

## KOBE HPC サマースクール 2020（初級）

## 9. 1対1通信関数, 集団通信関数

2020/9/8

KOBE HPC サマースクール 2021

1

## MPIプログラム（M-2）：1対1通信関数

## 【問題】

1から100までの整数の和を2並列で求めなさい。

## ■ プログラムの方針

- ◆ プロセス0： 1から50までの和を求める。
- ◆ プロセス1： 51から100までの和を求める。
  
- ◆ プロセス1の結果をプロセス0に転送
  
- ◆ プロセス0で、自分の結果と転送された結果を足して出力する。

2020/9/8

KOBE HPC サマースクール 2021

2

## MPIプログラム M-2 (sum.c)

```
#include <stdio.h>
#include <mpi.h>
int main( int argc, char **argv )
{
    int start, end, i, sum_local, sum_recv;
    int nprocs, myrank, tag;
    MPI_Status status;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nprocs );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    tag = 100;
    start = myrank * 50 + 1;
    end = (myrank+1)*50;
    sum_local = 0;
    for( i=start; i<=end; i++ ) {
        sum_local = sum_local + i ;
    } ;

    if( myrank == 1 ) {
        MPI_Send( &sum_local, 1, MPI_INT, 0, tag, MPI_COMM_WORLD );
    } else {
        MPI_Recv( &sum_recv, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &status );
    };
    if( myrank == 0 ) printf("Sum = %d\n", sum_local+sum_recv );

    MPI_Finalize();
    return 0 ;
}
```

## 1対1通信 – 送信関数 MPI\_Send (送り出し側)

```
int MPI_Send(void *buff, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

- ◆ buff: 送信するデータの変数の先頭アドレス
- ◆ count: 送信するデータの個数
- ◆ datatype: 送信するデータの型
  - MPI\_CHAR, MPI\_INT, MPI\_DOUBLE など
- ◆ dest: 送信先のMPIプロセス番号 (destination)
- ◆ tag: メッセージ識別番号, 送るデータを区別するための番号
- ◆ comm: コミュニケータ (例えば, MPI\_COMM\_WORLD)

※ 関数の戻りコードは, エラーコード

## 1対1通信 – 受信関数 MPI\_Recv (受け取り側)

