講義２

並列プログラミングの基本（MPI）

改訂版

理化学研究所 AICS
システムソフトウェア研究チーム
堀　敦史

並列プログラミングの基本（MPI）

- 本日　8/5（月）
  - 並列プログラムとは
  - MPIの基礎
  - MPIの基本的な通信

- MPI-IO　8/7（水）9:00〜
  - +MPI の高度な部分について

- 代講：最適化 I　8/9（金）9:00〜

並列プログラミングの基本（MPI）

- **並列プログラムとは**
- MPIの基礎
- MPIの基本的な通信

- MPI-IO　8/7（水）9:00〜
  - +MPI の高度な部分について

逐次と並列の違い (1)

逐次プログラム

```
main() {
  ....
}
```

並列
プログラム

```
main() {
  ....
}
```

```
main() {
  ....
}
```

••

```
main() {
  ....
}
```

逐次と並列の違い (2)

並列プログラム

Program A
```
main() {
  ....
}
```

Program B
```
main() {
  ....
}
```

・・

Program X
```
main() {
  ....
}
```

Program A
```
main() {
  if(rank==0)
  ....
}
```

Program A
```
main() {
  if(rank==1)
  ....
}
```

・・

Program A
```
main() {
  if(rank==X)
  ....
}
```

SPMD (Single Program Multiple Data)

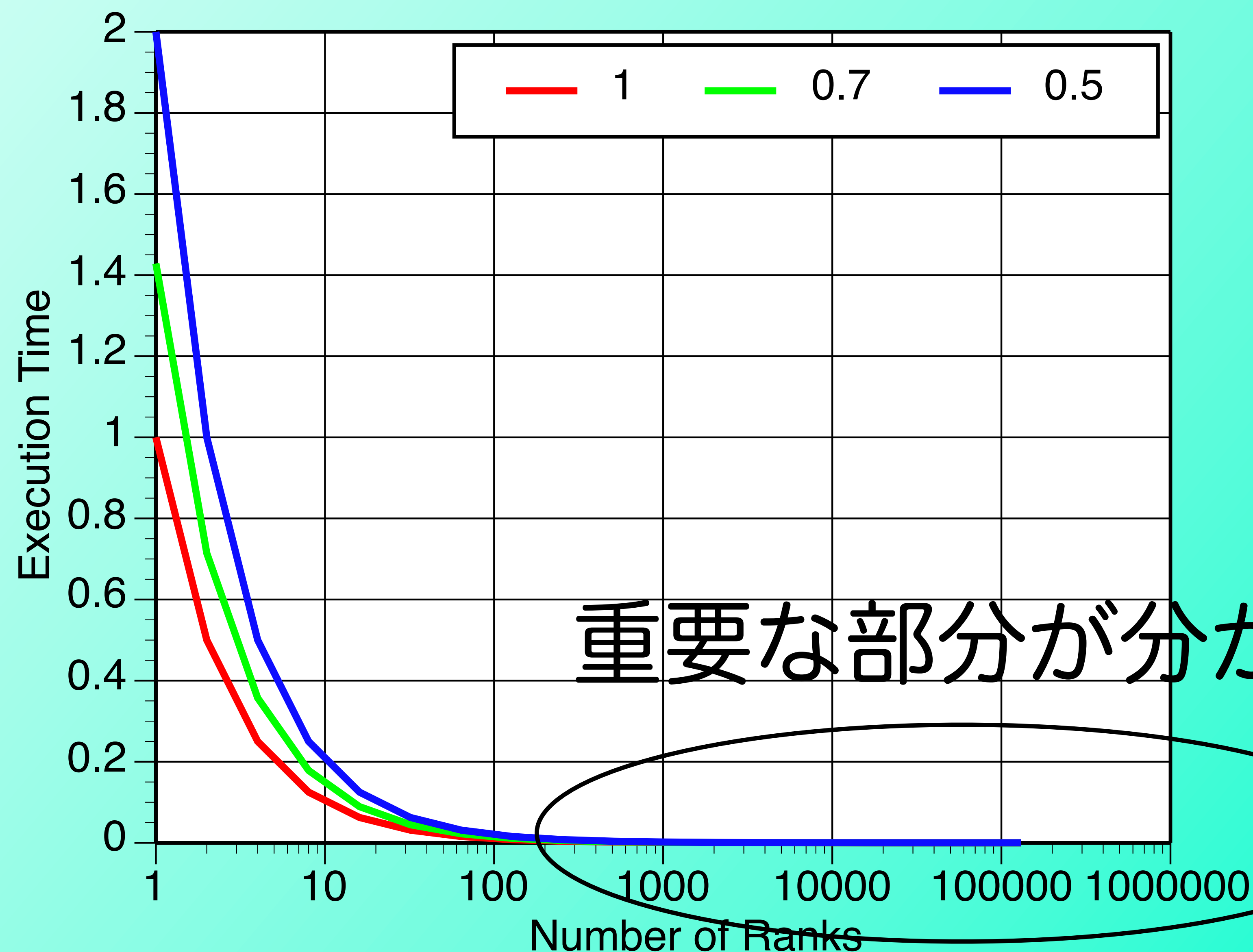
並列プログラム

- 目的
  - プログラムの実行を**速く**する

- どのように並列化するか、並列化の指標
  - **並列化の手法**
  - **並列化効率**
    - 2並列で2倍、N並列でN倍になるハズに対し、どの程度の高速化が実際に実現できたか
    - *P(N)*：N並列での実際の速度
    - 並列化効率 = $P(N) / P(M) * (M / N)$

