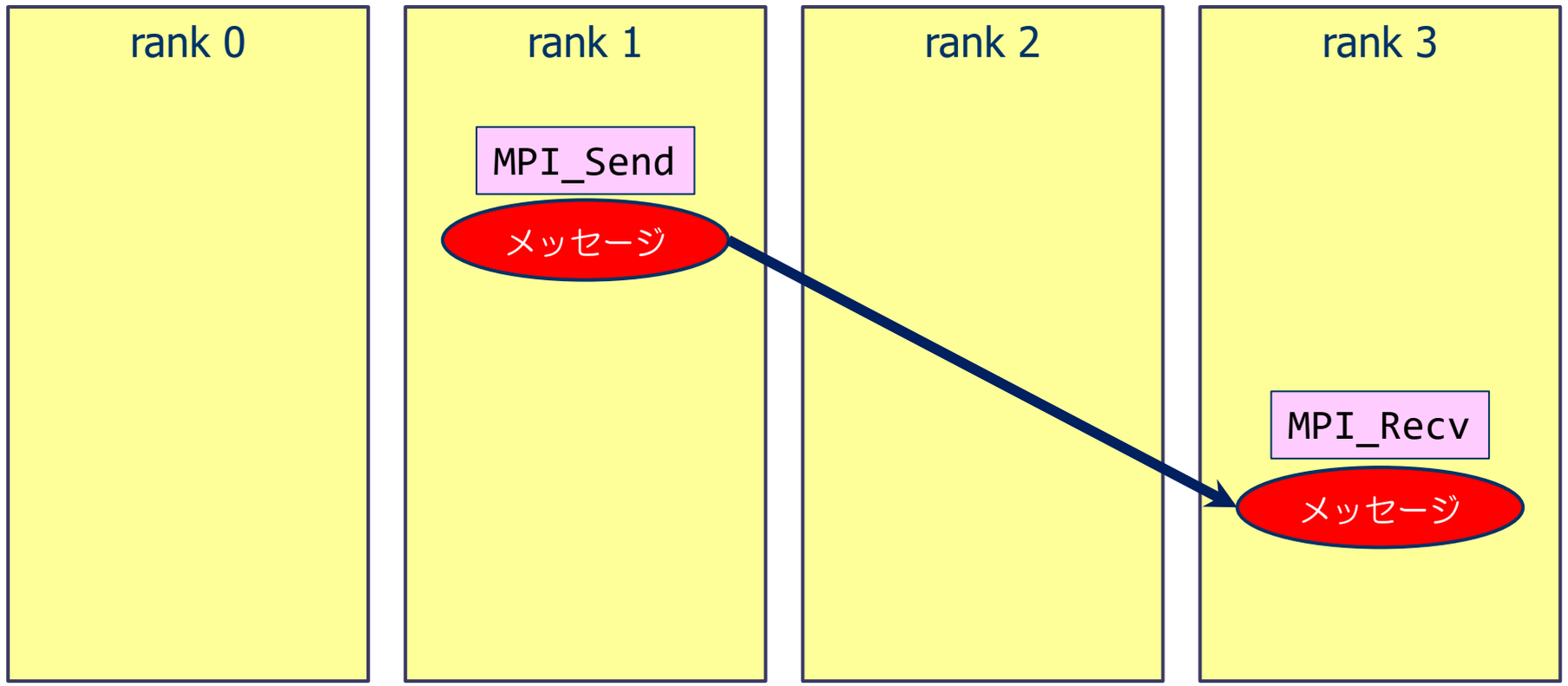
KOBE HPC サマースクール 2024（初級）

10. 1対1通信関数，集団通信関数

1対1通信とは：通信関数MPI_Send, MPI_Recv

あるrank（例えばRank 1のプロセスから，rank 3のプロセスにデータ（メッセージ）を送る

例）



１対１通信関数を用いたMPIプログラム

【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.

