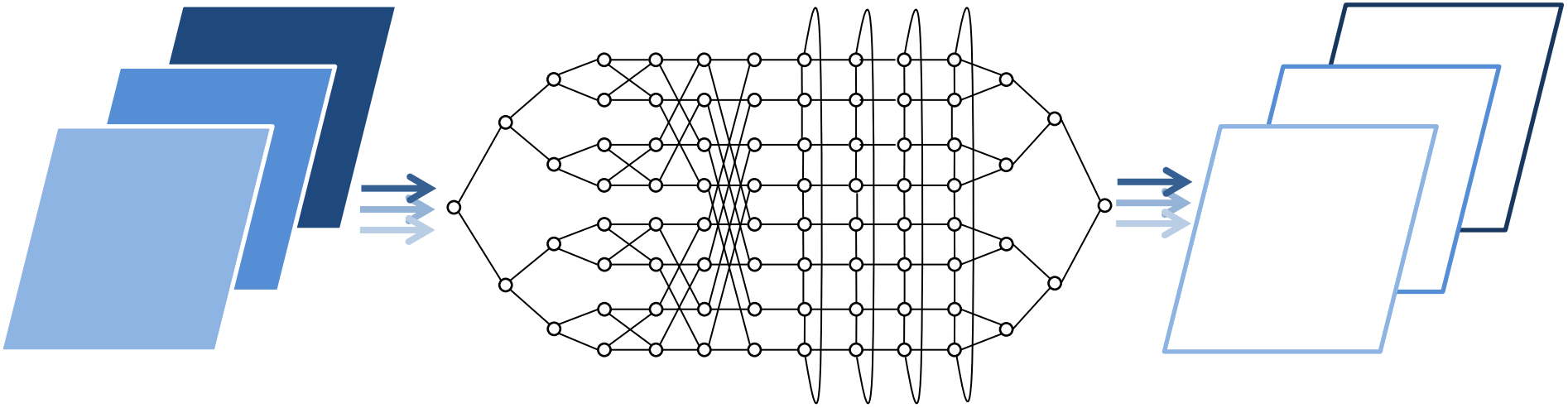
ENSIIE-HPC/BigData

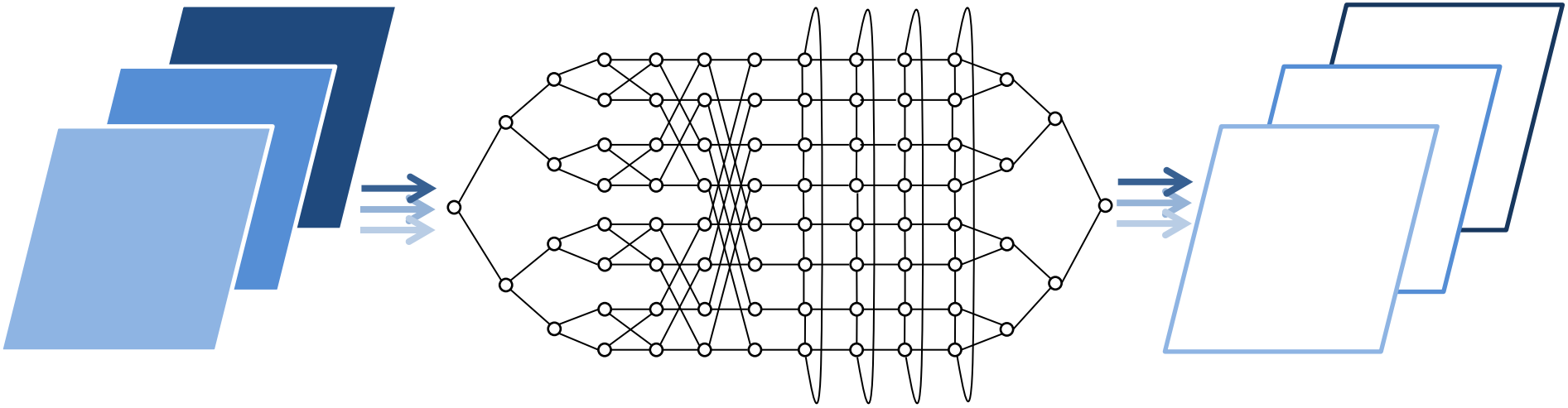
# Programmation Parallèle

## **MPI**: Message Passing Interface



# Collective communications

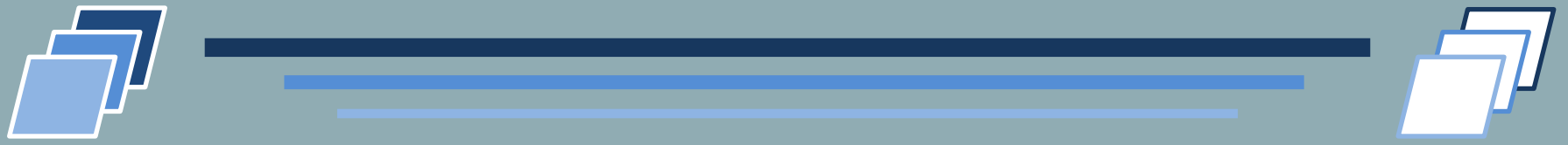
ENSIIE-HPC/BigData-PP-IIP-Lecture 2



# Collectives

- Like p2p, every collective operation requires a communicator
  - Define set of processes that will participate
- All processes in the communicator participate in the collective communication, and have to call the collective function
- All communications which are not p2p are collectives
  - p2p:
    - One-to-one
  - Collectives:
    - One-to-All
    - All-to-One
    - All-to-all

# COLLECTIVE COMMUNICATION



Synchronization

# synchronization

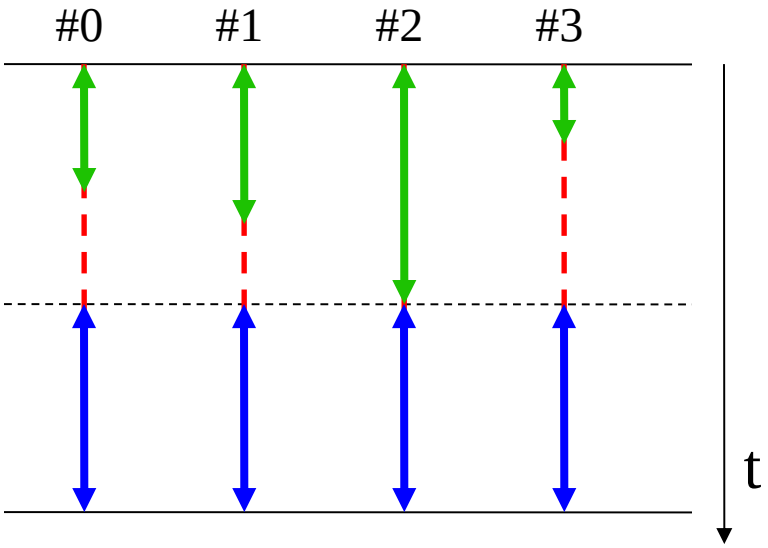
- Communication and synchronization :
  - To ensure the consistency of a calculation, parallel actions can have a « meeting point » before resuming their execution
- These « meeting points » are called **synchronization** :
  - If the synchronization concerns all parallel actions
    - Global or collective synchronization
  - If the actions have different addressing space
    - Communications
- Communications
  - Global synchronization => **global or collective communication**
  - synchronization between 2 actions => **point-to-point communication**

# Barrier

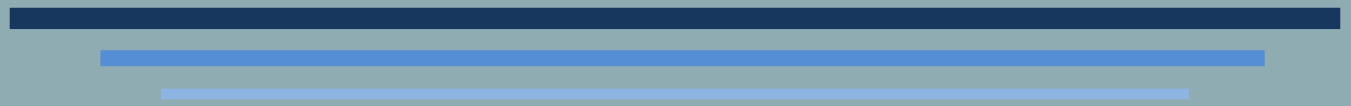
- Synchronize all processes belonging to target communicator

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

```
MPI_Init(&argc, &argv);

/* work 1 */

MPI_Barrier(MPI_COMM_WORLD);

/* work 2 */

MPI_Finalize();
```



# COLLECTIVE COMMUNICATION

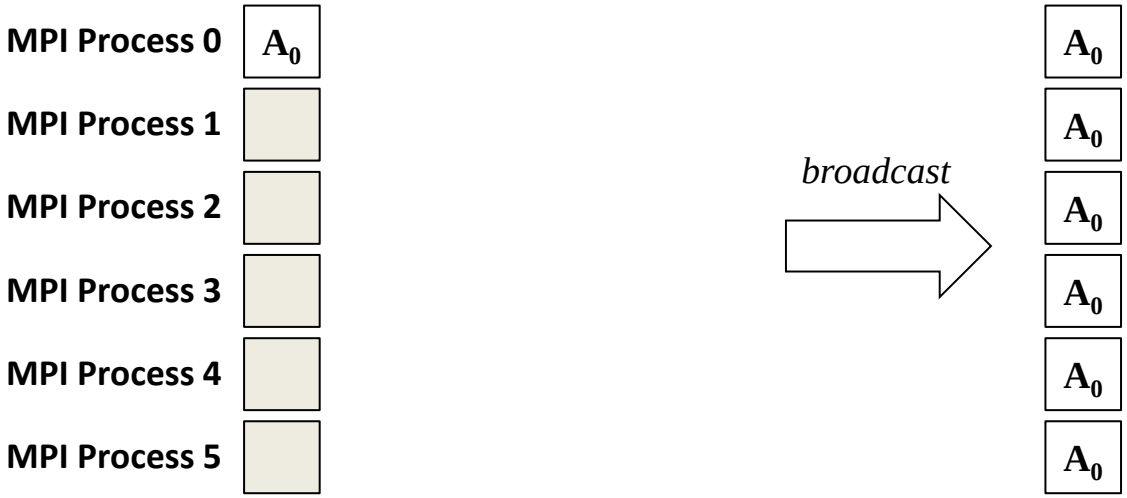


Data Exchange

# Broadcast

- Send data owned by one process to all other processes inside target communicator
- Process emitting data ➔ *root*

- One-to-all collective communication



# Broadcast

```
int MPI_Bcast (
   void *buf(inout),
   int count(in),
   MPI_Datatype datatype(in),
   int root(in),
   MPI_Comm comm(in),
);
```

• rank == **root** → address of memory zone to send
• rank != **root** → address where to store broadcasted data

Output memory segment should be allocated by user.

