MPI: Datatypes

MPI_Datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long in
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	no conversion, bitpattern transferred as is
MPI_PACKED	grouped messages

Querying for Information

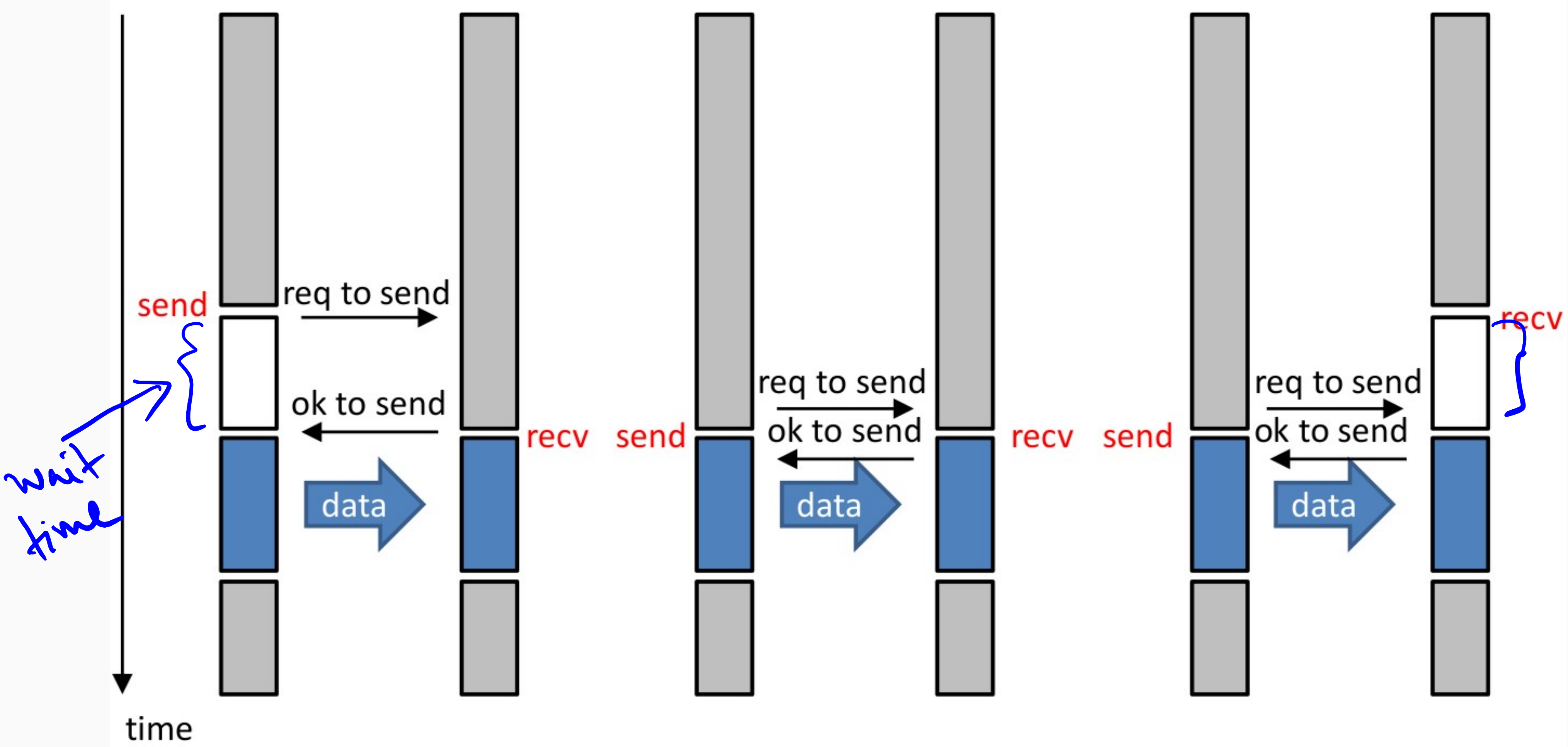
- `MPI_Status`
  - Stores information about the MPI_Recv operation

```
typedef struct MPI_Status {
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
}
```

  - Does not contain the size of the received message

- `int MPI_Get_count (MPI_Status *status, MPI_Datatype datatype, int *count)`
  - returns the number of data items received in the count variable
  - not directly accessible from status variable

Send and Receive



a) sender comes first, **idling at sender** (no buffering of message)

b) sending/receiving at about the same time, **idling minimized**

c) receiver comes first, **idling at receiver**

Deadlocks in MPI

```
int a[10], b[10], myRank;
MPI_Status s1, s2;
MPI_Comm_rank( MPI_COMM_WORLD, &myRank);

if ( myRank == 0 ) {
    MPI_Send( a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD );
    MPI_Send( b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD );
}
else if ( myRank == 1 ) {
    MPI_Recv( b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD, &s1 );
    MPI_Recv( a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD, &s2 );
}
```

rank 1 is trying to receive message from rank 0 with tag 2, but rank 0 has sent message to rank 1 with tag 1.

Is this a deadlock. If yes, then why?

Since 1st MPI_Send does not match 1st MPI_Recv, it is a deadlock.

