The Second Biomedical Engineering Workshop

A brief introduction to MPI

Dr. Hisham Mohamed

Sensirion AG, Switzerland

SENSIRION
THE SENSOR COMPANY

Overview

- MPI basic concepts
- MPI functions
- MPI communication
  - Point to point communication
  - Collective communication

Message Passing Interface (MPI)

- MPI is a standard
  - C and C++
  - MPI-1 and MPI-2
- Proposed for distributed memory architecture
  - Different address space
  - Network Communication needed (synchronization, data exchange)
- Can be used in shared memory architecture

Message Passing Interface (MPI)

- Programmer is responsible for everything
  - Identify parallelism
  - Data exchange
  - Communication type
- Code reengineering is always needed

MPI execution model

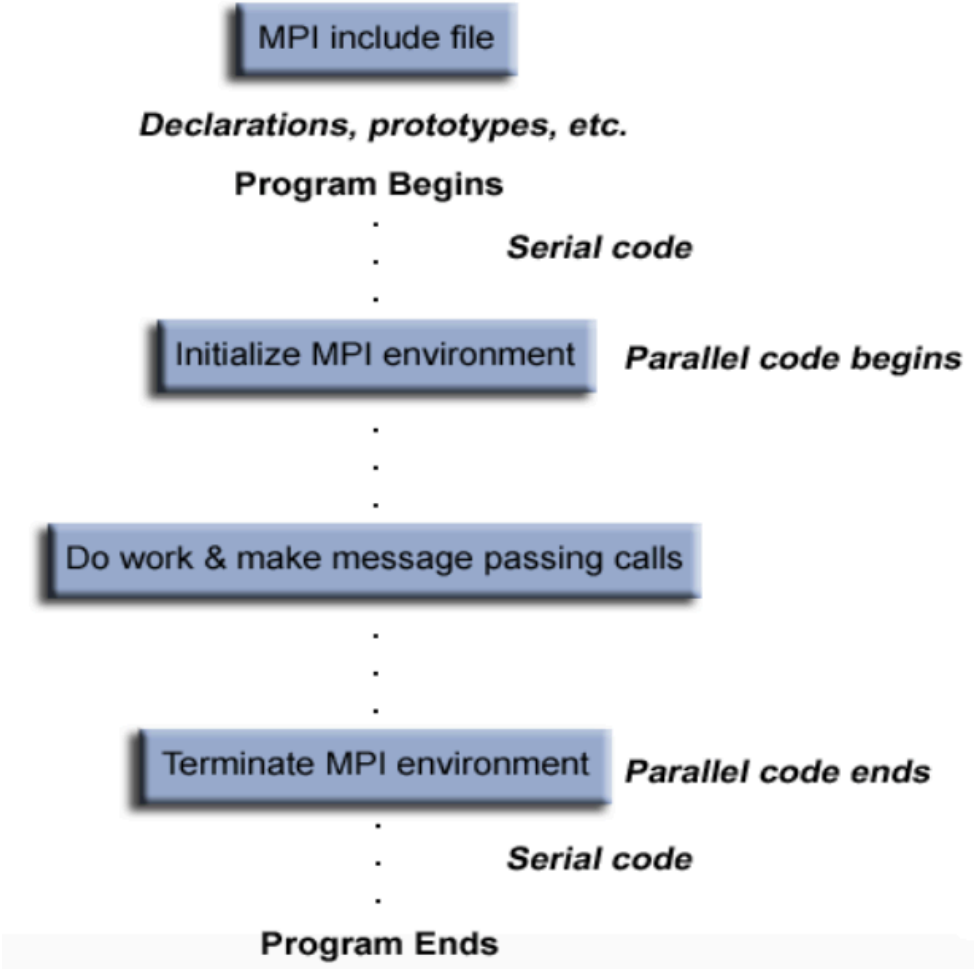
- **S**ingle **P**rogram **M**ultiple **D**ata (SPMD)
- Parallel processes
- Processes divide the work between them
- Processes work independently unless a communication is needed

MPI execution model

- **S**ingle **P**rogram **M**ultiple **D**ata (SPMD)
- Parallel processes
- Processes divide the work between them
- Processes work independently unless a communication is needed

Programmer responsibility!

MPI Programs general structure



https://computing.llnl.gov/tutorials/mpi/

MPI functions main parameters

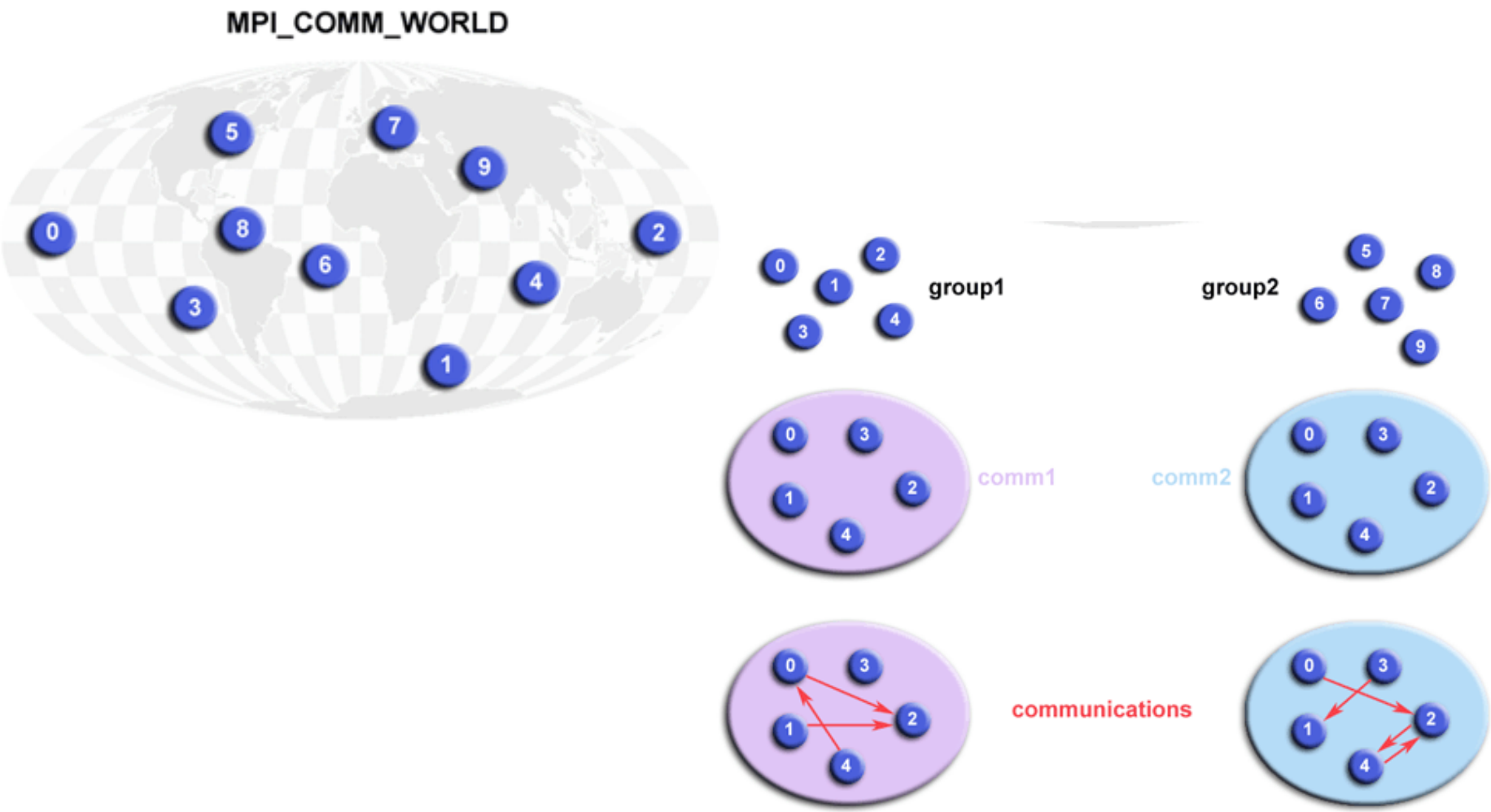
- **Buffer:**
  - Sent: data to be sent
  - Received: where data will be saved
- **Data Count:** Number of elements to be sent
- **Data Type:**
  - Elementary (e.g. MPI_INT, MPI_DOUBLE, MPI_CHAR, ….)
  - Derived data type (e.g. Your own struct)
- **Destination (Rank):** Process ID