Size of broadcasted data (number of elements of type datatype).

# Broadcast

```
int MPI_Bcast (
  void *buf(inout),
  int count(in),
  MPI_Datatype datatype(in),
  int root(in),
  MPI_Comm comm(in),
);
```

Root rank.

This rank is valid inside `comm` communicator.

All processes involved in this collective should have the same root.

# Broadcast

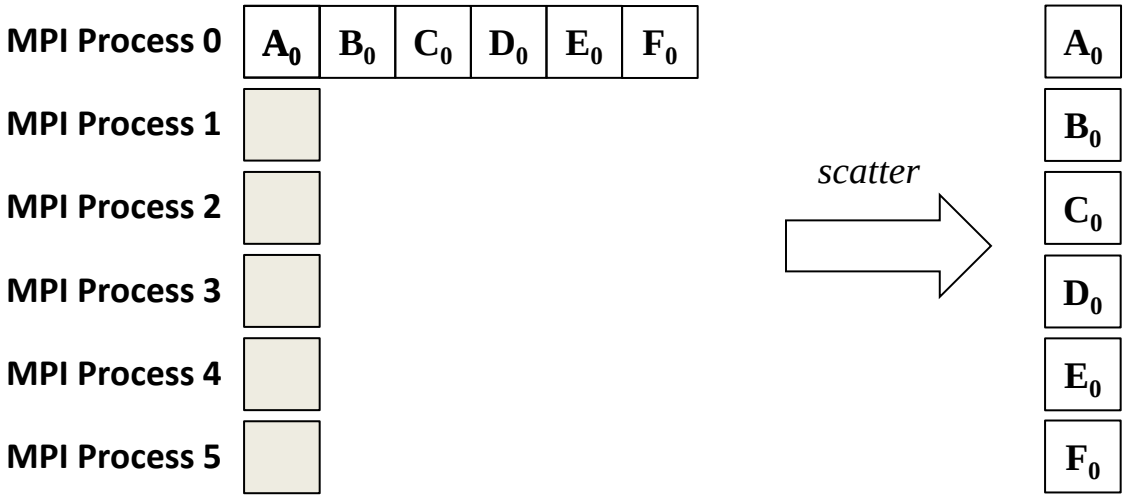
```
int me, root;
float pi = 0.0 ;

root = 0; /* Process 0 is the root */
…
MPI_Comm_rank(MPI_COMM_WORLD, &me);
…
if (me == root)
   pi = 3.14; /* Only root has the right initial value */

/* All processes have to call MPI_Bcast */
MPI_Bcast(&pi, 1, MPI_FLOAT, root, MPI_COMM_WORLD);

printf("P%d: pi = %f\n", me, pi);
```

```
% srun -n 4 ./a.out
P0: pi = 3.14
P3: pi = 3.14
P1: pi = 3.14
P2: pi = 3.14
%
```

# Scatter

- One root send different data to other processes inside target communicator
- Sent data: same size and same type

- One-to-all collective communication



# Scatter

```
int MPI_Scatter (
    void *sendbuf(in),
    int sendcount(in),
    MPI_Datatype sndtatyp(in),
    void *recvbuf(out),
    int recvcount(in),
    MPI_Datatype rcvtatyp(in),
    int root(in),
    MPI_Comm comm(in) );
```

Start address of memory segment to be sent to each process in `comm` communicator.

`sendbuf` is only valid for root process.

# Scatter

```
int MPI_Scatter (
    void *sendbuf(in),
    int sendcount(in),
    MPI_Datatype sndtatyp(in),
    void *recvbuf(out),
    int recvcount(in),
    MPI_Datatype rcvtatyp(in),
    int root(in),
    MPI_Comm comm(in) );
```

Message size to send to each target process.

Thus, sendbuf array is of size sendcount × |comm| elements.

Elements to be sent to rank p start at address sendbuf + $p$× sendcount.

# Scatter

```
int MPI_Scatter (
    void *sendbuf(in),
    int sendcount(in),
    MPI_Datatype sndtatyp(in),
    void *recvbuf(out),
    int recvcount(in),
    MPI_Datatype rcvtatyp(in),
    int root(in),
    MPI_Comm comm(in) );
```

Address where to put received data.

This memory segment should be allocated before calling the collective function.

# Scatter

```
int MPI_Scatter (
    void *sendbuf(in),
    int sendcount(in),
    MPI_Datatype sndtatyp(in),
    void *recvbuf(out),
    int recvcount(in),
    MPI_Datatype rcvtatyp(in),
    int root(in),
    MPI_Comm comm(in) );
```

Size of message to receive (equals to `sendcount`).

Size is in number of elements of type `rcvtatyp`.

# Scatter

```
int MPI_Scatter (
    void *sendbuf(in),
    int sendcount(in),
    MPI_Datatype sndtatyp(in),
    void *recvbuf(out),
    int recvcount(in),
    MPI_Datatype rcvtatyp(in),
    int root(in),
    MPI_Comm comm(in) );
```

Root rank.

This rank should be valid inside communicator comm.

All processes should have the same root.

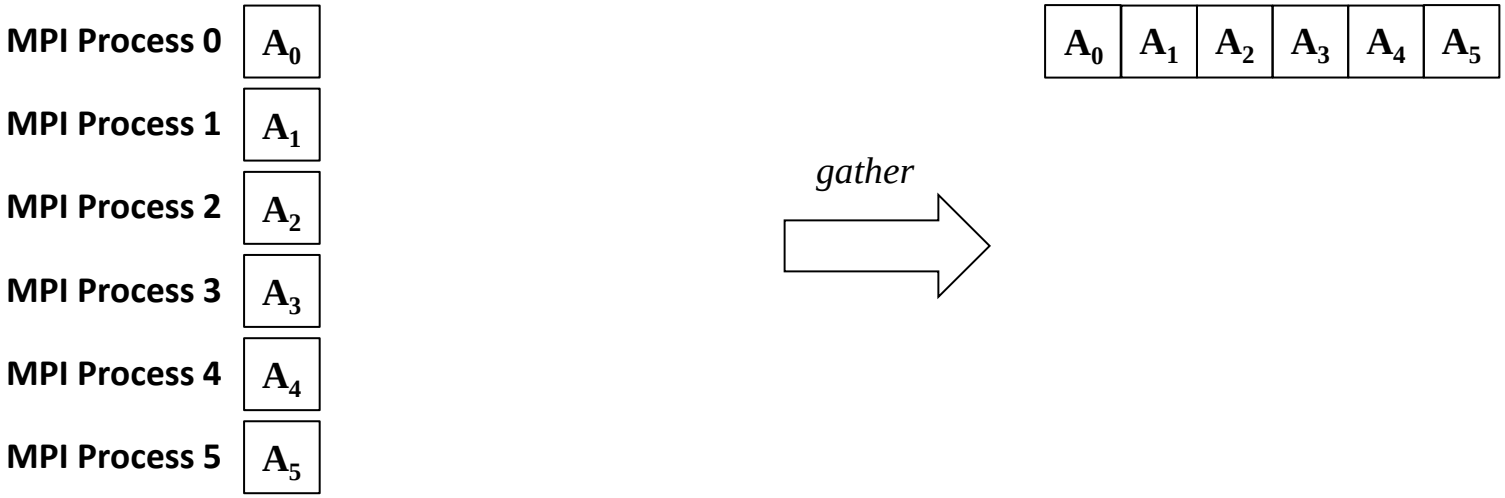
# Scatter

```
int me, v, root, P;
int *sdbuf;

root = 0; /* Process 0 is the root */

MPI_Comm_rank(MPI_COMM_WORLD, &me);
MPI_Comm_size(MPI_COMM_WORLD, &P);

if (me == root) {
   sdbuf = (int*)malloc(P*sizeof(int));
   fd = fopen(« my_data", "r");
   for( p = 0 ; p < P ; p++ )
      sdbuf[p] = read_file_value(fd, p);
   fclose( fd );
} else { sdbuf = NULL; }

MPI_Scatter(sdbuf, 1, MPI_INT,
            &v, 1, MPI_INT, root, MPI_COMM_WORLD);

printf("P%d: received value = %d\n", me, v);
```

```
% cat my_data
18
25
6
3

% srun -n 4 ./a.out
P0: received value = 18
P3: received value = 3
P1: received value = 25
P2: received valud = 6
%
```

# Gather

- Root process collects data from other processes (reverse of scatter)
- Sent data: same size and same type
- All-to-one collective communication



# Gather

```
int MPI_Gather (
  void *sendbuf(in),
  int sendcount(in),
  MPI_Datatype sndtatyp(in),
  void *recvbuf(out),
  int recvcount(in),
  MPI_Datatype rcvtatyp(in),
  int root(in),
  MPI_Comm comm(in) );
```

Address of memory segment containing data to send to root process.

# Gather

```
int MPI_Gather (
  void *sendbuf(in),
  int sendcount(in),
  MPI_Datatype sndtatyp(in),
  void *recvbuf(out),
  int recvcount(in),
  MPI_Datatype rcvtatyp(in),
  int root(in),
  MPI_Comm comm(in) );
```

Size (in number of elements of type `sndtatyp`) of message to send to root process.

# Gather

```
int MPI_Gather (
    void *sendbuf(in),
    int sendcount(in),
    MPI_Datatype sndtatyp(in),
    void *recvbuf(out),
    int recvcount(in),
    MPI_Datatype rcvtatyp(in),
    int root(in),
    MPI_Comm comm(in) );
```

Address of memory segment to receive data from each process inside `comm` communicator.

Only valid for root.

# Gather

```
int MPI_Gather (
    void *sendbuf(in),
    int sendcount(in),
    MPI_Datatype sndtatyp(in),
    void *recvbuf(out),
    int recvcount(in),
    MPI_Datatype rcvtatyp(in),
    int root(in),
    MPI_Comm comm(in) );
```

Size of data to receive (per process).

Array `recvbuf` must be of size `recvcount` × |`comm`| elements.

