MPI – Collective Communication Operations

Operations: Broadcast, Reduction, Scatter, Gather, All-to-All

Communicators and Groups

Bibliography

- [Pacheco]: Peter Pacheco, Matthew Malensek, *Introduction to Parallel Programming*, 2nd Edition, Morgan Kaufmann Publisher, March 2020, Chapter 3
- [GGKK] Grama, Gupta, Karypis, Kumar, *Introduction to Parallel Computing*, Chapter 4, Chapter 6
- https://hpc-tutorials.llnl.gov/mpi/
- https://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/
- https://enccs.github.io/intermediate-mpi/collective-communication-pt1/

Message-based communication

- One-to-one: Send-Receive
- Group communication
  - One-to-All Broadcast and All-to-One Reduction
  - All-to-All Broadcast and Reduction
  - All-Reduce
  - Scatter and Gather
  - All-to-All Personalized Communication

Basic Communication Operations: Introduction

- Many interactions in practical parallel programs occur in well-defined *patterns involving groups of processes*
- A program can achieve the semantic of the communication pattern involving a group of processes by using only send-receive, BUT:
- Efficient implementations of these patterns of group communication as new operations can improve performance!
  - Efficient implementations of these group communication operations must leverage underlying architecture
  - Efficient implementations of the group communication patterns in messaging middleware usually use the *recursive doubling or recursive halving* techniques

Collective Communication Operations

- MPI provides an extensive set of functions for performing common collective communication operations.
- Each of these operations is defined over a group corresponding to the communicator.
- All processors in a communicator must call these operations.

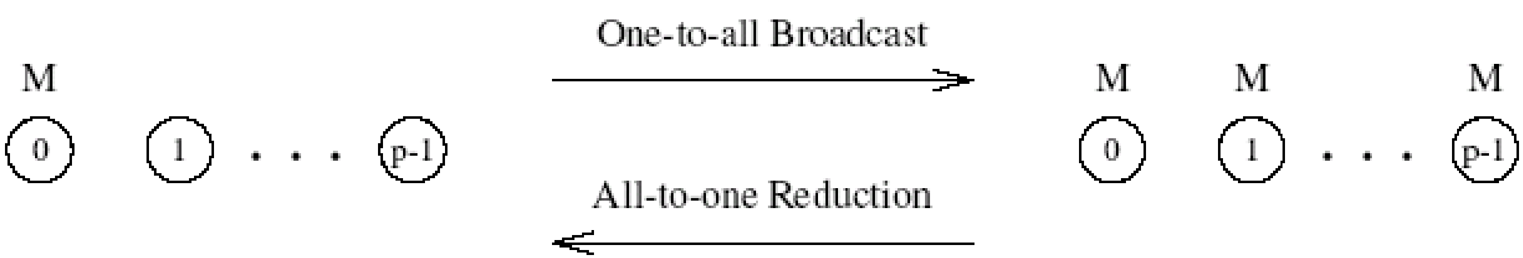
MPI implementations of group communication operations

Table 4.2 MPI names of the various operations discussed in this chapter.

Operation	MPI Name
One-to-all broadcast	MPI_Bcast
All-to-one reduction	MPI_Reduce
All-to-all broadcast	MPI_Allgather
All-to-all reduction	MPI_Reduce_scatter
All-reduce	MPI_Allreduce
Gather	MPI_Gather
Scatter	MPI_Scatter
All-to-all personalized	MPI_Alltoall

One-to-All Broadcast and All-to-One Reduction

- One-to-all Broadcast: One processor has a piece of data (of size *m*) it needs to send to everyone.
- The dual of one-to-all broadcast is *all-to-one reduction*.
- In all-to-one reduction, each processor has *m* units of data. These data items must be combined piece-wise (using some associative operator, such as addition or min), and the result made available at a target processor.

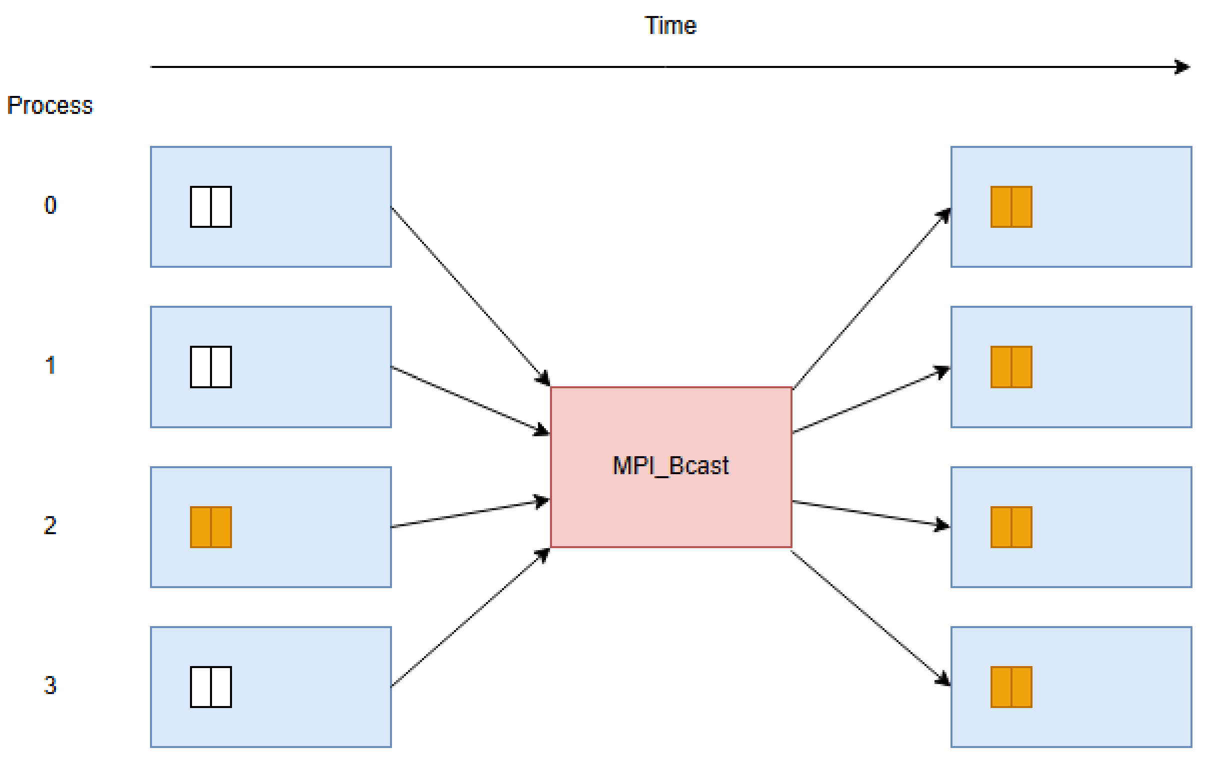


One-to-All Broadcast in MPI

- The one-to-all broadcast operation is:

```
int MPI_Bcast(void *buf, int count,
                              MPI_Datatype datatype,
                              int source, MPI_Comm comm)
```

