファクトスタンダードとなっています。

MPIについての詳細は2.3節にて述べます。

## 本書について

本書は並列プログラムの記述方法と、MPIを使った簡単な並列計算プログラミングの習得を目的としています。また、読者として情報工学科の学生を対象としています。

2章では、並列プログラムの記述方法について述べます。プログラミング言語、言語拡張、ライブラリについて、概要とサンプルコードを紹介します。

3章では、実際にMPIを使う方法について紹介します。また、よく使われるバインディングについて実例を交えて説明します。理解を深めるための練習問題を用意しているので、是非挑戦してみて下さい。

# 並列プログラミング

## 2.1 プログラミング言語

この節では、言語仕様で並列プログラムを記述する例としてErlangとGoを紹介します。

ErlangやGoでは、言語の特徴(パターンマッチ、クロージャなど)と組み合わせて並列プログラムを実装することが簡単にできます。

**Erlang**

Erlang(アーラン)はスウェーデンの通信機器メーカーであるエリクソンが開発し、現在はオープンソースとして公開されているプログラミング言語です。

Erlangの主な特徴は下記の通りです。

- 関数型

　- 動的型付け

　- 正格(eager)評価

　- 単一代入

- ガーベージコレクタ付き

- メッセージの受信部分がパターンマッチング

- フォールトトレラント

- 並列計算

　- 独自の軽量プロセス

　- アクターモデル

並列計算に関係する特徴について説明します。

Erlangの軽量プロセスは、OSのプロセスやスレッドではない、プログラミング言語内独自の処理単位です。

Erlangにおけるアクターは下記の通りです。

- 軽量プロセスはアクター

- 新たなアクターを生成 (spawn)

- 他のアクターにメッセージを非同期に送信 (!)

　- 受信するアクターのメールボックスに送信

- 受信したメッセージによって処理を決定 (receive)

**Erlangのコード例**

erlang\_sample.erl

|  |
| --- |
| 1: -module(erlang\_sample).  2: -compile(export\_all).  3:  4: loop(X) ->  5: receive  6: 1 -> io:format("1~n"), % 整数1受信時  7: loop(X);  8: "Finish" -> io:format("Finish~n"), % 文字列"Finish"受信時  9: loop(X);  10: Other -> io:format("Received:~p~n",[Other]), %上記以外受信時  11: loop(X)  12: end.  13:  14: start() ->  15: Pid = spawn(fun() -> loop([]) end),  16: Pid ! 1, % 整数1送信  17: Pid ! [], % 空リスト送信  18: Pid ! "Finish". % 文字列"Finish"送信 |

下記実行を行うと、メインプロセスはstart() から開始します。

15行目の spawn() にて、loop() を評価する軽量プロセス Pid を生成します。

16行目で、軽量プロセスPidに対して整数1を送信します。 !はErlangの送信演算子です。

17行目で、Pidに対して空リストを送信します。

18行目で、Pidに対して文字列”Finish”を送信します。

軽量プロセスPidはメインプロセスから生成された後、4行目のloop()を実行します。

5行目から12行目の処理がパターンマッチです。パターンマッチにより、整数1を受信した場合は6行目の受信処理(“1”を出力)を行い、7行目で受信に戻ります。

文字列”Finish”を受信した場合は8行目の受信処理(“Finish”の出力)を行い、9行目で受信に戻ります。

1,”Finish”以外を受信した場合は10行目の受信処理(“Received:”)を行い、11行目で受信に戻ります。

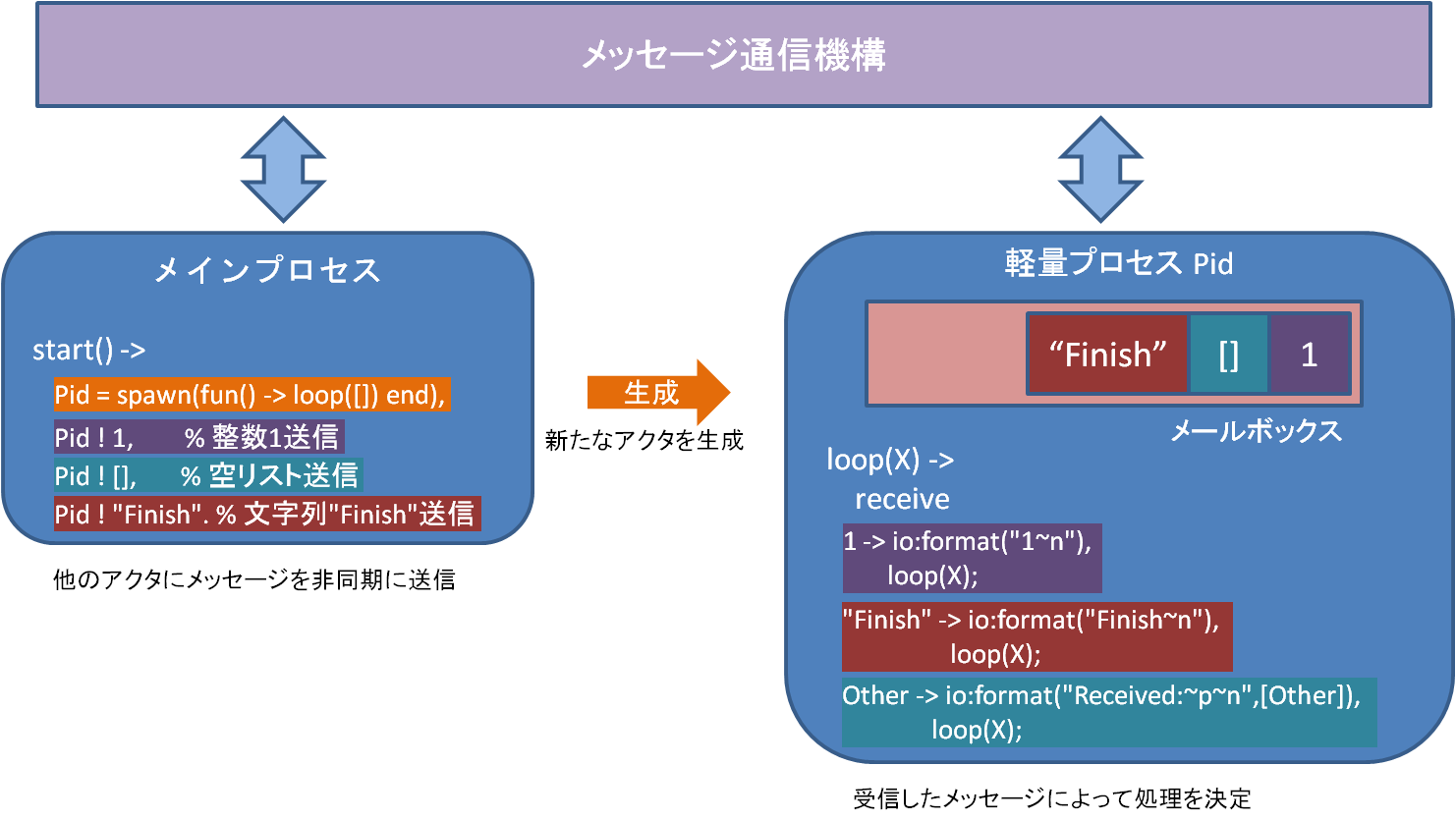


図 5 Erlang並列処理の実行例

このように、Erlangでは送受信処理を簡単に記述することが可能です。

**実行結果\***

|  |
| --- |
| $ erlc ./erlang\_sample.erl  $ erl -noshell -s erlang\_sample start -s init stop  1  Received:[]  Finish |

メインプロセスから送信したデータを、軽量プロセスで受信していることが分かります。

\* erlc、erl はErlangのコンパイルと実行のためのコマンドですが、本書の範疇を越えるため説明は省略します。

**Go**

GoはGoogleが開発し、現在はオープンソースとして公開されているプログラミング言語です。

Goの主な特徴は下記の通りです。

- 静的型付け

- 型安全

- ガベージコレクタ付き

- 並列計算

　- goroutine 軽量プロセス

　- チャネルによる通信

並列計算に関係する特徴について説明します。

goroutineは、OSのプロセスやスレッドではない、プログラミング言語内独自の処理単位です。

goroutineは、チャネル通信によってデータの送受信、および同期を行います。チャネル通信はチャネル変数への読み書きによって行います。読み込みが受信、書き込みが送信となります。

**Goのコード例**

go\_sample.go

|  |
| --- |
| 1 : package main  2 :  3 : import (  4 : "fmt"  5 : "time"  6 : )  7 :  8 : const DATA\_NUM = 10 // メッセージの数  9 : const WORKER\_NUM = 3 // 軽量プロセスの数  10 :  11 : // i: プロセス番号  12 : // c: チャネル変数  13 : func worker(i int, c chan int) {  14 : for {  15 : d := <- c // メッセージ受信  16 : fmt.Println("[",i,"] work!",d)  17 : time.Sleep(1)  18 : }  19 : }  20 :  21 : func main() {  22 : c := make(chan int, DATA\_NUM) // チャネル変数の生成  23 :  24 : // 軽量プロセスの生成  25 : for i := 0; i < WORKER\_NUM; i++ {  26 : go worker(i, c)  27 : }  28 :  29 : // メッセージ送信  30 : for i := 0; i < DATA\_NUM; i++ {  31 : c <- i  32 : }  33 :  34 : time.Sleep(1000\*time.Millisecond)  35 : } |

メインプロセスは21行目のmain()から実行を開始します。

22行目のmake(chan int, DATA\_NUM)により、int型変数を送受信するためのチャネル変数cを生成します。第二引数のDATA\_NUMで、チャネルのバッファサイズを指定します。

26行目でworker()を実行する軽量プロセスを生成します。

メインプロセスは31行目でチャネル変数cにint型のデータを送信します。メインプロセスは、チャネルのバッファがいっぱいになるまで送信できます。サンプルコードでは、バッファサイズがDATA\_NUM個のチャネルに、DATA\_NUM個のデータを送信しています。もし、DATA\_NUM個より多くのデータをチャネルに送信する場合は、軽量プロセスがチャネルからデータを受信し、バッファに空きができるまで送信を待ちます。

