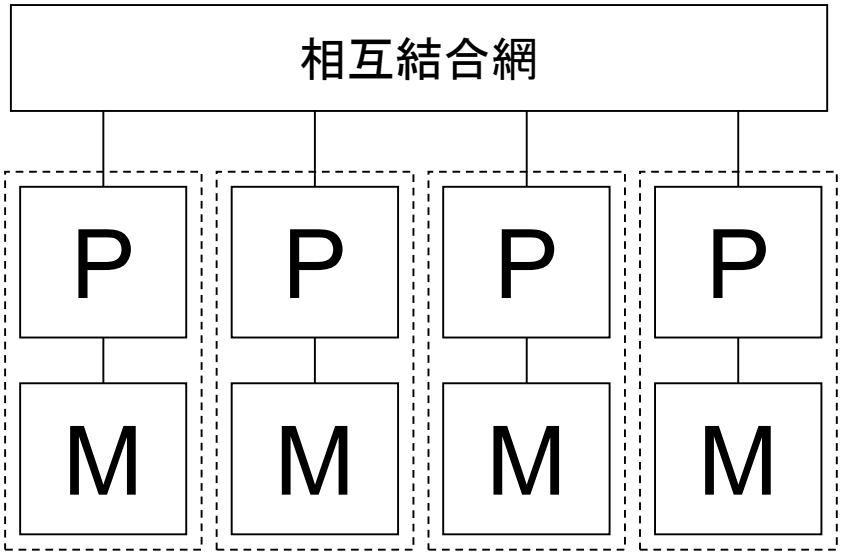
筑波大学計算科学研究センター
CCS HPCセミナー
「MPI」

建部修見
tatebe@cs.tsukuba.ac.jp
筑波大学計算科学研究センター

分散メモリ型並列計算機（PCクラスタ）

- 計算ノードはプロセッサとメモリで構成され，相互結合網で接続
- ノード内のメモリは直接アクセス
- 他ノードとはネットワーク通信により情報交換
- いわゆるPCクラスタ

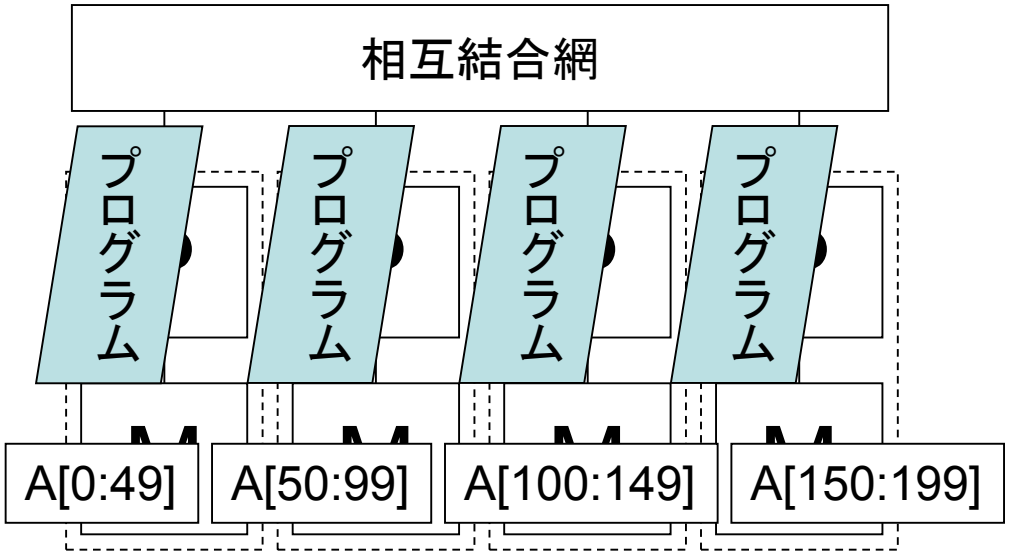


MPI – The Message Passing Interface

- メッセージ通信インターフェースの標準
- 1992年より標準化活動開始
- 1994年，MPI-1.0リリース
  - ポータブルな並列ライブラリ，アプリケーション
  - 8つの通信モード，コレクティブ操作，通信ドメイン，プロセストポロジ
  - 100以上の関数が定義
  - 仕様書　http://www.mpi-forum.org/
    - MPI-2.2が2009年9月にリリース。647ページ
      - 片側通信、プロセス生成、並列I/O
    - MPI-3.1が2015年6月にリリース。868ページ
      - ノンブロッキング集団通信

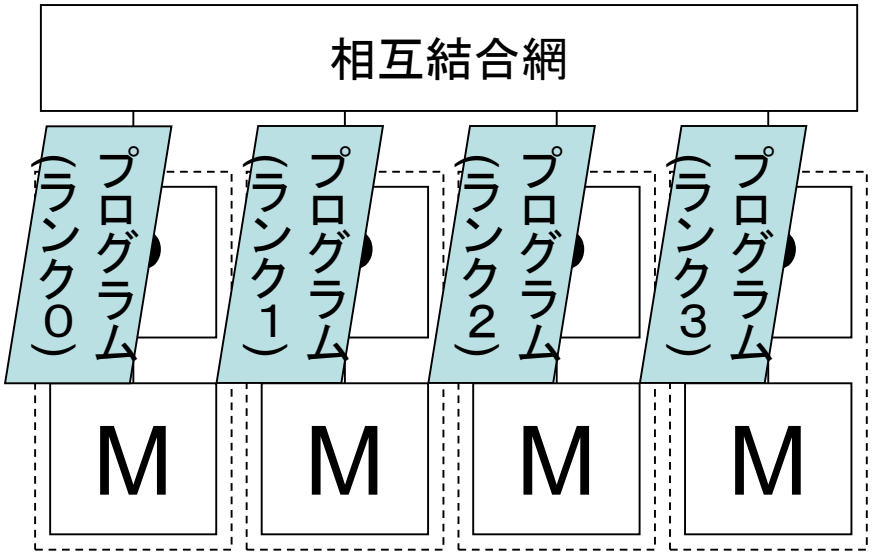
SPMD – Single Program, Multiple Data

- 異なるプロセッサで同一プログラムを独立に実行（cf. SIMD）
- 同一プログラムで異なるデータを処理
- メッセージ通信でプログラム間の相互作用を行う



MPI実行モデル

- (同一の)プロセスを複数のプロセッサで起動
  - プロセス間は(通信がなければ)同期しない
- 各プロセスは固有のプロセス番号をもつ
- MPIによりプロセス間の通信を行う



初期化・終了処理

int **MPI_Init**(int **argc*, char ***argv*)

MPI_INIT(IERROR)