One-to-All Broadcast

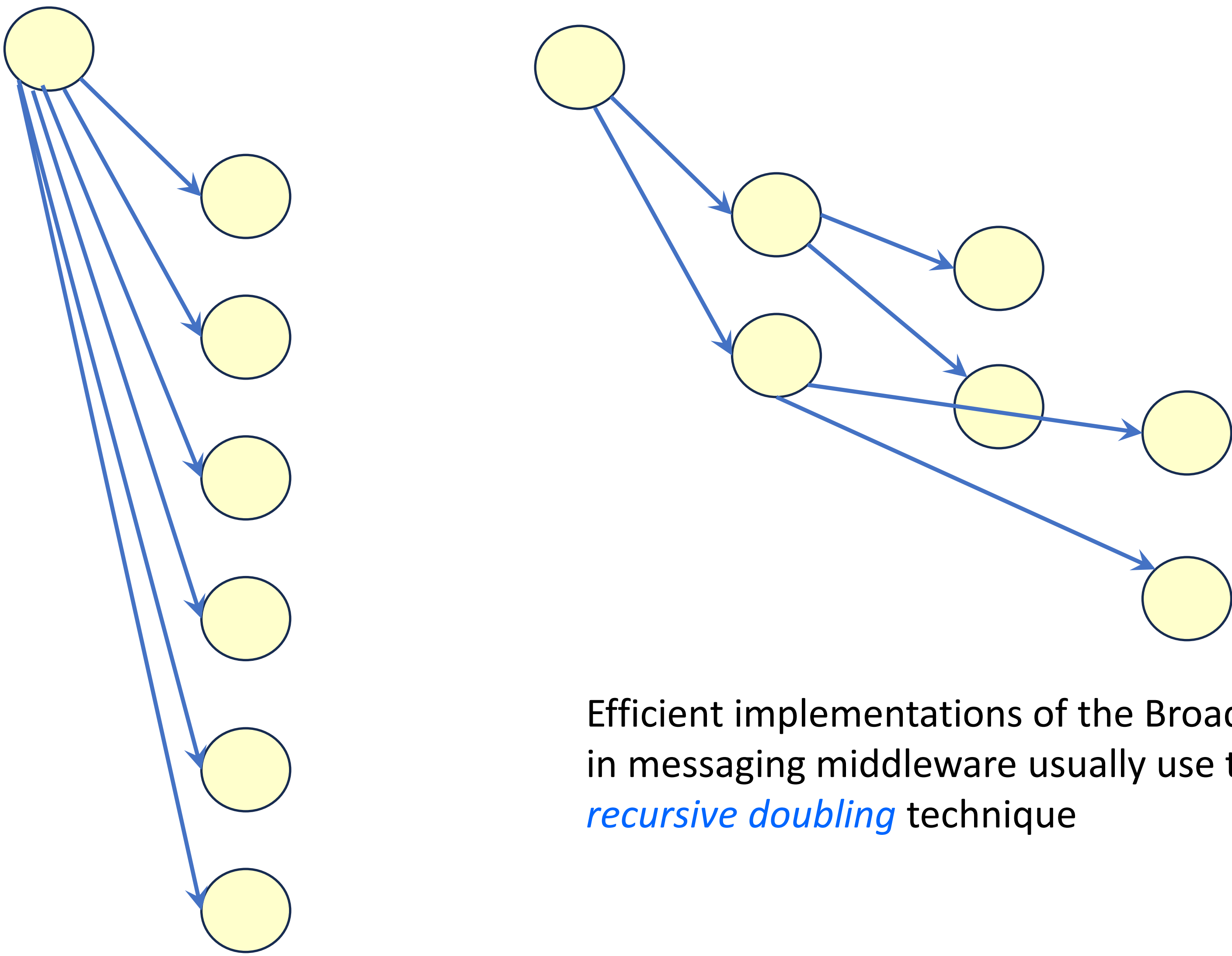


One-to-all broadcast sends a buffer of data from one rank to all other participating ranks.

From https://enccs.github.io/intermediate-mpi/collective-communication-pt1/

```c
int main(int argc, char** argv) {
    int rank, size;
    int a =0; // small buffer - just one integer

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 2) {
        // Rank 2 is the source and sets the value of the buffer a
        a = 7;
        printf("Rank 0 broadcasting a = %d\n", a);
    }

    printf("Before broadcast: Rank %d has a = %d\n", rank, a);

    // Broadcast from rank 2 to all other processes
    MPI_Bcast(&a, 1, MPI_INT, 2, MPI_COMM_WORLD);

    printf("After broadcast: Rank %d has a = %d\n", rank, a);

    MPI_Finalize();
    return 0;
}
```

Efficient Implementation of One-to-All Broadcast



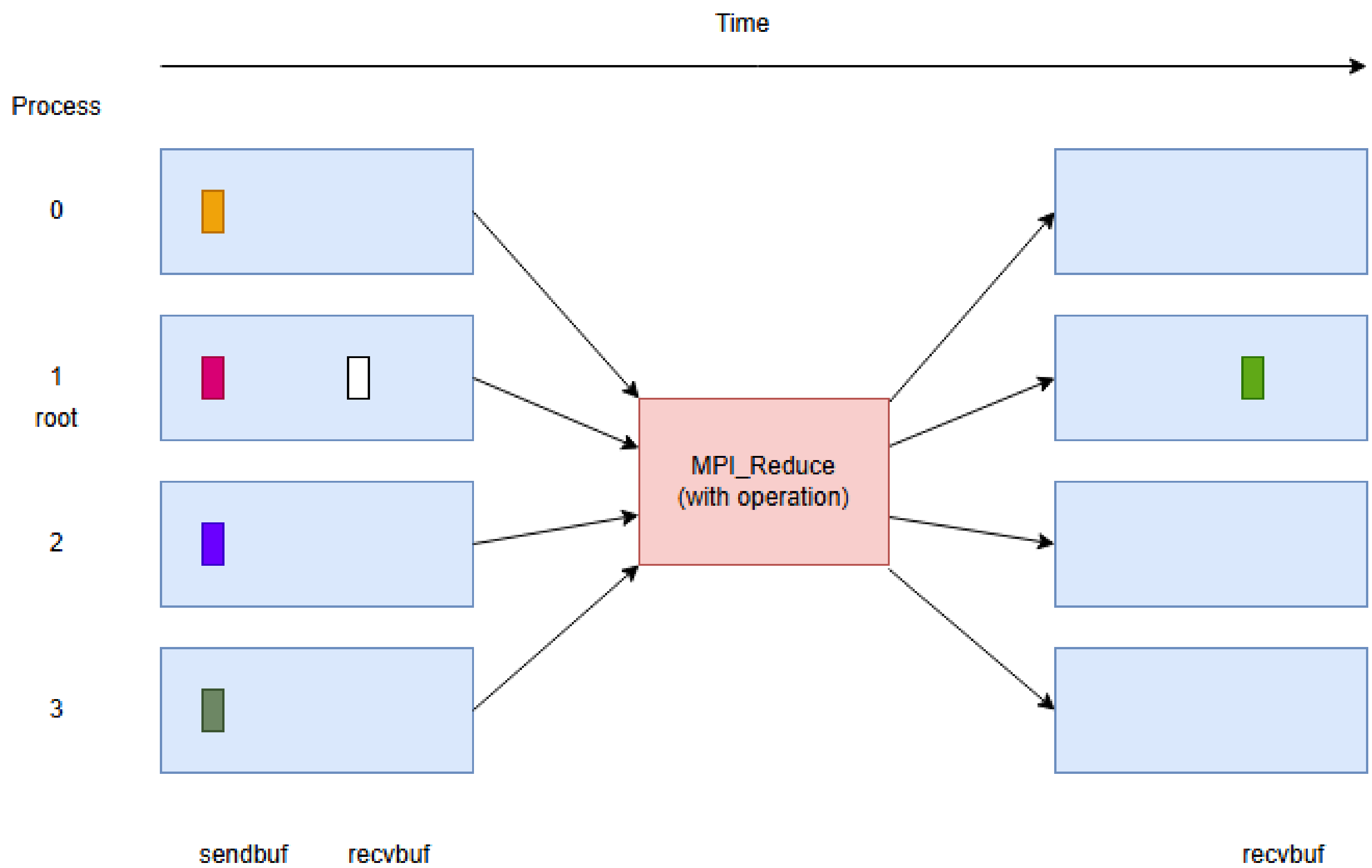
Efficient implementations of the Broadcast in messaging middleware usually use the *recursive doubling* technique

All-to-One Reduction in MPI

- The all-to-one reduction operation is:

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

All-to-one Reduction



All-to-one Reduction combines data from all ranks using an operation and returns values to a single rank