# Gather

```
int MPI_Gather (
    void *sendbuf(in),
    int sendcount(in),
    MPI_Datatype sndtatyp(in),
    void *recvbuf(out),
    int recvcount(in),
    MPI_Datatype rcvtatyp(in),
    int root(in),
    MPI_Comm comm(in) );
```

Root rank.

This rank is valid inside `comm` communicator

All processes must use the same root.

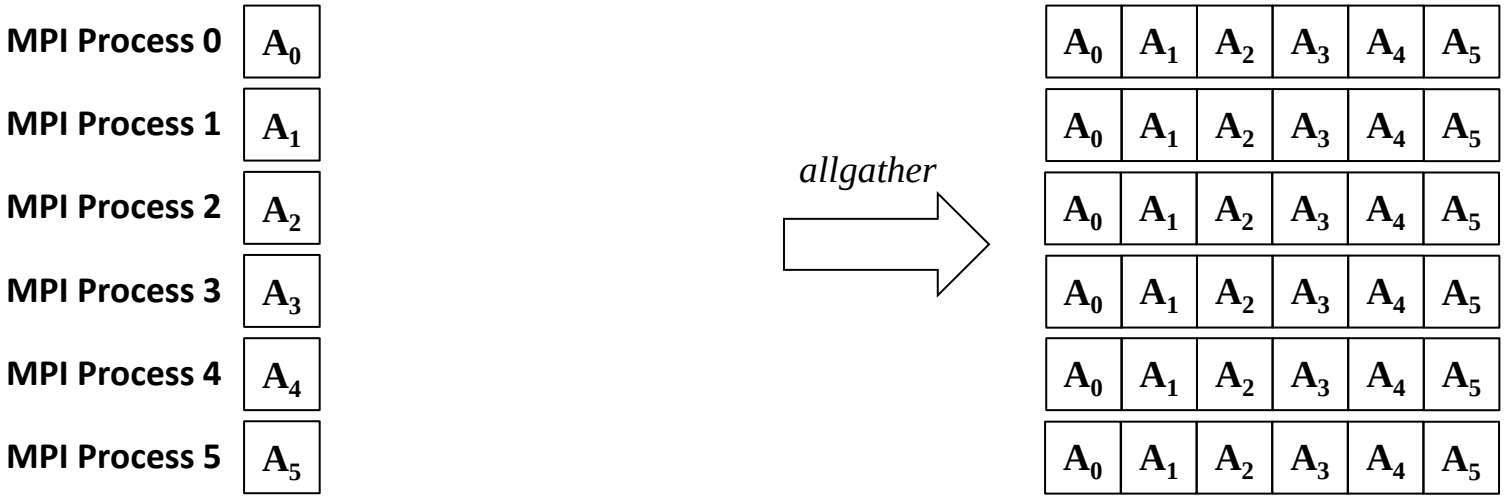
# Gather

```
int me, v, root, P; int *rcvbuf;
root = 0; /* Process 0 is the root */

MPI_Comm_rank(MPI_COMM_WORLD, &me);
MPI_Comm_size(MPI_COMM_WORLD, &P);
…
if (me == root)
   rcvbuf = (int*)malloc(P*sizeof(int));
else rcvbuf = NULL;

v = … ;
MPI_Gather(&v, 1, MPI_INT,
           rcvbuf, 1, MPI_INT, root, MPI_COMM_WORLD);

if (me == root) {
    fd = fopen("results", "w");
    for( p = 0 ; p < P ; p++ )
       write_file_value(fd, p, rcvbuf[p]);
    fclose( fd );
}
```

```
% srun -n 4 a.out
% cat results
30
-100
23
19
```

# Allgather

- Equivalent to gather collective operation except that every process involved inside the communication receives the final result.
- Allgather ≈ Gather + Broadcast



# Allgather

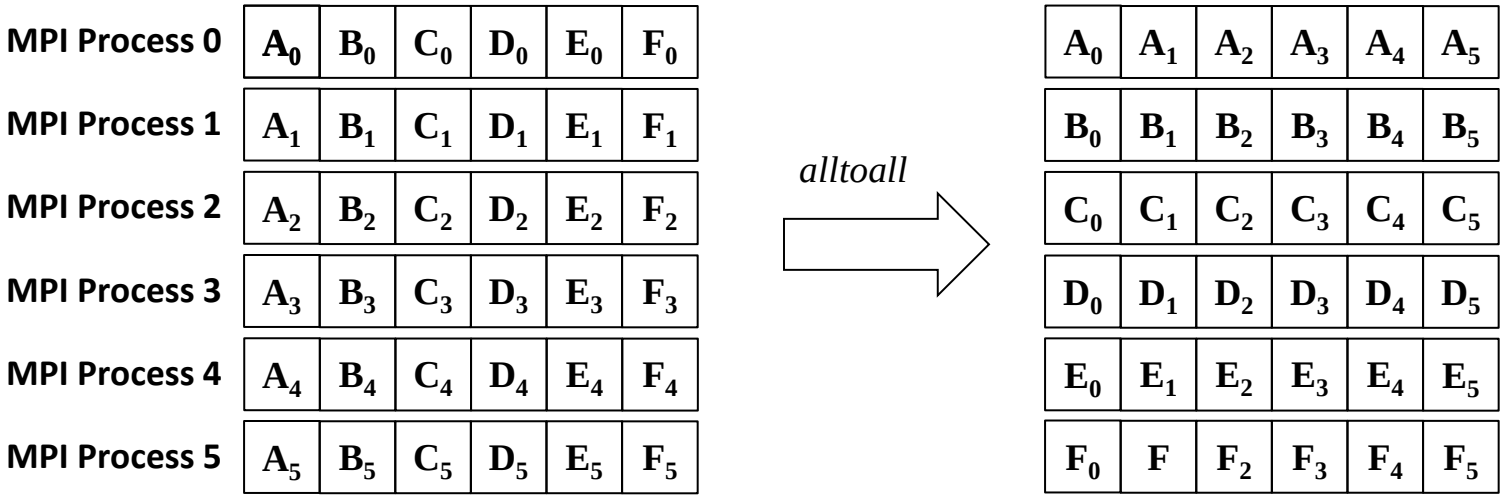
```
int MPI_Allgather (
  void *sendbuf(in),
  int sendcount(in),
  MPI_Datatype sndtatyp(in),
  void *recvbuf(out),
  int recvcount(in),
  MPI_Datatype rcvtatyp(in),
  MPI_Comm comm(in),
);
```

Arguments corresponding to sent data

Arguments corresponding to received data

# Alltoall

- Every Process sends and receives data
- Each process send its $i^{th}$ pack of data to the $i^{th}$ rank
- Realize a sort of matrix transpose



# Alltoall

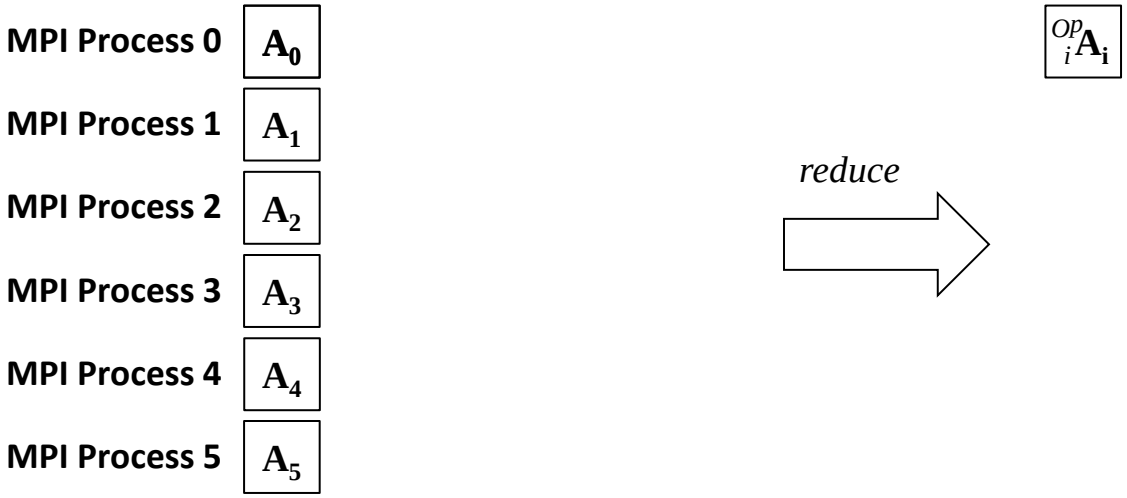
```
int MPI_Alltoall (
  void *sendbuf(in),
  int sendcount(in),
  MPI_Datatype sendtype(in),
  void *recvbuf(out),
  int recvcount(in),
  MPI_Datatype recvtype(in),
  MPI_Comm comm(in),
);
```

Arguments corresponding to sent data

Arguments corresponding to received data

# Reduction

- Root rank collects data from other processes and apply one specific operation
- MPI allows multiple reductions at the same time



# Reduction

```
int MPI_Reduce (
  void *sendbuf(in),
  void *recvbuf(out),
  int count(in),
  MPI_Datatype datatype(in),
  MPI_Op op(in),
  int root(in),
  MPI_Comm comm(in),
);
```

Address of data to send to root process.

Array of `count` elements of `datatype` type.

# Reduction

```
int MPI_Reduce (
  void *sendbuf(in),
  void *recvbuf(out),
  int count(in),
  MPI_Datatype datatype(in),
  MPI_Op op(in),
  int root(in),
  MPI_Comm comm(in),
);
```

Address of final data (after reduction).

Only valid for root process.

Array of `count` elements of `datatype` type.

# Reduction

```
int MPI_Reduce (
  void *sendbuf(in),
  void *recvbuf(out),
  int count(in),
  MPI_Datatype datatype(in),
  MPI_Op op(in),
  int root(in),
  MPI_Comm comm(in),
);
```

Operation to perform on each element of every process.

Upon return, root will have:
recvbuf[i] = $op_{0 \le p < P}$ $sendbuf_p$ [i]
with $0 \le i <$ count
and P size of comm.