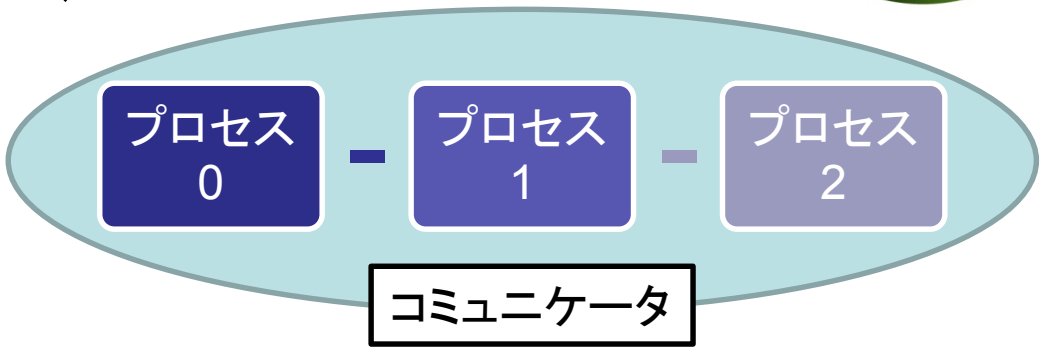
- MPI実行環境を初期化する
- OpenMP等マルチスレッドの場合は **MPI_Init_thread**

int **MPI_Finalize**(void)

MPI_FINALIZE(IERROR)

- MPI実行環境を終了する

コミュニケータ(1)



- 通信ドメイン
  - プロセスの集合
  - プロセス数，プロセス番号(ランク)
  - プロセストポロジ
    - 一次元リング，二次元メッシュ，トーラス，グラフ
- MPI_COMM_WORLD
  - 全プロセスを含む初期コミュニケータ

コミュニケータに対する操作

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

- コミュニケータcommのプロセスグループの総数をsizeに返す

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

- コミュニケータcommのプロセスグループにおける自プロセスのランク番号をrankに返す

コミュニケータ(2)

- 集団通信の“スコープ”(通信ドメイン)を自由に作成可能
- プロセスの分割
  - 2/3のプロセスで天気予報，1/3のプロセスで次の初期値計算
- イントラコミュニケータとインターコミュニケータ

並列処理の例（1）:C言語によるホスト名表示

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

int
main(int argc, char *argv[])
{
        int rank, len;
        char name[MPI_MAX_PROCESSOR_NAME];

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Get_processor_name(name, &len);
        printf("%03d %s¥n", rank, name);
        MPI_Finalize();
        return (0);
}
```

並列処理の例（1）：Fortranによるホスト名表示

```
program hostname
include ‘mpif.h’