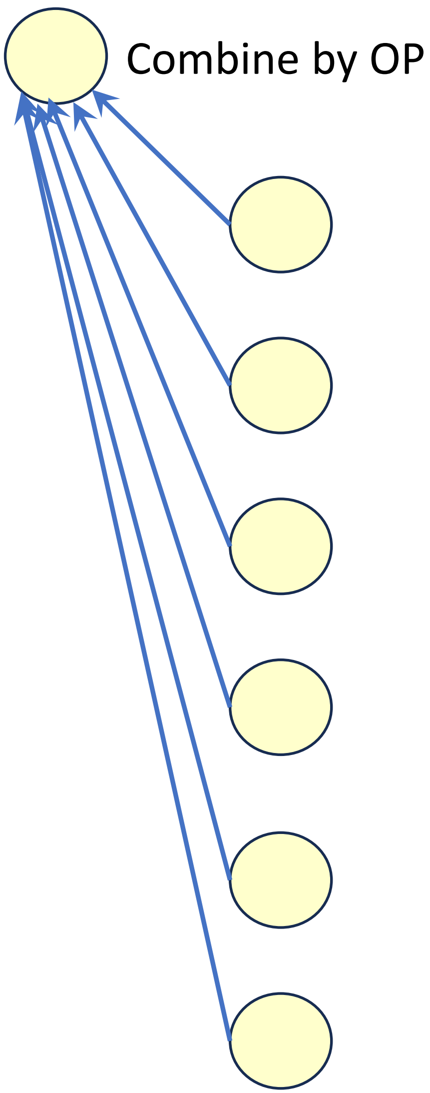
Predefined Reduction Operations

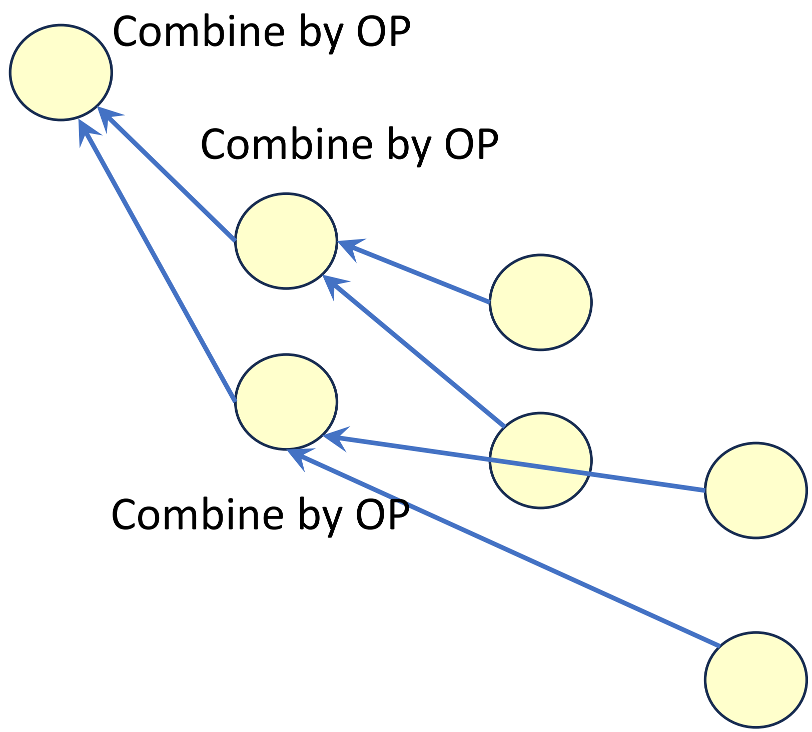
Operation	Meaning	Datatypes
`MPI_MAX`	Maximum	C integers and floating point
`MPI_MIN`	Minimum	C integers and floating point
`MPI_SUM`	Sum	C integers and floating point
`MPI_PROD`	Product	C integers and floating point
`MPI_LAND`	Logical AND	C integers
`MPI_BAND`	Bit-wise AND	C integers and byte
`MPI_LOR`	Logical OR	C integers
`MPI_BOR`	Bit-wise OR	C integers and byte
`MPI_LXOR`	Logical XOR	C integers
`MPI_BXOR`	Bit-wise XOR	C integers and byte
`MPI_MAXLOC`	max-min value-location	Data-pairs
`MPI_MINLOC`	min-min value-location	Data-pairs

```c
int main(int argc, char **argv)
{
    int rank, size;
    int sendbuf;
    int recvbuf;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    sendbuf = rank; // each process puts its rank in sendbuf
    recvbuf = -1;   // each process inits recvbuf with -1
    printf("Before reduce:  Rank %d sendbuf=%d  recvbuf=%d \n",
rank, sendbuf, recvbuf);

    // Reduce to target with rank 1
    MPI_Reduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, 1,
MPI_COMM_WORLD);

    printf("After reduce:  Rank %d sendbuf=%d  recvbuf=%d \n",
rank, sendbuf, recvbuf);

    MPI_Finalize();
```

```
PS C:\Users\MPI> mpiexec -n 4 mpi_reduce_demo
Before reduce:  Rank 0 sendbuf=0  recvbuf=-1
After reduce:  Rank 0 sendbuf=0  recvbuf=-1
Before reduce:  Rank 2 sendbuf=2  recvbuf=-1
After reduce:  Rank 2 sendbuf=2  recvbuf=-1
Before reduce:  Rank 3 sendbuf=3  recvbuf=-1
After reduce:  Rank 3 sendbuf=3  recvbuf=-1
Before reduce:  Rank 1 sendbuf=1  recvbuf=-1
After reduce:  Rank 1 sendbuf=1  recvbuf=6
```

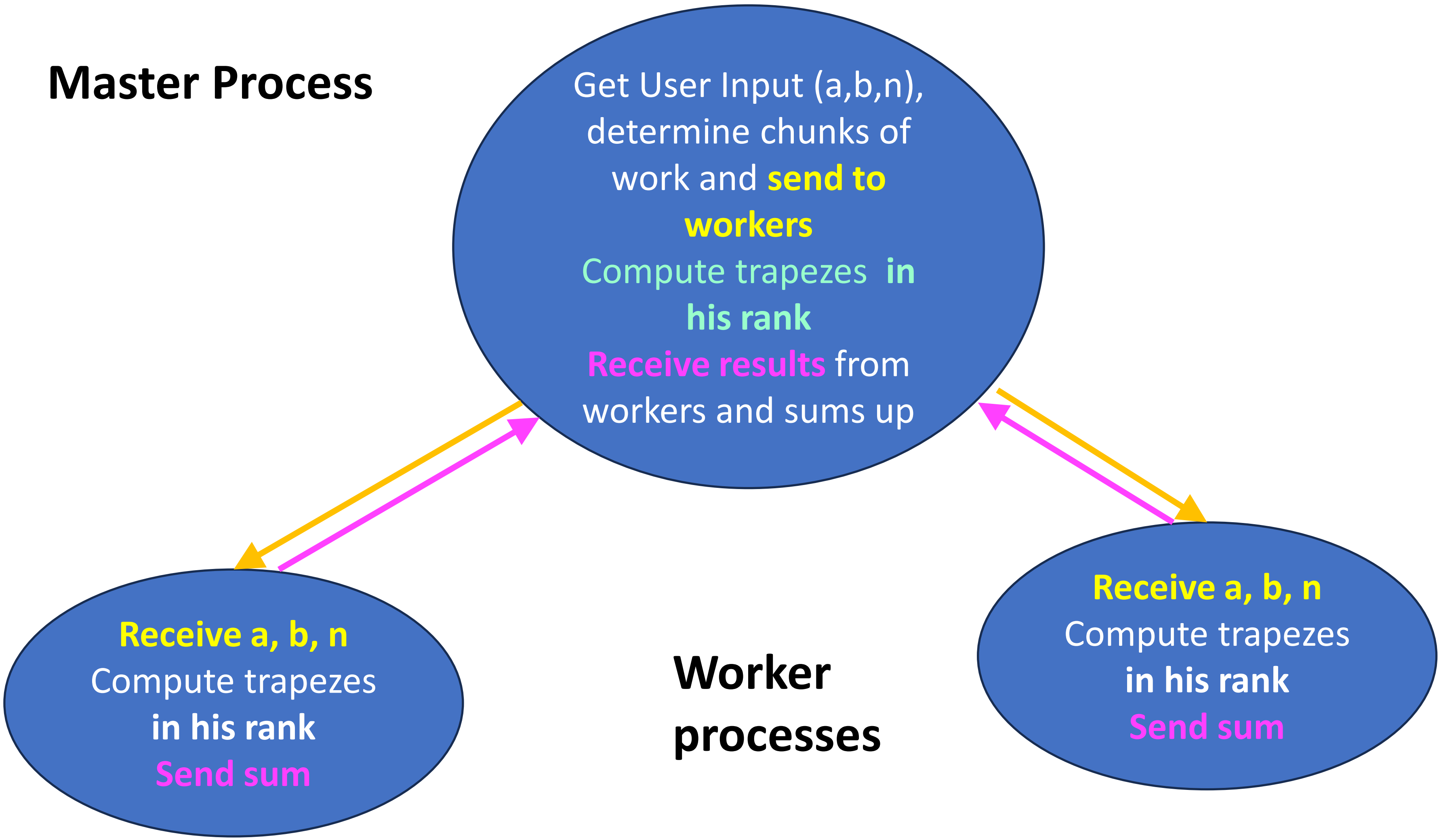
Efficient Implementation of All-to-One Reduction