MPI functions main parameters

- **Source:** Processor ID who sent the message
- **Tag:** Unique non-negative integer message identifier
- **Status (receive functions):** contains source and tag for a message
- **Request:** Allows the programmer to check if a message has been received, using other MPI functions

MPI functions main parameters

- **MPI Communicators and Groups**



MPI Hello World!

```
#include <stdio.h>
#include <mpi.h>
int main (argc, argv){

    int rank, size;
    MPI_Init (&argc, &argv);   // starts MPI
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    printf( "Hello world from process %d of %d\n", rank, size );

    MPI_Finalize(); // ends MPI
    return 0;

}
```

Determine the id of each process within the communicator

Determine the number of processes within a communicator

MPI Hello World!

```
#include <stdio.h>
#in
int
```

- **mpicc -o hello-world.exe hello-world.cc**
- **mpirun –n 8 hello-world.exe**

```
Hello MPI World from process 5!
Hello MPI World from process 1!
Hello MPI World from process 0!
Hello MPI World from process 6!
Hello MPI World from process 7!
Hello MPI World from process 2!
Hello MPI World from process 3!
Hello MPI World from process 4!
```

cesses

```
}
```

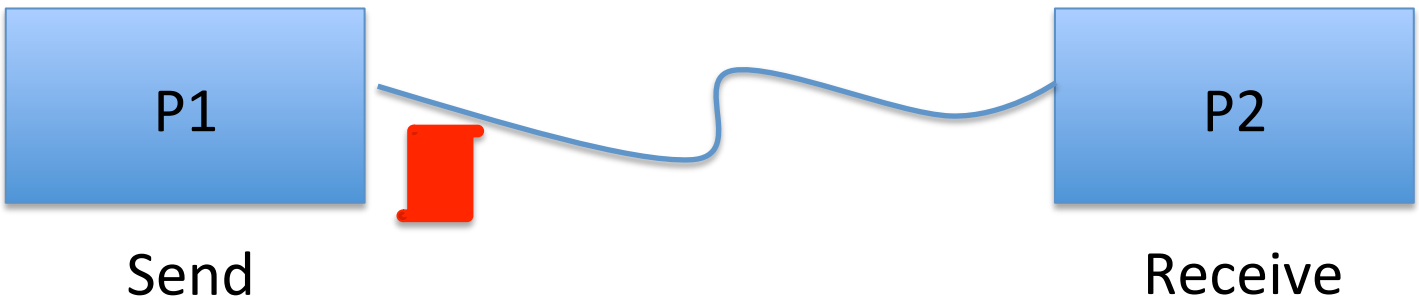
MPI Communication

- Point-to-Point
  - Process to Process within the same communicator
    - Blocking and Non-Blocking

- Collective
  - Multiple processes within the same communicator

MPI Point-to-Point communication

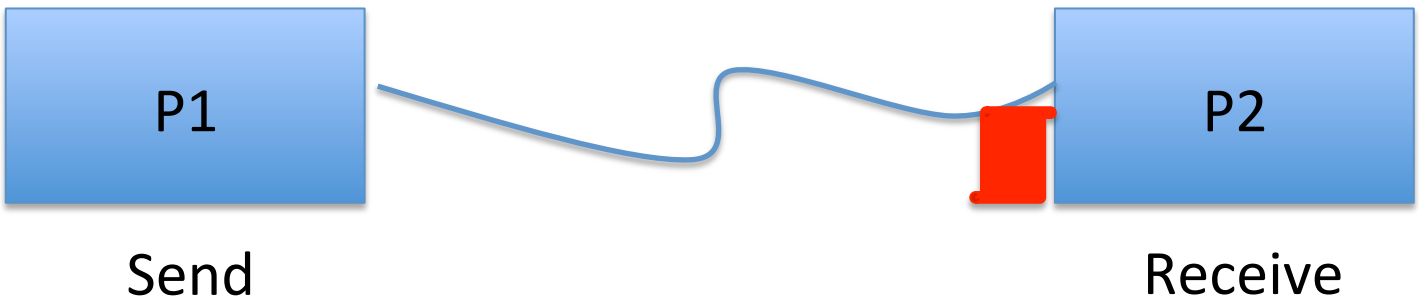
- Message passing between two processes



- For each receive function there should be a send function

MPI Point-to-Point communication

- Message passing between two processes



- For each receive function there should be a send function

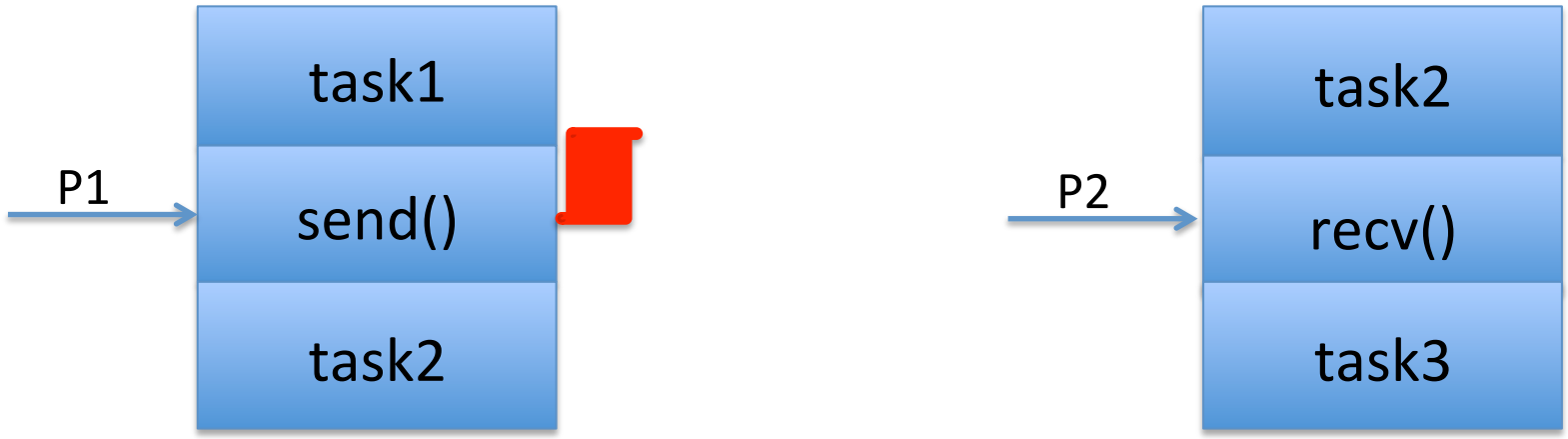
Point-to-Point: Blocking vs. Non-Blocking