integer rank, len, ierr
character(MPI_MAX_PROCESSOR_NAME) hostname

call MPI_INIT(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
call MPI_Get_processor_name(hostname, len, ierr)
write (*,*) rank, hostname
call MPI_Finalize(ierr)
end
```

解説

- mpi.hをインクルード

- 各プロセスはmainからプログラムが実行
- SPMD (single program, multiple data)
  - 単一のプログラムを各ノードで実行
  - 各プログラムは違うデータ(つまり、実行されているプロセスのデータ)をアクセスする

- 初期化
  - **MPI_Init**

解説（続き）

- プロセスランク番号の取得
  - **MPI_Comm_rank**(MPI_COMM_WORLD, &rank);
  - コミュニケータMPI_COMM_WORLDに対し，自ランクを取得
  - コミュニケータはopaqueオブジェクト，内容は関数でアクセス

- ノード名を取得
  - **MPI_Get_processor_name**(name, &len);

- 最後にexitの前で、全プロセッサで！

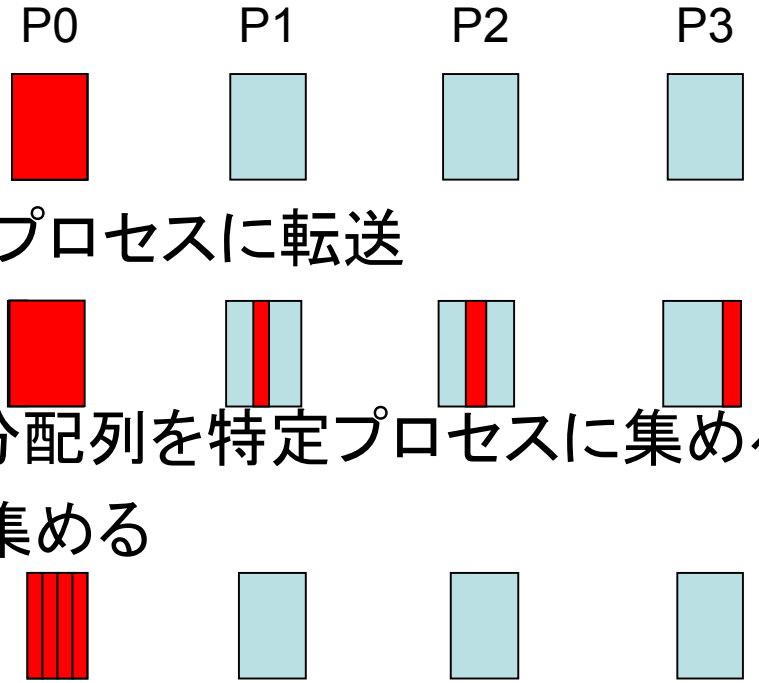
  MPI_Finalize();

集団通信

- コミュニケータに含まれる全プロセス間でのメッセージ通信
- バリア同期（データ転送なし）
- 大域データ通信
  - 放送（broadcast），ギャザ（gather），スキャタ（scatter），全プロセスへのギャザ（allgather），転置（alltoall）
- 縮約通信（リダクション）
  - 縮約（総和，最大値など），スキャン（プレフィックス計算）

x_0	$x_0 + x_1$	$x_0 + x_1 + x_2$	$\cdots$	$x_0 + x_1 + x_2 + \cdots + x_{n-1}$
P0	P1	P2		Pn-1

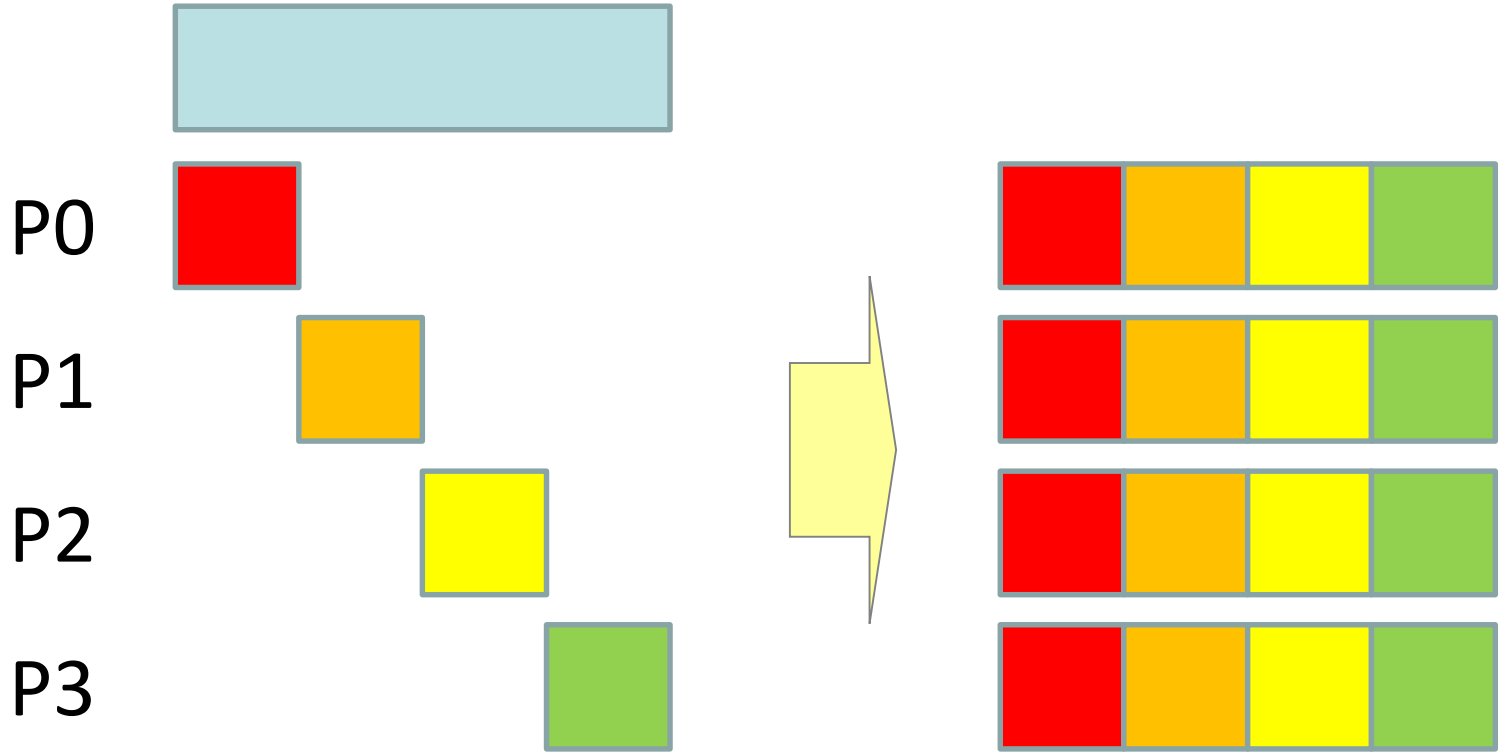
大域データ通信



- 放送
  - ルートプロセスのA[*]を全プロセスに転送
- ギャザ
  - プロセス間で分散した部分配列を特定プロセスに集める
  - allgatherは全プロセスに集める
- スキャタ
  - ルートプロセスのA[*]をプロセス間で分散させる
- Alltoall
  - 全プロセスから全プロセスにスキャタ・ギャザする
  - 二次元配列A[分散][*]→A^T[分散][*]

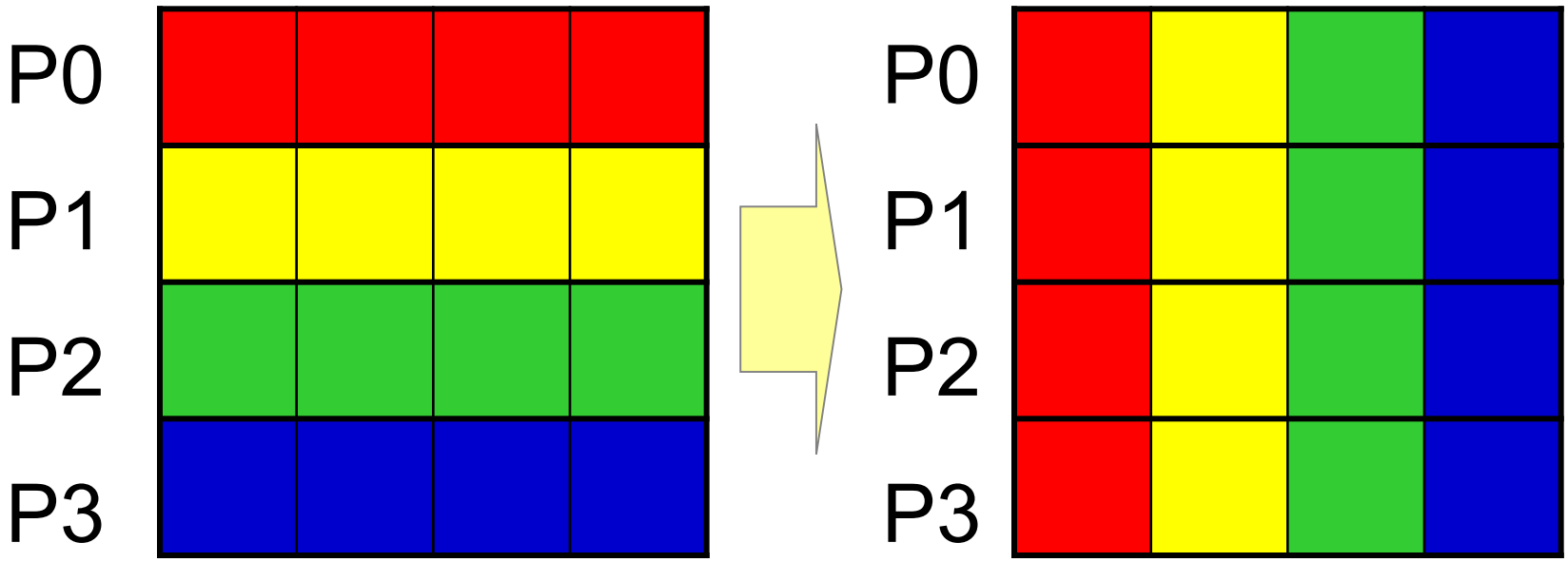
allgather

- 各プロセスの部分配列を集めて全プロセスで全体配列とする



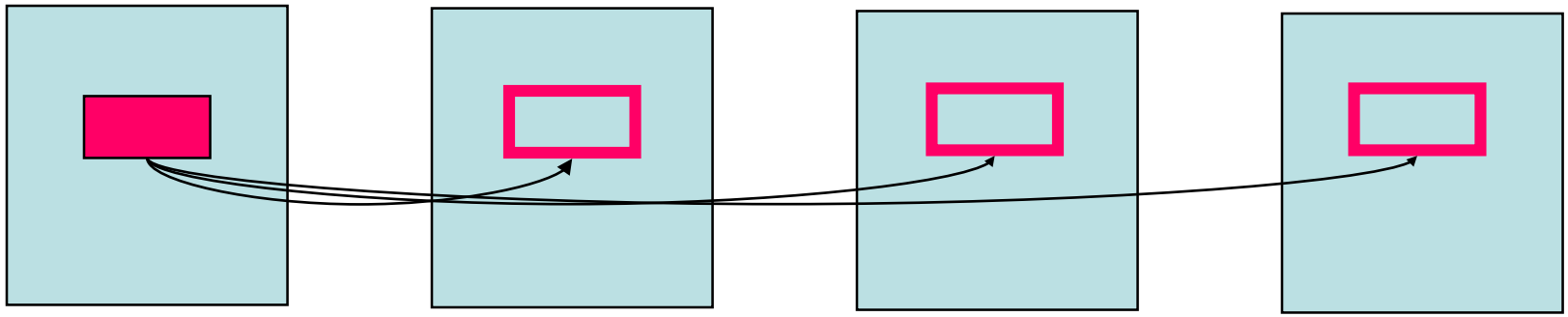
alltoall

- （行方向に）分散した配列を転置する



集団通信: ブロードキャスト