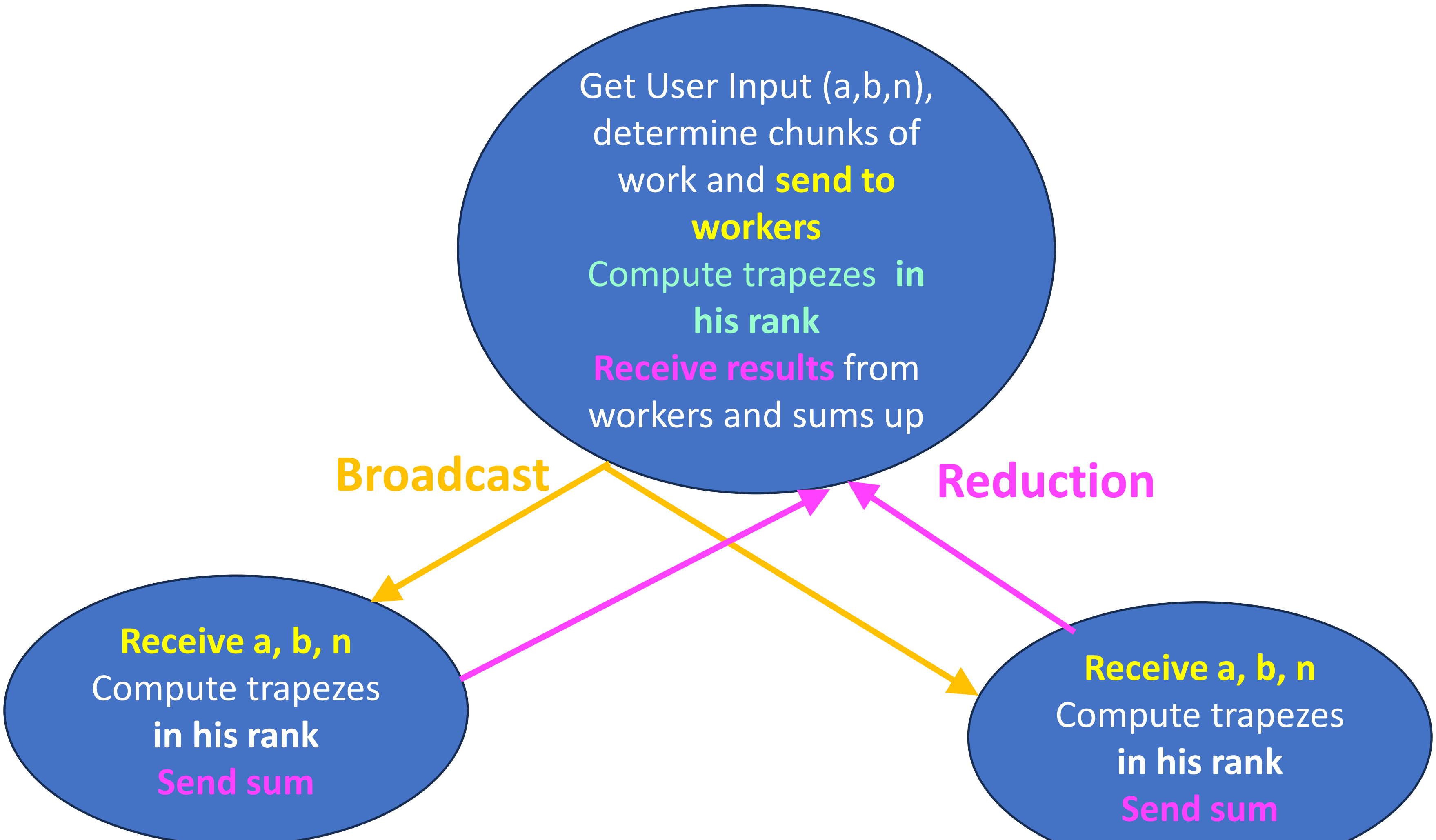


Efficient implementations of Reduction in messaging middleware usually use the *recursive halving* technique

Trapezoidal Rule with MPI Send-Recv



Trapezoidal Rule with MPI Broadcast and Reduction



The All-Reduce Operation

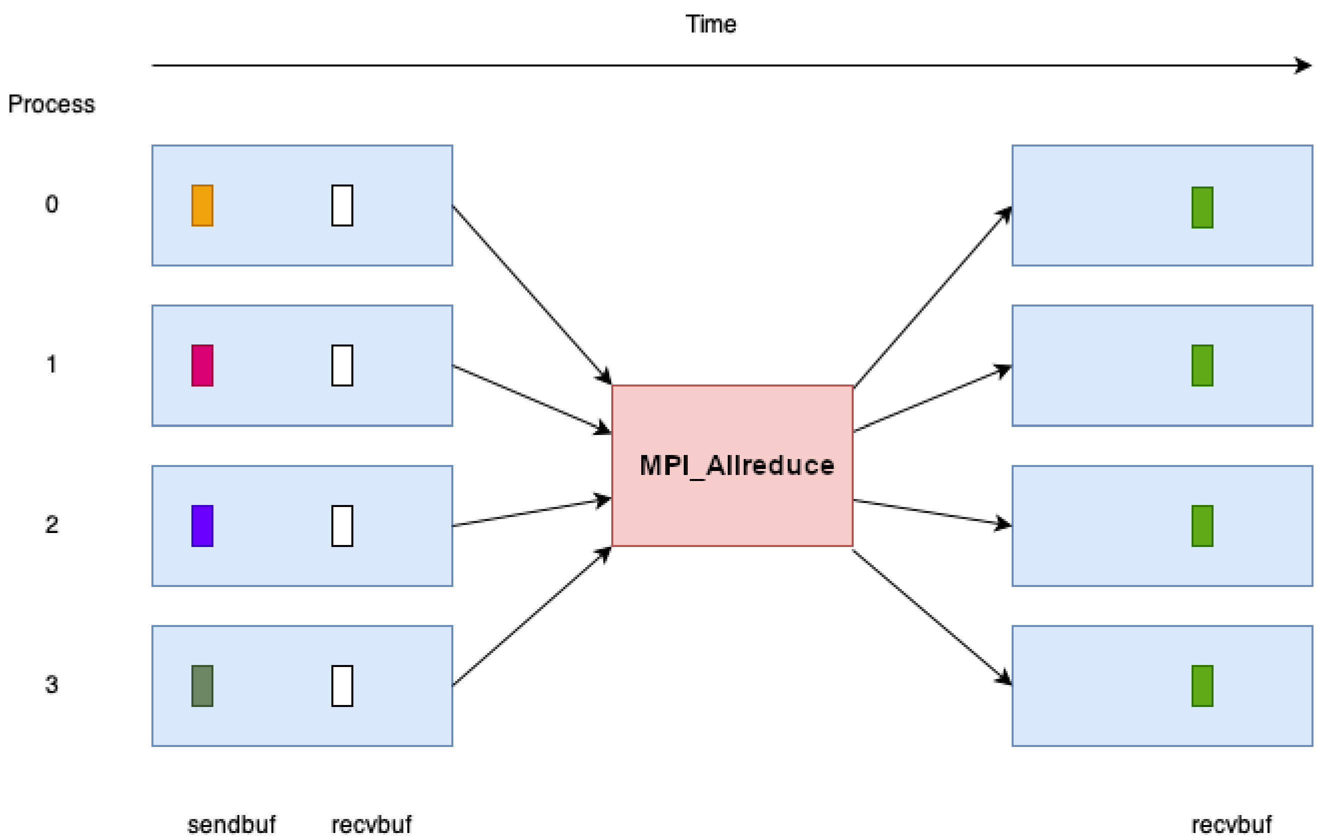
- In all-reduce, each node starts with a buffer of size *m* and the final results of the operation are identical buffers of size *m* on each node that are formed by combining the original *p* buffers using an associative operator.
- Its semantic is identical to all-to-one reduction followed by a one-to-all broadcast, but it can be implemented more efficiently as one operation
- It is different from all-to-all reduction, in which *p* simultaneous all-to-one reductions take place, each with a different destination for the result.

