

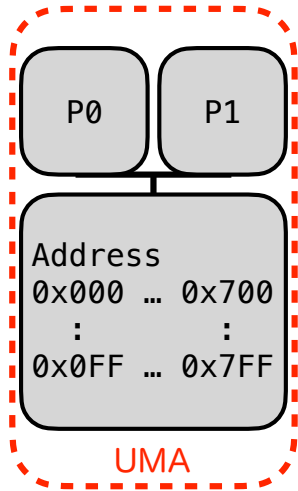
# 並列プログラミング入門（MPI編）

# 目次

- 第1部 (80分)
  - 並列処理に関する基礎概念 (pp.3-7)
  - MPI (Message-Passing Interface) 規格 (pp.8-12)
  - MPIプログラムの実行環境の一例 (pp.13-22)
  - 基本的なMPI関数の紹介1：最初のMPIプログラムの実行例 (pp.23-27)
- 第2部 (60分)
  - 基本的なMPI関数の紹介2：Point-to-Point Communicationの実行例を中心に (pp.28-52)
- 第3部 (80分)
  - 基本的なMPI関数の紹介3：Collective Communicationの実行例を中心に (pp.53-93)
- 第4部 (60分)
  - 基本的なMPI関数の紹介4：基本的なCollective Communication関数の紹介 (pp.94-134)
  - スケーリング則 (pp.135-137)

# 並列処理に関する基礎概念

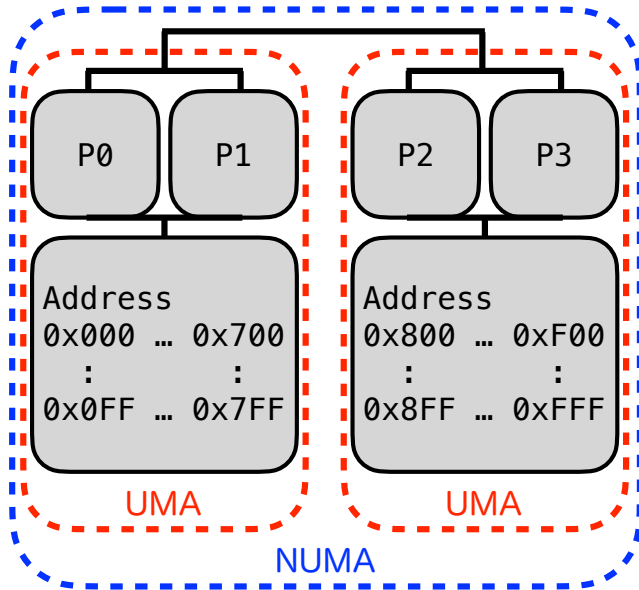
## 共有メモリ（Shared Memory）型並列計算機



- 1つのOS（Operating System）が、プロセッサ：「P0」「P1」と、メモリ空間「Address:0x000...0x7FF」の、全てを管理する。
- 各プロセッサは、それぞれ独立に、命令列を実行する。
- 各プロセッサは、それぞれ、メモリ空間の全体に、アクセスすることができる。
- 各プロセッサからメモリへのアクセスコストは均一。 → UMA（Uniform Memory Access）

# 並列処理に関する基礎概念

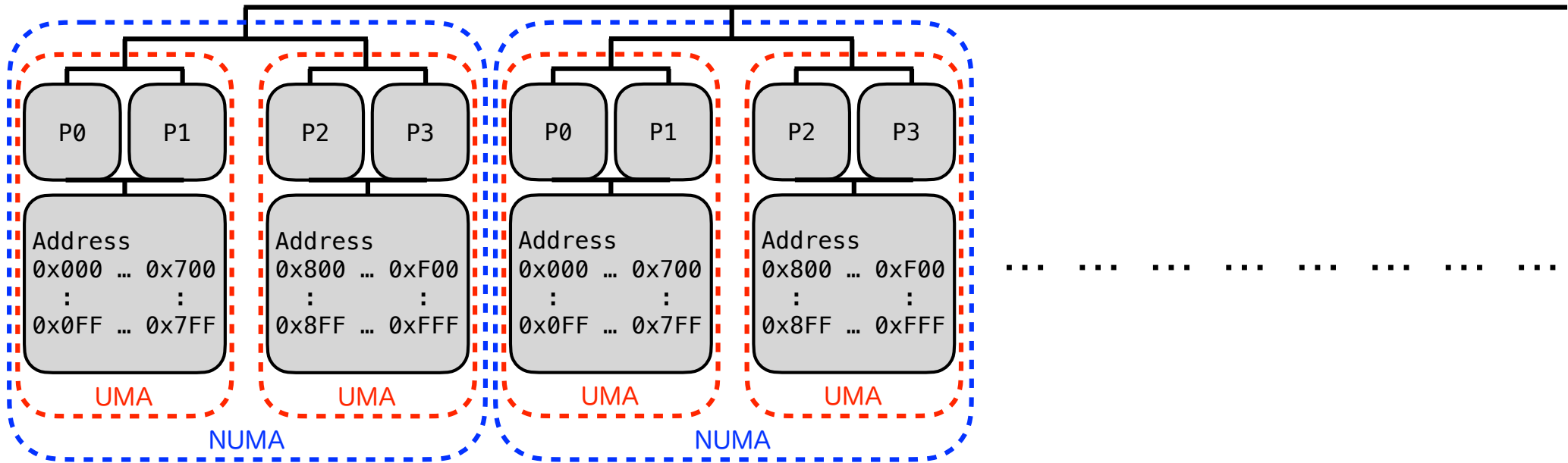
## 共有メモリ（Shared Memory）型並列計算機



- 1つのOS（Operating System）が、プロセッサ：「P0」「P1」「P2」「P3」と、メモリ空間：「Address:0x000...0x7FF」「Address:0x800...0xFFF」の、全てを管理する。
- 各プロセッサは、それぞれ独立に、命令列を実行する。
- 各プロセッサは、それぞれ、メモリ空間の全体に、アクセスすることができる。
- 「P0」から「0x800」へのアクセスコストは、「0x000」へのアクセスコストより大きい。  
各プロセッサからメモリへのアクセスコストは、アクセス先Addressにより異なる（不均一）。  
→ NUMA（Non-Uniform Memory Access）
- NUMAな共有メモリ型並列計算機の、UMAな一塊を、NUMAノードと呼ぶ。

# 並列処理に関する基礎概念

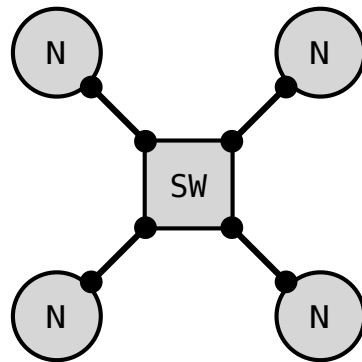
## 分散共有メモリ（Distributed Shared Memory）型並列計算機



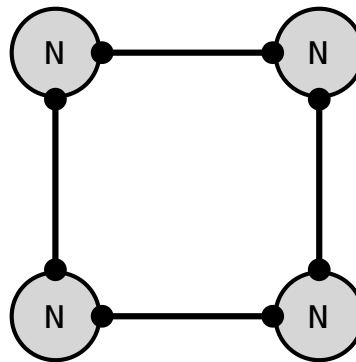
- 分散共有メモリ型並列計算機は、複数の共有メモリ型並列計算機が、相互結合網（ネットワーク）により連結されたもの。
- 分散共有メモリ型並列計算機を構成する、複数の共有メモリ型並列計算機の個々のことを、計算ノードと呼ぶ。  
（上図では、NUMAな一塊が1つの計算ノード。UMAな計算ノードもあり得る。）
- 計算ノード毎に個別にOS（Operating System）が動作し、各計算ノードのメモリ空間も互いに独立している。

# 並列処理に関する基礎概念

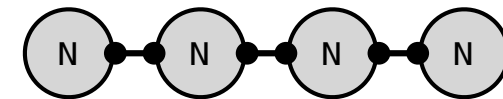
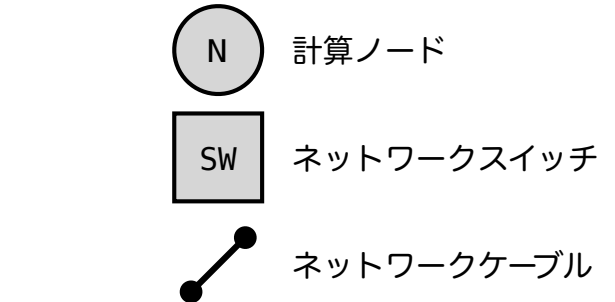
ネットワークの構造（Network Topology）の基本形



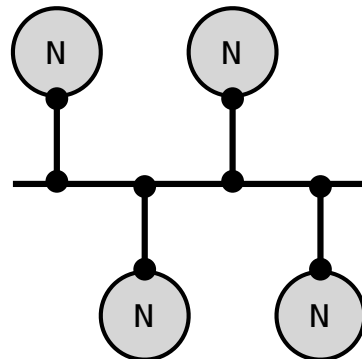
スター（Star）型



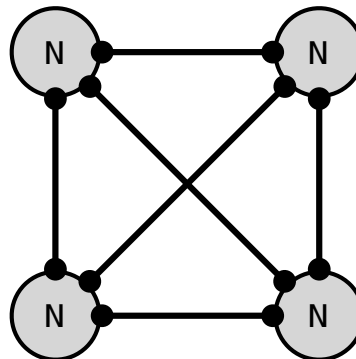
リング（Ring）型



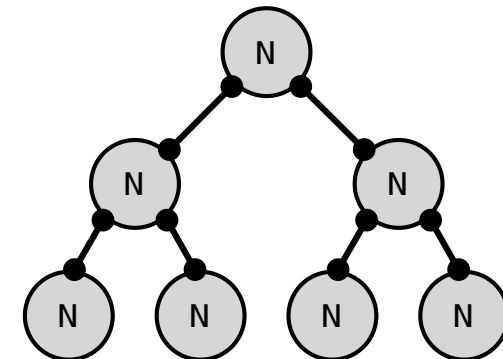
ライン（Line）型



バス（Bus）型




フルメッシュ（Full Mesh）型



ツリー（Tree）型

# 並列処理に関する基礎概念

## 分散共有メモリ型並列計算機を使う必要性

- 1つの計算ノードでは実行不可能な、大規模（演算量が多い or/and メモリ使用量が多い）な処理を行うためには、処理を分割して、複数の計算ノードに割り振る必要がある。  
 1つの計算ノードで十分足りる処理を行うために、MPIプログラムを作成する必要はない。
- 共有メモリ型並列計算機は、メモリアクセスの排他制御が必要なため、プロセッサ数をあまり多くすることができない。数個から数十個までが一般的。

# MPI (Message-Passing Interface) 規格

関数  $f(x, t)$  の時間発展を分散メモリ型並列計算機で計算する場合

- 空間微分 (中心差分)

$$\left( \frac{\partial f}{\partial x} \right)_i = \frac{f_{i+1} - f_{i-1}}{2\Delta x}$$

- 時間発展

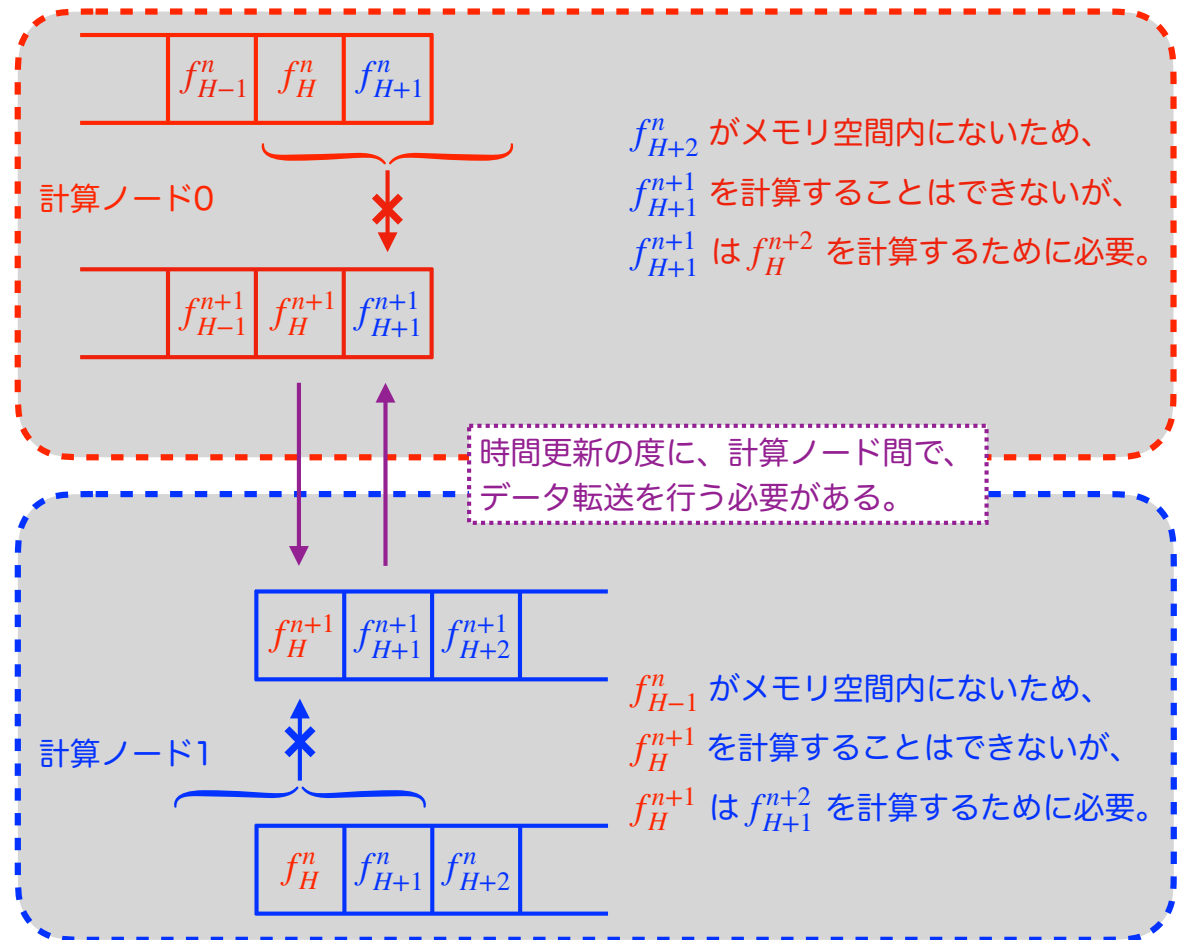
$$\begin{aligned} \{ f_{i-1}^n, f_i^n, f_{i+1}^n \} &\rightarrow f_i^{n+1} \\ \{ f_{i-1}^{n+1}, f_i^{n+1}, f_{i+1}^{n+1} \} &\rightarrow f_i^{n+2} \\ &\vdots \end{aligned}$$

- 領域分割

$f_i^n$  ( $i_{min} \leq i \leq H, 0 < n$ ) を計算ノード0で計算し、  
 $f_i^n$  ( $H < i \leq i_{max}, 0 < n$ ) は計算ノード1で計算する。

↓

各計算ノードが、それぞれ異なるデータに対して、  
同一のタスクを実行することを、データ並列と呼ぶ。



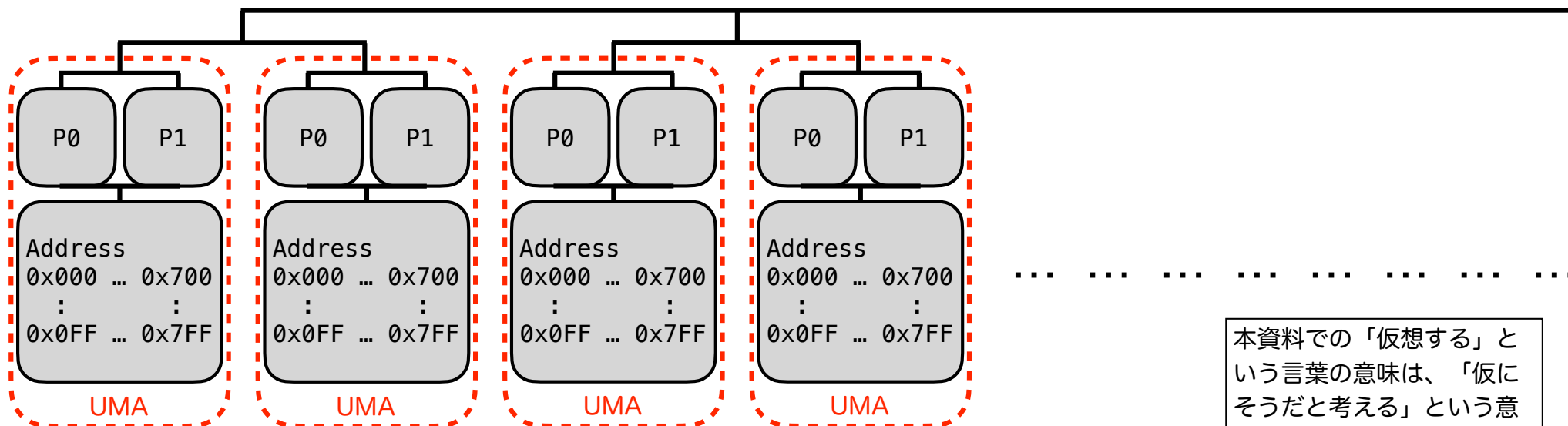
MPI規格とは、計算ノード間（正確にはプロセス間）でのデータ転送を行うための関数などのインターフェースの仕様を定めた規格

MPI3.1の規格書 (836p) → <https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>



# MPI (Message-Passing Interface) 規格

## MPIによる並列プログラミングの前提



本資料での「仮想する」という言葉の意味は、「仮にそうだと考える」という意味であり、「仮想マシン (VM) を構築する」という意味ではない。

- MPIプログラムを作成する（MPI規格で定義された関数などを含むソースプログラムを書く）際には、UMAな計算ノードをネットワークで接続した、分散共有メモリ型並列計算機を、独占していると仮想する。このとき、UMAな計算ノードの構成（プロセッサ数やメモリ容量など）、UMAな計算ノードの数、ネットワークの構造も仮想する。
- 仮想のUMAな計算ノードの個々のことを、プロセスと呼ぶ。
- プロセスの数は、MPIプログラムを実行するときに、その実行を指示する人（ユーザー）が決定し、指定する。
- どのプロセスを、どの実物の計算ノードに割り当てるかは、MPIプログラムを実行するときに、計算機システムのジョブスケジューラが決定する。その割り当ては、実行の度に変わり得る。
- 全てのプロセスにおいて、それぞれ独立に、同一ロードモジュールファイル内の命令列が実行される。  
→ SPMD (Single Program, Multiple Data streams)

# MPI (Message-Passing Interface) 規格

MPI規格に記載のある機能（MPI3.1の規格書の第3章から第14章までの題名）

3. Point-to-Point Communication
4. Datatype
5. Collective Communication
6. Groups, Contexts, Communicators and Caching
7. Process Topologies
8. MPI Environmental Management
9. The Info Object（情報オブジェクト）

MPI3.1の規格書には定めのない、MPI関数の動作を制御するための、追加の情報を格納するための箱。その箱の中に、どのような情報を格納するかについては、MPI3.1の規格書には定めはなく、計算機システムの実装に依存する。ユーザーアプリケーションプログラム中では、その箱の中身に直接アクセスすることはできず、MPI\_Info型の変数によりその箱を指定する。

10. Process Creation and Management
11. One-Sided Communications
12. External Interface
13. I/O
14. Tool Support（ツール支援）

MPIプログラムの実行状態を把握するためのツール（デバッガやプロファイラ等）のための機能。

本資料では、ユーザーアプリケーションプログラム中での使用が想定されるMPI3.1の機能（赤字）のそれぞれについて、基礎的な部分を、サンプルプログラム付きで紹介する（順序不同）。

# MPI (Message-Passing Interface) 規格

用語の定義 (MPI3.1の規格書の第2章の題名)

## 2. MPI Terms and Conventions

# MPI (Message-Passing Interface) 規格

## MPIプログラムを実行するためのコマンド

- C言語で書かれたMPIプログラムのコンパイルとリンク  
(MPI3.1規格では定められていない)  
(本資料には、Linux上のMPICHとGCCの組み合わせのものを記載する)

```
mpicc -o ロードモジュールファイル名 ソースプログラムファイル名
```

ソースプログラムファイルを基に、ロードモジュールファイルが生成される。

- MPIプログラムの実行 (MPI3.1の規格書の第8章の記述の要約)

```
mpiexec -n 実行プロセス数 ロードモジュールファイル名
```

指定された数のプロセスが生成され、各プロセスにおいて、それぞれ独立に、同一ロードモジュールファイル内の命令列が実行される。

# MPIプログラムの実行環境の一例

## 本資料における実行環境

本資料には、以下の3つ名称は、以下のものに置換して記載する。

- ・ 計算ノードのホスト名：cmptnd00, cmptnd01, ...
- ・ ジョブパーティション名：testq
- ・ ユーザー名：user01

### ・ ハードウェア

- ・ ログインノード：1台
- ・ 計算ノード：多数
  - ・ cmptnd00：NUMAノード（論理プロセッサ数：2個以上）数：2個以上
  - ・ cmptnd01：NUMAノード（論理プロセッサ数：2個以上）数：2個以上
  - ・ cmptnd02：NUMAノード（論理プロセッサ数：2個以上）数：2個以上

： HPC向けの計算機では、ユーザーが、計算ノードにログインして、そこでコマンドを実行することは、通常ない。  
ログインノードにて、ジョブ投入コマンドを実行することにより、計算機システムに処理の実行を依頼するのが一般的。

### ・ ソフトウェア

- ・ OS：Linux
- ・ Cコンパイラ：GCC
- ・ MPIライブラリ：MPICH
- ・ ジョブスケジューラ：SLURM
  - ・ ジョブを投入するためのコマンド

```
sbatch -p ジョブパーティション名 -t ジョブの実行時間の最大値（時:分:秒） ジョブスクリプトファイル名
```

- ・ ジョブスクリプトファイル内で指定された数の計算ノードを確保し、ジョブスクリプトファイル内に記載された処理を実行することを、計算機システムに依頼する。
- ・ このコマンドを発行すると、ジョブIDが振られ、ジョブスケジューラのキューに登録され、指定された数の計算ノードが確保されるまで、実行待ち状態となる。
- ・ ジョブパーティション名は、使用する計算機資源の区分を示す、各計算機システムにおいて既定の文字列。
- ・ ジョブのリストを表示するためのコマンド

```
squeue
```

- ・ ジョブスケジューラのキューに登録されているジョブのリストを表示させる。

# MPIプログラムの実行環境の一例

## 本資料における実行環境

- ・ サンプルプログラム

- ・ 22個のサンプルプログラムを紹介する。
  - ・ 22個の内、21個の先頭18行は同一。残り1個については、先頭18行の内の2行に、文字列が追加される。
  - ・ 22個全ての、末尾4行は同一。

- ・ 実行プロセス数

- ・ サンプルプログラムのそれぞれを、4プロセスで実行した結果を紹介する。
  - ・ 1計算ノードあたりに、2プロセスを割り当てての実行。
  - ・ 各プロセスを、別々のNUMAノードにバインドしての実行。

各計算ノードに、計算ノード内のNUMAノードと同数のプロセスを割り当て、**各々のプロセスを別々のNUMAノードにバインドして実行するのが**、MPIプログラムの**標準的**な実行様式（計算ノードが2つのNUMAノードで構成されているのであれば、その計算ノードには2つのプロセスを割り当てるのが標準的）。

# MPIプログラムの実行環境の一例

## コンパイルとリンクをする（ロードモジュールファイルを作成する）

コンパイルとリンクの実行前後のファイルリスト

（ログインノードで計算ノード用のコンパイルとリンクができる場合の、ログインノードの画面表示）

```
・ クリーム色の文字は実際の画面上には実在しない文字。
$
$ ls -l <--・ カレントディレクトリ内のファイルリストを表示させる。
exec.sh
mk_machinefile.awk
numa_bind_exec.sh
world.c <-----・ MPIプログラム「world」のソースプログラムファイル
$
$ mpicc -o world world.c <--・ C言語で書かれたMPIプログラム「world」のコンパイルとリンクをする。
$
$ ls -l <--・ カレントディレクトリ内のファイルリストを表示させる。
exec.sh
mk_machinefile.awk
numa_bind_exec.sh
world <-----・ MPIプログラム「world」のロードモジュールファイル
world.c
$
```

# MPIプログラムの実行環境の一例

ジョブを投入する（計算ノードの割当てと、ジョブスクリプトの実行を、計算機システムに依頼する）

ジョブ実行前後のファイルリスト（ログインノードの画面表示）

```
・ クリーム色の文字は実際の画面上には実在しない文字。
$
$ ls -l <----- ・ カレントディレクトリ内のファイルリストを表示させる。
exec.sh <----- ・ ジョブスクリプトファイル（第1引数として指定された名前のロードモジュールを、2ノード4プロセスで実行する）
mk_machinefile.awk <-- ・ 「exec.sh」 内で呼び出されるAWKスクリプトファイル（machinefileを作成する）
numa_bind_exec.sh <-- ・ 「exec.sh」 内で呼び出されるシェルスクリプトファイル
world <----- ・ MPIプログラム「world」のロードモジュールファイル（「numa_bind_exec.sh」内で実行される）
world.c
$
$ sbatch -p testq -t 00:01:00 exec.sh world <-- ・ ジョブを投入する。
Submitted batch job 93373 <----- ・ ジョブIDが「93373」のジョブとして受け付けられたとの表示
$
$ squeue <-- ・ ジョブIDが「93373」のジョブが、ジョブスケジューラに登録されていることを確認する。
      JOBID PARTITION   NAME   USER ST   TIME  NODES NODELIST(REASON)
      93373      testq  exec.sh  user01 PD    0:00       2 (Resources)
      ↑-- ・ ジョブスクリプトファイル名が、ジョブ名となる。

・ squeueコマンドを何度か実行し、ジョブIDが「93373」のジョブに関する情報が表示されなくなる（ジョブが終了する）まで待つ。
$
$ ls -l <-- ・ カレントディレクトリ内のファイルリストを表示させる。
exec.sh
exec.sh_93373.machinefile <-- ・ 「mk_machinefile.awk」の出力ファイル（machinefile）
exec.sh_93373.stderr <----- ・ ジョブの標準エラー出力ファイル
exec.sh_93373.stdout <----- ・ ジョブの標準出力ファイル
mk_machinefile.awk
numa_bind_exec.sh
world
world.c
world_0.txt <-- ・ MPIプログラム「world」の出力ファイル
world_1.txt <-- ・ MPIプログラム「world」の出力ファイル
world_2.txt <-- ・ MPIプログラム「world」の出力ファイル
world_3.txt <-- ・ MPIプログラム「world」の出力ファイル
$
```

・ 「exec.sh」 内にて、  
ジョブ名は、環境変数「SLURM\_JOB\_NAME」  
ジョブIDは、環境変数「SLURM\_JOB\_ID」  
として参照可能。



# MPIプログラムの実行環境の一例

ジョブを投入する（計算ノードの割当てと、ジョブスクリプトの実行を、計算機システムに依頼する）

ジョブスクリプトファイル「exec.sh」の内容

割り当てられた計算ノードのうちの1つで、以下の実行文が実行される。

「#SBATCH」で始まる行は、sbatchコマンドのオプションを指定するための行であり、実行文ではない。

・茶色の文字は実際のジョブスクリプトファイル中には実在しない文字

```
1 #!/bin/bash
2 #SBATCH --nodes=2                ## SLURM_JOB_NUM_NODES <--- ・このジョブのために計算ノードを2つ要求する。
3 #SBATCH --output=%x_%j.stdout   ## %x : SLURM_JOB_NAME <--- ・このジョブの標準出力ファイルの名前を指定する。
4 #SBATCH --error=%x_%j.stderr   ## %j : SLURM_JOB_ID <----- ・このジョブの標準エラー出力ファイルの名前を指定する。
5 set -x
6 #
7 LOAD_MODULE_PATH=. <----- ・MPIプログラムのロードモジュールファイルが存在するディレクトリ名
8 LOAD_MODULE_NAME=${1} <--- ・MPIプログラムのロードモジュールファイル名（「exec.sh」の第1引数として与えられた値）
9 NUM_PROCS=4 <----- ・MPIプログラムの実行プロセス数
10 MAX_PROCS_PER_NODE=2 <---- ・1つの計算ノードに割り当てるプロセス数の最大値（計算ノード内のNUMAノード数とするのが基本）
```

```
11 MACHINE_FILE_NAME=${SLURM_JOB_NAME}_${SLURM_JOB_ID}.machinefile
12 #                               ↑-- ・各プロセスを、それぞれ、どの計算ノードにおいて生成するかを指定するための
                               マップを格納するためのファイル（machinefile）の名前
```

・環境変数「SLURM\_JOB\_NODELIST」には、このジョブのために確保された計算ノードのリストが入っている。

```
13 echo ${SLURM_JOB_NODELIST} | \
14   awk -f mk_machinefile.awk -v mppn=${MAX_PROCS_PER_NODE} > ${MACHINE_FILE_NAME}
15 #                               ↑-- ・環境変数「SLURM_JOB_NODELIST」を基に、machinefileを作成する。
```

```
16 mpiexec -n ${NUM_PROCS} --machinefile ${MACHINE_FILE_NAME} \ <--- ・指定された数のプロセスを、各々指定された
17   bash numa_bind_exec.sh ${MAX_PROCS_PER_NODE} \           計算ノードにおいて生成し、そのそれぞれに
18   ${LOAD_MODULE_PATH} ${LOAD_MODULE_NAME}                  おいて、ファイル「numa_bind_exec.sh」内の
19 #                                                            実行文を実行する。
```

・16～18行目を「mpiexec -n \${NUM\_PROCS} \${LOAD\_MODULE\_NAME}」と書くことも可能だが、その場合、各プロセスを、それぞれ、どの計算ノードにおいて生成するかは、使用する計算機システムにおける既定の方式による。

# MPIプログラムの実行環境の一例

ジョブを投入する（計算ノードの割当てと、ジョブスクリプトの実行を、計算機システムに依頼する）

AWKスクリプトファイル「mk\_machinefile.awk」の内容

ジョブスクリプトファイル「exec.sh」内の実行文を実行中の計算ノードで、以下の実行文が実行される。

仮定のUMAな計算ノード（プロセス）のそれぞれを、実物の計算ノードのどれに割り当ててを決定する。

```
1 BEGIN{
2     FS="[[,]"
3     num=0
4 }
5 {
6     for(i=2;i<NF;i++){
7         split($i,a,"-")
8         switch(length(a)){
9             case 1:
10                 j=a[1]
11                 host_names[num]=sprintf("%s%02d",$1,j)
12                 num++
13                 break
14             case 2:
15                 for(j=a[1];j<=a[2];j++){
16                     host_names[num]=sprintf("%s%02d",$1,j)
17                     num++
18                 }
19                 break
20             }
21         delete a
22     }
23 }
24 END{
25     for(i=0;i<num;i++){
26         for(j=0;j<mppn;j++){
27             print host_names[i] <--
28         }
29     }
30 }
```

・茶色の文字は実際のAWKスクリプトファイル中には実在しない文字

・このジョブのために、2つの計算ノード「cmptnd01, cmptnd02」が確保された場合には、環境変数「SLURM\_JOB\_NODELIST」には、文字列「cmptnd[01-02]」が格納されており、その文字列がこのAWKスクリプトの処理の対象。  
(4つの計算ノード「cmptnd01, cmptnd02, cmptnd05, cmptnd06」が確保された場合には、環境変数「SLURM\_JOB\_NODELIST」には、文字列「cmptnd[01-02,05-06]」が格納されている)

・このジョブに割り当てられた全ての計算ノードのホスト名を、順番に、1つの計算ノードに割り当ててるプロセス数「mppn」回ずつ書き出す。

# MPIプログラムの実行環境の一例

ジョブを投入する（計算ノードの割当てと、ジョブスクリプトの実行を、計算機システムに依頼する）

ジョブスクリプトファイル「exec.sh」が実行された時に、  
実際に発行されたコマンドと、生成されたmachinefileの内容

・ クリーム色の文字は実際の画面上には実在しない文字

```
$  
$ head -n 8 exec.sh_93373.stderr <--・「exec.sh」により、実際に発行されたコマンドを表示させる。  
+ LOAD_MODULE_PATH=.  
+ LOAD_MODULE_NAME=world  
+ NUM_PROCS=4  
+ MAX_PROCS_PER_NODE=2  
+ MACHINE_FILE_NAME=exec.sh_93373.machinefile  
+ awk -f mk_machinefile.awk -v mppn=2 <-----・machinefileを生成した。生成されたmachinefileの中身は下記の通り。  
+ echo 'cmptnd[06-07]'  
+ mpiexec -n 4 --machinefile exec.sh_93373.machinefile bash numa_bind_exec.sh 2 . world  
$                                     ↑--・2ノード4プロセスで、ロードモジュールファイル「world」内の命令列を実行した。  
$                                     その際、各プロセスを、それぞれ、別々のNUMAノードにバインドした。  
$  
$ cat exec.sh_93373.machinefile <--・machinefileの内容を表示させる。  
cmptnd06 <-----・「${PMI_RANK} -eq 0」のプロセスが生成される計算ノードのホスト名前  
cmptnd06 <-----・「${PMI_RANK} -eq 1」のプロセスが生成される計算ノードのホスト名前  
cmptnd07 <-----・「${PMI_RANK} -eq 2」のプロセスが生成される計算ノードのホスト名前  
cmptnd07 <-----・「${PMI_RANK} -eq 3」のプロセスが生成される計算ノードのホスト名前  
$  
・環境変数「PMI_RANK」には、mpiexecコマンドで生成された  
  各プロセスに付けられた「0」から始まる通し番号が入っている。
```

# MPIプログラムの実行環境の一例

ジョブを投入する（計算ノードの割当てと、ジョブスクリプトの実行を、計算機システムに依頼する）

シェルスクリプトファイル「numa\_bind\_exec.sh」の内容

「numa\_bind\_exec.sh」を実行する各プロセスにおいて、それぞれ以下の実行文が実行される。

各プロセスが、それぞれ別々の実物のNUMAノードを選択し、それぞれ独立に、

3番目の引数の値として与えられた名前の、ロードモジュールファイル内の命令列を実行する。

・ 茶色の文字は実際のシェルスクリプトファイル中には実在しない文字

1 export PATH=\${PATH}:\${2} <--- ・ 「numa\_bind\_exec.sh」の2番目の引数として与えられた値をパスに追加する。

2 NUMA\_NODE\_NUM=\$(( \${PMI\_RANK} % \${1} )) <--- ・ バインド先のNUMAノード番号を計算する。

環境変数「PMI\_RANK」には、mpiexecコマンドで生成された各プロセスに付けられた「0」から始まる通し番号が入っている。

「numa\_bind\_exec.sh」の1番目の引数には、1つの計算ノードに割り当てるプロセス数の最大値（計算ノード内のNUMAノード数とするのが基本）が与えられている。