```
int MPI_Bcast(
  void       *data_buffer,   // ブロードキャスト用送受信バッファのアドレス
  int        count,          // ブロードキャストデータの個数
  MPI_Datatype data_type,    // ブロードキャストデータの型(*1)
  int        source,         // ブロードキャスト元プロセスのランク
  MPI_Comm   communicator    // 送受信を行うグループ
);
```

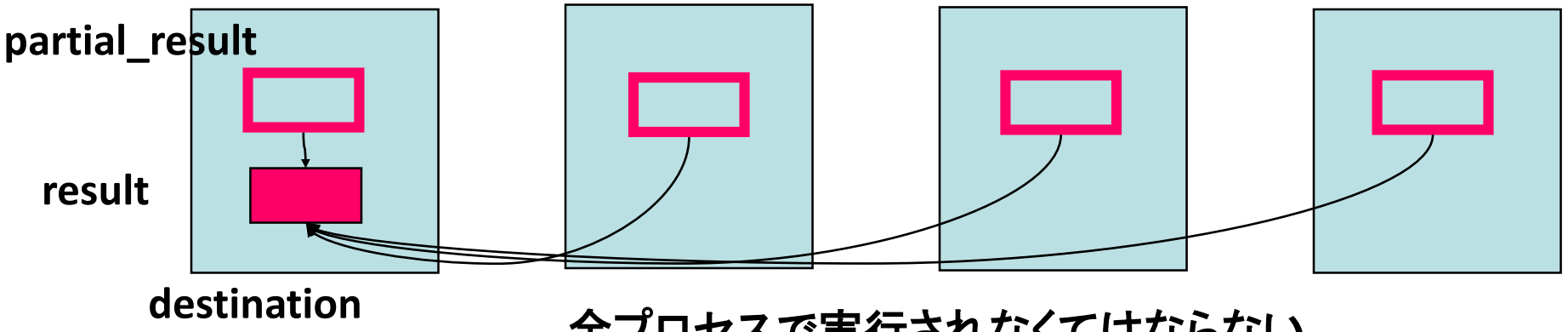


source

全プロセスで実行されなくてはならない

集団通信: リダクション

```
int MPI_Reduce(
  void      *partial_result, // 各ノードの処理結果が格納されているアドレス
  void      *result,         // 集計結果を格納するアドレス
  int       count,           // データの個数
  MPI_Datatype data_type,    // データの型(*1)
  MPI_Op    operator,        // リデュースオペレーションの指定(*2)
  int       destination,     // 集計結果を得るプロセス
  MPI_Comm  communicator     // 送受信を行うグループ
);
```



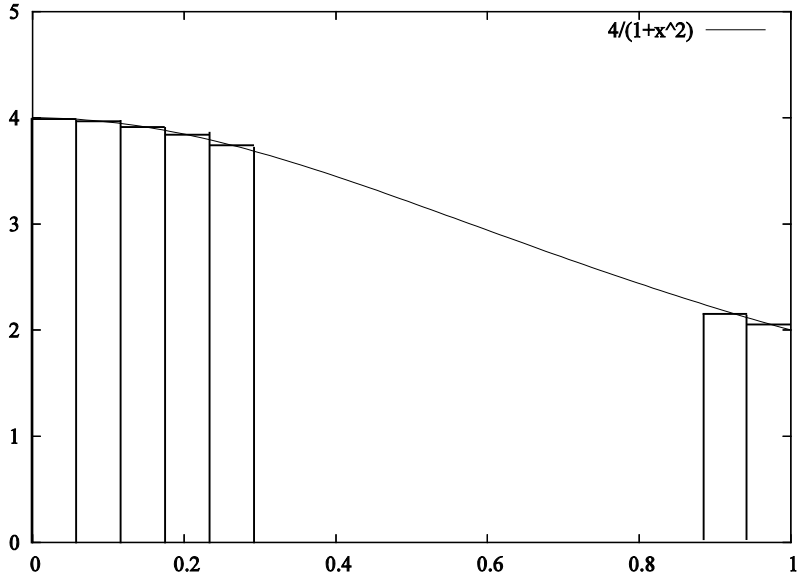
全プロセスで実行されなくてはならない

Resultを全プロセスで受け取る場合は、MPI_Allreduce

並列処理の例（2）：Cpi

- 積分して、円周率を求めるプログラム
- MPICHのテストプログラム
  - 変数nの値をBcast
  - 最後にreduction
  - 計算は、プロセスごとに飛び飛びにやっている

$$\pi = \int_0^1 \frac{4}{1+t^2}\,dt$$



…

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

h = 1.0 / n;
sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs){
        x = h * (i - 0.5);
        sum += f(x);
}
mypi = h * sum;

MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE,
           MPI_SUM, 0, MPI_COMM_WORLD);
```

for (i = 1; i <= n; i++)

```
/* cpi mpi version */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <mpi.h>

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

int
main(int argc, char *argv[])
{
        int n = 0, myid, numprocs, i;
        double PI25DT = 3.14159265358979323846264643;
        double mypi, pi, h, sum, x;
        double startwtime = 0.0, endwtime;
        int namelen;
        char processor_name[MPI_MAX_PROCESSOR_NAME];
```