Modelling Transmission cost in MPI

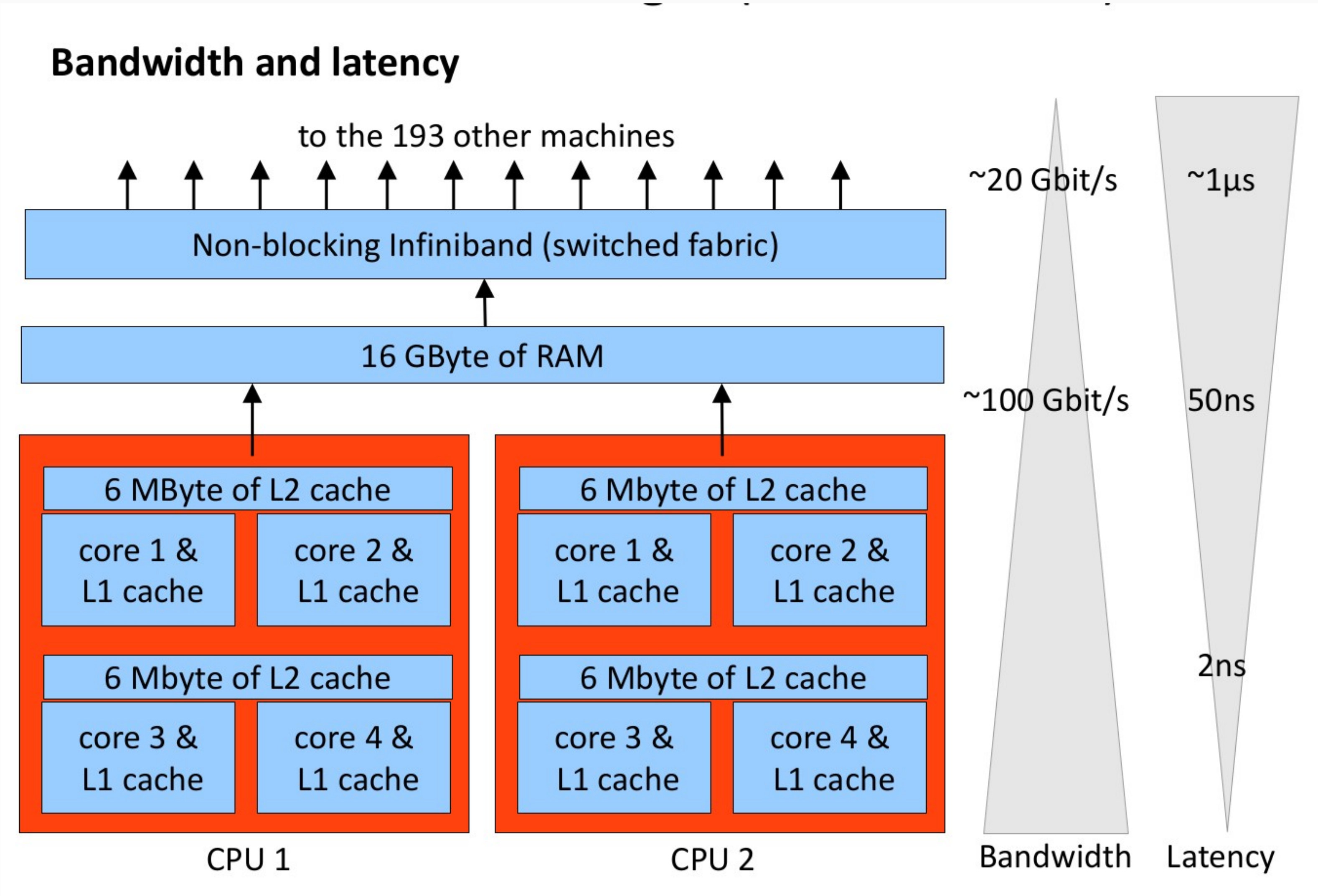
- Various choices for interconnection networks
  - Gigabit Ethernet: cheap, but far too slow for HPC applications
  - Infiniband /Myrinet: high speed interconnect
- Performance model for point to point communication

$$T_{comm} = \alpha + \beta n$$

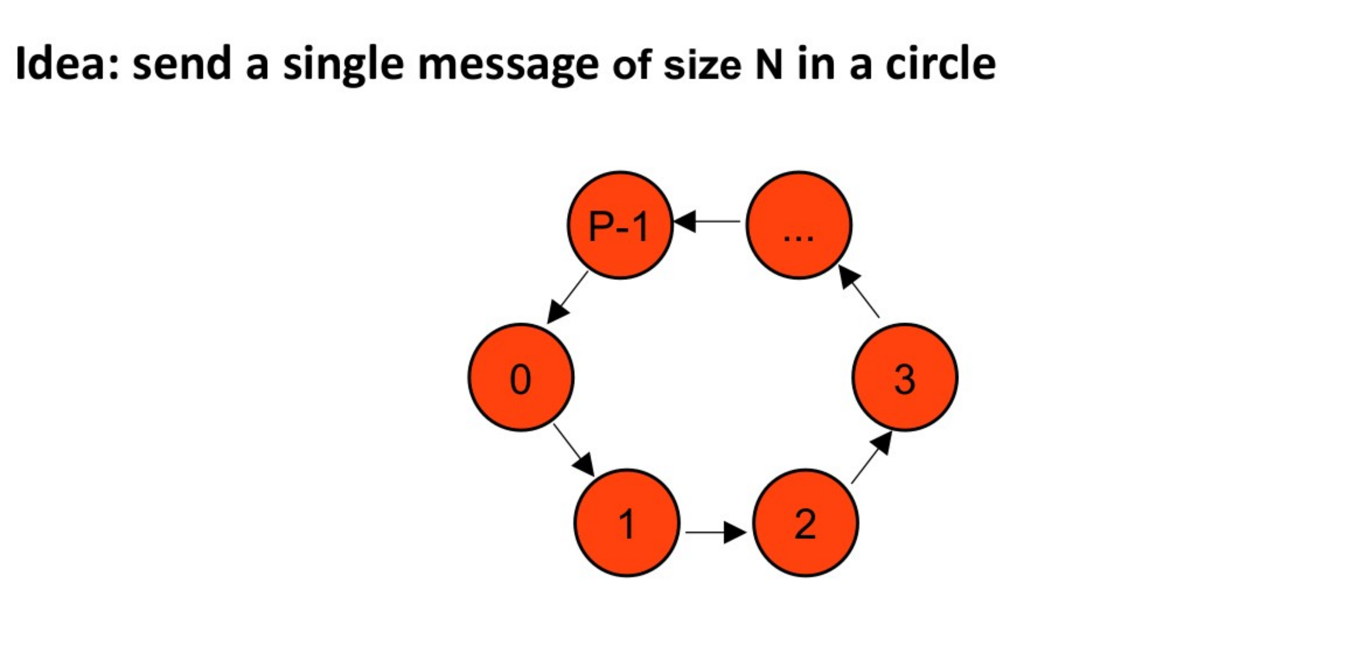
- α : latency (time to transfer zero bytes)
- $B = 1/\beta =$ saturation bandwidth (bytes/s)
- n number of bytes to transmit
- Effective bandwidth:

$$B_{eff} = \frac{n}{\alpha + \beta n} = \frac{n}{\alpha + n/B}$$

Bandwidth and Latency



Ring Test



- Increase the message size N
  - 1 byte, 2 bytes, ..., 1024 bytes, 2048 bytes, 4096 bytes
- Benchmark the results
  - Bandwidth $= N * P / T$

Send Ring

```
void sendRing( char *buffer, int length ) {
    /* send message in a ring here */
}

int main( int argc, char * argv[] )
{
    ...
    char *buffer = (char*) calloc ( 1048576, sizeof(char) );
    int msgLen = 8;
    for (int i = 0; i < 18; i++, msgLen *= 2) {
        double startTime = MPI_Wtime();
        sendRing( buffer, msgLen );
        double stopTime = MPI_Wtime();
        double elapsedSec = stopTime - startTime;
        if (rank == 0)
            printf( "Bandwidth for size %d is : %f\", ... );
    }
    ...
}
```