MPIプログラム（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_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, MPI_STATUS_IGNORE );
   }
   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，この引数は無いので注意．
  ※特に状況情報が必要ない場合は MPI_STATUS_IGNORE を指定しておくことが可能．

※ 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
  - 送受信するデータの先頭アドレスを指定する.
    - 変数なら「&buff」，配列の要素 i が先頭なら「&buff[i]」など（C言語の場合）．Fortranなら「buff」，「buff(i)」.
  - 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ｘｘ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10-1　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 /home/guest60/share/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.

- icx sum.c -lmpi

③ jobm.sh を必要に応じて修正して，ジョブを実行.

④ バッチジョブ出力結果（sum.onnnnnn）を確認する.

- 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.

ジョブキャンセル

- 3分待って計算が完了しなければ，ジョブは強制終了されます．ジョブスクリプトの walltime で上限時間を変更可能です．ただし本演習の範囲では3分経っても完了しない場合，間違いがある可能性大です．
  - バッチジョブ・スクリプトの例 (プロセス数 2 の場合)

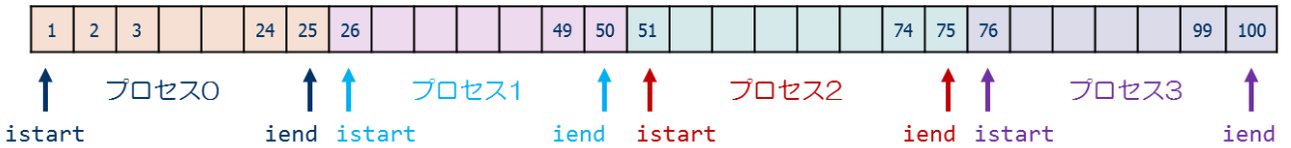
```
#!/bin/bash
#PBS -q WS
#PBS -l select=1:ncpus=2:mpiprocs=2
#PBS -l walltime=00:03:00

#PBS -N hello
#PBS -j oe

source /etc/profile.d/modules.sh
module load intel/2022.3.1
module load mpt
cd ${PBS_O_WORKDIR}
mpiexec_mpt -np 2 dplace -s1 ./a.out
```

- ジョブを途中で終了したい場合は「qdel ジョブ番号」でキャンセルして下さい．

演習10-2（発展）ここまで完了した人は講師・TAに知らせてください

- 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.

集団通信関数を用いたMPIプログラム

- 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（+，×，max，minなど）を適用する関数

集団通信関数を用いたMPIプログラム（sum_reduction.c）

※ついでに解く問題を「1から10000まで整数の和（答えは50005000）」にする

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

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

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

   if( myrank == 0 ) n = 10000;
   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 );
   // ※配布プログラムではここにMPI_Gatherを用いた別解
   if( myrank == 0 ) printf("Sum = %d¥n", sum );

   MPI_Finalize();
   return 0;
}
```

MPIプログラム（sum_reduction.c）

- 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

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

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


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

   if( myrank == 0 ) n = 10000;
   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 );
   // ※配布プログラムではここにMPI_Gatherを用いた別解
   if( myrank == 0 ) printf("Sum = %d¥n", sum );

   MPI_Finalize();
   return 0;
}
```