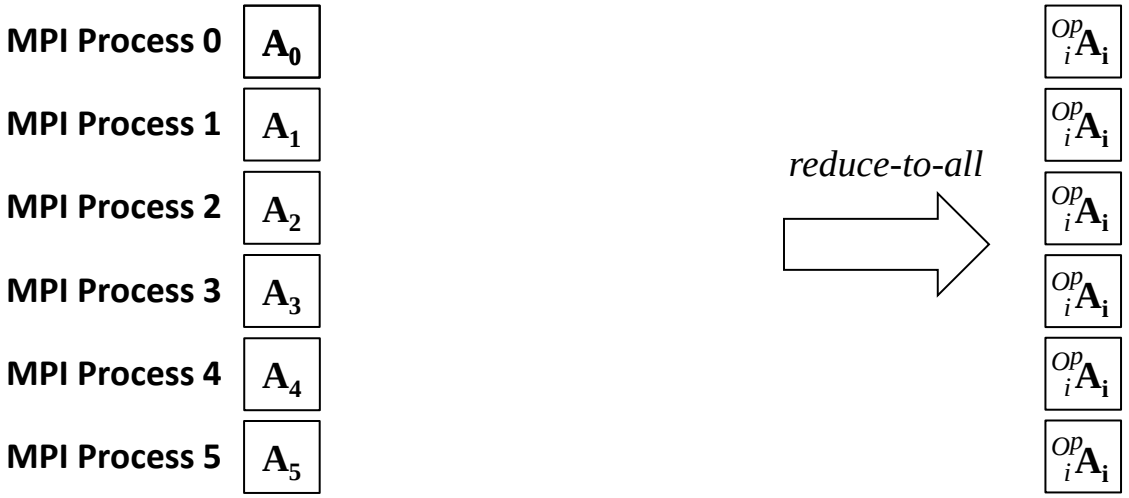
# Reduction

- Reduction operations are of type `MPI_Op`
- Possibility to create user-defined ones: `MPI_Op_create()`

| ***MPI Operation*** | ***Meaning*** | ***Corresponding Type*** |
|---|---|---|
| `MPI_MAX` | Maximum | Integers and real |
| `MPI_MIN` | Minimum | Integers and real |
| `MPI_SUM` | Sum | Integer and real |
| `MPI_PROD` | Product | Integer and real |
| `MPI_LAND` | Logical AND | Integer |
| `MPI_BAND` | Binary AND | Integer and `MPI_BYTE` |
| `MPI_LOR` | Logical OR | Integer |
| `MPI_BOR` | Binary OR | integer `MPI_BYTE` |
| `MPI_LXOR` | Logical XOR | Integer |
| `MPI_BXOR` | Binary XOR | Integer and `MPI_BYTE` |
| `MPI_MAXLOC` | Maximum w/ index | Structures w/ 2 integers |
| `MPI_MINLOC` | Minimum w/ index | Structures w/ 2 integers |

# Reduce to All

- Equivalent to a reduction
- But every process has the final result
- No need to specify a root process
- Allreduce ≈ Reduce + Broadcast



# Reduce to All

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

Same signature as `MPI_Reduce` but no root is needed.

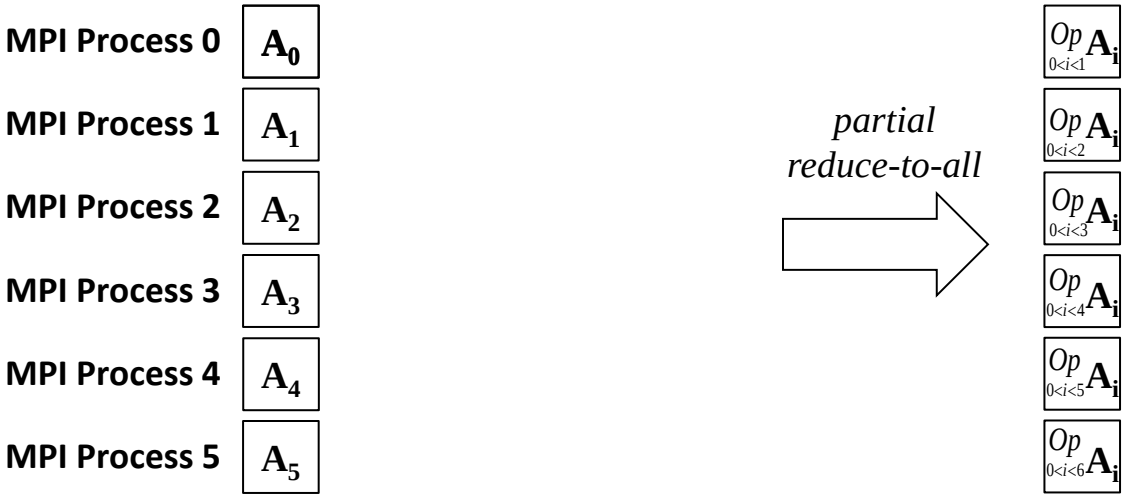
# Reduce to All

```
int P, N = 100;
int me, i, sum, sum_loc = 0;

MPI_Comm_rank(MPI_COMM_WORLD, &me);
MPI_Comm_size(MPI_COMM_WORLD, &P);
…
/* Works if N%P == 0 */
for( i = 1 + me*N/P ; i <= (me+1)*N/P ; i++ )
   sum_loc += i;

MPI_Allreduce(&sum_loc, &sum_glob, 1, MPI_INT,
              MPI_SUM, MPI_COMM_WORLD);

/* After reduction, all processes have, in sum_glob, the sum of N first integers
*/

printf("1+…+%d = %d\n", N, sum_glob);
```

```
% srun -p 4 ./a.out
1+…+100 = 5050
1+…+100 = 5050
1+…+100 = 5050
1+…+100 = 5050
%
```

# Partial Reduction

- All ranks collect data from other processes **with smaller rank number** and apply one specific operation, including its contribution
- MPI allows multiple reductions as the same time

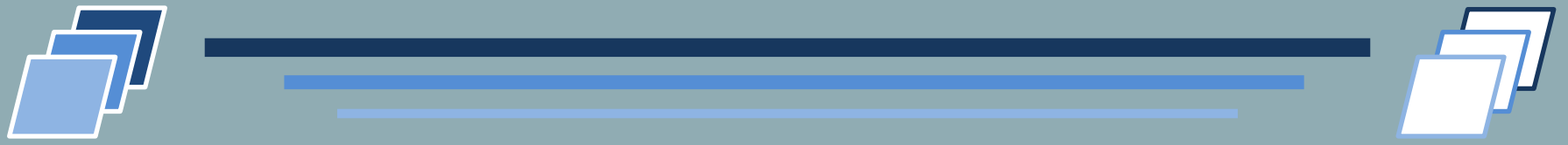


# Partial Reduction

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

Same signature as MPI_Allreduce

# COLLECTIVE COMMUNICATIONS



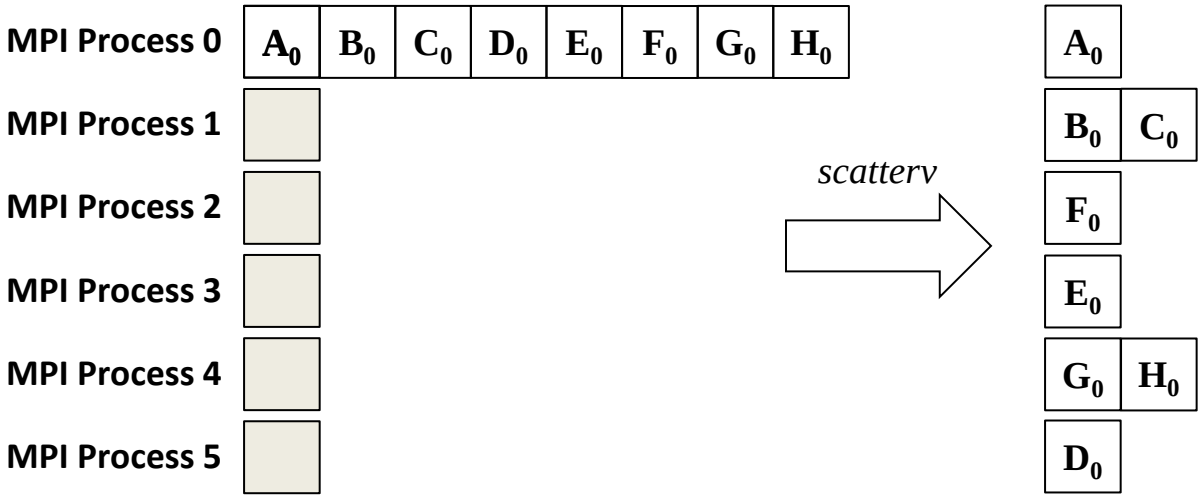
Multiple Sizes

# Per-Rank Data Size

- Some collective communications propose multiple versions including one to handle different size for different ranks.
  - E.g., Broadcast, Gather
- Corresponding names have the suffix `v`
  - `v` = varying or vector or variant
- Examples
  - `MPI_Gather` → `MPI_Gatherv`
  - `MPI_Allgather` → `MPI_Allgatherv`
  - `MPI_Scatter` → `MPI_Scatterv`
  - `MPI_Alltoall` → `MPI_Alltoallv`

# MPI_Scatterv

- Data $A_i$ have the same type
- Unlike `MPI_Scatter`, $A_0$, $A_1$… may not have the same size
  - Need another argument to specify size of each data
- Unlike `MPI_Scatter`, data Ai may not be contiguous (in memory)
  - Need another argument to specify base address of each data



# MPI_Scatterv

```
int MPI_Scatterv (
    void *sendbuf(in),
    int *sendcounts(in),
    int *displs(in),
    MPI_Datatype sendtype(in),
    void *recvbuf(out),
    int recvcount(in),
    MPI_Datatype recvtype(in),
    int root(in), MPI_Comm comm(in) );
```

sendbuf = array of data to distribute to each process inside communicator comm.

sendcounts[$p$] elements have to send to process $p$ ($0 \leq p < P$).
Those elements are store at address

sendbuf + displs[$p$] .

All elements are of type sendtype.

Those arguments are only valid for root.