34行目で軽量プロセスの標準出力を待ってから終了します。1000\*time.Millisecondで1秒待ちます。

軽量プロセスは13行目のworker()を実行します。

15行目でチャネル変数cからint型のデータを受信します。軽量プロセスは、メインプロセスがチャネル変数cにデータを送信するまで、待ちます。

14行目のforループにより、軽量プロセスはデータの受信に戻ります。

このコードでは、処理(“Work!”の出力) が終わった軽量プロセスが、次のデータを受信して処理を行います。

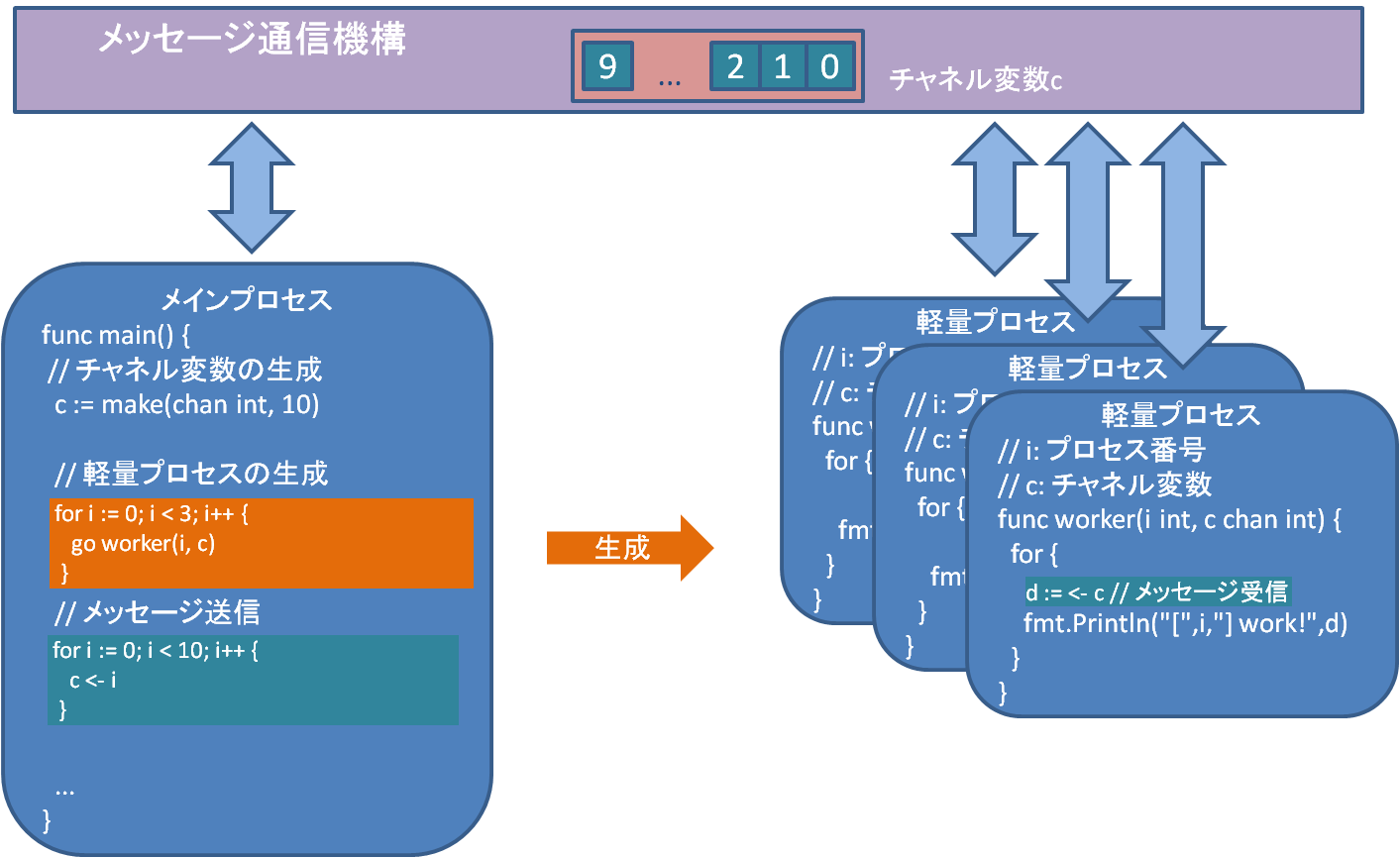


図 6 Go並列処理の実行例

**実行結果 \***

|  |
| --- |
| $ go run ./go\_sample.go  [ 0 ] work! 0  [ 1 ] work! 1  [ 2 ] work! 2  [ 0 ] work! 3  [ 1 ] work! 4  [ 2 ] work! 5  [ 0 ] work! 6  [ 1 ] work! 7  [ 2 ] work! 8  [ 0 ] work! 9 |

軽量プロセス0,1,2 がデータを受信していることが分かります。

なお、実行結果は環境によって異なります。

\* go run についてはgoのコンパイルと実行のコマンドですが、本書の範疇を越えるため、説明を省略します。

**ErlangとGoの違い**

ErlangとGoは、下記点が異なります。

- Erlang

- プロセスを指定してデータを送信 (Pid ! Message)

- 特定のプロセスへ終了要求を送信するなどの実装が容易

- Go

- チャネル変数を指定してデータを送信 (c<-i)

- 処理が終わったプロセスが次のデータを受信し、その処理をするなどの実装が容易

## 2.2 言語拡張

この節では、言語拡張によって並列プログラムを記述する例としてOpenMPを紹介します。

**OpenMP**

OpenMP は並列計算のための言語拡張です。

OpenMP を使用することで、逐次プログラムをほぼ書き直すことなしに、並列化できます。

ソースコード中の並列処理を行いたい箇所にOpenMP用のディレクティブ(#pragma ompなど)を記述するだけで、並列化することが可能です。

GCC、Intel C/C++/Fortran、Microsoft Visual C++、PGI C/FortranなどのコンパイラがOpenMPをサポートしています。

**OpenMPのコード例**

openmp\_sample.c

|  |
| --- |
| 1: #include <stdio.h>  2: #include <omp.h>  3:  4: int main(void)  5: {  6: const int NUM=12; /\* データ数 \*/  7: int data[NUM];  8: int i;  9:  10: #pragma omp parallel /\* 続くブロックを並列化 \*/  11: printf("[%d,%d] hello world\n",  12: omp\_get\_thread\_num(), omp\_get\_num\_threads());  13:  14: #pragma omp parallel for /\* 続くforループを分割して並列化 \*/  15: for(i = 0; i < NUM; i++) {  16: data[i] = omp\_get\_thread\_num();  17: }  18:  19: for(i = 0; i < NUM; i++) {  20: printf("data[%d]=%d\n", i, data[i]);  21: }  22:  23: return 0;  24: } |

10行目の #pragma omp parallel で、直後のブロックを環境変数OMP\_NUM\_THREADSの数で並列化します。

omp\_get\_thread\_num() はスレッド番号(0～OMP\_NUM\_THREAS-1)を取得する関数です。omp\_get\_num\_threads()はスレッド数(OMP\_NUM\_THREADS)を取得する関数です。

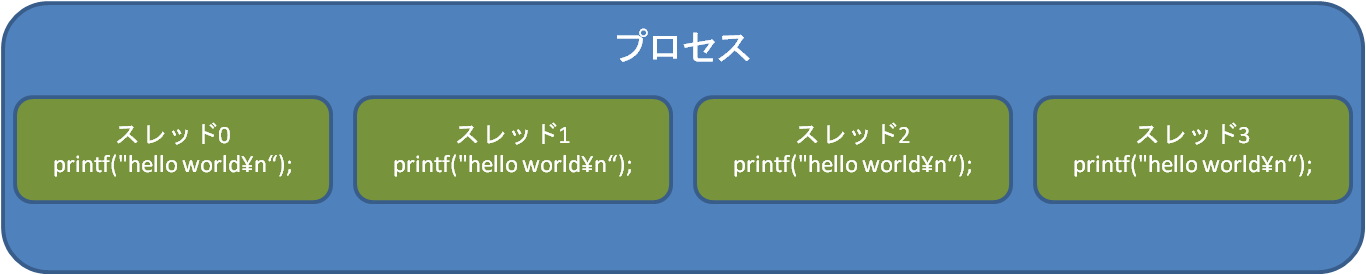
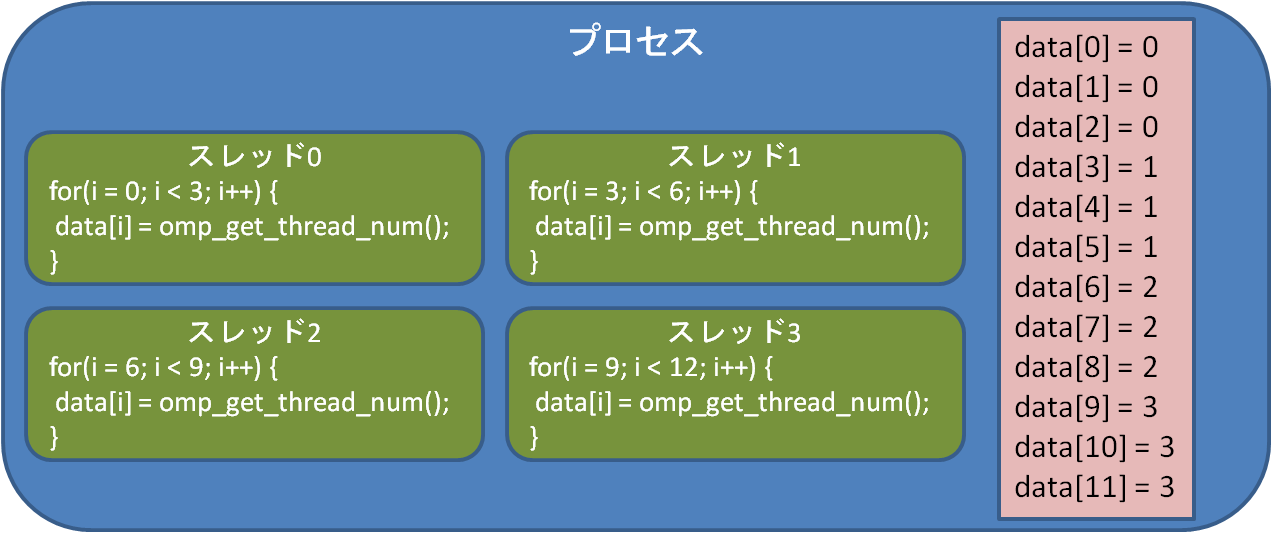


図 3 OpenMP(#pragma omp parallel)の実行例

14行目の #pragma omp parallel for で、直後のforループを環境変数OMP\_NUM\_THREADSの数に分割します。

どのように分割されたのかを確認するために data[i] にスレッド番号を入れています。



**図 4 OpenMP(#pragma omp parallel for)の実行例**

10行目、14行目を削除すると逐次処理のコードになります。

**実行結果\***

|  |
| --- |
| $ gcc -fopenmp -o openmp\_sample openmp\_sample.c  $ OMP\_NUM\_THREADS=4 ./openmp\_sample  [1,4] hello world  [2,4] hello world  [0,4] hello world  [3,4] hello world  data[0]=0  data[1]=0  data[2]=0  data[3]=1  data[4]=1  data[5]=1  data[6]=2  data[7]=2  data[8]=2  data[9]=3  data[10]=3  data[11]=3 |