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

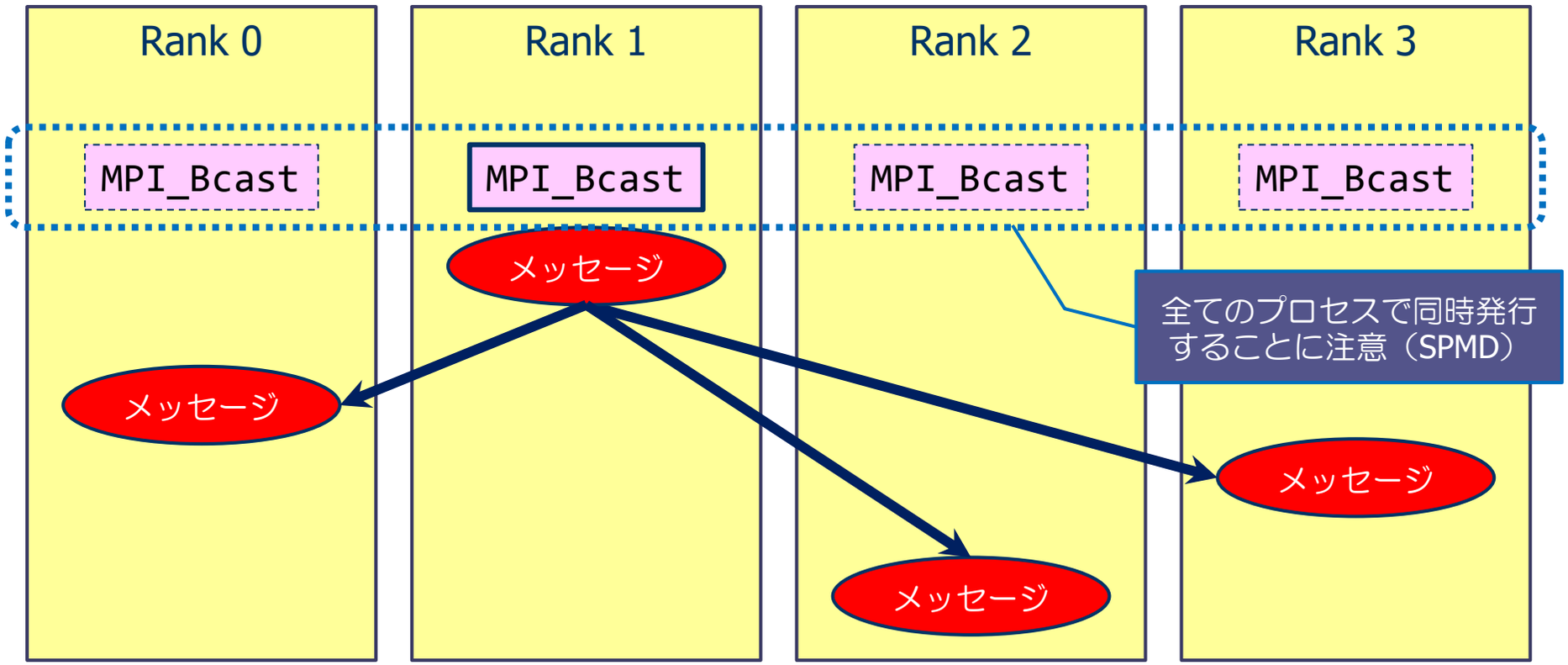
部分和の総和を計算
（プロセス0に集める）

プロセス0だけが結果を出力

集団通信 ― broadcast

ある単一のプロセスから，すべてのプロセスにデータ（メッセージ）を送る．

例）



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．