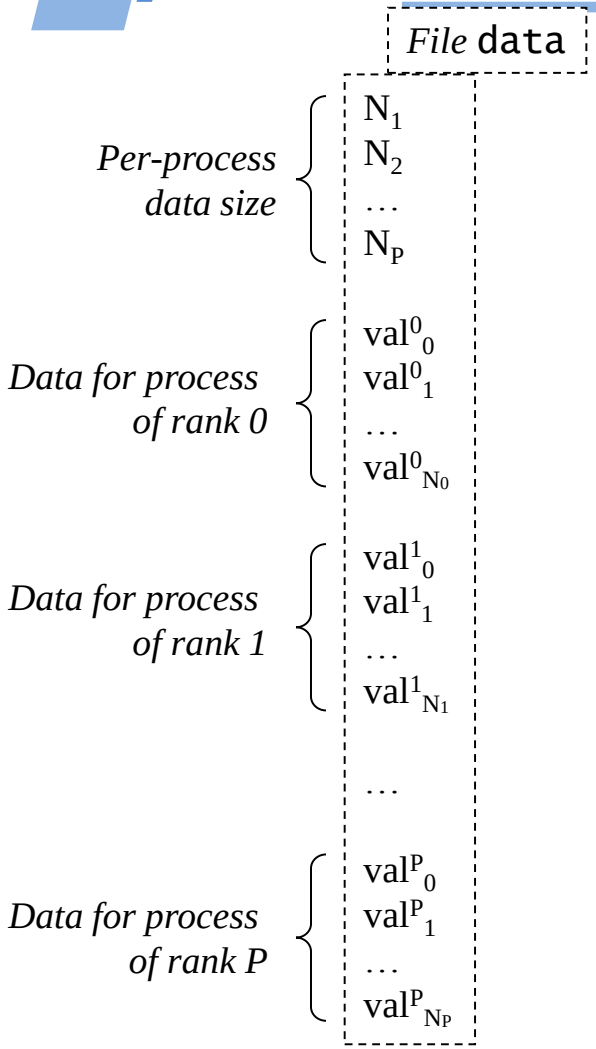
# MPI_Scatterv

```
int MPI_Scatterv (
    void *sendbuf(in),
    int *sendcounts(in),
    int *displs(in),
    MPI_Datatype sendtype(in),
    void *recvbuf(out),
    int recvcount(in),
    MPI_Datatype recvtype(in),
    int root(in), MPI_Comm comm(in) );
```

recvbuf = array of data to receive.

Array with recvcount elements of type rcvtatyp.

# MPI_Scatterv




- One process reads a file named data
- This file contains data for each process
- File format is described on the left-hand-side
  - Data size for each process
  - Data (e.g., float values) stored in a per-process manner
- After parsing step, all processes get their own piece of data

# MPI_Scatterv

```
int sz; int *szbuf, *displs;
double *sdbuf, *rcvbuf;

if (me == root) {
   szbuf = (int*)malloc(P*sizeof(int));
   displs = (int*)malloc((P+1)*sizeof(int));
   fd = fopen("data", "r");
   displs[0] = 0;
   for( p = 0 ; p < P ; p++ ) {
      szbuf[p] = read_int_value(fd, p);
      displs[p+1] = displs[p] + szbuf[p];
   }
   sdbuf = (double*)malloc(displs[P]*sizeof(double));
   for( p = 0 ; p < P ; p++ )
       for( i = 0 ; i < szbuf[p] ; i++ )
           sdbuf[displs[p]+i] = read_float_value(fd, p);
   fclose( fd );
}
```

```
/* Read data sizes for each
process (szbuf array)
Compute corresponding offset
(displs array) */
```

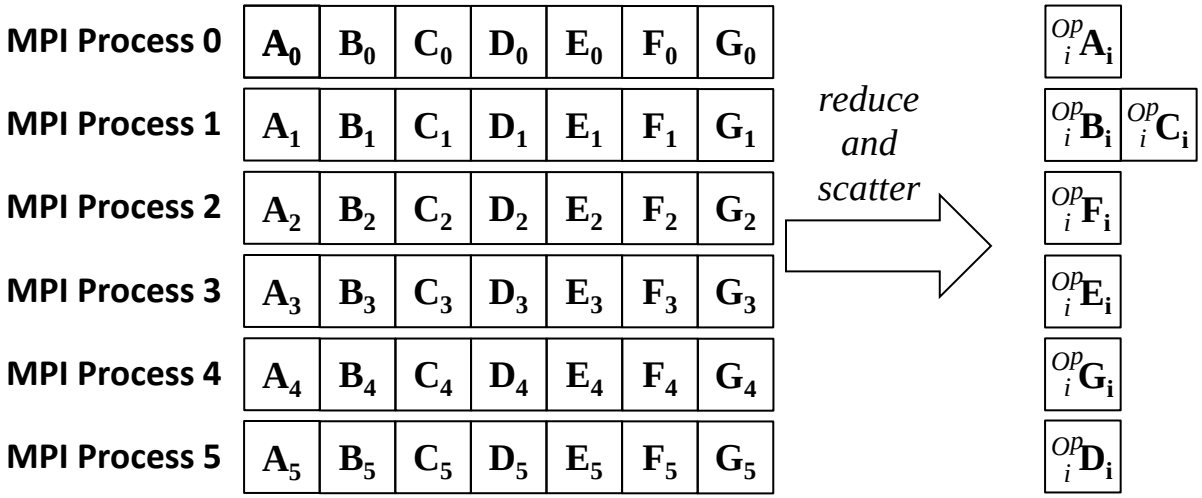
```
/* Read per-block
data (sdbuf array).
Blocks will be
distributed amon
ranks */
```

# MPI_Scatterv

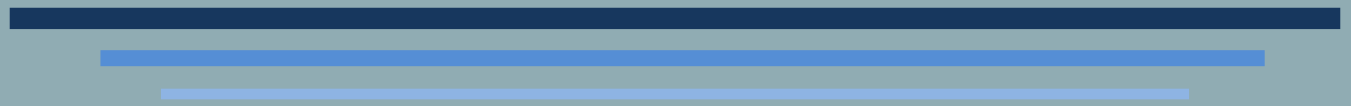
```
else {
   szbuf = displs = NULL;
   sdbuf = NULL;
}
/* All processes must call MPI_Scatter
   Corresponding data size is stored into sz
 */
MPI_Scatter(szbuf, 1, MPI_INT,
            &sz, 1, MPI_INT,
            root, MPI_COMM_WORLD);

rcvbuf = (double*)malloc(sz*sizeof(double));
/* All process must call MPI_Scatterv
   Corresponding data are stored in rcvbuf
 */
MPI_Scatterv(sdbuf, szbuf, displs, MPI_DOUBLE,
             rcvbuf, sz, MPI_DOUBLE,
             root, MPI_COMM_WORLD);
```

# Reduce-Scatter

- Combine a reduction and a scatterv
- Reduction on an array
- Every process gets a piece of array
- Like `MPI_Scatterv`, $A_0$, $A_1$… may not have the same size on the scatter part



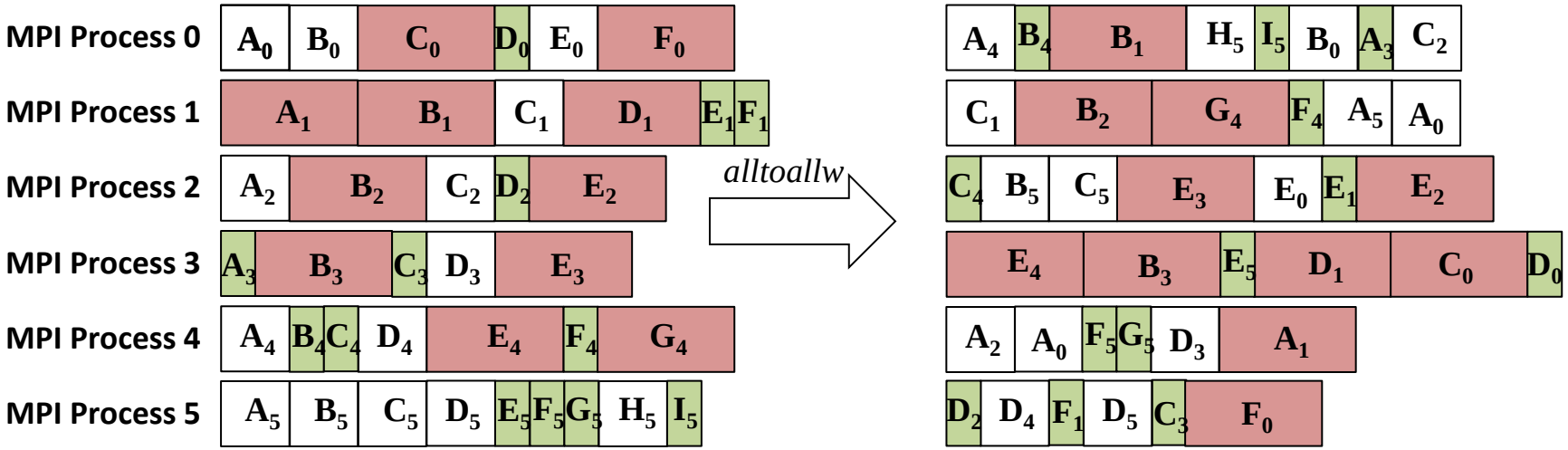
# THE THREE MISTAKES



Alltoallw, Exscan, Scatter_Reduce_block

# Alltoallw

- Like `MPI_Scatterv`, $A_0$, $A_1$… may not have the same size
  - Need another argument to specify size of each data
- Like `MPI_Scatterv`, data Ai may not be contiguous (in memory)
  - Need another argument to specify base address of each data
- In addition, data $A_i$ may not have the same type
  - Datatype argument is now an array



# MPI_Alltoallw

```
int MPI_Alltoallw (
    void *sendbuf(in),
    int *sendcounts(in),
    int *displs(in),
    MPI_Datatype *sendtypes(in),
    void *recvbuf(out),
    int *recvcounts(in),
    MPI_Datatype *recvtypes(in),
    int root(in), MPI_Comm comm(in) );
```

sendbuf = array of data to distribute to each process inside communicator comm.

sendcounts[$p$] elements have to send to process $p$ ($0 \leq p < P$).
Those elements are store at address

sendbuf + displs[$p$] .