```
int MPI_Recv(void *buff, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm,
             MPI_Status *status)
```

- ◆ buff: 送信するデータの変数名 (先頭アドレス)
  - ◆ count: 送信するデータの個数
  - ◆ datatype: 送信するデータの型
    - MPI\_INT, MPI\_DOUBLE, MPI\_CHAR など
  - ◆ source: 送信元のMPIプロセス番号
  - ◆ tag: メッセージ識別番号. 送るデータを区別するための番号
  - ◆ comm: コミュニケータ (例えば, MPI\_COMM\_WORLD)
  - ◆ status: 状況オブジェクト. MPI\_Sendには, この引数はないので注意.
- ※ 関数の戻りコードは, エラーコード

## 関数の引数に関する注意 (共通)

- buff
  - ◆ 送信するデータは領域は, **メモリ上で連続アドレス**でなければならない.
    - "先頭アドレスからx xバイトを送れ" という関数なので.
  - ◆ 他の通信関数でも同じ.
  - ◆ したがって, メモリ上で離れたところにある複数の変数を, 1回の通信で同時に送りたい場合は, 他の変数に連続してパック (pack) させてから, 送る必要がある.
- datatype: 予約語 (決まっている)
  - ◆ MPI\_INT (整数型), MPI\_DOUBLE (倍精度実数型), MPI\_CHAR (文字型) などが使用できる.
  - ◆ バイト数を計算するために必要
- tag
  - ◆ 同じプロセスに対し, 複数回メッセージを送るとき, メッセージを受取ったプロセスが, どのメッセージかを区別するために使用する.
  - ◆ 受取側の MPI\_Recv では, メッセージに対応したtagで受け取らなければならない.
  - ◆ 複数回のメッセージでも, 送受信の順番などを区別できる場合は, 同じtagでも良い.

## 演習9-1 1から100までの和を2並列で求めるプログラムの実行

- 1 から 100 までの整数の和を2並列で求めるプログラム (sum.c) を2プロセスで実行し、結果を確認せよ。

### 【手順】

- ① /tmp/Summer/M-2/sum.c を適切なディレクトリにコピーする。
- ② sum.c をコンパイルする。
  - ・ `icc sum.c -lmpi`
- ③ /tmp/Summer/M-2/go.sh をコピーして、ジョブを実行。
- ④ バッチジョブ出力結果 (sum.onnnnnn) を確認する。
  - 正しい答え (Sum = 5050) が出力されているか？
    - プロセス0 (rank 0) だけが出力していることに注意。

## 演習9-2 (発展)

- 1 から 100 までの整数の和を求めるプログラムを、4並列で実行できるように修正し、4プロセスで実行せよ。
  - ◆ プロセス0が結果を出力する。
  - ◆ MPI\_Send, MPI\_Recv関数だけを使うこと。
- プログラム改良のヒント
  - ◆ 各プロセスの部分和を計算する範囲を、myrank をうまく使って求める。myrankは、0 から 3の整数である。



- ◆ myrank ≠ 0 以外のプロセスから、プロセス0 (myrank=0) に部分和を送信する。
- ◆ プロセス0 (myrank=0) は、他の3つのプロセスから送られた部分和を受信 (forループ) し、受信するごとに受信したデータを加え、全体の和を計算する。

## 集団通信関数

- 1対1通信関数の煩雑な点
  - ◆ プロセス数が多くなると、1対1通信関数を用いたプログラムは複雑
  - ◆ 煩雑になるとバグが入りやすい。
  
- もっと簡単な方法はないのか？ → **集団通信関数**
  - ◆ MPI\_Bcast
    - あるプロセスから、すべてのプロセスに値を一斉に配る関数
  
  - ◆ MPI\_Reduce
    - すべてのプロセスから、あるプロセス（例えば rank 0）に値を集めて、何らかの演算（+、x、max、minなど）を適用する関数

## MPIプログラム sum\_reduction（集団通信関数を使う）

```

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

int main( int argc, char **argv )
{
    int start, end, i, sum_local, sum, n ;
    int nprocs, myrank ;
    MPI_Status status;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nprocs );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );

    if( myrank == 0 ) n = 100 ;
    MPI_Bcast( &n, 1, MPI_INT, 0, MPI_COMM_WORLD );

    start = myrank * (n/nprocs) + 1;
    end   = (myrank+1)*(n/nprocs) ;
    sum_local = 0;
    for( i=start; i<=end; i++ ) {
        sum_local += i ;
    } ;

    sum = 0 ;
    MPI_Reduce( &sum_local, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD );

    if( myrank == 0 ) printf("Sum = %d\n", sum ) ;

    MPI_Finalize();
    return 0 ;
}
    
```

## MPIプログラム sum\_reduction (集団通信関数を使う)

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

int main( int argc, char **argv )
{
    int start, end, i, sum_local, sum, n ;
    int nprocs, myrank ;
    MPI_Status status;

    MPI_Init( &argc, &argv);
    MPI_Comm_size( MPI_COMM_WORLD, &nprocs );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );

    if( myrank == 0 ) n = 100 ;
    MPI_Bcast( &n, 1, MPI_INT, 0, MPI_COMM_WORLD );

    start = myrank *(n/nprocs) + 1;
    end   = (myrank+1)*(n/nprocs) ;
    sum_local = 0;
    for( i=start; i<=end; i++ ) {
        sum_local += i ;
    } ;

    sum = 0 ;
    MPI_Reduce( &sum_local, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD );

    if( myrank == 0 ) printf("Sum = %d\n", sum) ;

    MPI_Finalize();
    return 0 ;
}
```

- 青：MPIプログラムのおまじない（既出）
- 緑：プロセス番号（ランク）に応じた処理
- 赤：MPI関数によるプロセス間通信

プロセス0が nの値をセットする

nの値を全プロセスに放送

ランクの値から自分の計算範囲を求める

各プロセスが  
部分和を計算

部分和の総和を計算  
(プロセス0に集める)