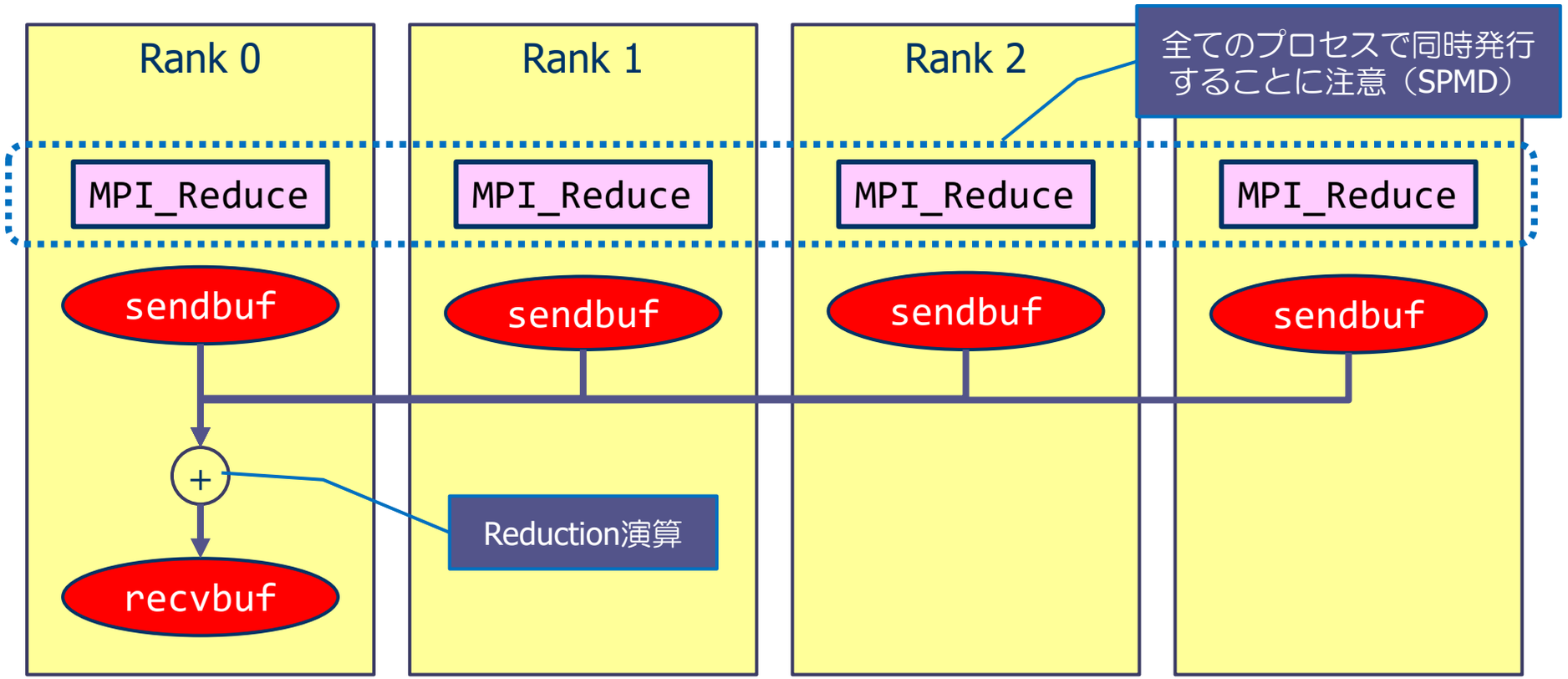
- buff:　　送り主（root）が送信するデータの変数名（先頭アドレス）
  他のMPIプロセスは，同じ変数名でデータを受け取る．
- count:　　データの個数
- datatype:　　送信するデータの型
  - MPI_INT, MPI_DOUBLE, MPI_CHAR など
- root:　　送り主のMPIプロセス番号
- comm:　　コミュニケータ（例えば，MPI_COMM_WORLD）