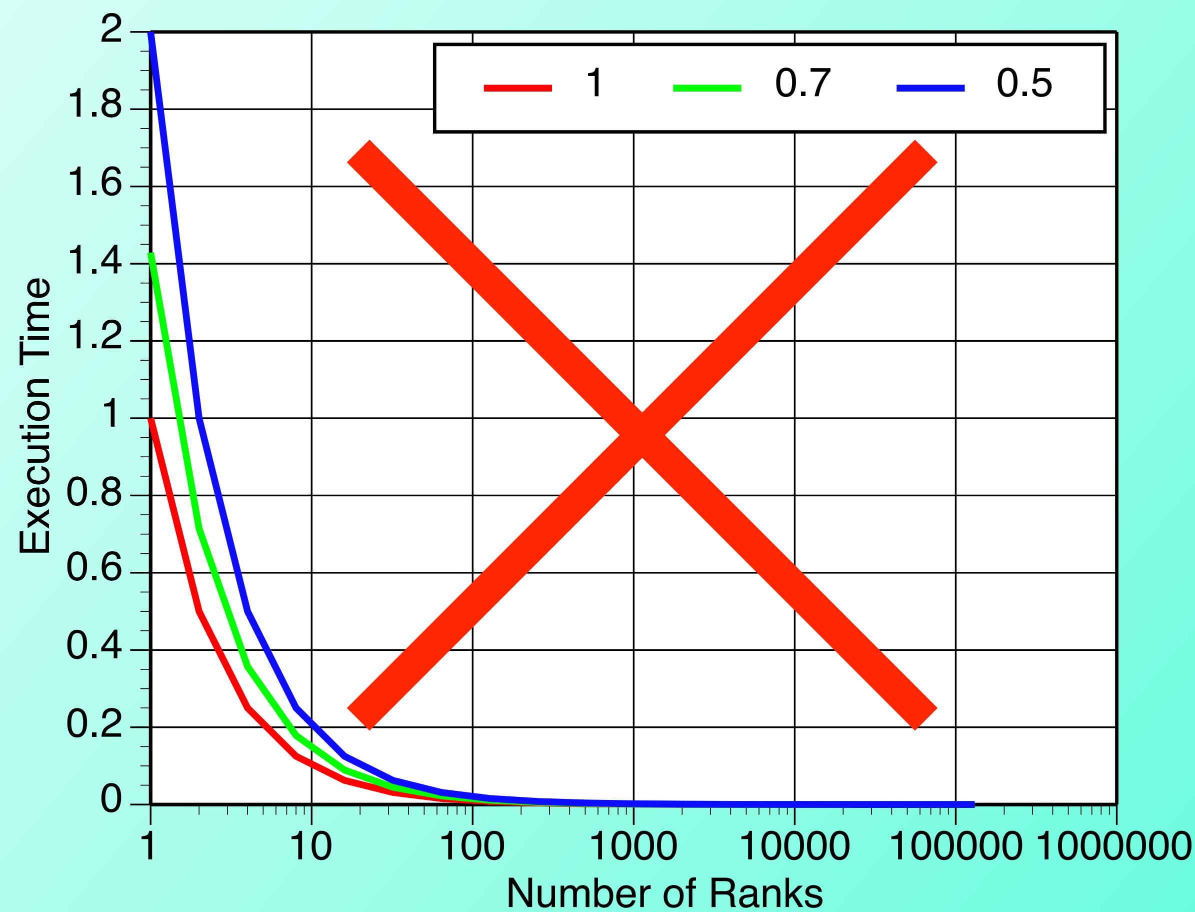
並列化効率のグラフ

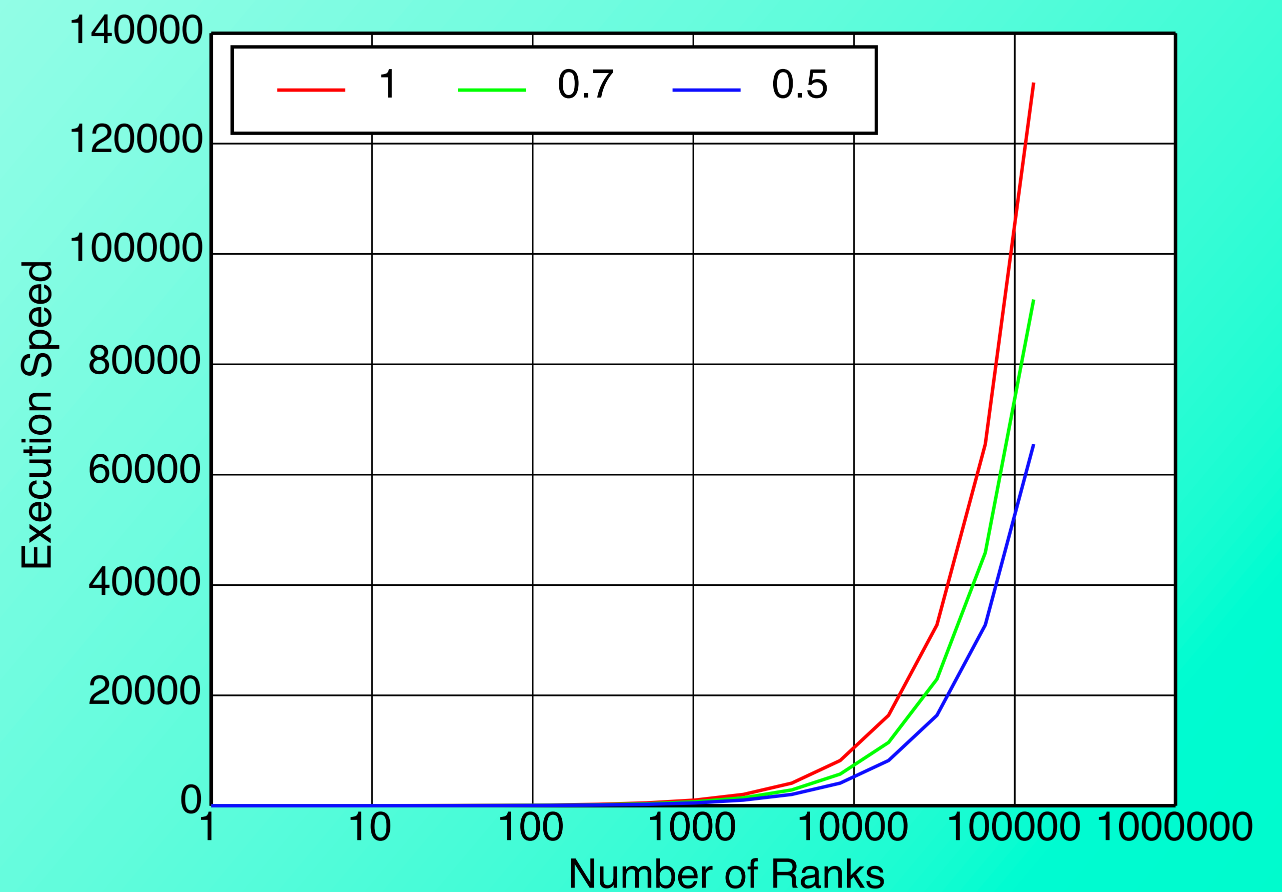
- 実行時間のグラフは並列化効率が分かり難い
- 並列化効率のグラフを書くように



並列化効率のグラフ改

- 縦軸を速度（時間の逆数）にすると分かり易い
  - 実際には rank 数が大きい程、並列化効率が落ちるケースが大半





並列プログラミングの基本（MPI）

- 並列プログラムとは
- **MPIの基礎**
- MPIの基本的な通信

- MPI-IO　8/8（水）9:00〜
  - +MPI の高度な部分について

MPI とは

- MPI : Message Passing Interface
  - 通信ライブラリ
  - ほとんどの並列計算機で利用可能
    - ポータビリティが高い

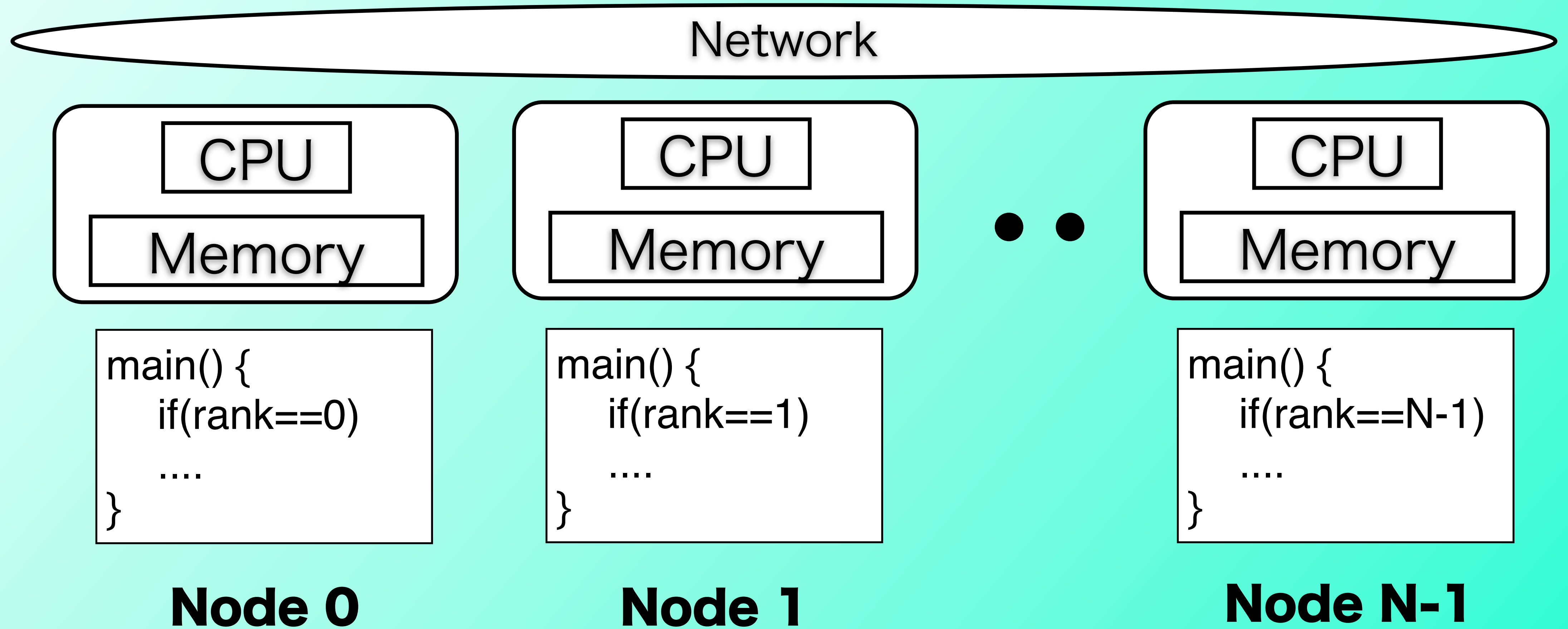
- 以下、最新の MPI Version 2.2 に基づいて
  - 参考資料を参照のこと

MPI の基礎

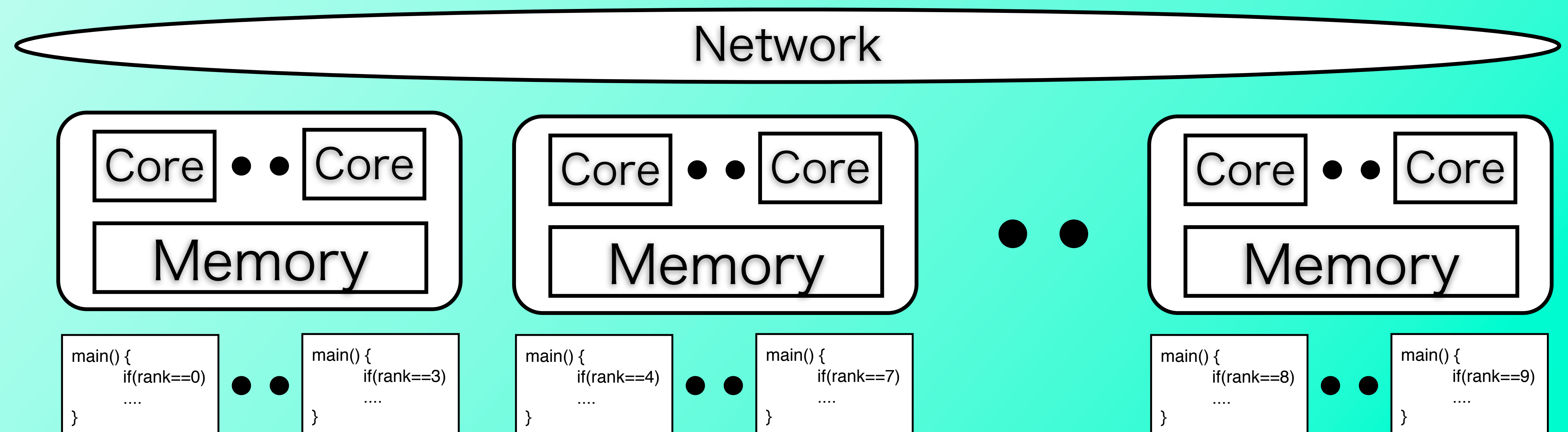
- 基本的に全て同じプログラムが並列に動作する
- 並列に動作する単位を**プロセス**と呼ぶ
- 個々のプロセスを識別するIDを**rank**と呼ぶ
- rank の違いによりプロセス毎に違う動作をさせることができる
- 他のプロセスのデータをアクセスするにはMPIの**通信**をおこなう必要がある
- プロセスは通信の単位でもある
- MPIには様々な通信が用意されている

MPIにおける通信の単位

初期



現在

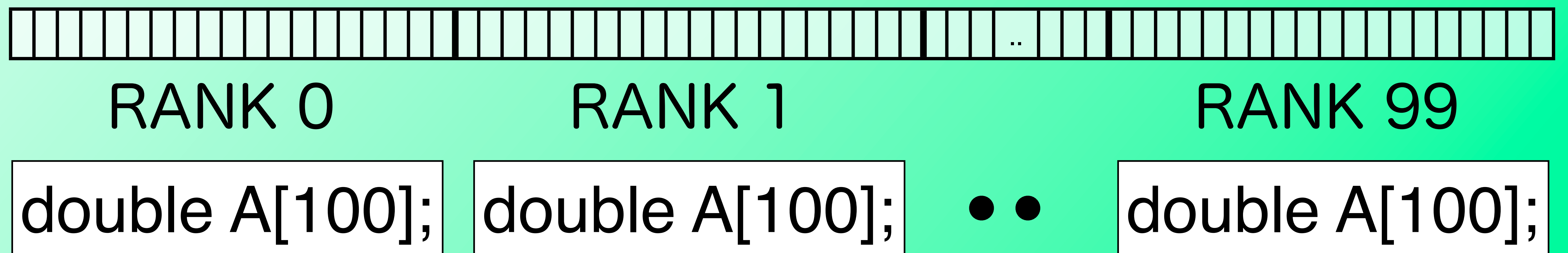


MPIによる並列化

- CPU Core 毎に MPI プロセスを実行する
  - **Flat MPI**
    - Rank数はコアの数に等しい
- Node 間を MPI で並列化、Node 内は OpenMP（8/6 講義4）やコンパイラで並列化
  - **Hybrid MPI**
    - Rank数はノード数に等しい
    - 「京」は主にこの方式