- Blocking

- Non-Blocking

Point-to-Point: Blocking vs. Non-Blocking

- Blocking



- Non-Blocking

Point-to-Point: Blocking vs. Non-Blocking

- Blocking

- Non-Blocking

Point-to-Point: Blocking vs. Non-Blocking

- Blocking

- Non-Blocking

Point-to-Point: Blocking vs. Non-Blocking

- Blocking

- Non-Blocking

Point-to-Point: Blocking vs. Non-Blocking

- Blocking

- Non-Blocking

Point-to-Point: Blocking modes

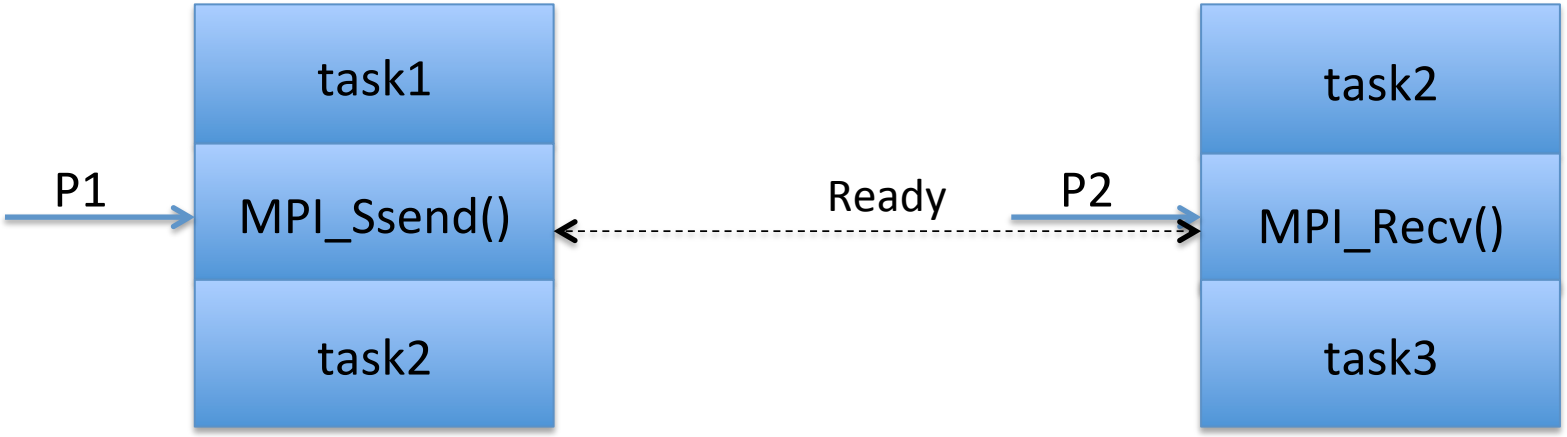
- Synchronous(MPI_Ssend())
  - No buffer
  - Processes wait for each other

Point-to-Point: Blocking modes

- Synchronous(MPI_Ssend())
  - No buffer
  - Processes wait for each other

Point-to-Point: Blocking modes

- Synchronous(MPI_Ssend())
  - No buffer
  - Processes wait for each other



Point-to-Point: Blocking modes

- Synchronous(MPI_Ssend())
  - No buffer
  - Processes wait for each other

Point-to-Point: Blocking modes

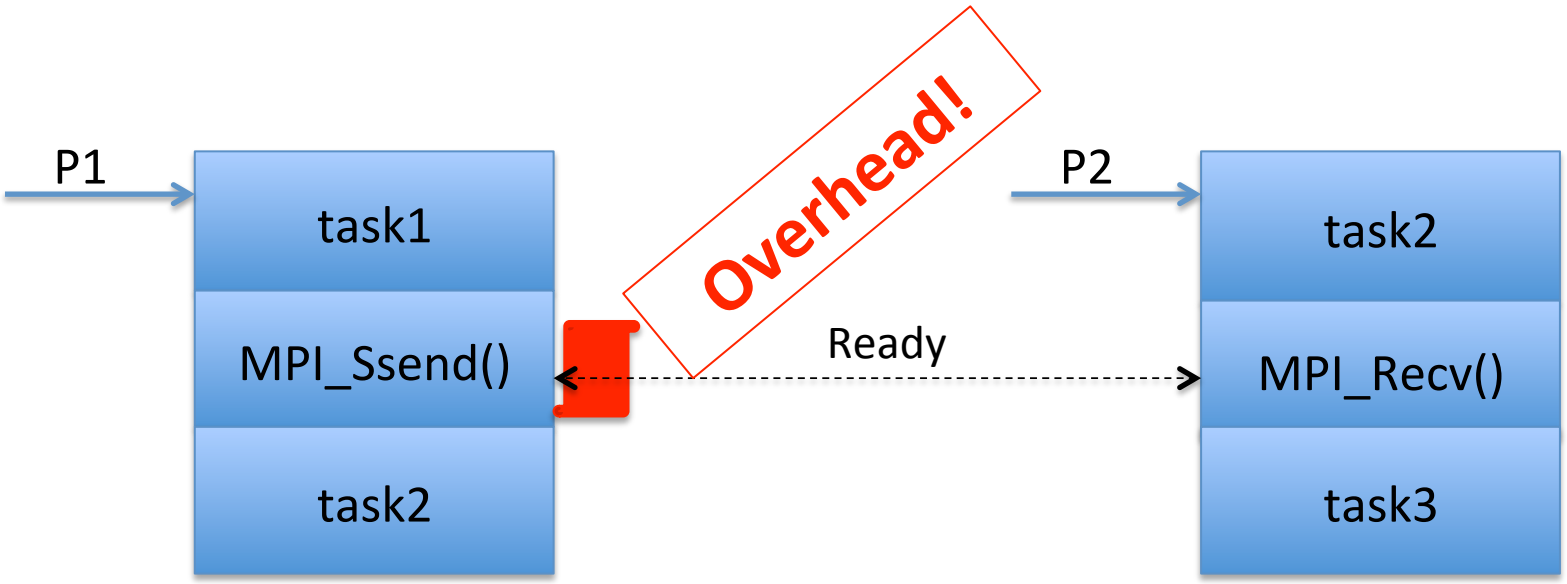
- Synchronous(MPI_Ssend())
  - No buffer
  - Processes wait for each other

Point-to-Point: Blocking modes

- Synchronous(MPI_Ssend())
  - No buffer
  - Processes wait for each other

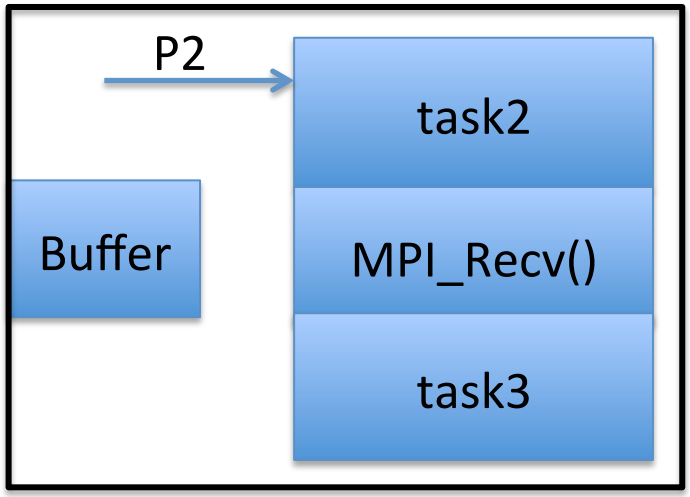
Point-to-Point: Blocking modes

- Synchronous(MPI_Ssend())
  - No buffer
  - Processes wait for each other



Point-to-Point: Blocking modes

- Buffered(MPI_Bsend())
  - Buffer needed
  - Send can start before receive



- Less overhead
- Overflow Error

Point-to-Point: Blocking modes

- Ready (MPI_Rsend())
  - No buffer no synchronization
  - Receiver must initiate it (I want Data!)

Lower sender overhead!

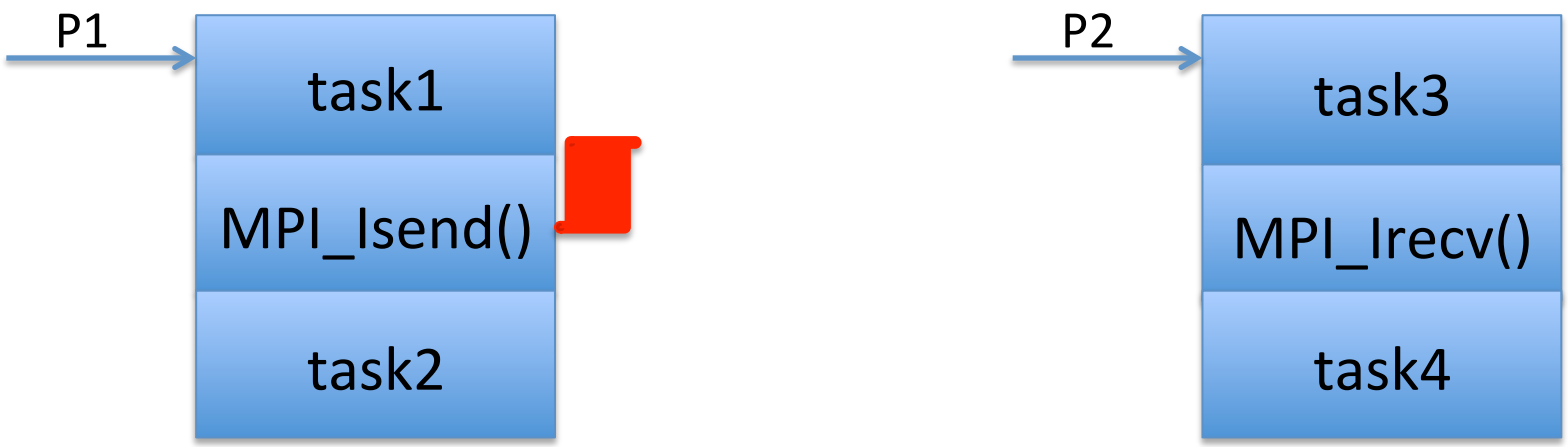
Point-to-Point: Blocking modes

- Standard (MPI_Send)
  - Mix of synchronized and buffered
  - Becomes synchronous if “buffer size” was exceeded
  - Buffer can be on receiver or sender side

Use it if you are not sure which blocking mode would fit your problem!