※ 関数の戻りコードは，エラーコードを表す．

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．

例）



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．

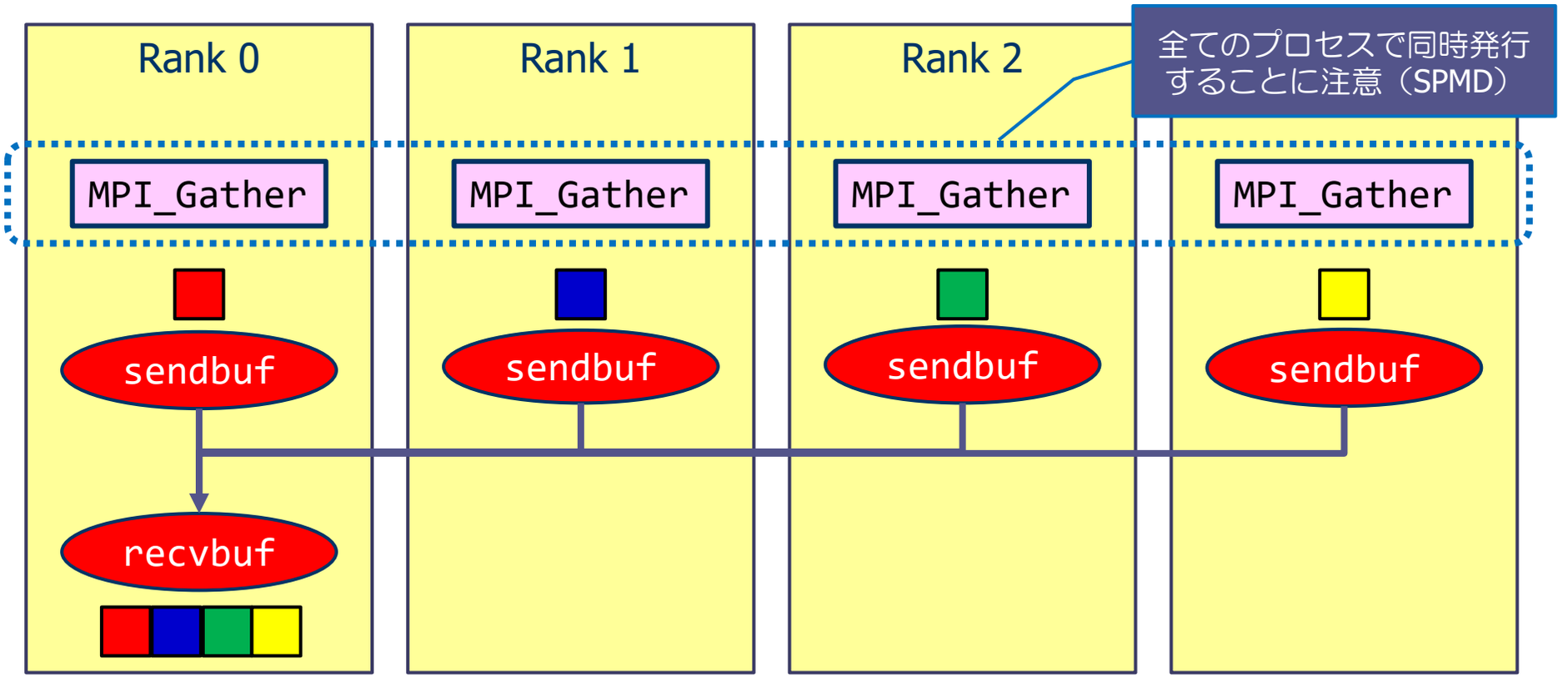
リダクション演算とは

- 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 $\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \cdots, \boldsymbol{x}_m$ をそれぞれ長さ n のベクトルとするとき，それらの和 $\boldsymbol{x} = \boldsymbol{x}_1 + \boldsymbol{x}_2 + \boldsymbol{x}_3 + \cdots + \boldsymbol{x}_m$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．

1つのプロセスへの集約：MPI_Gather

全プロセス上の同一長のデータを一つのプロセスに集めて連結する.

例）



集団通信 ー gather

```
int MPI_Gather(  void *sendbuff, int sendcount, MPI_Datatype sendtype,
                 void *recvbuff, int recvcount, MPI_Datatype recvtype,
                 int root, MPI_Comm comm )
```

- sendbuff:　送信するデータの変数名（先頭アドレス）
- sendcount: 送信データの個数（整数型）
- recvbuff:　受信するデータの変数名（先頭アドレス）
- recvcount: 受信データの個数（整数型）
  - この要素数は，1プロセス当たりの送信データ数と同じにする
- sendtype, recvtype: 送受信するデータの型
  - MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR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．

演習10-3a　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 /home/guest60/share/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.

- icx sum_reduction.c -lmpi

③ jobm.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 = 50005000）が出力されているか？

演習10-3b（発展1）

- プロセス毎に部分和を出力した後，どこかのプロセス（例えば rank #0）で総和を計算し，その総和をすべてのプロセスで出力をせよ.
  - 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.