通信はなぜ必要か？

- プロセスが持つデータは、そのプロセス内でしかアクセスできない
- 他のプロセスのデータにアクセスするには通信する必要がある



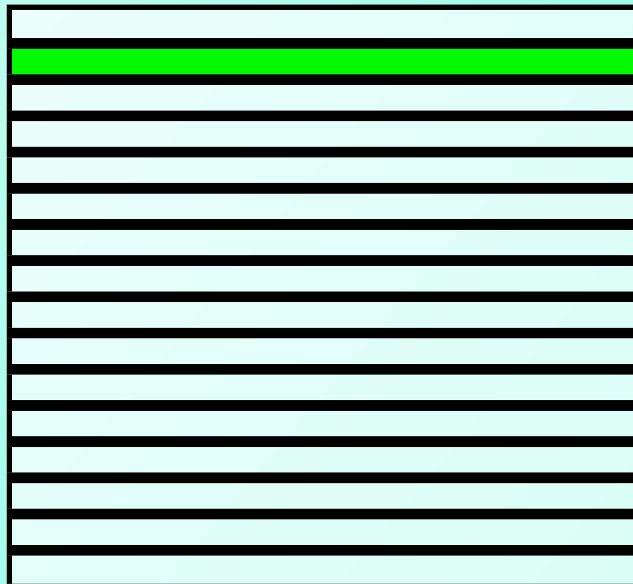
並列プログラムの難しさ

- 逐次プログラム
  - 基本となる問題解法アルゴリズム
  - 最適化
  - コードの読み易さ
- **並列プログラム**
  - **データの分散（個々のプロセスへの割振り）**
  - **個々のプロセスの処理量を均等化（負荷分散）**
  - **通信の最適化**
    - **通信量、通信の頻度、ネットワークトポロジー**
    - **通信遅延の隠蔽**

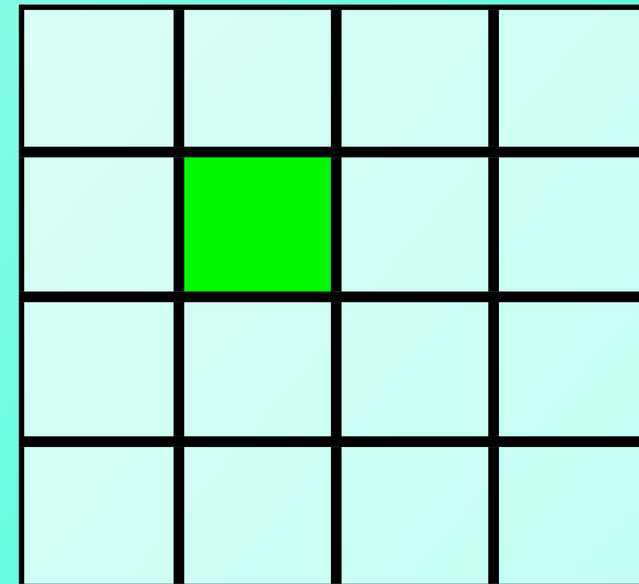
データの分散（分割）の例

- 2次元配列の分割し、個々のプロセスに分散する
- ここで、通信は分割の境界で発生し、境界線の長さに通信量は比例するものとする
- 下の図は、16分割した場合の例

1次元分割



2次元分割



- 通信量の観点からは2次元分割が有利
- 通信の頻度は、1次元が２回、2次元が4回
- 1次元分割では、分割数の上限が小さい
- プログラムは1次元分割の方が簡単

MPI の初期化と終了

```
C:    int MPI_Init( int **argc, char **argv )
      int MPI_Finalize( void )
F:    MPI_INIT( ierr )
      MPI_FINALIZE( ierr )
```

- MPI ライブラリの初期化と終了
  - MPI_Finalize() の呼出でプログラムが終了する訳ではない

ランクの問い合わせ

```
C:      int MPI_Comm_rank( MPI_COMM_WORLD, int *rank )
        int MPI_Comm_size( MPI_COMM_WORLD, int *size )
F:      MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
        MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
```

- **コミュニケータ** (communicator)
  - **MPI_COMM_WORLD**
    - MPI ライブラリで定義された定数
  - MPI の通信において常に必要となる
  - 詳細については本日の最後に
- MPI_Comm_rank() - ランク番号を返す
- MPI_Comm_size() - ランク数を返す

C と FORTRAN

- C 言語
  - ほとんどの関数 ”int” を返し、正常に終了したか否かを返す
  - 以下、説明では戻り値については省略する

- FORTRAN
  - ほとんどのサブルーチンで、”integer ierr” 引数を最後に持ち、正常終了したか否かを返す

並列プログラミングの基本（MPI）

- 並列プログラムとは
- MPIの基礎
- **MPIの基本的な通信**
- MPIの仕様について

- MPI-IO　8/8（水）09:00〜
  - +MPI の高度な部分について

MPI通信の種別

- 1対1通信 (point-to-point communication)
- 集団通信 (collective communication)
- 片方向通信 (one-sided communication)

MPI通信の種別

- **1対1通信**
  (point-to-point communication)
- 集団通信 (collective communication)
- 片方向通信 (one-sided communication)

メッセージの送信

```
C:  MPI_Send( void *data, int count, MPI_Datatype type,
             int dest, int tag, MPI_COMM_WORLD )
F:  MPI_SEND( data, count, type, dest, tag, MPI_COMM_WORLD,
             ierr )
```

- MPI_Send() - メッセージを送信する
  - data　送信するデータ
  - count データの個数
  - type　**データの型**
  - dest　**送り先**
  - tag　**タグ**
  - ierr　エラーの有無

メッセージの受信

```
C:    MPI_Recv( void *data, int count, MPI_Datatype type,
          int src, int tag, MPI_COMM_WORLD, MPI_Status status )
F:    MPI_RECV( data, count, type, src, tag, MPI_COMM_WORLD,
          status, ierr )
```

- MPI_Send() - メッセージを受信する
  - data　送信するデータの格納場所
  - count　データの個数
  - type　**データの型**
  - src　**送り元**
  - tag　**タグ**
  - status　受信メッセージの情報（送り元、データ数など）
  - ierr　エラーの有無

MPIにおけるデータ型

C言語のデータ型との対応

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 (defined in <stddef.h>) (treated as printable character)
MPI_C_BOOL	_Bool
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	

FORTRAN言語のデータ型との対応

MPI datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

C言語とFORTRAN言語両方に対応するデータ型

MPI datatype	C datatype	Fortran datatype
MPI_AINT	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)
MPI_OFFSET	MPI_Offset	INTEGER (KIND=MPI_OFFSET_KIND)

MPIの通信モデル

- 通信モデル
  - 1対1 または connection base
    - 最初に相手と「つなぐ (connection)」操作が必要
    - 電話、TCP/IP など
  - 1対N または connection less（つなぐ必要がない）
    - 誰とでも通信可能
    - MPI に代表される並列計算用通信
    - 受信時の問題
      - 受信したメッセージは誰が送ったものなのか？
      - 期待したメッセージが何時来るか？

メッセージの「封筒」

- 「封筒 (envelope)」普通の郵便とのメタフォ
- 相手にメッセージを正しく送る、あるいは
  メッセージを正しく受け取るための仕組み
  - 送信と受信において
    - 送受信の相手（ランク番号）
    - タグ（32,767より小さい正の整数）
    - が合致した場合に送受信が完了する
  - 匿名 (anonymous) を受信に指定することができる
    - ランク番号　MPI_ANY_SRC
    - タグ　　　　MPI_ANY_TAG

MPI通信の注意点

- データ型は、対応する送受信で同一であること
  - MPI は型変換しない
- 送信データの長さ（count）と受信データの長さが異なる場合
  - 受け取り側が短い場合：切り捨てられる
  - 受け取り側が長い場合：データ長のみバッファに格納される
- **MPI_ANY_SRC タグは出来るだけ使わない**
  - 実行時の最適化を妨げる場合がある

1対1通信の完了

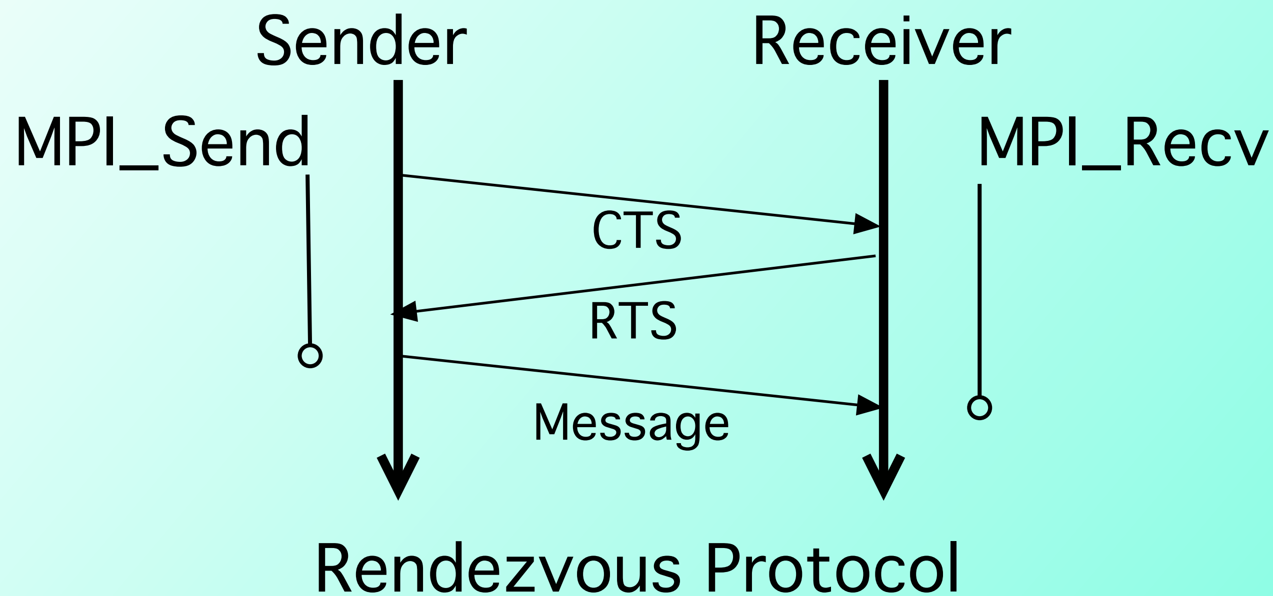
- 送信と受信が互いにマッチ（タグと送受信の相手注）した場合に「通信が完了（終了）」する
  - 終了する前にしてはいけないこと
    - 送信バッファの内容の変更
    - 受信バッファの内容の参照あるいは変更
- 通信が完了するまで関数（サブルーチン）から戻ってこない (かもしれない) => **Blocking 通信**