MPI_Allreduce

- If the result of the reduction operation is needed by all processes, MPI provides:

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

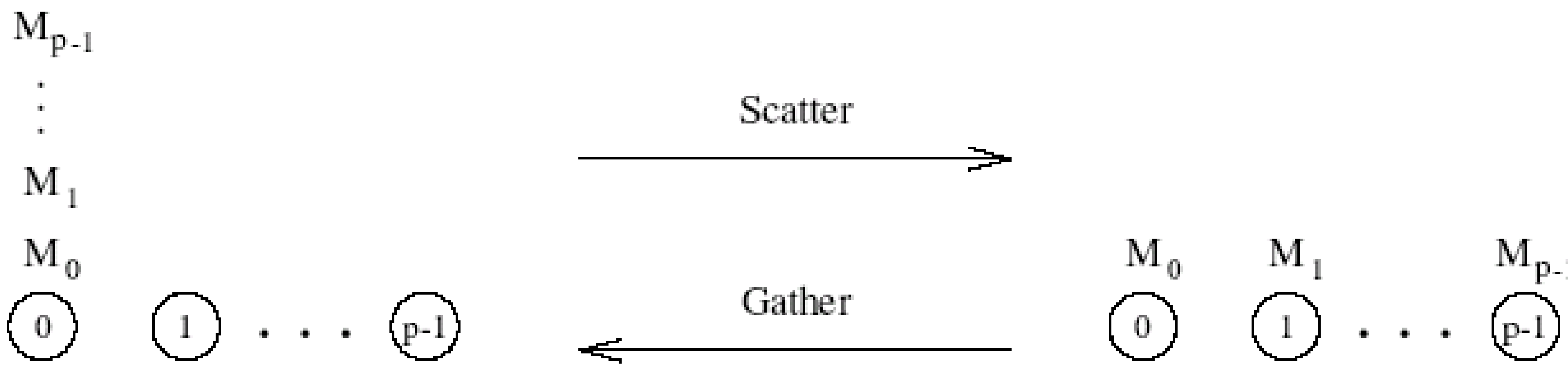
MPI_Allreduce



Scatter and Gather

- In the *scatter* operation, a single node sends a unique message of size *m* to every other node (also called a one-to-all personalized communication).
- In the *gather* operation, a single node collects a unique message from each node.
- While the semantic of scatter operation is fundamentally different from broadcast, their implementation is internally based on the same principle, except for differences in message sizes (messages get smaller in scatter and stay constant in broadcast).
- The gather operation is exactly the inverse of the scatter operation and can be executed as such.

Gather and Scatter Operations



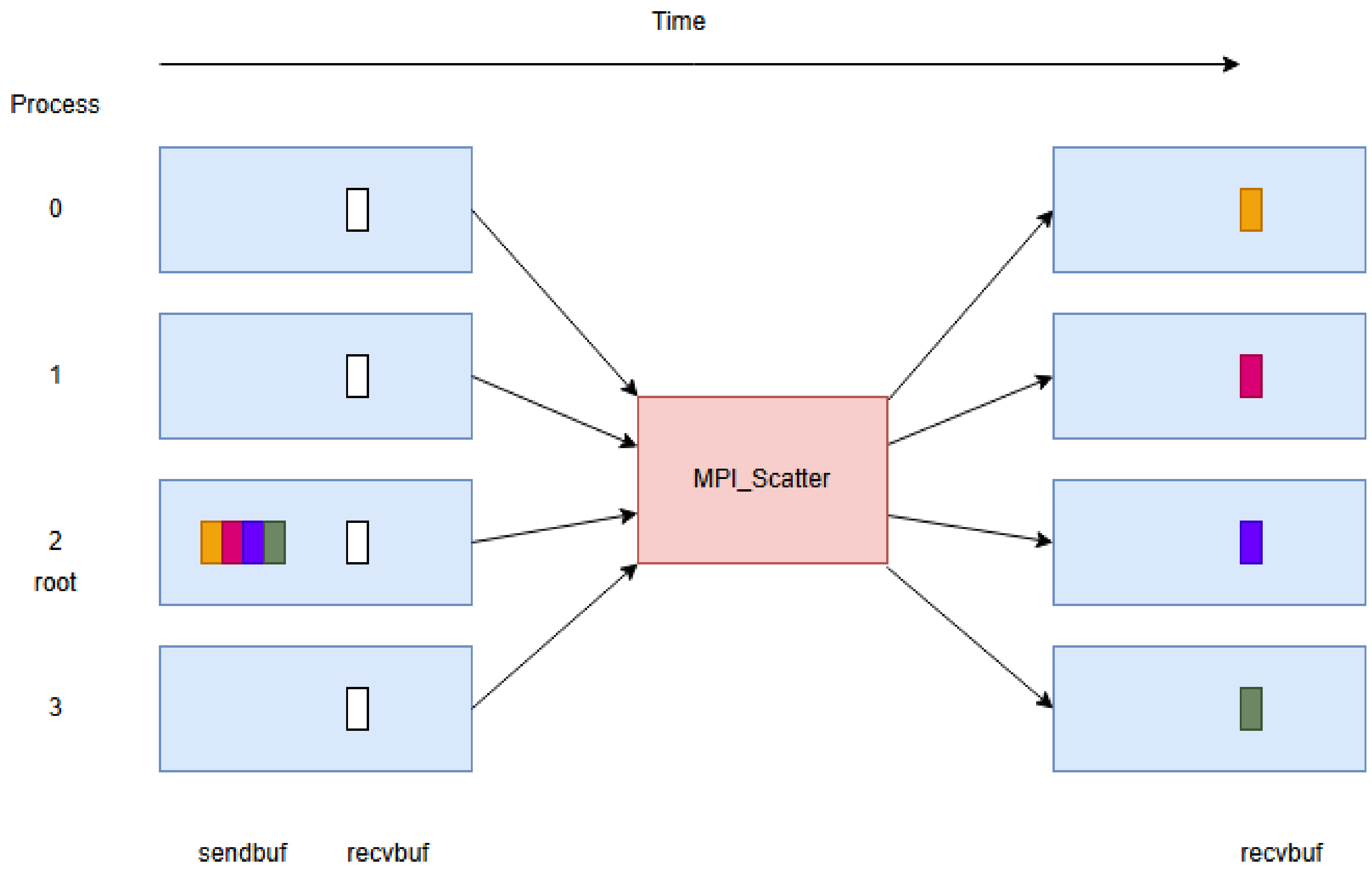
Scatter and gather operations.

Scatter in MPI

- The corresponding scatter operation is:

```
int MPI_Scatter(void *sendbuf, int sendcount,
         MPI_Datatype senddatatype, void *recvbuf,
         int recvcount, MPI_Datatype recvdatatype,
         int source, MPI_Comm comm)
```