sendtype[$p$] is the type of elements to send to process p

# MPI_Alltoallw

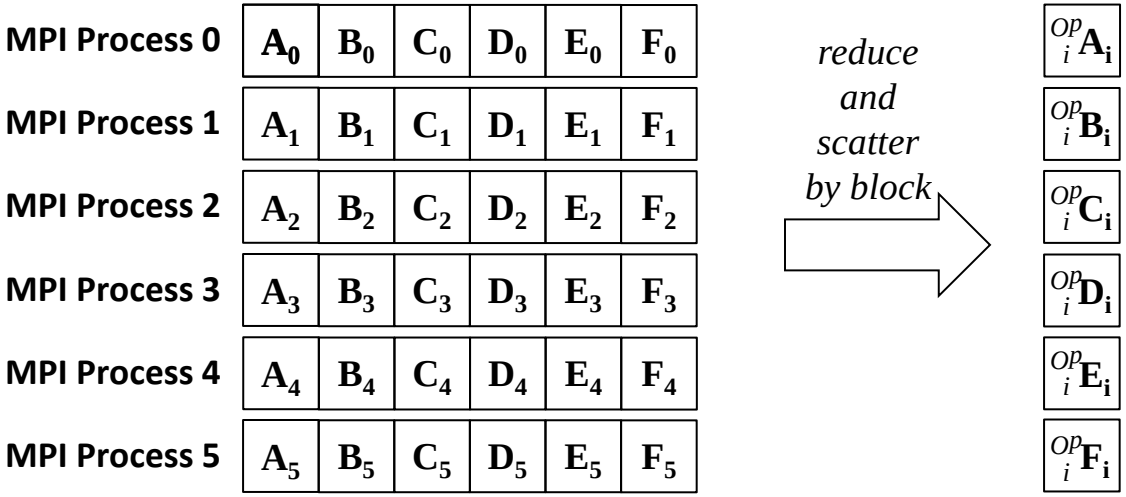
```
int MPI_Alltoallw (
    void *sendbuf(in),
    int *sendcounts(in),
    int *displs(in),
    MPI_Datatype *sendtypes(in),
    void *recvbuf(out),
    int *recvcounts(in),
    MPI_Datatype *recvtypes(in),
    int root(in), MPI_Comm comm(in) );
```

recvbuf = array of data to receive.

Array composed of recvcount[*p*] elements of type recvtype[*p*] coming from all other processes *p*.

# Reduce-Scatter block

- Combine a reduction and a scatter
- Reduction on an array
- Every process gets a piece of array of same size
- Data chunk are contiguous (no displacement arg)



# Reduction-Scatter block

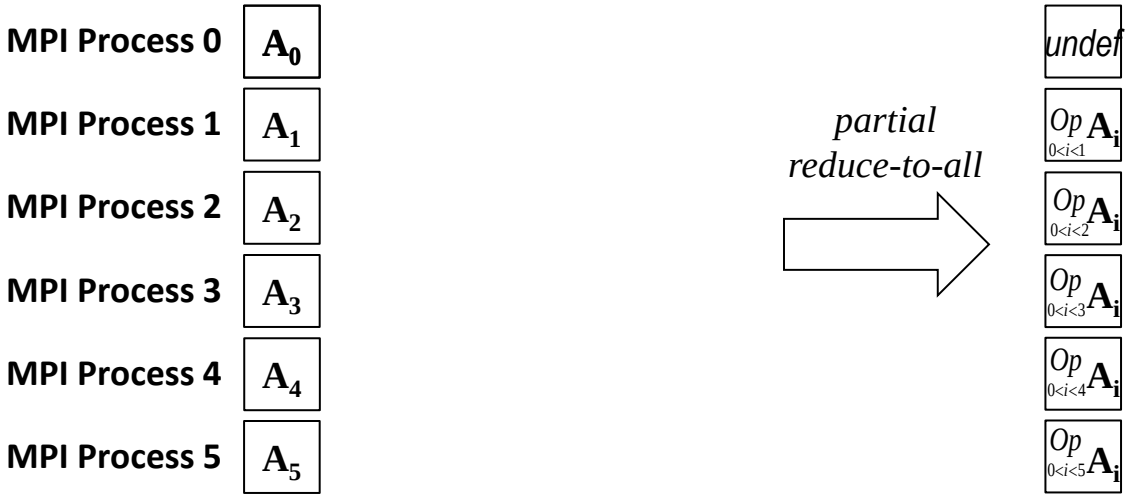
```
int MPI_Reduce_Scatter_block (
   void *sendbuf(in),
   void *recvbuf(out),
   int recvcount(in),
   MPI_Datatype datatype(in),
   MPI_Op op(in),
   MPI_Comm comm(in),
);
```

Address of data to send to root process.

Fixed `recvcount` elements of `datatype` type.

# Partial Reduction

- All ranks collect data from other processes **with smaller rank number** and apply one specific operation, **excluding** its contribution
- MPI allows multiple reductions as the same time



# Partial Reduction

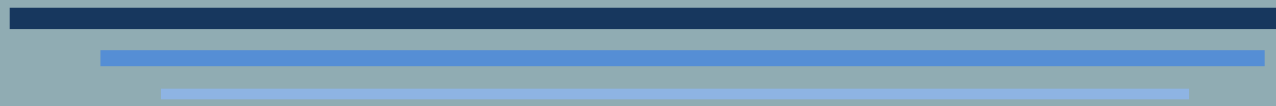
```
int MPI_Exscan (
   void *sendbuf(in),
   void *recvbuf(out),
   int count(in),
   MPI_Datatype datatype(in),
   MPI_Op op(in),
   MPI_Comm comm(in),
);
```

Same signature as MPI_Allreduce

# Why mistakes

- Reduce_scatter_block is the simplified version of reduce_scatter
  - should have been the first one to appear
  - For coherency in naming convention, reduce_scatter_block should be reduce_scatter, and reduce_scatter should be reduce_scatterv
- MPI_Scan is just (MPI_Exscan + local variable)
  - Do not need to create two functions
  - Function MPI_Exscan is enough. Should have been the first one
- MPI_Alltoallw
  - Allows the user to exchange anything
  - Multiple datatypes -> complex internal implementation and algorithms
  - Why is it the only collective to allow multiple datatypes?

# COLLECTIVES



Usage

# Collectives

- Completing a collective call on a process does not mean that the collective is done for all processes
  - It just means that the contribution of the current process is done
- If the current process does not require the result but only send data, it basically is the same as MPI_Send
  - After the collective call, the user can reuse the input buffer
  - It does not even mean that the data has been send
- If all processes provide input data and require an output result, the collective is synchronizing
  - However, Barrier is the only official synchronizing collective in the API

# Collectives

- Every collective operation requires a communicator
  - Define set of processes that will participate
- What happen if two ranks call a different collective operation?
- What happen if two ranks in different communicator call the same collective communication at the same time?

# Rules

- All processes in the group identified by the communicator must call the collective routine
- Inside a communicator, processes should call the same sequence of collective communication
- Between two ranks in different communicators, there are no restrictions

# Examples

```
void f ( int r ) {
         if( r == 0 )
            MPI_Barrier(MPI_COMM_WORLD);
         return;
}
```

May be wrong (depends on value of r)

```
void g ( int r ) {
         if( r == 0 )
            MPI_Barrier(MPI_COMM_WORLD);
         else
            MPI_Barrier(MPI_COMM_WORLD);
         return;
}
```

Correct

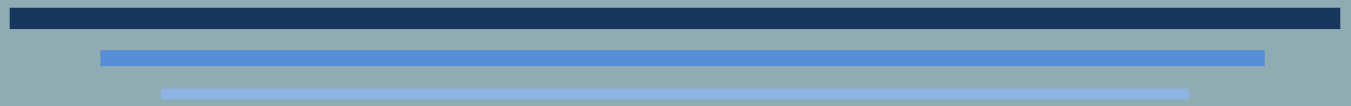
```
void h ( int r ) {
         if( r == 0 ) {
            MPI_Reduce(MPI_COMM_WORLD, …);
            MPI_Barrier(MPI_COMM_WORLD);
         } else {
            MPI_Barrier(MPI_COMM_WORLD);
            MPI_Reduce(MPI_COMM_WORLD, …);
         }
         return; }
```

May be wrong (depends on value of r)

# Hints

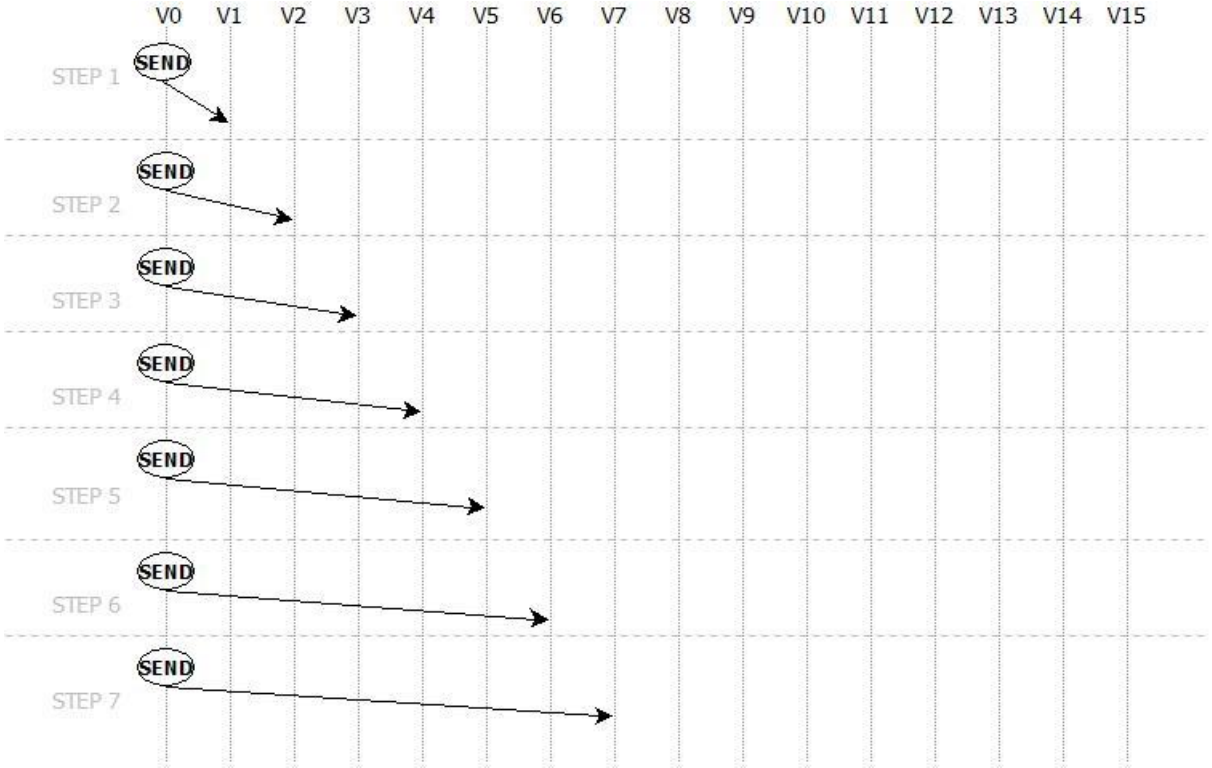
- Be careful of arguments to MPI collective communications
  - Same root
  - Compatible count for elements to send and to receive
- Be careful of control flow inside the program
  - Not related to lexical order
  - Related to dynamic order
- Be careful of communicators