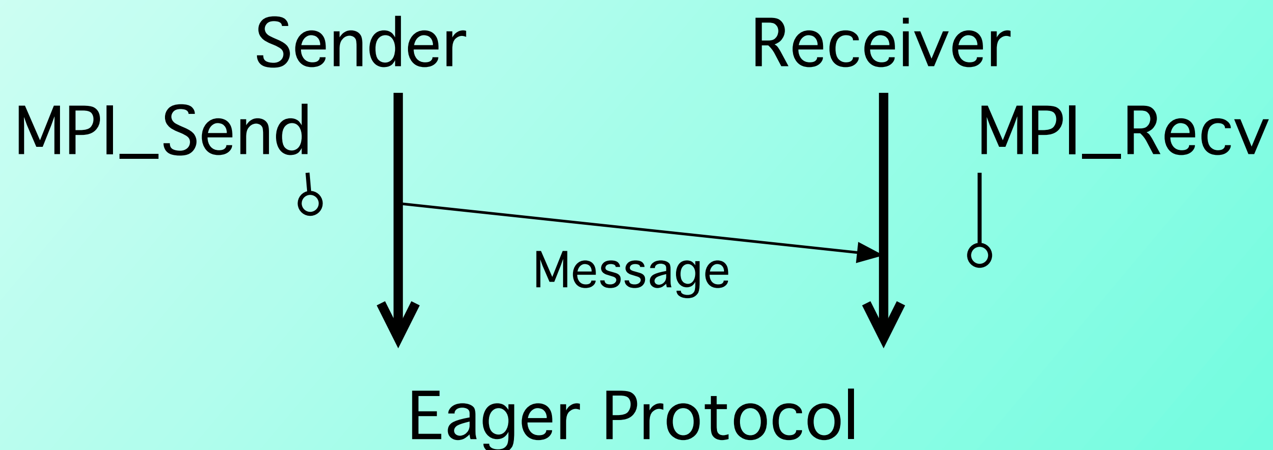
- 注）後述する Communicator を参照のこと

Eager通信とRendezvous通信

- 基本的にMPIの通信は送信と受信がマッチした時点で通信が完了する => **Rendezvous（ランデブー）通信**
- 送信側が受信の完了を待つのは時間の無駄
  - 送信側はマッチする受信の有無に関わらず完了する、という実装 => **Eager 通信**
  - 受信側では、
    - 既にマッチする受信があった場合
      - 受信が完了する
    - 未だマッチする受信がなかった場合
      - 受信したメッセージを内部バッファに保持し、後にマッチする受信が発行された時に、**バッファ内のメッセージを受信バッファにコピー**

Eager通信とRendezvous通信





- 一般に、短いメッセージでは eager が速く、長いメッセージでは rendezvous が速い
- Eager通信をより高速化する方法として、送信する前に受信関数を呼び出しておく
  - コピーの手間が省ける

初心者の間違い

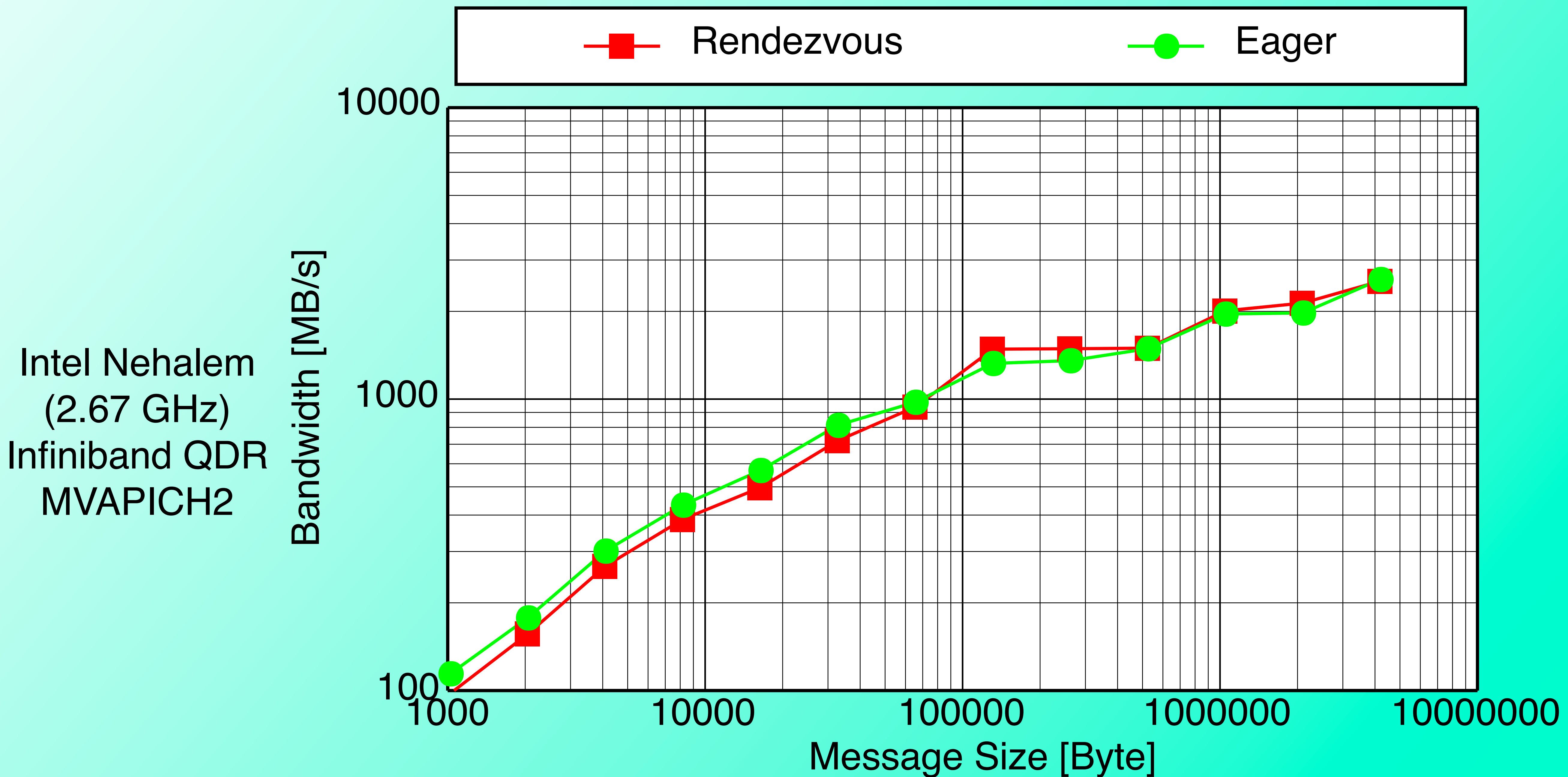
```
int rank;
char data[LEN]
MPI_Status status;

if( rank == 0 ) {
   MPI_Send( data, LEN, MPI_CHAR, 1, TAG, MPI_COMM_WORLD );
   MPI_Recv( data, LEN, MPI_CHAR, 1, TAG, MPI_COMM_WORLD, &status );
} else if( rank == 1 ) {
   MPI_Send( data, LEN, MPI_CHAR, 0, TAG, MPI_COMM_WORLD );
   MPI_Recv( data, LEN, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &status );
}
```

- Rendezvous プロトコルでは正しく動かない
  - MPI_Send に対応する MPI_Recv が動かないため
- Eager プロトコルでは動く
- 多くの実装では rendezvous と eager の切替はメッセージサイズ（LEN）で決まるが、その値は実装依存

あるMPI実装におけるバンド幅

多くのMPI実装では、Eager 通信と Rendezvous 通信をメッセージの大きさで切り替えている

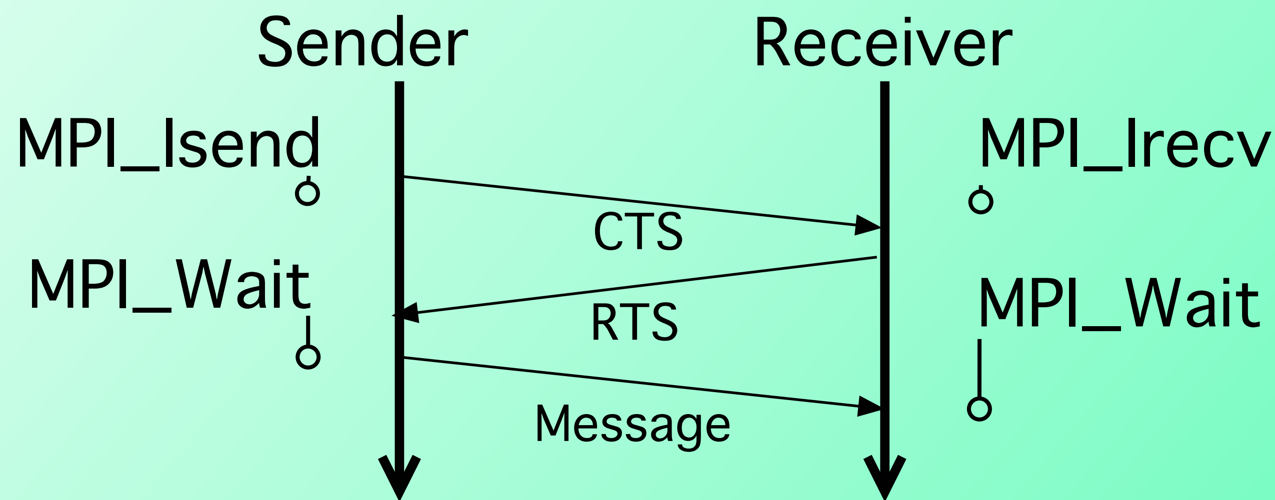


Non-blocking 通信

```
C:  MPI_Isend( void *data, int count, MPI_Datatype type,
              int dst, int tag, MPI_COMM_WORLD,  MPI_Request *req)
    MPI_Irecv( void *data, int count, MPI_Datatype type,
              int src, int tag, MPI_COMM_WORLD, MPI_Request *req )
    MPI_Wait( MPI_Request *req, MPI_Status *status )
F:  MPI_ISEND( data, count, type, dst, tag, MPI_COMM_WORLD,
              req, ierr )
    MPI_IRECV( data, count, type, src, tag, MPI_COMM_WORLD,
              req, ierr )
    MPI_WAIT( req, status, ierr )
```