Scatter

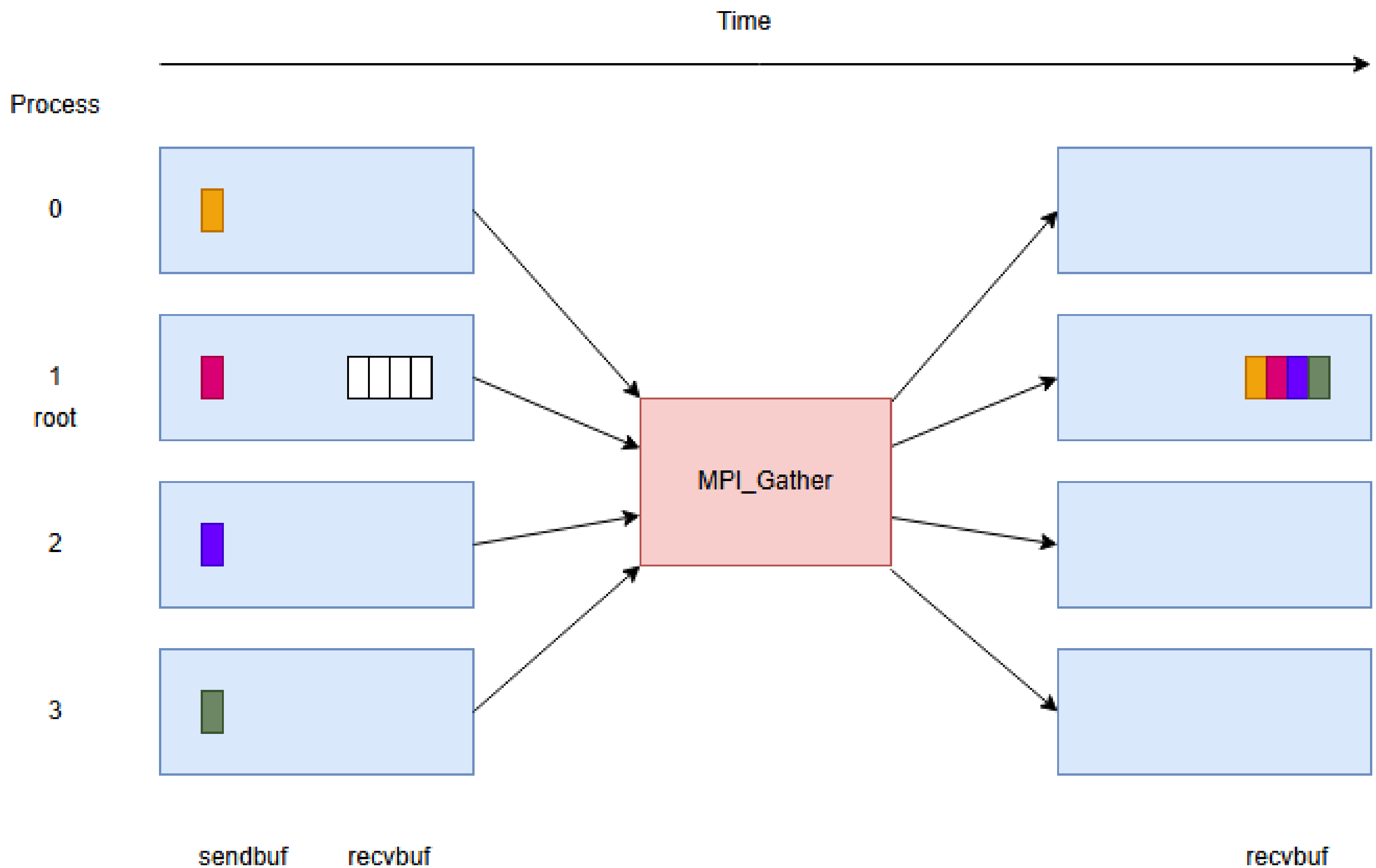

Time
Process
0
1
2
root
3
MPI_Scatter
sendbuf
recvbuf
recvbuf

Gather in MPI

- The gather operation is performed in MPI using:

```
int MPI_Gather(void *sendbuf, int sendcount,
         MPI_Datatype senddatatype, void *recvbuf,
         int recvcount, MPI_Datatype recvdatatype,
         int target, MPI_Comm comm)
```