3 numactl --cpunodebind=\${NUMA\_NODE\_NUM} --membind=\${NUMA\_NODE\_NUM} \${3}

↑--- ・ 使用するNUMAノードを指定して

コマンド（ここでは、「numa\_bind\_exec.sh」の3番目の引数として与えられた値、MPIプログラム本体）を実行する。

# MPIプログラムの実行環境の一例

## numactlコマンドの紹介

numactlコマンドによる、NUMA構成の確認

計算ノードを対話型操作した場合の画面表示（少数の計算ノードを短時間使用する場合に、対話型操作が可能な計算機システムも多い）

```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$ numactl --hardware <----- ・ 現行プロセスが存在する（実物の）計算ノードの、NUMA構成を表示させる。
available: 2 nodes (0-1) <-- ・ この計算ノードは、2つのNUMAノード（「0」と「1」）により構成されている。
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102
103 104 105 106 107 108 109 110 111 <-- ・ NUMAノード「0」の論理プロセッサ数は112個。
node 0 size: 64813 MB <----- ・ NUMAノード「0」のメモリ量は約64GiB。
node 0 free: 52328 MB
node 1 cpus: 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134
135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160
161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186
187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212
213 214 215 216 217 218 219 220 221 222 223 <-- ・ NUMAノード「1」の論理プロセッサ数は112個。
node 1 size: 65311 MB <----- ・ NUMAノード「1」のメモリ量は約64GiB。
node 1 free: 59646 MB
node distances:
node  0  1
  0:  10  20
  1:  20  10
$
```

# MPIプログラムの実行環境の一例

## numactlコマンドの紹介

numactlコマンドによる、NUMAポリシー設定の確認

計算ノードを対話型操作した場合の画面表示（少数の計算ノードを短時間使用する場合に、対話型操作が可能な計算機システムも多い）

・ クリーム色の文字は実際の画面上には実在しない文字

```
$
$ numactl --show <--- 現行プロセスの、NUMAポリシー設定を表示させる。
policy: default
preferred node: current
physcpubind: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102
103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128
129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154
155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180
181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206
207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223
cpubind: 0 1
nodebind: 0 1 <--- 現行プロセスでは、実行文の実行にあたって、「0」と「1」のどちらのNUMAノードのプロセッサも使用され得る。
membind: 0 1 <--- 現行プロセスでは、実行文の実行にあたって、「0」と「1」のどちらのNUMAノードのメモリ空間も使用され得る。
$
$
$ numactl --cpunodebind=1 --membind=1 numactl --show <--- NUMAノード「1」のみを使用して、
policy: bind
preferred node: 1
physcpubind: 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134
135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160
161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186
187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212
213 214 215 216 217 218 219 220 221 222 223
cpubind: 1
nodebind: 1 <--- NUMAノード「1」のプロセッサのみが使用される状態で、コマンドが実行されていることの表示
membind: 1 <--- NUMAノード「1」のメモリ空間のみが使用される状態で、コマンドが実行されていることの表示
$
```

# 基本的なMPI関数の紹介

最初のMPIプログラムの実行例：world.c

## ・このプログラムの目的

- ・ MPIプログラム「world」を2ノード4プロセスで実行し、各プロセスがそれぞれ別々の、実物のNUMAノードにバインドされていることを確認する。

## ・各プロセスでの処理の概要

- ・ MPIプログラムの実行にあたり、起動されたプロセスの数を表示する。
- ・ 現行プロセスの、MPI\_COMM\_WORLD内でのランク番号を表示する。  
(MPI\_COMM\_WORLD内でのランク番号とは、MPIプログラムの実行にあたり、起動されたプロセスの1つ1つに付された、ゼロから始まる通し番号)
- ・ 現行プロセスが動作している、計算ノードのホスト名を表示する。
- ・ 現行プロセスのプロセスIDを表示する。  
(各プロセスには、OSによる認識番号としてプロセスIDが付されている。計算ノード毎にOSが動作しているため、プロセスIDは、各計算ノード内でのみ有効な認識番号である)
- ・ 現行プロセスが動作しているNUMAノードのIDを表示する。
- ・ 現行プロセスのマスタースレッドが動作しているコアのIDを表示する。

現行プロセス: ロードモジュール側から見た  
自分を実行中のプロセス

# 基本的なMPI関数の紹介

最初のMPIプログラムの実行例：world.c

使用するMPI関数の説明（MPI3.1の規格書の第8章の記述の要約）

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

MPI関数の使用の開始を宣言する。

引数には、main関数の引数か、「NULL,NULL」を指定する。

```
int MPI_Finalize(void)
```

MPI関数の使用の終了を宣言する。

MPI関数は、エラー値を返却値とするが、その値は実装依存。  
エラー値ではない数字（秒数など）を返却するMPI関数も一部  
存在はする（MPI\_Wtimeなど）

Fortranの場合は、MPI関数ではなくMPIサブルーチンとなり、  
エラー返却値を格納するための整数型の変数を引数の最後に付加  
する必要がある。ただし、返却値がエラー値ではない数字（秒数  
など）の一部のMPI関数については、サブルーチンではなく関数  
のまま。

```
call MPI_Init(ierr)  
call MPI_Finalize(ierr)
```

コンパイル用コマンド（本資料における実行環境では）

C言語: mpicc

Fortran: mpifort

include文

C言語: include "mpi.h"

Fortran90以降: use mpi

Fortran2008以降: use mpi\_f08



# 基本的なMPI関数の紹介

## 最初のMPIプログラムの実行例：world.c

使用するMPI関数と定数の説明（MPI3.1の規格書の第6章の記述の要約）      引数の文字の色は、引数の属性（**入力**、**出力**、**入出力**）を表す。

```
MPI_Comm MPI_COMM_WORLD
```

MPIプログラム「world.c」を共に実行する、全てのプロセスをメンバーとするグループにおいて、プロセス間通信を行うためのコミュニケータを指定するための、MPI\_Comm型のハンドル（定数）。

コミュニケータ：プロセス間通信を行うための背景情報が格納されたブラックボックス

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

コミュニケータ「comm」内のプロセスの数を、変数「size」に格納する。

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

コミュニケータ「comm」内での、現行プロセスのID番号（ランク番号）を、変数「rank」に格納する。  
ランク番号の値は、ゼロから「size-1」までの何れかの整数。

# 基本的なMPI関数の紹介

## 最初のMPIプログラムの実行例：world.c

ソースプログラムファイル「world.c」の内容

各プロセスが、それぞれ独立に、以下の命令列を実行する。

**argv[0]**: プログラム名

**sizeW**: 「MPI\_COMM\_WORLD」内のプロセス数

**rankW**: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4 #include <unistd.h>
5 #include <time.h>
6 #include <sys/syscall.h>
7 #include <omp.h>
8 #include "mpi.h" <-----・MPI関数や定数の定義を読み込む。
9 int main(int argc, char *argv[]){
10     int sizeW, rankW;
11     char nameF0[256];
12     FILE *fp0;
13     MPI_Init(NULL, NULL); <-----・MPI関数の使用の開始を宣言する。
14     MPI_Comm_size(MPI_COMM_WORLD, &sizeW); <---・「MPI_COMM_WORLD」内のプロセス数を、変数「sizeW」に格納する。
15     MPI_Comm_rank(MPI_COMM_WORLD, &rankW);
        ↑---・現行プロセスの、「MPI_COMM_WORLD」内でのランク番号を、変数「rankW」に格納する。
16     sprintf(nameF0, "%s_%d.txt", argv[0], rankW); <-----・ファイル名が「argv[0]_rankW.txt」の形式のファイルを、
17     fp0 = fopen(nameF0, "w"); <-----各プロセスの標準的な出力先ファイル「fp0」とする。
18     fprintf(fp0, "rankW=%2d sizeW=%2d\n", rankW, sizeW);
19     int procid;
        ↑---・ファイル「fp0」の第1行目に、変数「rankW」と「sizeW」の値を書き込む。
20     unsigned int coreid, numaid;
21     char nameH[9] = "@@@@@@@";
22     gethostname(nameH, 8); nameH[8] = '\0'; <-----・現行プロセスを実行中のホスト名の最初の8文字、
23     procid = (int) getpid(); <-----現行プロセスのプロセスID、
24     syscall(__NR_getcpu, &coreid, &numaid, NULL); <-----現行プロセスを実行中のNUMAノードID、
25     fprintf(fp0, "hostname:%8s\n procid :%8d\n", nameH, procid); <---現行プロセスを実行中のCPUコアIDを、
26     fprintf(fp0, "numaid :%8u\n coreid :%8u\n", numaid, coreid); <---ファイル「fp0」に書き込む。
27     fclose(fp0);
28     MPI_Finalize(); <---・MPI関数の使用の終了を宣言する。
29     return 0;
30 }
```

・茶色の文字は実際のソースプログラムファイル中には実在しない文字。  
・白黒反転表示の文は、MPIプログラムに固有の文。  
・18行目までと、27行目以降は、原則として、今後示す全てのC言語のサンプルプログラム内に共通して存在する文であるため、今後は、例外的に異なる場合以外は省略する。

# 基本的なMPI関数の紹介

## 最初のMPIプログラムの実行例：world.c

ログインノードの画面表示

各プロセスの出力内容（2ノード4プロセスで実行した場合）

```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$ mpicc -o world world.c <-----・ C言語で書かれたMPIプログラムのコンパイルとリンクをした。
$ sbatch -p testq -t 00:01:00 exec.sh world <--・ ジョブを投入した。
Submitted batch job 93373 <-----・ ジョブ番号が「93373」のジョブとして受け付けられたとの表示
$
$ head -n 8 exec.sh_93373.stderr | tail -n 1
+ mpiexec -n 4 --machinefile exec.sh_93373.machinefile bash numa_bind_exec.sh 2 . world
$                                     ↑--・ 2ノード4プロセスで、ロードモジュールファイル「world」内の命令列を実行した。
$                                     その際、各プロセスを、それぞれ、別々のNUMAノードにバインドした。
$
$ ls -m *.txt <-----・ 各プロセスより、ファイル名が
world_0.txt, world_1.txt, world_2.txt, world_3.txt    「ロードモジュールファイル名_ランク番号.txt」
$                                                       の形式のファイルが出力されたことを確認した。
$ ↓--・ 出力されたファイルの内容を横並びに併記する。
$ paste -d @ world_[0-3].txt | sed 's/@/ /g'
rankW= 0 sizeW= 4    rankW= 1 sizeW= 4    rankW= 2 sizeW= 4    rankW= 3 sizeW= 4
hostname:cmptnd06    hostname:cmptnd06    hostname:cmptnd07    hostname:cmptnd07
procid  : 1023833    procid  : 1023834    procid  : 1091183    procid  : 1091182
numaid  :          0    numaid  :          1    numaid  :          0    numaid  :          1
coreid  :          88    coreid  :         112    coreid  :          11    coreid  :         114
$
```

# 基本的なMPI関数の紹介

## Point-to-Point Communicationの実行例：send.c

- このプログラムの目的

Blocking通信の動作を確認する。←プロセス間でデータ転送を行うための通信は、Blocking通信と、Non-blocking通信に大別される。

- 各プロセスでの処理の概要

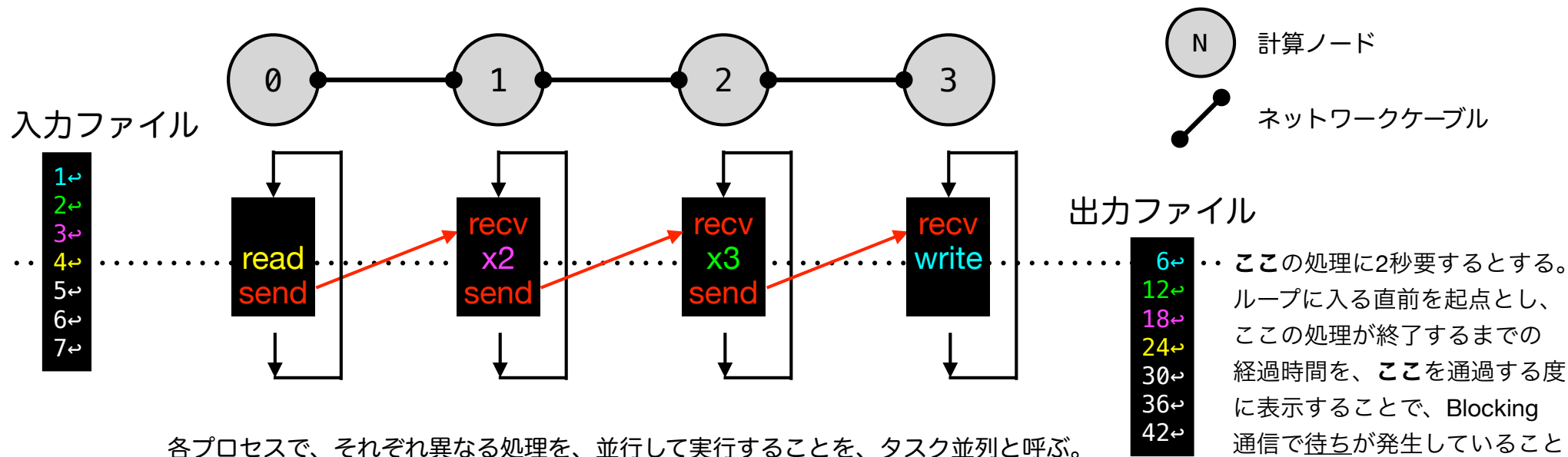
現行プロセスのランク番号により、処理内容が決まるように、4つのプロセスのそれぞれで、以下の処理が繰り返し行われる。←プログラムを書いておく。ロードモジュールファイルは1つ。

ランク番号0のプロセス：入力ファイルから値を1つ読み込み、それをランク番号1のプロセスに送る。

ランク番号1のプロセス：ランク番号0のプロセスから送られてきた値に、2を掛けて、その結果の値を、ランク番号2のプロセスに送る。

ランク番号2のプロセス：ランク番号1のプロセスから送られてきた値に、3を掛けて、その結果の値を、ランク番号3のプロセスに送る。

ランク番号3のプロセス：ランク番号2のプロセスから送られてきた値を、出力ファイルに書き出す。



# 基本的なMPI関数の紹介

## Point-to-Point Communicationの実行例：send.c

使用するMPI関数の説明（MPI3.1の規格書の第3章の記述の要約）

引数の文字の色は、引数の属性（**入力**、**出力**、**入出力**）を表す。

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

メモリ空間内のアドレス「buf」から始まる場所に格納されている「datatype」型の値を「count」個、コミュニケータ「comm」内のランク番号「dest」のプロセスに向けて、「tag」を付けて送信する。送信するデータを、メモリ空間内のアドレス「buf」から始まる場所より、読み出し終わるまで、待つことから、「Blocking Send」とも呼ばれる。なお、「tag」には、0以上「MPI\_TAG\_UB」以下の整数を指定する。「MPI\_TAG\_UB」は、32,767以上の整数。

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

「source」、「tag」、「comm」の値が指定したものと一致する通信を受信するまで待ち、受信したメッセージデータを、メモリ空間内のアドレス「buf」から始まる場所に格納する。「Blocking Receive」とも呼ばれる。

# 基本的なMPI関数の紹介

MPIデータ型とC言語のデータ型の対応の一覧 (MPI3.1の規格書の第3章より引用)

MPI datatype	C datatype
MPI_CHAR	char (treated as printable character)
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	signed long long int
MPI_LONG_LONG (as a synonym)	signed long long int
MPI_SIGNED_CHAR	signed char (treated as integral value)
MPI_UNSIGNED_CHAR	unsigned char (treated as integral value)
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wchar_t (dened in <stddef.h>)
MPI_C_BOOL	(treated as printable character) _Bool

# 基本的なMPI関数の紹介

MPIデータ型とC言語のデータ型の対応の一覧（MPI3.1の規格書の第3章より引用）

MPI datatype	C datatype
MPI_INT8_T	int8_t
MPI_INT16_T	int16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_UINT32_T	uint32_t
MPI_UINT64_T	uint64_t
MPI_C_COMPLEX	float _Complex
MPI_C_FLOAT_COMPLEX (as a synonym)	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_BYTE	
MPI_PACKED	
MPI_CHAR	

# 基本的なMPI関数の紹介

## Point-to-Point Communicationの実行例：send.c

ソースプログラムファイル「send.c」の内容

「rankW==0」のプロセスでは以下の命令列が実行される。

argv[0]: プログラム名

sizeW: 「MPI\_COMM\_WORLD」内のプロセス数

rankW: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

- ・ 茶色の文字は実際のソースプログラムファイル中には実在しない文字。
- ・ 白黒反転表示の文は、MPIプログラムに固有の文。
- ・ 19行目より前は「world.c」と全く同じ。

```
19  int nd,tag,nl,i;
20  int *sendbuf,*recvbuf;
21  time_t time_1,time_2;
22  nd=1;
23  sendbuf=(int *)malloc(sizeof(int)*nd);
24  recvbuf=(int *)malloc(sizeof(int)*nd);
25  sendbuf[0]=-1; <----- ・ 配列要素「sendbuf[0]」に、初期値「-1」を格納する。
26  recvbuf[0]=-1; <----- ・ 配列要素「recvbuf[0]」に、初期値「-1」を格納する。
27  tag=32767;
28  nl=7;
29  time_1=time(NULL); <-- ・ ここを経過時間計測の起点とする。
30  if(rankW==0){
31      FILE *fp1;      ↓-- ・ 入力ファイル「send_input.txt」を「fp1」とする。
32      fp1=fopen("send_input.txt","r");
33
34      for(i=0;i<nl;i++){ <-- ・ 34行目から38行目の処理を、「nl (==7)」回繰り返す。
35                          ループインデックス「i」の値は、「0」から始まり、ループの回転毎に1つずつ大きくなる。
36          fscanf(fp1,"%d",&(sendbuf[0])); <-- ・ 「fp1」より、値を1つ読み込み、配列要素「sendbuf[0]」に代入する。
37          sleep(2); <----- ・ 説明の都合上、前行の読み込み処理に約2秒掛かったことにする。
38
39          ↓-- ・ 主要処理終了時までの経過時間を記録する。      ↓-- ・ 経過時間(秒)と配列要素「sendbuf[0]」の内容を、
40          time_2=time(NULL);      ↓      「second」と「sendW」の値として、「fp0」に書き込む。
41          fprintf(fp0,"second=%2d sendW=%2d\n",(int)(time_2-time_1),sendbuf[0]);
42          MPI_Send(sendbuf,nd,MPI_INT,rankW+1,tag,MPI_COMM_WORLD);
43      }      ↑-- ・ 「MPI_INT(整数)」型の「nd (==1)」個の配列要素「sendbuf[0]」の内容を、
44      fclose(fp1);      ランク番号が現行プロセスより1つ大きいプロセスに向けて送信する。
45  }else if(0<rankW&&rankW<sizeW-1){      送信手続きが完了するまで待つ。
46      /* 中略 */
47  }else{
48      /* 中略 */
49  }
50  }
```



# 基本的なMPI関数の紹介

## Point-to-Point Communicationの実行例：send.c

ソースプログラムファイル「send.c」の内容

「0<rankW<sizeW-1(==3)」のプロセスでは以下の命令列が実行される。

argv[0]: プログラム名

sizeW: 「MPI\_COMM\_WORLD」内のプロセス数

rankW: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

```
19  int nd,tag,nl,i;
20  int *sendbuf,*recvbuf;
21  time_t time_1,time_2;
22  nd=1;
23  sendbuf=(int *)malloc(sizeof(int)*nd);
24  recvbuf=(int *)malloc(sizeof(int)*nd);
25  sendbuf[0]=-1; <----- ・配列要素「sendbuf[0]」に、初期値「-1」を格納する。
26  recvbuf[0]=-1; <----- ・配列要素「recvbuf[0]」に、初期値「-1」を格納する。
27  tag=32767;
28  nl=7;
29  time_1=time(NULL);
30  if(rankW==0){
    /* 中略 */
41  }else if(0<rankW&&rankW<sizeW-1){
42      for(i=0;i<nl;i++){ <----- ・43行目から48行目までの処理を、「nl (==7)」回繰り返す。
43          MPI_Recv(recvbuf,nd,MPI_INT,rankW-1,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
                ↑-- ・ランク番号が現行プロセスより1つ小さいプロセスから、「MPI_INT (整数)」型のデータが
                1つ送られて来るまで待ち、来たら配列要素「recvbuf[0]」に格納する。
44          sendbuf[0]=recvbuf[0]*(rankW+1); <-- ・配列要素「recvbuf[0]」の内容を、「rankW+1」倍して、
                配列要素「sendbuf[0]」に格納する。
45          sleep(2); <----- ・説明の都合上、前行の計算処理に約2秒掛かったことにする。
46          time_2=time(NULL); <----- ・主要処理終了時までの経過時間を記録する。
47          fprintf(fp0,"second=%2d sendW=%2d\n",(int)(time_2-time_1),sendbuf[0]);
                ↑-- ・経過時間 (秒) と配列要素「sendbuf[0]」の内容を、「second」と「sendW」の値として、「fp0」に書き込む。
48          MPI_Send(sendbuf,nd,MPI_INT,rankW+1,tag,MPI_COMM_WORLD);
49      }
                ↑-- ・「MPI_INT (整数)」型の「nd (==1)」個の配列要素「sendbuf[0]」の内容を、
50      }else{
                ランク番号が現行プロセスより1つ大きいプロセスに向けて送信する。送信手続きが完了するまで待つ。
    /* 中略 */
61 }
```

・茶色の文字は実際のソースプログラムファイル中には実在しない文字。  
・白黒反転表示の文は、MPIプログラムに固有の文。  
・19行目より前は「world.c」と全く同じ。

# 基本的なMPI関数の紹介

## Point-to-Point Communicationの実行例：send.c

ソースプログラムファイル「send.c」の内容

「rankW==sizeW-1(==3)」のプロセスでは以下の命令列が実行される。

argv[0]: プログラム名

sizeW: 「MPI\_COMM\_WORLD」内のプロセス数

rankW: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

- ・ 茶色の文字は実際のソースプログラムファイル中には実在しない文字。
- ・ 白黒反転表示の文は、MPIプログラムに固有の文。
- ・ 19行目より前は「world.c」と全く同じ。

```
19  int nd,tag,nl,i;
20  int *sendbuf,*recvbuf;
21  time_t time_1,time_2;
22  nd=1;
23  sendbuf=(int *)malloc(sizeof(int)*nd);
24  recvbuf=(int *)malloc(sizeof(int)*nd);
25  sendbuf[0]=-1; <----- ・ 配列要素「sendbuf[0]」に、初期値「-1」を格納する。
26  recvbuf[0]=-1; <----- ・ 配列要素「recvbuf[0]」に、初期値「-1」を格納する。
27  tag=32767;
28  nl=7;
29  time_1=time(NULL);
30  if(rankW==0){
    /* 中略 */
41  }else if(0<rankW&&rankW<sizeW-1){
    /* 中略 */
50  }else{
51  FILE *fp1;      ↓-- ・ 出力ファイル「send_output.txt」を「fp1」とする。
52  fp1=fopen("send_output.txt","w");
53  for(i=0;i<nl;i++){ <----- ・ 54行目から58行目の処理を、「nl (==7)」回繰り返す。
54  MPI_Recv(recvbuf,nd,MPI_INT,rankW-1,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
      ↑-- ・ ランク番号が現行プロセスより1つ小さいプロセスから、「MPI_INT (整数)」型のデータが
      1つ送られて来るまで待ち、来たら配列要素「recvbuf[0]」に格納する。
55  fprintf(fp1,"%2d\n",recvbuf[0]); <-- ・ 配列要素「recvbuf[0]」の値を、「fp1」に書き込む。
56  sleep(2); <----- ・ 説明の都合上、前行の書き出し処理に約2秒掛かったことにする。
57  time_2=time(NULL); <----- ・ 主要処理終了時までの経過時間を記録する。
58  fprintf(fp0,"second=%2d recvW=%2d\n",(int)(time_2-time_1),recvbuf[0]);
59  }
60  fclose(fp1);
61  }
```

↑-- ・ 経過時間（秒）と配列要素「recvbuf[0]」の内容を、「second」と「recvW」の値として、「fp0」に書き込む。

# 基本的なMPI関数の紹介

Point-to-Point Communicationの実行例：send.c

	sendbuf	recvbuf	sendbuf	recvbuf	sendbuf	recvbuf
rankW=0: Input rankW=1: Calculation rankW=2: Calculation rankW=3: Output	4	3	6	4	12	6
rankW=0: MPI_Send rankW=1: MPI_Send + MPI_Recv rankW=2: MPI_Send + MPI_Recv rankW=3: MPI_Recv	4	4	6	6	12	12
rankW=0: Input rankW=1: Calculation rankW=2: Calculation rankW=3: Output	5	4	8	6	18	12
rankW=0: MPI_Send rankW=1: MPI_Send + MPI_Recv rankW=2: MPI_Send + MPI_Recv rankW=3: MPI_Recv	5	5	8	8	18	18
rankW=0: Input rankW=1: Calculation rankW=2: Calculation rankW=3: Output	6	5	10	8	24	18
rankW=0: MPI_Send rankW=1: MPI_Send + MPI_Recv rankW=2: MPI_Send + MPI_Recv rankW=3: MPI_Recv	6	6	10	10	24	24
rankW=0: Input rankW=1: Calculation rankW=2: Calculation rankW=3: Output	7	6	12	10	30	24

時間進行

# 基本的なMPI関数の紹介

## Point-to-Point Communicationの実行例：send.c

ログインノードの画面表示

各プロセスの出力内容（2ノード4プロセスで実行した場合）

```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$ ls -m *.txt <--・ ランク番号が「0」のプロセスが読み込む、ファイル「send_input.txt」が存在していることを確認した。
send_input.txt
$
$ mpicc -o send send.c <-----・ C言語で書かれたMPIプログラムのコンパイルとリンクをした。
$ sbatch -p testq -t 00:01:00 exec.sh send <--・ ジョブを投入した。
Submitted batch job 93369 <-----・ ジョブ番号が「93369」のジョブとして受け付けられたとの表示
$
$ head -n 8 exec.sh_93369.stderr | tail -n 1
+ mpiexec -n 4 --machinefile exec.sh_93369.machinefile bash numa_bind_exec.sh 2 . send
$
$                                     ↑--・ 2ノード4プロセスで、ロードモジュールファイル「send」内の命令列を実行した。
$                                     その際、各プロセスを、それぞれ、別々のNUMAノードにバインドした。
$
$ ls -m *.txt <-----・ 各プロセスより、ファイル名が
send_0.txt, send_1.txt, send_2.txt, send_3.txt,      「ロードモジュールファイル名_ランク番号.txt」
send_input.txt, send_output.txt                     の形式のファイルが出力されていることと、
$                                                       ランク番号が「3」のプロセスより、ファイル
$                                                       「send_output.txt」が出力されていることを確認した。
$ ↓--・ 出力されたファイルの内容を横並びに併記する。
$ paste -d @ send_[0-3].txt | sed 's/@/ /g'
rankW= 0 sizeW= 4      rankW= 1 sizeW= 4      rankW= 2 sizeW= 4      rankW= 3 sizeW= 4
second= 2 sendW= 1 --> second= 4 sendW= 2 --> second= 6 sendW= 6 --> second= 8 recvW= 6
second= 4 sendW= 2      second= 6 sendW= 4      second= 8 sendW=12      second=10 recvW=12
second= 6 sendW= 3      second= 8 sendW= 6      second=10 sendW=18      second=12 recvW=18
second= 8 sendW= 4 --> second=10 sendW= 8 --> second=12 sendW=24 --> second=14 recvW=24
second=10 sendW= 5      second=12 sendW=10      second=14 sendW=30      second=16 recvW=30
second=12 sendW= 6      second=14 sendW=12      second=16 sendW=36      second=18 recvW=36
second=14 sendW= 7      second=16 sendW=14      second=18 sendW=42      second=20 recvW=42
$
```

# 基本的なMPI関数の紹介

## Point-to-Point Communicationの実行例：send.c

ログインノードの画面表示

各プロセスの出力内容（2ノード4プロセスで実行した場合）

```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$   ↓--・ 入力ファイル「send_input.txt」と、出力ファイル「send_output.txt」の内容を、横並びに併記する。
$ paste -d @ send_input.txt send_output.txt | sed 's/@/      /g'
1      6 <--
2     12 <--
3     18 <--
4     24 <--・ 入力した値の、6(=2×3)倍の値が出力された。
5     30 <--
6     36 <--
7     42 <--
$
```

# 基本的なMPI関数の紹介

## Point-to-Point Communicationの実行例：isend.c

- このプログラムの目的

Non-blocking通信の動作を確認する。

- 各プロセスでの処理の概要

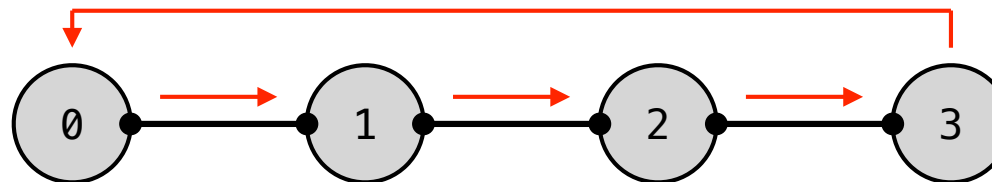
現行プロセスのランク番号により、処理内容が決まるように、4つのプロセスのそれぞれで、以下の処理が繰り返し行われる。← プログラムを書いておく。ロードモジュールファイルは1つ。

ランク番号0のプロセス：値を1つ、ランク番号1のプロセスに送る。値を1つ、ランク番号3のプロセスから受け取る。

ランク番号1のプロセス：値を1つ、ランク番号2のプロセスに送る。値を1つ、ランク番号0のプロセスから受け取る。

ランク番号2のプロセス：値を1つ、ランク番号3のプロセスに送る。値を1つ、ランク番号1のプロセスから受け取る。

ランク番号3のプロセス：値を1つ、ランク番号0のプロセスに送る。値を1つ、ランク番号2のプロセスから受け取る。



周期境界条件の問題を解く場合に、全ての計算ノードが、MPI\_SendとMPI\_Recvを順次実行するようにプログラムを記述すると、MPI\_Sendにおいて、送信完了待ちの状態のまま、その先に進まなくなってしまうことがある。この現象は「Dead Lock」と呼ばれる。



MPI\_Send及びMPI\_Recvのそれぞれにおける待つ動作を除いた関数、MPI\_Isend及びMPI\_Irecvを使用し、待ち合わせ場所を、MPI\_Waitall関数にて、別の場所に定めことにより「Dead Lock」を回避する。

# 基本的なMPI関数の紹介

## Point-to-Point Communicationの実行例：isend.c

使用するMPI関数の説明（MPI3.1の規格書の第3章の記述の要約）

引数の文字の色は、引数の属性（**入力**、**出力**、**入出力**）を表す。

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

メモリ空間内のアドレス「buf」から始まる場所に格納されている「datatype」型の値を「count」個、コミュニケーター「comm」内のランク番号「dest」のプロセスに向けて、「tag」を付して送信する手続きを開始する。送信するデータを、メモリ空間内のアドレス「buf」から始まる場所より読み出し終わるまでは、待たないことから、Nonblocking Sendとも呼ばれる。

変数「request」：関数「MPI\_Isend」や「MPI\_Irecv」による通信処理の個々を識別するためのハンドル。

→ bufから始まる送信用データを格納しているメモリ領域を書き換えて良くなるわけではない。他の命令を実行できるようになるだけ。

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

「source」、「tag」、「comm」の値が指定したものと一致する通信を受信したら、受信したメッセージデータを、メモリ空間内のアドレス「buf」から始まる場所に格納する準備をする。「source」、「tag」、「comm」の値が指定したものと一致する通信を受信するまでは、待たないことから、Nonblocking Receiveとも呼ばれる。

```
int MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status array_of_statuses[])
```

配列「array\_of\_requests」に格納された全ての「request」に関係する通信が終了するまで待つ。通信が終了したら、配列「array\_of\_requests」の各要素に、値「MPI\_REQUEST\_NULL」を代入する。

# 基本的なMPI関数の紹介

## Point-to-Point Communicationの実行例：isend.c

ソースプログラムファイル「isend.c」の内容

「rankW==0」のプロセスでは以下の命令列が実行される。

argv[0]: プログラム名

sizeW: 「MPI\_COMM\_WORLD」内のプロセス数

rankW: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

```
19  int nd;
20  int *sendbuf,*recvbuf;
21  nd=1;
22  sendbuf=(int *)malloc(sizeof(int)*nd);
23  recvbuf=(int *)malloc(sizeof(int)*nd);
24  sendbuf[0]=rankW*10; <----- ・ 配列要素「sendbuf[0]」に、変数「rankW」の値を10倍した値を格納する。
25  recvbuf[0]=-1; <----- ・ 配列要素「recvbuf[0]」に、値「-1」を格納する。
26  MPI_Request request[2];
27  int tag=32767;
28  if(rankW==0){
29      MPI_Irecv(recvbuf,nd,MPI_INT,sizeW-1,tag,MPI_COMM_WORLD,&(request[0]));
30      MPI_Isend(sendbuf,nd,MPI_INT,rankW+1,tag,MPI_COMM_WORLD,&(request[1]));
31  }else if(0<rankW&&rankW<sizeW-1){
    /* 中略 */
34  }else{
    /* 中略 */
37  }
38  MPI_Waitall(2,request,MPI_STATUSES_IGNORE); <--- ・ 上記2つの通信が完了するまで待つ。
39  fprintf(fp0," recvW=%02d sendW=%02d\n",recvbuf[0],sendbuf[0]);
    ↑-- ・ 配列要素「recvbuf[0]」と「sendbuf[0]」の内容を、「recvW」と「sendW」の値として、「fp0」に書き込む。
```

- ・ 茶色の文字は実際のソースプログラムファイル中には実在しない文字。
- ・ 白黒反転表示の文は、MPIプログラムに固有の文。
- ・ 19行目より前は「world.c」と全く同じ。



# 基本的なMPI関数の紹介

## Point-to-Point Communicationの実行例：isend.c

ソースプログラムファイル「isend.c」の内容

「0<rankW<sizeW-1(==3)」のプロセスでは以下の命令列が実行される。

argv[0]: プログラム名

sizeW: 「MPI\_COMM\_WORLD」内のプロセス数

rankW: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

```
19  int nd;
20  int *sendbuf,*recvbuf;
21  nd=1;
22  sendbuf=(int *)malloc(sizeof(int)*nd);
23  recvbuf=(int *)malloc(sizeof(int)*nd);
24  sendbuf[0]=rankW*10; <----- ・配列要素「sendbuf[0]」に、変数「rankW」の値を10倍した値を格納する。
25  recvbuf[0]=-1; <----- ・配列要素「recvbuf[0]」に、値「-1」を格納する。
26  MPI_Request request[2];
27  int tag=32767;
28  if(rankW==0){
/* 中略 */
31  }else if(0<rankW&&rankW<sizeW-1){