Send Ring

```
void sendRing( char *buffer, int msgLen )
{
    int myRank, numProc;
    MPI_Comm_rank( MPI_COMM_WORLD, &myRank );
    MPI_Comm_size( MPI_COMM_WORLD, &numProc );
    MPI_Status status;

    int prevR = (myRank - 1 + numProc) % numProc;
    int nextR = (myRank + 1 ) % numProc;

    if (myRank == 0) {        // send first, then receive
        MPI_Send( buffer, msgLen, MPI_CHAR, nextR, 0, MPI_COMM_WORLD);
        MPI_Recv( buffer, msgLen, MPI_CHAR, prevR, 0, MPI_COMM_WORLD,
                  &status );
    } else {                  // receive first, then send
        MPI_Recv( buffer, msgLen, MPI_CHAR, prevR, 0, MPI_COMM_WORLD,
                  &status );
        MPI_Send( buffer, msgLen, MPI_CHAR, nextR, 0, MPI_COMM_WORLD);
    }
}
```

Basic MPI Routines

Timing routines in MPI

- `double MPI_Wtime( void )`
  - returns the time in seconds relative to “some time” in the past
  - “some time” in the past is fixed during process
- `double MPI_Wtick( void )`
  - Returns the resolution of MPI_Wtime() in seconds
  - e.g. 10^{-3} = millisecond resolution

Basic MPI Routines

```
int MPI_Sendrecv( void *sendbuf, int sendcount, MPI_Datatype
    sendtype, int dest, int sendtag, void *recvbuf,
    int recvcount, MPI_Datatype recvtype, int source,
    int recvtag, MPI_Comm comm, MPI_Status *status )
```

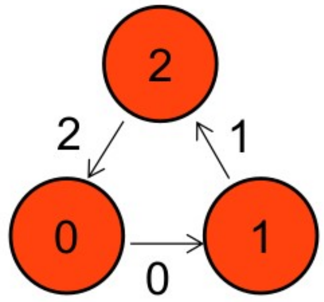
- **sendbuf**: pointer to the message to send
- **sendcount**: number of elements to transmit
- **sendtype**: datatype of the items to send
- **dest**: rank of destination process
- **sendtag**: identifier for the message
- **recvbuf**: pointer to the buffer to store the message (**disjoint** with sendbuf)
- **recvcount**: **upper bound (!)** to the number of elements to receive
- **recvtype**: datatype of the items to receive
- **source**: rank of the source process (or MPI_ANY_SOURCE)
- **recvtag**: value to identify the message (or MPI_ANY_TAG)
- **comm**: communicator specification (e.g. MPI_COMM_WORLD)
- **status**: structure that contains { MPI_SOURCE, MPI_TAG, MPI_ERROR }
- **sendbuf**: pointer to the buffer to send

```
int MPI_Sendrecv_replace( ... )
```

- Buffer is replace by received data

Basic MPI Routines

Sendrecv example




```
const int len = 10000;
int a[len], b[len];
if ( myRank == 0 ) {
    MPI_Send( a, len, MPI_INT, 1, 0, MPI_COMM_WORLD );
    MPI_Recv( b, len, MPI_INT, 2, 2, MPI_COMM_WORLD, &status
    );
} else if ( myRank == 1 ) {
    MPI_Sendrecv( a, len, MPI_INT, 2, 1, b, len, MPI_INT, 0,
                  0, MPI_COMM_WORLD, &status );
} else if ( myRank == 2) {
    MPI_Sendrecv( a, len, MPI_INT, 0, 2, b, len, MPI_INT, 1,
                  1, MPI_COMM_WORLD, &status );
}
```

safe to exchange !

- Compatibility between Sendrecv and normal send and recv
- Sendrecv can help to prevent deadlocks

Non blocking communication

Idea:

- Do something useful while waiting for communications to finish
- Try to overlap communications with computations

How?

- Replace blocking communication by non-blocking variants
  - Replace `MPI_Send()` by `MPI_Isend(..., MPI_Request *request`
  - Replace `MPI_Recv()` by `MPI_Irecv(..., MPI_Request *request`

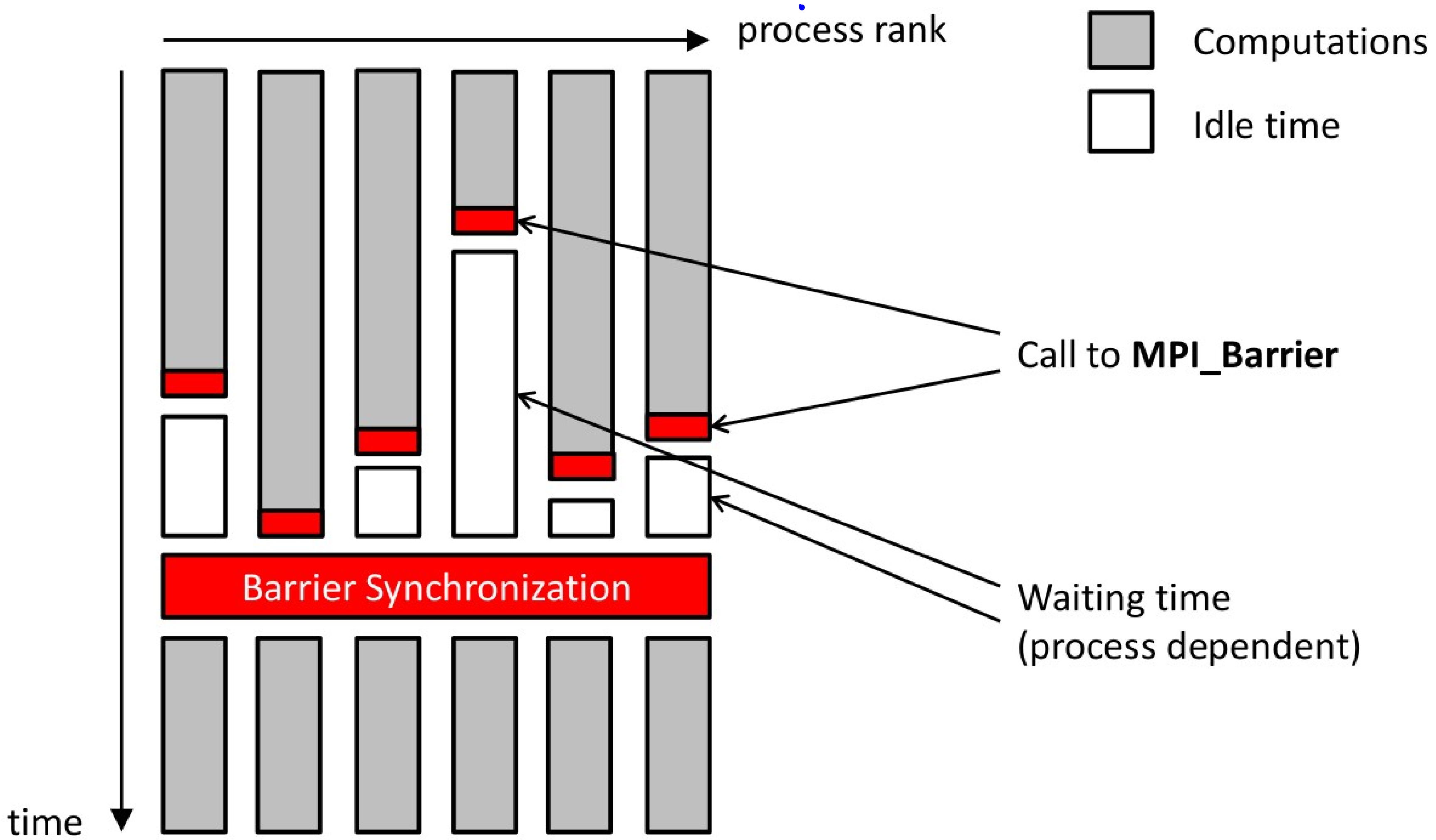
Exercise:
Replace both MPI_Send and MPI_Recv by MPI_Isend and MPI_Irecv and there won't be deadlock!