プロセス0だけが結果を出力

## 集団通信 - broadcast

```
int MPI_Bcast(void *buff, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
```

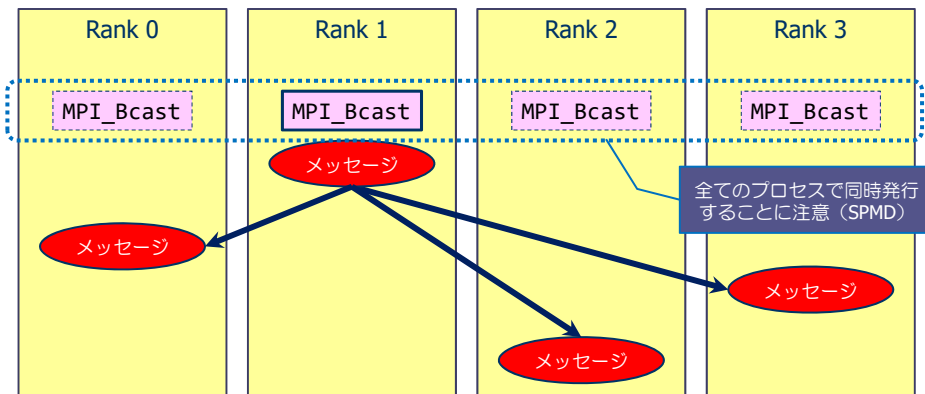
※ rootが持つbuffの値を、commで指定された他のプロセスのbuffに配布する。

- ◆ buff: 送り主（root）が送信するデータの変数名（先頭アドレス）  
他のMPIプロセスは、同じ変数名でデータを受け取る。
- ◆ count: データの個数
- ◆ datatype: 送信するデータの型
  - MPI\_INT, MPI\_DOUBLE, MPI\_CHAR など
- ◆ root: 送り主のMPIプロセス番号
- ◆ comm: コミュニケータ（例えば、MPI\_COMM\_WORLD）

※ 関数の戻りコードは、エラーコードを表す。

## 集団通信 - broadcast

Rank 1のプロセスから、すべてのプロセスにデータ（メッセージ）を送る。



## 集団通信 - reduction

```
int MPI_Reduce(void *sendbuff, void *recvbuff, int count, MPI_Datatype datatype, MPI_Op op,
               int root, MPI_Comm comm)
```

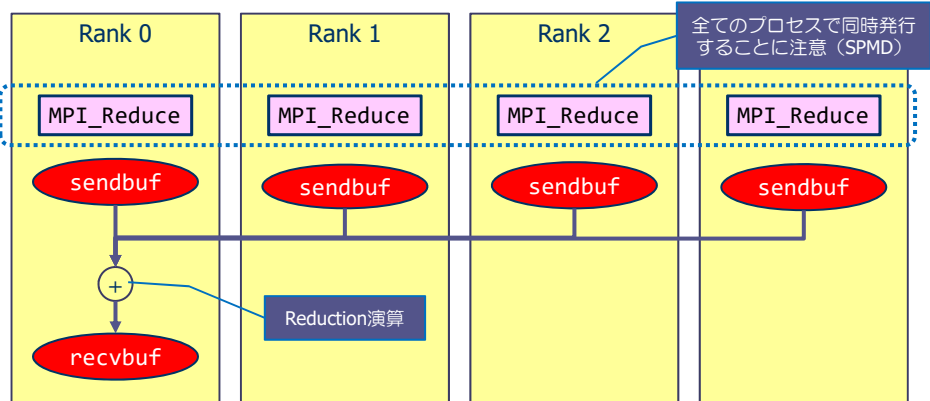
※ commで指定されたすべてのプロセスからデータをrootが集め、演算 (op) を適用する。

- ◆ sendbuff: 送信するデータの変数名 (先頭アドレス)
- ◆ recvbuff: 受信するデータの変数名 (先頭アドレス)
- ◆ count: データの個数
- ◆ datatype: 送信するデータの型
  - MPI\_INT, MPI\_DOUBLE, MPI\_CHAR など
- ◆ op: 集まってきたデータに適用する演算の種類
  - MPI\_SUM (総和), MPI\_PROD (掛け算), MPI\_MAX (最大値) など
- ◆ root: データを集めるMPIプロセス番号
- ◆ comm: コミュニケータ (例えば, MPI\_COMM\_WORLD)