10行目の #pragma omp parallel により、スレッド番号が異なる4つのhello worldが表示されていることが分かります。

14行目の #pragma omp parallel for により、12回のループが4つ(iが“0,1,2”、”3,4,5”、”6,7,8”、”9.10,11”の場合)に分割されて実行されていることが分かります。

なお、実行結果は環境によって異なります。

\* gcc -fopenmp についてはOpenMP用のコンパイル方法ですが本書の範疇を越えるため、説明を省略します。

## 2.3 ライブラリ

この節では、ライブラリを利用して並列プログラムを記述する方法として、MPIとその実装であるOpen MPI、MPICH2について紹介します。

**MPI**

MPIは、1章でも述べたように並列計算のための通信ライブラリのデファクトスタンダードです。

MPIではC、C++、Fortran用のバインディングが標準化されています。その他の言語(C#、Java、Pythonなど)では上記言語のバインディングを使用します。MPIライブラリは一般的なOS(Windows、Linux、Mac OS X、さまざまなUNIX)で利用可能です。

図7は、MPIライブラリが提供する MPI\_Send()、MPI\_Recv()を使ってプログラム間でメッセージパッシングを行う例です。

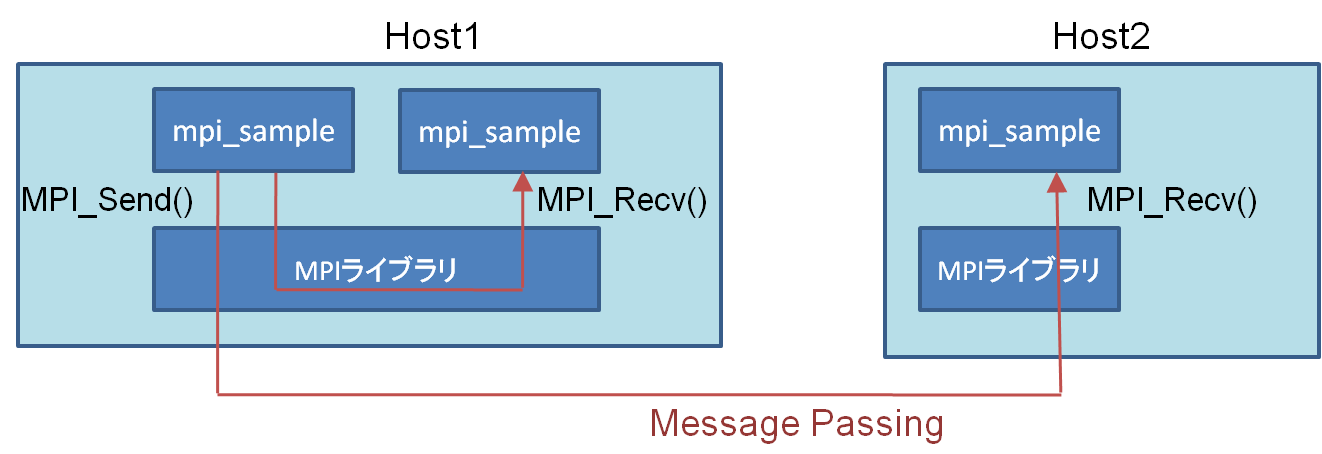


図 7 MPIによるメッセージパッシング

MPIの特徴は下記の通りです。

- 複数プロセスが通信しあって計算をすすめる

- 処理の単位はOSが管理するプロセス

- ランクは通信の相手を一意に指定するための整数

- MPIが実行ファイルを異なる機械へ送って、実行を開始

**MPIの歴史**

MPIフォーラム設立以前

　　計算機ベンダ毎に並列計算ライブラリの仕様が異なる。

1980年代末期PVM(Parallel Virtual Machine)というライブラリもあったが

大成功しなかった。

　1992年　MPIフォーラム設立

　　産業界、大学、政府系研究機関からなる国際フォーラムにより、

　　共通規格の検討を開始

　1994年　MPI-1 リリース

　　1対1通信、集団通信、コミュニケータ、派生データ型などの規格

1対1通信：送信、受信など

集団通信：ブロードキャストなど

コミュニケータ：通信を行うプロセスのグループ

派生データ型：独自のデータ型を定義(例：構造隊)

　1997年　MPI-2 リリース

　　並列入出力、リモートメモリ操作、動的プロセス生成などの規格

並列入出力：複数プロセスの同一ファイルアクセスなど

リモートメモリ操作：別プロセスのメモリにアクセス

動的プロセス生成：MPI通信を行うプロセスを動的に生成

　2012年　MPI-3 リリース

　　ノンブロッキング集団通信、Neighborhood集団通信などの規格

ノンブロッキング集団通信：非同期ブロードキャストなど

Neighborhood集団通信：指定したトポロジ上で隣接するノードへの通信など

本書の3章では、MPI-1について紹介します。

**Open MPI**

Open MPIは複数のMPIライブラリ(FT-MPI, LA-MPI, LAM/MPI, PACX-MPI)を引き継いで実装されたオープンソースのMPIライブラリです。

Open MPI は多くのTOP500スーパーコンピュータで使われおり、2008年6月から2009年11月まで世界最速だったRoadrunner、2011年6月から2012年6月まで世界最速だった理化学研究所の京がOpen MPIを使用しています。

**MPICH2**

MPICH2はアルゴンヌ国立研究所が開発したMPIライブラリです。

MPICH2はIBM MPI、Intel MPI、Cray MPI、Microsoft MPI などの多くのMPI実装の基盤となっています。

Open MPI、MPICH2はどちらも同じMPI規格に従って実装されているため、同じソースコードでOpen MPI、MPICH2を利用できます。

Open MPI、MPICH2はどちらも、コンパイルにmpicc、実行にmpirunもしくはmpiexecを使用します。Open MPIのmpicc、mpirun、mpiexecとMPICH2のmpicc、mpirun、mpiexecは、基本的には同じオプションを使えますが、一部異なるオプションがあります。詳しくはマニュアルを参照下さい。

Open MPIとMPICH2のマニュアルは下記となります。

　Open MPIマニュアル

<https://www.open-mpi.org/doc/current/>

　MPICH2マニュアル

<http://www.mpich.org/static/docs/latest/>

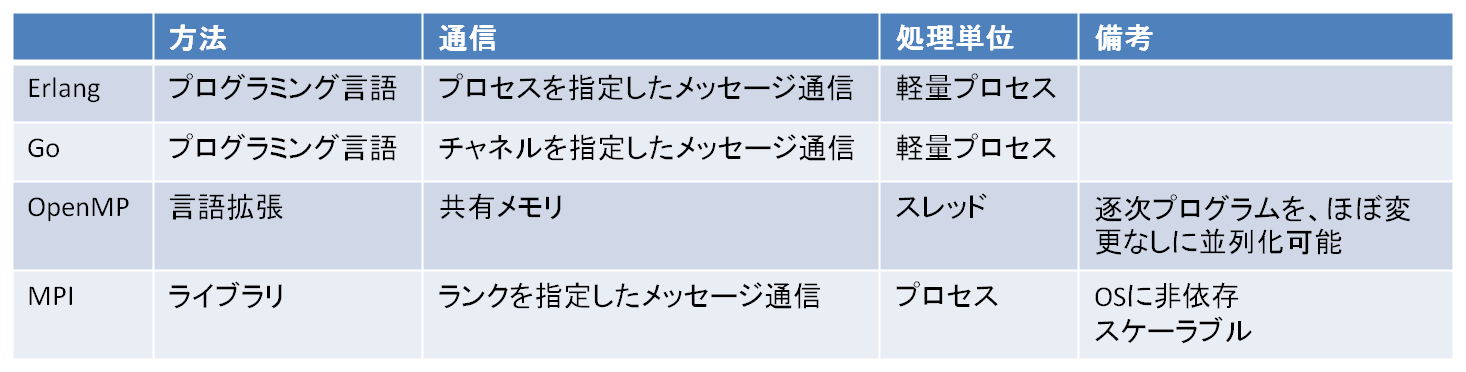
MPIのソースコードとコンパイル、実行方法については3章で紹介します。

マルチホストで実行するための設定ファイルについてもOpen MPIとMPICH2で記述方法が一部異なるため、注意が必要です。マルチホストで実行するための方法については3.3節を参照下さい。

## 2.4 まとめ

本章で紹介した並列プログラミングについて表1にまとめます。

**表1. 並列プログラミングのまとめ**



# 3. MPI入門

この章では、MPI-1の基本的なC言語バインディングについて例を挙げて説明します。

各関数の詳細については章末のC言語バインディングリファレンスを参照下さい。

本書のサンプルコードは Open MPI-1.6.5 で動作を確認しました。

## 3.1 Hello World

本節では、簡単なMPIプログラムのビルド、実行の方法について紹介します。

**mpi\_hello.c**

|  |
| --- |
| 1 : #include <stdio.h>  2 : #include <mpi.h>  3 :  4 : int main(int argc, char \*argv[])  5 : {  6 : int rank, size;  7 : int ret;  8 :  9 : /\* MPI実行環境の初期化処理 \*/  10 : ret = MPI\_Init(&argc, &argv);  11 : if (ret != MPI\_SUCCESS) {  12 : printf("Error: MPI\_Init(): %d\n", ret);  13 : }  14 :  15 : /\* ランク番号の取得 \*/  16 : ret = MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  17 : if (ret != MPI\_SUCCESS) {  18 : printf("Error: MPI\_Comm\_rank(): %d\n", ret);  19 : }  20 :  21 : /\* サイズの取得 \*/  22 : ret = MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);  23 : if (ret != MPI\_SUCCESS) {  24 : printf("Error: MPI\_Comm\_size(): %d\n", ret);  25 : }  26 :  27 : printf("[rank%d,size%d] Hello World!!\n", rank, size);  28 :  29 : /\* MPI実行環境の終了処理 \*/  30 : MPI\_Finalize();  31 :  32 : return 0;  33 : } |