Collective Communication: Barrier Synchronization

`MPI_Barrier(MPI_Comm comm)`

This function does not return until all processes in comm have called it.

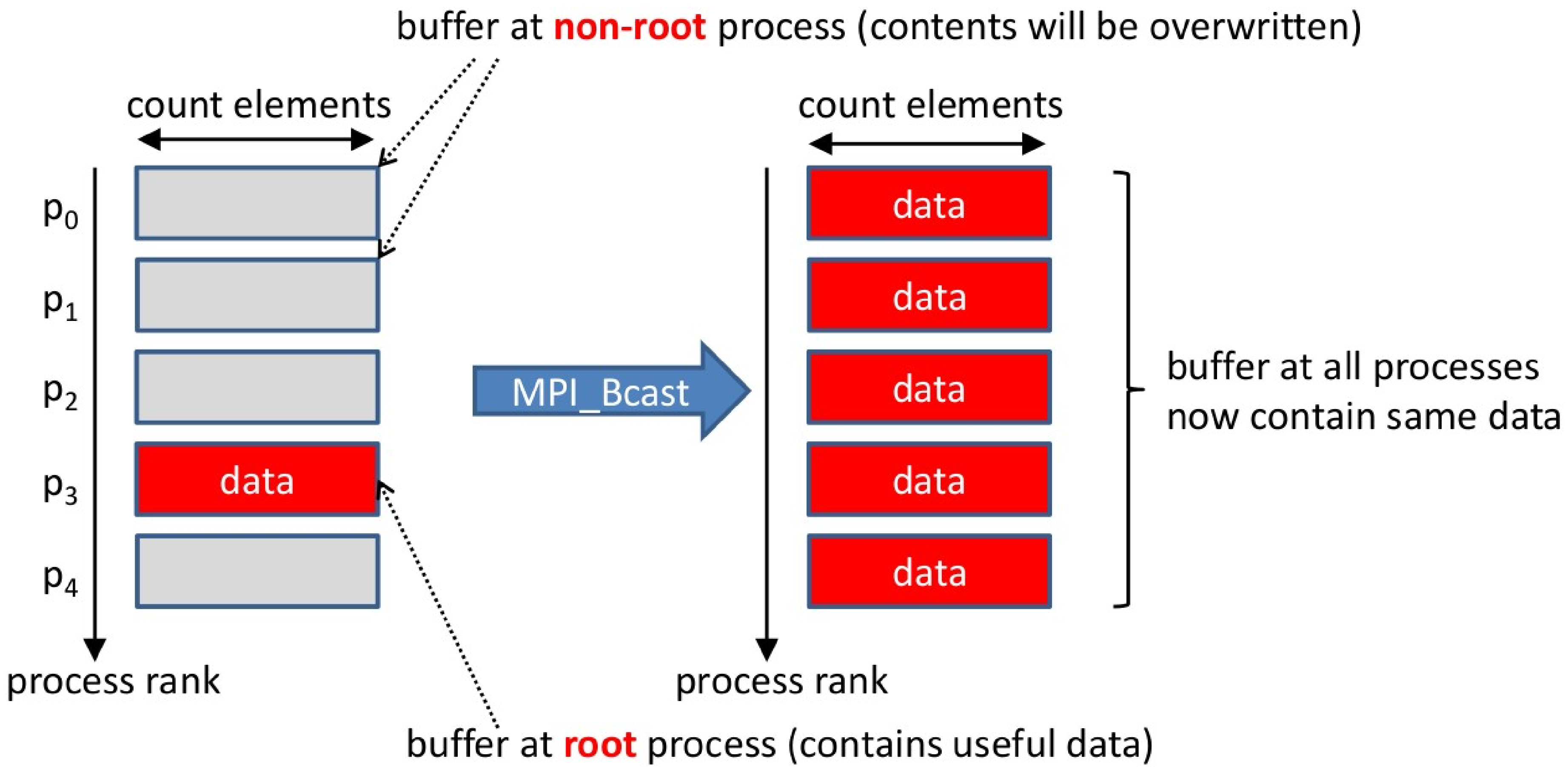
↑ note!



Collective Communication: Broadcast

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

MPI_Bcast broadcasts `count` elements of type `datatype` stored in `buffer` at the `root` process to all other processes in `comm` where this data is stored in `buffer`.