※ 関数の戻りコードは、エラーコードを表す。

## 1つのプロセスへのリダクション：MPI\_Reduce

すべてのプロセスから一つのプロセスにデータを集めて、演算をする。



## リダクション演算とは

- リダクション演算
  - ◆ 加算, 乗算, 最大値のように, 複数のデータを入力として1個の出力データを求める演算
- MPIで使えるリダクション演算
  - ◆ MPI\_SUM (和), MPI\_PROD (積),
  - ◆ MPI\_MAX (最大値), MPI\_MIN (最小値)
  - ※他にも論理和などがある
- ベクトルに対するリダクション演算も可能
  - ◆ ベクトルの各要素に対してリダクション演算を行い, その結果を要素とするベクトルを生成
  - ◆  $m$  個のベクトル  $x_1, x_2, x_3, \dots, x_m$  をそれぞれ長さ  $n$  のベクトルとすると, それらの和  $x = x_1 + x_2 + x_3 + \dots + x_m$  を求める計算
  - ◆ 引数 count には, ベクトルの長さ  $n$  を指定すればよい.



## 演習9-3 集団通信関数を使ったプログラムの実行

- プログラム `sum_reduction.c` を、2 MPIプロセス、4 MPIプロセスで実行し、結果を確認せよ。

### 【手順】

- ① `/tmp/Summer/M-2/sum_reduction.c` を適切なディレクトリにコピーする。
- ② `sum_reduction.c` をコンパイルする。
  - ・ `icc sum_reduction.c -lmpi`
- ③ `go.sh` を修正して、ジョブを実行。
- ④ バッチジョブ出力結果 (`sum.onnnnnn`) を確認する。
  - 出力に正しい答え (`Sum = 5050`) が出力されているか？

## 演習9-4 (発展1)

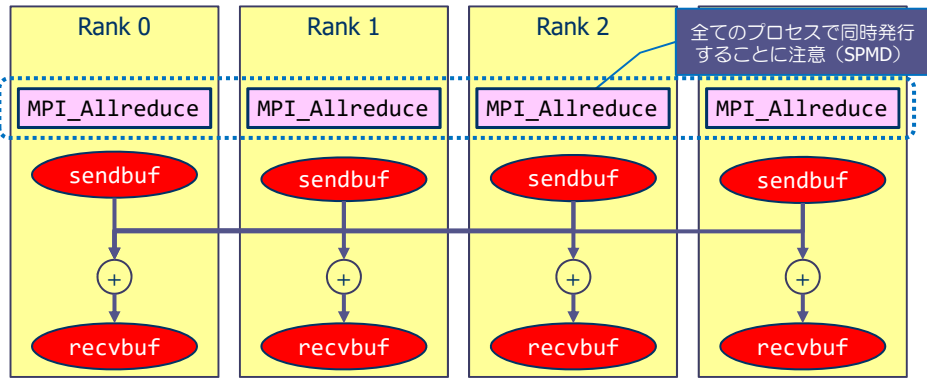
- プロセス毎に部分和を出力した後、すべてのプロセスで総和を計算し、プロセス毎に総和結果を出力をせよ。
  - ◆ それぞれのプロセスが出力する総和が同じであることを確認する)。
  - ◆ 出力のイメージ

```
Rank: n -> Local sum = xxxx
Rank: n -> Total sum = xxxx
```
- プログラム改良のヒント
  - ◆ `MPI_Reduce`, `MPI_Bcast` を順番に使う (集めてから配信)。
  - ◆ 各プロセスで、`printf` で出力させる。

```
printf("Rank: %d -> Local sum = %d", myrank, local_sum);
```
- 同じ処理をする関数 `MPI_Allreduce` がある。

## すべてのプロセスへのリダクション：MPI\_Allreduce

すべてのプロセスで、すべてのプロセスからデータを集めて、演算をする。



🔗 「sendbuf」と「recvbuf」はメモリ上で重複させないこと。そうでなければ、動作は保証されない。

## 集団通信 — MPI\_Allreduce

```
int MPI_Allreduce( void *sendbuff, void *recvbuff, int count, MPI_Datatype datatype, MPI_Op op,
                  MPI_Comm comm )
```