                                ↓-- ・ランク番号が「rankW-1」のプロセスから、MPI_INT型のデータが1つ
                                ↓   送られてきたら、それが配列要素「recvbuf[0]」に格納されるように準備する。
32  MPI_Irecv(recvbuf,nd,MPI_INT,rankW-1,tag,MPI_COMM_WORLD,&(request[0]));
33  MPI_Isend(sendbuf,nd,MPI_INT,rankW+1,tag,MPI_COMM_WORLD,&(request[1]));
34  }else{
                                ↑-- ・配列要素「sendbuf[0]」の内容を、ランク番号が1つ
                                    大きいプロセスに向けて、送信する手続きを開始する。

/* 中略 */
37  }
38  MPI_Waitall(2,request,MPI_STATUSES_IGNORE); <-- ・上記2つの通信が完了するまで待つ。
39  fprintf(fp0," recvW=%02d sendW=%02d\n",recvbuf[0],sendbuf[0]);
                                ↑-- ・配列要素「recvbuf[0]」と「sendbuf[0]」の内容を、「recvW」と「sendW」の値として、「fp0」に書き込む。
```

・茶色の文字は実際のソースプログラムファイル中には実在しない文字。  
・白黒反転表示の文は、MPIプログラムに固有の文。  
・19行目より前は「world.c」と全く同じ。

# 基本的なMPI関数の紹介

## Point-to-Point Communicationの実行例：isend.c

ソースプログラムファイル「isend.c」の内容

「rankW==sizeW-1(==3)」のプロセスでは以下の命令列が実行される。

argv[0]: プログラム名

sizeW: 「MPI\_COMM\_WORLD」内のプロセス数

rankW: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

```
19  int nd;
20  int *sendbuf,*recvbuf;
21  nd=1;
22  sendbuf=(int *)malloc(sizeof(int)*nd);
23  recvbuf=(int *)malloc(sizeof(int)*nd);
24  sendbuf[0]=rankW*10; <----- ・配列要素「sendbuf[0]」に、変数「rankW」の値を10倍した値を格納する。
25  recvbuf[0]=-1; <----- ・配列要素「recvbuf[0]」に、値「-1」を格納する。
26  MPI_Request request[2];
27  int tag=32767;
28  if(rankW==0){
    /* 中略 */
31  }else if(0<rankW&&rankW<sizeW-1){
    /* 中略 */
34  }else{
        ↓-- ・ランク番号が「rankW-1」のプロセスから、MPI_INT型のデータが1つ
        ↓   送られてきたら、それが配列要素「recvbuf[0]」に格納されるように準備する。
35  MPI_Irecv(recvbuf,nd,MPI_INT,rankW-1,tag,MPI_COMM_WORLD,&(request[0]));
36  MPI_Isend(sendbuf,nd,MPI_INT,0,tag,MPI_COMM_WORLD,&(request[1]));
        ↑-- ・配列要素「sendbuf[0]」の内容を、ランク番号が「0」の
            プロセスに向けて、送信する手続きを開始する。
37  }
38  MPI_Waitall(2,request,MPI_STATUSES_IGNORE);
39  fprintf(fp0," recvW=%02d sendW=%02d\n",recvbuf[0],sendbuf[0]);
        ↑-- ・配列要素「recvbuf[0]」と「sendbuf[0]」の内容を、「recvW」と「sendW」の値として、「fp0」に書き込む。
```

・茶色の文字は実際のソースプログラムファイル中には実在しない文字。  
・白黒反転表示の文は、MPIプログラムに固有の文。  
・19行目より前は「world.c」と全く同じ。

# 基本的なMPI関数の紹介

## Point-to-Point Communicationの実行例：isend.c

ログインノードの画面表示

各プロセスの出力内容（2ノード4プロセスで実行した場合）

```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$ mpicc -o isend isend.c <-----・ C言語で書かれたMPIプログラムのコンパイルとリンクをした。
$ sbatch -p testq -t 00:01:00 exec.sh isend <--・ ジョブを投入した。
Submitted batch job 93362 <-----・ ジョブ番号が「93362」のジョブとして受け付けられたとの表示
$
$ head -n 8 exec.sh_93362.stderr | tail -n 1
+ mpiexec -n 4 --machinefile exec.sh_93362.machinefile bash numa_bind_exec.sh 2 . isend
$
$          ↑--・ 2ノード4プロセスで、ロードモジュールファイル「isend」内の命令列を実行した。
$          その際、各プロセスを、それぞれ、別々のNUMAノードにバインドした。
$
$ ls -m *.txt <-----・ 各プロセスより、ファイル名が
isend_0.txt, isend_1.txt, isend_2.txt, isend_3.txt    「ロードモジュールファイル名_ランク番号.txt」
$                                                    の形式のファイルが出力されたことを確認した。
$ ↓--・ 出力されたファイルの内容を横並びに併記する。
$ paste -d @ isend_[0-3].txt | sed 's/@/ /g'
rankW= 0 sizeW= 4      rankW= 1 sizeW= 4      rankW= 2 sizeW= 4      rankW= 3 sizeW= 4
recvW=30 sendW=00      recvW=00 sendW=10      recvW=10 sendW=20      recvW=20 sendW=30
$
```

# 基本的なMPI関数の紹介

## Virtual Topologyの使用例：cart.c

問題意識：

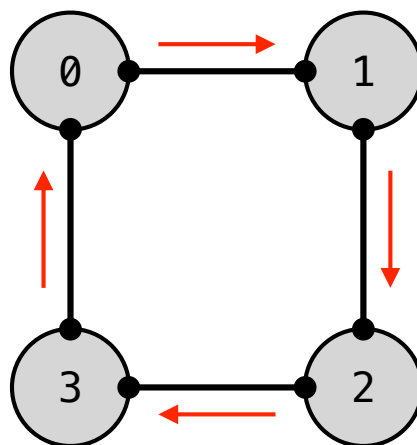
先の「isend.c」の例では、ソースプログラムを、現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号「rankW」の値により、三通りに場合分けして記述していた。

現行プロセスが、計算領域の端部を取り扱っているかを、一々判定するのは面倒くさい。

解決方法：

計算領域が、周期境界であることを、予め教えておけば良い。

トポロジー（接続関係・隣接関係）を指定することにより、「rankW」の値による場合分けの必要をなくすることができる。



# 基本的なMPI関数の紹介

## Virtual Topologyの使用例：cart.c

使用するMPI関数の説明（MPI3.1の規格書の第7章の記述の要約）

引数の文字の色は、引数の属性（**入力**、**出力**、**入出力**）を表す。

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],  
                    const int periods[], int reorder, MPI_Comm *comm_cart)
```

コミュニケータ「comm\_old」に、Cartesian topology 情報を付加した、新たなコミュニケータ「comm\_cart」を作成する。

「ndims」は、Cartesian grid の次元数、「dims[]」には、各次元のプロセス数、「periods[]」には、各次元が周期境界 (true) か否 (false) か、「reorder」は、ランク番号の再割り当てを行うか (true) 否か (false) の情報を与える。

「reorder」が「false」の場合には、各プロセスの新たなコミュニケータ「comm\_cart」内でのランク番号は、元のコミュニケータ「comm\_old」内での物と同一。

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

Cartesian topology 情報が付加されたコミュニケータ「comm」の、「dims[direction]」で表される座標軸での座標が、「現行ランク番号-disp」のプロセスからデータを受け取り、「現行ランク番号+disp」のプロセスに向けてデータを送る時の、送信元のランク番号「rank\_source」と、送信先のランク番号「rank\_dest」を取得する。

Cartesian topology 情報を付加した、新たなコミュニケータ「comm\_cart」を使用して「MPI\_Isend」を実行したとしても、全ページの図の「rankW=3」のプロセスにおいて、「rankW=0」のプロセスにデータを送るのに際し、送信先プロセスを以下のように「rankW+1」と指定することができるわけではない。

```
MPI_Isend(sendbuf,nd,MPI_INT,rankW+1,tag,comm_cart,&(request[1]));
```

送信先プロセスのランク番号「rank\_dest」を予め取得しておき、そこに向かって送信する必要がある（以下）。

```
MPI_Isend(sendbuf,nd,MPI_INT,rank_dest,tag,comm_cart,&(request[1]));
```

「reorder = false」の場合には、「comm\_cart」ではなく「comm\_old」を使用しても結果は同じ。

# 基本的なMPI関数の紹介

## Virtual Topologyの使用例：cart.c

ソースプログラムファイル「cart.c」の内容

各プロセスが、それぞれ独立に、以下の命令列を実行する。

**argv[0]**: プログラム名

**sizeW**: 「MPI\_COMM\_WORLD」内のプロセス数

**rankW**: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

```
19 int ndims,reorder,idim,shft,rank_src,rank_dst,nd;
20 int *dims,*prds,*sendbuf,*recvbuf;
21 ndims=1;
22 dims=(int *)malloc(sizeof(int)*ndims);
23 prds=(int *)malloc(sizeof(int)*ndims);
24 dims[0]=sizeW;
25 prds[0]=true;
26 reorder=false;
27 MPI_Comm mpi_comm_cart;
28 MPI_Cart_create(MPI_COMM_WORLD,ndims,dims,prds,reorder,&mpi_comm_cart);

29 idim=0;
30 shft=1;
31 MPI_Cart_shift(mpi_comm_cart,idim,shft,&rank_src,&rank_dst);
32 nd=1;
33 sendbuf=(int *)malloc(sizeof(int)*nd);
34 recvbuf=(int *)malloc(sizeof(int)*nd);
35 sendbuf[0]=rankW*10;
36 recvbuf[0]=-1;

37 MPI_Request request[2];
38 int tag=32767;
39 MPI_Irecv(recvbuf,nd,MPI_INT,rank_src,tag,MPI_COMM_WORLD,&(request[0]));
40 MPI_Isend(sendbuf,nd,MPI_INT,rank_dst,tag,MPI_COMM_WORLD,&(request[1]));

41 MPI_Waitall(2,request,MPI_STATUSES_IGNORE);
42 fprintf(fp0," recvW=%02d sendW=%02d\n",recvbuf[0],sendbuf[0]);
```

↓--・既存のコミュニケータ「MPI\_COMM\_WORLD」に、  
↓ 1次元周期境界条件の「Cartesian topology」情報を付加した、  
↓ 新たなコミュニケータ「mpi\_comm\_cart」を作成する。

↓--・「Cartesian topology」の1次元目の座標が「-1」のプロセスからデータを受け取り、  
↓ 「+1」のプロセスに向けてデータを送るとき、送信元のランク番号と、送り先のランク番号を取得する。

↓--・ランク番号が1つ小さいプロセスから、MPI\_INT型のデータが1つ送られてきたら、  
↓ それが配列要素「recvbuf[0]」に格納されるように準備する。

↑--・配列要素「sendbuf[0]」の内容を、ランク番号が1つ大きいプロセスに向けて、  
送信する手続きを開始する。

↑--・配列要素「recvbuf[0]」と「sendbuf[0]」の内容を、「recvW」と「sendW」の値として、「fp0」に書き込む。

・茶色の文字は実際のソースプログラムファイル中には実在しない文字。  
・白黒反転表示の文は、MPIプログラムに固有の文。  
・19行目より前は「world.c」と全く同じ。

# 基本的なMPI関数の紹介

## Virtual Topologyの使用例：cart.c

ログインノードの画面表示

各プロセスの出力内容（2ノード4プロセスで実行した場合）

```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$ mpicc -o cart cart.c <----- ・ C言語で書かれたMPIプログラムのコンパイルとリンクをした。
$ sbatch -p testq -t 00:01:00 exec.sh cart <-- ・ ジョブを投入した。
Submitted batch job 93355 <----- ・ ジョブ番号が「93355」のジョブとして受け付けられたとの表示
$
$ head -n 8 exec.sh_93355.stderr | tail -n 1
+ mpiexec -n 4 --machinefile exec.sh_93355.machinefile bash numa_bind_exec.sh 2 . cart
$                               ↑-- ・ 2ノード4プロセスで、ロードモジュールファイル「cart」内の命令列を実行した。
$                               その際、各プロセスを、それぞれ、別々のNUMAノードにバインドした。
$
$ ls -m *.txt <----- ・ 各プロセスより、ファイル名が
cart_0.txt, cart_1.txt, cart_2.txt, cart_3.txt      「ロードモジュールファイル名_ランク番号.txt」
$                                                    の形式のファイルが出力されたことを確認した。
$ ↓-- ・ 出力されたファイルの内容を横並びに併記する。
$ paste -d @ cart_[0-3].txt | sed 's/@/ /g'
rankW= 0 sizeW= 4      rankW= 1 sizeW= 4      rankW= 2 sizeW= 4      rankW= 3 sizeW= 4
recvW=30 sendW=00      recvW=00 sendW=10      recvW=10 sendW=20      recvW=20 sendW=30
$
$ for i in `seq 0 3`; do diff -s cart_${i}.txt ../isend/isend_${i}.txt; done
Files cart_0.txt and ../isend/isend_0.txt are identical
Files cart_1.txt and ../isend/isend_1.txt are identical
Files cart_2.txt and ../isend/isend_2.txt are identical
Files cart_3.txt and ../isend/isend_3.txt are identical
$
```

# 基本的なMPI関数の紹介

## スレッド並列領域内でのMPI関数の実行例：thread.c

問題意識：

先の「isend.c」の例では、関数「MPI\_Isend」及び「MPI\_Irecv」と、関数「MPI\_Waitall」との間に、計算を実行するための命令（演算命令）が記述されていたとしても、通信と計算を同時に行われること（重ね合わせ）は保証されてはいない。関数「MPI\_Waitall」が実行された時に、プロセス間の通信が行われるように実装されていることもあるが、それでも規格外ではない。

解決方法：

スレッド並列化された区間内において、MPI関数を実行する。

Blocking通信において待たされるのは、そのBlocking通信を実行中のスレッドのみ。

基本的には、他のスレッドにて、他のMPI関数を実行することもできる。  
（できない計算機システムも存在するが、規格外ではない）



# 基本的なMPI関数の紹介

## スレッド並列領域内でのMPI関数の実行例：thread.c

使用するMPI関数の説明（MPI3.1の規格書の第12章の記述の要約）

引数の文字の色は、引数の属性（**入力**、**出力**、**入出力**）を表す。

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

MPI関数の使用の開始を宣言するだけでなく、スレッド並列領域内からもMPI関数を使用できるような環境も整える。

関数「MPI\_Init」に替えて使用する関数

引数には、関数「MPI\_Init」の引数（main関数の引数若しくは「NULL,NULL」）に加えて、スレッドサポートレベルを示す変数「required」と「provided」の2つが必要となる。

スレッドサポートレベル（変数「required」と「provided」）が取り得る値と、その大小関係は以下の通り。

MPI\_THREAD\_SINGLE < MPI\_THREAD\_FUNNELED < MPI\_THREAD\_SERIALIZED < MPI\_THREAD\_MULTIPLE

MPI\_THREAD\_SINGLE: Only one thread will execute. <-- 関数「MPI\_Init」を使用した場合と同等

MPI\_THREAD\_FUNNELED: The process may be multi-threaded, but the application must ensure that only the main thread (the thread that called MPI\_INIT or MPI\_INIT\_THREAD) makes MPI calls.

MPI\_THREAD\_SERIALIZED: The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are “serialized”).

MPI\_THREAD\_MULTIPLE: Multiple threads may call MPI, with no restrictions.

変数「required」で指定する値は、あくまで、プログラマーが”希望”するスレッドサポートレベルの値。

関数「MPI\_Init\_thread」を実行した結果、実際に利用可能となったスレッドサポートレベルの値が、変数「provided」に入る。

「required <= provided」とならないこともあり得る。

# 基本的なMPI関数の紹介

## スレッド並列領域内でのMPI関数の実行例：thread.c

事前（その計算機システムの使用を決定する前など）に、計算機システムのスレッドサポートレベルの最高値を確認しておく必要がある。

計算ノードを対話型操作した場合の画面表示（少数の計算ノードを短時間使用する場合に、対話型操作が可能な計算機システムも多い）

```
・ クリーム色の文字は実際の画面上には実在しない文字
$ cat thread_support_level.c <----- ・ スレッドサポートレベル確認用のMPIプログラム（C言語）の内容を表示させた。
#include<stdio.h>
#include"mpi.h"
int main(){
    int provided;
    MPI_Init_thread(NULL,NULL,MPI_THREAD_MULTIPLE,&provided);
    switch(provided){
        case MPI_THREAD_SINGLE:
            printf("MPI_THREAD_SINGLE\n");
            break;
        case MPI_THREAD_FUNNELED:
            printf("MPI_THREAD_FUNNELED\n");
            break;
        case MPI_THREAD_SERIALIZED:
            printf("MPI_THREAD_SERIALIZED\n");
            break;
        default:
            printf("MPI_THREAD_MULTIPLE\n");
    }
    MPI_Finalize();
    return 0;
}
$
$ mpicc -o thread_support_level thread_support_level.c <--- ・ C言語で書かれたMPIプログラムのコンパイルとリンクをした。
$
$ mpiexec -n 1 ./thread_support_level <--- ・ 「ロードモジュールファイル」内の命令列を、1プロセスで素朴に実行した。
MPI_THREAD_MULTIPLE <----- ・ 当該計算機システムで使用可能なスレッドサポートレベルの最高値が表示される。
$                               （「MPI_THREAD_MULTIPLE」までサポートされていることが多い）
$
MPI_THREAD_SERIALIZED <--- ・ 参考：別の計算機システムでの結果。「MPI_THREAD_MULTIPLE」ではないことがたまにある。
                               （同じ計算機システムでも使用するMPIライブラリによってスレッドサポートレベルが異なる場合もある）
```

# 基本的なMPI関数の紹介

## スレッド並列領域内でのMPI関数の実行例：thread.c

ソースプログラムファイル「thread.c」の内容

各プロセスが、それぞれ独立に、以下の命令列を実行する。

**argv[0]**: プログラム名

**sizeW**: 「MPI\_COMM\_WORLD」内のプロセス数

**rankW**: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

- ・ 茶色の文字は実際のソースプログラムファイル中には実在しない文字。
- ・ 白黒反転表示の文は、MPIプログラムに固有の文。
- ・ 36行目までで「cart.c」と異なるのは、10行目と13行目のみ。

```
$ diff ../cart/cart.c thread.c
```

```
10c10
```

```
<  int sizeW,rankW;
```

```
---
```

```
>  int sizeW,rankW,provided;
```

```
13c13
```

```
<  MPI_Init(NULL,NULL);
```

```
---
```

```
>  MPI_Init_thread(NULL,NULL,MPI_THREAD_MULTIPLE,&provided);
```

```
37d36
```

```
29  idim=0;          ↓--・「Cartesian topology」の1次元目の座標が「-1」のプロセスからデータを受け取り、  
30  shft=1;         ↓      「+1」のプロセスに向けてデータを送るときの、送信元のランク番号と、送り先のランク番号を取得する。
```

```
31  MPI_Cart_shift(mpi_comm_cart,idim,shft,&rank_src,&rank_dst);
```

```
32  nd=1;
```

```
33  sendbuf=(int *)malloc(sizeof(int)*nd);
```

```
34  recvbuf=(int *)malloc(sizeof(int)*nd);
```

```
35  sendbuf[0]=rankW*10; <-----・配列要素「sendbuf[0]」に、変数「rankW」の値を10倍した値を格納する。
```

```
36  recvbuf[0]=-1; <-----・配列要素「recvbuf[0]」に、値「-1」を格納する。
```

```
37  int tag=32767;
```

```
38  #pragma omp parallel default(shared) num_threads(2) <--・39行目から44行目を、2スレッドで実行する。
```

```
39  {
```

```
40      int tn=omp_get_thread_num(); ↓--・ランク番号が1つ小さいプロセスから、MPI_INT型のデータが1つ送られて来るまで待ち、  
41      if(tn==0){                  ↓      来たら配列要素「recvbuf[0]」に格納する。
```

```
42      MPI_Recv(recvbuf,nd,MPI_INT,rank_src,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
```

```
43  }else if(tn==1){
```

```
44      MPI_Send(sendbuf,nd,MPI_INT,rank_dst,tag,MPI_COMM_WORLD);
```

```
45  }
```

↑--・配列要素「sendbuf[0]」の内容を、  
ランク番号が1つ大きいプロセスに向けて送信する。送信手続きが完了するまで待つ。

```
46  }
```

```
47  fprintf(fp0," recvW=%02d sendW=%02d\n",recvbuf[0],sendbuf[0]);
```

↑--・配列要素「recvbuf[0]」と「sendbuf[0]」の内容を、「recvW」と「sendW」の値として、「fp0」に書き込む。

# 基本的なMPI関数の紹介

## スレッド並列領域内でのMPI関数の実行例：thread.c

ログインノードの画面表示

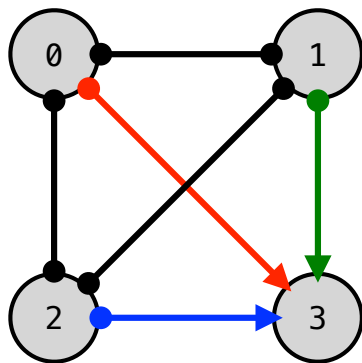
各プロセスの出力内容（2ノード4プロセスで実行した場合）

```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$ mpicc -fopenmp -o thread thread.c <----- ・ C言語で書かれたMPIプログラムのコンパイルとリンクをした。
$      ↑----- ・ OpenMPによるスレッド並列化を可能にするためのオプション。
$
$ sbatch -p testq -t 00:01:00 exec.sh thread <-- ・ ジョブを投入した。
Submitted batch job 93371 <----- ・ ジョブ番号が「93371」のジョブとして受け付けられたとの表示
$
$ head -n 8 exec.sh_93371.stderr | tail -n 1
+ mpiexec -n 4 --machinefile exec.sh_93371.machinefile bash numa_bind_exec.sh 2 . thread
$      ↑-- ・ 2ノード4プロセスで、ロードモジュールファイル「thread」内の命令列を実行した。
$      その際、各プロセスを、それぞれ、別々のNUMAノードにバインドした。
$
$ ls -m *.txt <----- ・ 各プロセスより、ファイル名が
thread_0.txt, thread_1.txt, thread_2.txt, thread_3.txt  「ロードモジュールファイル名_ランク番号.txt」
$      の形式のファイルが出力されたことを確認した。
$      ↓-- ・ 出力されたファイルの内容を横並びに併記する。
$ paste -d @ thread_[0-3].txt | sed 's/@/ /g'
rankW= 0 sizeW= 4      rankW= 1 sizeW= 4      rankW= 2 sizeW= 4      rankW= 3 sizeW= 4
recvW=30 sendW=00      recvW=00 sendW=10      recvW=10 sendW=20      recvW=20 sendW=30
$
$ for i in `seq 0 3`; do diff -s thread_${i}.txt ../cart/cart_${i}.txt; done
Files thread_0.txt and ../cart/cart_0.txt are identical
Files thread_1.txt and ../cart/cart_1.txt are identical
Files thread_2.txt and ../cart/cart_2.txt are identical
Files thread_3.txt and ../cart/cart_3.txt are identical
$
```

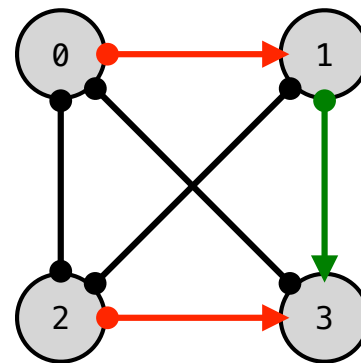
# 基本的なMPI関数の紹介

## Collective Communicationとは

- 複数のプロセス間でのデータ転送を一括して行うためのMPI関数を、Collective Communication関数と呼ぶ。
- 関係する全プロセスが、それぞれ、同じMPI関数を呼び出す必要がある。
- 大型電子計算機では、各メーカーが、ネットワークの構造を考慮し、最適化したMPI関数を提供していることが多い。
  - Collective Communication関数で書くことができる部分は、Point-to-Point Communication関数ではなく、Collective Communication関数で書いておいた方が、プログラムの実行時間が短くなることが多い。



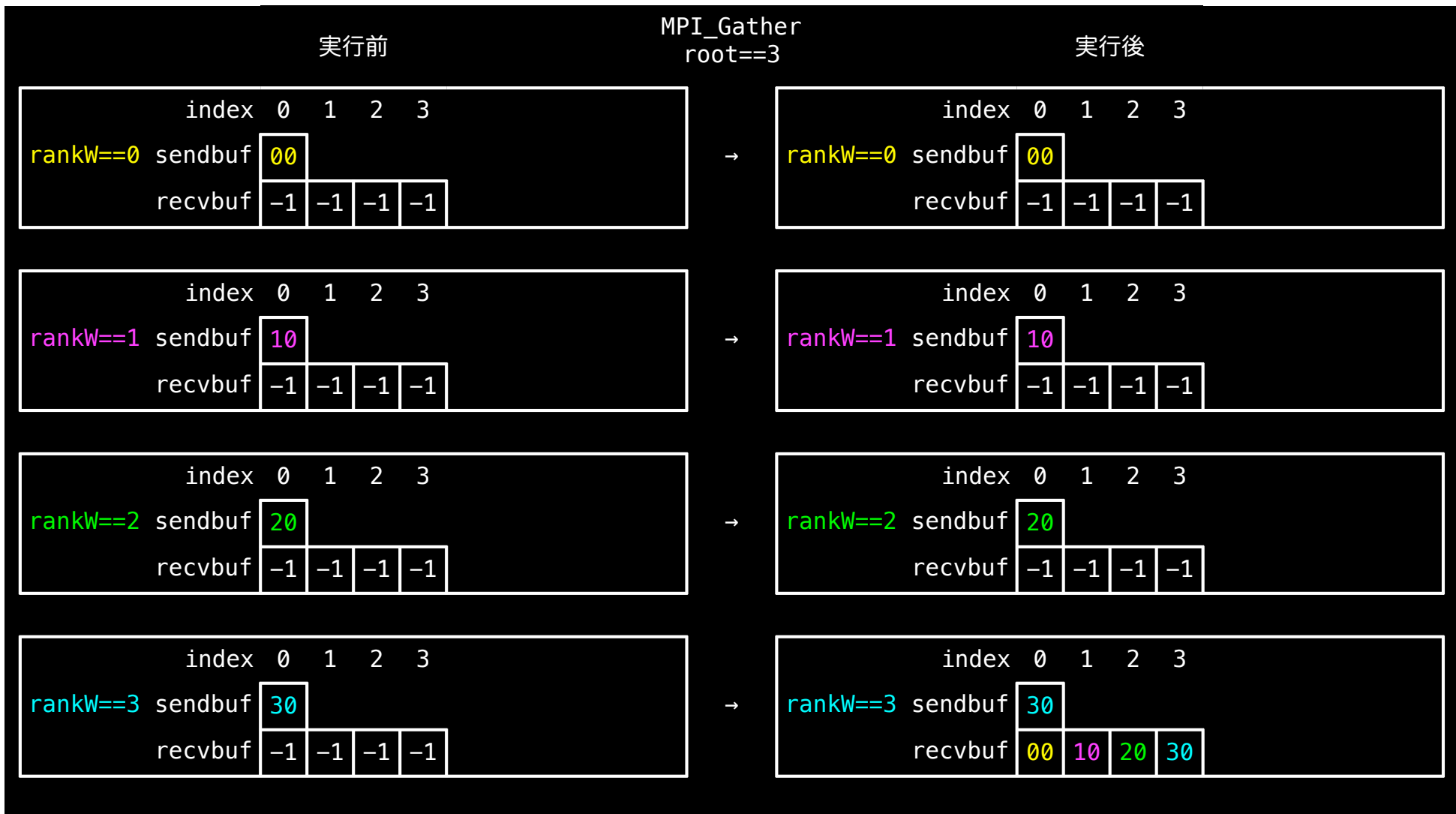
ネットワークの構造を考慮せず、  
単純に総和を求めた場合の、  
時間軸上での処理（=通信+加算）回数  
①+②+③=3回



ネットワークの構造を考慮し、  
処理を並列化しつつ総和を求めた場合の  
時間軸上での処理（=通信+加算）回数  
①+②=2回

# 基本的なMPI関数の紹介

Collective Communication (MPI\_Gather) の実行例：gather.c



# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Gather) の実行例：gather.c

使用するMPI関数の説明（MPI3.1の規格書の第5章の記述の要約）

引数の文字の色は、引数の属性（**入力**、**出力**、**入出力**）を表す。

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

コミュニケータ「comm」内に存在する全てのプロセスがこの関数を実行することにより、「comm」内に存在する全てのプロセスの、メモリ空間内のアドレス「sendbuf」から始まる「sendtype」型の値「sendcount」個が、ランク番号が「root」のプロセスの、メモリ空間内のアドレス「recvbuf」から始まる場所に、ランク番号順に格納される。ここで、変数「recvcount」は、各プロセスから受け取る「recvtype」型のデータの個数を表す。

# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Gather) の実行例：gather.c

ソースプログラムファイル「gather.c」の内容

各プロセスが、それぞれ独立に、以下の命令列を実行する。

**argv[0]**: プログラム名

**sizeW**: 「MPI\_COMM\_WORLD」内のプロセス数

**rankW**: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

```
19  int nd,rootW,i;
20  int *sendbuf,*recvbuf;
21  nd=1;
22  sendbuf=(int *)malloc(sizeof(int)*nd);
23  recvbuf=(int *)malloc(sizeof(int)*nd*sizeW);
24  sendbuf[0]=rankW*10; <----- ・配列要素「sendbuf[0]」に、変数「rankW」の値を10倍した値を格納する。
25  for(i=0;i<sizeW;i++){
26      recvbuf[i]=-1; <----- ・配列「recvbuf」の全要素に、値「-1」を格納する。
27  }
28  fprintf(fp0,"\n");
29  for(i=0;i<nd;i++){
30      fprintf(fp0," index=%2d  send=%02d\n",i,sendbuf[i]);
31  }
32  rootW=sizeW-1;
33  MPI_Gather(sendbuf,nd,MPI_INT,recvbuf,nd,MPI_INT,rootW,MPI_COMM_WORLD);
34  fprintf(fp0,"\n");
35  for(i=0;i<nd*sizeW;i++){
36      fprintf(fp0," index=%2d  recv=%02d\n",i,recvbuf[i]);
37  }
38  char nameF1[256];
39  FILE *fp1;
40  if(rankW==rootW){ ===== ・以下は、後の実行例における比較対象データを用意するための手続き。 ==
41      sprintf(nameF1,"%s_%d.dat",argv[0],rankW);
42      fp1=fopen(nameF1,"wb"); <----- ・ファイル名が「argv[0]_rootW.dat」の形式のファイルを、
                                         ランク番号が「rootW」のプロセスからのバイナリ形式データの
                                         出力先ファイル「fp1」とする。
43      fwrite(recvbuf,sizeof(int)*nd,sizeW,fp1);
44      fclose(fp1);
45  }
```

↓-- ・配列要素番号「i」と配列要素「sendbuf[i]」の内容を、  
「index」と「send」の値として、「fp0」に書き込む。

↓-- ・各プロセスの配列要素「sendbuf[0]」の内容を、ランク番号が「rootW(==3)」の  
プロセスの配列「recvbuf」にランク番号順に格納する。

↑----- ・配列要素番号「i」と配列要素「recvbuf[i]」の値を、  
「index」と「recvW」の値として、「fp0」に書き込む。

↑----- ・「fp1」の先頭に、配列「recvbuf」の内容を書き込む。

・ 茶色の文字は実際のソースプログラムファイル中には実在しない文字。  
・ 白黒反転表示の文は、MPIプログラムに固有の文。  
・ 19行目より前は「world.c」と全く同じ。



# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Gather) の実行例：gather.c

ログインノードの画面表示

各プロセスの出力内容 (2ノード4プロセスで実行した場合)

```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$ mpicc -o gather gather.c <----- ・ C言語で書かれたMPIプログラムのコンパイルとリンクをした。
$ sbatch -p testq -t 00:01:00 exec.sh gather <-- ・ ジョブを投入した。
Submitted batch job 93356 <----- ・ ジョブ番号が「93356」のジョブとして受け付けられたとの表示
$
$ head -n 8 exec.sh_93356.stderr | tail -n 1
+ mpiexec -n 4 --machinefile exec.sh_93356.machinefile bash numa_bind_exec.sh 2 . gather
$                                     ↑-- ・ 2ノード4プロセスで、ロードモジュールファイル「gather」内の命令列を実行した。
$                                     その際、各プロセスを、それぞれ、別々のNUMAノードにバインドした。
$
$ ls -m *.txt; ls -m *.dat <----- ・ 各プロセスより、ファイル名が
gather_0.txt, gather_1.txt, gather_2.txt, gather_3.txt      「ロードモジュールファイル名_ランク番号.txt」と
gather_3.dat          「ロードモジュールファイル名_ランク番号.dat」の
$                      形式のファイルが出力されたことを確認した。
$ ↓-- ・ 出力されたファイルの内容を横並びに併記する。
$ paste -d @ gather_[0-3].txt | sed 's/@/ /g'
rankW= 0 sizeW= 4      rankW= 1 sizeW= 4      rankW= 2 sizeW= 4      rankW= 3 sizeW= 4
-----
index= 0 send=00      index= 0 send=10      index= 0 send=20      index= 0 send=30
----- MPI_Gather
index= 0 recv=-1      index= 0 recv=-1      index= 0 recv=-1      index= 0 recv=00
index= 1 recv=-1      index= 1 recv=-1      index= 1 recv=-1      index= 1 recv=10
index= 2 recv=-1      index= 2 recv=-1      index= 2 recv=-1      index= 2 recv=20
index= 3 recv=-1      index= 3 recv=-1      index= 3 recv=-1      index= 3 recv=30
$
$ ↓- ・ 出力されたバイナリ形式ファイルの内容を、10進数で表示する。
$ od -A d -t dI gather_3.dat
00000000      0      10      20      30
00000016
$
```

# 基本的なMPI関数の紹介

## MPI\_IOの使用例：mpiio.c

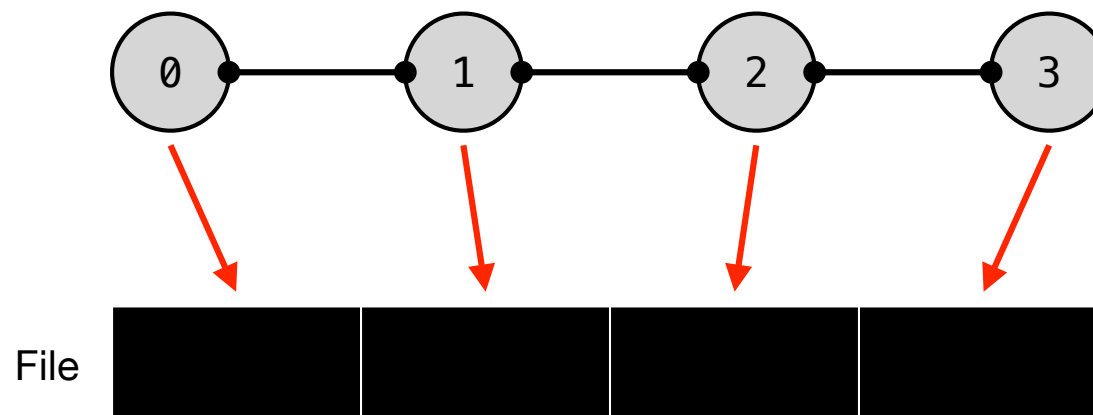
問題意識：

各プロセスが保持するデータを、1つのプロセスに集めてから、1つのファイルに書き出すのでは、

- ・ 全プロセス分のデータを集めるための通信時間
  - ・ 集めた全プロセス分のデータを格納するためのメモリ領域
  - ・ 1つのプロセスが、全プロセス分のデータをファイルに書き出すのに要する時間
- が大きくなる懸念がある。

解決方法：

各プロセスが、それぞれ独立に、1つのファイルの別々のデータ領域に書き出すようにすれば良い（並列 I/O）。  
（メタデータアクセスがボトルネックとなり、実行時間の短縮には繋がらない場合もある）



# 基本的なMPI関数の紹介

## MPI\_IOの使用例：mpiio.c

使用するMPI関数の説明（MPI3.1の規格書の第13章の記述の要約）

引数の文字の色は、引数の属性（**入力**、**出力**、**入出力**）を表す。

```
int MPI_File_open(MPI_Comm comm, const char *filename, int amode, MPI_Info info, MPI_File *fh)
```

コミュニケータ「comm」内の全プロセスにおいて、「filename」で指定された名前のファイルを、「amode」で指定されたアクセスモードでオープンし、ファイルハンドル「fh」を設定する。

「amode」に指定することができるアクセスモードは、以下の通り。

- MPI\_MODE\_RDONLY: read only,
- MPI\_MODE\_RDWR: reading and writing,
- MPI\_MODE\_WRONLY: write only,
- MPI\_MODE\_CREATE: create the file if it does not exist,
- MPI\_MODE\_EXCL: error if creating file that already exists,
- MPI\_MODE\_DELETE\_ON\_CLOSE: delete file on close,
- MPI\_MODE\_UNIQUE\_OPEN: file will not be concurrently opened elsewhere,
- MPI\_MODE\_SEQUENTIAL: file will only be accessed sequentially,
- MPI\_MODE\_APPEND: set initial position of all file pointers to end of file.