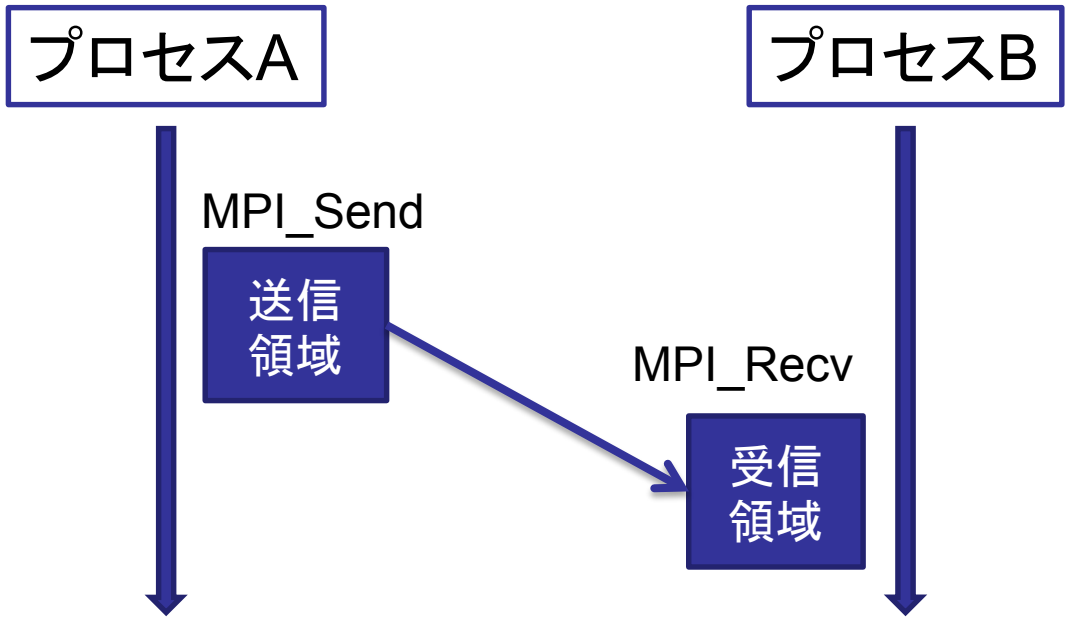
```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
MPI_Get_processor_name(processor_name, &namelen);
fprintf(stderr, "Process %d on %s¥n", myid, processor_name);

if (argc > 1)
        n = atoi(argv[1]);
startwtime = MPI_Wtime();
/* broadcast 'n' */
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (n <= 0) {
        fprintf(stderr, "usage: %s #partition¥n", *argv);
        MPI_Abort(MPI_COMM_WORLD, 1);
}
```

```
    /* calculate each part of pi */
    h = 1.0 / n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs){
        x = h * (i - 0.5);
        sum += f(x);
    }
    mypi = h * sum;
    /* sum up each part of pi */
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0) {
        printf("pi is approximately %.16f, Error is %.16f¥n",
            pi, fabs(pi - PI25DT));
        endwtime = MPI_Wtime();
        printf("wall clock time = %f¥n",
            endwtime - startwtime);
    }
    MPI_Finalize();
    return (0);
}
```

1対1通信（1）

- Point-to-Point通信とも呼ばれる
- プロセスのペア間でのデータ転送
  - プロセスAはプロセスBにデータを送信（send）
  - プロセスBは（プロセスAから）データを受信（recv）



1対1通信(2)

- 型の付いたデータの配列を転送
  - 基本データ型
    - MPI_INT, MPI_DOUBLE, MPI_BYTE, . . .
  - 構造体, ベクタ, ユーザ定義データ型
- コミュニケータ, メッセージタグ, 送受信プロセスランクでsendとrecvの対応を決定
  - 送信元を指定しない場合はMPI_ANY_SOURCEを指定
  - 同じタグを持っているSendとRecvがマッチ
  - どのようなタグでもRecvしたい場合はMPI_ANY_TAGを指定
- Statusで, 実際に受信したメッセージサイズ, タグ, 送信元などが分かる

ブロック型1対1通信

- Send/Receive

```
int MPI_Send(
  void          *send_data_buffer, // 送信データが格納されているメモリのアドレス
  int           count,             // 送信データの個数
  MPI_Datatype  data_type,         // 送信データの型(*1)
  int           destination,       // 送信先プロセスのランク
  int           tag,               // 送信データの識別を行うタグ
  MPI_Comm      communicator       // 送受信を行うグループ.
);
```

```
int MPI_Recv(
  void          *recv_data_buffer, // 受信データが格納されるメモリのアドレス
  int           count,             // 受信データの個数
  MPI_Datatype  data_type,         // 受信データの型(*1)
  int           source,            // 送信元プロセスのランク
  int           tag,               // 受信データの識別を行うためのタグ.
  MPI_Comm      communicator,      // 送受信を行うグループ.
  MPI_Status    *status            // 受信に関する情報を格納する変数のアドレス
);
```

ブロック型1対1通信のコード例

```
int n, rank, tag = 1;
MPI_Status st;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
        n = 1;
        MPI_Send(&n, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
} else if (rank == 1)
        MPI_Recv(&n, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &st);
```

1対1通信（3）

- ブロック型通信のセマンティクス
  - 送信バッファが再利用可能となったら送信終了
  - 受信バッファが利用可能となったら受信終了
- MPI_Send(A, . . .)が戻ってきたらAを変更しても良い
  - 同一プロセスの通信用のバッファにコピーされただけかも
  - メッセージの送信は保証されない

非ブロック型1対1通信

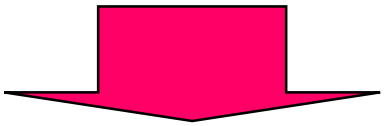
- 非ブロック型通信
  - post-send, complete-send
  - post-receive, complete-receive
- Post-{send,recv}で送信受信操作を開始
- Complete-{send,recv}で完了待ち
- デッドロックを防ぐ
- 計算と通信のオーバラップを可能に
  - マルチスレッドでも可能だが，しばしばより効率的

非ブロック型通信

- Send/recvを実行して、後で終了をチェックする通信方法
  - 通信処理が裏で行える場合は計算と通信処理のオーバラップが可能

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

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

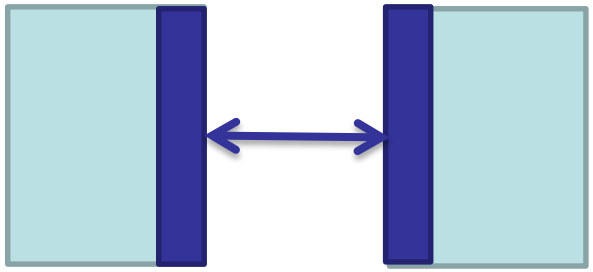


```
int MPI_Wait ( MPI_Request *request, MPI_Status *status)
```

1対1通信の通信モード

- ブロック型，非ブロック型通信のそれぞれに以下の通信モードがある
  - 標準モード
    - MPI処理系が送信メッセージをバッファリングするかどうか決定する。利用者はバッファリングされることを仮定してはいけない
  - バッファモード
    - 送信メッセージはバッファリングされる
    - 送信はローカルに終了
  - 同期モード
    - 送信は対応する受信が発行されたら完了（ランデブー通信）
    - 送信はノンローカルに終了
  - Readyモード
    - 対応する受信が発行されているときだけ送信が始まる
    - 送受信のハンドシェークを省ける