※ MPI\_ReduceとMPI\_Bcastを同時に行える関数。すべてのプロセスで同じ結果（総和など）が得られる。

- ◆ sendbuff: 送信するデータの変数名（先頭アドレス）
- ◆ recvbuff: 受信するデータの変数名（先頭アドレス）
- ◆ count: データの個数
- ◆ datatype: 送信するデータの型
  - MPI\_INT, MPI\_DOUBLE, MPI\_CHAR など
- ◆ op: 集まってきたデータに適用する演算の種類
  - MPI\_SUM（総和）, MPI\_PROD（掛け算）, MPI\_MAX（最大値）など
- ◆ comm: コミュニケータ（例えば, MPI\_COMM\_WORLD）

※ 関数の戻りコードは、エラーコードを表す。

## 演習9-5 (発展2)

- MPI\_Reduce, MPI\_Bcast の組を MPI\_Allreduce で書き換えよ。
  - ◆ 出力のイメージは、書き換え前と同じ。
    - Rank: n -> Local sum = xxxx
    - Rank: n -> Total sum = xxxx
  
- プログラム改良のヒント
  - ◆ MPI\_Allreduceの引数である sendbuff, recvbuff をうまく指定する。

## 演習9-6 MPI\_Allreduceの応用：ベクトルの正規化

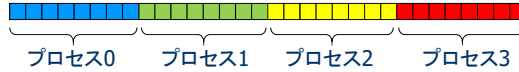
- $n$ 次元ベクトル  $x$  (double  $x[N];$ ) の第  $i$  要素を  $i + 1$  とする。
  - ◆  $x[i] = i + 1 \quad (i = 0, \dots, n - 1)$
  
- $x$  を正規化したベクトル  $x/\|x\|_2$  を求めるプログラム `normalize.c` の逐次実行動作を確認せよ。
  - ◆ 変数は倍精度 (double) とし、倍精度で計算する。  $\|x\|_2$  は  $x$  の各要素の2乗和の平方根である。
  - ◆ 正規化されたベクトルの要素は
 
$$\hat{x}[i] = (i + 1) / \sqrt{n(n + 1)(2n + 1) / 6}$$
 であるので (簡単に計算できる)、プログラムではこの値と比較することによって、正しく計算ができている (Error = 0.0) ことを確認している。

```
$ cp /tmp/Summer/M-2/normalize.c ./
$ icc -O3 normalize.c
$ ./a.out
Error = 0.000000
```

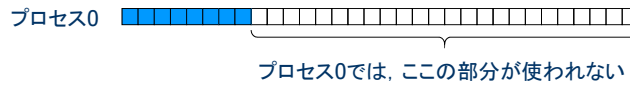
```
$ cp /tmp/Summer/M-2/normalize.f90 ./
$ ifort -O3 normalize.f90
$ ./a.out
Error = 0.0000
```

### 演習9-7 normalize.cを並列化せよ

- 各プロセスが自分の担当分の要素について、2乗和を計算する。
- 各プロセスの担当する要素 (nprocs はMPIプロセス数)
  - $istart = (n/nprocs)*myrank$
  - $iend = (n/nprocs)*(myrank+1)-1$



- ベクトルの格納方法
  - ◆ 各プロセスは長さ n の配列を持ち、そのうち自分の担当部分のみを使う



- 全プロセスの総和を MPI\_Allreduce関数で求め、各プロセスはその総和を使って、自分の担当する要素について正規化を行う。
- n=50000 としてプロセス数を1, 2, 4, 8と変えて計算せよ。結果の確認も工夫すること。