# 基本的なMPI関数の紹介

## MPI\_IOの使用例：mpiio.c

使用するMPI関数の説明（MPI3.1の規格書の第13章の記述の要約）

引数の文字の色は、引数の属性（**入力**、**出力**、**入出力**）を表す。

```
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,  
                      int count, MPI_Datatype datatype, MPI_Status *status)
```

ファイルハンドル「fh」で指定されたファイルの、先頭から「offset」で指定されたバイト数の位置に、メモリ空間内のアドレス「buf」から始まる「datatype」型の値「count」個を、書き込む。

```
int MPI_File_close(MPI_File *fh)
```

ファイルハンドル「fh」で指定されたファイルをクローズする。

# 基本的なMPI関数の紹介

## MPI\_IOの使用例：mpiio.c

ソースプログラムファイル「mpiio.c」の内容

各プロセスが、それぞれ独立に、以下の命令列を実行する。

**argv[0]**: プログラム名

**sizeW**: 「MPI\_COMM\_WORLD」内のプロセス数

**rankW**: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

```
19  int nd;  
20  int *sendbuf;  
21  nd=1;  
22  sendbuf=(int *)malloc(sizeof(int)*nd);  
23  sendbuf[0]=rankW*10; <----- ・配列要素「sendbuf[0]」に、変数「rankW」の値を10倍した値を格納する。  
24  char nameF1[256];  
25  sprintf(nameF1,"%s_m.dat",argv[0]);  
26  MPI_File fp1;  
27  MPI_File_open(MPI_COMM_WORLD,nameF1,MPI_MODE_WRONLY|MPI_MODE_CREATE,MPI_INFO_NULL,&fp1);  
28  MPI_File_write_at(fp1,rankW*sizeof(int)*nd,sendbuf,nd,MPI_INT,MPI_STATUS_IGNORE);  
29  MPI_File_close(&fp1); <--- ・ファイル「fp1」への書き込みが行える状態を解消する。
```

↓-- ・ファイル名が「argv[0]\_m.dat」の形式の1つのファイルを、  
↓ 全プロセスからのバイナリ形式データの出力先「fp1」とする。  
↑-- ・各プロセスがそれぞれ、配列要素「sendbuf[0]」の内容を、  
ファイル「fp1」内での位置を指定して、書き込む。

・ 茶色の文字は実際のソースプログラムファイル中には実在しない文字。  
・ 白黒反転表示の文は、MPIプログラムに固有の関数等を含む文。  
・ 19行目より前は「world.c」と全く同じ。

# 基本的なMPI関数の紹介

## MPI\_IOの使用例：mpiio.c

ログインノードの画面表示

各プロセスの出力内容（2ノード4プロセスで実行した場合）

```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$ mpicc -o mpiio mpiio.c <-----・ C言語で書かれたMPIプログラムのコンパイルとリンクをした。
$ sbatch -p testq -t 00:01:00 exec.sh mpiio <--・ ジョブを投入した。
Submitted batch job 93363 <-----・ ジョブ番号が「93363」のジョブとして受け付けられたとの表示
$
$ head -n 8 exec.sh_93363.stderr | tail -n 1
+ mpiexec -n 4 --machinefile exec.sh_93363.machinefile bash numa_bind_exec.sh 2 . mpiio
$                               ↑--・ 2ノード4プロセスで、ロードモジュールファイル「mpiio」内の命令列を実行した。
$                               その際、各プロセスを、それぞれ、別々のNUMAノードにバインドした。
$
$ ls -m *.txt; ls -m *.dat <-----・ 各プロセスより、ファイル名が
mpiio_0.txt, mpiio_1.txt, mpiio_2.txt, mpiio_3.txt   「ロードモジュールファイル名_ランク番号.txt」と
mpiio_m.dat                                           「ロードモジュールファイル名_m.dat」の
$                                                     形式のファイルが出力されたことを確認した。
$ ↓-・ 出力されたファイルの内容を横並びに併記する。
$ paste -d @ mpiio_[0-3].txt | sed 's/@/ /g'
rankW= 0 sizeW= 4      rankW= 1 sizeW= 4      rankW= 2 sizeW= 4      rankW= 3 sizeW= 4
$
$ ↓-・ 出力されたバイナリ形式データの内容を、10進数で表示する。
$ od -A d -t dI mpiio_m.dat
0000000      0      10      20      30
0000016
$
$ ↓-・ 出力されたバイナリ形式データの内容が、「gather_3.dat」と一致することを確認する。
$ diff -s mpiio_m.dat ../gather/gather_3.dat
Files mpiio_m.dat and ../gather/gather_3.dat are identical
$
```

# 基本的なMPI関数の紹介

## Intra-communicatorの使用例：intracomm.c

問題意識：

先の実行例「gather.c」では、関数「MPI\_Gather」を実行すると、コミュニケータ「MPI\_COMM\_WORLD」内の全プロセス間でのデータ集約が行われたが、全部ではなく一部のプロセスの間だけでのデータ集約を行いたいときはどうすれば良いか？

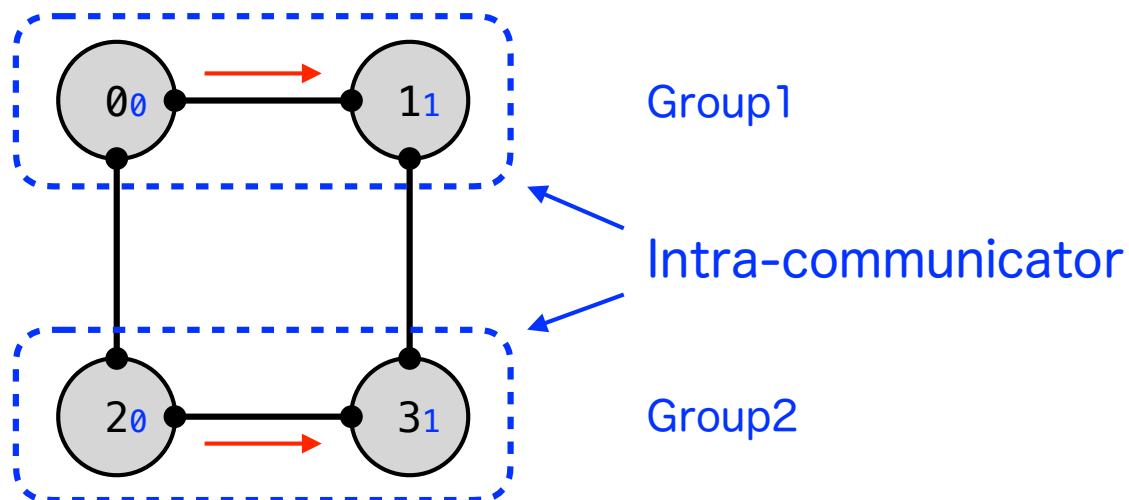
解決方法：

それ用のコミュニケータを作成すれば良い。

- ・ 「MPI\_Gather」により集約を行いたいデータを保持するプロセスのみで構成されるグループを作成する。
- ・ 作成されたグループ内でのMPI通信用のコミュニケータ（Intra-communicator）を作成する。

作成されたコミュニケータは、「MPI\_Gather」だけではなく、他のMPI関数においても使用することができる。

各プロセスには、「MPI\_COMM\_WORLD」内のものとは別に、作成されたコミュニケータ内でのランク番号が割り振られる。



# 基本的なMPI関数の紹介

## Intra-communicatorの使用例：intracomm.c

使用するMPI関数の説明（MPI3.1の規格書の第6章の記述の要約）

引数の文字の色は、引数の属性（**入力**、**出力**、**入出力**）を表す。

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

コミュニケータ「comm」で通信可能な全プロセスを1つのグループとし、そのグループ情報にアクセスするためのハンドラ「group」を設定する。

```
int MPI_Group_incl(MPI_Group group, int n, const int ranks[], MPI_Group *newgroup)
```

ハンドラ「group」で指定されたグループに属するプロセスのうち、そのグループ内でのランク番号が、大きさ「n」の配列「ranks」に格納されているもののみで、1つの新たなグループ（サブグループ）を作成し、そのサブグループ情報にアクセスするためのハンドラ「newgroup」を設定する。

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
```

コミュニケータ「comm」で通信可能な全プロセスを1つのグループとしたとき、そのサブグループをハンドラ「group」で指定し、そのサブグループ内での通信を可能とするコミュニケータ「newcomm」を作成する。



# 基本的なMPI関数の紹介

## Intra-communicatorの使用例：intracomm.c

ソースプログラムファイル「intracomm.c」の内容  
各プロセスが、それぞれ独立に、以下の命令列を実行する。

**argv[0]**: プログラム名  
**sizeW**: 「MPI\_COMM\_WORLD」内のプロセス数  
**rankW**: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

```
19 int sizeS,rankS,nd,rootS,i;
20 int *sendbuf,*recvbuf;
21 nd=1;
22 sendbuf=(int *)malloc(sizeof(int)*nd);
23 recvbuf=(int *)malloc(sizeof(int)*nd*sizeW);
24 sendbuf[0]=rankW*10; <----- ・配列要素「sendbuf[0]」に、変数「rankW」の値を10倍した値を格納する。
25 for(i=0;i<sizeW;i++){
26     recvbuf[i]=-1; <----- ・配列「recvbuf」の全要素に、値「-1」を格納する。
27 }
28 fprintf(fp0,"\n");
29 for(i=0;i<nd;i++){
30     fprintf(fp0," index=%2d  send=%02d\n",i,sendbuf[i]);
31 }
32 MPI_Group mpi_group_world,group1,group2;
33 MPI_Comm mpi_comm_group1,mpi_comm_group2;
34 MPI_Comm_group(MPI_COMM_WORLD,&mpi_group_world); <-- ・コミュニケータ「MPI_COMM_WORLD」内の全プロセスを
35 int ranks1[2]={0,1};                               1つのグループとし、そのグループ情報にアクセスするための
36 int ranks2[2]={2,3};                               ハンドラ「mpi_group_world」を作成する。

                ↓-- ・グループ「mpi_group_world」に属する全プロセスのうち、そのランク番号が、長さ「2」の配列「ranks1」に
                ↓   格納されているもの（ランク番号「0」と「1」のプロセス）のみで、新たなグループ「group1」を作成する。
37 MPI_Group_incl(mpi_group_world,2,ranks1,&group1);
38 MPI_Group_incl(mpi_group_world,2,ranks2,&group2);
                ↑-- ・グループ「mpi_group_world」に属する全プロセスのうち、そのランク番号が、長さ「2」の配列「ranks2」に
                ↑   格納されているもの（ランク番号「2」と「3」のプロセス）のみで、新たなグループ「group2」を作成する。

                ↓-- ・グループ「group1」内での通信用のコミュニケータ「mpi_comm_group1」を作成する。
39 MPI_Comm_create(MPI_COMM_WORLD,group1,&mpi_comm_group1);
40 MPI_Comm_create(MPI_COMM_WORLD,group2,&mpi_comm_group2);
                ↑-- ・グループ「group2」内での通信用のコミュニケータ「mpi_comm_group2」を作成する。
```

・ 茶色の文字は実際のソースプログラムファイル中には実在しない文字。  
・ 白黒反転表示の文は、MPIプログラムに固有の関数等を含む文。  
・ 19行目より前は「world.c」と全く同じ。

# 基本的なMPI関数の紹介

## Intra-communicatorの使用例：intracomm.c

ソースプログラムファイル「intracomm.c」の内容

「rankW==0 || rankW==1」のプロセスでは以下の命令列が実行される。

argv[0]: プログラム名

sizeW: 「MPI\_COMM\_WORLD」内のプロセス数

rankW: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

・ 前ページ (65ページ目) からの続き

・ 茶色の文字は実際のソースプログラムファイル中には実在しない文字。  
・ 白黒反転表示の文は、MPIプログラムに固有の関数等を含む文。  
・ 19行目より前は「world.c」と全く同じ。

```
41  rootS=1;           ↓--・現行プロセスが、グループ「group1」に属していれば、43行目から47行目を実行する。
42  if(mpi_comm_group1!=MPI_COMM_NULL){
    MPI_Comm_size(mpi_comm_group1,&sizeS);
    MPI_Comm_rank(mpi_comm_group1,&rankS);
    ↑--・グループ「group1」内のプロセス数を取得し、変数「sizeS」に格納する。
    ↑--・現行プロセスの、グループ「group1」内でのランク番号を取得し、変数「rankS」に格納する。
45  fprintf(fp0,"\n");
46  fprintf(fp0," rankS=%2d sizeS=%2d\n",rankS,sizeS);
    ↑--・ファイル「fp0」に、変数「rankS」と「sizeS」の値を書き込む。
    ↓--・グループ「group1」内において、各プロセスの配列要素「sendbuf[0]」内容を、グループ「group1」内での
    ↓ ランク番号が「rootS(=1)」のプロセスの配列「recvbuf」に、「group1」内でのランク番号順に格納する。
47  MPI_Gather(sendbuf,nd,MPI_INT,recvbuf,nd,MPI_INT,rootS,mpi_comm_group1);
48  }
    ↓--・現行プロセスが、グループ「group2」に属していれば、50行目から54行目を実行する。
49  if(mpi_comm_group2!=MPI_COMM_NULL){
/* 中略 */
55  }
56  fprintf(fp0,"\n");
57  for(i=0;i<sizeW;i++){
58    fprintf(fp0," index=%2d  recv=%02d\n",i,recvbuf[i]);
59  }
    ↑-----・配列要素番号「i」と配列要素「recvbuf[i]」の値を、
    「index」と「recvW」の値として、「fp0」に書き込む。
```

# 基本的なMPI関数の紹介

## Intra-communicatorの使用例：intracomm.c

ソースプログラムファイル「intracomm.c」の内容

「rankW==2 || rankW==3」のプロセスでは以下の命令列が実行される。

argv[0]: プログラム名  
sizeW: 「MPI\_COMM\_WORLD」内のプロセス数  
rankW: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

・ 前々ページ (65ページ目) からの続き

・ 茶色の文字は実際のソースプログラムファイル中には実在しない文字。  
・ 白黒反転表示の文は、MPIプログラムに固有の関数等を含む文。  
・ 19行目より前は「world.c」と全く同じ。

```
41  rootS=1;                ↓--・現行プロセスが、グループ「group1」に属していれば、43行目から47行目を実行する。
42  if(mpi_comm_group1!=MPI_COMM_NULL){
/* 中略 */
48  }
                                ↓--・現行プロセスが、グループ「group2」に属していれば、50行目から54行目を実行する。
49  if(mpi_comm_group2!=MPI_COMM_NULL){
                                ↓--・グループ「group2」内のプロセス数を取得し、変数「sizeS」に格納する。
50      MPI_Comm_size(mpi_comm_group2,&sizeS);
51      MPI_Comm_rank(mpi_comm_group2,&rankS);
                                ↑--・現行プロセスの、グループ「group2」内でのランク番号を取得し、変数「rankS」に格納する。
52      fprintf(fp0,"\n");
53      fprintf(fp0," rankS=%2d sizeS=%2d\n",rankS,sizeS);
                                ↑--・ファイル「fp0」に、変数「rankS」と「sizeS」の値を書き込む。
                                ↓--・group2 内において、各プロセスの配列要素 sendbuf[0] 内容を、
                                ↓ group2 内でのランク番号が 1 のプロセスの配列 recvbuf に、group2 内でのランク番号順に格納する。
54  MPI_Gather(sendbuf,nd,MPI_INT,recvbuf,nd,MPI_INT,rootS,mpi_comm_group2);
55  }
56  fprintf(fp0,"\n");
57  for(i=0;i<sizeW;i++){
58      fprintf(fp0," index=%2d  recv=%02d\n",i,recvbuf[i]);
59  }
                                ↑-----・配列要素番号「i」と配列要素「recvbuf[i]」の値を、
                                「index」と「recvW」の値として、「fp0」に書き込む。
```

# 基本的なMPI関数の紹介

## Intra-communicatorの使用例：intracomm.c

ログインノードの画面表示

各プロセスの出力内容（2ノード4プロセスで実行した場合）

```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$ mpicc -o intracomm intracomm.c <----- ・ C言語で書かれたMPIプログラムのコンパイルとリンクをした。
$ sbatch -p testq -t 00:01:00 exec.sh intracomm <-- ・ ジョブを投入した。
Submitted batch job 93361 <----- ・ ジョブ番号が「93361」のジョブとして受け付けられたとの表示
$
$ head -n 8 exec.sh_93361.stderr | tail -n 1
+ mpiexec -n 4 --machinefile exec.sh_93361.machinefile bash numa_bind_exec.sh 2 . intracomm
$                               ↑-- ・ 2ノード4プロセスで、ロードモジュールファイル「intracomm」内の命令列を実行した。
$                               その際、各プロセスを、それぞれ、別々のNUMAノードにバインドした。
$
$ ls -m *.txt <----- ・ 各プロセスより、ファイル名が
intracomm_0.txt, intracomm_1.txt, intracomm_2.txt, intracomm_3.txt 「ロードモジュールファイル名_ランク番号.txt」
$                               の形式のファイルが出力されたことを確認した。
$ ↓-- ・ 出力されたファイルの内容を横並びに併記する。
$ paste -d @ intracomm_[0-3].txt | sed 's/@/ /g'
rankW= 0 sizeW= 4      rankW= 1 sizeW= 4      rankW= 2 sizeW= 4      rankW= 3 sizeW= 4
index= 0 send=00      index= 0 send=10      index= 0 send=20      index= 0 send=30
rankS= 0 sizeS= 2      rankS= 1 sizeS= 2      rankS= 0 sizeS= 2      rankS= 1 sizeS= 2
----- MPI_Gather -----
index= 0 recv=-1      index= 0 recv=00      index= 0 recv=20      index= 0 recv=30
index= 1 recv=-1      index= 1 recv=10      index= 1 recv=30      index= 1 recv=-1
index= 2 recv=-1      index= 2 recv=-1      index= 2 recv=-1      index= 2 recv=-1
index= 3 recv=-1      index= 3 recv=-1      index= 3 recv=-1      index= 3 recv=-1
$                               group1                               group2
```

# 基本的なMPI関数の紹介

Inter-communicatorの使用例：intercomm.c

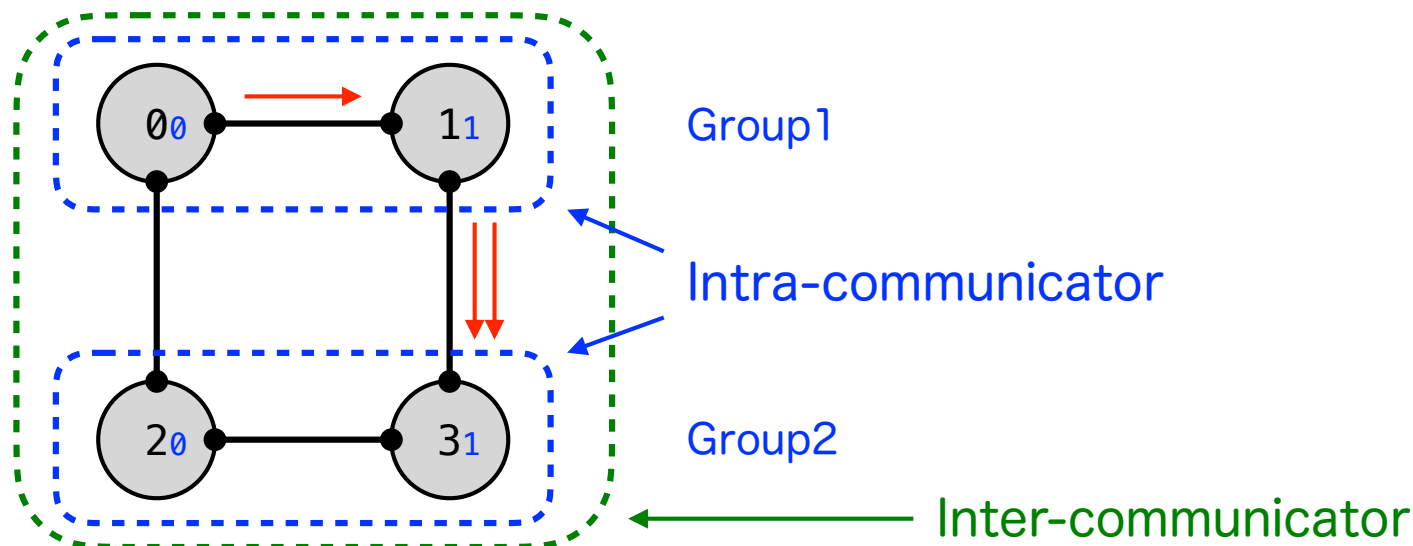
問題意識：

先の実行例「intracomm.c」では、関数「MPI\_Gather」を実行すると、指定したコミュニケータ内の全プロセス間でのデータ集約の結果が、そのコミュニケータ内の1つのプロセスに格納されたが、格納先をデータ集約が行われたコミュニケータの外のプロセスにしたいときはどうすれば良いか？

解決方法：

「MPI\_Gather」により集約を行いたいデータを保持するプロセスのみで構成されるグループ（下の絵では「Group1」）と、集約されたデータの格納先のプロセスを含む別のグループ（下の絵では「Group2」）との間のMPI通信用のコミュニケータ（Inter-communicator）を作成する。

作成されたコミュニケータは、「MPI\_Gather」だけではなく、他のMPI関数においても使用することができる。



# 基本的なMPI関数の紹介

## Inter-communicatorの使用例：intercomm.c

使用するMPI関数の説明（MPI3.1の規格書の第6章の記述の要約）

引数の文字の色は、引数の属性（**入力**、**出力**、**入出力**）を表す。

```
int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader, MPI_Comm peer_comm,  
                        int remote_leader, int tag, MPI_Comm *newintercomm)
```

ある1つのコミュニケータ「local\_comm」に属するプロセスと、別のコミュニケータ「remote\_comm」に属するプロセスとの間の通信を可能にするためのコミュニケータ「newintercomm」を作成する。

- ・ コミュニケータ「local\_comm」に属する少なくとも1つのプロセス「local\_leader」は、コミュニケータ「remote\_comm」に属する少なくとも1つのプロセス「remote\_leader」と、共通のコミュニケータ「peer\_comm」内において、通信可能でなくてはならない。
- ・ プロセス「local\_leader」は、コミュニケータ「local\_comm」内でのランク番号にて指定する。
- ・ プロセス「remote\_leader」は、共通のコミュニケータ「peer\_comm」内でのランク番号にて指定する。

# 基本的なMPI関数の紹介

## Inter-communicatorの使用例：intercomm.c

ソースプログラムファイル「intercomm.c」の内容

各プロセスが、それぞれ独立に、以下の命令列を実行する。

**argv[0]**: プログラム名

**sizeW**: 「MPI\_COMM\_WORLD」内のプロセス数

**rankW**: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

・40行目までは、先の実行例「intracomm.c」と全く同じ。

```
19 int sizeS,rankS,nd,rootS,i;
20 int *sendbuf,*recvbuf;
21 nd=1;
22 sendbuf=(int *)malloc(sizeof(int)*nd);
23 recvbuf=(int *)malloc(sizeof(int)*nd*sizeW);
24 sendbuf[0]=rankW*10; <-----・配列要素「sendbuf[0]」に、変数「rankW」の値を10倍した値を格納する。
25 for(i=0;i<sizeW;i++){
26     recvbuf[i]=-1; <-----・配列「recvbuf」の全要素に、値「-1」を格納する。
27 }
28 fprintf(fp0,"\n");
29 for(i=0;i<nd;i++){
30     fprintf(fp0," index=%2d  send=%02d\n",i,sendbuf[i]);
31 }
32 MPI_Group mpi_group_world,group1,group2;
33 MPI_Comm mpi_comm_group1,mpi_comm_group2;
34 MPI_Comm_group(MPI_COMM_WORLD,&mpi_group_world); <---・コミュニケータ「MPI_COMM_WORLD」内の全プロセスを
35 int ranks1[2]={0,1};                               1つのグループとし、そのグループ情報にアクセスするための
36 int ranks2[2]={2,3};                               ハンドラ「mpi_group_world」を作成する。
```

↓--・グループ「mpi\_group\_world」に属する全プロセスのうち、そのランク番号が、長さ「2」の配列「ranks1」に格納されているもの（ランク番号「0」と「1」のプロセス）のみで、新たなグループ「group1」を作成する。

```
37 MPI_Group_incl(mpi_group_world,2,ranks1,&group1);
38 MPI_Group_incl(mpi_group_world,2,ranks2,&group2);
```

↑--・グループ「mpi\_group\_world」に属する全プロセスのうち、そのランク番号が、長さ「2」の配列「ranks2」に格納されているもの（ランク番号「2」と「3」のプロセス）のみで、新たなグループ「group2」を作成する。

↓--・グループ「group1」内での通信用のコミュニケータ「mpi\_comm\_group1」を作成する。

```
39 MPI_Comm_create(MPI_COMM_WORLD,group1,&mpi_comm_group1);
40 MPI_Comm_create(MPI_COMM_WORLD,group2,&mpi_comm_group2);
```

↑--・グループ「group2」内での通信用のコミュニケータ「mpi\_comm\_group2」を作成する。

・茶色の文字は実際のソースプログラムファイル中には実在しない文字。  
・白黒反転表示の文は、MPIプログラムに固有の関数等を含む文。  
・19行目より前は「world.c」と全く同じ。



# 基本的なMPI関数の紹介

## Inter-communicatorの使用例：intercomm.c

ソースプログラムファイル「intercomm.c」の内容

「rankW==0 || rankW==1」のプロセスでは以下の命令列が実行される。

argv[0]: プログラム名

sizeW: 「MPI\_COMM\_WORLD」内のプロセス数

rankW: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

・前ページ（71ページ目）からの続き

・茶色の文字は実際のソースプログラムファイル中には実在しない文字。  
・白黒反転表示の文は、MPIプログラムに固有の関数等を含む文。  
・19行目より前は「world.c」と全く同じ。

```
41 MPI_Comm mpi_comm_inter;
42 int tag=32767;
43 rootS=1;
44 if(mpi_comm_group1!=MPI_COMM_NULL){
    ↓--・現行プロセスが、グループ「group1」に属していれば、45行目から50行目を実行する。
    ↓--・グループ「group1」内のプロセス数を取得し、変数「sizeS」に格納する。
45 MPI_Comm_size(mpi_comm_group1,&sizeS);
46 MPI_Comm_rank(mpi_comm_group1,&rankS);
    ↑--・現行プロセスの、グループ「group1」内でのランク番号を取得し、変数「rankS」に格納する。
47 fprintf(fp0,"\n");
48 fprintf(fp0," rankS=%2d sizeS=%2d\n",rankS,sizeS);
    ↑--・ファイル「fp0」に、変数「rankS」と「sizeS」の値を書き込む。
    ↓--・グループ「group1」内のランク番号「0」のプロセスを「local_leader」とし、
    ↓ コミュニケータ「MPI_COMM_WORLD」内のランク番号「ranks2[0] (==2)」のプロセスを「remote_leader」とする、
    ↓ インターコミュニケータ「mpi_comm_inter」を作成する。
49 MPI_Intercomm_create(mpi_comm_group1,0,MPI_COMM_WORLD,ranks2[0],tag,&mpi_comm_inter);
    ↓--・グループ「group1」内の各プロセスの配列要素「sendbuf[0]」の内容を、
    ↓ グループ「group2」内のランク番号「rootS (==1)」のプロセスの配列「recvbuf」に、
    ↓ グループ「group1」内でのランク番号「rankS」の値の順に格納する。
50 MPI_Gather(sendbuf,nd,MPI_INT,recvbuf,nd,MPI_INT,rootS,mpi_comm_inter);
51 }
    ↓--・現行プロセスが、グループ「group2」に属していれば、53行目から62行目を実行する。
52 if(mpi_comm_group2!=MPI_COMM_NULL){
/* 中略 */
63 }
64 fprintf(fp0,"\n");
65 for(i=0;i<sizeW;i++){
    ↓-----・配列要素番号「i」と配列要素「recvbuf[i]」の値を、
66     fprintf(fp0," index=%2d  recv=%02d\n",i,recvbuf[i]);    「index」と「recvW」の値として、「fp0」に書き込む。
67 }
```



# 基本的なMPI関数の紹介

## Inter-communicatorの使用例：intercomm.c

ソースプログラムファイル「intercomm.c」の内容

「rankW==2 || rankW==3」のプロセスでは以下の命令列が実行される。

argv[0]: プログラム名

sizeW: 「MPI\_COMM\_WORLD」内のプロセス数

rankW: 現行プロセスの「MPI\_COMM\_WORLD」内のランク番号

・ 前々ページ (71ページ目) からの続き

・ 茶色の文字は実際のソースプログラムファイル中には実在しない文字。  
・ 白黒反転表示の文は、MPIプログラムに固有の関数等を含む文。  
・ 19行目より前は「world.c」と全く同じ。

```
41 MPI_Comm mpi_comm_inter;
42 int tag=32767;
43 rootS=1;
44 if(mpi_comm_group1!=MPI_COMM_NULL){
/* 中略 */
51 }
52 if(mpi_comm_group2!=MPI_COMM_NULL){
53     MPI_Comm_size(mpi_comm_group2,&sizeS); <-- グループ「group2」内のプロセス数を取得し、変数「sizeS」に格納する。
54     MPI_Comm_rank(mpi_comm_group2,&rankS); <-- 現行プロセスの、グループ「group2」内でのランク番号を取得し、
                                         変数「rankS」に格納する。

55     fprintf(fp0,"\n");
56     fprintf(fp0," rankS=%2d sizeS=%2d\n",rankS,sizeS);
                                         ↑-- ファイル「fp0」に、変数「rankS」と「sizeS」の値を書き込む。

    ↓-- グループ「group2」内のランク番号「0」のプロセスを「local_leader」とし、
    ↓ コミュニケータ「MPI_COMM_WORLD」内のランク番号「ranks1[0] (==0)」のプロセスを「remote_leader」とする、
    ↓ インターコミュニケータ「mpi_comm_inter」を作成する。
57 MPI_Intercomm_create(mpi_comm_group2,0,MPI_COMM_WORLD,ranks1[0],tag,&mpi_comm_inter);
58 if(rankS==rootS){
                                         ↓-- 「rankS==rootS」のプロセスでは、
                                         ↓ 「MPI_ROOT」と記述する。
59     MPI_Gather(sendbuf,nd,MPI_INT,recvbuf,nd,MPI_INT,MPI_ROOT,mpi_comm_inter);
60 }else{
    ↑-- グループ「group1」内のMPI_Gatherの結果を、
    ↓ グループ「group2」内でのランク番号が「rootS(==1)」のプロセスの配列「recvbuf」に格納する。
61     MPI_Gather(sendbuf,nd,MPI_INT,recvbuf,nd,MPI_INT,MPI_PROC_NULL,mpi_comm_inter);
62 }
63 }
64 fprintf(fp0,"\n");
65 for(i=0;i<sizeW;i++){
66     fprintf(fp0," index=%2d  recv=%02d\n",i,recvbuf[i]);
67 }
```

# 基本的なMPI関数の紹介

## Inter-communicatorの使用例：intercomm.c

ログインノードの画面表示

各プロセスの出力内容（2ノード4プロセスで実行した場合）