- 通信の完了を待たないメッセージの送受信とそれらの完了待ち（MPI_Wait)
- **完了する以前の、送信バッファの更新、受信バッファの参照の結果は保証されない**

Non-Blocking 通信の利点



- MPI_Isend（あるいは MPI_Irecv）呼出し直後から、MPI_Wait を呼び出すまでの間に、計算することができる
  - **通信（遅延）の隠蔽**
  - 大規模なスパコンでは遅延が大きくなる傾向にある
    - 重要な高速化テクニックのひとつ
- 先の例で、先に MPI_Irecv() を呼ぶことで、メッセージのコピー回数を減らすことができる

初心者の間違い【修正版】

```
int rank;
char data[LEN];
MPI_Status status;
MPI_Request request;

if(   rank == 0 ) {
   MPI_Irecv( data, LEN, MPI_CHAR, 1, TAG, MPI_COMM_WORLD, &request );
   MPI_Send( data, LEN, MPI_CHAR, 1, TAG, MPI_COMM_WORLD );
} else if( rank == 1 ) {
   MPI_Irecv( data, LEN, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &request );
   MPI_Send( data, LEN, MPI_CHAR, 0, TAG, MPI_COMM_WORLD );
}
MPI_Wait( &request, &status );
```

- Non-blocking なので、先に受信を呼んでおくことができる。

初心者の間違い【修正版その2】

```
int rank;
char data0[LEN] data1[LEN];
MPI_Status status;

if( rank == 0 ) {
   MPI_Sendrecv(  data0, LEN, MPI_CHAR, 1, TAG,  % 送信の指定
                  data1, LEN, MPI_CHAR, 1, TAG,  % 受信の指定
                  MPI_COMM_WORLD, &status );
} else if( rank == 1 ) {
   MPI_Sendrecv(  data0, LEN, MPI_CHAR, 0, TAG,  % 送信の指定
                  data1, LEN, MPI_CHAR, 0, TAG,  % 受信の指定
                  MPI_COMM_WORLD, &status );
}
```

- MPI_Sendrecv()
  - 送信と受信を同時におこなう

MPI通信の種別

- 1対1通信 (point-to-point communication)
- **集団通信 (collective communication)**
- 片方向通信 (one-sided communication)
  - 8/8（水）

1対1通信と集団通信の違い

- 1対1通信
  - プロセス集合のなかの任意のペア間での通信

- 集団通信
  - プロセス集合の全てのプロセスが同じ目的の通信に同時に関与する
  - 全てのプロセスで同じMPI関数を、同じ引数で呼ぶ（値は違う場合がある）
  - タグは指定しない

集団通信の概略

- データの放送：**MPI_Bcast**
- データの集約：**MPI_Gather**, **MPI_Allgather**
- データの散布：**MPI_Scatter**
- データの集約と分散：**MPI_Alltoall**
- データの縮小：**MPI_Reduce**, **MPI_Allreduce**
- データの縮小と分散：MPI_Reduce_scatter
- データの条件付き縮小：MPI_Scan, MPI_Exscan
- 同期：**MPI_Barrier**

集団通信の通信パターン

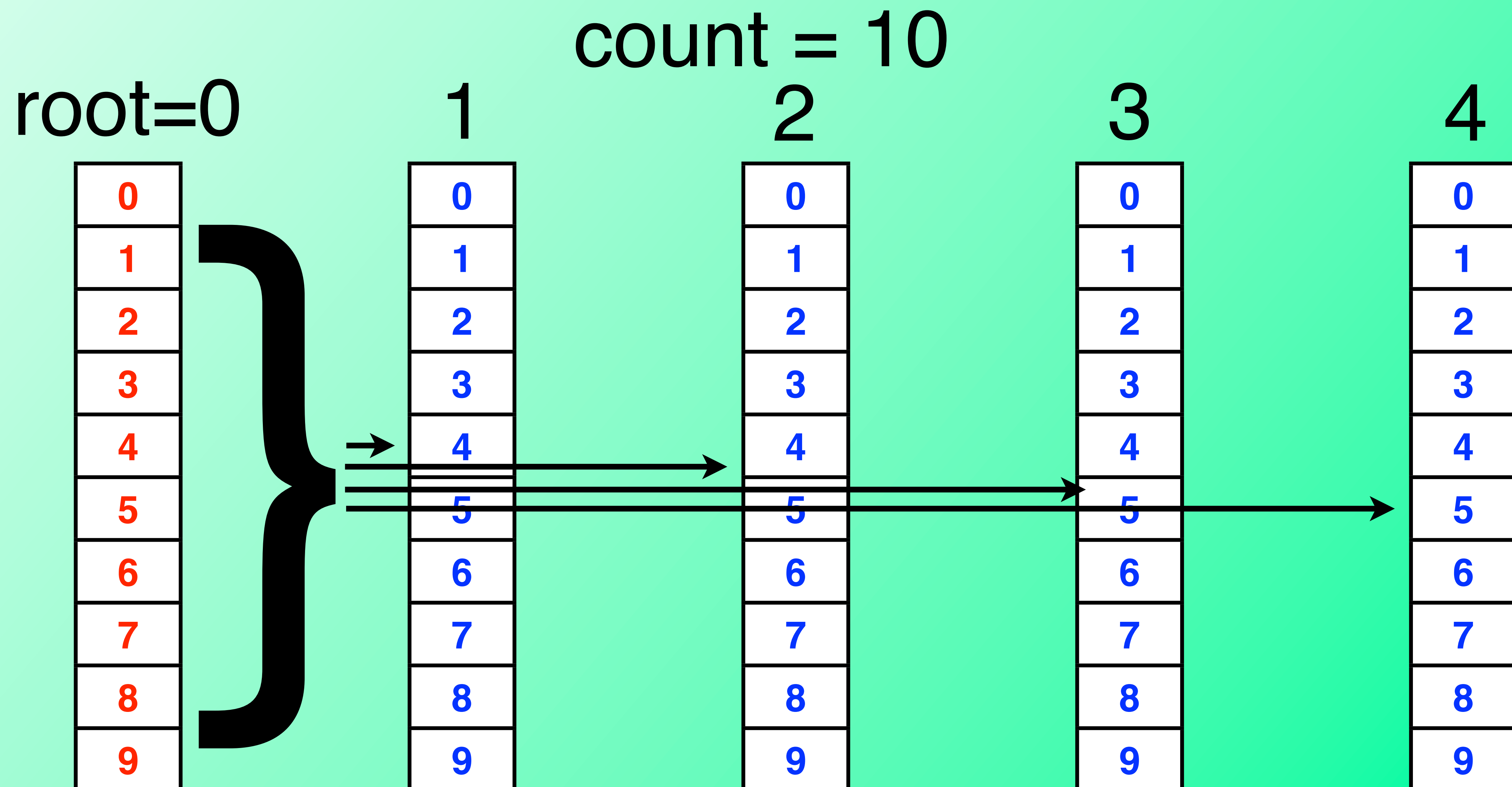
- 1対全（root プロセスあり）
  - MPI_Bcast, MPI_Scatter
- 全対1（root プロセスあり）
  - MPI_Gather, MPI_Reduce
- 全対全（root プロセスなし）
  - MPI_Barrier, MPI_Reduce_scatter,
  - MPI_All***

集団通信関数の名前の規則

- MPI_All***　root プロセスがない
- MPI_***v　要素毎に、任意の長さを送受信可能
- MPI_***w　要素毎に、任意の長さ，任意の DATA_TYPE を送受信可能

MPI_Bcast

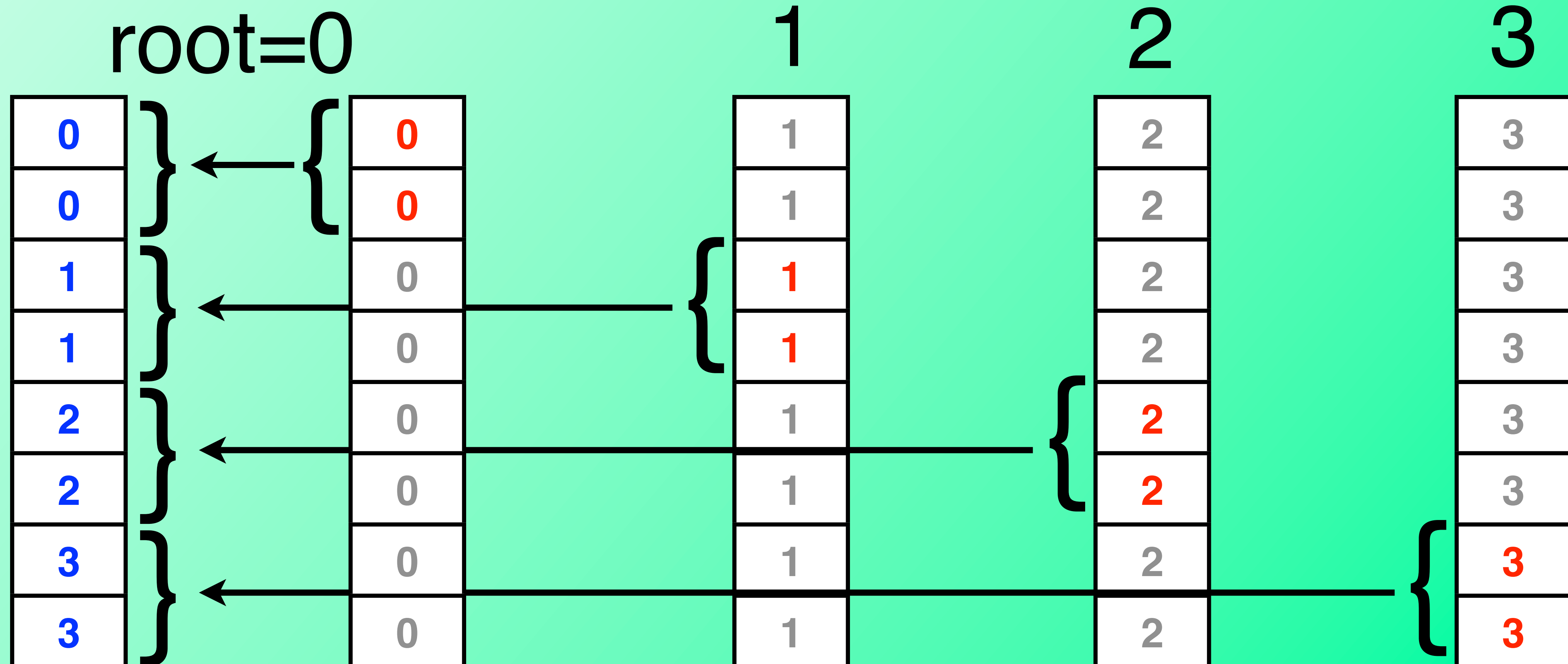
```
C: MPI_Bcast( void *data, int count, MPI_Datatype type,
                 int root, MPI_COMM_WORLD )
F: MPI_BCAST( data, count, type, root, MPI_COMM_WORLD,
              ierr )
```