Collective Communication: Broadcast

```
...
int rank, size;

...   // init MPI and rank and size variables

int root = 0;
char buffer[12];

if (rank == root)
       sprintf(buffer, "Hello world");

MPI_Bcast(buffer, 12, MPI_CHAR, root, MPI_COMM_WORLD);

printf("Process %d has %s stored in the buffer.\n", buffer, rank);

...
```

fill the buffer at the root process only

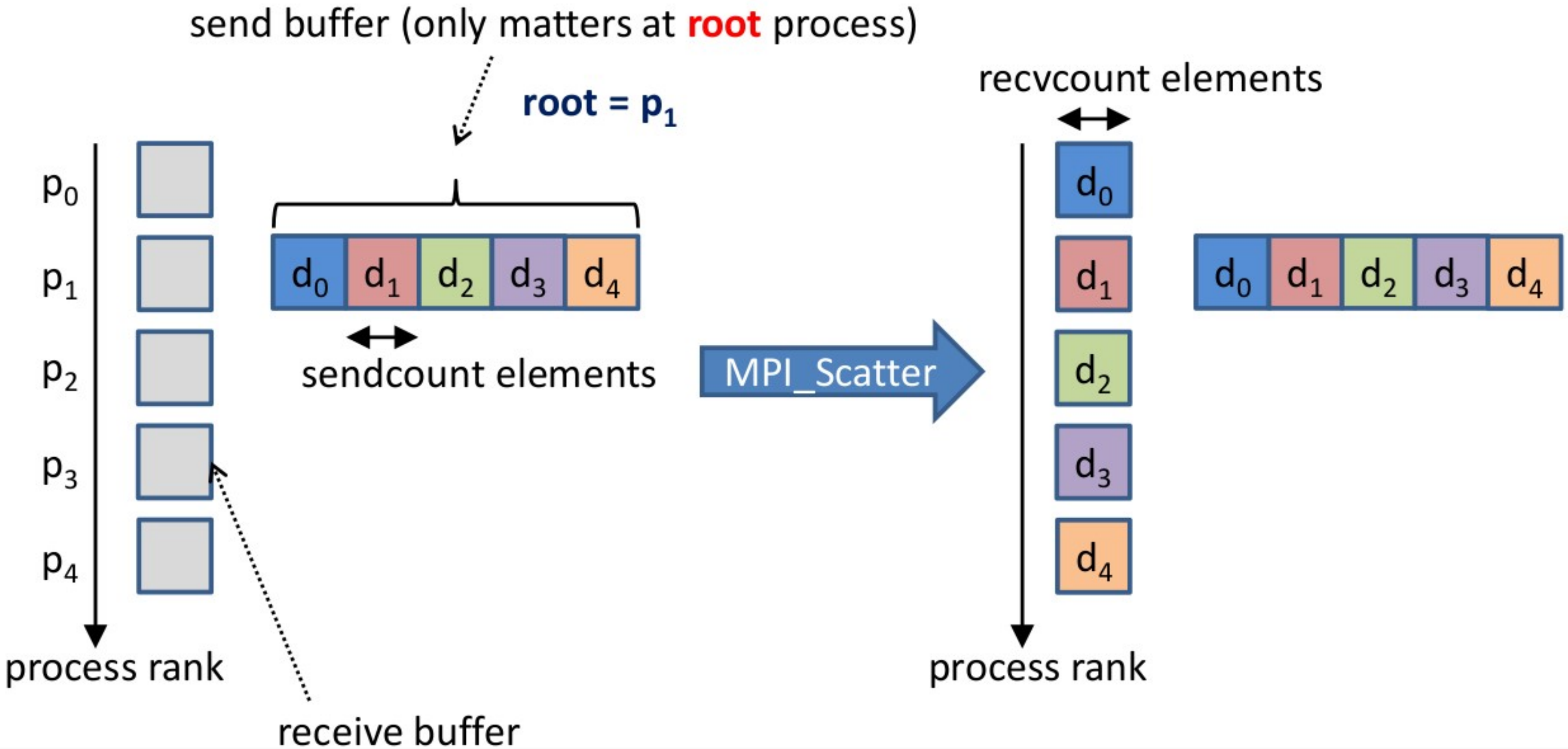
all processes must call MPI_Bcast

```
john@doe ~]$ mpirun -np 4 ./broadcast
Process 1 has Hello World stored in the buffer.
Process 0 has Hello World stored in the buffer.
Process 3 has Hello World stored in the buffer.
Process 2 has Hello World stored in the buffer.
```

Collective Communication: Scatter

```
MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendType,
            void *recvbuf, int recvcount, MPI_Datatype recvType,
            int root, MPI_Comm comm)
```

MPI_Scatter partitions a `sendbuf` at the `root` process into P equal parts of size `sendcount` and sends each process in `comm` (including root) a portion in rank order.



Collective Communication: Scatter Example

```
int root = 0;
char recvBuf[7];

if (rank == root) {
    char sendBuf[25];
    sprintf(sendBuf, "This is the source data.");

    MPI_Scatter(sendBuf, 6, MPI_CHAR, recvBuf, 6, MPI_CHAR,
                root, MPI_COMM_WORLD);
} else {
    MPI_Scatter(NULL, 0, MPI_CHAR, recvBuf, 6, MPI_CHAR,
                root, MPI_COMM_WORLD);
}

recvBuf[6] = '\0';
printf("Process %d has %s in receive buffer\n", rank, recvBuf);

...
```