```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$ mpicc -o intercomm intercomm.c <----- ・ C言語で書かれたMPIプログラムのコンパイルとリンクをした。
$ sbatch -p testq -t 00:01:00 exec.sh intercomm <-- ・ ジョブを投入した。
Submitted batch job 93360 <----- ・ ジョブ番号が「93360」のジョブとして受け付けられたとの表示
$
$ head -n 8 exec.sh_93360.stderr | tail -n 1
+ mpiexec -n 4 --machinefile exec.sh_93360.machinefile bash numa_bind_exec.sh 2 . intercomm
$                               ↑-- ・ 2ノード4プロセスで、ロードモジュールファイル「intercomm」内の命令列を実行した。
$                               その際、各プロセスを、それぞれ、別々のNUMAノードにバインドした。
$
$ ls -m *.txt <----- ・ 各プロセスより、ファイル名が
intercomm_0.txt, intercomm_1.txt, intercomm_2.txt, intercomm_3.txt 「ロードモジュールファイル名_ランク番号.txt」
$                               の形式のファイルが出力されたことを確認した。
$ ↓-- ・ 出力されたファイルの内容を横並びに併記する。
$ paste -d @ intercomm_[0-3].txt | sed 's/@@/ /g'
rankW= 0 sizeW= 4      rankW= 1 sizeW= 4      rankW= 2 sizeW= 4      rankW= 3 sizeW= 4
index= 0  send=00      index= 0  send=10      index= 0  send=20      index= 0  send=30
rankS= 0 sizeS= 2      rankS= 1 sizeS= 2      rankS= 0 sizeS= 2      rankS= 1 sizeS= 2
----- MPI_Gather -----
index= 0  recv=-1      index= 0  recv=-1      index= 0  recv=00      index= 0  recv=10
index= 1  recv=-1      index= 1  recv=-1      index= 1  recv=-1      index= 1  recv=-1
index= 2  recv=-1      index= 2  recv=-1      index= 2  recv=-1      index= 2  recv=-1
index= 3  recv=-1      index= 3  recv=-1      index= 3  recv=-1      index= 3  recv=-1
$                               group1                               group2
```

# 基本的なMPI関数の紹介

## プロセス生成の実行例：spawn.c

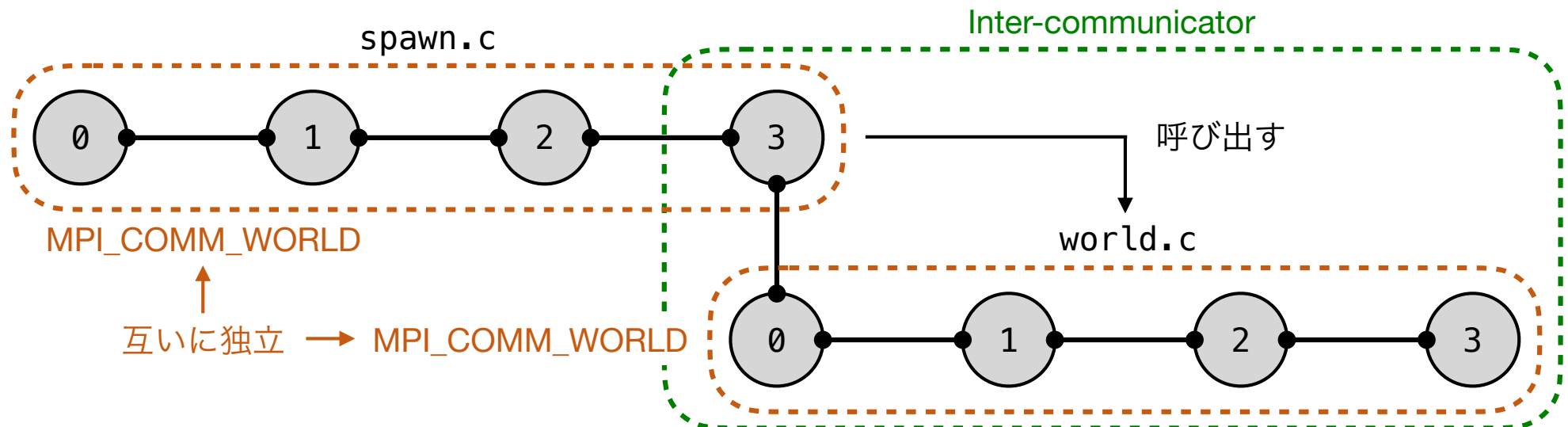
この実行例の目的：

実行中のMPIプログラムから、新たにプロセスを生成し、それにMPIプログラムを実行させる方法を紹介する。

この実行例の概要：

4つのプロセスにて実行中のMPIプログラム「spawn.c」から、新たなプロセスを4つ生成し、それらに、別のMPIプログラム「world.c」（最初のMPIプログラムの実行例で紹介したMPIプログラム）を、SPMD型並列実行させる。

「spawn.c」を実行中のプロセスのグループのコミュニケータ「MPI\_COMM\_WORLD」と、「world.c」を実行するためのプロセスのグループの「MPI\_COMM\_WORLD」は互いに独立。その2つの間のMPI通信用の「Inter-communicator」が作成される。



# 基本的なMPI関数の紹介

## プロセス生成の実行例：spawn.c

使用するMPI関数の説明（MPI3.1の規格書の第10章の記述の要約）

引数の文字の色は、引数の属性（**入力**、**出力**、**入出力**）を表す。

```
int MPI_Comm_spawn(const char *command, char *argv[], int maxprocs, MPI_Info info,  
                  int root, MPI_Comm comm, MPI_Comm *intercomm, int array_of_errcodes[])
```

コミュニケータ「comm」のランク番号「root」のプロセスが、新たなプロセスを「maxprocs」個生成し、その「maxprocs」個のプロセスで、MPIプログラム「command」を、SPMD型並列実行する。「argv[]」は、MPIプログラム「command」に与えるコマンドライン引数。

起動された「maxprocs」個のプロセスをメンバーとするグループは、独自の「MPI\_COMM\_WORLD」を持つ。コミュニケータ「comm」を「local\_comm」、起動されたプロセスグループの「MPI\_COMM\_WORLD」を「remote\_comm」とする、インターコミュニケータ「intercomm」が作成される。

# 基本的なMPI関数の紹介

## プロセス生成の実行例：spawn.c

ソースプログラムファイル「spawn.c」の内容

各プロセスが、それぞれ独立に、以下の命令列を実行する。

**argv[0]**: プログラム名

**sizeW**: 「MPI\_COMM\_WORLD」内のプロセス数

**rankW**: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

```
19  int procid;
20  unsigned int coreid,numaid;
21  char nameH[9]="@@@@@@@";
22  gethostname(nameH,8);nameH[8]='\0'; <-----
23  procid=(int)getpid(); <-----
24  syscall(__NR_getcpu,&coreid,&numaid,NULL); <-----
25  fprintf(fp0," hostname:%8s\n procid  :%8d\n",nameH,procid); <---
26  fprintf(fp0," numaid  :%8u\n coreid  :%8u\n",numaid,coreid); <--

/* ここ（25行目）までは「world.c」と全く同じ */

27  MPI_Comm mpi_comm_child;
28  int rankS=sizeW-1;
29  char *command="bash";
30  char *arguments[]={"numa_bind_exec.sh","2","../world","world",NULL};
31  MPI_Comm_spawn(command,arguments,4,MPI_INFO_NULL,
32                rankS,MPI_COMM_WORLD,&mpi_comm_child,MPI_ERRCODES_IGNORE);

↑--・ランク番号が「rankS(=sizeW-1)」のプロセスに、新たなプロセスを4つ生成させ、
    その4つのプロセスに、MPIプログラム「../world」を、SPMD型並列実行させる。
    新たに生成された4つのプロセスをメンバーとする別の独立した「MPI_COMM_WORLD」が存在する。
```

- ・ 茶色の文字は実際のソースプログラムファイル中には実在しない文字。
- ・ 白黒反転表示の文は、MPIプログラムに固有の関数等を含む文。
- ・ 19行目より前は「world.c」と全く同じ。

- ・ 現行プロセスを実行中のホスト名の最初の8文字、
- ・ 現行プロセスのプロセスID、
- ・ 現行プロセスを実行中のNUMAノードID、
- ・ 現行プロセスを実行中のCPUコアIDを、
- ・ ファイル「fp0」に書き込む。

# 基本的なMPI関数の紹介

## プロセス生成の実行例：spawn.c

ログインノードの画面表示

各プロセスの出力内容（2ノード4プロセスで実行した場合）

```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$ mpicc -o spawn spawn.c <----- ・ C言語で書かれたMPIプログラムのコンパイルとリンクをした。
$
$ diff ../world/exec.sh exec.sh
2c2
< #SBATCH --nodes=2                ## SLURM_JOB_NUM_NODES
-
> #SBATCH --nodes=4                ## SLURM_JOB_NUM_NODES <-- ・ このジョブのためには、計算ノードを4つ要求した。
$
$ sbatch -p testq -t 00:01:00 exec.sh spawn <-- ・ ジョブを投入した。
Submitted batch job 93370 <----- ・ ジョブ番号が「93370」のジョブとして受け付けられたとの表示
$
$ cat exec.sh_93370.machinefile <----- ・ このジョブのために4つの計算ノードが確保されたことを確認した。
cmptnd02
cmptnd02
cmptnd03
cmptnd03
cmptnd06
cmptnd06
cmptnd07
cmptnd07
↓-- ・ 2ノード4プロセスで、ロードモジュールファイル「spawn」内の命令列を実行し、
↓   その際、各プロセスを、それぞれ、別々のNUMAノードにバインドしたことを確認した。
$ head -n 8 exec.sh_93370.stderr | tail -n 1
+ mpiexec -n 4 --machinefile exec.sh_93370.machinefile bash numa_bind_exec.sh 2 . spawn
$
$ ls -m *.txt <----- ・ 各プロセスより、ファイル名が
spawn_0.txt, spawn_1.txt, spawn_2.txt, spawn_3.txt, 「ロードモジュールファイル名_ランク番号.txt」
world_0.txt, world_1.txt, world_2.txt, world_3.txt の形式のファイルが出力されたことを確認した。
$
```

# 基本的なMPI関数の紹介

## プロセス生成の実行例：spawn.c

ログインノードの画面表示

各プロセスの出力内容（2ノード4プロセスで実行した場合）

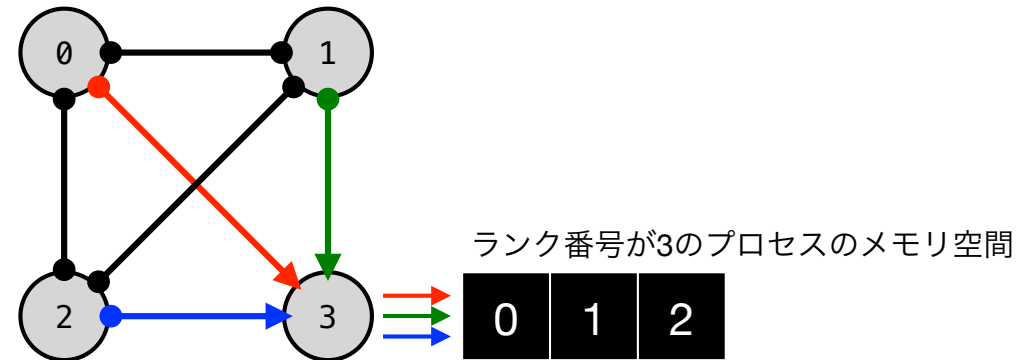
```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$   ↓ー・ 出力されたファイルの内容を横並びに併記する。
$ paste -d @ spawn_[0-3].txt | sed 's/@/      /g'
  rankW= 0 sizeW= 4      rankW= 1 sizeW= 4      rankW= 2 sizeW= 4      rankW= 3 sizeW= 4
hostname:cmptnd02      hostname:cmptnd02      hostname:cmptnd03      hostname:cmptnd03
procid  : 1102842      procid  : 1102841      procid  : 1044270      procid  : 1044269
numaid  :          0      numaid  :          1      numaid  :          0      numaid  :          1
coreid  :          110      coreid  :          143      coreid  :          21      coreid  :          113
$
$ paste -d @ world_[0-3].txt | sed 's/@/      /g'
  rankW= 0 sizeW= 4      rankW= 1 sizeW= 4      rankW= 2 sizeW= 4      rankW= 3 sizeW= 4
hostname:cmptnd06      hostname:cmptnd06      hostname:cmptnd07      hostname:cmptnd07
procid  : 1023733      procid  : 1023732      procid  : 1091110      procid  : 1091109
numaid  :          0      numaid  :          1      numaid  :          0      numaid  :          1
coreid  :          7      coreid  :          112      coreid  :          42      coreid  :          112
$
```

# 基本的なMPI関数の紹介

## One-sided Communicationの実行例：put.c

### Point-to-point Communication（既出）のイメージ

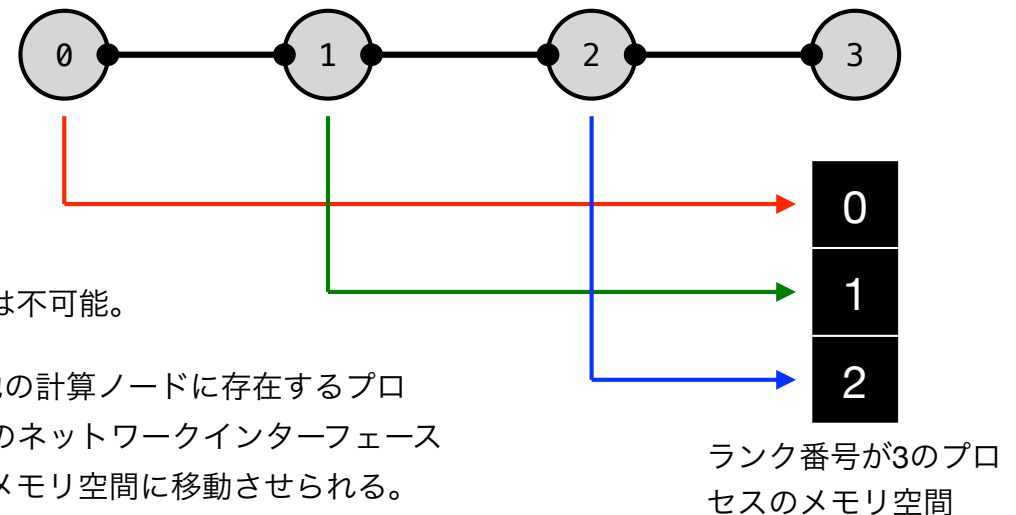
- ランク番号が3のプロセスが、他のプロセスのそれぞれから、データを受け取り、自分のメモリ空間内に格納する。



---

### One-sided Communicationのイメージ

- ランク番号が3のプロセスのメモリ空間内に、他のプロセスのそれぞれが独立に、直接データを格納する（Remote Memory Access）。
- 実際には、他のプロセスのメモリ空間に、直接アクセスすることは不可能。
- 4つのプロセスが、複数の計算ノードに跨がって存在する場合、他の計算ノードに存在するプロセスから送られて来たデータは、先ず受信側の計算ノードに付属のネットワークインターフェースのバッファ領域に一時的に格納され、その後、送信先プロセスのメモリ空間に移動させられる。
- 要所要所において、各プロセスのメモリ領域と、ネットワークインターフェースのバッファ領域などとの間で同期を取ること（RMA同期）が必要。





# 基本的なMPI関数の紹介

## One-sided Communicationの実行例：put.c

使用するMPI関数の説明（MPI3.1の規格書の第11章の記述の要約）

引数の文字の色は、引数の属性（**入力**、**出力**、**入出力**）を表す。

```
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit,  
                  MPI_Info info, MPI_Comm comm, MPI_win *win)
```

この関数を、コミュニケータ「comm」内の全てのプロセスが実行することにより、「comm」内の全てのプロセスが、互いにRMA(Remote Memory Access)するための、ウィンドウ「win」を作成する。

各プロセスは、他のプロセスからのRMAを許すメモリ領域を、アドレス「base」から始まる「size」バイトとして指定する。

基本的には、「disp\_unit」には「1」を、「info」には「MPI\_INFO\_NULL」を指定する。

```
int MPI_Win_fence(int assert, MPI_Win win)
```

この関数を、コミュニケータ「comm」内の全てのプロセスが実行することにより、各プロセスのメモリ領域と、ネットワークインターフェースのバッファ領域などとの間で同期を取る（RMA同期）。

基本的には、「assert」には「0」を指定する。

2回のRMA同期の間（epoch）において、One-sided Communication（RMA通信）を行うことができる。

# 基本的なMPI関数の紹介

## One-sided Communicationの実行例：put.c

使用するMPI関数の説明（MPI3.1の規格書の第11章の記述の要約）

引数の文字の色は、引数の属性（**入力**、**出力**、**入出力**）を表す。

```
int MPI_Put(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype,  
            int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype,  
            MPI_Win win)
```

この関数がある1つのプロセス「origin process」が実行すると、そのプロセスのメモリ空間内のアドレス「origin\_addr」から始まる場所にある、「origin\_datatype」型の要素「origin\_count」個のデータを、宛先のプロセス「target process」のメモリ空間内のアドレス「target\_addr」から始まる場所に、「target\_datatype」型の要素「target\_count」個のデータとして格納する。

「target process」は、ウィンドウ「win」を作成するとき（前ページ参照）に指定したコミュニケータ内でのランク番号が「target\_rank」のプロセス。

「target\_addr」は、式「window\_base + target\_disp x window\_disp\_unit」で計算される値。

「window\_base」は、ウィンドウ「win」を作成するとき（前ページ参照）に指定したアドレス「base」の値。

「window\_disp\_unit」は、ウィンドウ「win」を作成するとき（前ページ参照）に指定した「disp\_unit」の値。

宛先プロセスである「target process」において、「MPI\_Put」に対応する受信のためのMPI関数を実行する必要はない。

関数「MPI\_Put」の反対の動作をする（他のプロセスからデータをコピーしてくる）関数「MPI\_Get」も定義されている。

```
int MPI_Win_free(MPI_Win *win)
```

この関数を、ウィンドウ「win」を作成した全てのプロセスが実行することにより、ウィンドウ「win」を破棄する。各プロセスの変数「win」には、定数値「MPI\_WIN\_NULL」が格納される。

# 基本的なMPI関数の紹介

## One-sided Communicationの実行例：put.c

ソースプログラムファイル「put.c」の内容

各プロセスが、それぞれ独立に、以下の命令列を実行する。

**argv[0]**: プログラム名

**sizeW**: 「MPI\_COMM\_WORLD」内のプロセス数

**rankW**: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

- ・ 茶色の文字は実際のソースプログラムファイル中には実在しない文字。
- ・ 白黒反転表示の文は、MPIプログラムに固有の関数等を含む文。
- ・ 19行目より前は「world.c」と全く同じ。

```
19  int nd,i,rootW;
20  int *sendbuf,*winbuf;
21  nd=1;
22  sendbuf=(int *)malloc(sizeof(int)*nd);
23  winbuf=(int *)malloc(sizeof(int)*nd*sizeW);
24  sendbuf[0]=rankW*10; <----- ・ 配列要素「sendbuf[0]」に、変数「rankW」の値を10倍した値を格納する。
25  for(i=0;i<sizeW;i++){
26      winbuf[i]=-1; <----- ・ 配列「winbuf」の全要素に、値「-1」を格納する。
27  }
28  fprintf(fp0,"\n");
29  for(i=0;i<nd;i++){
30      fprintf(fp0," index=%2d  send=%02d\n",i,sendbuf[i]);
31  }      ↑-- ・ 配列要素番号「i」と配列要素「sendbuf[i]」の値を、「index」と「send」の値として、「fp0」に書き込む。

                ↓-- ・ コミュニケータ「MPI_COMM_WORLD」に属する全てのプロセスが、
32  MPI_Win_winobj;      ↓      互いにRMAを行うためのウィンドウ「winobj」を設定する。
33  MPI_Win_create(winbuf,sizeof(int)*nd*sizeW,1,MPI_INFO_NULL,MPI_COMM_WORLD,&winobj);

34  MPI_Win_fence(0,winobj); <--- ・ ウィンドウ「winobj」でのRMAの同期を取る。-----

                ↓-- ・ ランク番号「rankW」の値が変数「rootW」の値以外のプロセスが、それぞれの配列
35  rootW=sizeW-1;      ↓      要素「sendbuf[0]」の内容を、ランク番号「rankW」の値が変数「rootW」の値と
36  if(rankW!=rootW){    ↓      同じプロセスの配列要素「winbuf[rankW]」に書き込む。
37      MPI_Put(&(sendbuf[0]),nd,MPI_INT,rootW,nd*sizeof(int)*rankW,nd,MPI_INT,winobj);
38  }
39  MPI_Win_fence(0,winobj); <--- ・ ウィンドウ「winobj」でのRMAの同期を取る。-----
40  MPI_Win_free(&winobj); <---- ・ RMAを行うためのウィンドウ「winobj」の設定を解除する。
41  fprintf(fp0,"\n");
42  for(i=0;i<sizeW;i++){
43      fprintf(fp0," index=%2d  wnbf=%02d\n",i,winbuf[i]);
44  }      ↑-- ・ 配列要素番号「i」と配列要素「winbuf[i]」の値を、「index」と「wnbf」の値として、「fp0」に書き込む。
```

An access  
epoch for  
winobj

# 基本的なMPI関数の紹介

## One-sided Communicationの実行例：put.c

ログインノードの画面表示

各プロセスの出力内容（2ノード4プロセスで実行した場合）

```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$ mpicc -o put put.c <-----・ C言語で書かれたMPIプログラムのコンパイルとリンクをした。
$ sbatch -p testq -t 00:01:00 exec.sh put <--・ ジョブを投入した。
Submitted batch job 93364 <-----・ ジョブ番号が「93364」のジョブとして受け付けられたとの表示
$
$ head -n 8 exec.sh_93364.stderr | tail -n 1
+ mpiexec -n 4 --machinefile exec.sh_93364.machinefile bash numa_bind_exec.sh 2 . put
$                                     ↑--・ 2ノード4プロセスで、ロードモジュールファイル「put」内の命令列を実行した。
$                                     その際、各プロセスを、それぞれ、別々のNUMAノードにバインドした。
$
$ ls -m *.txt <-----・ 各プロセスより、ファイル名が
put_0.txt, put_1.txt, put_2.txt, put_3.txt      「ロードモジュールファイル名_ランク番号.txt」
$                                               の形式のファイルが出力されたことを確認した。
$ ↓-・ 出力されたファイルの内容を横並びに併記する。
$ paste -d @ put_[0-3].txt | sed 's/@/ /g'
rankW= 0 sizeW= 4      rankW= 1 sizeW= 4      rankW= 2 sizeW= 4      rankW= 3 sizeW= 4
index= 0  send=00      index= 0  send=10      index= 0  send=20      index= 0  send=30
----- MPI_Put
index= 0  recv=-1      index= 0  recv=-1      index= 0  recv=-1      index= 0  recv=00
index= 1  recv=-1      index= 1  recv=-1      index= 1  recv=-1      index= 1  recv=10
index= 2  recv=-1      index= 2  recv=-1      index= 2  recv=-1      index= 2  recv=20
index= 3  recv=-1      index= 3  recv=-1      index= 3  recv=-1      index= 3  recv=-1
$
```

# 基本的なMPI関数の紹介

Timerの使用例：wtime.c

- このプログラムの目的

ユーザープログラム内の、ある区間の、実行に要した時間を計測するための、MPI関数「MPI\_Wtime」の動作を確認する。

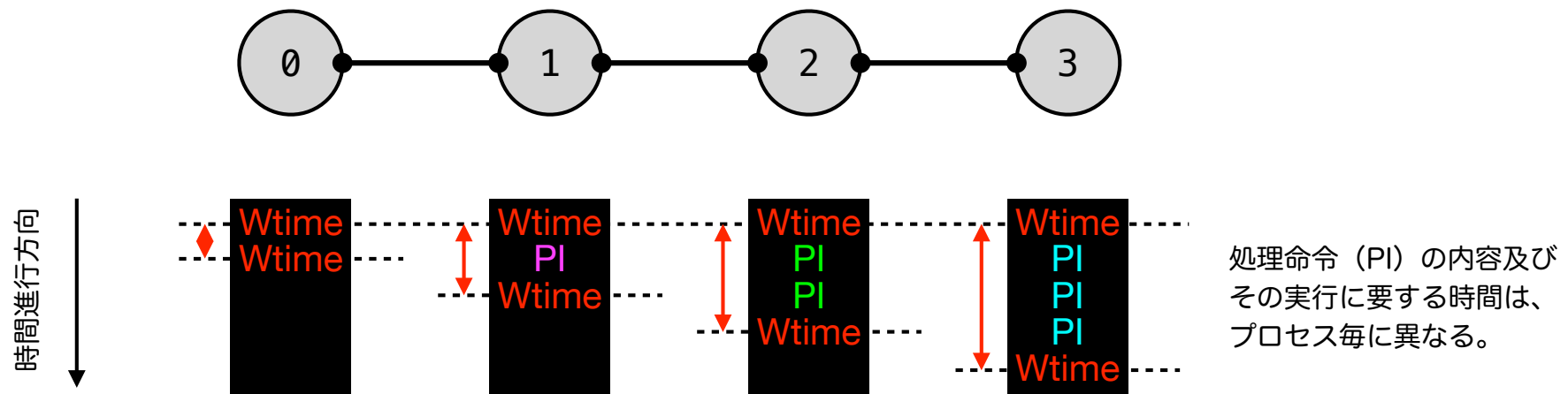
- 各プロセスでの処理の概要

ランク番号0のプロセス：Wtimeを1回実行し、何もせず（このプログラムでは0秒間のスリープをする）、Wtimeを1回実行する。

ランク番号1のプロセス：Wtimeを1回実行し、実行に1秒かかる処理命令（PI）を実行した後に、Wtimeを1回実行する。

ランク番号2のプロセス：Wtimeを1回実行し、実行に2秒かかる処理命令（PI）を実行した後に、Wtimeを1回実行する。

ランク番号3のプロセス：Wtimeを1回実行し、実行に3秒かかる処理命令（PI）を実行した後に、Wtimeを1回実行する。



MPI関数「MPI\_Wtime」の実行結果として得られた値の差が、その間の処理命令（PI）の実行に要した時間となる。

# 基本的なMPI関数の紹介

## Timerの使用例：wtime.c

使用するMPI関数の説明（MPI3.1の規格書の第8章の記述の要約）

引数の文字の色は、引数の属性（**入力**、**出力**、**入出力**）を表す。

```
double MPI_Wtime(void)
```

この関数は、エラーコードではなく、取得した値を、返り値とする。

過去のある時点からの経過時間（秒）を取得する。

OSが管理している過去のある時点（1970年1月1日0時0分0秒であることが多い）からの経過時間の情報を参照し、それをユーザーが指定した変数に格納することにより、記録する。

時間計測対象区間の直前と直後で、それぞれ1回ずつ、経過時間の情報の記録を行い、得られた2つの値の差を、その時間計測対象区間の実行に要した時間（Elapsed Time）の計測値とすることができる。

過去のある時点からの経過時間の情報を参照し記録するという行為そのものにもある程度の時間が掛かる。

OSが管理している過去のある時点からの経過時間の情報の更新間隔が十分に短ければ、ユーザープログラム内で、過去のある時点からの経過時間の情報を参照し記録するということを、立て続けに2回行ったとしても、その記録された2つの値の差はゼロにはならない。

OSが管理している過去のある時点からの経過時間の情報の更新間隔の短さが不十分な場合には、記録された2つの値の差がゼロとなることもあり得る。

```
double MPI_Wtick(void)
```

この関数は、エラーコードではなく、取得した値を、返り値とする。

関数「MPI\_Wtime」による時間計測の分解能（秒）を取得する。

使用する計算機システムによっては、関数「MPI\_Wtime」により測定可能な最小の時間差ではなく、OSが管理している過去のある時点からの経過時間の情報の更新間隔が得られる。

# 基本的なMPI関数の紹介

## Timerの使用例：wtime.c

ソースプログラムファイル「wtime.c」の内容

各プロセスが、それぞれ独立に、以下の命令列を実行する。

**argv[0]**: プログラム名

**sizeW**: 「MPI\_COMM\_WORLD」内のプロセス数

**rankW**: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

- ・ 茶色の文字は実際のソースプログラムファイル中には実在しない文字。
- ・ 白黒反転表示の文は、MPIプログラムに固有の関数等を含む文。
- ・ 19行目より前は「world.c」と全く同じ。

```
19 double wtick,time1,time2;
```

```
20 wtick=MPI_Wtick(); <--・関数「MPI_Wtime」による時間計測の分解能（秒）を取得し、double型の変数「wtick」に代入する。
```

```
21 time1=MPI_Wtime(); <--・過去のある時間からの経過時間（秒）を取得し、double型の変数「time1」に代入する。
```

```
22 sleep(rankW); <--・処理命令（PI）：現行プロセスを、「MPI_COMM_WORLD」内でのランク番号の値と同じ秒数の間sleepさせる。
```

```
23 time2=MPI_Wtime(); <--・過去のある時間からの経過時間（秒）を取得し、double型の変数「time2」に代入する。
```

```
24 fprintf(fp0," %4.2e %4.2e\n",time2-time1,wtick);
```

↑--・「time1」の値と「time2」の値の差（22行目の「sleep」の実行に要した時間と解釈されるもの）と、関数「MPI\_Wtime」による時間計測の分解能（秒）を、ファイル「fp0」に書き込む。

# 基本的なMPI関数の紹介

## Timerの使用例：wtime.c

ログインノードの画面表示

各プロセスの出力内容（2ノード4プロセスで実行した場合）

```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$ mpicc -o wtime wtime.c <-----・ C言語で書かれたMPIプログラムのコンパイルとリンクをした。
$ sbatch -p testq -t 00:01:00 exec.sh wtime <--・ ジョブを投入した。
Submitted batch job 93374 <-----・ ジョブ番号が「93374」のジョブとして受け付けられたとの表示
$
$ head -n 8 exec.sh_93374.stderr | tail -n 1
+ mpiexec -n 4 --machinefile exec.sh_93374.machinefile bash numa_bind_exec.sh 2 . wtime
$                                     ↑--・ 2ノード4プロセスで、ロードモジュールファイル「wtime」内の命令列を実行した。
$                                     その際、各プロセスを、それぞれ、別々のNUMAノードにバインドした。
$
$ ls -m *.txt <-----・ 各プロセスより、ファイル名が
wtime_0.txt, wtime_1.txt, wtime_2.txt, wtime_3.txt    「ロードモジュールファイル名_ランク番号.txt」
$                                                       の形式のファイルが出力されたことを確認した。
$ ↓--・ 出力されたファイルの内容を横並びに併記する。
$ paste -d @ wtime_[0-3].txt | sed 's/@/      /g'
rankW= 0 sizeW= 4      rankW= 1 sizeW= 4      rankW= 2 sizeW= 4      rankW= 3 sizeW= 4
6.08e-05 1.00e-09      1.00e+00 1.00e-09      2.00e+00 1.00e-09      3.00e+00 1.00e-09
$                                     ↑--・ 使用する計算機システムによっては、関数「MPI_Wtime」により測定可能な最小の時間差ではなく、
$                                     OSが管理している過去のある時点からの経過時間の情報の更新間隔が得られる。
```



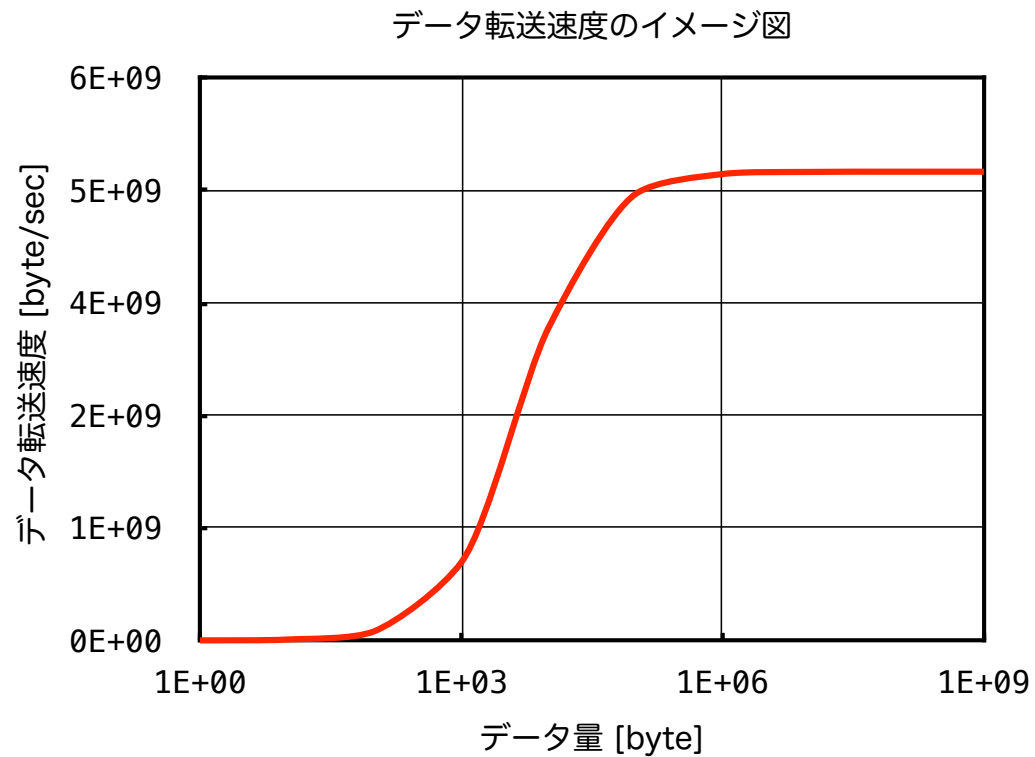
# 基本的なMPI関数の紹介

## ネットワークの転送性能 (Throughput)

ネットワーク通信によるデータ転送速度は、1回に送受信するデータ量が少ない時には遅い。

ネットワーク通信を行うための準備にも時間がかかる

データ転送は、出来るだけ纏めて行い、ネットワーク通信の回数を減らすのが原則。



# 基本的なMPI関数の紹介

## 派生データ型の使用例：typevec.c

問題意識：

複数の互いに離れた場所にあるデータを転送する際に、MPI通信を複数回実行するのではなく、1回で済ませたい。

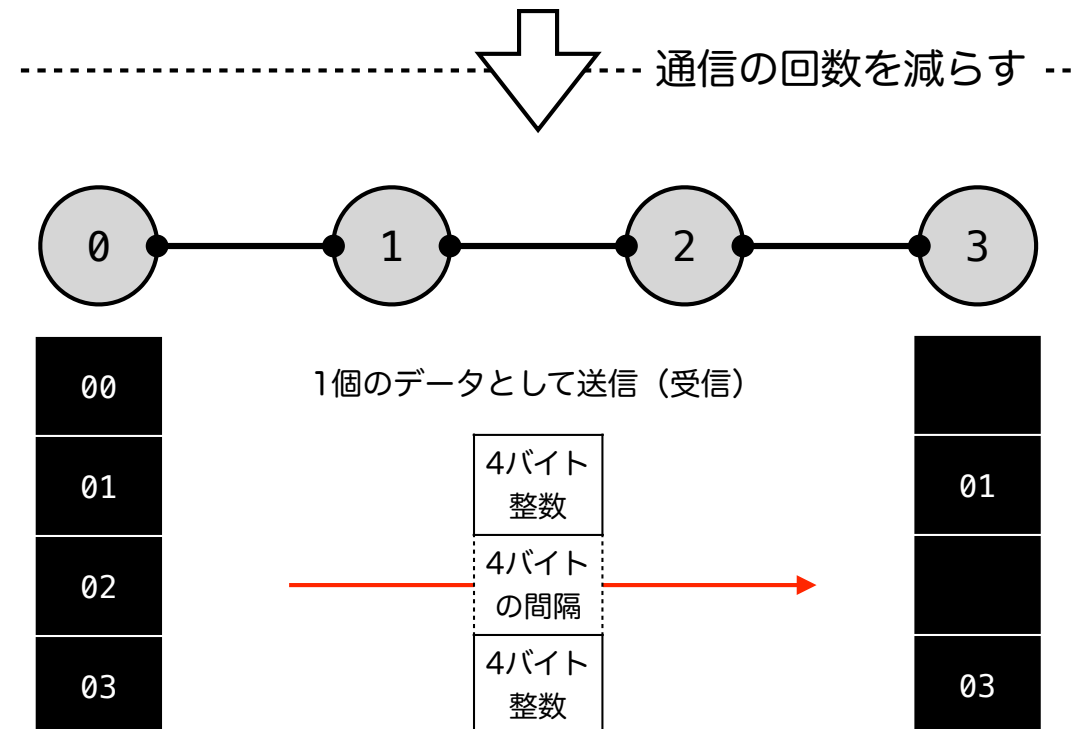
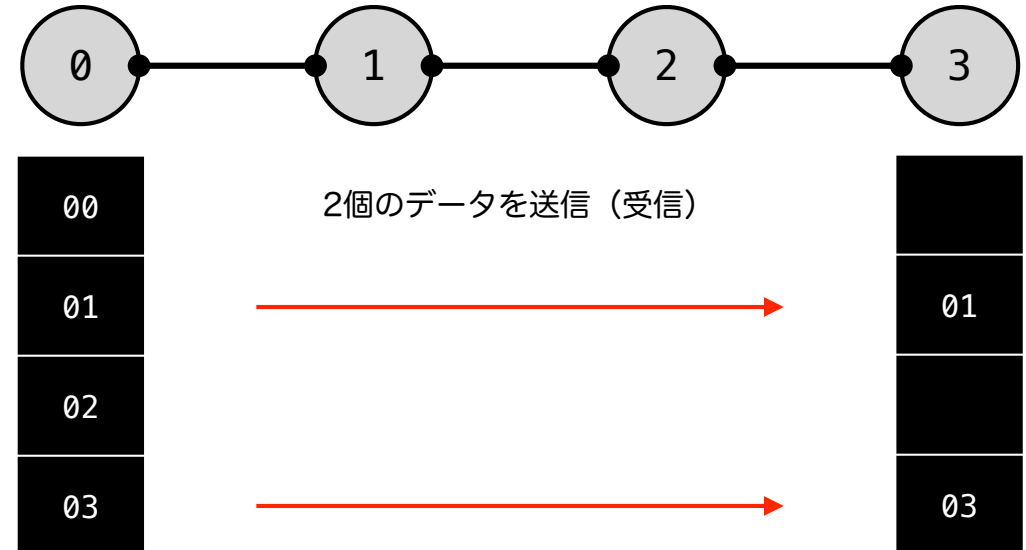
解決方法：

データの配置の情報を含めた新たなデータ型（派生データ型）を定義し、そのデータ型のデータを1つ転送することとすれば良い。

注意事項：

実際には、送信側プロセスと受信側プロセスの両方にて、MPIライブラリが作業用配列を1つずつ用意し、送信側プロセスでは、それに送信する複数のデータを隙間なく詰め込み、その内容をMPI通信を1回実行することにより送信し、受信側プロセスでは、MPI通信を1回実行することにより受信したデータを、作業用配列に一旦格納し、その内容を新たに定義されたデータ型に含まれるデータ配置情報に基づき、MPI通信関数実行の際に指定したアドレスから始まるメモリ領域に展開するようにMPIライブラリが実装され、そこでのデータの詰め込み及び展開に伴うコピー操作がスレッド並列化されていないことが多い。

派生データ型を使用せず、作業配列の作成と、データの詰め込み及び展開を、OpenMPによるスレッド並列化を行う形で、陽に書いた方が実行時間が短いことが多い。



# 基本的なMPI関数の紹介

## 派生データ型の使用例：typevec.c

使用するMPI関数の説明（MPI3.1の規格書の第4章の記述の要約）

引数の文字の色は、引数の属性（**入力**、**出力**、**入出力**）を表す。

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

データ型が「oldtype」型のデータ「blocklength」個からなるブロックが、データ型が「oldtype」型のデータ「stride」個の間隔で、「count」個並んで存在している時に、そこにデータ型が「newtype」型のデータが1個存在していると見なすための新たなデータ型（派生データ型）「newtype」型を定義する。

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

派生データ型「datatype」を、MPI通信関数にて使用することを可能とする。