参考：Fortran版

## MPIプログラム (sum.f90)

```

program sum100_by_mpi
use mpi
implicit none
integer :: i, istart, iend, isum_local, isum_tmp
integer :: nprocs, myrank, ierr
integer :: istat(MPI_STATUS_SIZE)
call mpi_init( ierr )
call mpi_comm_size( MPI_COMM_WORLD, nprocs, ierr )
call mpi_comm_rank( MPI_COMM_WORLD, myrank, ierr )
istart = myrank*50 + 1
iend   = (myrank+1)*50
isum_local = 0
do i = istart, iend
    isum_local = isum_local + i
enddo
if( myrank == 1 ) then
    call mpi_send( isum_local, 1, MPI_INTEGER, 0, 100, MPI_COMM_WORLD, ierr )
else
    call mpi_recv( isum_tmp, 1, MPI_INTEGER, 1, 100, MPI_COMM_WORLD, istat, ierr )
end if
if( myrank == 0 ) print *, 'sum =', isum_local+isum_tmp
call mpi_finalize( ierr )
end program sum100_by_mpi
    
```

2020/9/8

KOBE HPC サマースクール 2021

25

## MPIプログラム (sum.f90)

```

program sum100_by_mpi
use mpi
implicit none
integer :: i, istart, iend, isum_local, isum_tmp
integer :: nprocs, myrank, ierr
integer :: istat(MPI_STATUS_SIZE)
call mpi_init( ierr )
call mpi_comm_size( MPI_COMM_WORLD, nprocs, ierr )
call mpi_comm_rank( MPI_COMM_WORLD, myrank, ierr )
istart = myrank*50 + 1
iend   = (myrank+1)*50
isum_local = 0
do i = istart, iend
    isum_local = isum_local + i
enddo
if( myrank == 1 ) then
    call mpi_send( isum_local, 1, MPI_INTEGER, 0, 100, MPI_COMM_WORLD, ierr )
else
    call mpi_recv( isum_tmp, 1, MPI_INTEGER, 1, 100, MPI_COMM_WORLD, istat, ierr )
end if
if( myrank == 0 ) print *, 'sum =', isum_local+isum_tmp
call mpi_finalize( ierr )
end program sum100_by_mpi
    
```

- 青：MPIプログラムのおまじない（既出）
- 緑：プロセス番号（ランク）に応じた処理
- 赤：MPI関数によるプロセス間通信

ランクの値から自分の計算範囲を求める

各プロセスが  
部分和を計算

プロセス1はプロセス0に自分  
の部分和を送信

プロセス0はプロセス1から部  
分和を受信（変数名が違うこ  
とに注意）

プロセス0が、総和を出力

2020/9/8

KOBE HPC サマースクール 2021

26

## 1対1通信 – 送信関数 `mpi_send` (送り出し側)

```
mpi_send( buff, count, datatype, dest, tag, comm, ierr )
```

※ ランク番号`dest`のプロセスに、変数`buff`の値を送信する。

- ◆ `buff`: 送信するデータの変数名 (先頭アドレス)
- ◆ `count`: 送信するデータの数 (整数型)
- ◆ `datatype`: 送信するデータの型
  - `MPI_INTEGER`, `MPI_REAL`, `MPI_DOUBLE_PRECISION`など
- ◆ `dest`: 送信先プロセスのランク番号
- ◆ `tag`: メッセージ識別番号. 送るデータを区別するための番号
- ◆ `comm`: コミュニケータ (例えば, `MPI_COMM_WORLD`)
- ◆ `ierr`: 戻りコード (整数型)

## 1対1通信 – 受信関数 `mpi_recv` (受け取り側)

```
mpi_recv( buff, count, datatype, source, tag, comm, status, ierr )
```

※ ランク番号`source`のプロセスから送られたデータを、変数`buff`に格納する。

- ◆ `buff`: 受信するデータのための変数名 (先頭アドレス)
- ◆ `count`: 受信するデータの数 (整数型)
- ◆ `datatype`: 受信するデータの型
  - `MPI_INTEGER`, `MPI_REAL`, `MPI_DOUBLE_PRECISION`など
- ◆ `source`: 送信してくる相手プロセスのランク番号
- ◆ `tag`: メッセージ識別番号. 送られて来たデータを区別するための番号
- ◆ `comm`: コミュニケータ (例えば, `MPI_COMM_WORLD`)
- ◆ `status`: 受信の状態を格納するサイズ`MPI_STATUS_SIZE`の配列 (整数型)
- ◆ `ierr`: 戻りコード (整数型)