fill the send buffer at the root process only

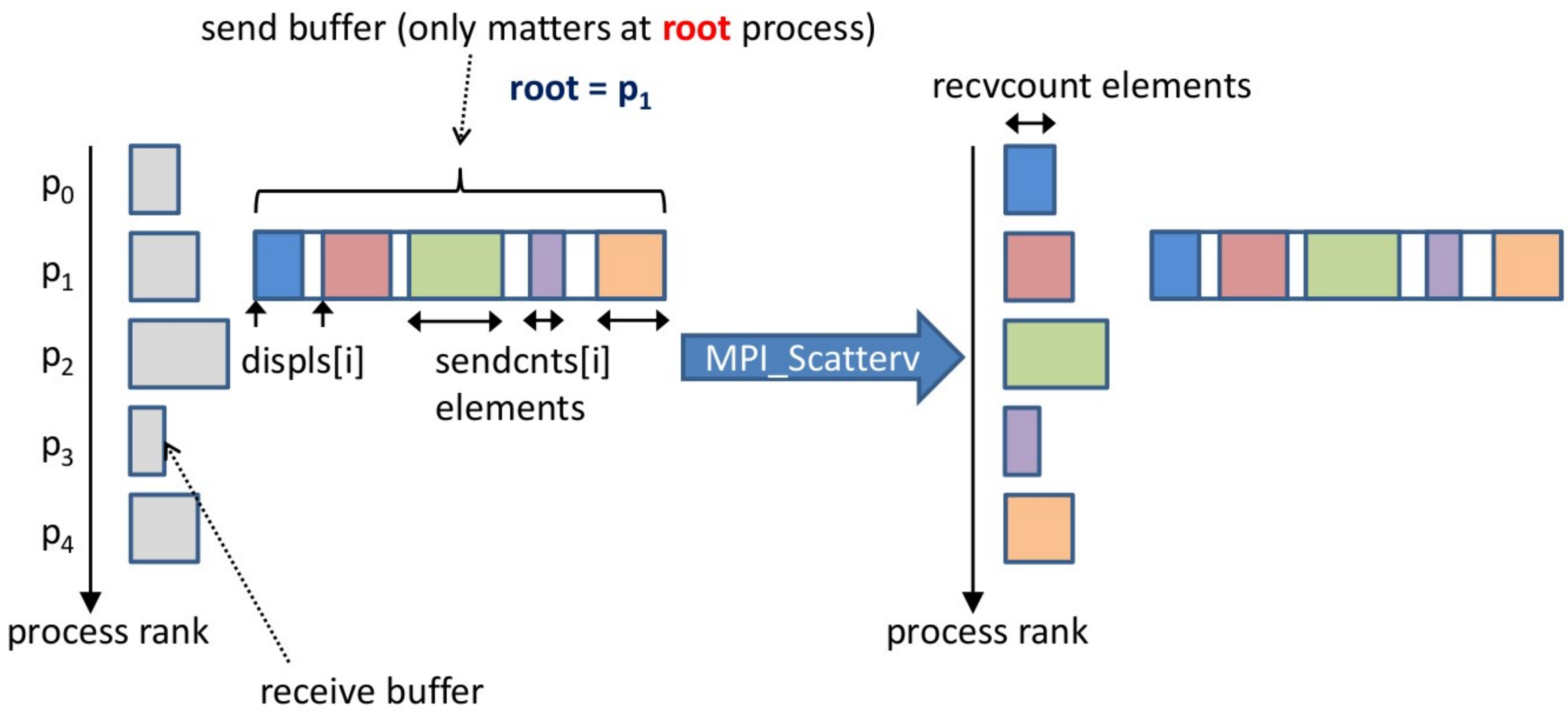
first three parameters are ignored on non-root processes

```
john@doe ~]$ mpirun -np 4 ./scatter
Process 1 has s the  stored in the buffer.
Process 0 has This i stored in the buffer.
Process 3 has  data. stored in the buffer.
Process 2 has source stored in the buffer.
```

Collective Communication: Variable Scatter Example

```
MPI_Scatterv(void *sendbuf, int *sendcnts, int *displs,
             MPI_Datatype sendType, void *recvbuf, int recvcnt,
             MPI_Datatype recvType, int root, MPI_Comm comm)
```

Partitions of `sendbuf` don't need to be of equal size and are specified per receiving process: the first index by the `displs` array, their size by the `sendcnts` array.

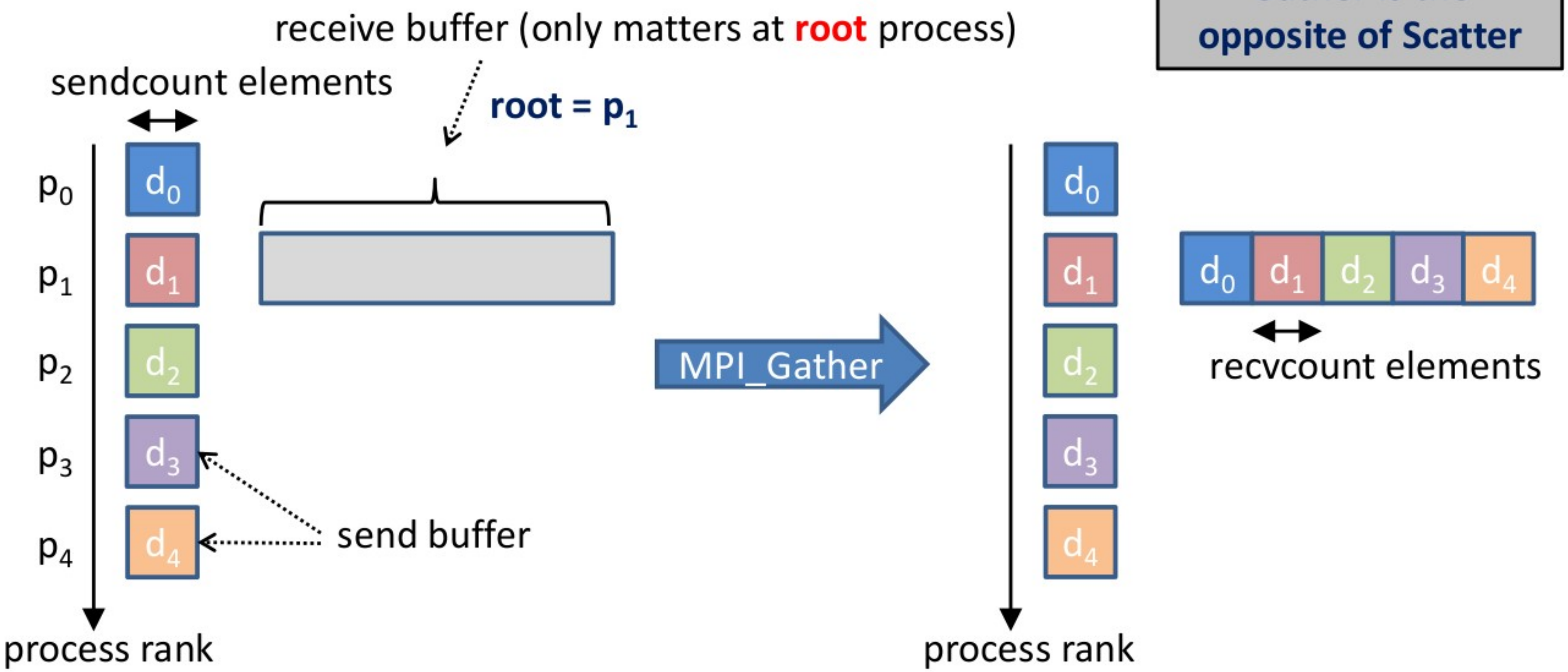


Collective Communication: Gather

```
MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendType,
           void *recvbuf, int recvcount, MPI_Datatype recvType,
           int root, MPI_Comm comm)
```

MPI_Gather gathers equal partitions of size `recvcount` from each of the P processes in `comm` (including root) and stores them in `recvbuf` at the `root` process in rank order.