MPI_Gather

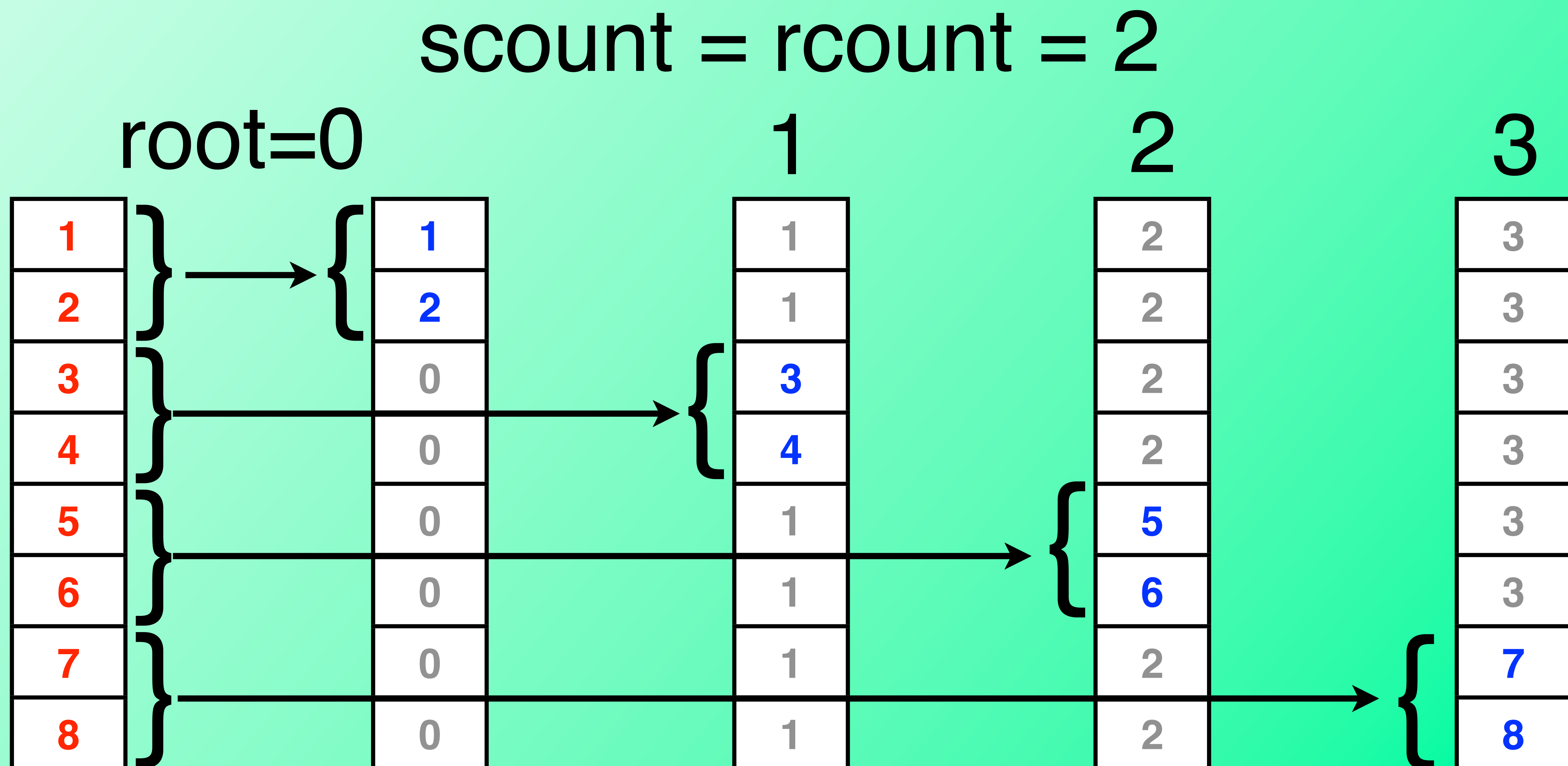
```
C: MPI_Gather( void *sdat, int scount, MPI_Datatype stype,
               void *rdat, int rcount, MPI_Datatype rtype,
               int root, MPI_COMM_WORLD )
F: MPI_GATHER( sdat, scount, stype, rdat, rcount, rtype,
               root, MPI_COMM_WORLD, ierr )
```

scount = rcount = 2



MPI_Scatter

```
C: MPI_Scatter( void *sdat, int scount, MPI_Datatype stype,
                void *rdat, int rcount, MPI_Datatype rtype,
                int root, MPI_COMM_WORLD )
F: MPI_SCATTER( sdat, scount, stype, rdat, rcount, rtype,
                root, MPI_COMM_WORLD, ierr )
```

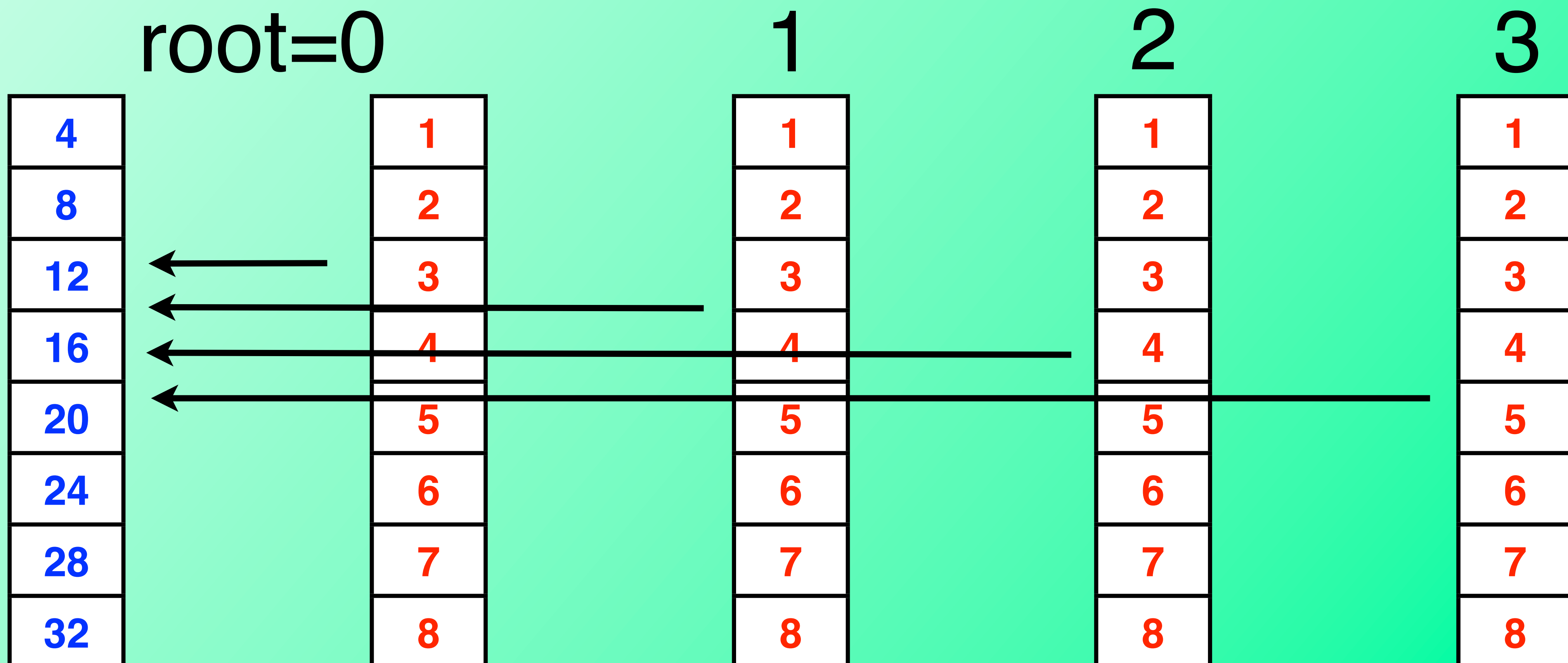


13年8月6日火曜日

MPI_Reduce

```
C: MPI_Reduce( void *sdat, void *rdat, int count,
               MPI_Datatype type, MPI_Op op, int root,
               MPI_COMM_WORLD )
F: MPI_REDUCE( sdat, rdat, count, type, op, root,
               MPI_COMM_WORLD, ierr )
```