メッセージの交換



- ブロック型

  MPI_Send(送信先、データ)
  MPI_Recv(受信元、データ)

- MPI_Sendがバッファリングされる場合のみ実行可能
  - されない場合はデッドロック
- MPI_Sendrecvを利用する

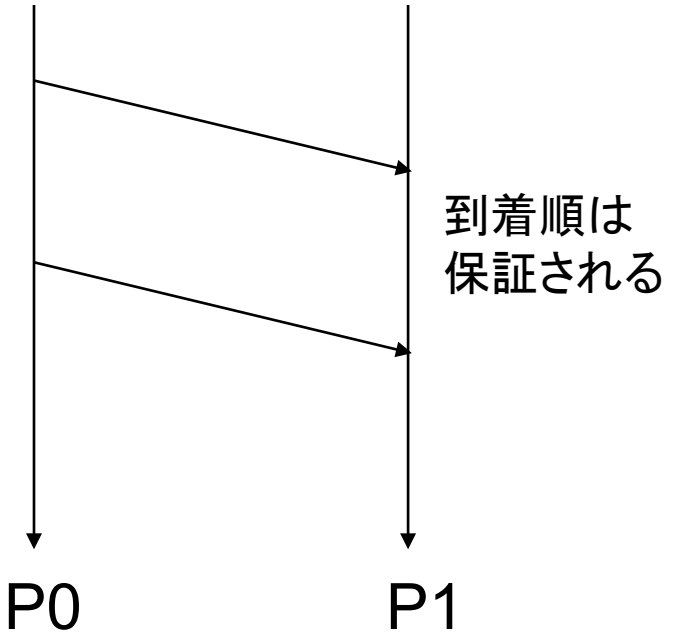
- 非ブロック型

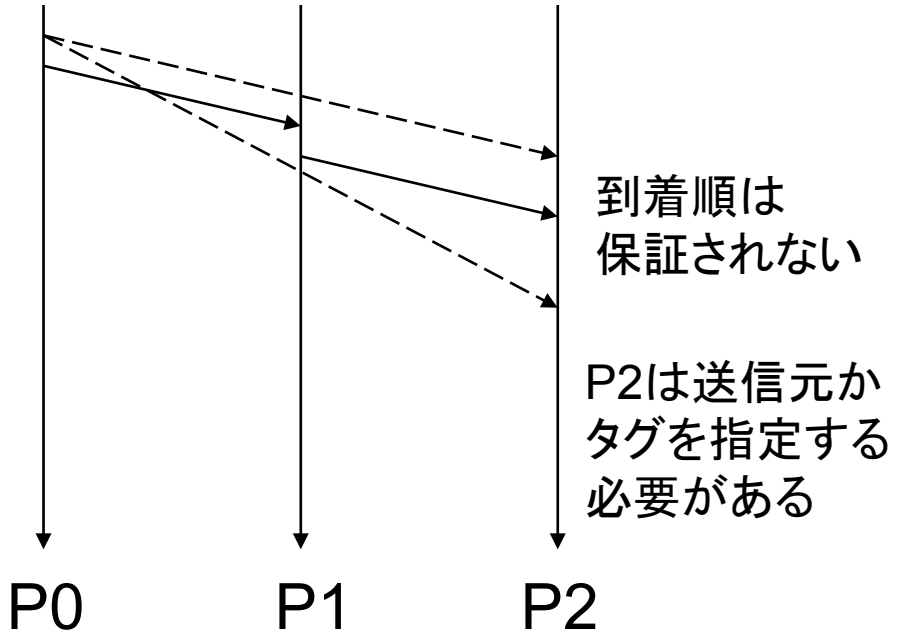
  MPI_Isend(送信先、データ、リクエスト)
  MPI_Recv(受信元、データ)
  MPI_Waitall(リクエスト)

- 必ず実行可能
- ポータブル

1対1通信の注意点(1)

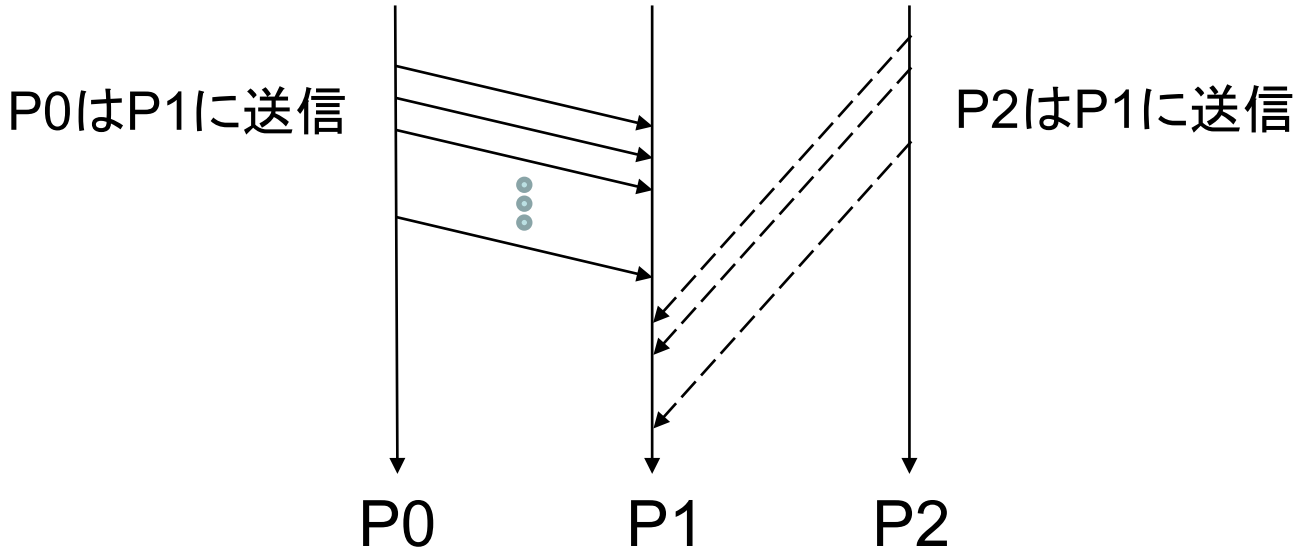
- メッセージ到着順
  - (2者間では)メッセージは追い越されない
  - 3者間以上では追い越される可能性がある





1対1通信の注意点(2)