Gather is the opposite of Scatter



A vector variant, **MPI_Gatherv**, exists, a similar generalization as MPI_Scatterv

Collective Communication: Gather Example

```
int root = 0;
int sendBuf = rank;

if (rank == root) {
    int *recvBuf = new int[size];

    MPI_Gather(&sendBuf, 1, MPI_INT, recvBuf, 1, MPI_INT,
               root, MPI_COMM_WORLD);
    cout << "Receive buffer at root process: " << endl;
    for (size_t i = 0; i < size; i++)
        cout << recvBuf[i] << " ";
    cout << endl;

    delete [] recvBuf;
} else {
    MPI_Gather(&sendBuf, 1, MPI_INT, NULL, 1, MPI_INT,
               root, MPI_COMM_WORLD);
}
```

receive buffer exists at the root process only

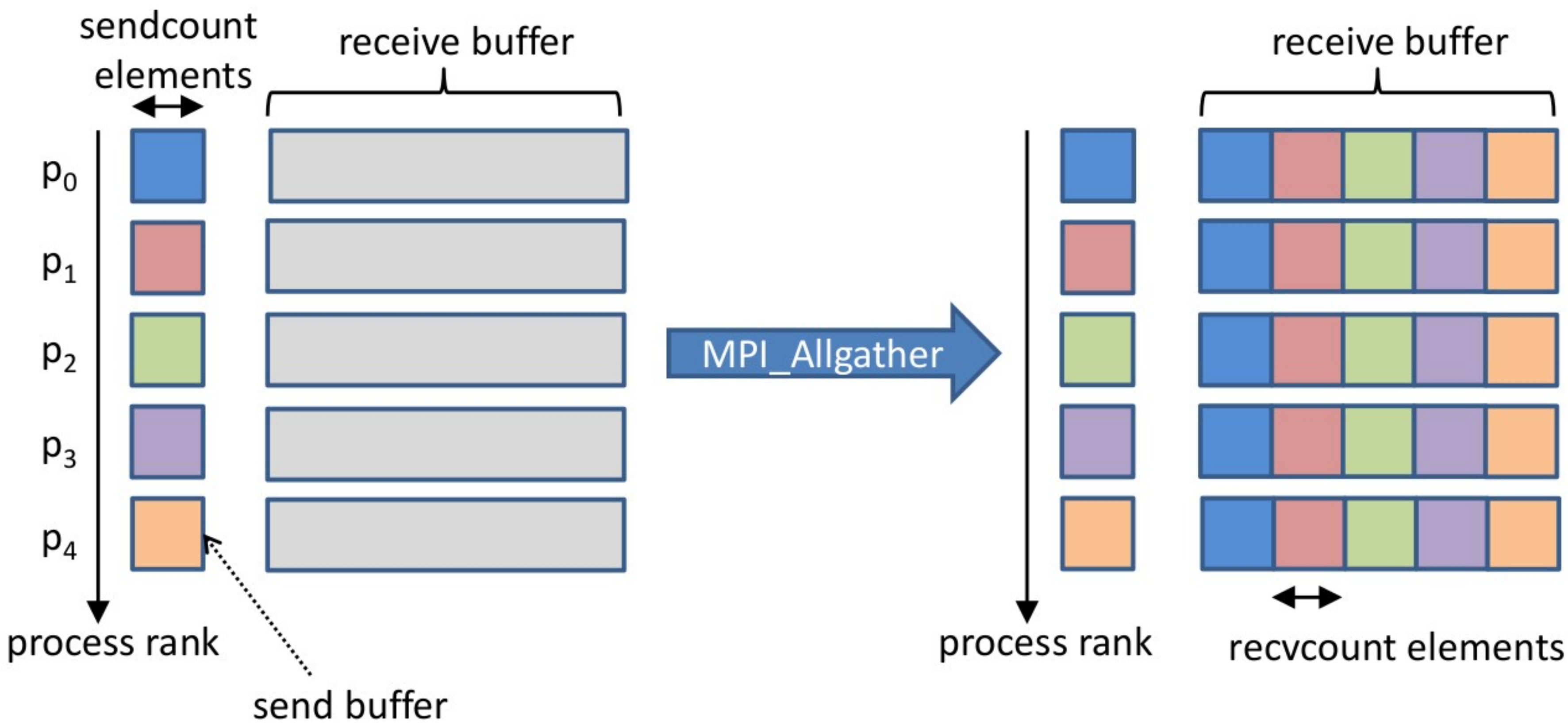
receive parameters are ignored on non-root processes

```
john@doe ~]$ mpirun -np 4 ./gather
Receive buffer at root process:
0 1 2 3
```

Collective Communication: AllGather

```
MPI_Allgather(void *sendbuf, int sendcnt, MPI_Datatype sendType,
              void *recvbuf, int recvcnt, MPI_Datatype recvType,
              MPI_Comm comm)
```