**10行目 MPI\_Init()**

MPI 実行環境の初期化処理を行います。MPI\_Comm\_rank(),MPI\_Comm\_size()などを使用するために必要となります。

**16行目 MPI\_Comm\_rank()**

mpirun -np 10 で生成された全てのプロセス群を対象(MPI\_COMM\_WORLD)に、ランク番号を取得します。

ランク番号はプロセス毎に異なる値を持ちます。"mpirun -np 10"で実行した場合、ランク番号は0〜9のいずれかの値となります。

**22行目 MPI\_Comm\_size()**

サイズ(プロセス数)を取得します。"mpirun -np 10"の場合、10となります。

**30行目 MPI\_Finalize()**

MPI実行環境の終了処理を行います。

**ビルド方法**

|  |
| --- |
| $ mpicc -o mpi\_hello mpi\_hello.c |

mpicc のオプションについては付録を参照して下さい。

**実行方法**

|  |
| --- |
| $ mpirun -np 10 ./mpi\_hello |

mpirun –np 10 によって 10個のプロセスが生成されます。

mpirun のオプションについては付録を参照して下さい。

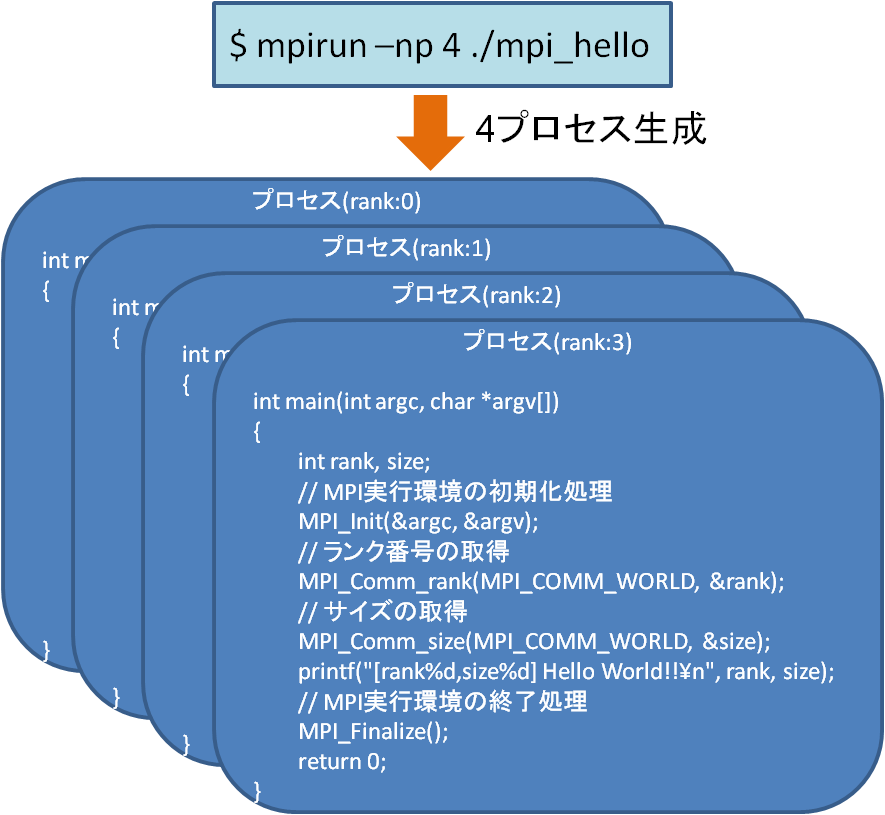


図 8 MPI(Hello World)の実行例(プロセス数4)

**実行結果**

|  |
| --- |
| [rank0,size10] Hello World!!  [rank2,size10] Hello World!!  [rank3,size10] Hello World!!  [rank4,size10] Hello World!!  [rank5,size10] Hello World!!  [rank6,size10] Hello World!!  [rank9,size10] Hello World!!  [rank8,size10] Hello World!!  [rank1,size10] Hello World!!  [rank7,size10] Hello World!! |

ランク番号の異なる10の出力が確認できます。

なお、実行結果は環境によって異なります。

**練習問題**

1. ランク番号が偶数のプロセスだけ"Hello World!!"を表示するように修正して下さい。

## 3.2 メッセージ通信

本節では、メッセージ通信について紹介します。

下記にMPIプログラムの定石を記します。

- 通常はすべてのランクで同一の実行ファイルを使用ことが多い

- 通常はランク0を特別扱いすることが多い

**Send/Recv**

Send/Recvを使った1対1のメッセージ通信を紹介します。

**mpi\_send\_recv.c**

|  |
| --- |
| 1 : #include <stdio.h>  2 : #include <mpi.h>  3 :  4 : #define DATA\_NUM (3) /\* データ数 \*/  5 : #define TAG (0) /\* タグ \*/  6 :  7 : static void setData(long\* data, int num, int rank)  8 : {  9 : int i;  10 : for (i=0; i<num; i++) {  11 : data[i] = rank;  12 : }  13 : }  14 :  15 : static void printData(const long\* data, int num)  16 : {  17 : int i,ret;  18 : int rank;  19 :  20 : ret = MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  21 : if (ret != MPI\_SUCCESS) {  22 : printf("Error: MPI\_Comm\_rank(): %d\n", ret);  23 : }  24 :  25 : for (i=0; i<num; i++) {  26 : printf("rank%d: %ld\n", rank, data[i]);  27 : }  28 : }  29 :  30 : static void sendTo(int toRank)  31 : {  32 : int ret;  33 : long data[DATA\_NUM];  34 :  35 : /\* data[]をtoRankで埋める \*/  36 : setData(data, DATA\_NUM, toRank);  37 :  38 : /\* toRankランクのプロセスへ、dataの送信 \*/  39 : ret = MPI\_Send(data, DATA\_NUM, MPI\_LONG, toRank, TAG,  40 : MPI\_COMM\_WORLD);  41 : if (ret != MPI\_SUCCESS) {  42 : printf("Error: MPI\_Send(): %d\n", ret);  43 : }  44 : }  45 :  46 : static void recvFrom(int fromRank)  47 : {  48 : int ret;  49 : long data[DATA\_NUM];  50 : MPI\_Status s;  51 :  52 : /\* fromRankランクのプロセスから、dataの受信 \*/  53 : ret = MPI\_Recv(data, DATA\_NUM, MPI\_LONG, fromRank, TAG,  54 : MPI\_COMM\_WORLD, &s);  55 : if (ret != MPI\_SUCCESS) {  56 : printf("Error: MPI\_Recv(): %d\n", ret);  57 : }  58 :  59 : /\* data[]を表示する \*/  60 : printData(data, DATA\_NUM);  61 : }  62 :  63 : int main(int argc, char \*argv[])  64 : {  65 : int rank;  66 : int ret;  67 :  68 : ret = MPI\_Init(&argc, &argv);  69 : if (ret != MPI\_SUCCESS) {  70 : printf("Error: MPI\_Init(): %d\n", ret);  71 : }  72 :  73 : ret = MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  74 : if (ret != MPI\_SUCCESS) {  75 : printf("Error: MPI\_Comm\_rank(): %d\n", ret);  76 : }  77 :  78 : /\* rank0のプロセスからそれ以外のプロセスへデータを送信する \*/  79 : if (rank == 0) {  80 : /\* rank0のプロセスは下記を実行 \*/  81 : int size;  82 : int i;  83 : ret = MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);  84 : if (ret != MPI\_SUCCESS) {  85 : printf("Error: MPI\_Comm\_size(): %d\n", ret);  86 : }  87 : for (i=1; i<size; i++) {  88 : sendTo(i);  89 : }  90 : } else {  91 : /\* rank0以外のプロセスは下記を実行 \*/  92 : recvFrom(0);  93 : }  94 :  95 : MPI\_Finalize();  96 :  97 : return 0;  98 : } |

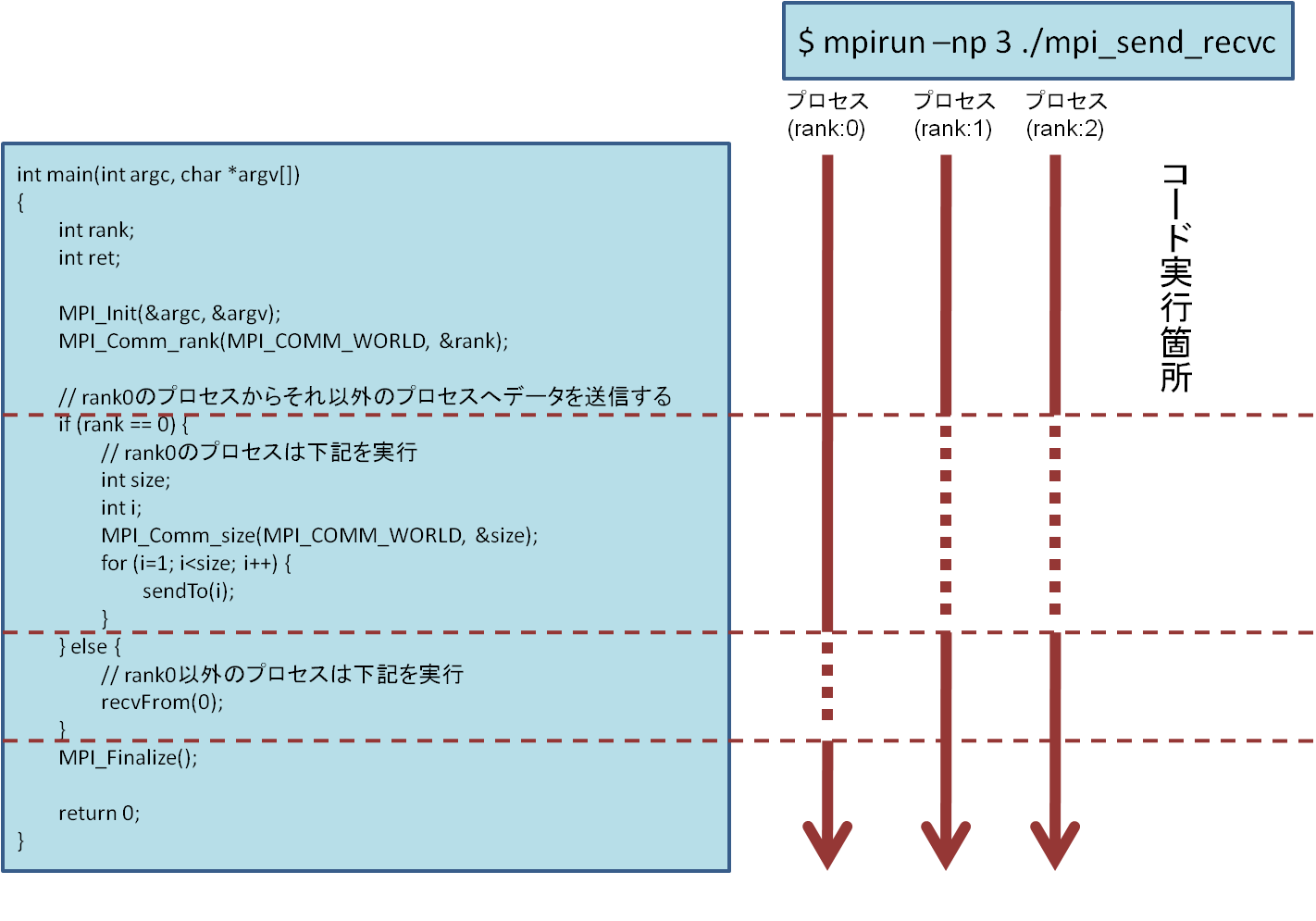


図 9 MPI(Send/Recv) の実行例1

**39行目 MPI\_Send()**

MPI\_Send(data, DATA\_NUM, MPI\_LONG, toRank, TAG,

MPI\_COMM\_WORLD);