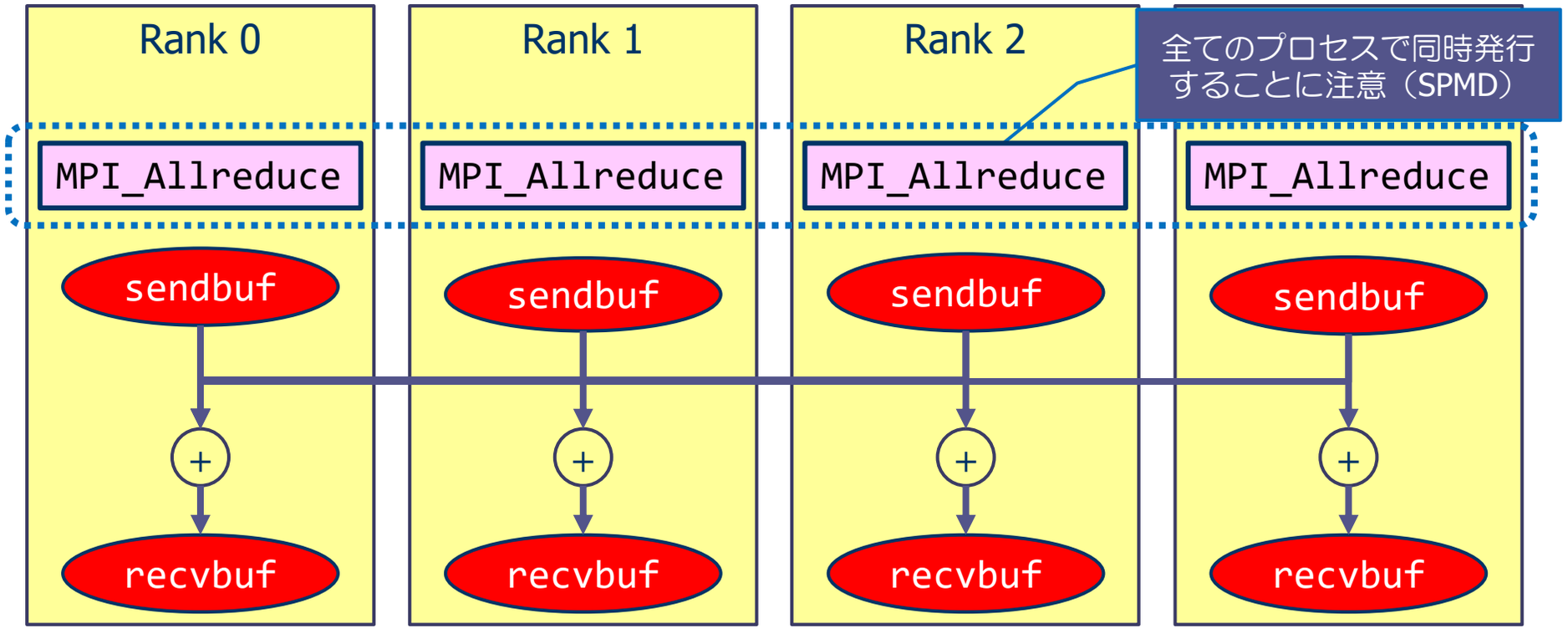
```
printf("Rank: %d -> Local sum = %d", myrank, sum_local);
```

- 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．

例）



「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．

良く使われる集団通信関数

放送	`MPI_Bcast`
リダクション	`MPI_Reduce`
	`MPI_Allreduce`
集約	`MPI_Gather`
	`MPI_Allgather`
拡散	`MPI_Scatter`
完全交換	`MPI_Alltoall`
バリア同期	`MPI_Barrier`

- 引数仕様は使い方はWeb上に多数情報提供されている
  - 基本的に必要になったら調べながら使う
- 集団通信は全プロセス（正確にはコミュニケータに含まれる全プロセス）が呼び出すようにすること

演習10-3c（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10-4a MPI_Allreduceの応用：ベクトルの正規化