count = 8, op = MPI_SUM



MPI_Op

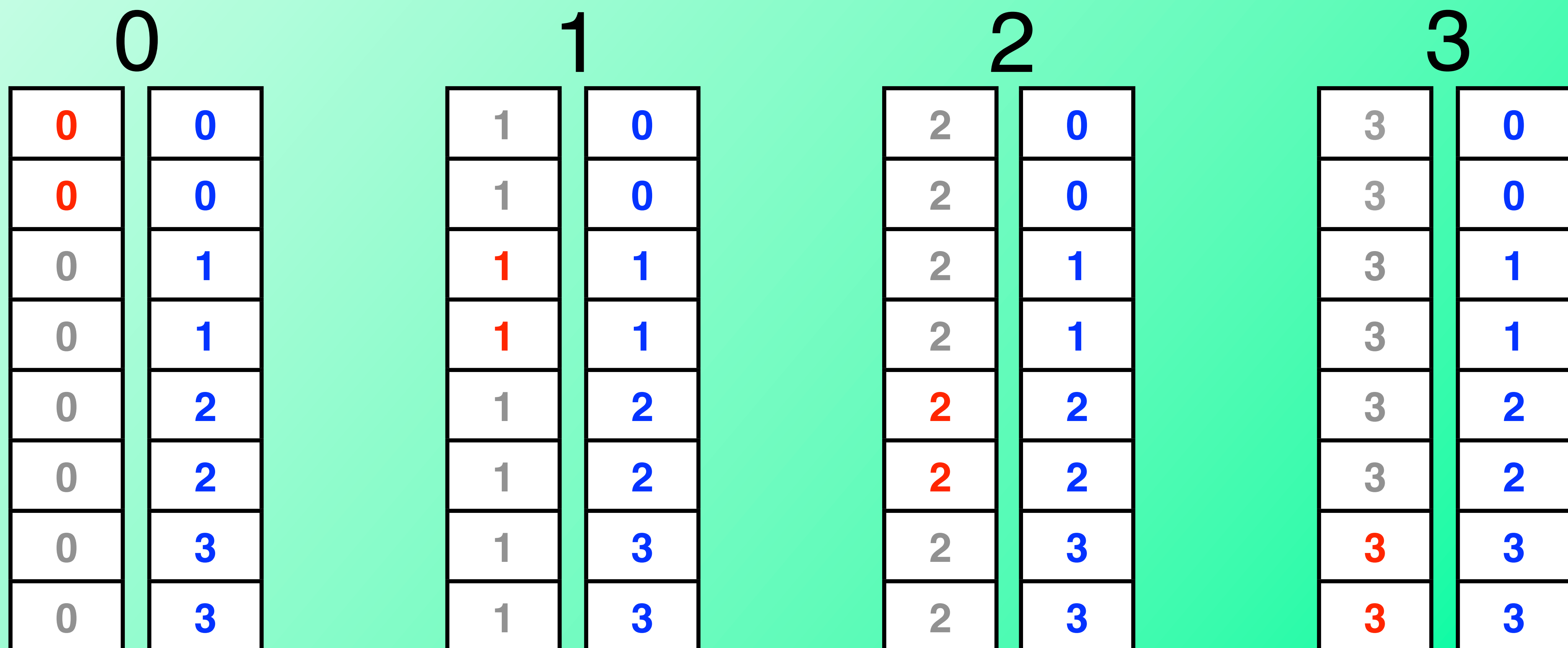
注意：計算の順序は決まっていないので、同じデータでも結果が異なる場合がある！

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

MPI_Allgather

```
C: MPI_Allgather( void *sdat, int scount, MPI_Datatype stype,
                  void *rdat, int rcount, MPI_Datatype rtype,
                  MPI_COMM_WORLD )
F: MPI_ALLGATHER( sdat, scount, stype, rdat, rcount, rtype,
                  MPI_COMM_WORLD, ierr )
```

scount = rcount = 2

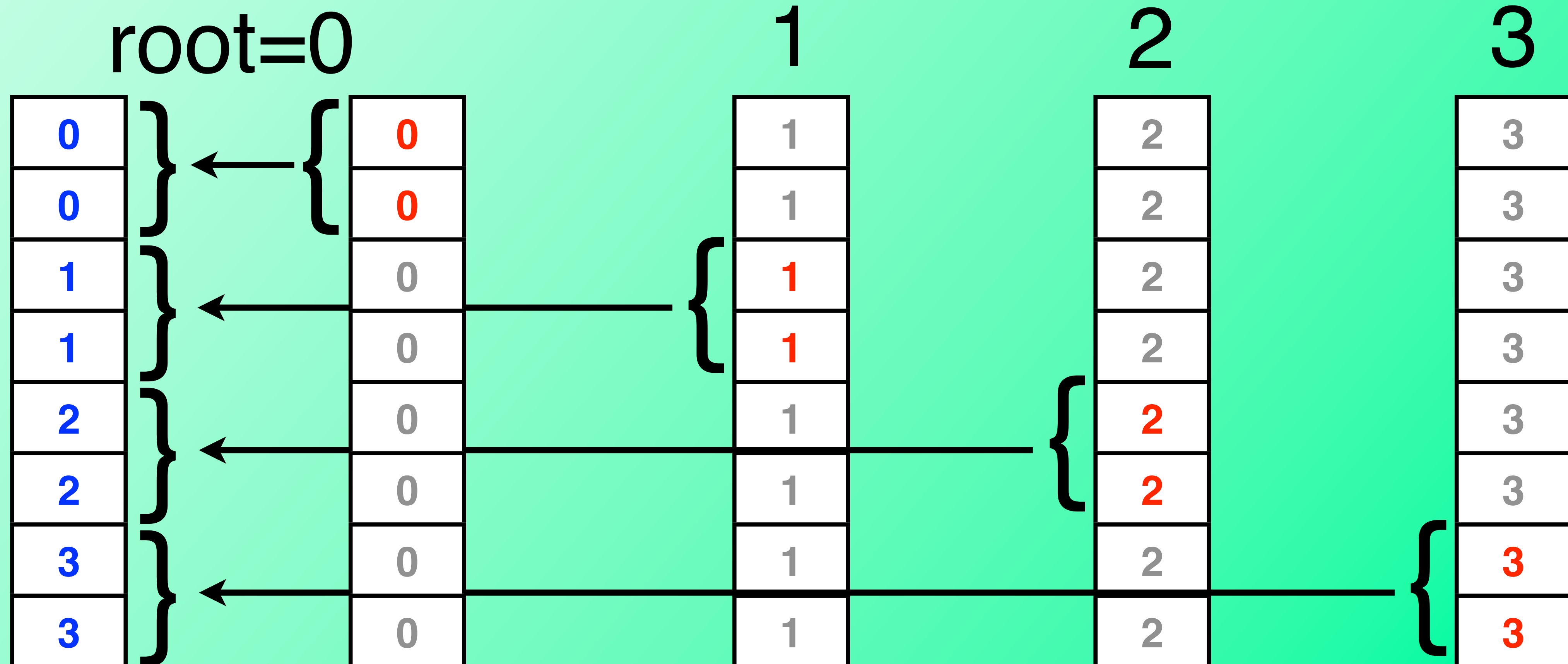


13年8月6日火曜日

MPI_Gather [再掲]

```
C: MPI_Gather( void *sdat, int scount, MPI_Datatype stype,
               void *rdat, int rcount, MPI_Datatype rtype,
               int root, MPI_COMM_WORLD )
F: MPI_GATHER( sdat, scount, stype, rdat, rcount, rtype,
               root, MPI_COMM_WORLD, ierr )
```

scount = rcount = 2



MPI_Allreduce

```
C: MPI_Allreduce( void *sdat, void *rdat, int count,
                  MPI_Datatype type, MPI_Op op,
                  MPI_COMM_WORLD )
F: MPI_ALLREDUCE( sdat, rdat, count, type, op,
                  MPI_COMM_WORLD, ierr )
```

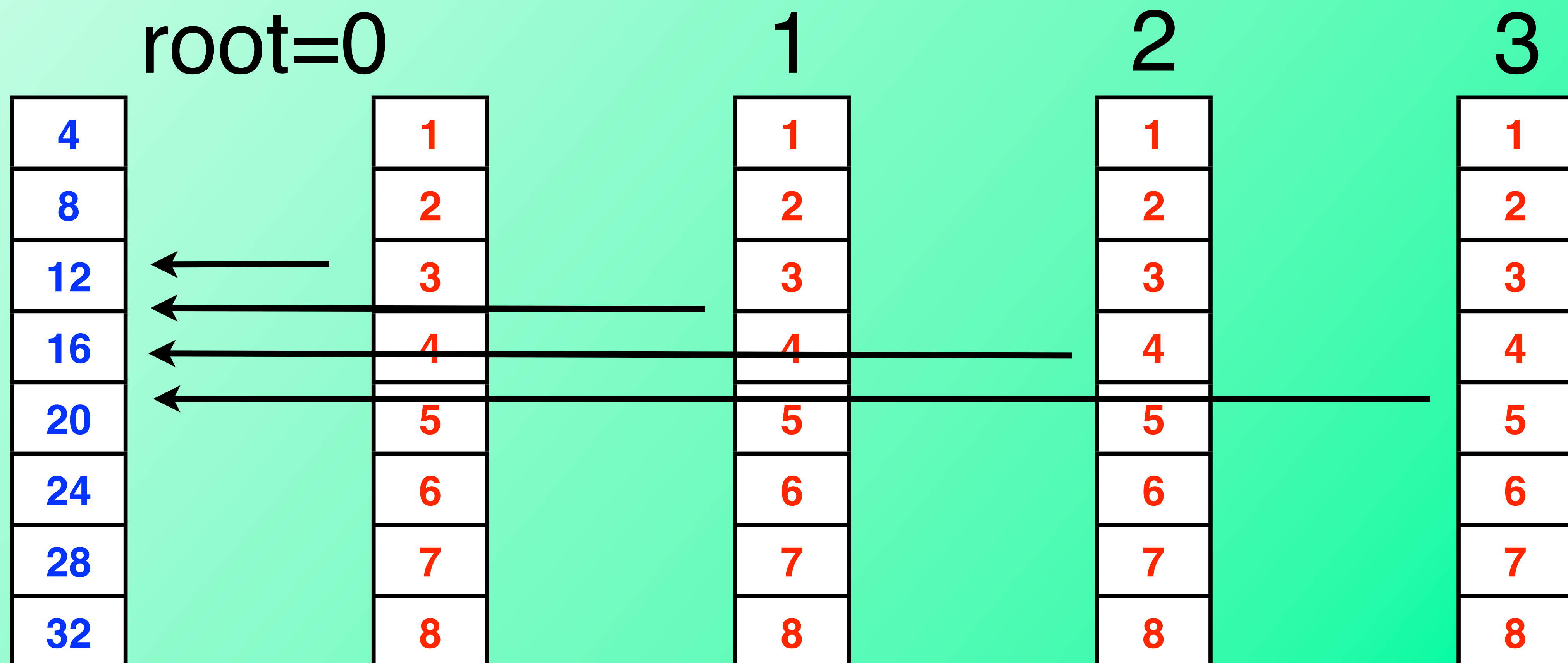
count = 8, op = MPI_PROD

0		1		2		3	
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
2	16	2	16	2	16	2	16
3	81	3	81	3	81	3	81
4	256	4	256	4	256	4	256
5	625	5	625	5	625	5	625
6	1296	6	1296	6	1296	6	1296
7	12201	7	12201	7	12201	7	12201

MPI_Reduce [再掲]

```
C: MPI_Reduce( void *sdat, void *rdat, int count,
               MPI_Datatype type, MPI_Op op, int root,
               MPI_COMM_WORLD )
F: MPI_REDUCE( sdat, rdat, count, type, op, root,
               MPI_COMM_WORLD, ierr )
```

count = 8, op = MPI_SUM



MPI_Allgather() と MPI_Allreduce

- MPI_Allgather() は以下と等価

```
for( root=0; root<N-1; root++ )
    MPI_Gather( ..., root, ... );
```