Point-to-Point: Non-Blocking modes

- Sender and receiver do not wait!
- Wait for a message:
  - MPI_Wait() (blocking)
- Check the status of the message:
  - MPI_Test() (non-blocking)



Point-to-Point

- Which one to use ?
  - MPI_Send()
  - MPI_Isend()
  - MPI_Rsend()
  - MPI_Bsend()
  - MPI_Recv()
  - MPI_Irecv()

Point-to-Point

- Which one to use ?
  - MPI_Send()
  - MPI_Isend()
  - MPI_Rsend()
  - MPI_Bsend()
  - MPI_Recv()
  - MP

Less dependency means higher performance!
However, it depends on the problem!

Example

```
#include <stdio.h>
#include <mpi.h>
int main (argc, argv){
	int rank, size;
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	MPI_Comm_size(MPI_COMM_WORLD, &size);
	int number;
	if (rank == 0) {
		number = 100;
		int K;
		for(K=1; K<size;K++){
			MPI_Send(&(number+K), 1, MPI_INT, K, 0, MPI_COMM_WORLD);
		}
	} else if (rank > 0) {
		MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
		printf("Process %d received number %d from process 0\n”, rank, number);
	}
	MPI_Finalize();
```

Example

```
#include <stdio.h>
#inclu
int ma
    in
    M
    M
    in
    if

    }

MPI_S
                                                                    r);
    }
    MPI_Finalize();
}
```

- **mpicc -o SR-hello-world.exe SR-hello-world.cc**
- **mpirun –n 8 SR-hello-world.exe**

```
Process 1 received number 101 from process 0!
Process 5 received number 105 from process 0!
Process 7 received number 107 from process 0!
Process 3 received number 103 from process 0!
Process 4 received number 104 from process 0!
Process 6 received number 106 from process 0!
Process 2 received number 102 from process 0!
```

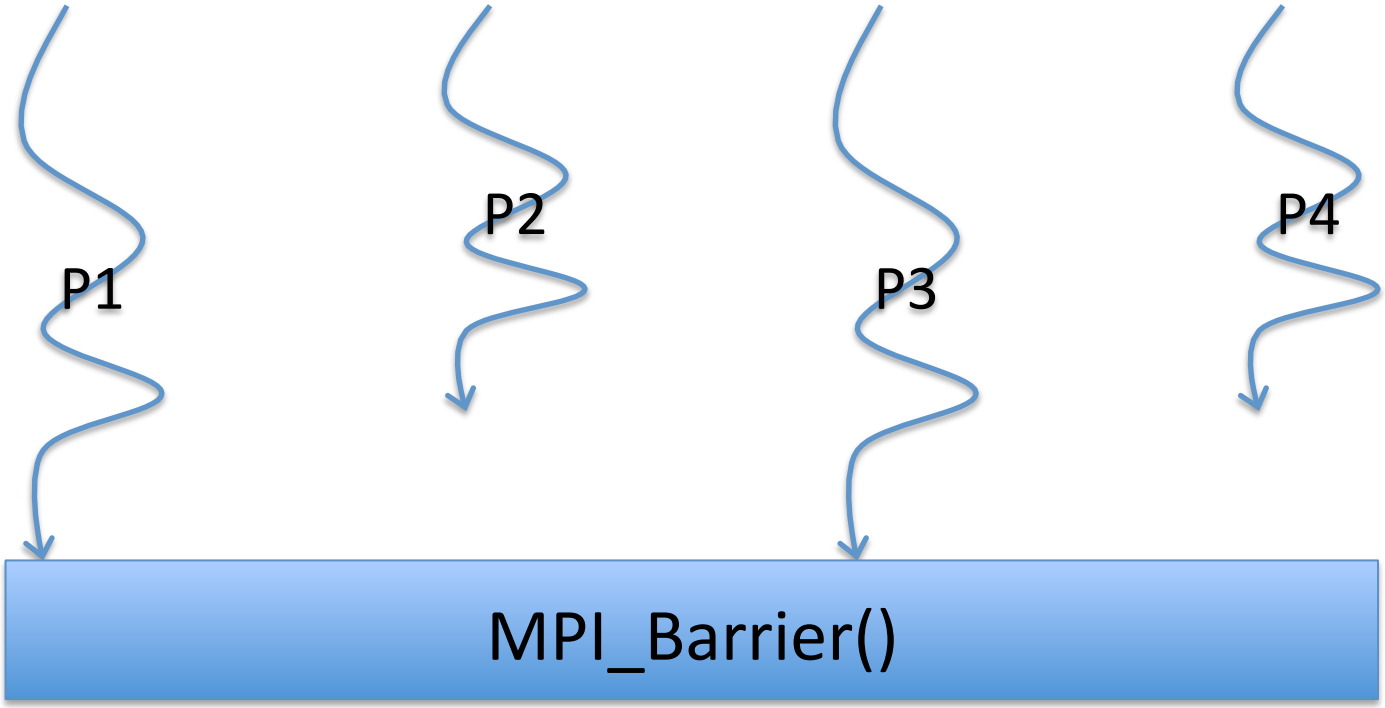
Collective operations

- Between multiple processes within the same communicator
  - One-to-many
  - many-to-one
  - many-to-many