toRankランクのプロセスへ、dataアドレスから始まるDATA\_NUM個のlong型データを送信します。

送信メッセージにはTAGを関連付けます。

**53行目 MPI\_Recv()**

MPI\_Recv(data, DATA\_NUM, MPI\_LONG, fromRank, TAG,

MPI\_COMM\_WORLD, &s);

fromRankランクのプロセスからTAGに関連づけられたdataアドレスから始まるDATA\_NUM個のlong型データを受信します。

受信情報(送信元ランク番号、タグ、エラーコードなど)はsに入ります。

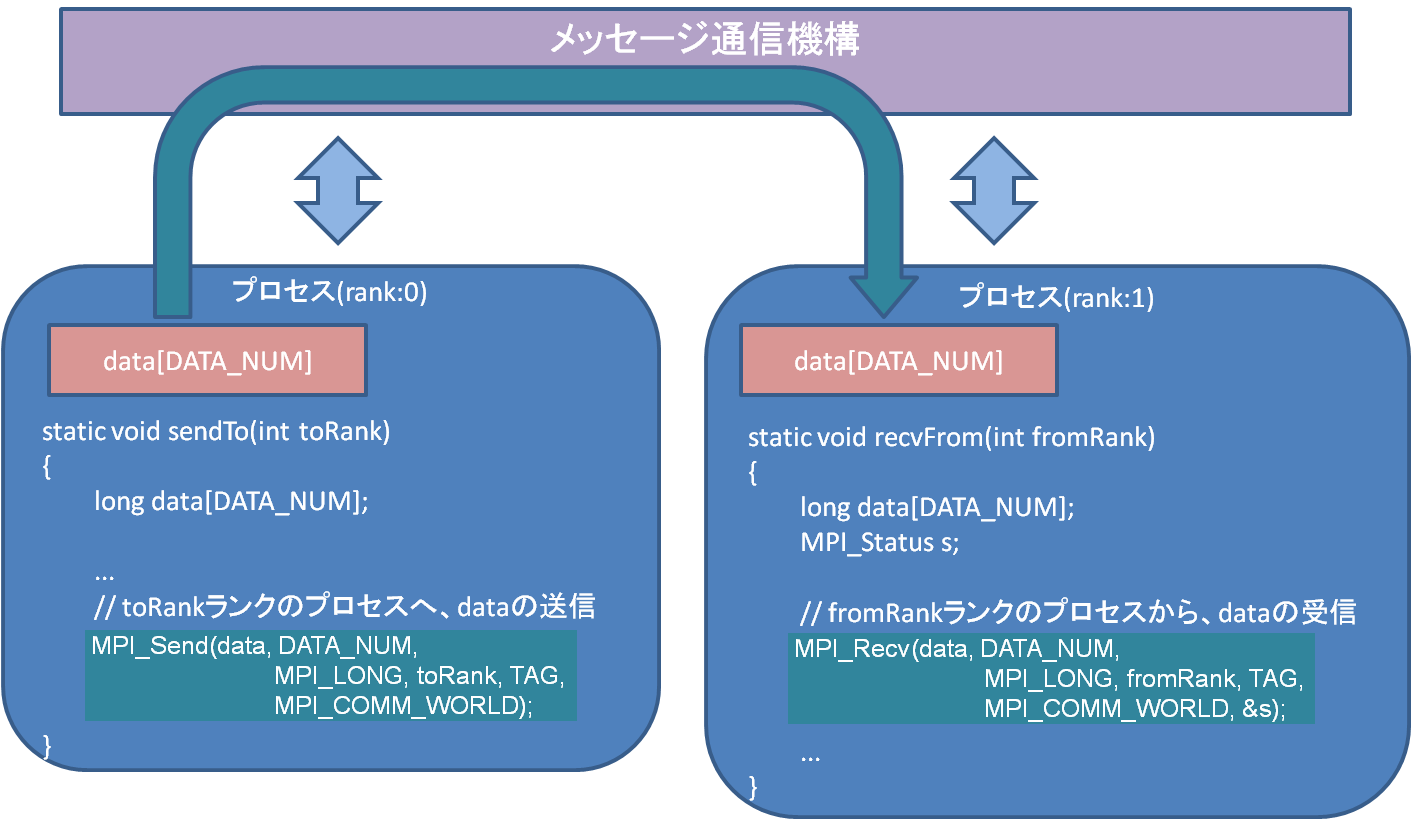


図 10 MPI(Send/Recv) の実行例2

**ビルド方法**

|  |
| --- |
| $ mpicc -o mpi\_send\_recv mpi\_send\_recv.c |

**実行方法**

|  |
| --- |
| $ mpirun -np 10 ./mpi\_send\_recv |

**実行結果**

|  |
| --- |
| rank7: 7  rank7: 7  rank7: 7  rank4: 4  rank4: 4  rank4: 4  rank5: 5  rank5: 5  rank5: 5  rank9: 9  rank9: 9  rank9: 9  rank8: 8  rank8: 8  rank8: 8  rank6: 6  rank6: 6  rank6: 6  rank2: 2  rank2: 2  rank2: 2  rank1: 1  rank1: 1  rank1: 1  rank3: 3  rank3: 3  rank3: 3 |

rank0以外のプロセスがrank0のプロセスからデータを受信することが確認できます。

なお、実行結果は環境によって異なります。

**練習問題**

1. double型のデータを送受信するように修正して下さい。

2. MPI\_Send()とMPI\_Recv()で異なるタグを指定した場合、どのような動作になるかを確認して下さい。

3. 受信データ数を MPI\_Get\_count()を用いて取得して下さい。

4. 送受信するデータ数を2個に修正して下さい。受信するバイト数を計算して下さい。

**Scatter**

Scatter関数を使うと、あるプロセスから全プロセスにメッセージを送信することができます。

**mpi\_scatter.c**

|  |
| --- |
| 1 : #include <stdio.h>  2 : #include <stdlib.h>  3 : #include <mpi.h>  4 :  5 : #define PROC\_NUM (3) /\* プロセス数 \*/  6 : #define DATA\_NUM (3) /\* データ数 \*/  7 : #define FROM\_RANK (0) /\* 送信を行うプロセスのランク番号 \*/  8 :  9 : static long sendData[DATA\_NUM\*PROC\_NUM]; /\* FROM\_RANK プロセスで使用 \*/  10 :  11 : static void setData(long\* data, int num)  12 : {  13 : int i;  14 : for (i=0; i<num; i++) {  15 : data[i] = i;  16 : }  17 : }  18 :  19 : static void printData(int rank, const long\* data, int num)  20 : {  21 : int i;  22 : for (i=0; i<num; i++) {  23 : printf("rank%d: %ld\n",rank, data[i]);  24 : }  25 : }  26 :  27 : static void prepareSendData(long \*data)  28 : {  29 : int size;  30 : int i, ret;  31 :  32 : ret = MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);  33 : if (ret != MPI\_SUCCESS) {  34 : printf("Error: MPI\_Comm\_size(): %d\n", ret);  35 : }  36 :  37 : for (i=0; i<size; i++) {  38 : setData(data+i\*DATA\_NUM, DATA\_NUM);  39 : }  40 : }  41 :  42 : static void scatter(void)  43 : {  44 : int ret;  45 : int rank;  46 : long recvData[DATA\_NUM] = {}; /\* 全てのプロセスで使用 \*/  47 :  48 : ret = MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  49 : if (ret != MPI\_SUCCESS) {  50 : printf("Error: MPI\_Comm\_rank(): %d\n", ret);  51 : }  52 :  53 : if (rank == FROM\_RANK) {  54 : /\* sendData[]を埋める \*/  55 : prepareSendData(sendData);  56 : }  57 :  58 : /\* FROM\_RANKプロセスのsendDataを、全てのプロセスの  59 : \* recvDataへばらまく(scatter) \*/  60 : ret = MPI\_Scatter(sendData, DATA\_NUM, MPI\_LONG,  61 : recvData, DATA\_NUM, MPI\_LONG,  62 : FROM\_RANK, MPI\_COMM\_WORLD);  63 : if (ret != MPI\_SUCCESS) {  64 : printf("Error: MPI\_Scatter(): %d\n", ret);  65 : }  66 :  67 : printData(rank, recvData, DATA\_NUM);  68 :  69 : }  70 :  71 : int main(int argc, char \*argv[])  72 : {  73 : int ret;  74 :  75 : ret = MPI\_Init(&argc, &argv);  76 : if (ret != MPI\_SUCCESS) {  77 : printf("Error: MPI\_Init(): %d\n", ret);  78 : }  79 :  80 : /\* FROM\_RANK プロセスから全てのプロセスへデータを送信する \*/  81 : scatter();  82 :  83 : MPI\_Finalize();  84 :  85 : return 0;  86 : } |

**60行目 MPI\_Scatter()**

MPI\_Scatter(sendData, DATA\_NUM, MPI\_LONG,

recvData, DATA\_NUM, MPI\_LONG,

FROM\_RANK, MPI\_COMM\_WORLD);