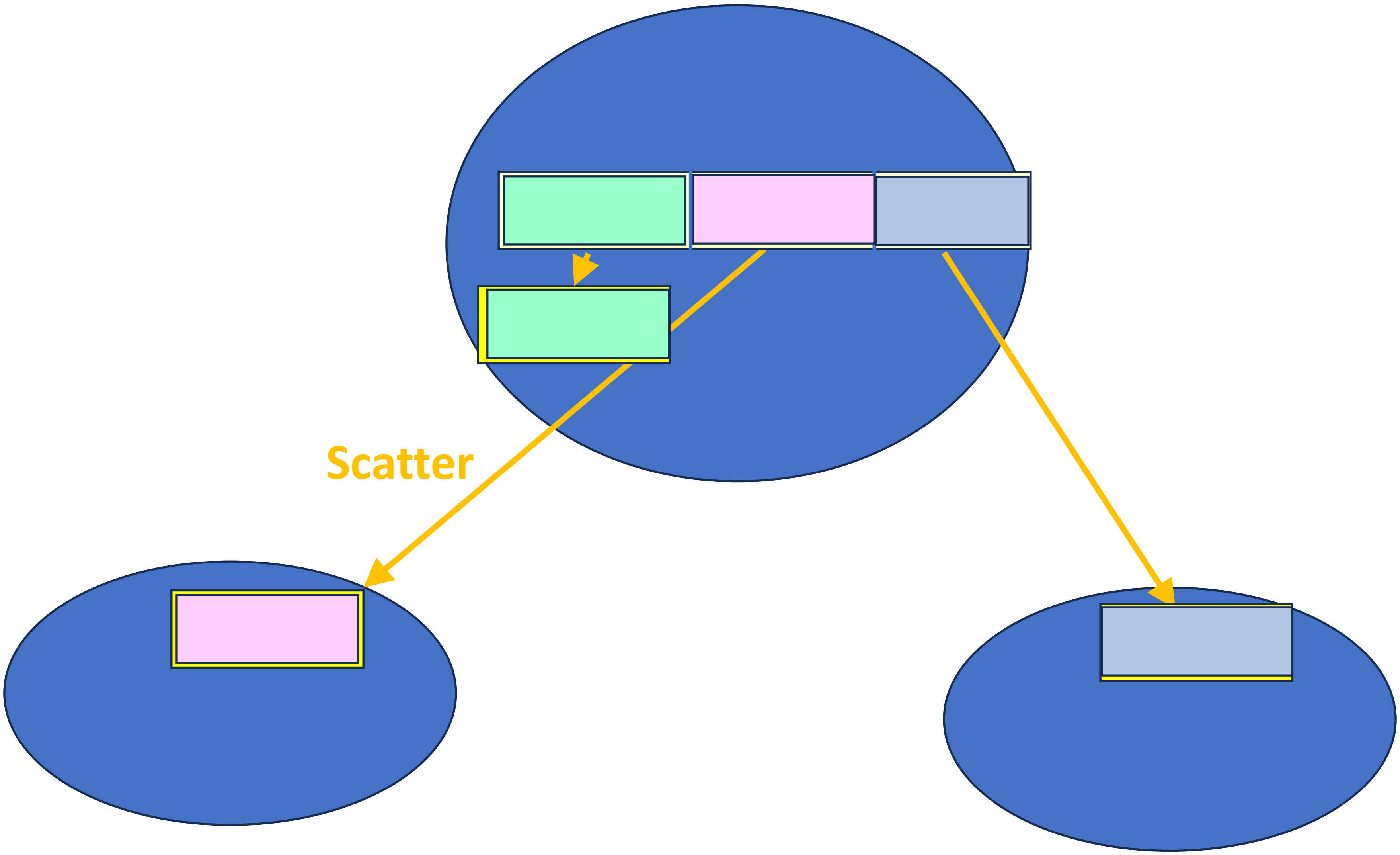
Gather



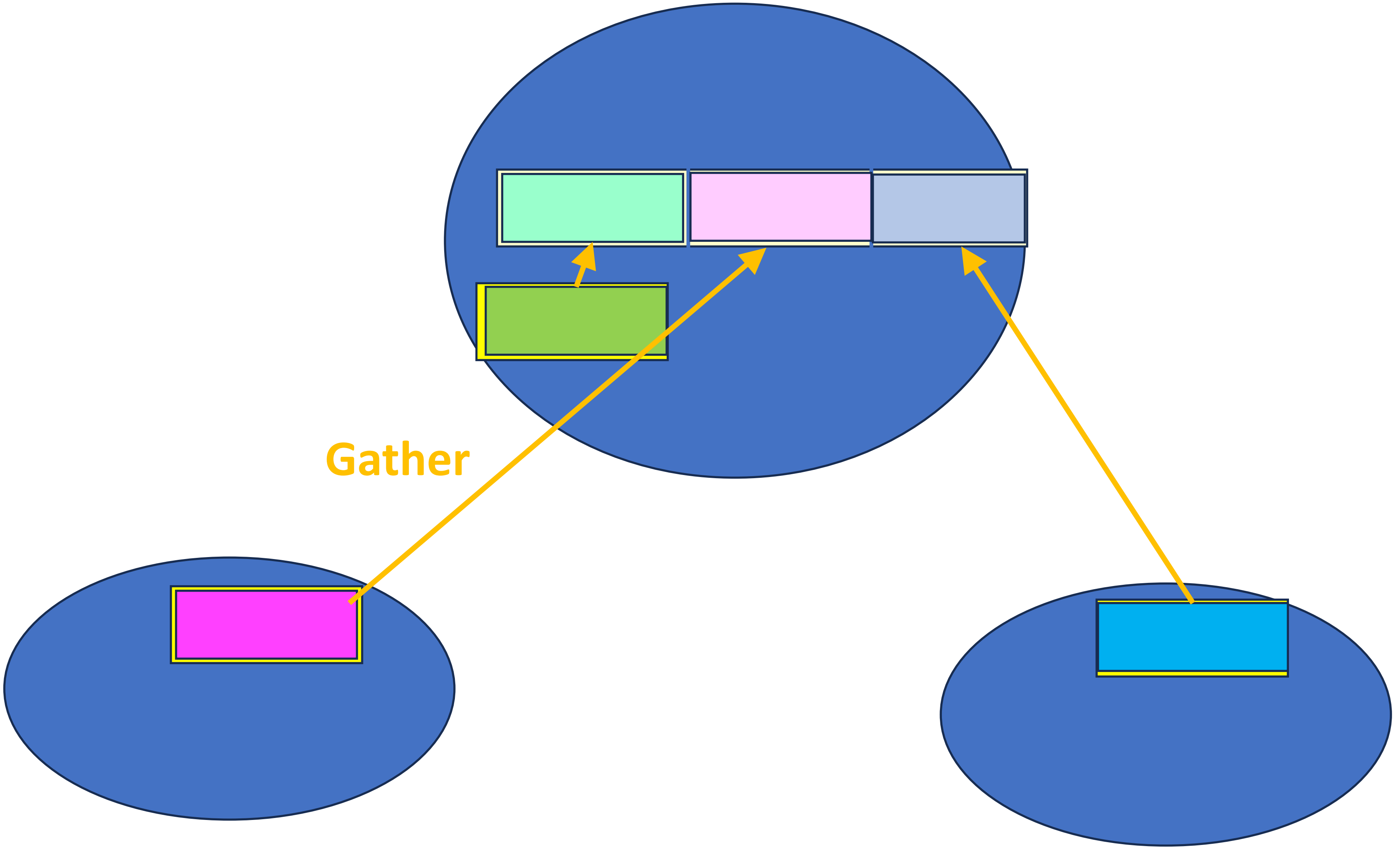
Using Scatter and Gather

- Scatter is very useful when a process (the master) needs to distribute chunks of data to other processes (the workers)
- Gather can be used to assemble the processed chunks back to the master
- Example:
  - Master process holds an array of integers. It splits the array in equal chunks and distributes to workers. Master also gets one chunk. (scatter)
  - Workers double the value of each element and return the chunks back to master. (gather)

Using Scatter and Gather



Using Scatter and Gather



```c
define N 20 // Size of initial array

int main(int argc, char *argv[])
{
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);


    int *array = NULL;    // initial array, allocated only in master
    int *chunk = NULL;    // Chunk size based on number of processes
    int *received = NULL; // final array, allocated only in master

    // Master process initializes the array
    if (rank == 0)
    {
        array = (int *)malloc(N * sizeof(int));
        for (int i = 0; i < N; i++)
        {
            array[i] = i + 1;
        }
    }
```

Continues on next slide ...

Continues from previous slide ...

```c
// every process allocates its chunk
 chunk = (int *)malloc(N / size * sizeof(int));

 // Scatter the array to all processes
 MPI_Scatter(array, N / size, MPI_INT, chunk, N / size,
                              MPI_INT, 0, MPI_COMM_WORLD);

 // Each worker doubles the element values in their chunk
 for (int i = 0; i < N / size; i++)
 {
     chunk[i] *= 2; // Double each element
 }
```

Continues from previous slide ...

```c
    // Master process allocates the received array
    if (rank == 0)  {
        received = (int *)malloc(N * sizeof(int));
    }
    // Gather the results back to the master process
    MPI_Gather(chunk, N / size, MPI_INT, received, N / size,
                                      MPI_INT, 0, MPI_COMM_WORLD);

    // Master process prints the result
    if (rank == 0)  {
        printf("Final array after gather:\n");
        for (int i = 0; i < N; i++)
        {
            printf("%d ", received[i]);
        }
        printf("\n");
        fflush(stdout);
    }
    // Finalize MPI
    MPI_Finalize();
    return 0;
}
```

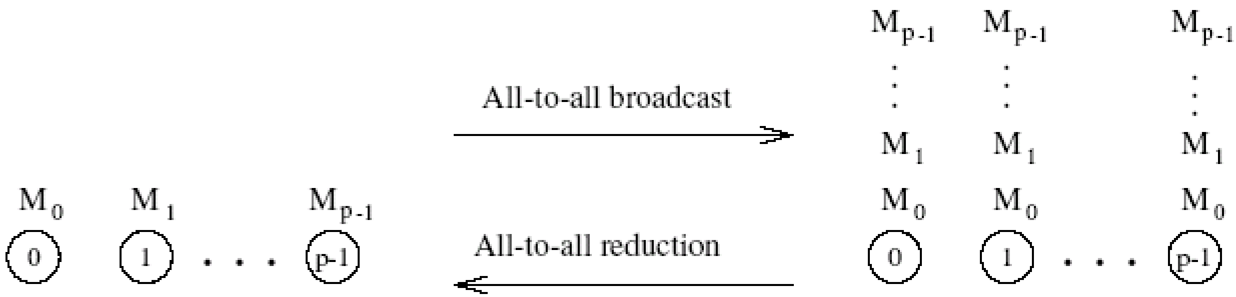
Source Code

- https://staff.cs.upt.ro/~ioana/apd/mpi/mpi_scatter_demo.c

All-to-All Broadcast and All-to-All Reduction

- Generalization of broadcast in which each processor is the source as well as destination.
- A process sends the same *m*-word message to every other process, but different processes may broadcast different messages.

All-to-All Broadcast and Reduction

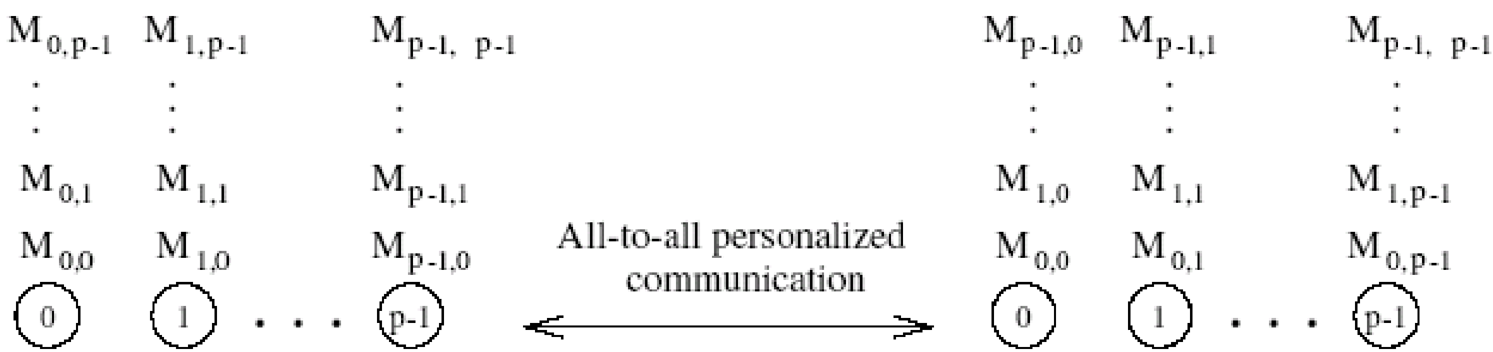


All-to-all broadcast and all-to-all reduction.

All-to-All Personalized Communication

- Each node has a distinct message of size *m* for every other node.
- This is unlike all-to-all broadcast, in which each node sends the same message to all other nodes.
- All-to-all personalized communication is also known as *total exchange*.