# COLLECTIVE ALGORITHMS



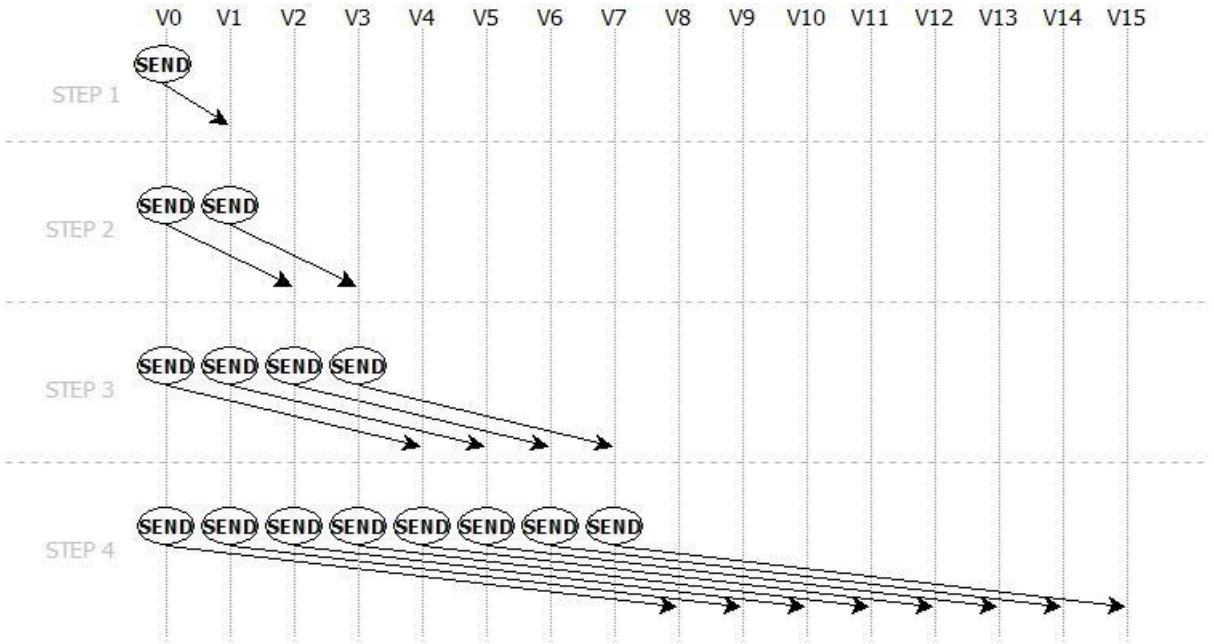
# Broadcast algorithms

- Linear algorithm
- Root send its data to each other process
- Iterative and linear: for p processes, need p-1 step



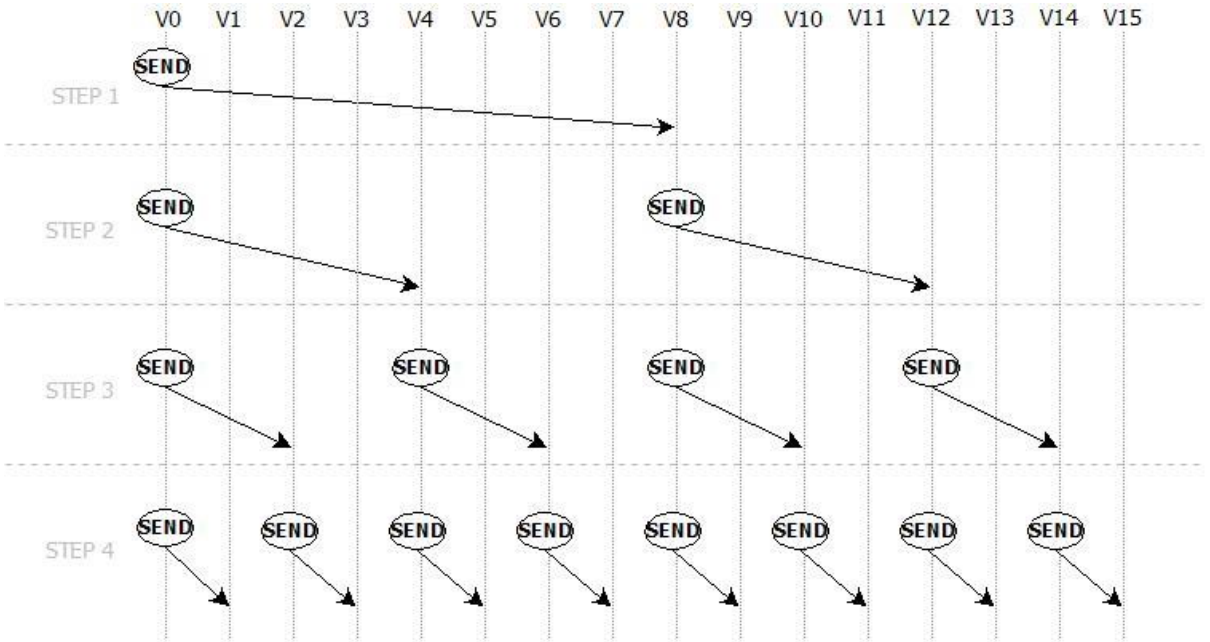
# Broadcast algorithms

- Binary tree: at each step, each process with the data to broadcast send the data to another process
- Need log(p) steps
- For(0; i<log(p); i++) if(rank<=i) send(data, rank+$2^i$)



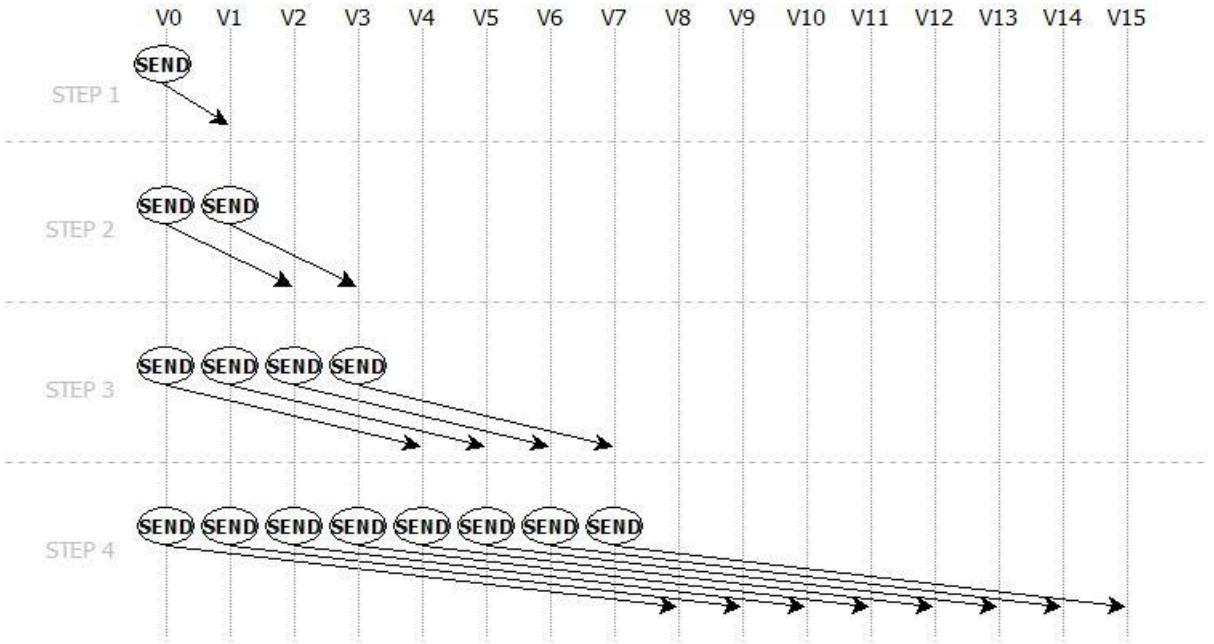
# Broadcast algorithms

- Binary tree
- For(log(p); i>0; i--) if(rank%$2^i$==0) send(data, rank+$2^{i-1}$)
- Are those two algorithms equivalent?