FROM\_RANKプロセスのsendDataを、全てのプロセスのrecvDataへばらまきます(scatter)。

この関数一つで、送受信両方の動作を行います。送受信データはDATA\_NUM個のlong型データとなります。

FROM\_RANKプロセスは、各プロセスに異なるデータを送信します。

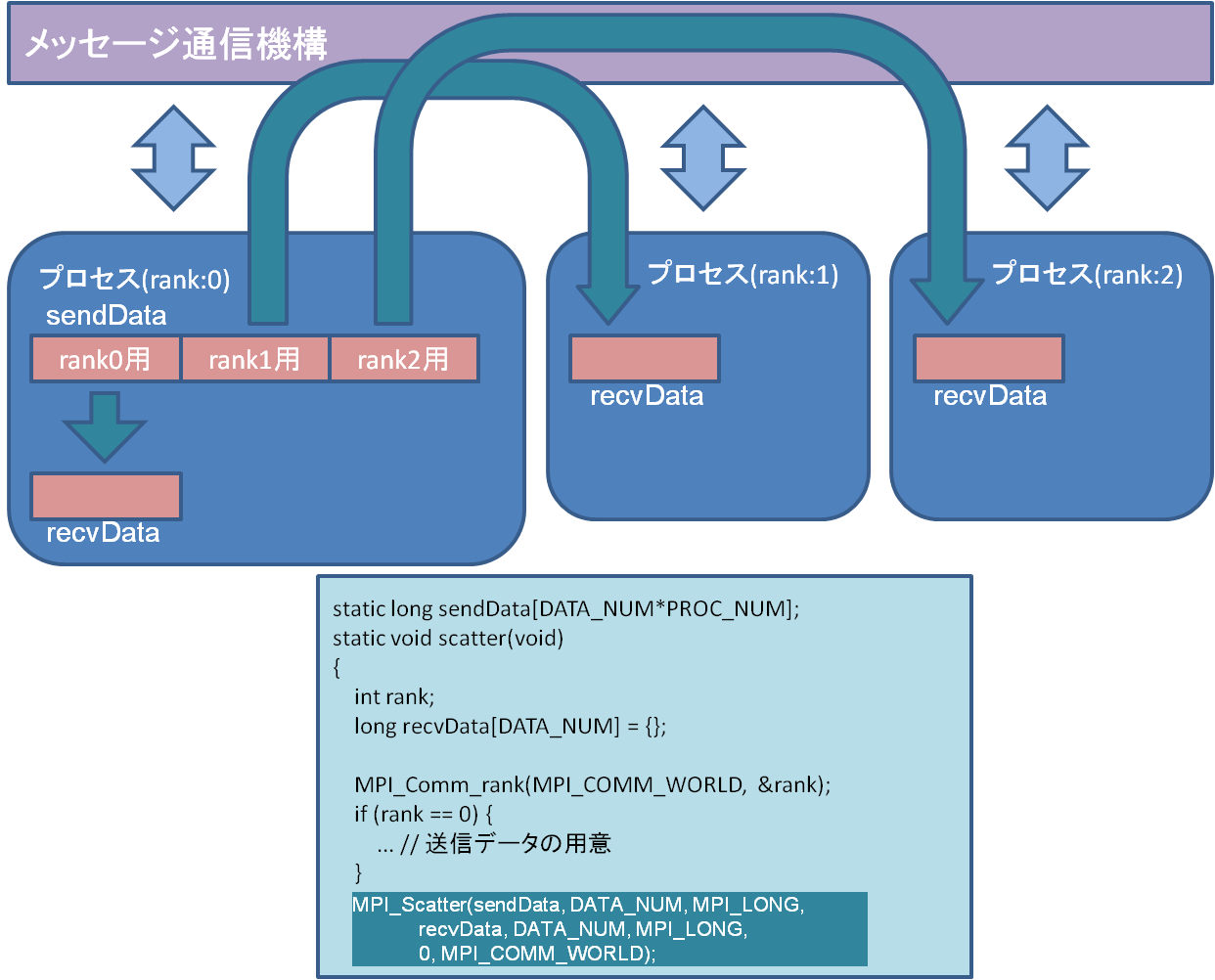


図 11 MPI(Scatter)の実行例

**ビルド方法**

|  |
| --- |
| $ mpicc -o mpi\_scatter mpi\_scatter.c |

**実行方法**

|  |
| --- |
| $ mpirun -np 10 ./mpi\_scatter |

**実行結果**

|  |
| --- |
| rank0: 0  rank0: 1  rank0: 2  rank1: 0  rank1: 1  rank1: 2  rank4: 0  rank4: 1  rank4: 2  rank2: 0  rank2: 1  rank2: 2  rank5: 0  rank5: 1  rank5: 2  rank8: 0  rank8: 1  rank8: 2  rank6: 0  rank6: 1  rank6: 2  rank3: 0  rank3: 1  rank3: 2  rank9: 0  rank9: 1  rank9: 2  rank7: 0  rank7: 1  rank7: 2 |

rank0のプロセスから全てのプロセスへデータを送信することが確認できます。

なお、表示されるrank番号の順番は環境によって異なります。

**練習問題**

1. ランク番号3のプロセスから全てのプロセスへデータを送信するように修正して下さい。

2. ランク番号1のプロセスがMPI\_Scatter()を行わないようにした場合、全体としてどのような動作となるか確認して下さい。

**Gather**

Gather関数を使うと、全プロセスから、あるプロセスにメッセージを送信することができます。

**mpi\_gather.c**

|  |
| --- |
| 1 : #include <stdio.h>  2 : #include <stdlib.h>  3 : #include <mpi.h>  4 :  5 : #define PROC\_NUM (3) /\* プロセス数 \*/  6 : #define DATA\_NUM (3) /\* データ数 \*/  7 : #define TO\_RANK (0) /\* 受信を行うプロセスのランク番号 \*/  8 :  9 : static long recvData[DATA\_NUM\*PROC\_NUM]; /\* TO\_RANK プロセスで使用 \*/  10 :  11 : static void setData(long\* data, int num)  12 : {  13 : int i;  14 : int rank;  15 :  16 : MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  17 : for (i=0; i<num; i++) {  18 : data[i] = rank;  19 : }  20 : }  21 :  22 : static void printData(int rank, const long\* data, int num)  23 : {  24 : int i;  25 : for (i=0; i<num; i++) {  26 : printf("rank%d: %ld\n",rank, data[i]);  27 : }  28 : }  29 :  30 : static void freeSendData(long \*sendData)  31 : {  32 : free(sendData);  33 : }  34 :  35 : static void gather(void)  36 : {  37 : int ret;  38 : int rank;  39 : long sendData[DATA\_NUM] = {}; /\* 全てのプロセスで使用 \*/  40 :  41 : ret = MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  42 : if (ret != MPI\_SUCCESS) {  43 : printf("error: mpi\_comm\_rank(): %d\n", ret);  44 : }  45 :  46 : /\* sendData[]を埋める \*/  47 : setData(sendData, DATA\_NUM);  48 :  49 : /\* 全てのプロセスのsendDataを、TO\_RANKプロセスの  50 : \* recvDataへ集める(gather) \*/  51 : ret = MPI\_Gather(sendData, DATA\_NUM, MPI\_LONG,  52 : recvData, DATA\_NUM, MPI\_LONG,  53 : TO\_RANK, MPI\_COMM\_WORLD);  54 : if (ret != MPI\_SUCCESS) {  55 : printf("Error: MPI\_Gather(): %d\n", ret);  56 : }  57 :  58 : if (rank == TO\_RANK) {  59 : int size;  60 : MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);  61 : printData(rank, recvData, DATA\_NUM\*size);  62 : }  63 : }  64 :  65 : int main(int argc, char \*argv[])  66 : {  67 : int ret;  68 :  69 : ret = MPI\_Init(&argc, &argv);  70 : if (ret != MPI\_SUCCESS) {  71 : printf("Error: MPI\_Init(): %d\n", ret);  72 : }  73 :  74 : /\* 全てのプロセスからTO\_RANK プロセスへデータを送信する \*/  75 : gather();  76 :  77 : MPI\_Finalize();  78 :  79 : return 0;  80 : } |

**ビルド方法**

|  |
| --- |
| $ mpicc -o mpi\_gather mpi\_gather.c |

**実行方法**

|  |
| --- |
| $ mpirun -np 10 ./mpi\_gather |

**MPI\_Gather()**

MPI\_Gather(sendData, DATA\_NUM, MPI\_LONG,

recvData, DATA\_NUM, MPI\_LONG,

TO\_RANK, MPI\_COMM\_WORLD)

全プロセスのsendDataを、TO\_RANKのプロセスのrecvDataへ集めます(gather)。

この関数一つで、送受信両方の動作を行います。

送受信データはDATA\_NUM個のlong型データとなります。

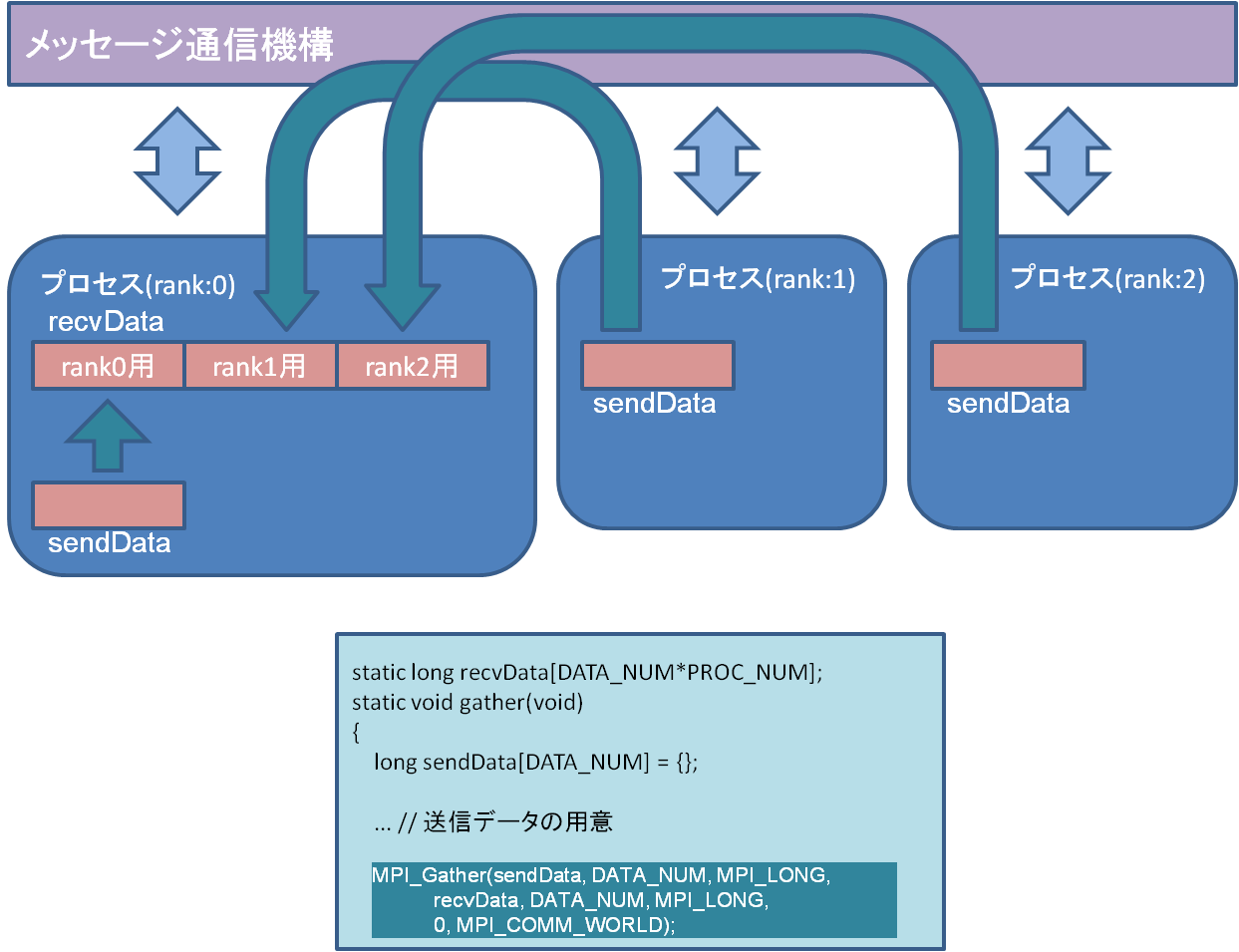


図 12 MPI(Gather)の実行例

**実行結果**

|  |
| --- |
| rank0: 0  rank0: 0  rank0: 0  rank0: 1  rank0: 1  rank0: 1  rank0: 2  rank0: 2  rank0: 2  rank0: 3  rank0: 3  rank0: 3  rank0: 4  rank0: 4  rank0: 4  rank0: 5  rank0: 5  rank0: 5  rank0: 6  rank0: 6  rank0: 6  rank0: 7  rank0: 7  rank0: 7  rank0: 8  rank0: 8  rank0: 8  rank0: 9  rank0: 9  rank0: 9 |