- 公平性
  - 通信処理において公平性は保証されない



P1はP0からのメッセージばかり受信し、P2からのメッセージがstarvationを引き起こす可能性有り

並列処理の例（3）：laplace

$$\frac{\partial^2 f}{\partial x^2}+\frac{\partial^2 f}{\partial y^2}=0$$

離散化

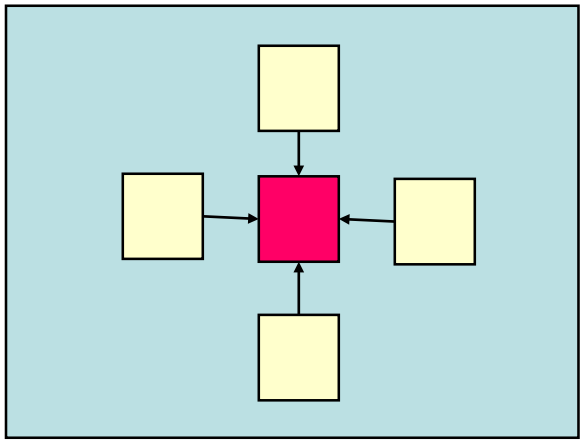
*$f(-1,0)$ means $f(x-\Delta x, y)$

$$f(0,-1)+f(-1,0)+f(1,0)+f(0,1)-4f(0,0)=0$$

- Laplace方程式の陽的解法

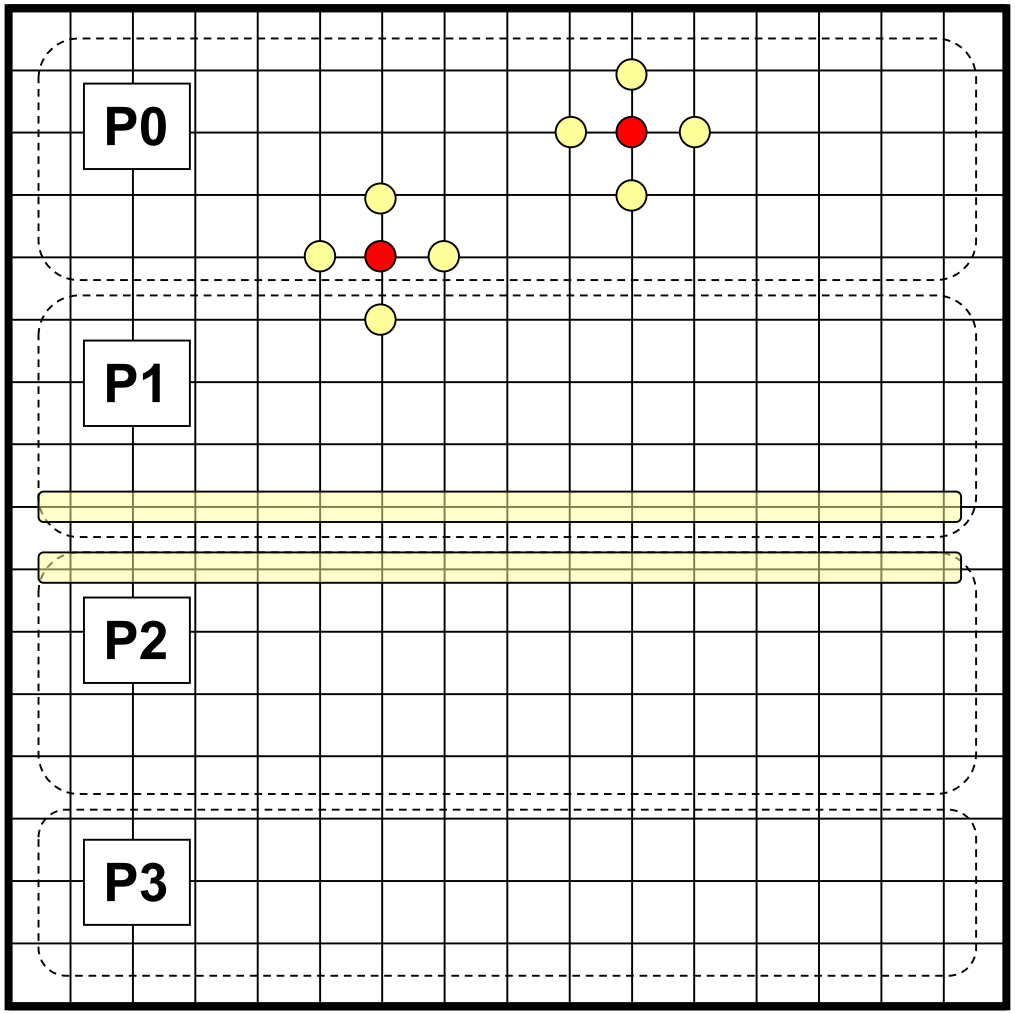
$$f(0,0)_{new}=\frac{1}{4}\left(f_{old}(0,-1)+f_{old}(-1,0)+f_{old}(1,0)+f_{old}(0,1)\right)$$

  - 上下左右の4点の平均で、更新していく
  - Oldとnewを用意して直前の値をコピー
  - 典型的な領域分割
  - 最後に残差をとる



行列分割と隣接通信

- 二次元領域をブロック分割
- 境界の要素は隣のプロセスが更新
- 境界データを隣接プロセスに転送



プロセストポロジ

```
int MPI_Cart_create(MPI_Comm comm_old, int
    ndims, int *dims, int *periods, int reorder,
    MPI_Comm *comm_cart);
```

- ndims次元のハイパーキューブのトポロジをもつコミュニケータcomm_cartを作成
- dimsはそれぞれの次元のプロセス数
- periodsはそれぞれの次元が周期的かどうか
- reorderは新旧のコミュニケータでrankの順番を変更するかどうか

シフト通信の相手先

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

- directionはシフトする次元
  - ndims次元であれば0～ndims-1
- dispだけシフトしたとき，受け取り先がrank_source，送信先がrank_destに返る
- 周期的ではない場合，境界を超えるとMPI_PROC_NULLが返される

```
/* calculate process ranks for 'down' and 'up' */
MPI_Cart_shift(comm, 0, 1, &down, &up);

/* recv from down */
MPI_Irecv(&uu[x_start-1][1], YSIZE, MPI_DOUBLE, down, TAG_1,
          comm, &req1);
/* recv from up */
MPI_Irecv(&uu[x_end][1], YSIZE, MPI_DOUBLE, up, TAG_2,
          comm, &req2);

/* send to down */
MPI_Send(&u[x_start][1], YSIZE, MPI_DOUBLE, down, TAG_2, comm);
/* send to up */
MPI_Send(&u[x_end-1][1], YSIZE, MPI_DOUBLE, up, TAG_1, comm);

MPI_Wait(&req1, &status1);
MPI_Wait(&req2, &status2);
```