```
int MPI_Type_free(MPI_Datatype *datatype)
```

派生データ型「datatype」の使用を、ひとまず終えたことを宣言する。  
変数「datatype」には、定数値「MPI\_DATATYPE\_NULL」が格納される。

# 基本的なMPI関数の紹介

## 派生データ型の使用例：typevec.c

ソースプログラムファイル「typevec.c」の内容

各プロセスが、それぞれ独立に、以下の命令列を実行する。

**argv[0]**: プログラム名

**sizeW**: 「MPI\_COMM\_WORLD」内のプロセス数

**rankW**: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

```
19 int nd,tag,i,bcount,blklen,stride;
20 int *sendbuf,*recvbuf;
21 nd=4;
22 sendbuf=(int *)malloc(sizeof(int)*nd);
23 recvbuf=(int *)malloc(sizeof(int)*nd);
24 for(i=0;i<nd;i++){
25     sendbuf[i]=rankW*10+i; <-----
26     recvbuf[i]=-1; <-----
27 }
28 fprintf(fp0,"\n");
29 for(i=0;i<nd;i++){
30     fprintf(fp0," index=%2d send=%02d\n",i,sendbuf[i]);
31 }
32 bcount=2;
33 blklen=1;
34 stride=2;
35 MPI_Datatype newtype;
36 MPI_Type_vector(bcount,blklen,stride,MPI_INT,&newtype);
37 MPI_Type_commit(&newtype);
38 tag=32767;
39 if(rankW==0){
40     MPI_Send(&sendbuf[1],1,newtype,sizeW-1,tag,MPI_COMM_WORLD);
41 }
42 if(rankW==sizeW-1){
43     MPI_Recv(&recvbuf[1],1,newtype,0,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
44 }
45 MPI_Type_free(&newtype);
46 fprintf(fp0,"\n");
47 for(i=0;i<sizeW;i++){
48     fprintf(fp0," index=%2d recv=%02d\n",i,recvbuf[i]);
49 }
```

・茶色の文字は実際のソースプログラムファイル中には実在しない文字。  
・白黒反転表示の文は、MPIプログラムに固有の関数等を含む文。  
・19行目より前は「world.c」と全く同じ。

・配列「sendbuf」の各要素に、現行プロセスの「MPI\_COMM\_WORLD」内での  
ランク番号を10倍した値と、配列要素番号「i」の値を合計した値を格納する。

・配列「recvbuf」  
の全要素に、値「-1」を格納する。

↑--・配列要素番号「i」と配列要素「sendbuf[i]」の値を、「index」と「send」の値として、「fp0」に書き込む。

↓--・「MPI\_INT」型のデータ2つが、「MPI\_INT」型のデータが1つ分の隙間を開けてメモリ空間内で  
並んでいる状態を、そこに新たなデータ型のデータが1つ存在していると認識することとする  
↓ ための、1つの新たなデータ型「newtype」型を定義する。

↑--・「&(sendbuf[1])」が指すアドレスから始まる場所にある「newtype」型のデータ1つを、ランク番号が  
「sizeW-1」のプロセスに向けて送信する。

↓--・ランク番号が「0」のプロセスから「newtype」型のデータが1つ送られてきたら「&(recvbuf[1])」  
が指すアドレスから始まる場所に格納する。

↑--・「newtype」型のデータの送受信が  
ひとまず終了したことを宣言する。

↑--・配列要素番号「i」と配列要素「recvbuf[i]」の値を、「index」と「recv」の値として、「fp0」に書き込む。

# 基本的なMPI関数の紹介

## 派生データ型の使用例：typevec.c

ログインノードの画面表示

各プロセスの出力内容（2ノード4プロセスで実行した場合）

```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$ mpicc -o typevec typevec.c <----- ・ C言語で書かれたMPIプログラムのコンパイルとリンクをした。
$ sbatch -p testq -t 00:01:00 exec.sh typevec <-- ・ ジョブを投入した。
Submitted batch job 93372 <----- ・ ジョブ番号が「93372」のジョブとして受け付けられたとの表示
$
$ head -n 8 exec.sh_93372.stderr | tail -n 1
+ mpiexec -n 4 --machinefile exec.sh_93372.machinefile bash numa_bind_exec.sh 2 . typevec
$                               ↑-- ・ 2ノード4プロセスで、ロードモジュールファイル「typevec」内の命令列を実行した。
$                               その際、各プロセスを、それぞれ、別々のNUMAノードにバインドした。
$
$ ls -m *.txt <----- ・ 各プロセスより、ファイル名が
typevec_0.txt, typevec_1.txt, typevec_2.txt, typevec_3.txt 「ロードモジュールファイル名_ランク番号.txt」
$                               の形式のファイルが出力されたことを確認した。
$ ↓-- ・ 出力されたファイルの内容を横並びに併記する。
$ paste -d @ typevec_[0-3].txt | sed 's/@/ /g'
rankW= 0 sizeW= 4      rankW= 1 sizeW= 4      rankW= 2 sizeW= 4      rankW= 3 sizeW= 4

index= 0  send=00      index= 0  send=10      index= 0  send=20      index= 0  send=30
index= 1  send=01      index= 1  send=11      index= 1  send=21      index= 1  send=31
index= 2  send=02      index= 2  send=12      index= 2  send=22      index= 2  send=32
index= 3  send=03      index= 3  send=13      index= 3  send=23      index= 3  send=33
-----
index= 0  recv=-1      index= 0  recv=-1      index= 0  recv=-1      index= 0  recv=-1
index= 1  recv=-1      index= 1  recv=-1      index= 1  recv=-1      index= 1  recv=01
index= 2  recv=-1      index= 2  recv=-1      index= 2  recv=-1      index= 2  recv=-1
index= 3  recv=-1      index= 3  recv=-1      index= 3  recv=-1      index= 3  recv=03
$
```

MPI\_Send  
MPI\_Recv





# 基本的なMPI関数の紹介

## Collective Communicationとは

- ・ 複数のプロセス間でのデータ転送を一括して行うためのMPI関数を、Collective Communication関数と呼ぶ。
- ・ 関係する全プロセスが、それぞれ、同じMPI関数を呼び出す必要がある。
- ・ 大型電子計算機では、各メーカーが、ネットワークの構造を考慮し、最適化したMPI関数を提供していることが多い。
  - Collective Communication関数で書くことができる部分は、Point-to-Point Communication関数ではなく、Collective Communication関数で書いておいた方が、プログラムの実行時間が短くなることが多い。

(ここまでは、既出)

- ・ MPI3.1規格では、多数のCollective Communication関数が定義されている。
  - ・ Collective Communicationは、Blocking通信が基本。
  - ・ Nonblocking通信関数もあるが、Blocking通信関数の変種（派生関数）の一部。

	Blocking	Nonblocking
Point-to-Point		
Collective		

# 基本的なMPI関数の紹介

## Collective Communication関数の一覧 (MPI3.1の規格書の第5章から抜粋して引用)

- **MPI\_BARRIER**, MPI\_IBARRIER  
Barrier synchronization across all members of a group.
- **MPI\_BCAST**, MPI\_IBCAST  
Broadcast from one member to all members of a group.
- **MPI\_GATHER**, **MPI\_IGATHER**, MPI\_GATHERV, MPI\_IGATHERV  
Gather data from all members of a group to one member.
- **MPI\_SCATTER**, MPI\_ISCATTER, MPI\_SCATTERV, MPI\_ISCATTERV  
Scatter data from one member to all members of a group.
- **MPI\_ALLGATHER**, MPI\_IALLGATHER, MPI\_ALLGATHERV, MPI\_IALLGATHERV  
A variation on Gather where all members of a group receive the result.
- **MPI\_ALLTOALL**, MPI\_IALLTOALL, MPI\_ALLTOALLV, MPI\_IALLTOALLV, MPI\_ALLTOALLW, MPI\_IALLTOALLW  
Scatter/Gather data from all members to all members of a group.
- **MPI\_ALLREDUCE**, MPI\_IALLREDUCE, MPI\_REDUCE, MPI\_IREDUCE  
Global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all members of a group and a variation where the result is returned to only one member.
- **MPI\_REDUCE\_SCATTER\_BLOCK**, MPI\_IREDUCE\_SCATTER\_BLOCK, MPI\_REDUCE\_SCATTER, MPI\_IREDUCE\_SCATTER  
A combined reduction and scatter operation.
- **MPI\_SCAN**, MPI\_ISCAN, MPI\_EXSCAN, MPI\_IEXSCAN  
Scan across all members of a group (also called prefix).

本資料では、**代表的な関数（赤字）**について、次頁以降で、順次紹介する。

# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Barrier) の実行例：barrier.c

- このプログラムの目的

関数「MPI\_Barrier」の動作を確認する。全てのプロセスが関数「MPI\_Barrier」を実行するまで、各プロセスを待たせる。

- 各プロセスでの処理の概要

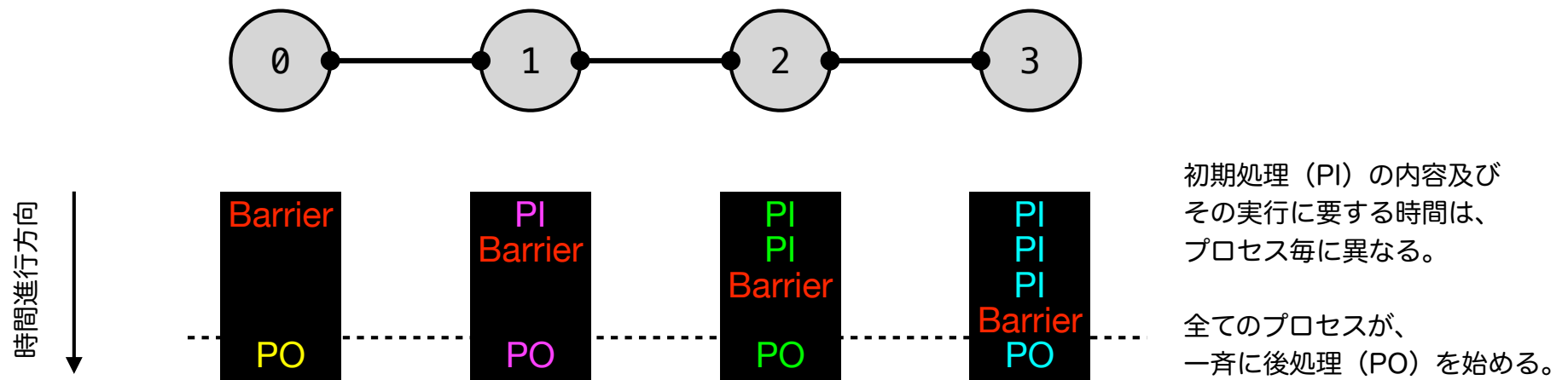
ランク番号0のプロセス：初期処理（PI）は実行しない（このプログラムでは0秒間のスリープをする）。

全プロセスがPIを実行し終わるまで待つ（MPI\_Barrierを実行する）。後処理（PO）を実行する。

ランク番号1のプロセス：実行に1秒かかるPIを実行した後に、全プロセスがPIを実行し終わるまで待つ。POを実行する。

ランク番号2のプロセス：実行に2秒かかるPIを実行した後に、全プロセスがPIを実行し終わるまで待つ。POを実行する。

ランク番号3のプロセス：実行に3秒かかるPIを実行した後に、全プロセスがPIを実行し終わるまで待つ。POを実行する。





# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Barrier) の実行例：barrier.c

使用するMPI関数の説明（MPI3.1の規格書の第5章の記述の要約）

引数の文字の色は、引数の属性（**入力**、**出力**、**入出力**）を表す。

```
int MPI_Barrier(MPI_Comm comm)
```

コミュニケータ「comm」内の全てのプロセスが、この関数「MPI\_Barrier」を実行するまで待つ。

# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Barrier) の実行例：barrier.c

ソースプログラムファイル「barrier.c」の内容

各プロセスが、それぞれ独立に、以下の命令列を実行する。

**argv[0]**: プログラム名

**sizeW**: 「MPI\_COMM\_WORLD」内のプロセス数

**rankW**: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

- ・茶色の文字は実際のソースプログラムファイル中には実在しない文字。
- ・白黒反転表示の文は、MPIプログラムに固有の関数等を含む文。
- ・19行目より前は「world.c」と全く同じ。

```
19  double wtick,time1,time2;
20  wtick=MPI_Wtick(); <--・関数「MPI_Wtime」による時間計測の分解能（秒）を取得し、double型の変数「wtick」に代入する。
21  time1=MPI_Wtime(); <--・過去のある時間からの経過時間（秒）を取得し、double型の変数「time1」に代入する。
22  sleep(rankW); <--・初期処理（PI）：現行プロセスを、「MPI_COMM_WORLD」内でのランク番号の値と同じ秒数の間sleepさせる。
23  MPI_Barrier(MPI_COMM_WORLD); <--・「MPI_COMM_WORLD」内の全てのプロセスが、この命令文を実行するまで待つ。
24  time2=MPI_Wtime(); <--・過去のある時間からの経過時間（秒）を取得し、double型の変数「time2」に代入する。
25  fprintf(fp0," %4.2e %4.2e\n",time2-time1,wtick);
    ↑--・後処理（PO）：「time1」と「time2」の値の差（22行目と23行目の実行に要した時間と解釈されるもの）と、
        関数「MPI_Wtime」による時間計測の分解能（秒）を、ファイル「fp0」に書き込む。
```

先述の「wtime.c」に、23行目の一文を追加しただけ。

# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Barrier) の実行例：barrier.c

ログインノードの画面表示

各プロセスの出力内容 (2ノード4プロセスで実行した場合)

```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$ mpicc -o barrier barrier.c <-----・ C言語で書かれたMPIプログラムのコンパイルとリンクをした。
$ sbatch -p testq -t 00:01:00 exec.sh barrier <--・ ジョブを投入した。
Submitted batch job 93353 <-----・ ジョブ番号が「93353」のジョブとして受け付けられたとの表示
$
$ head -n 8 exec.sh_93353.stderr | tail -n 1
+ mpiexec -n 4 --machinefile exec.sh_93353.machinefile bash numa_bind_exec.sh 2 . barrier
$
$          ↑--・ 2ノード4プロセスで、ロードモジュールファイル「barrier」内の命令列を実行した。
$          その際、各プロセスを、それぞれ、別々のNUMAノードにバインドした。
$
$ ls -m *.txt <-----・ 各プロセスより、ファイル名が
barrier_0.txt, barrier_1.txt, barrier_2.txt, barrier_3.txt  「ロードモジュールファイル名_ランク番号.txt」
$          の形式のファイルが出力されたことを確認した。
$          ↓--・ 出力されたファイルの内容を横並びに併記する。
$ paste -d @ barrier_[0-3].txt | sed 's/@/ /g'
rankW= 0 sizeW= 4      rankW= 1 sizeW= 4      rankW= 2 sizeW= 4      rankW= 3 sizeW= 4
3.00e+00 1.00e-09      3.00e+00 1.00e-09      3.00e+00 1.00e-09      3.00e+00 1.00e-09
$
$          ↑--・ 「MPI_Barrier」を挿入した結果、当該区間の実行時間が、全てのプロセスで同じになった。
$
$          ↓--・ 参考：「MPI_Barrier」未挿入の場合（「wtime.c」）の結果（既出）
$ paste -d @ ../wtime/wtime_[0-3].txt | sed 's/@/ /g'
rankW= 0 sizeW= 4      rankW= 1 sizeW= 4      rankW= 2 sizeW= 4      rankW= 3 sizeW= 4
6.08e-05 1.00e-09      1.00e+00 1.00e-09      2.00e+00 1.00e-09      3.00e+00 1.00e-09
$
```

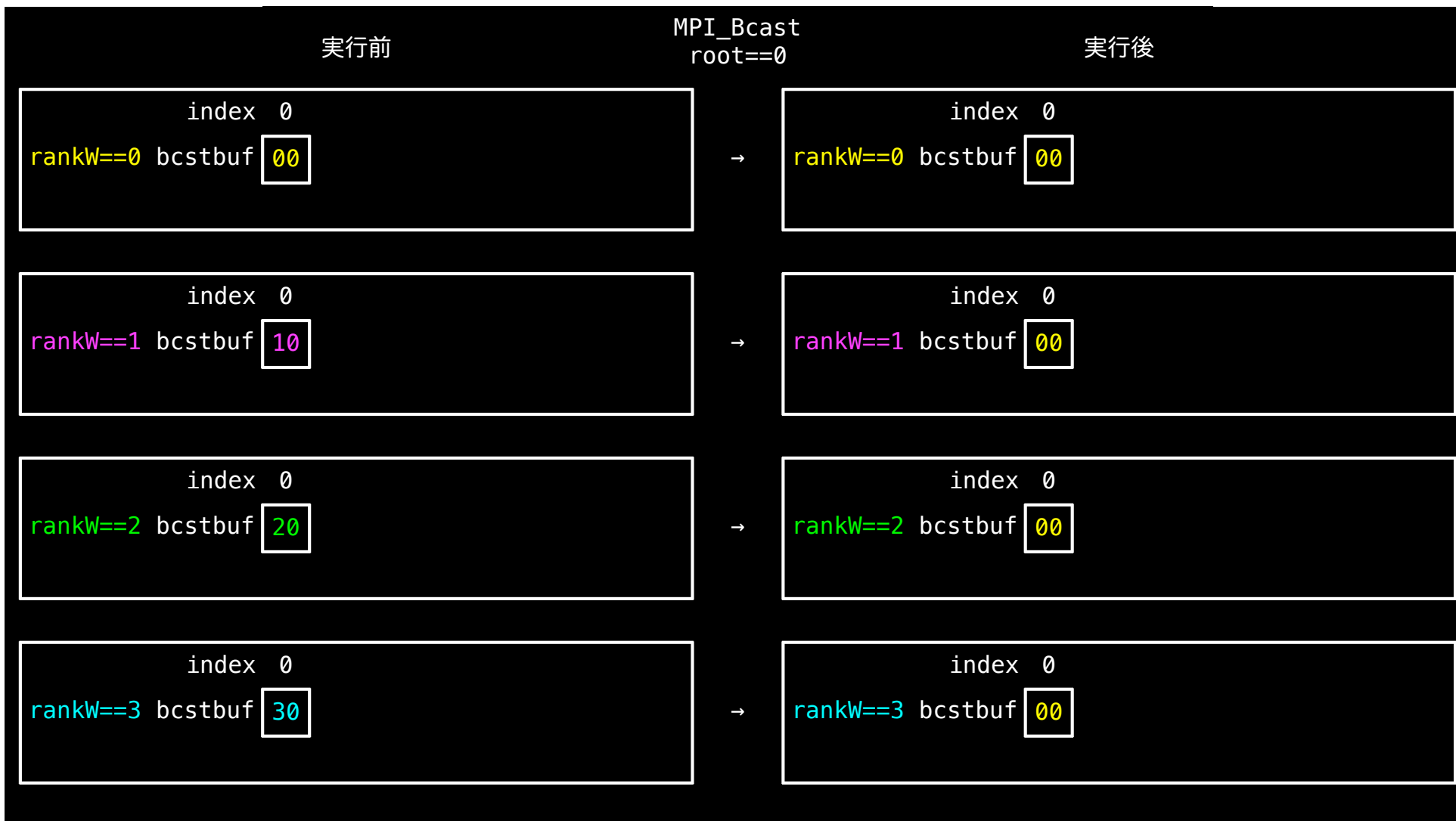
# 基本的なMPI関数の紹介

Collective Communication (MPI\_Bcast) の実行例 : bcast.c

ポンチ絵省略 (pp.100-129)

# 基本的なMPI関数の紹介

Collective Communication (MPI\_Bcast) の実行例：bcast.c



# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Bcast) の実行例：bcast.c

使用するMPI関数の説明（MPI3.1の規格書の第5章の記述の要約）

引数の文字の色は、引数の属性（**入力**、**出力**、**入出力**）を表す。

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

コミュニケータ「comm」内に存在する全てのプロセスがこの関数を実行することにより、「comm」内の全てのプロセスのそれぞれの、メモリ空間内のアドレス「buffer」から始まる場所に存在する「datatype」型のデータ「count」個が、ランク番号「root」のプロセスでの値で書き換えられる。

# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Bcast) の実行例：bcast.c

ソースプログラムファイル「bcast.c」の内容

各プロセスが、それぞれ独立に、以下の命令列を実行する。

**argv[0]**: プログラム名

**sizeW**: 「MPI\_COMM\_WORLD」内のプロセス数

**rankW**: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

```
19 int nd, rootW, i;  
20 int *bcstbuf;  
21 nd=1;  
22 bcstbuf=(int *)malloc(sizeof(int)*nd);
```

```
23 bcstbuf[0]=rankW*10; <--・配列要素「bcstbuf[0]」に、変数「rankW」の値を10倍した値を格納する。  
24 fprintf(fp0, "\n");  
25 for(i=0; i<nd; i++){  
26     fprintf(fp0, " index=%2d  bcst=%02d\n", i, bcstbuf[i]);  
27 }      ↑--・配列要素番号「i」と配列要素「bcstbuf[i]」の値を、「index」と「bcst」の値として、「fp0」に書き込む。  
          (ここでは、配列要素数「nd」は「1」)  
28 rootW=0;  
29 MPI_Bcast(bcstbuf, nd, MPI_INT, rootW, MPI_COMM_WORLD);  
          ↑--・「MPI_COMM_WORLD」内の全プロセスの配列要素「bcstbuf[0]」の内容を、  
          ランク番号が「rootW (==0)」のプロセスのもので上書きする。  
30 fprintf(fp0, "\n");  
31 for(i=0; i<nd; i++){  
32     fprintf(fp0, " index=%2d  bcst=%02d\n", i, bcstbuf[i]);  
33 }      ↑--・配列要素番号「i」と配列要素「bcstbuf[i]」の値を、「index」と「bcst」の値として、「fp0」に書き込む。  
          (ここでは、配列要素数「nd」は「1」)
```

・茶色の文字は実際のソースプログラムファイル中には実在しない文字。  
・白黒反転表示の文は、MPIプログラムに固有の関数等を含む文。  
・19行目より前は「world.c」と全く同じ。

# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Bcast) の実行例：bcast.c

ログインノードの画面表示

各プロセスの出力内容 (2ノード4プロセスで実行した場合)

```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$ mpicc -o bcast bcast.c <-----・ C言語で書かれたMPIプログラムのコンパイルとリンクをした。
$ sbatch -p testq -t 00:01:00 exec.sh bcast <--・ ジョブを投入した。
Submitted batch job 93354 <-----・ ジョブ番号が「93354」のジョブとして受け付けられたとの表示
$
$ head -n 8 exec.sh_93354.stderr | tail -n 1
+ mpiexec -n 4 --machinefile exec.sh_93354.machinefile bash numa_bind_exec.sh 2 . bcast
$
$          ↑--・ 2ノード4プロセスで、ロードモジュールファイル「bcast」内の命令列を実行した。
$          その際、各プロセスを、それぞれ、別々のNUMAノードにバインドした。
$
$ ls -m *.txt <-----・ 各プロセスより、ファイル名が
bcast_0.txt, bcast_1.txt, bcast_2.txt, bcast_3.txt    「ロードモジュールファイル名_ランク番号.txt」
$          の形式のファイルが出力されたことを確認した。
$ ↓--・ 出力されたファイルの内容を横並びに併記する。
$ paste -d @ bcast_[0-3].txt | sed 's/@/ /g'
rankW= 0 sizeW= 4      rankW= 1 sizeW= 4      rankW= 2 sizeW= 4      rankW= 3 sizeW= 4
index= 0  bcst=00      index= 0  bcst=10      index= 0  bcst=20      index= 0  bcst=30
-----↓-----↓-----↓-----↓----- MPI_Bcast
index= 0  bcst=00      index= 0  bcst=00      index= 0  bcst=00      index= 0  bcst=00
$
$ 配列要素「bcastbuf[0]」の中身が書き換わった（送信データ格納用配列と、受信データ格納用配列は、別々には存在しない）。
```



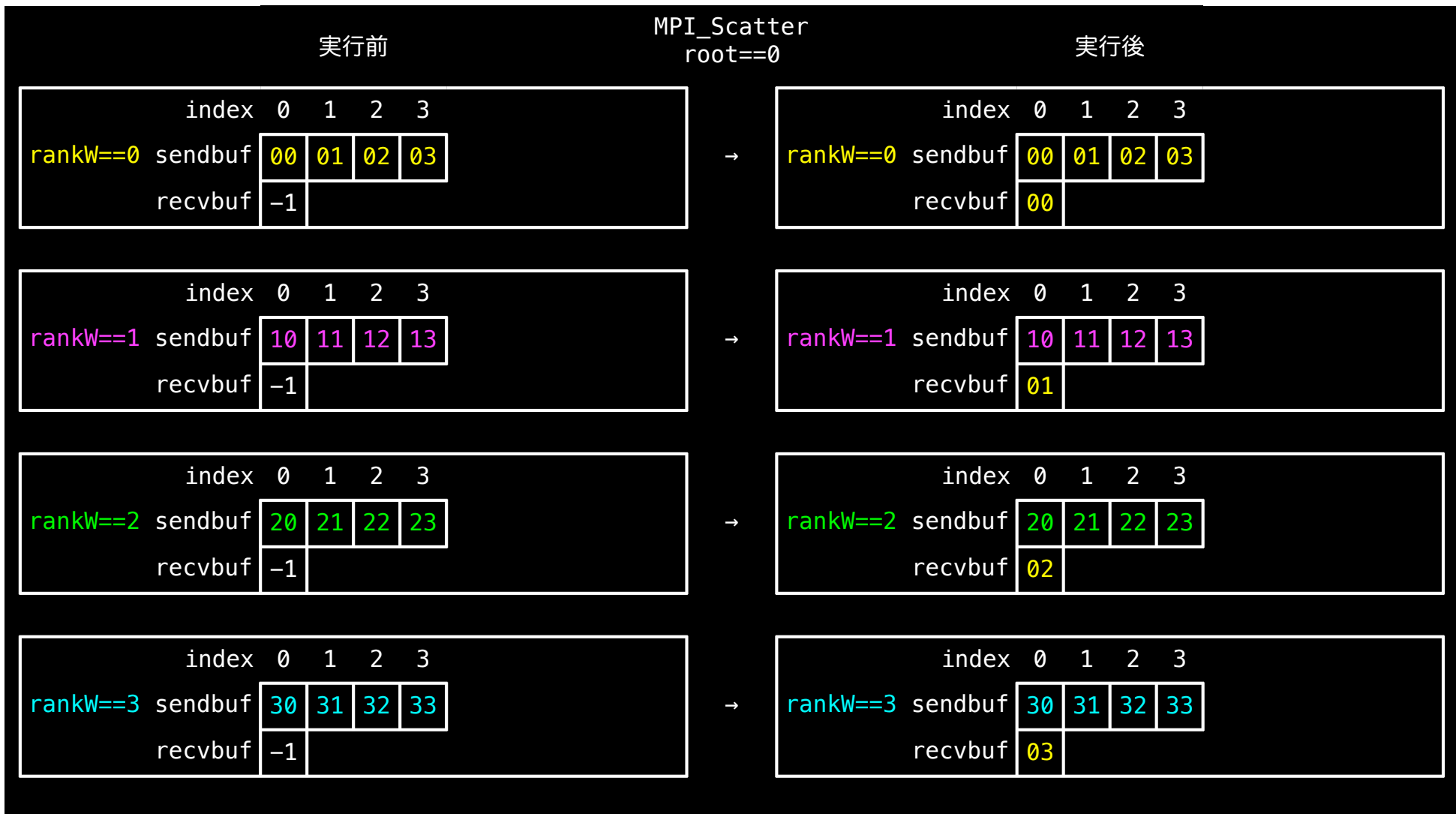
# 基本的なMPI関数の紹介

Collective Communication (MPI\_Gather) の実行例：gather.c

既出につき省略 (pp.54-57)

# 基本的なMPI関数の紹介

Collective Communication (MPI\_Scatter) の実行例：scatter.c



関数「MPI\_Scatter」は、関数「MPI\_Gather」の逆の動作をする。

# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Scatter) の実行例：scatter.c

使用するMPI関数の説明（MPI3.1の規格書の第5章の記述の要約）

引数の文字の色は、引数の属性（**入力**、**出力**、**入出力**）を表す。

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

コミュニケータ「comm」内に存在する全てのプロセスがこの関数を実行することにより、ランク番号「root」のプロセスのメモリ空間内のアドレス「sendbuf」から始まる場所に存在する「sendtype」型のデータ複数個が、「comm」内のプロセス数等分され、「comm」内の各プロセスのメモリ空間内のアドレス「recvbuf」から始まる場所に配分される。

変数「sendcount」は、各プロセスに送られるデータの個数を表す。

変数「recvcount」は、データ型が「recvtype」型の配列「recvbuf」の要素数を表す。

# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Scatter) の実行例：scatter.c

ソースプログラムファイル「scatter.c」の内容

各プロセスが、それぞれ独立に、以下の命令列を実行する。

**argv[0]**: プログラム名

**sizeW**: 「MPI\_COMM\_WORLD」内のプロセス数

**rankW**: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