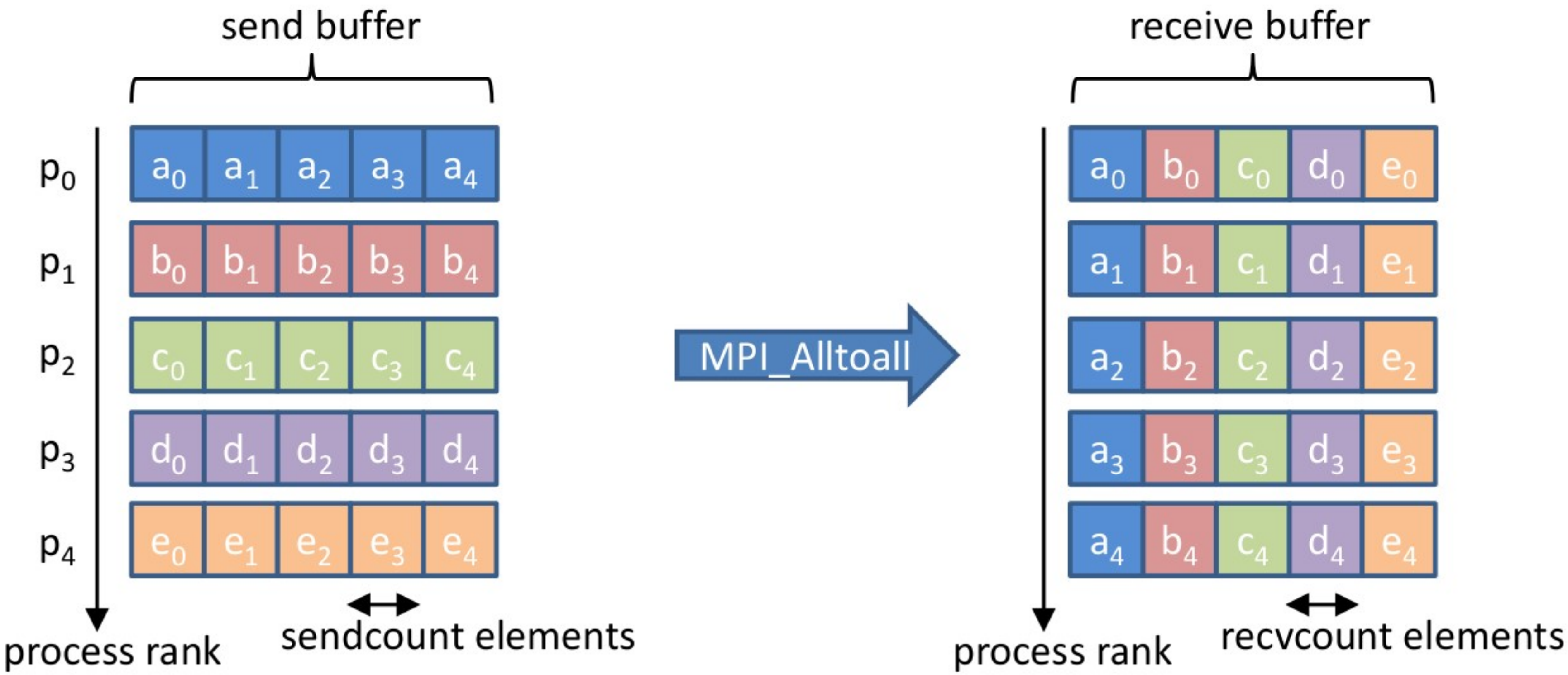
MPI_Allgather is a generalization of MPI_Gather, in that sense that the data is gathered by all processed, instead of just the root process.



Collective Communication: All to All

```
MPI_Alltoall(void *sendbuf, int sendcnt, MPI_Datatype sendType,
             void *recvbuf, int recvcnt, MPI_Datatype recvType,
             int root, MPI_Comm comm)
```

Using MPI_Alltoall, every process sends a distinct message to every other process.

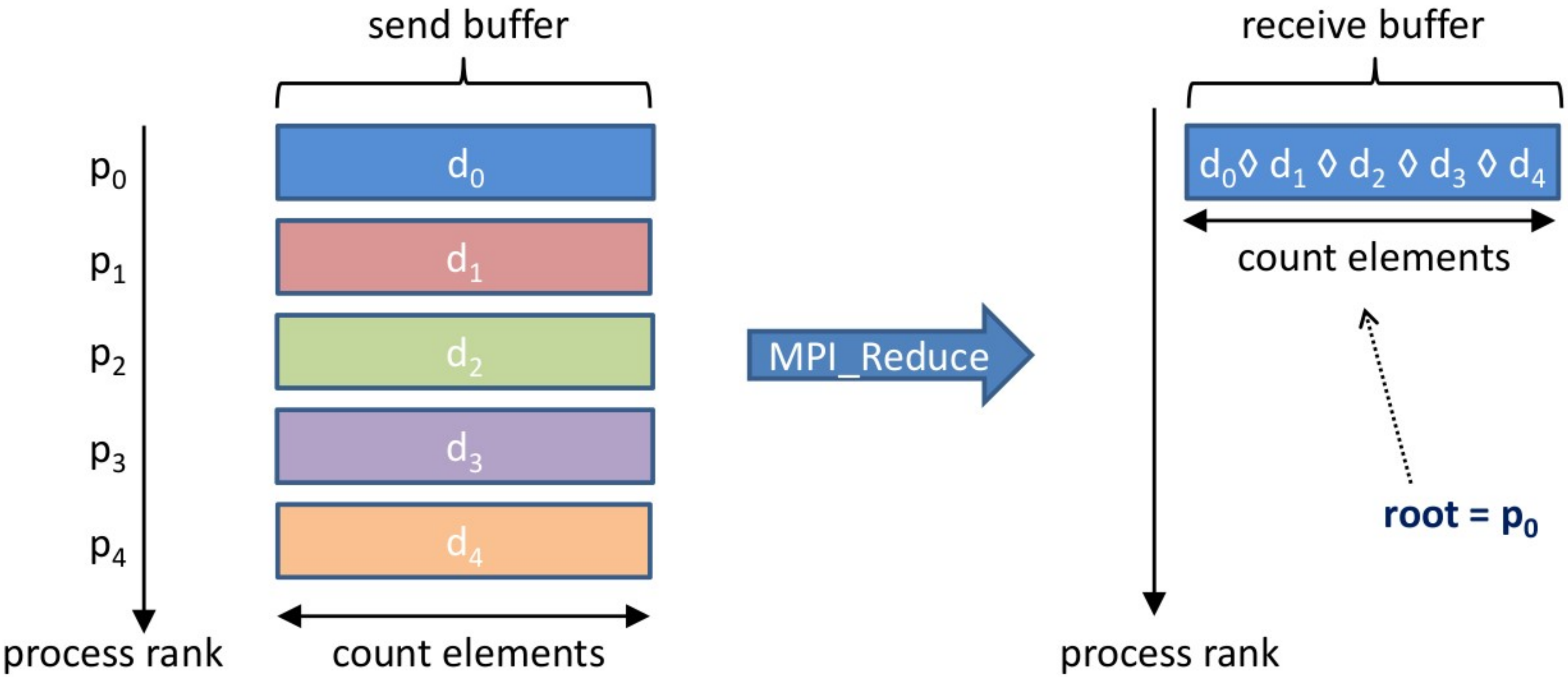


A vector variant, **MPI_Alltoallv**, exists, allowing for different sizes for each process

Collective Communication: Reduce

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

The reduce operation aggregates (“reduces”) scattered data at the root process



◊ = operation, like sum, product, maximum, etc.

Collective Communication: Reduce Operations

Available reduce operations (associative and commutative)
User defined operations are also possible

MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical AND
MPI_BAND	bitwise AND
MPI_LOR	logical OR
MPI_BOR	bitwise OR
MPI_LXOR	logical exclusive OR
MPI_BXOR	bitwise exclusive OR
MPI_MAXLOC	maximum and its location
MPI_MINLOC	maximum and its location