端（0とnumprocs-1)のプロセッサについてはMPI_PROC_NULLが指定され特別な処理は必要ない

```
/*
 * Laplace equation with explicit method
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

/* square region */
#define XSIZE 256
#define YSIZE 256
#define PI 3.1415927
#define NITER 10000
double u[XSIZE + 2][YSIZE + 2], uu[XSIZE + 2][YSIZE + 2];
double time1, time2;
void lap_solve(MPI_Comm);
int myid, numprocs;
int namelen;
char processor_name[MPI_MAX_PROCESSOR_NAME];
int xsize;
```

二次元対象領域
uuは更新用配列

```
void
initialize()
{
        int x, y;

        /* 初期値を設定 */
        for (x = 1; x < XSIZE + 1; x++)
                for (y = 1; y < YSIZE + 1; y++)
                        u[x][y] = sin((x - 1.0) / XSIZE * PI) +
                                cos((y - 1.0) / YSIZE * PI);
        /* 境界をゼロクリア */
        for (x = 0; x < XSIZE + 2; x++) {
                u [x][0] = u [x][YSIZE + 1] = 0.0;
                uu[x][0] = uu[x][YSIZE + 1] = 0.0;
        }
        for (y = 0; y < YSIZE + 2; y++) {
                u [0][y] = u [XSIZE + 1][y] = 0.0;
                uu[0][y] = uu[XSIZE + 1][y] = 0.0;
        }
}
```

```
#define TAG_1 100
#define TAG_2 101

#ifndef FALSE
#define FALSE 0
#endif

void lap_solve(MPI_Comm comm)
{
        int x, y, k;
        double sum;
        double t_sum;
        int x_start, x_end;
        MPI_Request req1, req2;
        MPI_Status status1, status2;
        MPI_Comm comm1d;
        int down, up;
        int periods[1] = { FALSE };
```

```
/*
 * Create one dimensional cartesian topology with
 * nonperiodical boundary
 */
MPI_Cart_create(comm, 1, &numprocs, periods, FALSE, &comm1d);
/* calculate process ranks for 'down' and 'up' */
MPI_Cart_shift(comm1d, 0, 1, &down, &up);

x_start = 1 + xsize * myid;
x_end = 1 + xsize * (myid + 1);
```

- Comm1dを1次元トポロジで作成
  - 境界は周期的ではない
- 上下のプロセス番号をup, downに取得
  - 境界ではMPI_PROC_NULLとなる

```
for (k = 0; k < NITER; k++){
        /* old <- new */
        for (x = x_start; x < x_end; x++)
                for (y = 1; y < YSIZE + 1; y++)
                        uu[x][y] = u[x][y];

        /* recv from down */
        MPI_Irecv(&uu[x_start - 1][1], YSIZE, MPI_DOUBLE,
                down, TAG_1, comm1d, &req1);
        /* recv from up */
        MPI_Irecv(&uu[x_end][1], YSIZE, MPI_DOUBLE,
                up, TAG_2, comm1d, &req2);
        /* send to down */
        MPI_Send(&u[x_start][1], YSIZE, MPI_DOUBLE,
                down, TAG_2, comm1d);
        /* send to up */
        MPI_Send(&u[x_end - 1][1], YSIZE, MPI_DOUBLE,
                up, TAG_1, comm1d);

        MPI_Wait(&req1, &status1);
        MPI_Wait(&req2, &status2);
```

```
        /* update */
        for (x = x_start; x < x_end; x++)
            for (y = 1; y < YSIZE + 1; y++)
                u[x][y] = .25 * (uu[x - 1][y] + uu[x + 1][y] +
                                 uu[x][y - 1] + uu[x][y + 1]);
    }
    /* check sum */
    sum = 0.0;
    for (x = x_start; x < x_end; x++)
        for (y = 1; y < YSIZE + 1; y++)
            sum += uu[x][y] - u[x][y];
    MPI_Reduce(&sum, &t_sum, 1, MPI_DOUBLE, MPI_SUM, 0, comm1d);
    if (myid == 0)
        printf("sum = %g¥n", t_sum);
    MPI_Comm_free(&comm1d);
}
```

```
int
main(int argc, char *argv[])
{
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD, &myid);
        MPI_Get_processor_name(processor_name, &namelen);
        fprintf(stderr, "Process %d on %s¥n", myid, processor_name);

        xsize = XSIZE / numprocs;
        if ((XSIZE % numprocs) != 0)
                MPI_Abort(MPI_COMM_WORLD, 1);
        initialize();
        MPI_Barrier(MPI_COMM_WORLD);
        time1 = MPI_Wtime();
        lap_solve(MPI_COMM_WORLD);
        MPI_Barrier(MPI_COMM_WORLD);
        time2 = MPI_Wtime();
        if (myid == 0)
                printf("time = %g¥n", time2 - time1);
        MPI_Finalize();
        return (0);
}
```

改善すべき点

- 配列の一部しか使っていないので、使うところだけにする
  - 配列のindexの計算が面倒になる
  - 大規模計算では本質的な点
- １次元分割だけだが、２次元分割したほうが効率がよい
  - 通信量が減る
  - 多くのプロセッサが使える

MPIとマルチスレッド

int **MPI_Init_thread**(int **argc*, char ****argv*, int *required*, int **provided*)

MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)

required, providedは整数で以下をとる

- MPI_THREAD_SINGLE　　MT不可
- MPI_THREAD_FUNNELED　　MT可、MPIはメインスレッド
- MPI_THREAD_SERIALIZED　　MT可、MPIは同時に呼べない
- MPI_THREAD_MULTIPLE　　MT可

MPI_THREAD_FUNNELED

マスタースレッドだけがMPI呼出し可能

```
#pragma omp parallel
{
#pragma omp master
        MPI call
}
```

MPI_THREAD_SERIALIZED

同時には単一スレッドだけがMPI呼出し可能

```
#pragma omp parallel
{
#pragma omp single
        MPI call
}
```

MPI_THREAD_MULTIPLE

- 複数のスレッドで同時にMPIが呼べる
- ブロッキングMPI呼出しはそのスレッドだけがブロックされる
- 集団通信は注意が必要。同一コミュニケータの集団通信は同時に一つだけ

Open Source MPI

- OpenMPI
  - http://www.open-mpi.org/
- MPICH
  - http://www.mpich.org/

コンパイル・実行の仕方

- コンパイル

  % mpicc … test.c …

  - MPI用のコンパイルコマンドがある
  - 手動で-lmpiをリンクすることもできる

- 実行

  % mpiexec –n #procs a.out …

  - a.outが#procsプロセスで実行される
  - 以前の処理系ではmpirunが利用され，de factoとなっているが，ポータブルではない

    % mpirun –np #procs a.out …

  - 実行されるプロセス群はマシン構成ファイルなどで指定する
  - あらかじめデーモンプロセスを立ち上げる必要があるものも