## MPIプログラム sum\_reduction.f90

```

program sum_by_reduction
use mpi
implicit none
integer :: n, i, istart, iend, isum_local, isum
integer :: nprocs, myrank, ierr

call mpi_init( ierr )
call mpi_comm_size( MPI_COMM_WORLD, nprocs, ierr )
call mpi_comm_rank( MPI_COMM_WORLD, myrank, ierr )
if( myrank==0 ) n=10000
call mpi_bcast( n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr )
istart = (n/nprocs)*myrank + 1
iend   = (n/nprocs)*(myrank+1)
isum_local = 0
do i = istart, iend
    isum_local = isum_local + i
enddo
call mpi_reduce( isum_local, isum, 1, MPI_INTEGER, MPI_SUM, 0, MPI_COMM_WORLD, ierr )

if( myrank == 0 ) print *, 'Sum (by reduction function) =', isum

call mpi_finalize( ierr )
end program sum_by_reduction
    
```

- 青：MPIプログラムのおまじない（既出）
- 緑：プロセス番号（ランク）に応じた処理
- 赤：MPI関数によるプロセス間通信

プロセス0がnの値をセットする

nの値を放送

ランクの値から自分の計算範囲を求める

各プロセスが部分和を計算

部分和の総和を計算（プロセス0に集める）

プロセス0だけが結果を出力

## 集団通信 — broadcast

```
mpi_bcast( buff, count, datatype, root, comm, ierr )
```

※ ランク番号rootのプロセスが持つbuffの値を、commで指定された他のすべてのプロセスのbuffに配布する。

- ◆ buff: 送り主（root）が送信するデータの変数名（先頭アドレス）  
他のMPIプロセスは、同じ変数名でデータを受け取る。
- ◆ count: データの個数（整数型）
- ◆ datatype: 送信するデータの型
  - MPI\_INTEGER, MPI\_REAL, MPI\_DOUBLE\_PRECISION, MPI\_REAL8など
- ◆ root: 送り主のMPIプロセス番号
- ◆ comm: コミュニケータ（例えば、MPI\_COMM\_WORLD）
- ◆ ierr: 戻りコード（整数型）

## 集団通信 — reduction

```
mpi_reduce( sendbuff, recvbuff, count, datatype, op, root, comm, ierr )
```

※ commで指定されたすべてのプロセスからデータを、ランク番号 root のプロセスに集め、演算 (op) を適用した結果を recvbuffに設定する。

- ◆ sendbuff: 送信するデータの変数名 (先頭アドレス)
- ◆ recvbuff: 受信するデータの変数名 (先頭アドレス)
- ◆ count: データの個数 (整数型)
- ◆ datatype: 送信するデータの型
  - MPI\_INTEGER, MPI\_REAL, MPI\_DOUBLE\_PRECISION, MPI\_REAL8など
- ◆ op: 集まってきたデータに適用する演算の種類
  - MPI\_SUM (総和), MPI\_PROD (掛け算), MPI\_MAX (最大値) など
- ◆ root: データを集めるMPIプロセス番号
- ◆ comm: コミュニケータ (例えば, MPI\_COMM\_WORLD)
- ◆ ierr: 戻りコード (整数型)

## 集団通信 — mpi\_allreduce()

```
mpi_allreduce( sendbuff, recvbuff, count, datatype, op, comm, ierr )
```

※ mpi\_reduceとmpi\_bcastを同時に行える関数。すべてのプロセスで同じ結果 (総和など) が得られる。

- ◆ sendbuff: 送信するデータの変数名 (先頭アドレス)
- ◆ recvbuff: 受信するデータの変数名 (先頭アドレス)
- ◆ count: データの個数 (整数型)
- ◆ datatype: 送信するデータの型
  - MPI\_INTEGER, MPI\_REAL8, MPI\_CHARACTER など
- ◆ op: 集まってきたデータに適用する演算の種類
  - MPI\_SUM (総和), MPI\_PROD (掛け算), MPI\_MAX (最大値) など
- ◆ comm: コミュニケータ (例えば, MPI\_COMM\_WORLD)
- ◆ ierr: 戻りコード (整数型)