全てのプロセスからrank0のプロセスへデータを送信することが確認できます。

**練習問題**

1. 全てのプロセスからランク番号5のプロセスへデータを送信するように修正して下さい。

2. ランク番号1のプロセスがMPI\_Gather()を行わないようにした場合、全体としてどのような動作となるか確認して下さい。

## 3.3 マルチホスト

本節では、複数のマシンで並列計算を行う方法について紹介します。

この節で使用するコマンドは、Open MPI 1.6.5、MPICH2 1.5 で動作を確認しました。

**準備**

プログラムを実行したいサーバへ、パスワード無しで ssh 接続できるように準備します。

これについては本書の範囲を外れるため説明を省略します。

また、ローカルホストとサーバで同じMPIライブラリを使用する必要があります。MPIライブラリのバージョンが異なると問題が発生する場合があります。

そのほかに、ファイアウォールが通信を妨げることがあるので、注意が必要です。

ローカルホストとサーバでユーザ名が異なる場合やsshdのポート番号がデフォルトの22ではない場合は、~/.ssh/config へ下記を追加する必要があります。

|  |
| --- |
| Host サーバ名  User ユーザ  Port ポート番号 |

準備ができたら、下記を実行してパスワード入力無しでサーバにログインできることを確認して下さい。

|  |
| --- |
| $ ssh サーバ名 |

**ビルド方法**

以下は、ローカスホストで行います。

|  |
| --- |
| $ mpicc -o mpi\_sample mpi\_sample.c |

**実行方法**

まず、ローカルホストとサーバでプログラムを共有します。

4つの方法がよく使われます。

1. mpi\_sampleをNFSで共有する

2. scpなどで転送する

|  |
| --- |
| $ scp mpi\_sample <サーバ名>: |

3. それぞれのホストでコンパイルを行う

4. mpirun実行時に転送する

(--preload-binaryオプションはOpen MPIで使用可能です)

|  |
| --- |
| $ mpirun --preload-binary -np 10 -host <サーバ名> ./mpi\_sample |

次に、プログラムを実行します。

|  |
| --- |
| $ mpirun -np 10 -host <サーバ名>,localhost ./mpi\_sample |

上記例では、サーバとローカルホストで合わせて10個のプロセスが動作します。サーバとローカルホストのそれぞれで動作するプロセス数については、マニュアルを参照下さい。

**設定ファイル**

設定ファイルを使用して、実行するホストを指定することもできます。

Open MPIとMPICH2で設定ファイルの記述方法が異なるので注意が必要です。

下記にそれぞれの例を記します。詳細はマニュアルを参照下さい。

Open MPIの場合

|  |
| --- |
| $ cat hosts  localhost slots=X  サーバ名 slots=Y  $ mpirun -np 20 -hostfile hosts ./mpi\_sample |

MPICH2の場合

|  |
| --- |
| $ cat hosts  localhost:X  サーバ名:Y  $ mpirun -np 20 -hostfile hosts ./mpi\_sample |

一般的に、X、Yにはホストのコア数を指定します。

**環境変数**

実行時に影響を与える環境変数を記します。

MPIEXEC\_TIMEOUT

　最大実行時間(秒で指定)。指定されたMPIEXEC\_TIMEOUT秒経過すると、プログラムを終了する。MPIEXEC\_TIMEOUTを指定しない場合、タイムアウトしない。

　例) 30分でタイムアウトする場合

　　export MPIEXEC\_TIMEOUT=1800

MPIEXEC\_PORT\_RANGE

　ポート範囲指定。指定した範囲のポートでホスト間の通信を行う。

MPIEXEC\_PORT\_RANGEを指定しない場合、システムが割り当てたポートを使用する。

　　例) ポート10000～10100を使用する場合

　　　export MPIEXEC\_PORT\_RANGE=10000:10100

**練習問題**

1. これまでに作成したMPIプログラムをローカルホストとサーバの両方で実行して動作を確認して下さい。

## 3.4 C言語バインディングリファレンス

### 3.4.1 基本バインディング

**MPI\_Init()**

|  |  |
| --- | --- |
| int MPI\_Init(int \*argc, char \*\*\*argv) |  |
| int \*argc  main関数に渡されたargcのポインタ | **引数** |
| char \*\*\*argv  main関数に渡されたargvのポインタ |
| MPI\_SUCCESS  成功 | **戻り値** |
| MPI\_SUCCESS以外  失敗 |
| MPI実行環境の初期化。MPI関数を呼び出す前に実行する必要があります。 | **機能** |

**MPI\_Comm\_rank()**

|  |  |
| --- | --- |
| int MPI\_Comm\_rank(MPI\_Comm comm, int \*rank) |  |
| MPI\_Comm comm  コミュニケータ(通信可能なプロセスのグループ) | **引数** |
| int \*rank  関数を呼び出したプロセスのcomm内でのランク |
| MPI\_SUCCESS  成功 | **戻り値** |
| MPI\_SUCCESS以外  失敗 |
| comm内でのランクを取得します。通常、MPI\_COMM\_WORD を指定します。 | **機能** |

**MPI\_Comm\_size()**

|  |  |
| --- | --- |
| int MPI\_Comm\_size(MPI\_Comm comm, int \*size) |  |
| MPI\_Comm comm  コミュニケータ(通信可能なプロセスのグループ) | **引数** |
| int \*size  commに所属するプロセスの数 |
| MPI\_SUCCESS  成功 | **戻り値** |
| MPI\_SUCCESS以外  失敗 |
| commに所属するプロセスの数を取得します。通常、MPI\_COMM\_WORD を指定します。 | **機能** |

**MPI\_Finalize()**

|  |  |
| --- | --- |
| int MPI\_Finalize(void) |  |
| MPI\_SUCCESS  成功 | **戻り値** |
| MPI\_SUCCESS以外  失敗 |
| MPI実行環境を終了します。プロセスの終了前に実行する必要があります。 | **機能** |

### 3.4.2 同期通信バインディング

**MPI\_Send()**

|  |  |
| --- | --- |
| int MPI\_Send(void \*buf, int count, MPI\_Datatype datatype, int dest,  int tag, MPI\_Comm comm) |  |
| void \*buf  送信バッファの先頭アドレス | **引数** |
| int count  送信データ数  (datatypeがMPI\_LONGの場合、送信バイト数はcount\*sizeof(long)) |
| MPI\_Datatype datatype  　送信データタイプ |
| int dest  　宛先ランク |
| int tag  　タグ。メッセージの識別に利用する  (受信プロセスは、指定した送信元ランクとタグが一致するメッセージのみ受信できる) |
| MPI\_Comm comm  　コミュニケ―タ |
| MPI\_SUCCESS  成功 | **戻り値** |
| MPI\_SUCCESS以外  失敗 |
| 宛先プロセスにメッセージを送信します。  (宛先プロセスが受信するまで処理はブロッキングされます) | **機能** |

**MPI\_Recv()**

|  |  |
| --- | --- |
| int MPI\_Recv(void \*buf, int count, MPI\_Datatype datatype,  int source, int tag, MPI\_Comm comm, MPI\_Status \*status) |  |
| void \*buf  受信バッファの先頭アドレス | **引数** |
| int count  受信データ数 |
| MPI\_Datatype datatype  　受信データタイプ |
| int source  　送信元ランク |
| int tag  　タグ  (送信プロセスが指定したタグと一致するタグを指定しないと受信できません) |
| MPI\_Comm comm  　コミュニケ―タ |
| MPI\_Status \*status  ステータス(メッセージサイズ、エラーコードなど) |
| MPI\_SUCCESS  成功 | **戻り値** |
| MPI\_SUCCESS以外  失敗 |
| 送信元プロセスからメッセージを受信します。  (送信元プロセスが送信するまで処理はブロッキングされます) | **機能** |

**MPI\_Get\_count()**

|  |  |
| --- | --- |
| int MPI\_Get\_count(MPI\_Status \*status, MPI\_Datatype datatype,  int \*count) |  |
| MPI\_Status \*status  MPI\_Recv()などで取得するステータス | **引数** |
| MPI\_Datatype datatype  　受信データタイプ |
| int \*count  　受信データ数 |
| MPI\_SUCCESS  成功 | **戻り値** |
| MPI\_SUCCESS以外  失敗 |
| 受信ステータスから、受信データ数を取得します | **機能** |

### 3.4.3 非同期通信バインディング

**MPI\_Isend()**

|  |  |
| --- | --- |
| int MPI\_Isend(void \*buf, int count, MPI\_Datatype datatype, int dest,  int tag, MPI\_Comm comm, MPI\_Request \*request) |  |
| void \*buf  送信バッファの先頭アドレス | **引数** |
| int count  送信データ数 |
| MPI\_Datatype datatype  　送信データタイプ |
| int dest  　宛先ランク |
| int tag  　タグ。メッセージの識別に利用する |
| MPI\_Comm comm  　コミュニケ―タ |
| MPI\_Request \*request  　リクエスト |
| MPI\_SUCCESS  成功 | **戻り値** |
| MPI\_SUCCESS以外  失敗 |
| 宛先プロセスにメッセージを送信します。  処理はブロッキングされません。MPI\_Wait(request,status)で宛先プロセスの受信に同期します。 | **機能** |

**MPI\_Irecv()**

|  |  |
| --- | --- |
| int MPI\_Irecv(void \*buf, int count, MPI\_Datatype datatype,  int source, int tag, MPI\_Comm comm, MPI\_Request \*request) |  |
| void \*buf  受信バッファの先頭アドレス | **引数** |
| int count  受信データ数 |
| MPI\_Datatype datatype  　受信データタイプ |
| int source  　送信元ランク |
| int tag  　タグ |
| MPI\_Comm comm  　コミュニケ―タ |
| MPI\_Status \*status  ステータス(メッセージサイズ、エラーコードなど) |
| MPI\_Request \*request  　リクエスト |
| MPI\_SUCCESS  成功 | **戻り値** |
| MPI\_SUCCESS以外  失敗 |
| 送信元プロセスからメッセージを受信します。処理はブロッキングされません。MPI\_Wait(request,status)で送信元プロセスの送信に同期します。 | **機能** |

**MPI\_Wait()**

|  |  |
| --- | --- |
| int MPI\_Wait(MPI\_Request \*request, MPI\_Status \*status) |  |
| MPI\_Request \*request  　リクエスト | **引数** |
| MPI\_Status \*status  　ステータス |
| MPI\_SUCCESS  成功 | **戻り値** |
| MPI\_SUCCESS以外  失敗 |
| 非同期送信、非同期受信の完了を待ちます。 | **機能** |