- Blocking operations

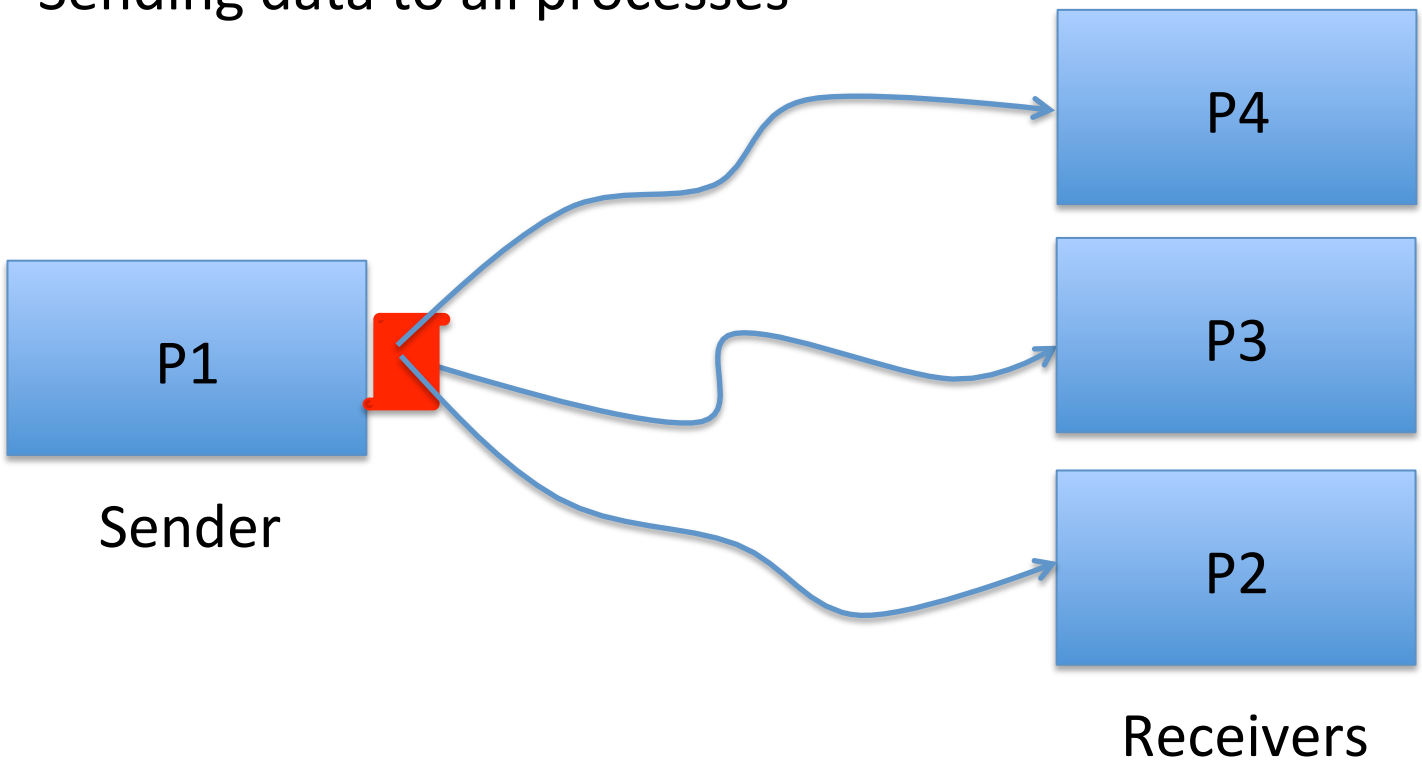
Collective operations: Synchronization

- MPI_Barrier(MPI_COMM_WORLD)
  - Stops all the processes within a communicator



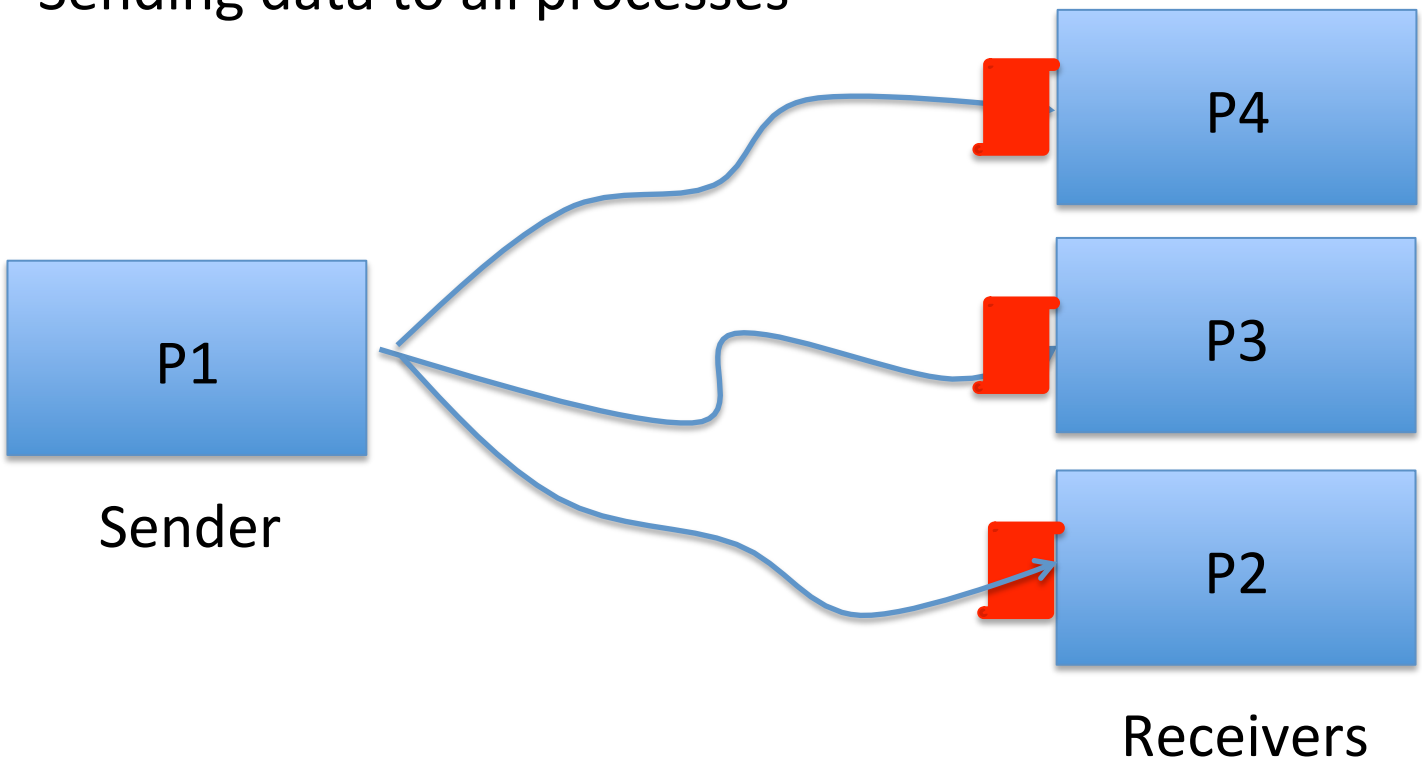
Collective operations: Broadcasting

- MPI_Bcast()
  - Announcing!
  - Sending data to all processes



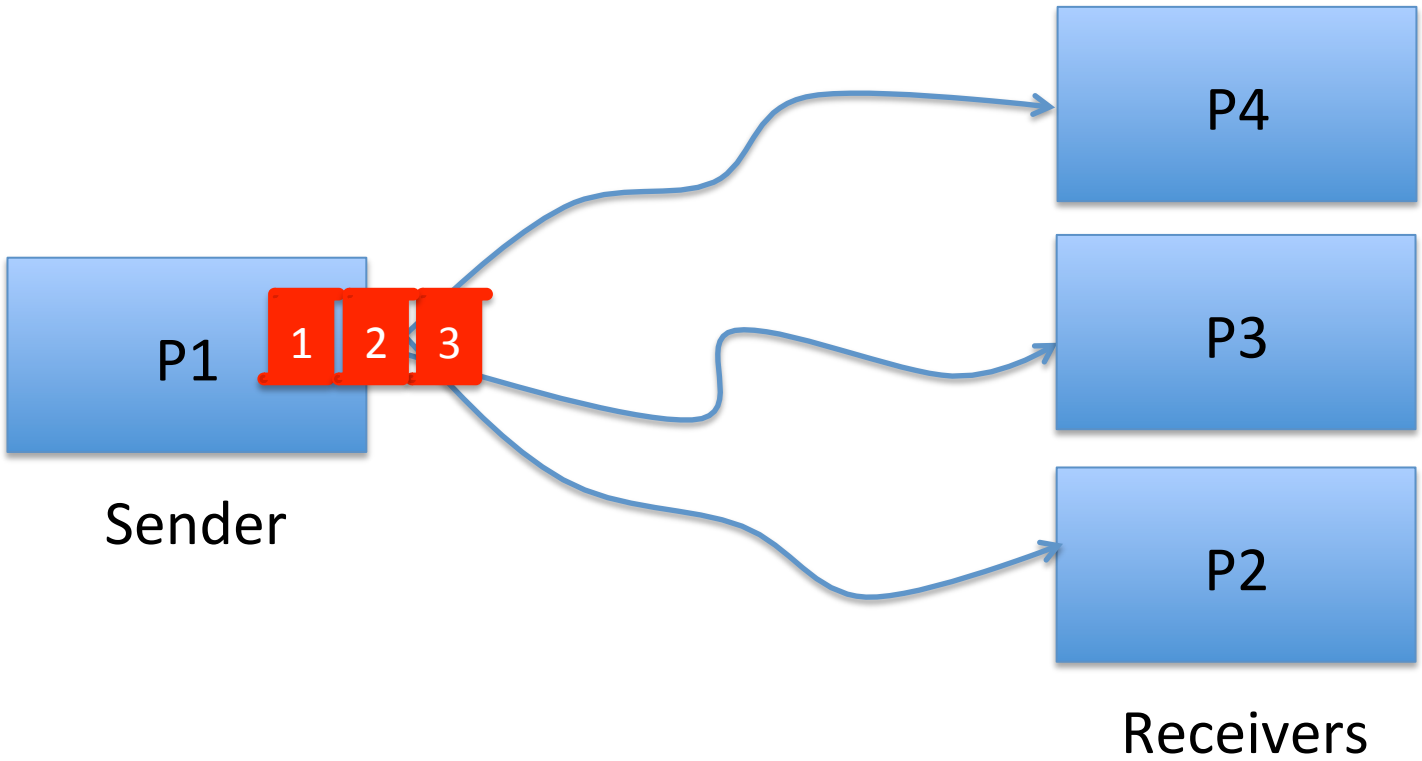