All-to-All Personalized Communication



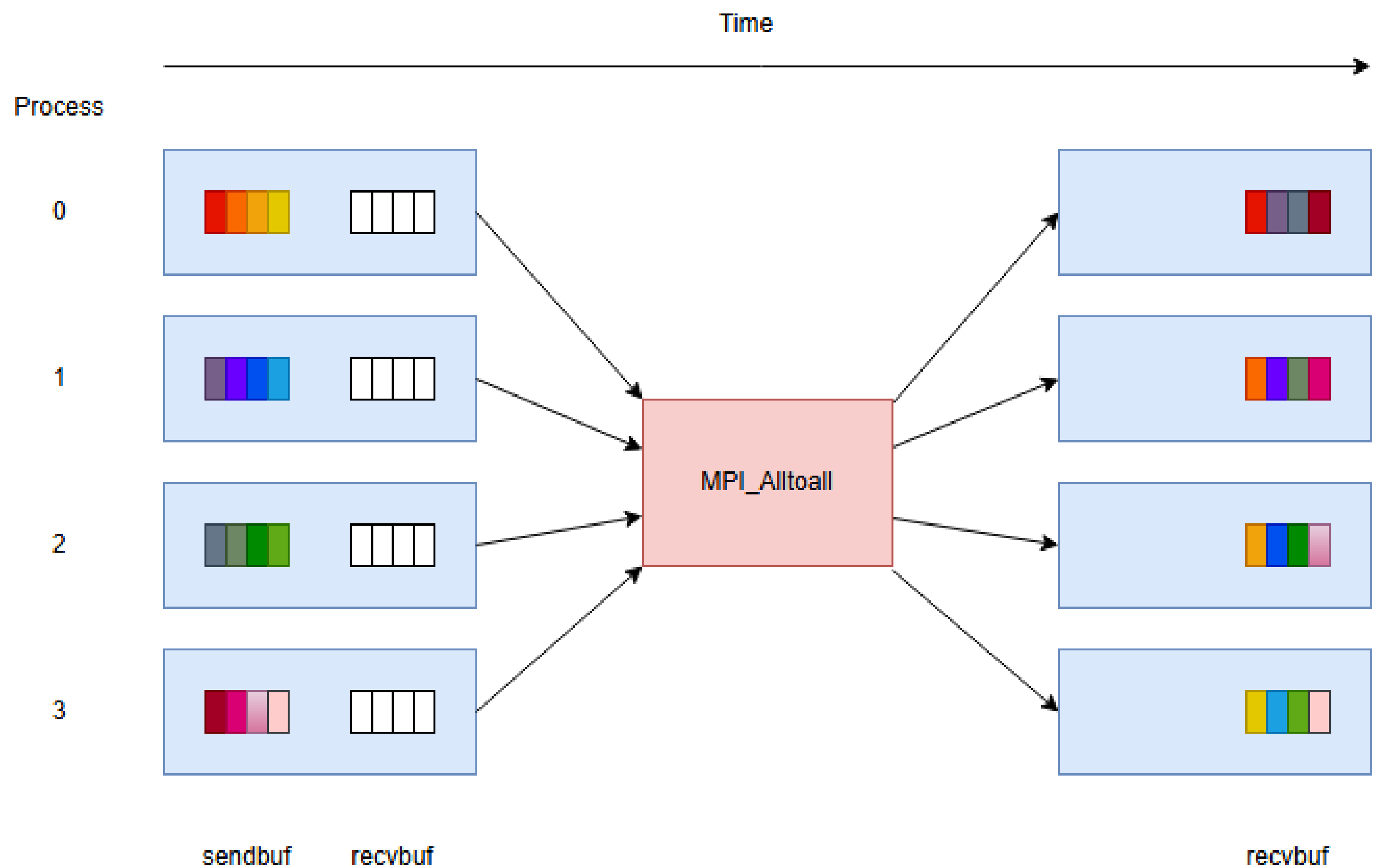
All-to-all personalized communication.

All-to-All Personalized Communication in MPI

- The all-to-all personalized communication operation is performed by:

```
int MPI_Alltoall(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf,
        int recvcount, MPI_Datatype recvdatatype,
        MPI_Comm comm)
```

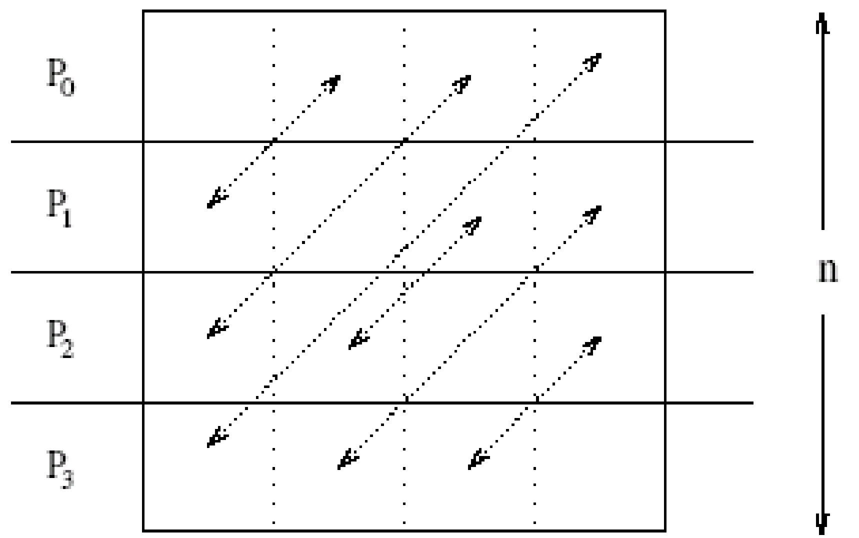
All-to-All Personalized



All-to-All Personalized Communication: Example

- Using the set of collective operations, a number of programs can be greatly simplified.
- Consider the problem of transposing a matrix.
- Each processor contains one full row of the matrix.
- The transpose operation in this case is identical to an all-to-all personalized communication operation.

All-to-All Personalized Communication: Example



All-to-all personalized communication in transposing a *4 x 4* matrix using four processes.

Communicators

- Communicators provides a separate communication space. Default:MPI_COMM_WORD
- All communications take place in a Communicator
- All collective operations happen inside a Communicator
- Sometimes you want only a subset of all processes to participate
- It is possible to treat *a subset of processes* as a communication universe
- Can create sub-groups of processes, or sub-communicators
- A process can belong to several communicators
- A process has a rank inside each communicator

Splitting a Communicator

```
int MPI_Comm_split( MPI_Comm comm, int
      color, int key, MPI_Comm *newcomm );
```

- Creates new communicators based on colors and keys.
- This function partitions the group associated with comm into disjoint subgroups, one for each value of color (a nonnegative integer)
- Each subgroup contains all processes of the same color. Within each subgroup, the processes are ranked in the order defined by the value of the argument key, with ties broken according to their rank in the old group.
- This is a collective call. Each process can provide its own color and key.

Using Communicator Split

- Example: we need to perform a broadcast only among the group of odd ranked processes and another broadcast among the group of even ranked processes

```c
int main(int argc, char *argv[]) {
    int rank, size, new_rank, new_size;
    MPI_Comm new_comm;
    int message = 0;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Split the communicator
    // Processes with even ranks go to one group (color = 0),
    // processes with odd ranks to another (color = 1)
    int color = rank % 2;
    MPI_Comm_split(MPI_COMM_WORLD, color, rank, &new_comm);

    // Get the new rank and size in the new communicator
    MPI_Comm_rank(new_comm, &new_rank);
    MPI_Comm_size(new_comm, &new_size);
```

Continues on next slide …

Continues from previous slide ...

```c
if (color == 0) {
    if (new_rank == 0) {
        message = 7;  // Process 0 in the even group broadcasts
    }
    // Broadcast in the even group
    MPI_Bcast(&message, 1, MPI_INT, 0, new_comm);

    printf("Even communicator, Rank %d (GlobalRank %d) received
                message: %d\n",  new_rank, rank, message);
    fflush(stdout);
} else {
    // Odd rank communicator (color = 1)
    if (new_rank == 0) {
        message = 99;  // Process 0 in the odd group broadcasts
    }
    // Broadcast in the odd group
    MPI_Bcast(&message, 1, MPI_INT, 0, new_comm);

    printf("Odd communicator, Rank %d (GlobalRank %d) received
                message: %d\n",  new_rank, rank, message);
    fflush(stdout);
}
```

Source Code

- https://staff.cs.upt.ro/~ioana/apd/mpi/mpi_comm_split.c

More Collective Operations

- https://learn.microsoft.com/en-us/message-passing-interface/mpi-collective-functions

Conclusions

- Many interactions in parallel programs occur in well-defined *patterns involving groups of processes*
- A program can achieve the semantic of these communication pattern involving a group of processes by using only send-receive, BUT:
- Messaging middleware offers implementations of these patterns of group communication as new collective communication operations that have better performance!
- Group communication
    - One-to-All Broadcast and All-to-One Reduction
    - All-Reduce
    - Scatter and Gather
    - All-to-All Broadcast and Reduction
    - All-to-All Personalized Communication