### 3.4.4 集団通信バインディング

**MPI\_Scatter()**

|  |  |
| --- | --- |
| int MPI\_Scatter(void \*sendbuf, int sendcount, MPI\_Datatype sendtype,  void \*recvbuf, int recvcount, MPI\_Datatype recvtype,  int root, MPI\_Comm comm) |  |
| void \*sendbuf  　送信バッファの先頭アドレス(rootプロセスのみで使用)  送信バッファサイズは、送信データ数\*sizeof(送信データタイプ)\*全プロセス数 | **引数** |
| int sendcount  　送信データ数(rootプロセスのみで使用) |
| MPI\_Datatype sendtype  　送信データタイプ(rootプロセスのみで使用) |
| void \*recvbuf  　recvbuf 受信バッファの先頭アドレス  受信バッファサイズは受信データ数\*sizeof(受信データタイプ) |
| int recvcount  　受信データ数 |
| MPI\_Datatype recvtype  　受信データタイプ |
| int root  　送信プロセスのランク |
| MPI\_Comm comm  　コミュニケ―タ |
| MPI\_SUCCESS  成功 | **戻り値** |
| MPI\_SUCCESS以外  失敗 |
| 全てのプロセスへデータを送信します。この関数で、送信処理と受信処理の両方を行います。  指定した送信プロセスで送信処理を行い、全てのプロセスで受信処理を行います。 | **機能** |

**MPI\_Gather()**

|  |  |
| --- | --- |
| int MPI\_Gather(void \*sendbuf, int sendcount, MPI\_Datatype sendtype,  void \*recvbuf, int recvcount, MPI\_Datatype recvtype,  int root, MPI\_Comm comm) |  |
| void \*sendbuf  　送信バッファの先頭アドレス  送信バッファサイズは送信データ数\*sizeof(送信データタイプ) | **引数** |
| int sendcount  　送信データ数 |
| MPI\_Datatype sendtype  　送信データタイプ |
| void \*recvbuf  受信バッファの先頭アドレス(rootプロセスのみで使用)  受信バッファサイズは、受信データ数\*sizeof(受信データタイプ)\*全プロセス数 |
| int recvcount  　受信データ数(rootプロセスのみで使用) |
| MPI\_Datatype recvtype  　受信データタイプ(rootプロセスのみで使用) |
| int root  　受信プロセスのランク |
| MPI\_Comm comm  　コミュニケ―タ |
| MPI\_SUCCESS  成功 | **戻り値** |
| MPI\_SUCCESS以外  失敗 |
| 全てのプロセスからデータを受信します。この関数で、送信処理と受信処理の両方を行います。  指定した受信プロセスで受信処理を行い、全てのプロセスで送信処理を行います。 | **機能** |

**MPI\_Bcast()**

|  |  |
| --- | --- |
| int MPI\_Bcast(void \*buffer, int count, MPI\_Datatype datatype,  int root, MPI\_Comm comm) |  |
| void \*buffer  　送信バッファの先頭アドレス(rootプロセス)  　受信バッファの先頭アドレス(root以外のプロセス) | **引数** |
| int count  　データ数 |
| MPI\_Datatype datatype  　データタイプ |
| int root  　送信プロセスのランク |
| MPI\_Comm comm  　コミュニケ―タ |
| MPI\_SUCCESS  成功 | **戻り値** |
| MPI\_SUCCESS以外  失敗 |
| rootプロセスからその他のプロセスへデータをブロードキャストします。この関数で、送信処理と受信処理の両方を行います。  指定した送信プロセスで送信処理を行い、全てのプロセスで受信処理を行います。  MPI\_Scatter()はプロセス毎に異なるデータを送信しますが、MPI\_Bcast()は同じデータを送信します。 | **機能** |

**MPI\_Reduce()**

|  |  |
| --- | --- |
| int MPI\_Reduce(void \*sendbuf, void \*recvbuf, int count,  MPI\_Datatype datatype, MPI\_Op op,  int root, MPI\_Comm comm) |  |
| void \*sendbuf  　送信バッファの先頭アドレス | **引数** |
| void \*recvbuf  　受信バッファの先頭アドレス |
| int count  　データ数 |
| MPI\_Datatype datatype  　データタイプ |
| MPI\_Op op  　演算の種類  MPI\_MAX 最大値  MPI\_MIN 最小値  MPI\_SUM 和  MPI\_PROD 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  MPI\_MAXLOC 最大値と位置  MPI\_MINLOC 最大値と位置 |
| int root  　受信プロセスのランク |
| MPI\_Comm comm  　コミュニケータ |
| MPI\_SUCCESS  成功 | **戻り値** |
| MPI\_SUCCESS以外  失敗 |
| 全プロセスのデータに演算opを行い、結果をrootプロセスのバッファに格納する。  全プロセスのsendbuf[i]に演算opを行った結果をrecvbuf[i]へ格納する。  例) データ数2、プロセス数3、op:MPI\_SUMの場合  　recvbuf[0] = (プロセス0の)sendbuf[0]+(プロセス1の)sendbuf[0]+(プロセス2の)sendbuf[0];  　recvbuf[1] = (プロセス0の)sendbuf[1]+(プロセス1の)sendbuf[1]+(プロセス2の)sendbuf[1]; | **機能** |

### 3.4.5 データ型

**MPI\_Comm**

MPI\_Commはプロセスのグループ(コミュニケータ)を表します。

mpirunで実行された全プロセスが所属するグループはMPI\_COMM\_WORLDとなります。

MPI\_Comm\_split()などで、新たなコミュニケ―タを作成することができます。

int MPI\_Comm\_split(MPI\_Comm comm, int color, int key,

MPI\_Comm \*newcomm)

**MPI\_Datatype**

MPI\_Datatypeは通信するデータの型を表します。

よく使うMPI\_Datatypeを記します。

**表2 よく使うMPI\_Datatype**

|  |  |
| --- | --- |
| **MPI\_Datatype** | **C言語の型** |
| MPI\_CHAR | signed char |
| MPI\_SHORT | signed short int |
| MPI\_INT | signed int |
| MPI\_LONG | signed long int |
| MPI\_DOUBLE | double |
| MPI\_Byte | char |

**MPI\_Status**

MPI\_Statusは受信時の情報を保持します。MPI\_StatusはMPI\_Recv()などの受信関数で取得します。

また、下記のようにして受信情報を取得できます

MPI\_Status s;

int s.MPI\_SOURCE /\* 送信元ランク \*/

int s.MPI\_TAG /\* タグ \*/

int s.MPI\_ERROR /\* エラーコード \*/

### 3.4.6 エラーコード

ほとんどのMPI C言語バインディングは、戻り値に成功(MPI\_SUCCESS)かエラーコードを返します。

よく使うエラーコードを記します。

**表3 よく使うエラーコード**

|  |  |
| --- | --- |
| **エラーコード** | **内容** |
| MPI\_ERR\_BUFFER | 不正なバッファポインタ |
| MPI\_ERR\_COUNT | 不正なデータ数 |
| MPI\_ERR\_TYPE | 不正なデータタイプ |
| MPI\_ERR\_TAG | 不正なタグ |
| MPI\_ERR\_COMM | 不正なコミュニケ―タ |
| MPI\_ERR\_RANK | 不正なランク番号 |

### 3.4.7 その他のバインディング

上記以外のバインディングについては、下記URLにて確認することができます。

Open MPI version 1.6.x のマニュアル

<http://www.open-mpi.org/doc/v1.6/>

## 3.5 章末問題

下記問題を解くにあたって、本書で紹介した以外のバインディングを使用しても構いません。是非、効率のよい回答を見つけて下さい。

ポイントは、データをどのように各プロセスに割り当てるか、どのように集計するか、です。

1. 100000個の要素を持つlong型の配列の各要素に0から99までの整数をランダムにセットします。

　(1) MPIを使用しない逐次処理で、0から99までの値のそれぞれの出現頻度を求めて下さい。

　(2) MPIを使用した並列処理で、0から99までの値のそれぞれの出現頻度を求めて下さい。配列はrank番号0のプロセスで値をセットして下さい。

2. 10000x10000の正方行列を2つ作成します。各要素はlong型とし、0から99までの整数をランダムにセットします。

　(1) MPIを使用しない逐次処理で、2つの行列の内積を計算して下さい。

　(2) MPIを使用した並列処理で、2つの行列の内積を計算して下さい。行列はrank番号0のプロセスで値をセットして下さい。また、結果はrank番号0のプロセスに集めるようにして下さい。

# 付録A.

## A.1 mpicc

MPIプログラムはmpiccでコンパイルします。

mpicc はバックエンド処理に gcc などのコンパイラを呼び出します。

mpicc はユーザが指定したオプション(-oなど)をバックエンドコンパイラに渡します。

下記にオプション-oの例を記します。

-o: 出力ファイル名を指定する

|  |
| --- |
| $ mpicc -o <出力ファイル> <入力ファイル> |

## A.2 mpirun

mpirun は指定された数のプロセスでMPIプログラムを実行します。

mpirunの代わりにmpiexecを使用できます。

mpirun では、Single Program Multiple Data (SPMD) と Multiple Program Multiple Data (MPMD) を実行できます。

SPMDでは、同一の実行ファイル(Single Program)を複数プロセスで実行し、各プロセスで異なるデータ(Multiple Data)を処理します。下記に SPMD で実行する方法を記します。

|  |
| --- |
| $ mpirun [options] <program> [ <args> ] |

<args>には、<program>の引数を指定します。

options の例を記します。

-np 10 : プロセスを10個起動する

-host 192.168.0.1 : IP アドレス 192.168.0.1 のホストで実行する

詳細はmpirun のマニュアルを参照下さい。

MPMDでは、複数の異なる実行ファイル(Multiple Program)を複数プロセス実行し各プロセスで異なるデータ(Multiple Data)を処理します。下記に MPMD で実行する方法を記します。

|  |
| --- |
| $ mpirun [options] <program1> [ <args1> ] :  [options] <program2> [ <args2> ] :  [options] <program3> [ <args3> ] |

[options] は SPMD と同じです。

program1,program2,program3間で通信することが可能です。

下記の場合、MPI\_Comm\_size(MPI\_COMM\_WORLD,&size)の結果は35となります。

|  |
| --- |
| $ mpirun -np 10 <program1> [ <args1> ] :  -np 5 <program2> [ <args2> ] :  -np 20 <program3> [ <args3> ] |

# 参考文献

* Peter S. Pacheco, Parallel Programming with MPI, Morgan Kaufmann Publishers, 1997.
* P.パチェコ 著、秋葉 博 訳、MPI 並列プログラミング、培風館、2001.
* ウイリアム グロップ, ラジーブ タークル, ユーイング ラスク 著、畑崎 隆雄 訳、実践 MPI-2、ピアソン・エデュケーション、2002.

# 変更履歴

|  |  |  |
| --- | --- | --- |
| **版数** | **変更内容** | **年月日** |
| 1.0 | 新規作成 | 2014/MAR/3 |