Collective operations: Broadcasting

- MPI_Bcast()
  - Announcing!
  - Sending data to all processes



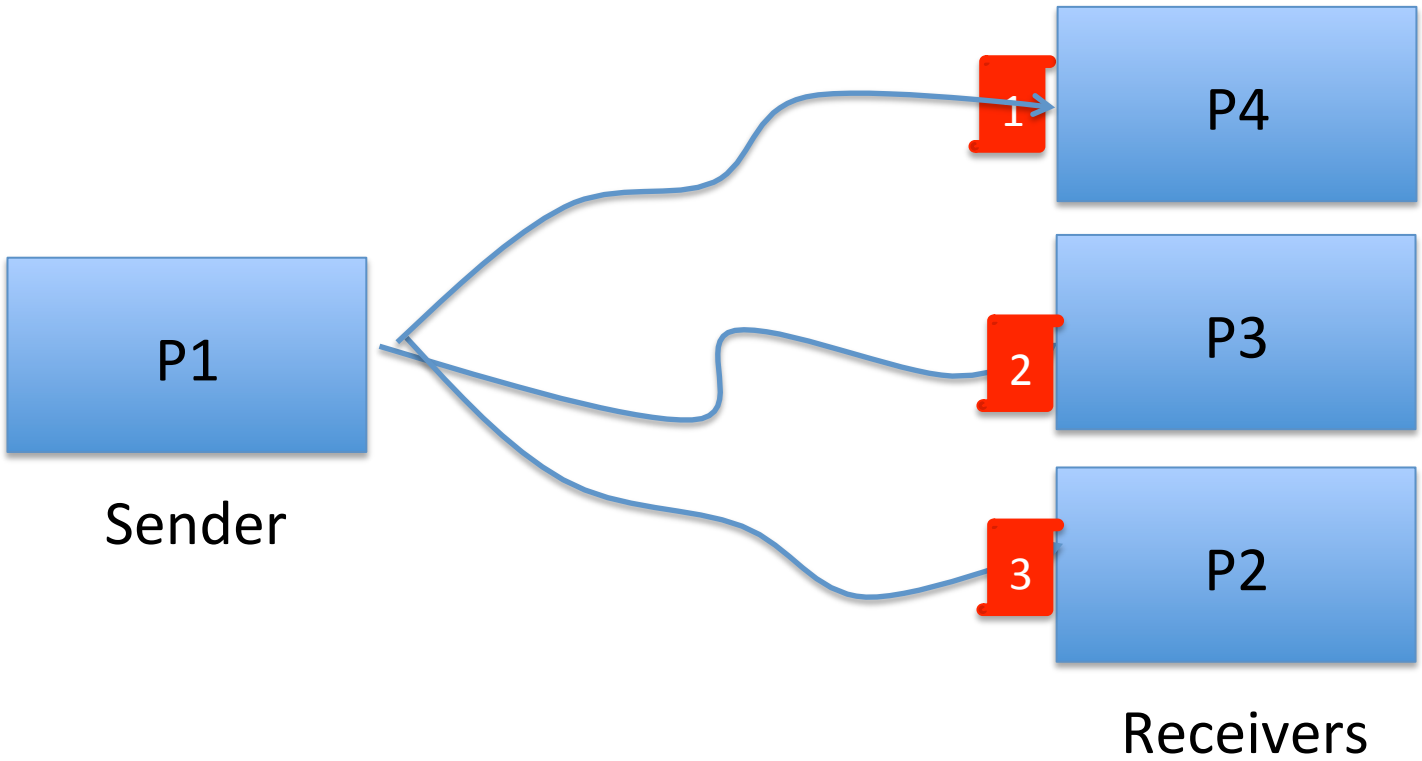
Collective operations: Scattering

- MPI_Scatter()
  - Divide data between processes



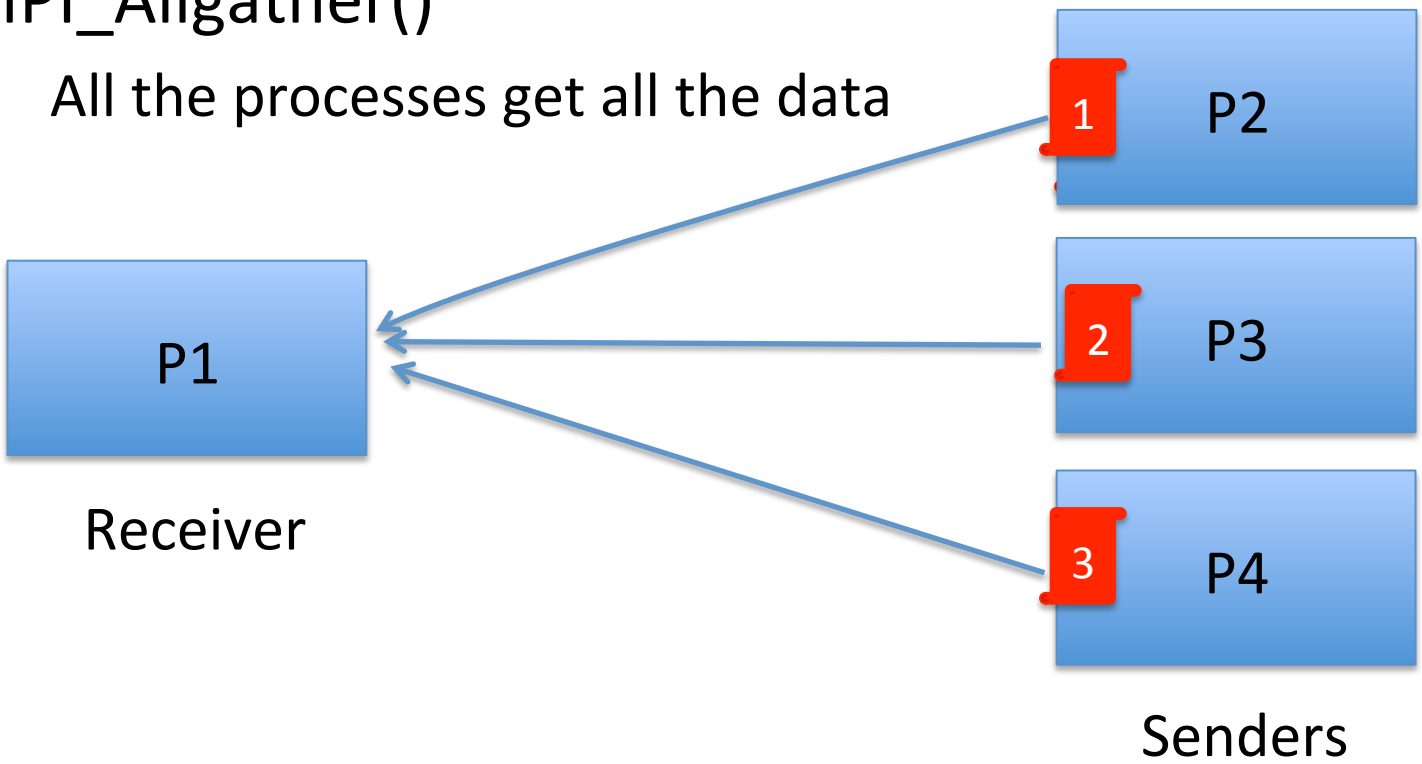
Collective operations: Scattering

- MPI_Scatter()
  - Divide data between processes