```
19  int nd,rootW,i;
20  int *sendbuf,*recvbuf;
21  nd=1;
22  sendbuf=(int *)malloc(sizeof(int)*nd*sizeW);
23  recvbuf=(int *)malloc(sizeof(int)*nd);
24  for(i=0;i<nd*sizeW;i++){
25      sendbuf[i]=rankW*10+i; <--- ・配列「sendbuf」の各要素に、現行プロセスの「MPI_COMM_WORLD」内での
26  }                               ランク番号を10倍した値と、配列要素番号「i」の値を合計した値を格納する。
27  recvbuf[0]=-1; <----- ・配列要素「recvbuf[0]」に、値「-1」を格納する。
28  fprintf(fp0,"\n");
29  for(i=0;i<nd*sizeW;i++){
30      fprintf(fp0," index=%2d  send=%02d\n",i,sendbuf[i]);
31  }      ↑-- ・配列要素番号「i」と配列要素「sendbuf[i]」の内容を、「index」と「send」の値として、「fp0」に書き込む。
32  rootW=0;
33  MPI_Scatter(sendbuf,nd,MPI_INT,recvbuf,nd,MPI_INT,rootW,MPI_COMM_WORLD);
      ↑-- ・ランク番号「root(==0)」のプロセスの、メモリ空間内のアドレス「sendbuf」から始まる場所に存在する「MPI_INT」型の
          データ複数(==nd*sizeW)個を、コミュニケータ「MPI_COMM_WORLD」内のプロセスの数「sizeW」に等分し、
          コミュニケータ「MPI_COMM_WORLD」内の各プロセスのメモリ空間内のアドレス「recvbuf」から始まる場所に配分する。
34  fprintf(fp0,"\n");
35  for(i=0;i<nd;i++){
36      fprintf(fp0," index=%2d  recv=%02d\n",i,recvbuf[i]);
37  }      ↑-- ・配列要素番号「i」と配列要素「recvbuf[i]」の内容を、「index」と「recv」の値として、「fp0」に書き込む。
```

・茶色の文字は実際のソースプログラムファイル中には実在しない文字。  
・白黒反転表示の文は、MPIプログラムに固有の関数等を含む文。  
・19行目より前は「world.c」と全く同じ。

# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Scatter) の実行例：scatter.c

□グインノードの画面表示

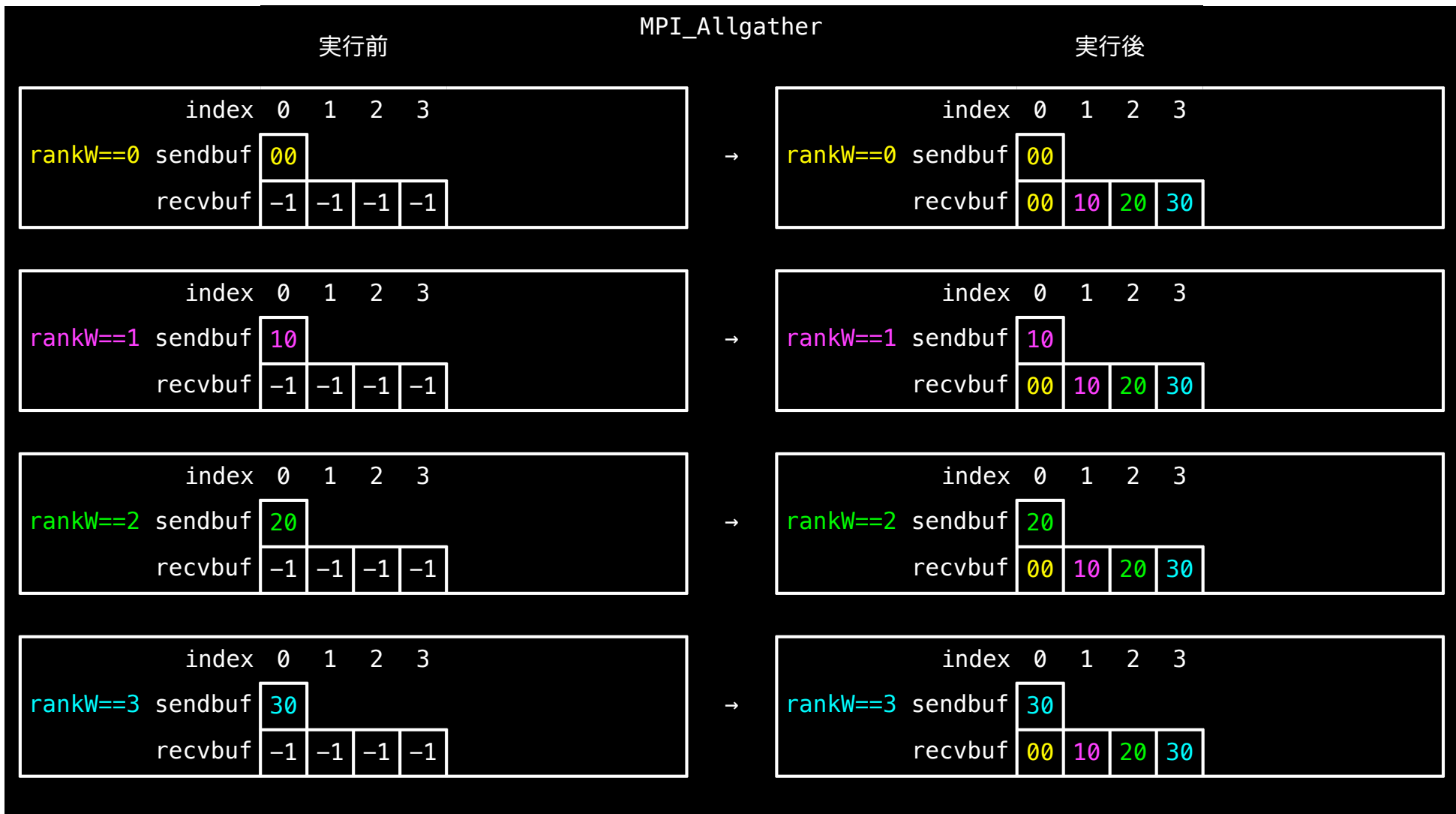
各プロセスの出力内容 (2ノード4プロセスで実行した場合)

```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$ mpicc -o scatter scatter.c <----- ・ C言語で書かれたMPIプログラムのコンパイルとリンクをした。
$ sbatch -p testq -t 00:01:00 exec.sh scatter <-- ・ ジョブを投入した。
Submitted batch job 93368 <----- ・ ジョブ番号が「93368」のジョブとして受け付けられたとの表示
$
$ head -n 8 exec.sh_93368.stderr | tail -n 1
+ mpiexec -n 4 --machinefile exec.sh_93368.machinefile bash numa_bind_exec.sh 2 . scatter
$                               ↑-- ・ 2ノード4プロセスで、ロードモジュールファイル「scatter」内の命令列を実行した。
$                               その際、各プロセスを、それぞれ、別々のNUMAノードにバインドした。
$
$ ls -m *.txt <----- ・ 各プロセスより、ファイル名が
scatter_0.txt, scatter_1.txt, scatter_2.txt, scatter_3.txt 「ロードモジュールファイル名_ランク番号.txt」
$                               の形式のファイルが出力されたことを確認した。
$ ↓-- ・ 出力されたファイルの内容を横並びに併記する。
$ paste -d @ scatter_[0-3].txt | sed 's/@/ /g'
rankW= 0 sizeW= 4      rankW= 1 sizeW= 4      rankW= 2 sizeW= 4      rankW= 3 sizeW= 4

index= 0  send=00      index= 0  send=10      index= 0  send=20      index= 0  send=30
index= 1  send=01      index= 1  send=11      index= 1  send=21      index= 1  send=31
index= 2  send=02      index= 2  send=12      index= 2  send=22      index= 2  send=32
index= 3  send=03      index= 3  send=13      index= 3  send=23      index= 3  send=33
----- MPI_Scatter
index= 0  recv=00      index= 0  recv=01      index= 0  recv=02      index= 0  recv=03
$
```

# 基本的なMPI関数の紹介

Collective Communication (MPI\_Allgather) の実行例：allgather.c



# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Allgather) の実行例：allgather.c

使用するMPI関数の説明（MPI3.1の規格書の第5章の記述の要約）

引数の文字の色は、引数の属性（**入力**、**出力**、**入出力**）を表す。

```
int MPI_Allgather(const void* sendbuf, int sendcount, MPI_Datatype sendtype,  
                  void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

コミュニケータ「comm」内に存在する全てのプロセスがこの関数を実行することにより、「comm」内に存在する全てのプロセスの、メモリ空間内のアドレス「sendbuf」から始まる「sendtype」型の値「sendcount」個が、「comm」内の各プロセスの、メモリ空間内のアドレス「recvbuf」から始まる場所に、ランク番号順に格納される。

変数「sendcount」は、各プロセスから送り出される「sendtype」型のデータの個数を表す。

変数「recvcount」は、各プロセスから受け取る「recvtype」型のデータの個数を表す。

# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Allgather) の実行例：allgather.c

ソースプログラムファイル「allgather.c」の内容

各プロセスが、それぞれ独立に、以下の命令列を実行する。

**argv[0]**: プログラム名

**sizeW**: 「MPI\_COMM\_WORLD」内のプロセス数

**rankW**: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

```
19  int nd,i;
20  int *sendbuf,*recvbuf;
21  nd=1;
22  sendbuf=(int *)malloc(sizeof(int)*nd);
23  recvbuf=(int *)malloc(sizeof(int)*nd*sizeW);

24  sendbuf[0]=rankW*10; <--・配列要素「sendbuf[0]」に、変数「rankW」の値を10倍した値を格納する。
25  for(i=0;i<sizeW;i++){
26      recvbuf[i]=-1; <-----・配列「recvbuf」の全要素に、値「-1」を格納する。
27  }
28  fprintf(fp0,"\n");
29  for(i=0;i<nd;i++){
30      fprintf(fp0," index=%2d  send=%02d\n",i,sendbuf[i]);
31  }      ↑--・配列要素番号「i」と配列要素「sendbuf[i]」の内容を、「index」と「send」の値として、「fp0」に書き込む。

32  MPI_Allgather(sendbuf,nd,MPI_INT,recvbuf,nd,MPI_INT,MPI_COMM_WORLD);
      ↑--・コミュニケータ「MPI_COMM_WORLD」内の全てのプロセスのそれぞれの、「MPI_INT」型の配列「sendbuf」
      の内容を、「MPI_COMM_WORLD」内の各プロセスの配列「recvbuf」に、ランク番号順に格納する。

33  fprintf(fp0,"\n");
34  for(i=0;i<nd*sizeW;i++){
35      fprintf(fp0," index=%2d  recv=%02d\n",i,recvbuf[i]);
36  }      ↑--・配列要素番号「i」と配列要素「recvbuf[i]」の内容を、「index」と「recv」の値として、「fp0」に書き込む。
```

- ・ 茶色の文字は実際のソースプログラムファイル中には実在しない文字。
- ・ 白黒反転表示の文は、MPIプログラムに固有の関数等を含む文。
- ・ 19行目より前は「world.c」と全く同じ。



# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Allgather) の実行例：allgather.c

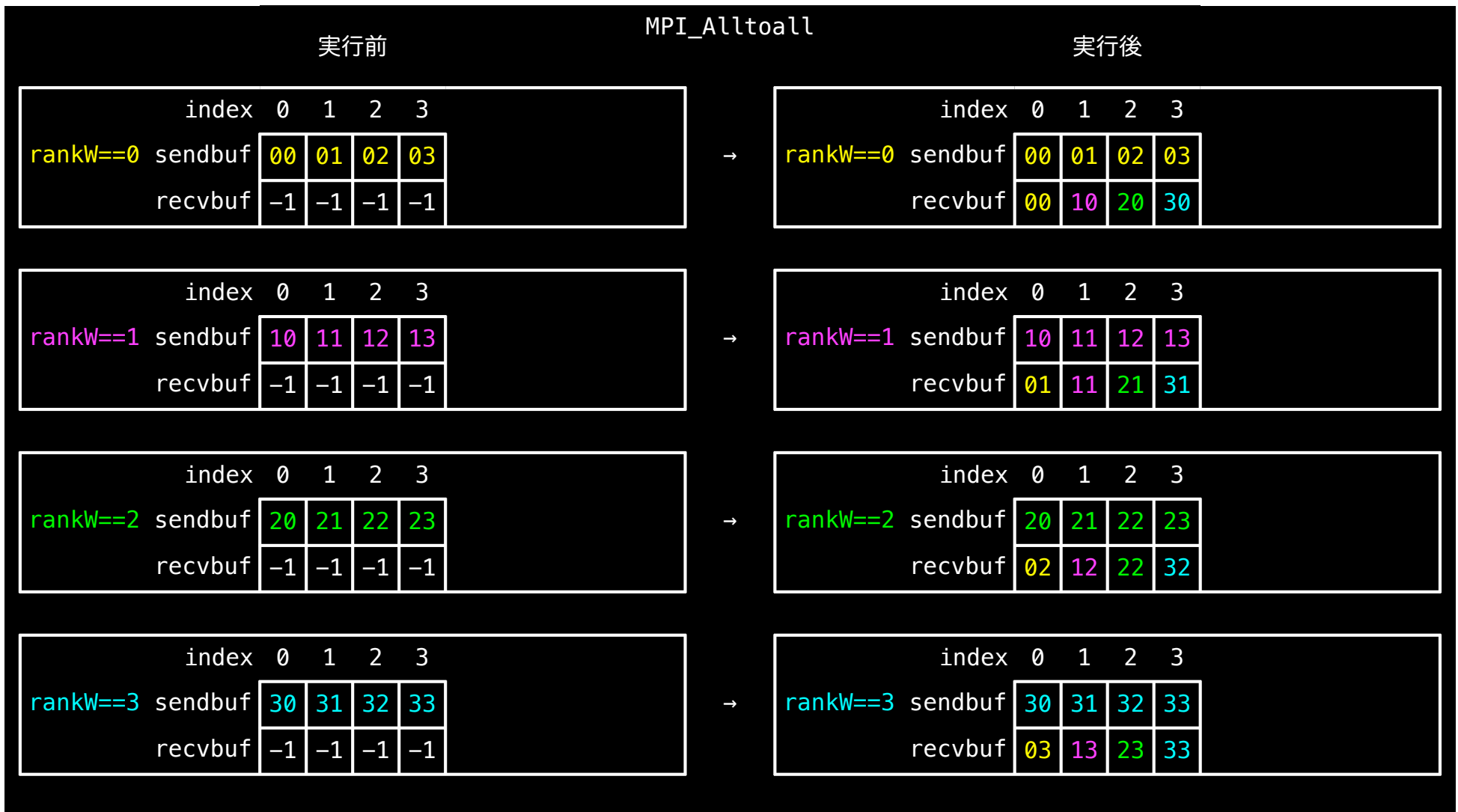
ログインノードの画面表示

各プロセスの出力内容 (2ノード4プロセスで実行した場合)

```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$ mpicc -o allgather allgather.c <----- ・ C言語で書かれたMPIプログラムのコンパイルとリンクをした。
$ sbatch -p testq -t 00:01:00 exec.sh allgather <-- ・ ジョブを投入した。
Submitted batch job 93350 <----- ・ ジョブ番号が「93350」のジョブとして受け付けられたとの表示
$
$ head -n 8 exec.sh_93350.stderr | tail -n 1
+ mpiexec -n 4 --machinefile exec.sh_93350.machinefile bash numa_bind_exec.sh 2 . allgather
$                                     ↑-- ・ 2ノード4プロセスで、ロードモジュールファイル「allgather」内の命令列を実行した。
$                                     その際、各プロセスを、それぞれ、別々のNUMAノードにバインドした。
$
$ ls -m *.txt <----- ・ 各プロセスより、ファイル名が
allgather_0.txt, allgather_1.txt, allgather_2.txt, allgather_3.txt 「ロードモジュールファイル名_ランク番号.txt」
$                                     の形式のファイルが出力されたことを確認した。
$ ↓-- ・ 出力されたファイルの内容を横並びに併記する。
$ paste -d @ allgather_[0-3].txt | sed 's/@/      /g'
rankW= 0 sizeW= 4      rankW= 1 sizeW= 4      rankW= 2 sizeW= 4      rankW= 3 sizeW= 4
index= 0  send=00      index= 0  send=10      index= 0  send=20      index= 0  send=30
----- MPI_Allgather
index= 0  recv=00      index= 0  recv=00      index= 0  recv=00      index= 0  recv=00
index= 1  recv=10      index= 1  recv=10      index= 1  recv=10      index= 1  recv=10
index= 2  recv=20      index= 2  recv=20      index= 2  recv=20      index= 2  recv=20
index= 3  recv=30      index= 3  recv=30      index= 3  recv=30      index= 3  recv=30
$
```

# 基本的なMPI関数の紹介

Collective Communication (MPI\_Alltoall) の実行例：alltoall.c



4行4列の行列Aの各行が、4つのプロセスに分散配置されている時に、「MPI\_Alltoall」を実行することは、行列Aの転置を行い、その結果の行列Aの各行を当該4プロセスに分散することに等しい。

# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Alltoall) の実行例：alltoall.c

使用するMPI関数の説明（MPI3.1の規格書の第5章の記述の要約）

引数の文字の色は、引数の属性（**入力**、**出力**、**入出力**）を表す。

```
int MPI_Alltoall(const void* sendbuf, int sendcount, MPI_Datatype sendtype,  
                void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

コミュニケータ「comm」内の全てのプロセスがこの関数を実行することにより、以下のことが起こる。

- ・ 各プロセスが、それぞれのメモリ空間内のアドレス「sendbuf」から始まる場所に存在する「sendtype」型のデータを複数個を、コミュニケータ「comm」内のプロセス数と同数のブロックに等分し、コミュニケータ「comm」内の全プロセスに配分（送信）する。最初のブロックは、コミュニケータ「comm」内でのランク番号が「0」のプロセスに、次のブロックはランク番号が「1」のプロセスに、その次のブロックはランク番号が「2」のプロセスに、…。ここで、変数「sendcount」は、各プロセスに送られるデータの個数。つまり、上記の「複数個」とは、「sendcount」の値と、プロセスの数を掛け合わせた数のこと。
- ・ 各プロセスが、コミュニケータ「comm」内の全プロセスのそれぞれから、「recvtype」型のデータを「recvcount」個受け取り、メモリ空間内のアドレス「recvbuf」から始まる場所に格納する。格納する順番は、コミュニケータ「comm」内でのランク番号の順。

# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Alltoall) の実行例：alltoall.c

ソースプログラムファイル「alltoall.c」の内容

各プロセスが、それぞれ独立に、以下の命令列を実行する。

**argv[0]**: プログラム名

**sizeW**: 「MPI\_COMM\_WORLD」内のプロセス数

**rankW**: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

```
19  int nd,i;
20  int *sendbuf,*recvbuf;
21  nd=1;
22  sendbuf=(int *)malloc(sizeof(int)*nd*sizeW);
23  recvbuf=(int *)malloc(sizeof(int)*nd*sizeW);
24  for(i=0;i<sizeW;i++){
25      sendbuf[i]=rankW*10+i; <--- ・配列「sendbuf」の各要素に、現行プロセスの「MPI_COMM_WORLD」内での
                                   ランク番号を10倍した値と、配列要素番号「i」の値を合計した値を格納する。
26      recvbuf[i]=-1; <----- ・配列「recvbuf」の各要素に、値「-1」を格納する。
27  }
28  fprintf(fp0,"\n");
29  for(i=0;i<nd*sizeW;i++){
30      fprintf(fp0," index=%2d  send=%02d\n",i,sendbuf[i]);
31  }      ↑-- ・配列要素番号「i」と配列要素「sendbuf[i)」の内容を、「index」と「send」の値として、「fp0」に書き込む。

32  MPI_Alltoall(sendbuf,nd,MPI_INT,recvbuf,nd,MPI_INT,MPI_COMM_WORLD);
      ↑-- ・「MPI_INT」型の配列「sendbuf」を、コミュニケータ「MPI_COMM_WORLD」内のプロセス数と同数のブロックに
           等分し、コミュニケータ「comm」内の全プロセスに配分（送信）する。ここで、変数「nd(==1)」は、各プロセス
           に送られるデータの個数。同時に、コミュニケータ「MPI_COMM_WORLD」内の全プロセスのそれぞれから、
           「MPI_INT」型のデータを「nd(==1)」個受け取り、メモリ空間内のアドレス「recvbuf」から始まる場所に格納
           する。格納する順番はランク番号の順。

33  fprintf(fp0,"\n");
34  for(i=0;i<nd*sizeW;i++){
35      fprintf(fp0," index=%2d  recv=%02d\n",i,recvbuf[i]);
36  }      ↑-- ・配列要素番号「i」と配列要素「recvbuf[i)」の内容を、「index」と「recv」の値として、「fp0」に書き込む。
```

- ・ 茶色の文字は実際のソースプログラムファイル中には実在しない文字。
- ・ 白黒反転表示の文は、MPIプログラムに固有の関数等を含む文。
- ・ 19行目より前は「world.c」と全く同じ。

# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Alltoall) の実行例：alltoall.c

ログインノードの画面表示

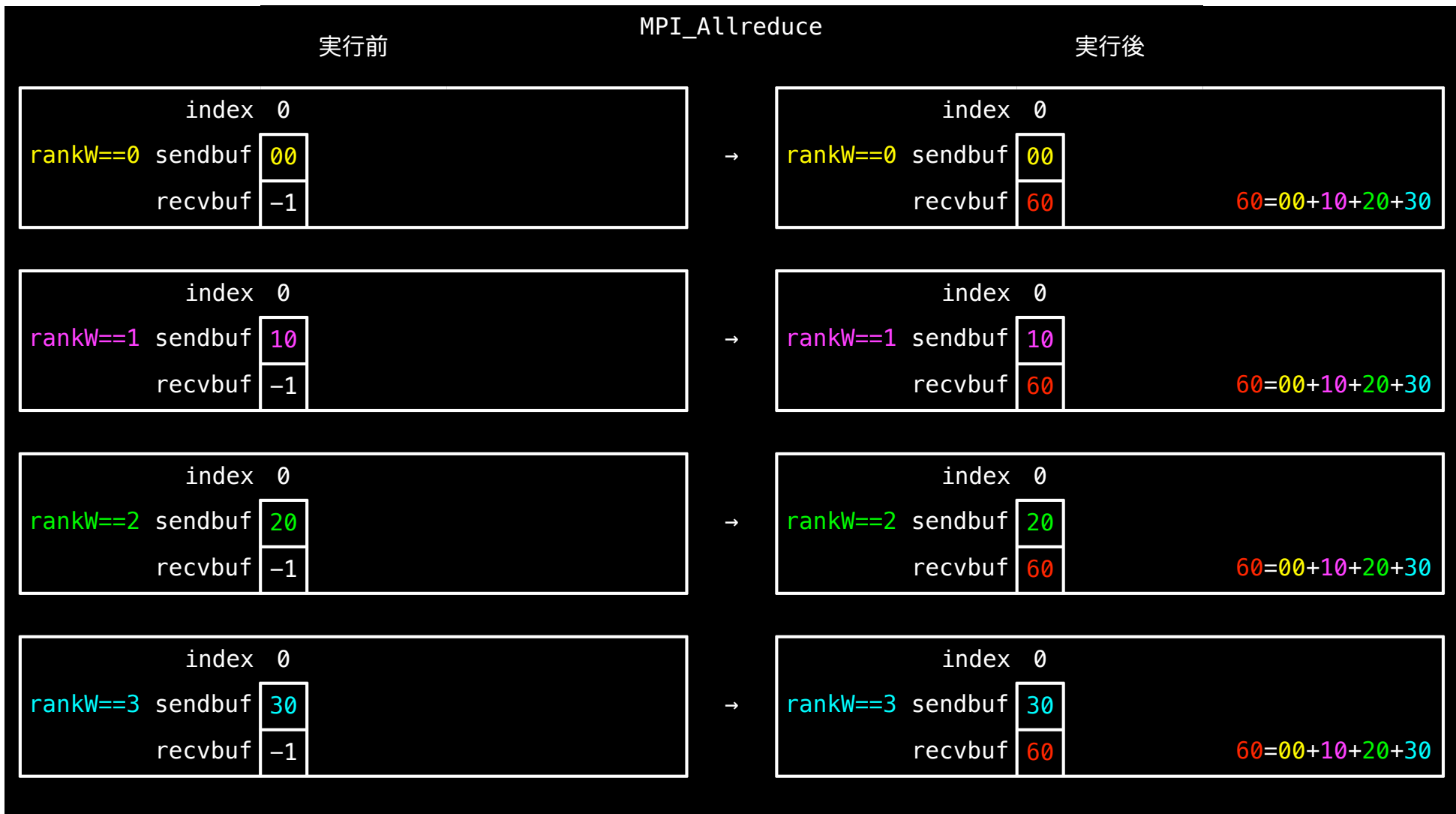
各プロセスの出力内容 (2ノード4プロセスで実行した場合)

```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$ mpicc -o alltoall alltoall.c <----- ・ C言語で書かれたMPIプログラムのコンパイルとリンクをした。
$ sbatch -p testq -t 00:01:00 exec.sh alltoall <-- ・ ジョブを投入した。
Submitted batch job 93352 <----- ・ ジョブ番号が「93352」のジョブとして受け付けられたとの表示
$
$ head -n 8 exec.sh_93352.stderr | tail -n 1
+ mpiexec -n 4 --machinefile exec.sh_93352.machinefile bash numa_bind_exec.sh 2 . alltoall
$                                     ↑-- ・ 2ノード4プロセスで、ロードモジュールファイル「alltoall」内の命令列を実行した。
$                                     その際、各プロセスを、それぞれ、別々のNUMAノードにバインドした。
$
$ ls -m *.txt <----- ・ 各プロセスより、ファイル名が
alltoall_0.txt, alltoall_1.txt, alltoall_2.txt, alltoall_3.txt 「ロードモジュールファイル名_ランク番号.txt」
$                                     の形式のファイルが出力されたことを確認した。
$ ↓-- ・ 出力されたファイルの内容を横並びに併記する。
$ paste -d @ alltoall_[0-3].txt | sed 's/@/ /g'
rankW= 0 sizeW= 4      rankW= 1 sizeW= 4      rankW= 2 sizeW= 4      rankW= 3 sizeW= 4

index= 0 send=00      index= 0 send=10      index= 0 send=20      index= 0 send=30
index= 1 send=01      index= 1 send=11      index= 1 send=21      index= 1 send=31
index= 2 send=02      index= 2 send=12      index= 2 send=22      index= 2 send=32
index= 3 send=03      index= 3 send=13      index= 3 send=23      index= 3 send=33
----- MPI_Alltoall
index= 0 recv=00      index= 0 recv=01      index= 0 recv=02      index= 0 recv=03
index= 1 recv=10      index= 1 recv=11      index= 1 recv=12      index= 1 recv=13
index= 2 recv=20      index= 2 recv=21      index= 2 recv=22      index= 2 recv=23
index= 3 recv=30      index= 3 recv=31      index= 3 recv=32      index= 3 recv=33
$
```

# 基本的なMPI関数の紹介

Collective Communication (MPI\_Allreduce) の実行例：allreduce.c



# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Allreduce) の実行例：allreduce.c

使用するMPI関数の説明（MPI3.1の規格書の第5章の記述の要約）

引数の文字の色は、引数の属性（**入力**、**出力**、**入出力**）を表す。

```
int MPI_Allreduce(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype,
                  MPI_Op op, MPI_Comm comm)
```

コミュニケータ「comm」内に存在する全てのプロセスがこの関数を実行することにより、「comm」内に存在する全てのプロセス間で、「datatype」型の要素数「count」の配列「sendbuf」の要素番号が同じ要素同士の演算「op」が行われ、その結果が全てのプロセスの「datatype」型の配列「recvbuf」に格納される。

「op」で指定できる演算（reduction演算）として既定のものは以下の通り。

MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BLAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical exclusive or
MPI_BXOR	bit-wise exclusive or
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

但し、その演算順序は不定。

→ 浮動小数点数についての総和「sum」などの場合には、結果の厳密な一致は期待できない。  
（同じ計算でも、実行環境により結果が変わる）

# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Allreduce) の実行例：allreduce.c

ソースプログラムファイル「allreduce.c」の内容

各プロセスが、それぞれ独立に、以下の命令列を実行する。

**argv[0]**: プログラム名

**sizeW**: 「MPI\_COMM\_WORLD」内のプロセス数

**rankW**: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

```
19  int nd,i;
20  int *sendbuf,*recvbuf;
21  nd=1;
22  sendbuf=(int *)malloc(sizeof(int)*nd);
23  recvbuf=(int *)malloc(sizeof(int)*nd);
24  sendbuf[0]=rankW*10; <----- ・ 配列要素「sendbuf[0]」に、変数「rankW」の値を10倍した値を格納する。
25  recvbuf[0]=-1; <----- ・ 配列要素「recvbuf[0]」に、値「-1」を格納する。
26  fprintf(fp0,"\n");
27  for(i=0;i<nd;i++){
28      fprintf(fp0," index=%2d  send=%02d\n",i,sendbuf[i]);
29  }      ↑-- ・ 配列要素番号「i」と配列要素「sendbuf[i]」の内容を、「index」と「send」の値として、「fp0」に書き込む。

30  MPI_Allreduce(sendbuf,recvbuf,nd,MPI_INT,MPI_SUM,MPI_COMM_WORLD);
      ↑-- ・ コミュニケータ「MPI_COMM_WORLD」内に存在する全てのプロセス間で、「MPI_INT」型の
          要素数「nd(==1)」の配列「sendbuf」の要素番号が同じ要素同士の（ここでは要素番号「0」のみの）、
          「MPI_SUM」（総和）演算を行い、その結果を配列「recvbuf」に格納する。

31  fprintf(fp0,"\n");
32  for(i=0;i<nd;i++){
33      fprintf(fp0," index=%2d  recv=%02d\n",i,recvbuf[i]);
34  }      ↑-- ・ 配列要素番号「i」と配列要素「recvbuf[i]」の内容を、「index」と「recv」の値として、「fp0」に書き込む。
```

- ・ 茶色の文字は実際のソースプログラムファイル中には実在しない文字。
- ・ 白黒反転表示の文は、MPIプログラムに固有の関数等を含む文。
- ・ 19行目より前は「world.c」と全く同じ。



# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Allreduce) の実行例：allreduce.c

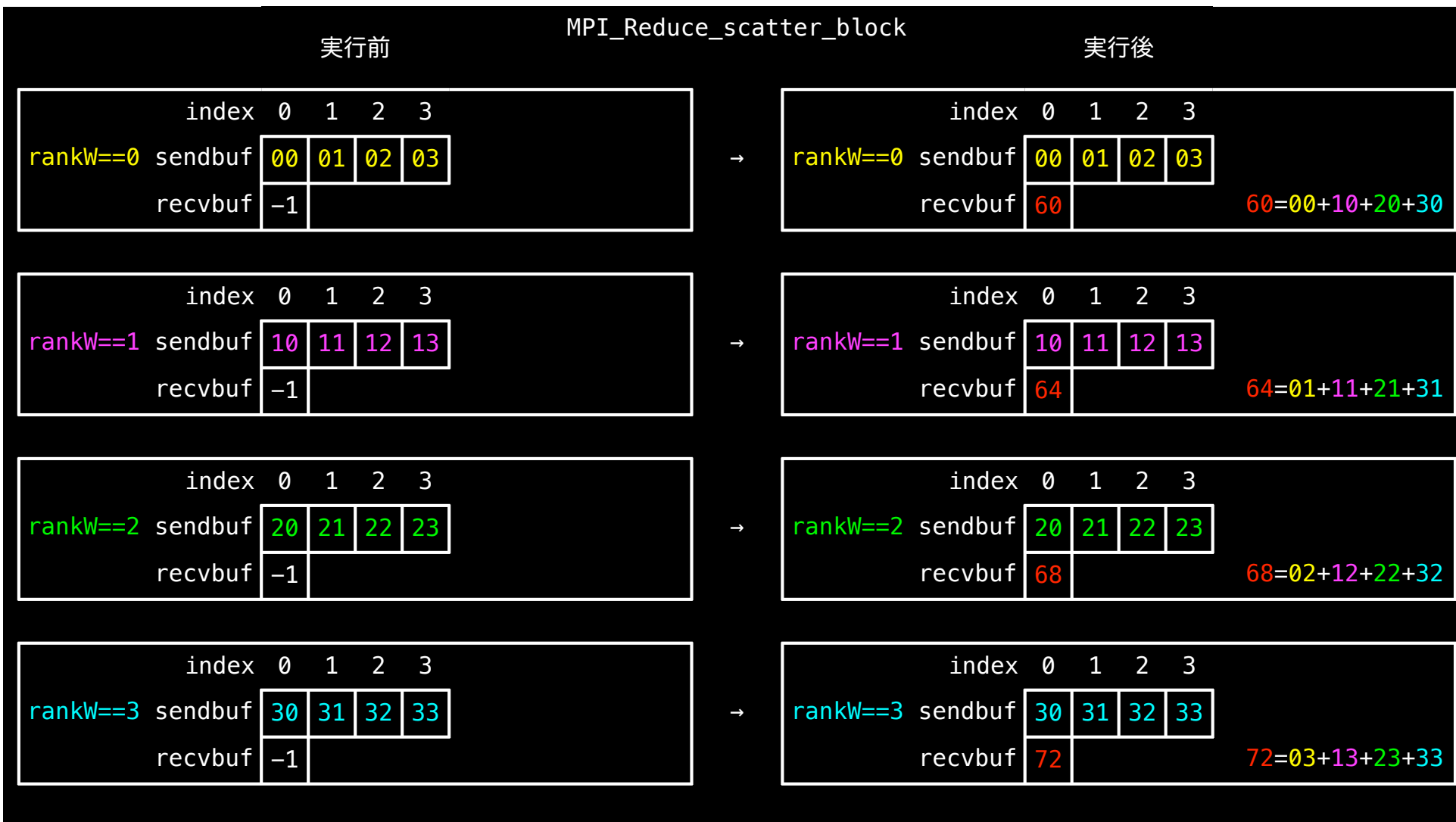
ログインノードの画面表示

各プロセスの出力内容 (2ノード4プロセスで実行した場合)

```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$ mpicc -o allreduce allreduce.c <----- ・ C言語で書かれたMPIプログラムのコンパイルとリンクをした。
$ sbatch -p testq -t 00:01:00 exec.sh allreduce <-- ・ ジョブを投入した。
Submitted batch job 93351 <----- ・ ジョブ番号が「93351」のジョブとして受け付けられたとの表示
$
$ head -n 8 exec.sh_93351.stderr | tail -n 1
+ mpiexec -n 4 --machinefile exec.sh_93351.machinefile bash numa_bind_exec.sh 2 . allreduce
$                               ↑-- ・ 2ノード4プロセスで、ロードモジュールファイル「allreduce」内の命令列を実行した。
$                               その際、各プロセスを、それぞれ、別々のNUMAノードにバインドした。
$
$ ls -m *.txt <----- ・ 各プロセスより、ファイル名が
allreduce_0.txt, allreduce_1.txt, allreduce_2.txt, allreduce_3.txt 「ロードモジュールファイル名_ランク番号.txt」
$                               の形式のファイルが出力されたことを確認した。
$ ↓-- ・ 出力されたファイルの内容を横並びに併記する。
$ paste -d @ allreduce_[0-3].txt | sed 's/@/      /g'
rankW= 0 sizeW= 4      rankW= 1 sizeW= 4      rankW= 2 sizeW= 4      rankW= 3 sizeW= 4
index= 0 send=00      index= 0 send=10      index= 0 send=20      index= 0 send=30
----- MPI_Allreduce
index= 0 recv=60      index= 0 recv=60      index= 0 recv=60      index= 0 recv=60
$
```

# 基本的なMPI関数の紹介

Collective Communication (MPI\_Reduce\_scatter\_block) の実行例：reduce\_scatter\_block.c



4行4列の行列Aの各列が、4つのプロセスに分散配置されている時に、ベクトルxとの積Axを行い、その結果のベクトルを4等分して、当該4プロセスに分散配置しようとする場合、まず各プロセスが、それぞれ保持している列のみについての行列ベクトル積を行い（その結果が図中の実行前の状態）、その後「MPI\_Reduce\_scatter\_block」を実行することにより、行列ベクトル積Axを実現することが出来る。

# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Reduce\_scatter\_block) の実行例：reduce\_scatter\_block.c

使用するMPI関数の説明（MPI3.1の規格書の第5章の記述の要約）

引数の文字の色は、引数の属性（**入力**、**出力**、**入出力**）を表す。