- n次元ベクトル $\boldsymbol{x}$（`double x[N]`, `real*8 :: x(N)`）の i 番目の要素を i とする.
  - C言語版（配列要素番号は 0 から開始）：$x[i] = i + 1$　（$i = 0, \cdots, n-1$）
  - Fortran言語版（配列要素番号は 1 から開始）：$x(i) = i$　（$i = 1, \cdots, n$）

- $\boldsymbol{x}$ を正規化したベクトル $\boldsymbol{x}/\|\boldsymbol{x}\|_2$ を求めるプログラム normalize.{c/f90} 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．$\|\boldsymbol{x}\|_2$ は $\boldsymbol{x}$ 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
    $\tilde{x}[i] = (i+1)/\sqrt{n(n+1)(2n+1)/6}$（C），$\tilde{x}(i) = i/\sqrt{n(n+1)(2n+1)/6}$（Fortran）
    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.

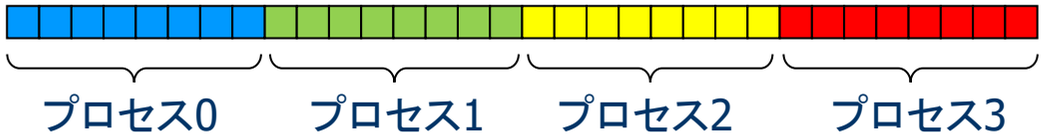
```
$ cp /home/guest60/share/normalize.c ./
$ icx normalize.c
$ ./a.out
Error = 0.000000
```

```
$ cp /home/guest60/share/normalize.f90 ./
$ ifx normalize.f90
$ ./a.out
Error = 0.0000
```

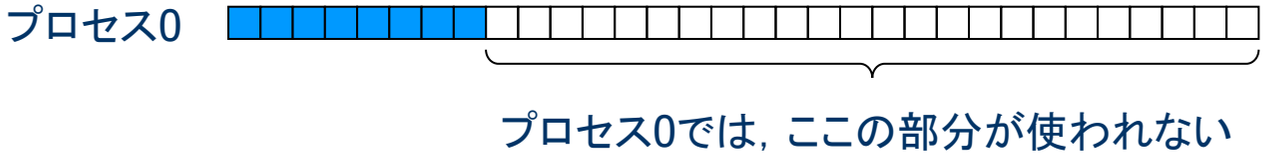
演習10-4b　normalize.{c/f90}を並列化せよ

※ここまで出来た人は今日は終了です．

- 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_recv
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 )
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_recv, 1, MPI_INTEGER, 1, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr )
end if
if( myrank == 0 )  print *, 'sum =', isum_local+isum_recv
call MPI_Finalize( ierr )
end program sum100_by_mpi
```

MPIプログラム（sum.f90）

- 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

```
program sum100_by_mpi
use mpi
implicit none
integer :: i, istart, iend, isum_local, isum_recv
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 )
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_recv, 1, MPI_INTEGER, 1, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr )
end if
if( myrank == 0 )  print *, 'sum =', isum_local+isum_recv
call MPI_Finalize( ierr )
end program sum100_by_mpi
```

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

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）

- 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

```
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 )
! ※配布プログラムではここにMPI_Gatherを用いた別解
if( myrank == 0 )  print *, 'sum (by reduction function) =', isum

call MPI_Finalize( ierr )
end program sum_by_reduction
```

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 ー 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 ー gather

```
mpi_gather( sendbuff, sendcount, sendtype, recvbuff, recvcount, recvtype, root, comm, ierr )
```

- sendbuff:　送信するデータの変数名（先頭アドレス）
- sendcount: 送信データの個数（整数型）
- recvbuff:　受信するデータの変数名（先頭アドレス）
- recvcount: 受信データの個数（整数型）
  - この要素数は，1プロセス当たりの送信データ数と同じにする
- sendtype, recvtype: 送受信するデータの型
  - MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_REAL8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 ー 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）