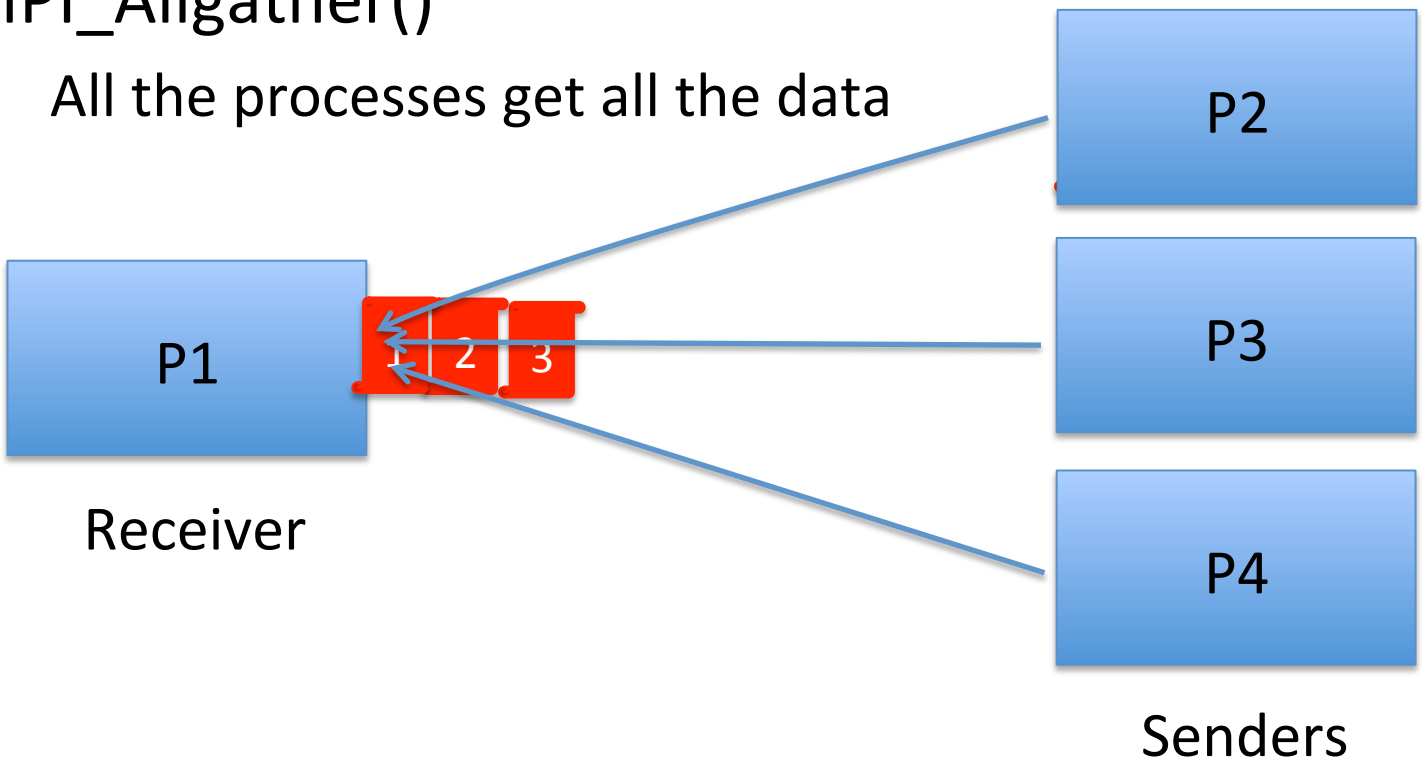
Collective operations: Gathering

- MPI_Gather()
  - Gathering data from all processes
- MPI_Allgather()
  - All the processes get all the data



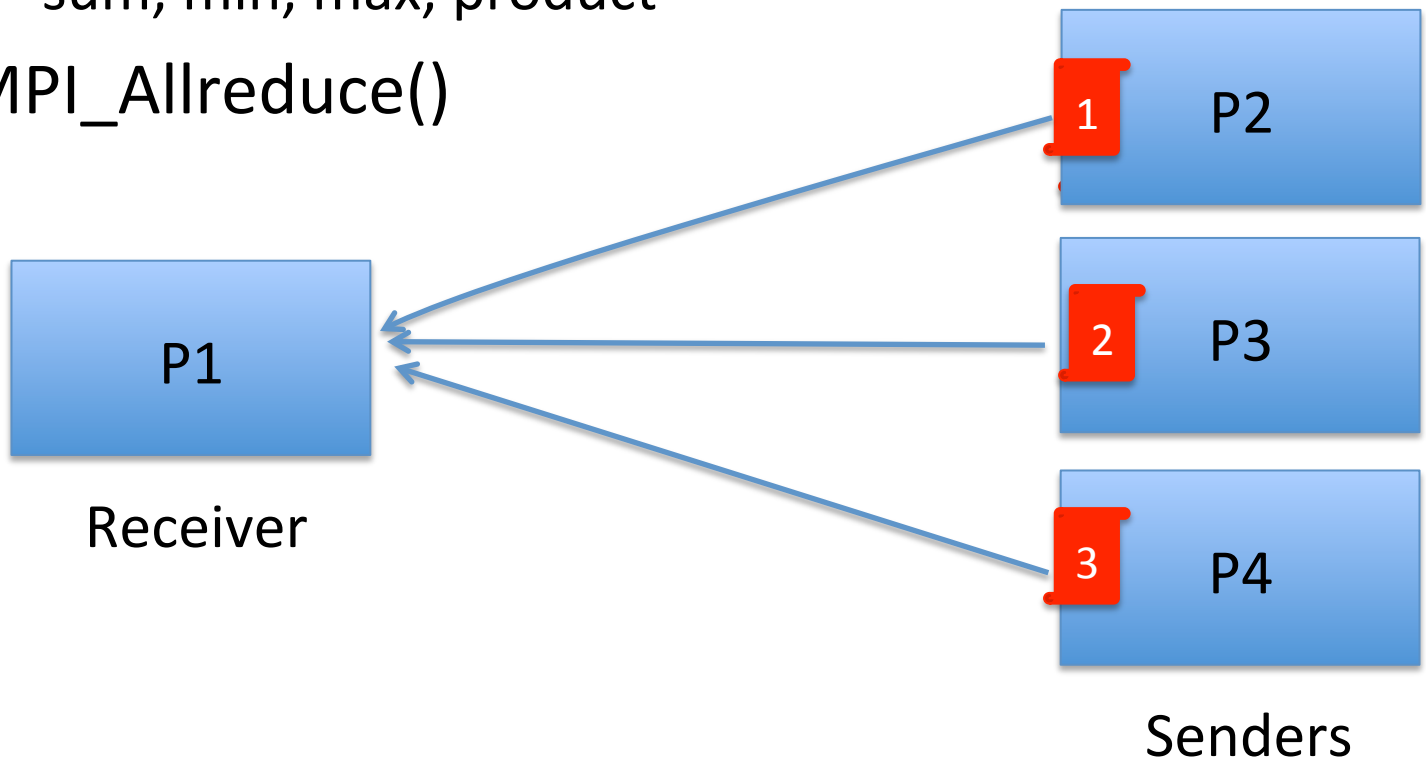
Collective operations: Gathering

- MPI_Gather()
  - Gathering data from all processes
- MPI_Allgather()
  - All the processes get all the data