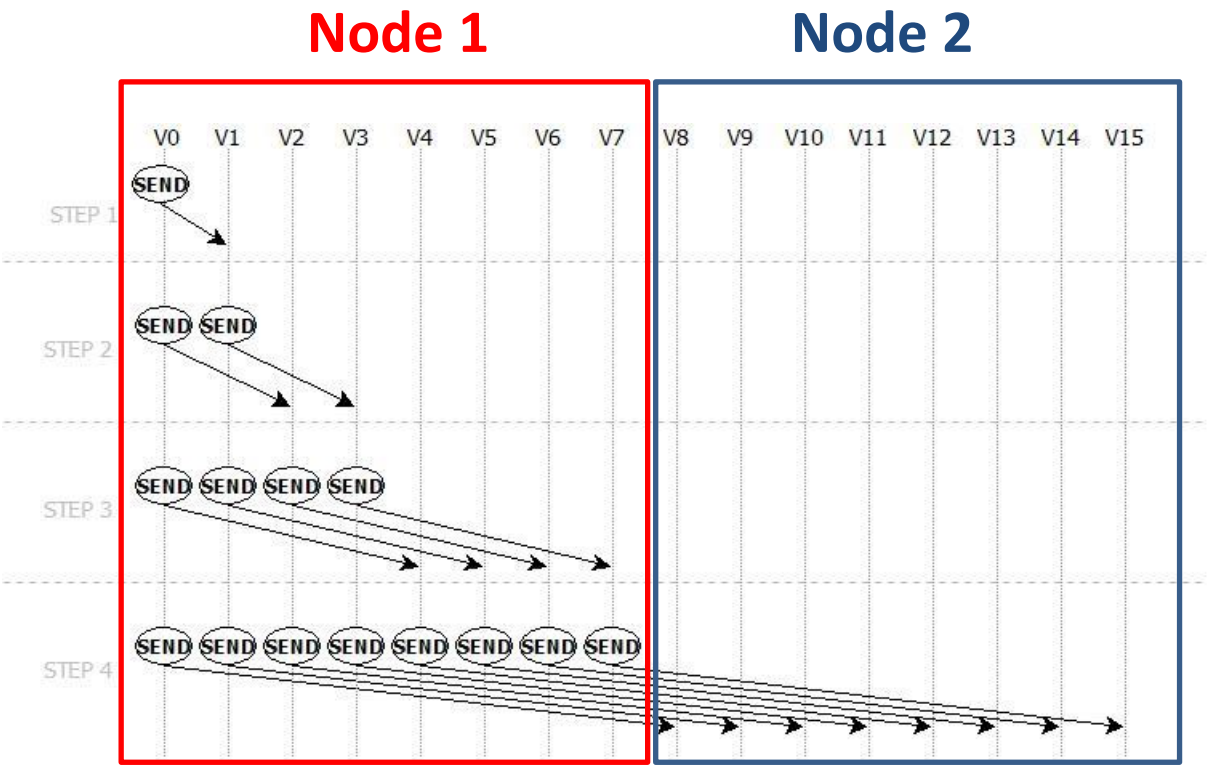
# Broadcast algorithms

- If we consider two nodes



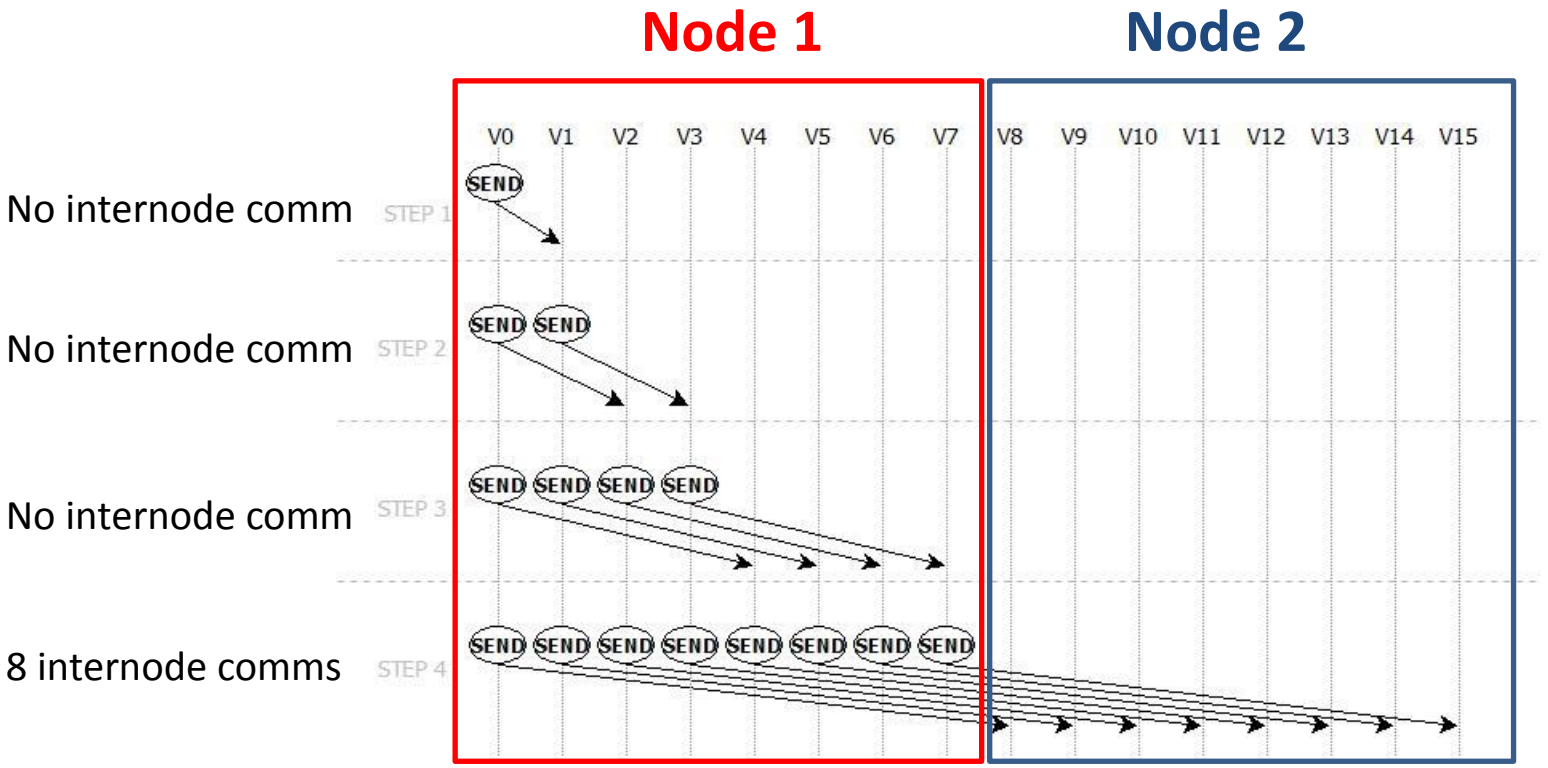
# Broadcast algorithms

- If we consider two nodes



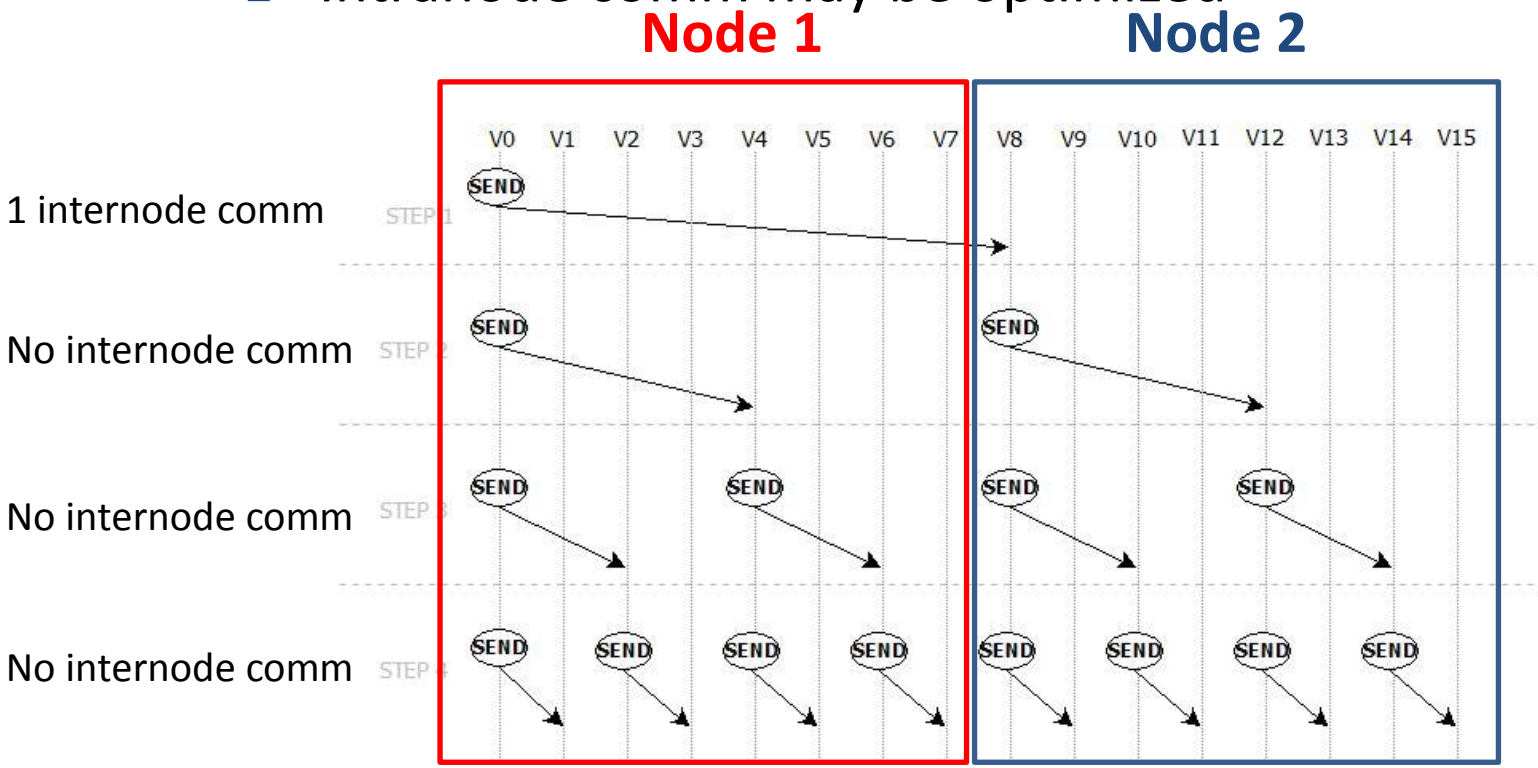
# Broadcast algorithms

- If we consider two nodes
- All internode communications ( network) happens at the same step



# Broadcast algorithms

- Only one internode communication
- No contention on the  newtwork
- Intranode comm may be optimized



# Collective algorithms

- Several algorithms may be implemented for one collective in a runtime
- Default selection of algorithms often depends on the size of the messages, and the number of MPI processes involved
- Algorithm selection may also be available to user through launch options, environment variables or communicator attributes