```
int MPI_Reduce_scatter_block(const void* sendbuf, void* recvbuf, int recvcount, MPI_Datatype datatype,  
                             MPI_Op op, MPI_Comm comm)
```

コミュニケータ「comm」内に存在する全てのプロセスがこの関数を実行することにより、「comm」内に存在する全てのプロセス間で、「sendtype」型の要素数「recvcount」の配列「sendbuf」の要素番号が同じ要素同士の演算「op」が行われる。その結果の配列が、コミュニケータ「comm」内のプロセスの数に等分され、各プロセスに配分される。等分された、最初のブロックをランク番号が「0」のプロセスの配列「recvbuf」に、次のブロックをランク番号が「1」のプロセスの配列「recvbuf」に、その次のブロックをランク番号が「2」のプロセスの…に格納される。

「op」で指定できる演算（reduction演算）として既定のものは、「MPI\_Allreduce」（118ページを参照）と同じ。

# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Reduce\_scatter\_block) の実行例：reduce\_scatter\_block.c

ソースプログラムファイル「reduce\_scatter\_block.c」の内容  
各プロセスが、それぞれ独立に、以下の命令列を実行する。

**argv[0]**: プログラム名  
**sizeW**: 「MPI\_COMM\_WORLD」内のプロセス数  
**rankW**: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

```
19  int nd,i;
20  int *sendbuf,*recvbuf;
21  nd=1;
22  sendbuf=(int *)malloc(sizeof(int)*nd*sizeW);
23  recvbuf=(int *)malloc(sizeof(int)*nd);
24  for(i=0;i<nd*sizeW;i++){
25      sendbuf[i]=rankW*10+i; <--- ・配列「sendbuf」の各要素に、現行プロセスの「MPI_COMM_WORLD」内での
26  }                               ランク番号を10倍した値と、配列要素番号「i」の値を合計した値を格納する。
27  recvbuf[0]=-1; <----- ・配列要素「recvbuf[0]」に、値「-1」を格納する。
28  fprintf(fp0,"\n");
29  for(i=0;i<nd*sizeW;i++){
30      fprintf(fp0," index=%2d  send=%02d\n",i,sendbuf[i]);
31  }      ↑-- ・配列要素番号「i」と配列要素「sendbuf[i]」の内容を、「index」と「send」の値として、「fp0」に書き込む。

32  MPI_Reduce_scatter_block(sendbuf,recvbuf,nd,MPI_INT,MPI_SUM,MPI_COMM_WORLD);
      ↑-- ・コミュニケータ「MPI_COMM_WORLD」内に存在する全てのプロセス間で、「MPI_INT」型の
          要素数「nd(==1)」の配列「sendbuf」の要素番号が同じ要素同士の（ここでは「0」のみの）
          「MPI_SUM」（総和）演算を行う。その結果の配列を、コミュニケータ「MPI_COMM_WORLD」
          内のプロセスの数に等分し、各プロセスに配分する。等分された、最初のブロックをランク
          番号が「0」のプロセスの配列「recvbuf」に、次のブロックをランク番号が「1」のプロセス
          の配列「recvbuf」に、その次のブロックをランク番号が「2」のプロセスの...に格納する。

33  fprintf(fp0,"\n");
34  for(i=0;i<nd;i++){
35      fprintf(fp0," index=%2d  recv=%02d\n",i,recvbuf[i]);
36  }      ↑-- ・配列要素番号「i」と配列要素「recvbuf[i]」の内容を、「index」と「recv」の値として、「fp0」に書き込む。
```

・ 茶色の文字は実際のソースプログラムファイル中には実在しない文字。  
・ 白黒反転表示の文は、MPIプログラムに固有の関数等を含む文。  
・ 19行目より前は「world.c」と全く同じ。

# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Reduce\_scatter\_block) の実行例：reduce\_scatter\_block.c

ログインノードの画面表示

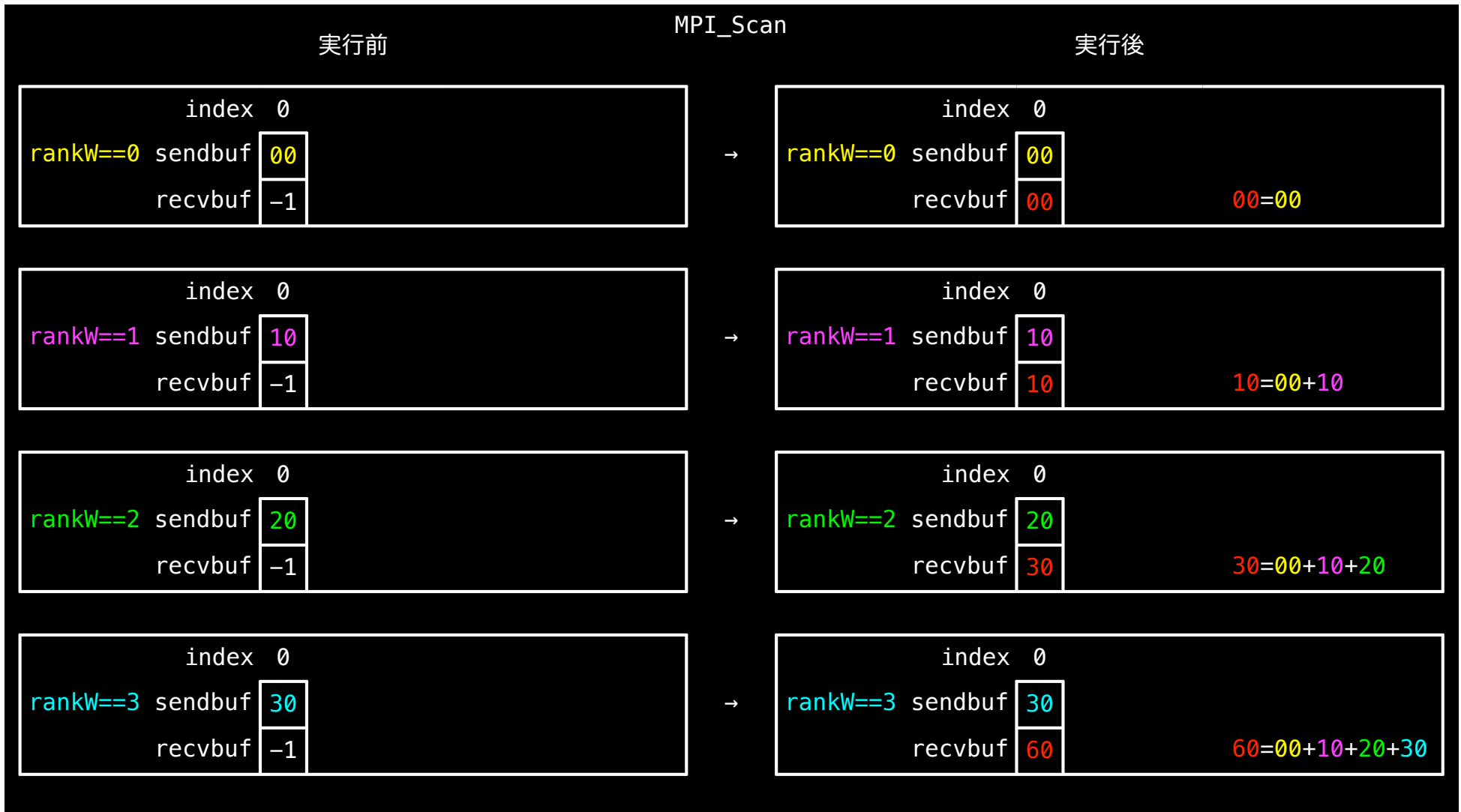
各プロセスの出力内容 (2ノード4プロセスで実行した場合)

```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$ mpicc -o reduce_scatter_block reduce_scatter_block.c <----- ・ C言語で書かれたMPIプログラムの
$                                     コンパイルとリンクをした。
$ sbatch -p testq -t 00:01:00 exec.sh reduce_scatter_block <-- ・ ジョブを投入した。
Submitted batch job 93365 <----- ・ ジョブ番号が「93365」のジョブとして
$                                     受け付けられたとの表示
$
$ head -n 8 exec.sh_93365.stderr | tail -n 1
+ mpiexec -n 4 --machinefile exec.sh_93365.machinefile bash numa_bind_exec.sh 2 . reduce_scatter_block
$                                     ↑-- ・ 2ノード4プロセスで、ロードモジュールファイル
$                                     「reduce_scatter_block」内の命令列を実行した。
$                                     その際、各プロセスを、それぞれ、別々のNUMAノードにバインドした。
$
$                                     ・ 各プロセスより、ファイル名が「ロードモジュールファイル名_ランク番号.txt」の形式の
$                                     ファイルが出力されたことを確認した。
$ ls -m *.txt <-----
reduce_scatter_block_0.txt, reduce_scatter_block_1.txt, reduce_scatter_block_2.txt,
reduce_scatter_block_3.txt
$
$ ↓-- ・ 出力されたファイルの内容を横並びに併記する。
$ paste -d @ reduce_scatter_block_[0-3].txt | sed 's/@/      /g'
rankW= 0 sizeW= 4      rankW= 1 sizeW= 4      rankW= 2 sizeW= 4      rankW= 3 sizeW= 4

index= 0 send=00      index= 0 send=10      index= 0 send=20      index= 0 send=30
index= 1 send=01      index= 1 send=11      index= 1 send=21      index= 1 send=31
index= 2 send=02      index= 2 send=12      index= 2 send=22      index= 2 send=32
index= 3 send=03      index= 3 send=13      index= 3 send=23      index= 3 send=33
-----
index= 0 recv=60      index= 0 recv=64      index= 0 recv=68      index= 0 recv=72      MPI_Reduce
                                                                    _scatter_block
$
```

# 基本的なMPI関数の紹介

Collective Communication (MPI\_Scan) の実行例：scan.c



# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Scan) の実行例：scan.c

使用するMPI関数の説明（MPI3.1の規格書の第5章の記述の要約）

引数の文字の色は、引数の属性（**入力**、**出力**、**入出力**）を表す。

```
int MPI_Scan(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype,  
             MPI_Op op, MPI_Comm comm)
```

コミュニケータ「comm」内に存在する全てのプロセスがこの関数を実行することにより、「comm」内に存在する全てのプロセスの内、各プロセス及びそのプロセスよりもランク番号が小さいプロセスの間で、「sendtype」型の要素数「count」の配列「sendbuf」の要素番号が同じ要素同士の演算「op」が行われ、その結果が各プロセスの配列「recvbuf」に格納される。

「op」で指定できる演算（reduction演算）として既定のものは、「MPI\_Allreduce」（118ページを参照）と同じ。

# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Scan) の実行例：scan.c

ソースプログラムファイル「scan.c」の内容

各プロセスが、それぞれ独立に、以下の命令列を実行する。

**argv[0]**: プログラム名

**sizeW**: 「MPI\_COMM\_WORLD」内のプロセス数

**rankW**: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

```
19  int nd,i;
20  int *sendbuf,*recvbuf;
21  nd=1;
22  sendbuf=(int *)malloc(sizeof(int)*nd);
23  recvbuf=(int *)malloc(sizeof(int)*nd);
24  sendbuf[0]=rankW*10;
25  recvbuf[0]=-1;
26  fprintf(fp0,"\n");
27  for(i=0;i<nd;i++){
28      fprintf(fp0," index=%2d  send=%02d\n",i,sendbuf[i]);
29  }      ↑--・配列要素番号「i」と配列要素「sendbuf[i]」の内容を、「index」と「send」の値として、「fp0」に書き込む。

30  MPI_Scan(sendbuf,recvbuf,nd,MPI_INT,MPI_SUM,MPI_COMM_WORLD);
      ↑--・コミュニケータ「MPI_COMM_WORLD」内に存在する全てのプロセスの内、現行プロセス及び現行プロセスよりも
          ランク番号が小さいプロセスの間で、「MPI_INT」型の要素数「nd(==1)」の配列「sendbuf」の要素番号が同じ
          要素同士の「MPI_SUM」（総和）演算を行い、その結果を現行プロセスの配列「recvbuf」に格納する。

31  fprintf(fp0,"\n");
32  for(i=0;i<nd;i++){
33      fprintf(fp0," index=%2d  recv=%02d\n",i,recvbuf[i]);
34  }      ↑--・配列要素番号「i」と配列要素「recvbuf[i]」の内容を、「index」と「recv」の値として、「fp0」に書き込む。
```

- ・ 茶色の文字は実際のソースプログラムファイル中には実在しない文字。
- ・ 白黒反転表示の文は、MPIプログラムに固有の関数等を含む文。
- ・ 19行目より前は「world.c」と全く同じ。



# 基本的なMPI関数の紹介

## Collective Communication (MPI\_Scan) の実行例：scan.c

ログインノードの画面表示

各プロセスの出力内容 (2ノード4プロセスで実行した場合)

```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$ mpicc -o scan scan.c <----- ・ C言語で書かれたMPIプログラムのコンパイルとリンクをした。
$ sbatch -p testq -t 00:01:00 exec.sh scan <-- ・ ジョブを投入した。
Submitted batch job 93367 <----- ・ ジョブ番号が「93367」のジョブとして受け付けられたとの表示
$
$ head -n 8 exec.sh_93367.stderr | tail -n 1
+ mpiexec -n 4 --machinefile exec.sh_93367.machinefile bash numa_bind_exec.sh 2 . scan
$                               ↑-- ・ 2ノード4プロセスで、ロードモジュールファイル「scan」内の命令列を実行した。
$                               その際、各プロセスを、それぞれ、別々のNUMAノードにバインドした。
$
$ ls -m *.txt <----- ・ 各プロセスより、ファイル名が
scan_0.txt, scan_1.txt, scan_2.txt, scan_3.txt      「ロードモジュールファイル名_ランク番号.txt」
$                                                    の形式のファイルが出力されたことを確認した。
$ ↓-- ・ 出力されたファイルの内容を横並びに併記する。
$ paste -d @ scan_[0-3].txt | sed 's/@/ /g'
rankW= 0 sizeW= 4      rankW= 1 sizeW= 4      rankW= 2 sizeW= 4      rankW= 3 sizeW= 4
index= 0 send=00      index= 0 send=10      index= 0 send=20      index= 0 send=30
----- MPI_Scan
index= 0 recv=00      index= 0 recv=10      index= 0 recv=30      index= 0 recv=60
$
```

# 基本的なMPI関数の紹介

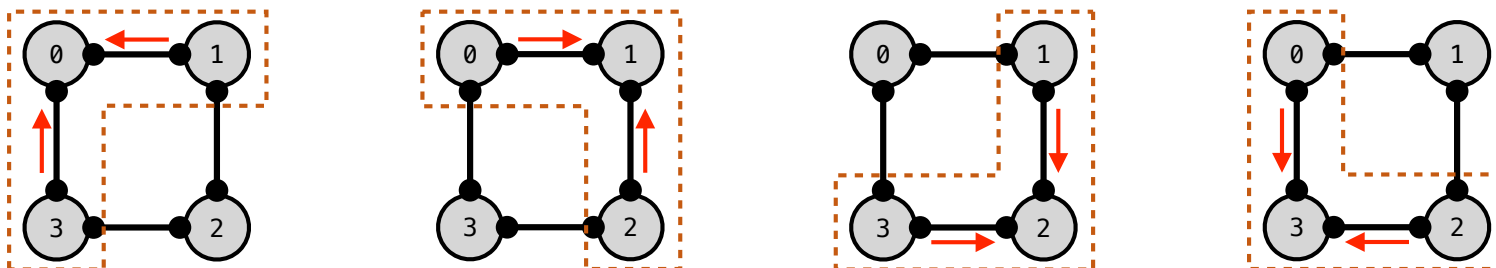
## Nonblocking Collective Communication (MPI\_Igather) の実行例：igather.c

問題意識：

Collective Communication関数の引数には「tag」が含まれていないため、送信側プロセスが実行したどの実行文に起因して送信されたデータを受け取ったのか、受信側プロセスは判断することができない。

→ 複数のCollective Communication関数の実行に際しては、同一コミュニケータ内の全てのプロセスが、同じCollective Communication関数を、同じ順序で実行する必要がある。

別々のコミュニケータを使用すれば判断可能となるが、コミュニケータに重なりがある場合には、Dead Lockに陥る危険がある。



周期境界条件の場合、各プロセスに、それよりランク番号が1つ小さいプロセスと、1つ大きいプロセスを含む、3プロセス分のデータを集めようとして、各プロセスが以下の3回のMPI\_Gatherを実行しようとする、Dead Lockに陥る。

```
MPI_Gather(sb,1,MPI_INT,rb,1,MPI_INT,1,mpi_comm_sub[ranks[0]]); ← ランク番号が1つ小さいプロセスへのGather
MPI_Gather(sb,1,MPI_INT,rb,1,MPI_INT,1,mpi_comm_sub[ranks[1]]); ← 現行プロセスへのGather
MPI_Gather(sb,1,MPI_INT,rb,1,MPI_INT,1,mpi_comm_sub[ranks[2]]); ← ランク番号が1つ大きいプロセスへのGather
```

解決方法：

Nonblocking Collective Communication関数を使用すれば良い。

# 基本的なMPI関数の紹介

## Nonblocking Collective Communication (MPI\_Igather) の実行例：igather.c

使用するMPI関数の説明（MPI3.1の規格書の第5章の記述の要約）

引数の文字の色は、引数の属性（**入力**、**出力**、**入出力**）を表す。

```
int MPI_Igather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf,
               int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)
```

コミュニケータ「comm」内に存在する全てのプロセスがこの関数を実行することにより、「comm」内に存在する全てのプロセスの、メモリ空間内のアドレス「sendbuf」から始まる「sendtype」型の値「sendcount」個を、ランク番号が「root」のプロセスの、メモリ空間内のアドレス「recvbuf」から始まる場所に、ランク番号順に格納する手続きを開始する。送信するデータを、メモリ空間内のアドレス「sendbuf」から始まる場所より読み出し終わるまでは、待たないことから、Nonblocking Gatherとも呼ばれる。ここで、変数「recvcount」は、各プロセスから受け取る「recvtype」型のデータの個数を表す。

変数「request」：関数「MPI\_Igather」による通信処理の個々を識別するためのハンドル。

→ 先に実行されたNonblocking Collective Communication関数によるデータの送受信の完了を待たずに、次々とNonblocking Collective Communication関数を実行することができる。

(↓39ページにて既出の事項の再掲)

```
int MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status array_of_statuses[])
```

配列「array\_of\_requests」に格納された全ての「request」に関係する通信が終了するまで待つ。通信が終了したら、配列「array\_of\_requests」の各要素に、値「MPI\_REQUEST\_NULL」を代入する。

# 基本的なMPI関数の紹介

## Nonblocking Collective Communication (MPI\_Igather) の実行例：igather.c

ソースプログラムファイル「igather.c」の内容

各プロセスが、それぞれ独立に、以下の命令列を実行する。

**argv[0]**: プログラム名

**sizeW**: 「MPI\_COMM\_WORLD」内のプロセス数

**rankW**: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

```
19 int ndims,reorder,idim,shft,rank_src,rank_dst,nd;
20 int *dims,*prds,*sendbuf,*recvbuf;
21 ndims=1;
22 dims=(int *)malloc(sizeof(int)*ndims);
23 prds=(int *)malloc(sizeof(int)*ndims);
24 dims[0]=sizeW;
25 prds[0]=true;
26 reorder=false;
27 MPI_Comm mpi_comm_cart;
28 MPI_Cart_create(MPI_COMM_WORLD,ndims,dims,prds,reorder,&mpi_comm_cart);

29 idim=0;
30 shft=1;
31 MPI_Cart_shift(mpi_comm_cart,idim,shft,&rank_src,&rank_dst);
32 nd=1;
33 sendbuf=(int *)malloc(sizeof(int)*nd);
34 recvbuf=(int *)malloc(sizeof(int)*nd*3);
35 sendbuf[0]=rankW*10;
36 for(int i=0;i<3;i++){
37     recvbuf[i]=-1;
38 }
39 fprintf(fp0,"\n");
40 for(int i=0;i<nd;i++){
41     fprintf(fp0," index=%2d send=%02d\n",i,sendbuf[i]);
42 }
```

↓--・既存のコミュニケータ「MPI\_COMM\_WORLD」に、  
↓ 1次元周期境界条件の「Cartesian topology」情報を付加した、  
↓ 新たなコミュニケータ「mpi\_comm\_cart」を作成する。

↓--・「Cartesian topology」の1次元目の座標が「-1」のプロセスからデータを受け取り、  
↓ 「+1」のプロセスに向けてデータを送るときの、送信元のランク番号と、送り先のランク番号を取得する。

==== ここまでは「cart.c」と同じ =====

・配列要素「sendbuf[0]」に、変数「rankW」の値を10倍した値を格納する。

・配列「recvbuf」の全要素に、値「-1」を格納する。

↑--・配列要素番号「i」と配列要素「sendbuf[i]」の内容を、「index」と「send」の値として、「fp0」に書き込む。

# 基本的なMPI関数の紹介

## Nonblocking Collective Communication (MPI\_Igather) の実行例：igather.c

ソースプログラムファイル「igather.c」の内容

各プロセスが、それぞれ独立に、以下の命令列を実行する。

**argv[0]**: プログラム名

**sizeW**: 「MPI\_COMM\_WORLD」内のプロセス数

**rankW**: 現行プロセスの「MPI\_COMM\_WORLD」内でのランク番号

- ・ 茶色の文字は実際のソースプログラムファイル中には実在しない文字。
- ・ 白黒反転表示の文は、MPIプログラムに固有の関数等を含む文。
- ・ 19行目より前は「world.c」と全く同じ。

```
43  int ranks[3]; <----- ・ ランク番号が現行プロセスより
44  ranks[0]=rank_src; <--- 1つ小さいプロセスと、
45  ranks[1]=rankW; <----- 現行プロセスと、
46  ranks[2]=rank_dst; <--- 1つ大きいプロセスの、3つのプロセスを1つのグループとする為の配列を用意する。

47  int (*ranks_tmp)[3]=(int (*)[3])malloc(sizeof(int[3])*sizeW);
48  MPI_Allgather(ranks,3,MPI_INT,ranks_tmp,3,MPI_INT,MPI_COMM_WORLD);
49  MPI_Comm *mpi_comm_sub=(MPI_Comm *)malloc(sizeof(MPI_Comm)*sizeW);
50  MPI_Group mpi_group_world,mpi_group_sub;
51  MPI_Comm_group(MPI_COMM_WORLD,&mpi_group_world);
52  for(int i=0;i<sizeW;i++){
53      MPI_Group_incl(mpi_group_world,3,&(ranks_tmp[i][0]),&mpi_group_sub);
54      MPI_Comm_create(MPI_COMM_WORLD,mpi_group_sub,&(mpi_comm_sub[i])); <----
55  }
56  MPI_Request requests[3];
57  MPI_Igather(sendbuf,nd,MPI_INT,recvbuf,nd,MPI_INT,1,mpi_comm_sub[ranks[0]],&(requests[0]));
58  MPI_Igather(sendbuf,nd,MPI_INT,recvbuf,nd,MPI_INT,1,mpi_comm_sub[ranks[1]],&(requests[1]));
59  MPI_Igather(sendbuf,nd,MPI_INT,recvbuf,nd,MPI_INT,1,mpi_comm_sub[ranks[2]],&(requests[2]));
    ↑-- ・ このMPIプログラムを実行中の全てのプロセスにおいて、各プロセスを「root」とする、それよりランク番号が1つ小さい
        プロセスと、1つ大きいプロセスとの、3つのプロセス間における、要素数「nd (=1)」のgatherを実行する為に必要な、
        3回のMPI_Igatherを実行する（必要なデータ送受信の手続きを開始する）。

60  MPI_Waitall(3,requests,MPI_STATUSES_IGNORE); <--- ・ 上記3回のMPI_Igatherに起因する
61  fprintf(fp0,"\n");                               データ送受信が完了するのを待つ。
62  for(int i=0;i<nd*3;i++){
63      fprintf(fp0," index=%2d  recv=%02d\n",i,recvbuf[i]);
64  }          ↑-- ・ 配列要素番号「i」と配列要素「recvbuf[i]」の内容を、「index」と「recv」の値として、「fp0」に書き込む。
```

# 基本的なMPI関数の紹介

## Nonblocking Collective Communication (MPI\_Igather) の実行例：igather.c

ログインノードの画面表示

各プロセスの出力内容 (2ノード4プロセスで実行した場合)

```
・ クリーム色の文字は実際の画面上には実在しない文字
$
$ mpicc -o igather igather.c <-----・ C言語で書かれたMPIプログラムのコンパイルとリンクをした。
$ sbatch -p testq -t 00:01:00 exec.sh igather <--・ ジョブを投入した。
Submitted batch job 93359 <-----・ ジョブ番号が「93359」のジョブとして受け付けられたとの表示
$
$ head -n 8 exec.sh_93359.stderr | tail -n 1
+ mpiexec -n 4 --machinefile exec.sh_93359.machinefile bash numa_bind_exec.sh 2 . igather
$                                     ↑--・ 2ノード4プロセスで、ロードモジュールファイル「igather」内の命令列を実行した。
$                                     その際、各プロセスを、それぞれ、別々のNUMAノードにバインドした。
$
$ ls -m *.txt <-----・ 各プロセスより、ファイル名が
igather_0.txt, igather_1.txt, igather_2.txt, igather_3.txt  「ロードモジュールファイル名_ランク番号.txt」
$                                     の形式のファイルが出力されたことを確認した。
$ ↓--・ 出力されたファイルの内容を横並びに併記する。
$ paste -d @ igather_*.txt | sed 's/@/ /g'
rankW= 0 sizeW= 4      rankW= 1 sizeW= 4      rankW= 2 sizeW= 4      rankW= 3 sizeW= 4
index= 0  send=00      index= 0  send=10      index= 0  send=20      index= 0  send=30
----- MPI_Igather
index= 0  recv=30      index= 0  recv=00      index= 0  recv=10      index= 0  recv=20
index= 1  recv=00      index= 1  recv=10      index= 1  recv=20      index= 1  recv=30
index= 2  recv=10      index= 2  recv=20      index= 2  recv=30      index= 2  recv=00
$
```

# スケーリング則

## 並列数を増やすことの効果

「並列数を増やす」 = 「使用する計算機資源量を増やす」

- 問題サイズを並列数によらず一定とした場合 (Strong Scaling)

Amdahlの法則

→ 並列数を増やすことの効果は限定的。

使用する計算機資源量を増やした分だけ、実行時間が短くなるわけではない。

- 問題サイズを並列数と共に大きくした場合 (Weak Scaling)

Gustafsonの法則

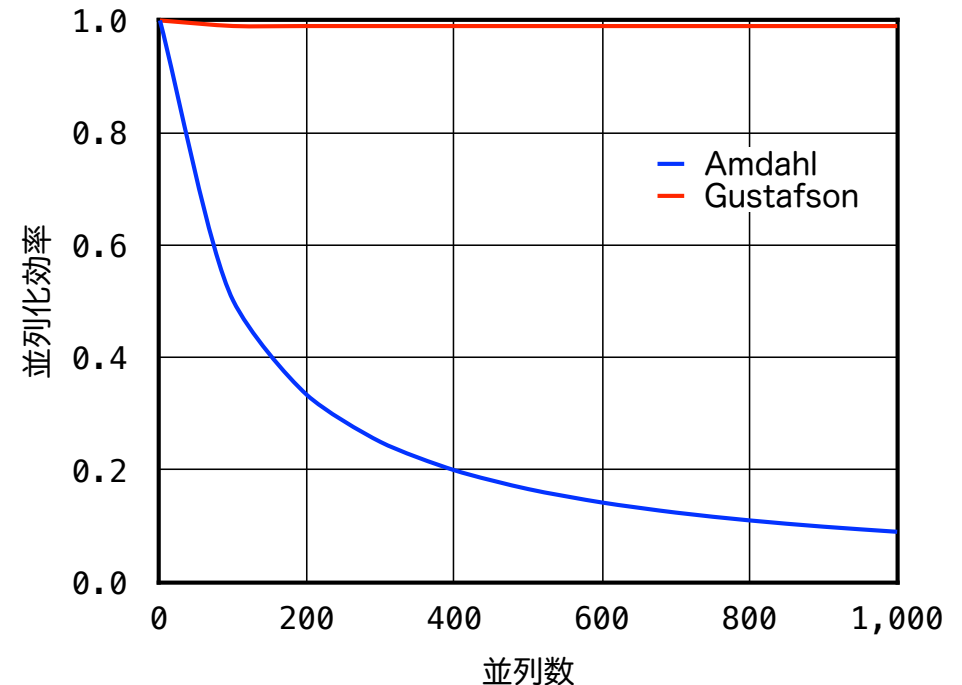
→ 並列数を増やした分だけ効果が得られる。

使用する計算機資源量を増やした分だけ、より大きな問題を解くことができる。

速度向上率：ある1つの問題を解く時に、  
並列数を  $N$  とした場合にかかる時間は、  
並列数を 1 とした場合にかかる時間の何分の1か。  
並列実行することにより何倍速くなるか。

並列化効率：速度向上率を並列数で割ったもの。  
並列数を 100 とした時に、  
逐次実行（並列数 1）の時より、  
100倍速くなれば 1.0、  
50倍しか速くならなければ 0.5。  
計算機資源の利用効率。

並列化効率のイメージ図



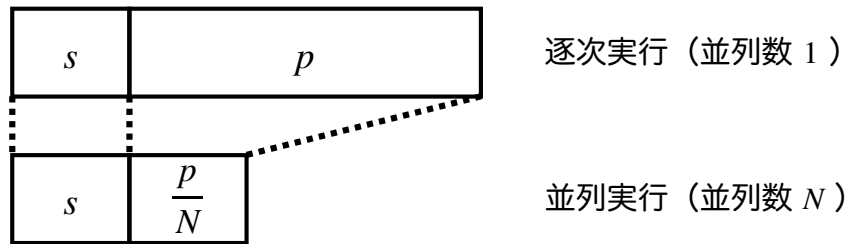
# スケーリング則

並列数を増やすことの効果（問題サイズを並列数によらず一定とした場合）

## Amdahlの法則

- ・プログラムを逐次実行（並列数 1）した場合の実行時間は、並列化可能部分  $p$  と並列化不可能部分  $s$  に分けることができると考える。
- ・並列実行（並列数  $N$ ）することにより、並列化可能部分の実行時間  $p$  は  $\frac{p}{N}$  に短縮されるとする。
- ・並列実行（並列数  $N$ ）しても、並列化不可能部分の実行時間  $s$  は変わらないとする。

プログラムの実行時間  $T_1 = s + p = 1$



プログラムの実行時間  $T_N = s + \frac{p}{N}$

Gustafson, J.L., 1988:  
REEVALUATING AMDAHL'S LAW.  
*Communications of the ACM*, **31**,  
532-533.

- ・速度向上率  $R_N := \frac{T_1}{T_N} = \frac{s+p}{s+\frac{p}{N}}$  ← 並列実行することにより何倍速くなるか。

- ・並列化効率  $E_N := \frac{R_N}{N}$

並列化効率（Amdahl）

		s	0.1000	0.0100	0.0010	0.0001
N	10	0.5263	0.9174	0.9911	0.9991	
	100	0.0917	0.5025	0.9099	0.9902	
	1,000	0.0099	0.0910	0.5003	0.9092	
	10,000	0.0010	0.0099	0.0909	0.5000	



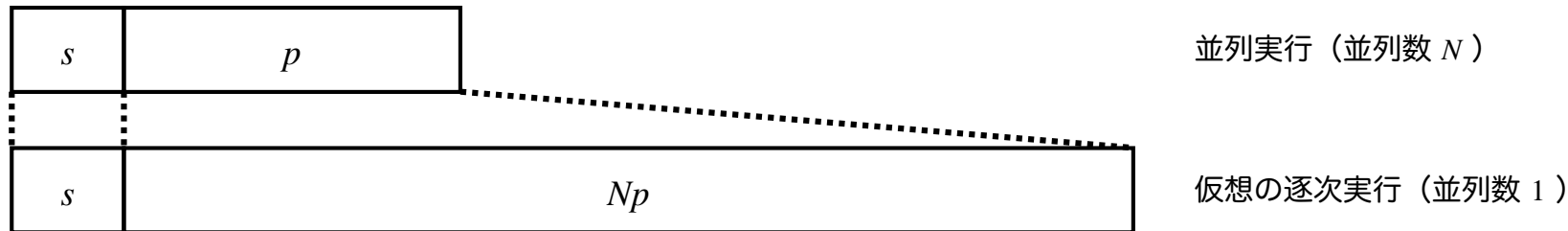
# スケーリング則

並列数を増やすことの効果（問題サイズを並列数と共に大きくした場合）

## Gustafsonの法則

- ・プログラムを並列実行（並列数  $N$ ）した場合の実行時間は、並列化可能部分  $p$  と並列化不可能部分  $s$  に分けることができる考える。
- ・並列化可能部分の実行時間  $p$  が、並列数  $N$  によらず一定となるように、問題サイズを調整するとする。
- ・並列化不可能部分の実行時間  $s$  は、並列数  $N$  によらず一定であるとする。 → プログラムの実行時間：並列数によらず一定  
問題サイズ：並列数に比例
- ・並列数  $N$  の場合と同じ問題を、もし仮に逐次実行で解いた場合には、プログラムの実行時間は  $s + Np$  となるとする。

プログラムの実行時間  $T_N = s + p = 1$



プログラムの実行時間  $T_1 = s + Np$

- ・速度向上率  $R_N := \frac{T_1}{T_N} = \frac{s + Np}{s + p}$  ← 並列実行することにより何倍速くなるか。

- ・並列化効率  $E_N := \frac{R_N}{N}$

並列化効率（Gustafson）

		s	0.1000	0.0100	0.0010	0.0001
		10	0.9100	0.9910	0.9991	0.9999
N	100	0.9010	0.9901	0.9990	0.9999	
	1,000	0.9001	0.9900	0.9990	0.9999	
	10,000	0.9000	0.9900	0.9990	0.9999	

Gustafson, J.L., 1988:  
REEVALUATING AMDAHL'S LAW.  
*Communications of the ACM*, **31**,  
532-533.

2024年5月

一般財団法人高度情報科学研究機構（著作者）

本資料を教育目的で利用いただいて構いません。利用に際しては以下の点に留意いただくとともに、下記のヘルプデスクにお問い合わせください。

- 本資料は、構成・文章・画像などの全てにおいて著作権法上の保護を受けています。
- 本資料の一部あるいは全部について、いかなる方法においても無断での転載・複製を禁じます。
- 本資料に記載された内容は、予告なく変更される場合があります。
- 本資料に起因して使用者に直接または間接的損害が生じても、著作者はいかなる責任も負わないものとします。

問い合わせ先：ヘルプデスク helpdesk[-at-]hpci-office.jp（[-at-]を@にしてください）