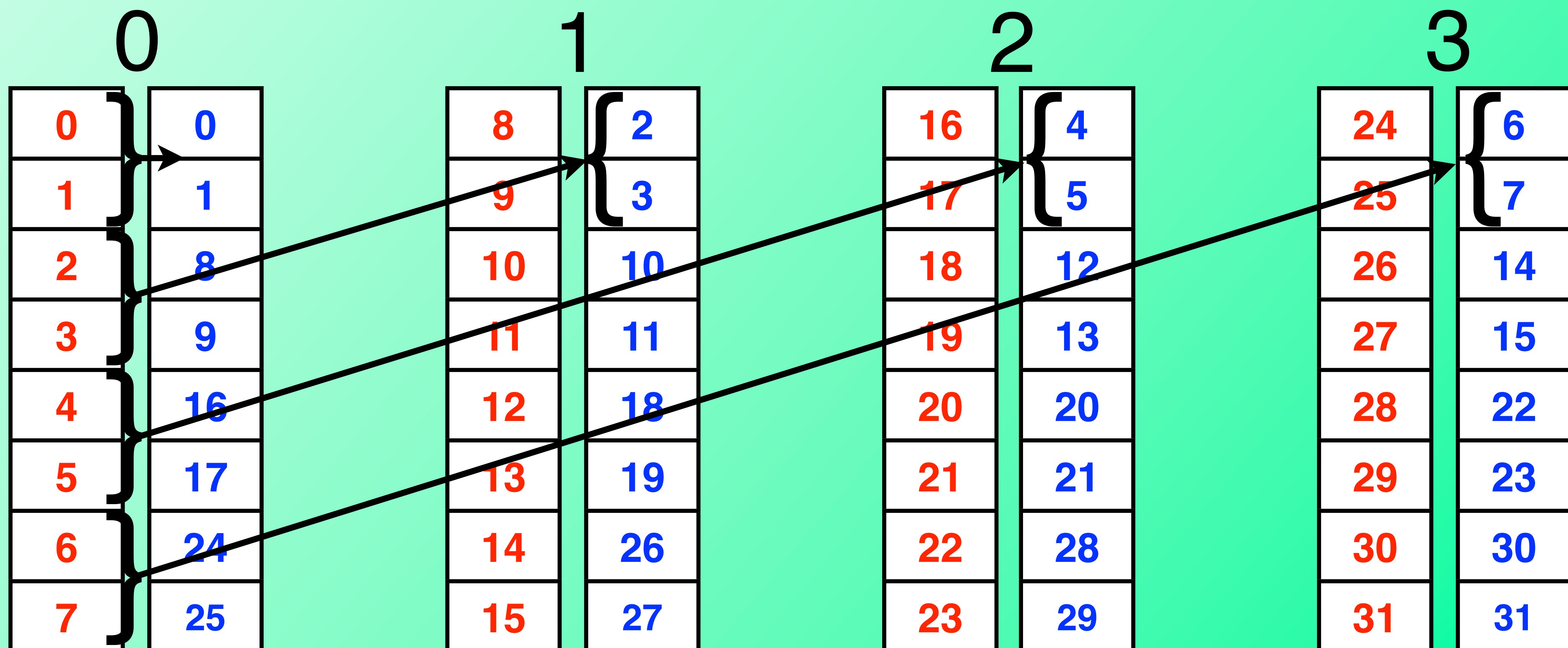
- MPI_Allreduce() は以下と等価

```
for( root=0; root<N-1; root++ )
    MPI_Reduce( ..., root, ... );
```

MPI_Alltoall

```
C: MPI_Alltoall( void *sdat, int scount, MPI_Datatype stype,
                 void *rdat, int rcount, MPI_Datatype rtype,
                 MPI_COMM_WORLD )
F: MPI_ALLTOALL( sdat, scount, stype, rdat, rcount, rtype,
                 MPI_COMM_WORLD, ierr )
```

scount = rcount = 2



MPI_Alltoall

```
C: MPI_Alltoall( void *sdat, int scount, MPI_Datatype stype,
                 void *rdat, int rcount, MPI_Datatype rtype,
                 MPI_COMM_WORLD )
F: MPI_ALLTOALL( sdat, scount, stype, rdat, rcount, rtype,
                 MPI_COMM_WORLD, ierr )
```

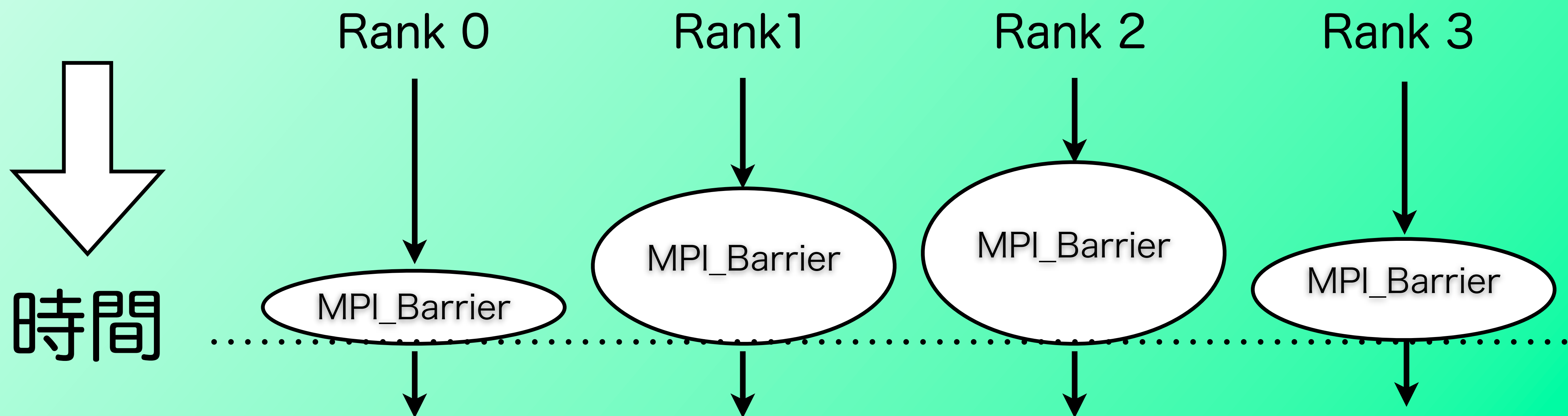
scount = rcount = 2

0		1			2		3	
0	0	8	2	rank 0	16	4	24	6
1	1	9	3		17	5	25	7
2	8	10	10	rank 1	18	12	26	14
3	9	11	11		19	13	27	15
4	16	12	18	rank 2	20	20	28	22
5	17	13	19		21	21	29	23
6	24	14	26	rank 3	22	28	30	30
7	25	15	27		23	29	31	31

MPI_Barrier

```
C: MPI_Barrier( MPI_COMM_WORLD )
F: MPI_BARRIER( MPI_COMM_WORLD, ierr )
```

- 全プロセスを（時間的に）同期する
- MPI_Barrier() 呼出しの前後で、異なるプロセスにおいても、実行順序が入れ替わることはない



Group と Communicator

- (Process) Group
  - プロセスの順序集合
- Communicator (C言語の型：MPI_Comm)
  - 通信の対象となるプロセスグループ
  - 通信の状態を保持する
  - 送受信のマッチ
    - Source/Destination, Tag, Communicator
  - Pre-defined communicator
    - **MPI_COMM_WORLD**：全体
    - **MPI_COMM_SELF**：自分自身のプロセス

Communicator の生成と開放

```
C: MPI_Comm_dup( MPI_Comm comm,  MPI_Comm *new )
   MPI_Comm_free( MPI_Comm *comm )
F: MPI_COMM_DUP( comm, new, ierr)
   MPI_COMM_FREE( comm, ierr )
```

- Group の生成と Group から Communicator を生成する方法 - 省略
- MPI_Comm_dup：複製を作る
  - 同じプロセスグループだが違うコミュニケータを生成する
  - 違うコミュニケータを使うことで、通信を分離できる
- MPI_Comm_free：開放する

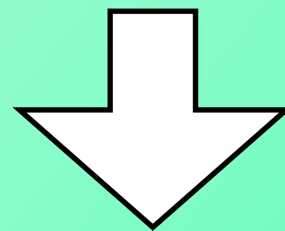
Communicator の分割

```
C: MPI_Comm_split( MPI_Comm comm, int color, int key,
                   MPI_Comm *new )
F: MPI_COMM_SPLIT( COMM, color, key, new, ierr )
```

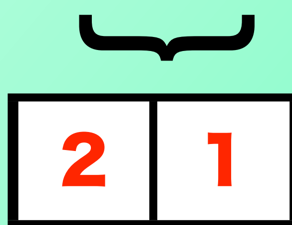
- Communicator comm を 同じ color を持つ（複数の、オーバラップのない）communicator に分割する。分割された communicator における rank 番号は、key の値の小さい順に割り当てられる。Key の値が同じ場合は、システムが適当に rank 番号を割り当てる。

MPI_COMM_SPLITの実行例

Rank	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Color	1	1	1	1	2	2	2	2	7	7	7	7	4	4	4	4
Key	**2**	**2**	5	0	2	6	1	0	2	7	1	0	2	8	1	9



元Rank	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
新Comm	comm0				comm1				comm2				comm3			
新Rank	**1**	**2**	3	0	2	3	1	0	2	3	1	0	1	2	0	3



2	**1**

となる場合もある

MPI Tips

- 集団通信を使えるところは集団通信に
  - ただし、１回しか使わない、1対1通信を、コミュニケータを作ってまで無理に集団通信にする必要はない
- MPI_Send() と MPI_Recv() がペアの場合は MPI_Sendrecv() を使おう
- 1対1通信において MPI_ANY_SRC tag はなるべく使わない

本日のまとめ

- MPI の基礎
  - Point-to-point 通信
  - Collective 通信
  - Communicator

MPI通信の種別

- 1対1通信 (point-to-point communication)
- 集団通信 (collective communication)
- **片方向通信 (one-sided communication)**
  - 次回 8/8

8/8（水）の予告

- ネットワークトポロジーと性能
- 片方向通信
- Derived Data Type
- MPI-IO
  - Collective IO
  - File Pointer
  - File View