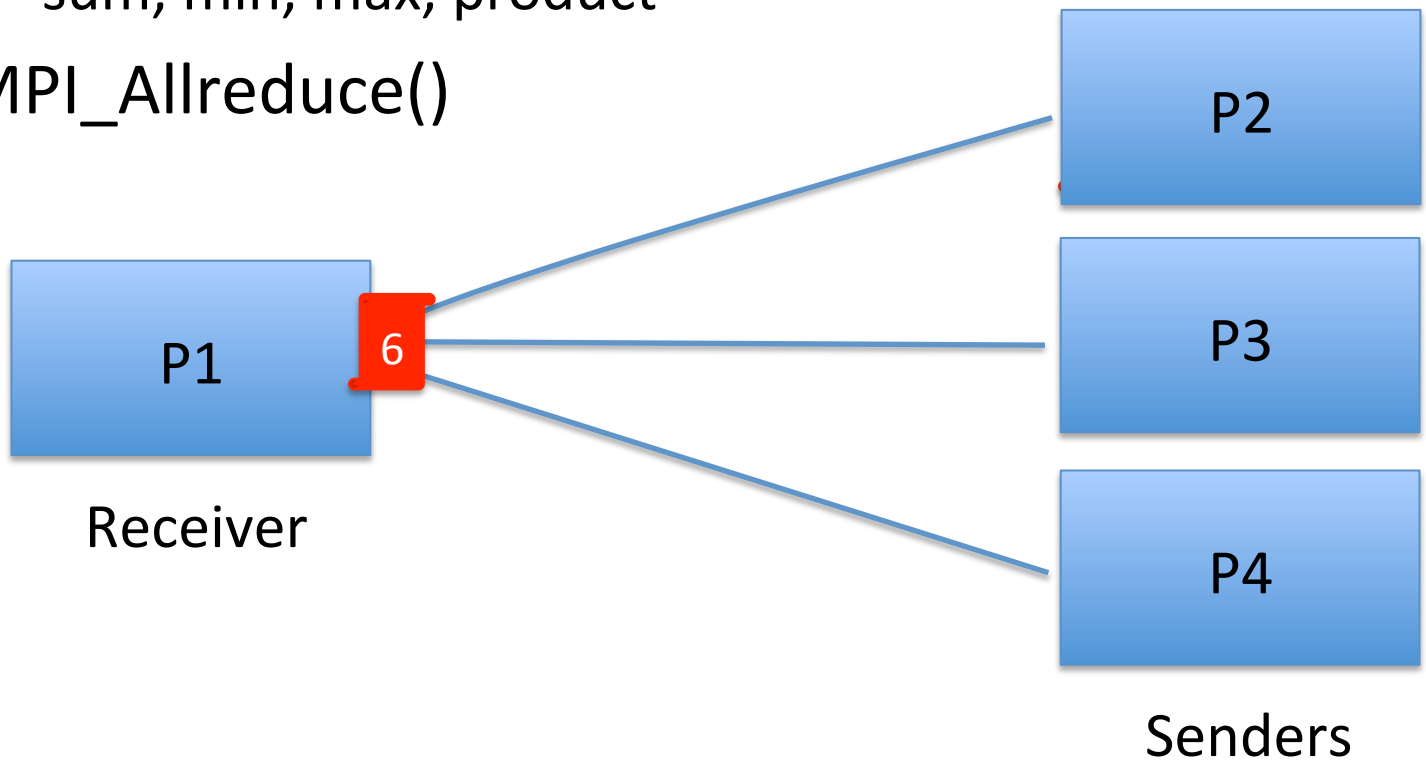
Collective operations: Reducing

- MPI_Reduce()
  - Apply reduction operation on the grouped data
  - sum, min, max, product
- MPI_Allreduce()



Collective operations: Reducing

- MPI_Reduce()
  - Apply reduction operation on the grouped data
  - sum, min, max, product
- MPI_Allreduce()



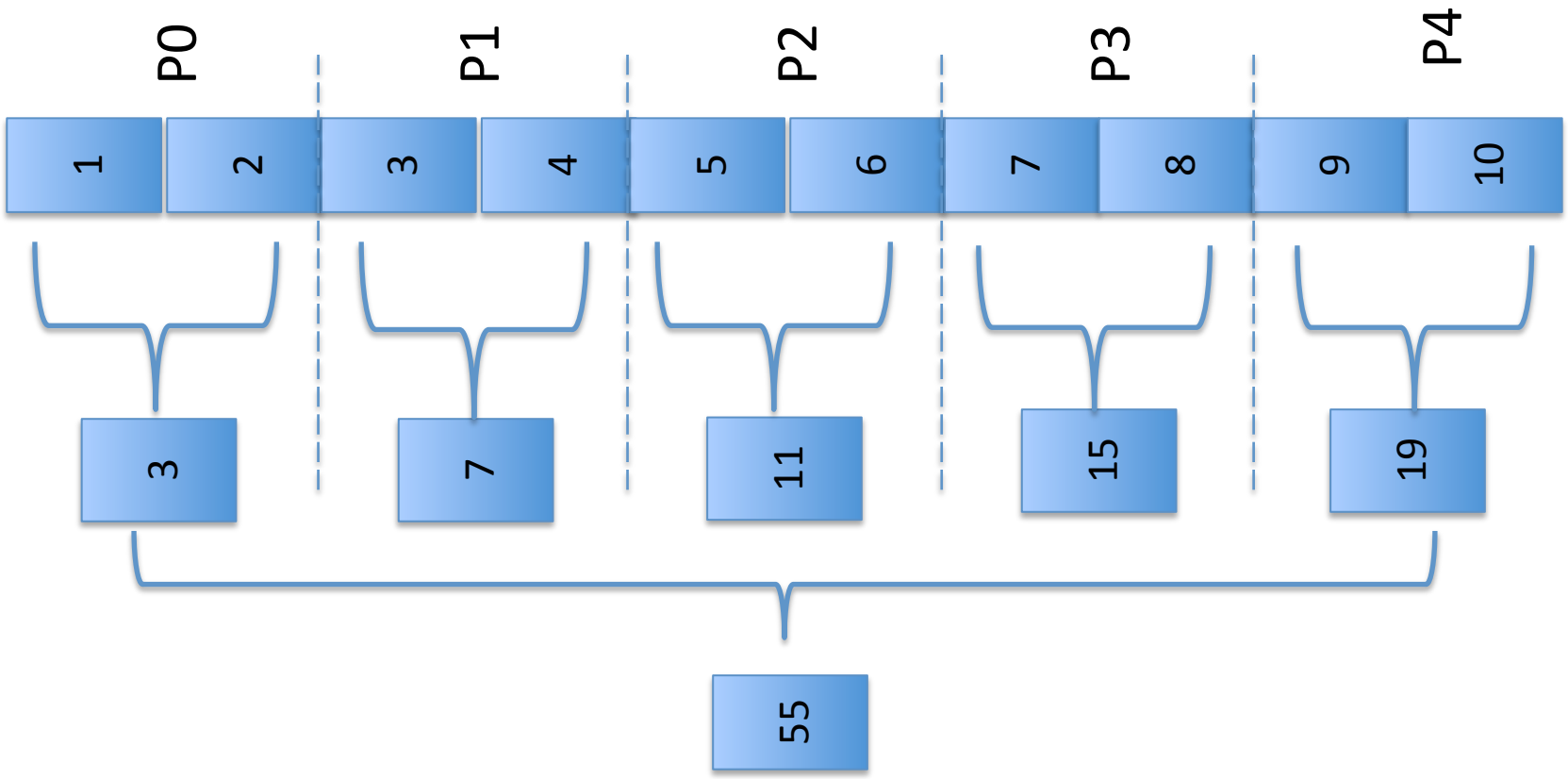
Collective operations: Other functions

- MPI_Reduce_scatter()
- MPI_All_to_all()
- MPI_Scatterv()
- MPI_Gatherv()
- …

Example: Sequential

```
#include <stdio.h>
int main (argc, argv){
    int i, sum, limit;
    limit = 10;
    sum=0;
    for(i=1; i<= limit; i++){
        sum = sum +i;
    }
    printf("\nSum= %d\n", sum);
    return 0;
}
```

Example: Parallel



```
#include <stdio.h>
#include ''mpi.h''
int main (argc, argv){
	int i, p_sum, limit, tsum;
	int start, end, size, rank;
	MPI_Init(&argc,&argv);
	MPI_Comm_size(MPI_COMM_WORLD, &size);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	start = rank*(limit/size) + 1;
	if (rank==(size-1)){
			end = limit;
	} else{
		end = start + (limit/size)-1;
	 }
	 psum = 0; tsum=0;
	for(i=start; i<= end; i++){
		psum = psum +i;
	}
	 MPI_Reduce (&psum, & tsum, 1, MPI_INT, MPI_SUM, 4, MPI_COMM_WORLD );
	printf("\nRank: %d, sum: %d, sumTotal: %d\n", rank, psum, tsum);
	 MPI_Finalize();
	return 0;
}
```

```
#include <stdio.h>
#include ''mpi.h''
int main (argc, argv){
    int
    int
    MP
    MP
    MP
    sta
    if (
    } e
     }
     ps
    for

    }
     M                                                                    0 );
    pri
     MPI_Finalize();
    return 0;
}
```

- **mpicc –o psum.exe parallelsum.cc**

- **mpirun –n 5 psum.exe**

```
Rank: 1, sum: 7, sumTotal: 0
Rank: 4, sum: 19, sumTotal: 55
Rank: 2, sum: 11, sumTotal: 0
Rank: 0, sum: 3, sumTotal: 0
Rank: 3, sum: 15, sumTotal: 0
```

Conclusion

- Message passing interface
- Point to point communication
- Collective operations

References

- https://computing.llnl.gov/tutorials/mpi/
- http://www.mpi-forum.org/docs/
- http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf