

Introduction to Parallel Computing (CMSC416 / CMSC616)



Advanced MPI

Abhinav Bhatele, Alan Sussman



UNIVERSITY OF
MARYLAND

Any downsides to using blocking calls?

Any downsides to using blocking calls?

- Performance: the receiver blocks for a message and cannot do anything else while waiting for a message

Non-blocking point-to-point calls

- Most important routines: `MPI_Isend` and `MPI_Irecv`
- Two parts to a non-blocking operation:
 - Posting: post the non-blocking operation
 - Completion: wait for its results at a later point in the program — done via calls to `MPI_Wait` or `MPI_Test`
- Can help facilitate overlap of computation with communication

MPI_Irecv: Non-blocking pt2pt receive

```
int MPI_Irecv( const void *buf, int count, MPI_Datatype datatype,  
int dest, int tag, MPI_Comm comm, MPI_Request *request )
```

buf: address of send buffer

count: number of elements in send buffer

datatype: datatype of each send buffer element

dest: rank of destination process

tag: message tag

comm: communicator

request: communication request

MPI_Irecv: Non-blocking pt2pt receive

```
int MPI_Irecv( void *buf, int count, MPI_Datatype datatype, int  
source, int tag, MPI_Comm comm, MPI_Request *request )
```

buf: address of receive buffer

count: maximum number of elements in receive buffer

datatype: datatype of each receive buffer element

source: rank of source process

tag: message tag

comm: communicator

request: communication request

MPI_Wait: blocking call

```
int MPI_Wait( MPI_Request *request, MPI_Status *status )
```

request: communication request

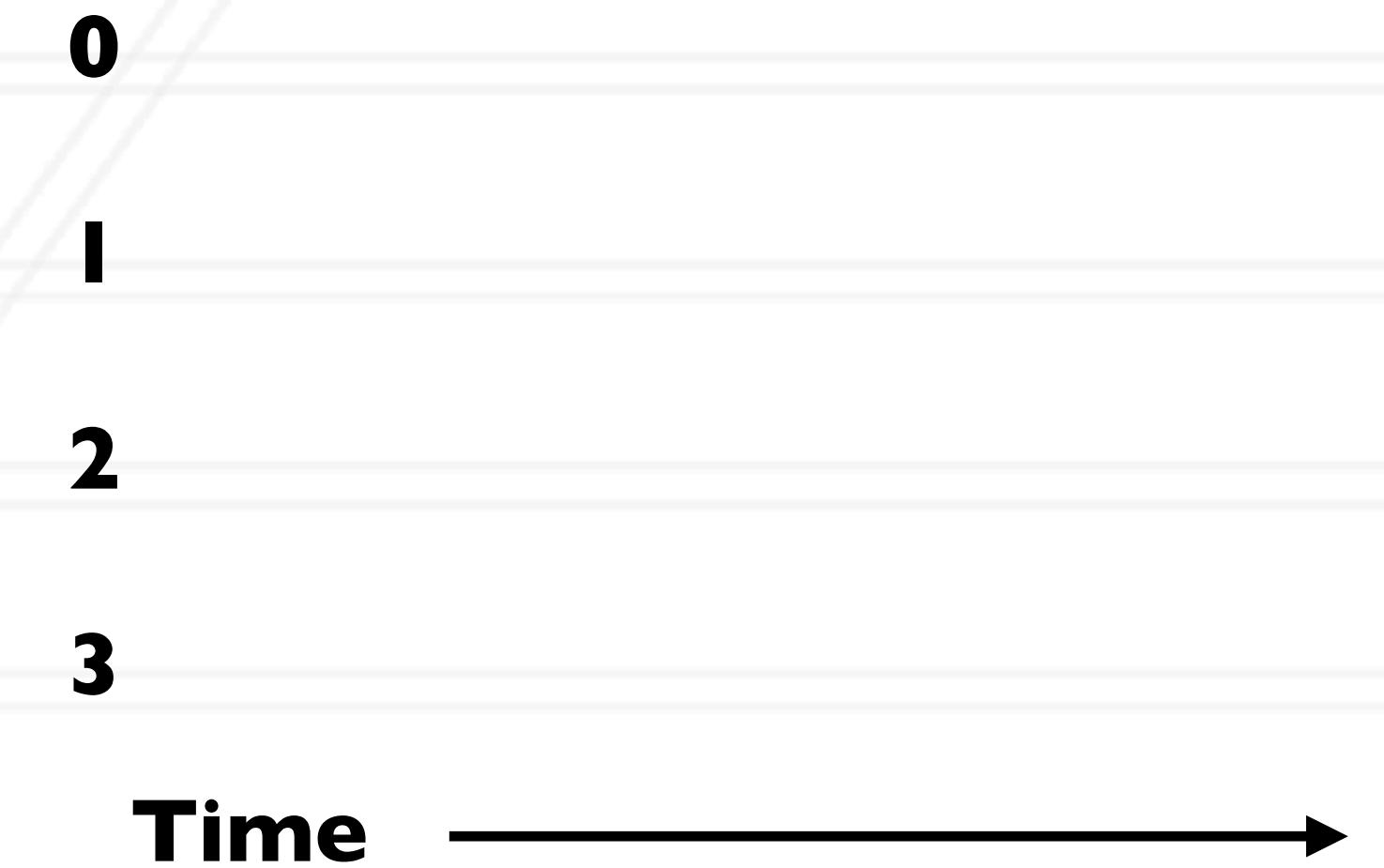
status: status object

- Status object can provide information about:
 - count: number of received entries
 - MPI_SOURCE: source of the message
 - MPI_TAG: tag value of the message
 - MPI_ERROR: error associated with the message
- If you don't want to inspect it, you can use MPI_STATUS_IGNORE

Request object is an opaque object in MPI “system” memory associated with a particular communication operation.
It links the posting to the completion.

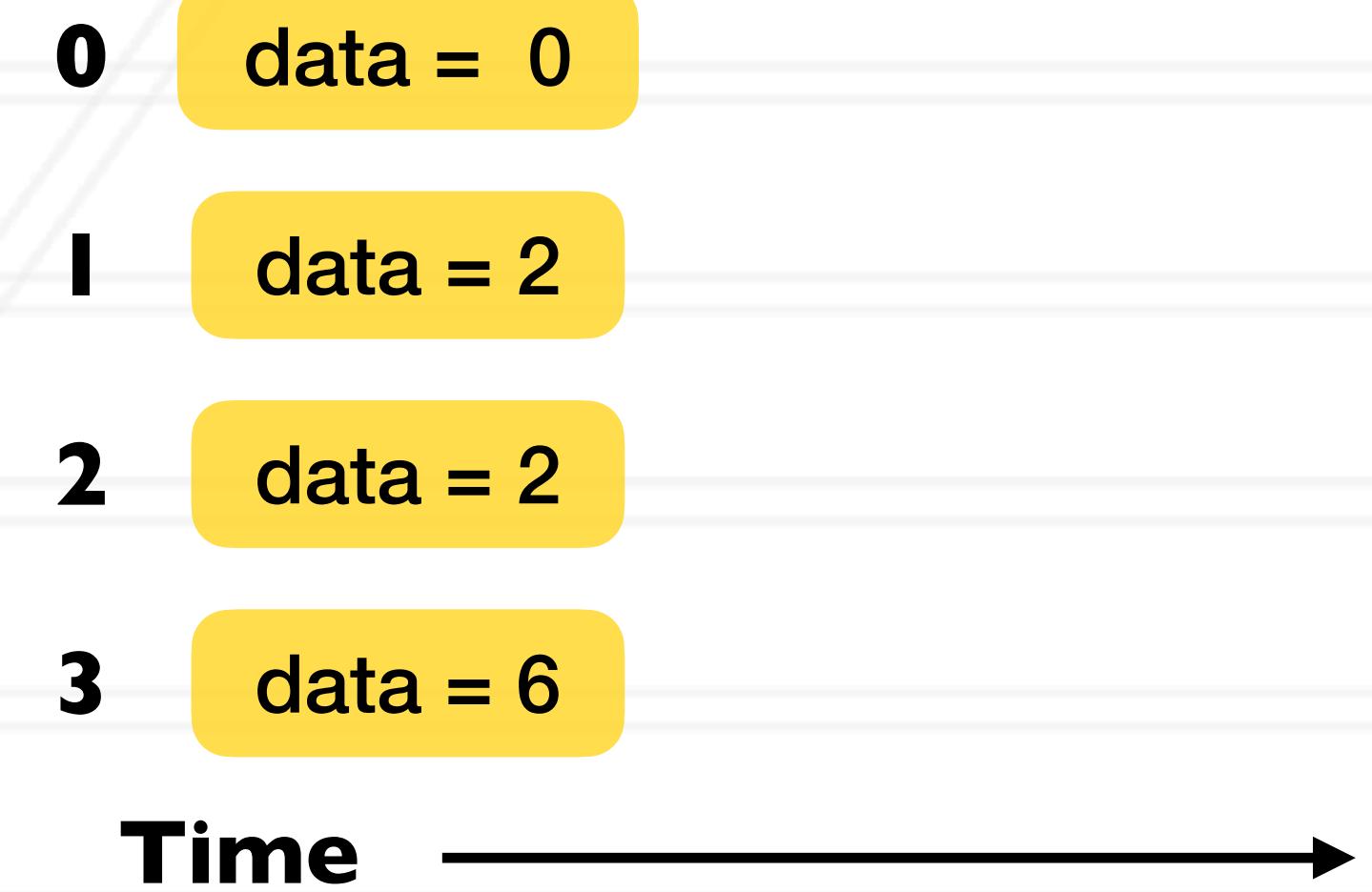
Using non-blocking send/recv

```
int main(int argc, char *argv[ ]) {  
    ...  
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);  
    MPI_Request req;  
    MPI_Status stat;  
    ...  
    if (myrank % 2 == 0) {  
        data = myrank;  
        MPI_Isend(&data, 1, MPI_INT, myrank+1, 0, ..., &req);  
    } else {  
        data = myrank * 2;  
        MPI_Irecv(&data, 1, MPI_INT, myrank-1, 0, ..., &req);  
    }  
    ...  
    MPI_Wait(&req, &stat);  
    printf("Process %d received data %d\n", myrank, data);  
}  
...  
}
```



Using non-blocking send/recv

```
int main(int argc, char *argv[ ]) {  
    ...  
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);  
    MPI_Request req;  
    MPI_Status stat;  
    ...  
    if (myrank % 2 == 0) {  
        data = myrank;  
        MPI_Isend(&data, 1, MPI_INT, myrank+1, 0, ..., &req);  
    } else {  
        data = myrank * 2;  
        MPI_Irecv(&data, 1, MPI_INT, myrank-1, 0, ..., &req);  
    }  
    ...  
    MPI_Wait(&req, &stat);  
    printf("Process %d received data %d\n", myrank, data);  
}  
...  
}
```



Using non-blocking send/recv

```
int main(int argc, char *argv[ ]) {  
    ...  
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);  
    MPI_Request req;  
    MPI_Status stat;  
    ...  
    if (myrank % 2 == 0) {  
        data = myrank;  
        MPI_Isend(&data, 1, MPI_INT, myrank+1, 0, ..., &req);  
    } else {  
        data = myrank * 2;  
        MPI_Irecv(&data, 1, MPI_INT, myrank-1, 0, ..., &req);  
    }  
    ...  
    MPI_Wait(&req, &stat);  
    printf("Process %d received data %d\n", myrank, data);  
}
```



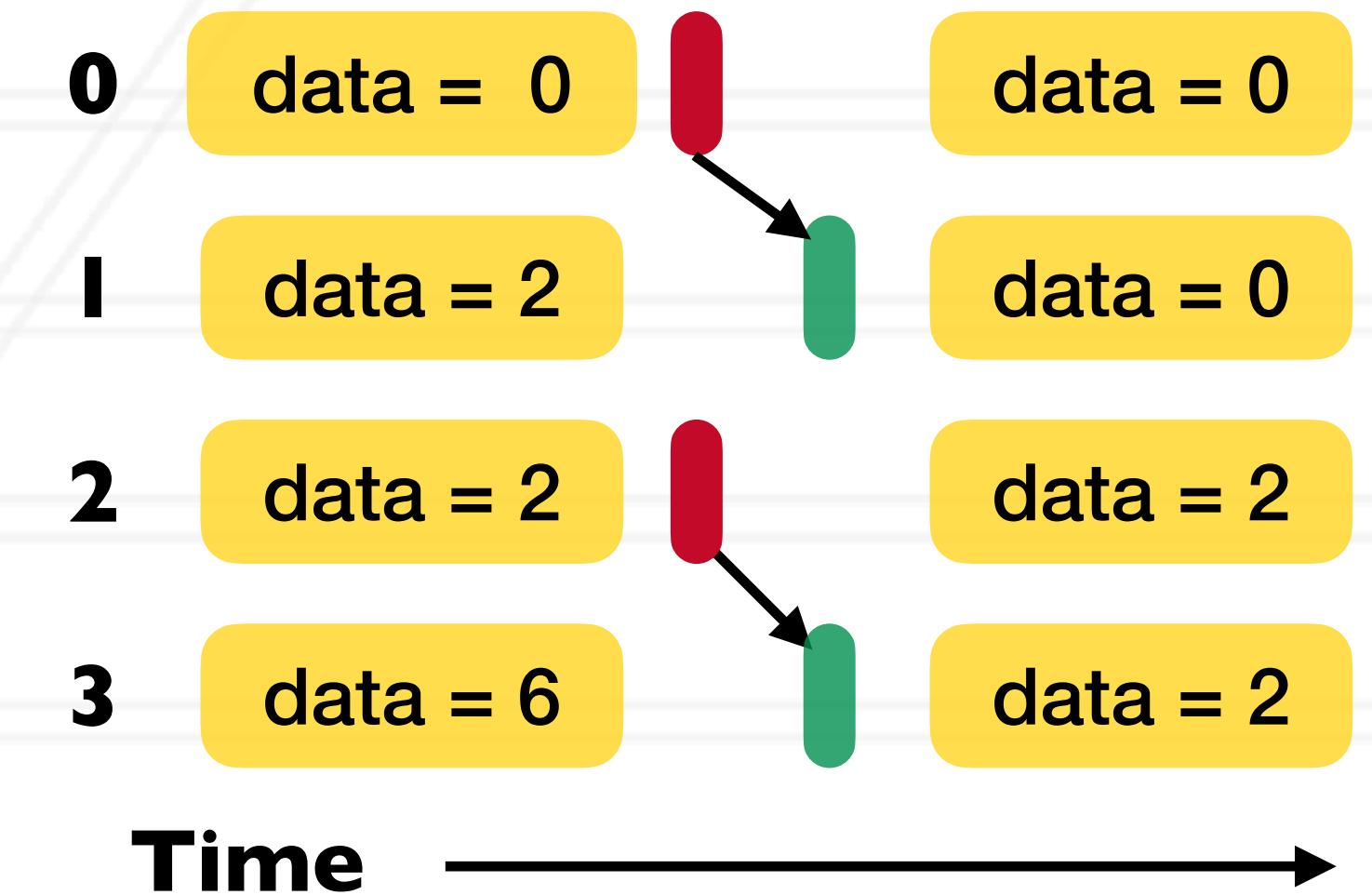
Using non-blocking send/recv

```
int main(int argc, char *argv[ ]) {  
    ...  
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);  
    MPI_Request req;  
    MPI_Status stat;  
    ...  
    if (myrank % 2 == 0) {  
        data = myrank;  
        MPI_Isend(&data, 1, MPI_INT, myrank+1, 0, ..., &req);  
    } else {  
        data = myrank * 2;  
        MPI_Irecv(&data, 1, MPI_INT, myrank-1, 0, ..., &req);  
    }  
    ...  
    MPI_Wait(&req, &stat);  
    printf("Process %d received data %d\n", myrank, data);  
}  
...  
}
```

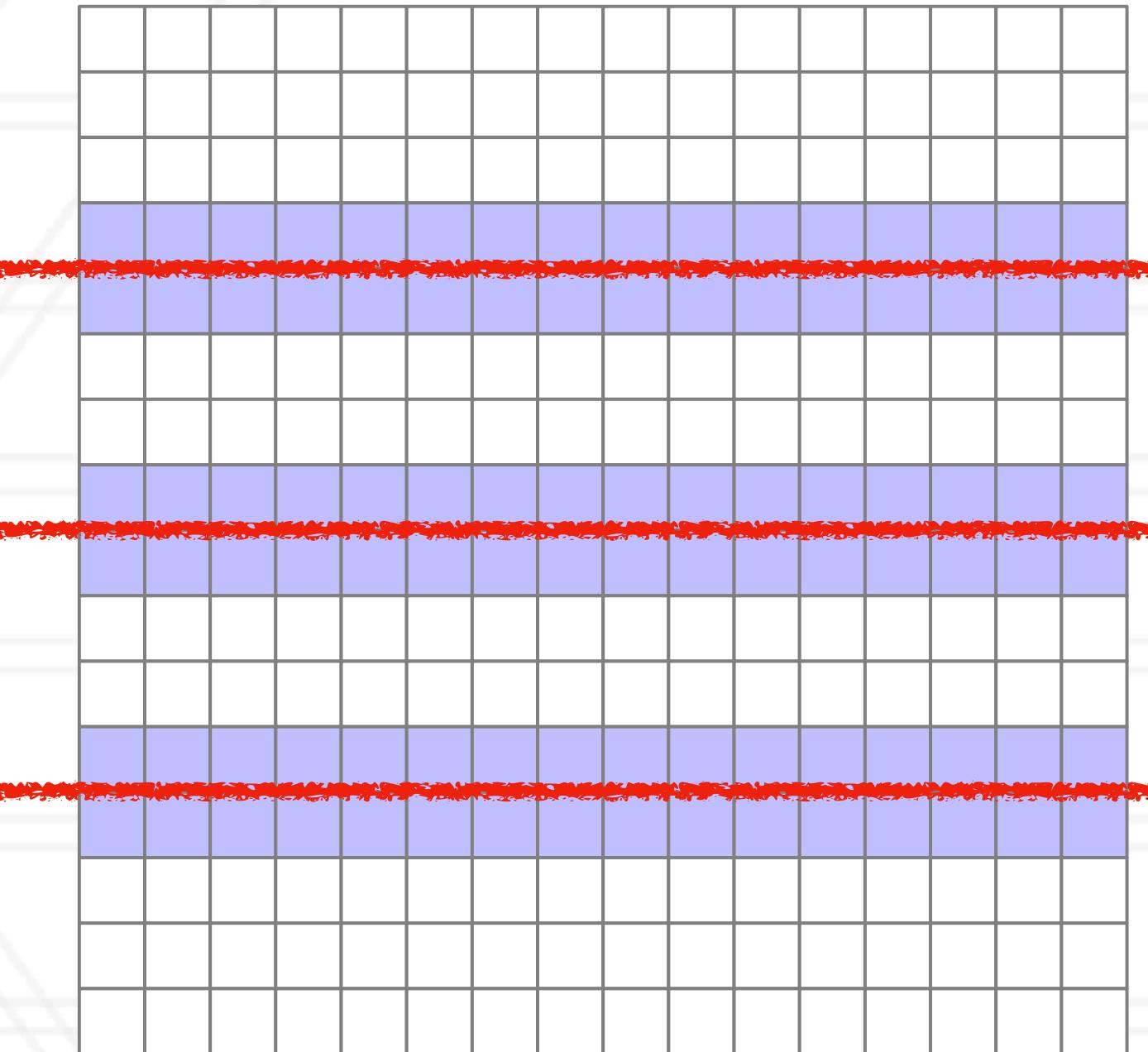


Using non-blocking send/recv

```
int main(int argc, char *argv[ ]) {  
    ...  
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);  
    MPI_Request req;  
    MPI_Status stat;  
    ...  
    if (myrank % 2 == 0) {  
        data = myrank;  
        MPI_Isend(&data, 1, MPI_INT, myrank+1, 0, ..., &req);  
    } else {  
        data = myrank * 2;  
        MPI_Irecv(&data, 1, MPI_INT, myrank-1, 0, ..., &req);  
    }  
    ...  
    MPI_Wait(&req, &stat);  
    printf("Process %d received data %d\n", myrank, data);  
}
```

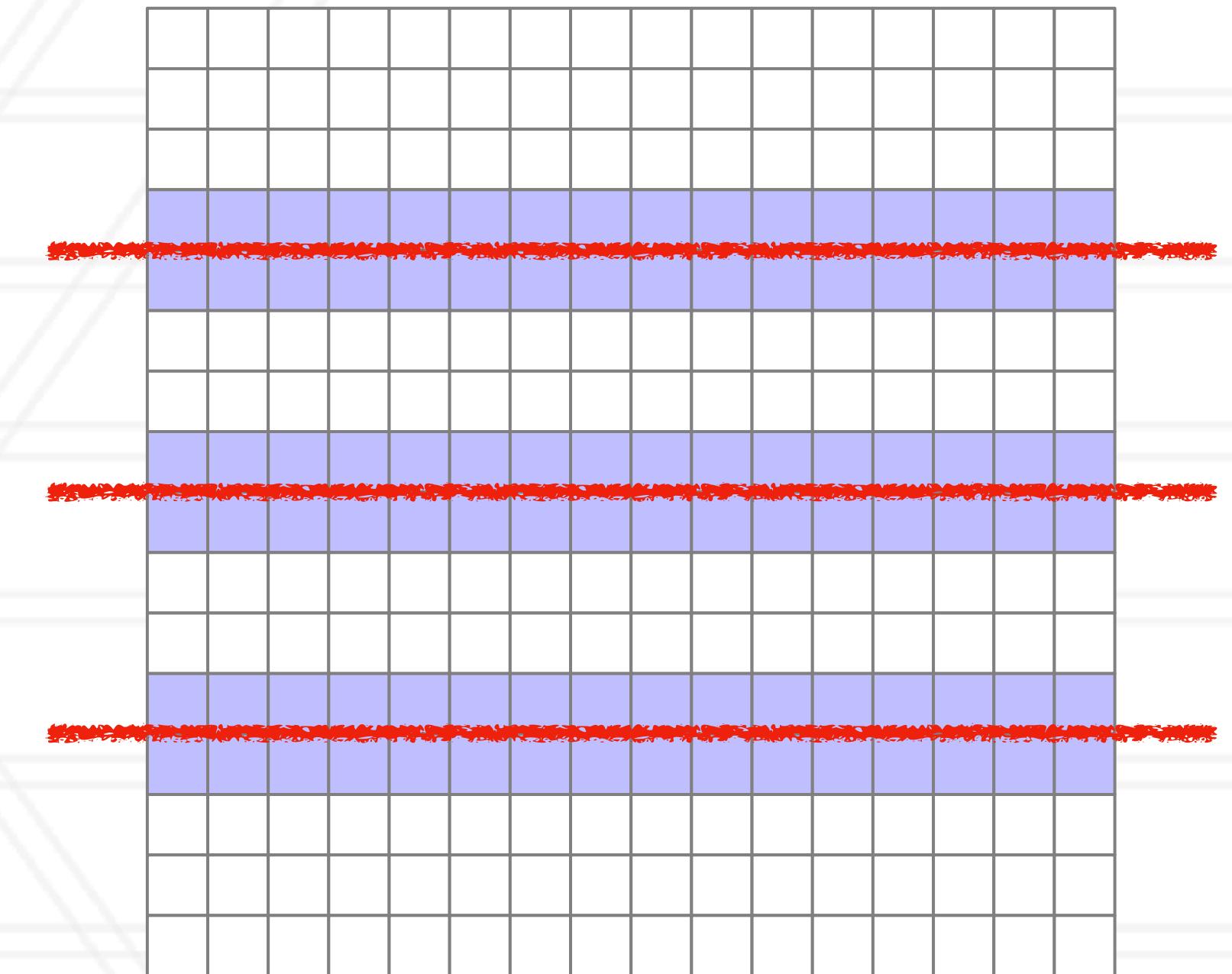


2D stencil computation



2D stencil computation

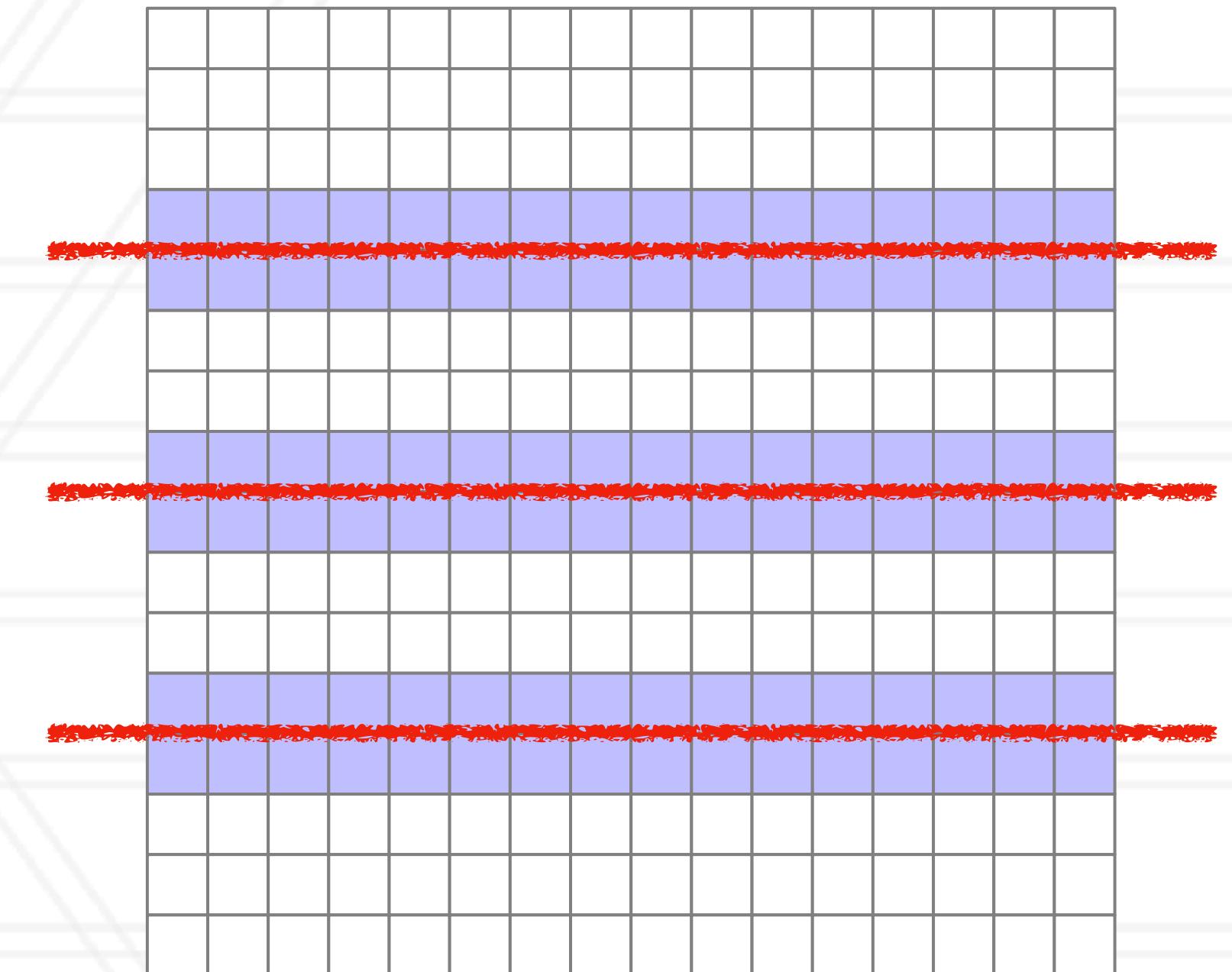
```
int main(int argc, char *argv) {  
    ...  
  
    for(int t=0; t<num_steps; t++) {  
  
        MPI_Irecv(&data1, 16, MPI_DOUBLE, (myrank-1)%4, 0, ...);  
        MPI_Irecv(&data2, 16, MPI_DOUBLE, (myrank+1)%4, 0, ...);  
  
        MPI_Isend(&data3, 16, MPI_DOUBLE, (myrank-1)%4, 0, ...);  
        MPI_Isend(&data4, 16, MPI_DOUBLE, (myrank+1)%4, 0, ...);  
  
        MPI_Waitall(...);  
  
        compute();  
    }  
    ...  
}
```



2D stencil computation

```
int main(int argc, char *argv) {  
    ...  
  
    for(int t=0; t<num_steps; t++) {  
  
        MPI_Irecv(&data1, 16, MPI_DOUBLE, (myrank-1)%4, 0, ...);  
        MPI_Irecv(&data2, 16, MPI_DOUBLE, (myrank+1)%4, 0, ...);  
  
        MPI_Isend(&data3, 16, MPI_DOUBLE, (myrank-1)%4, 0, ...);  
        MPI_Isend(&data4, 16, MPI_DOUBLE, (myrank+1)%4, 0, ...);  
  
        MPI_Waitall(...);  
  
        compute(); // copy contents of A_new into A  
    }  
    ...  
}
```

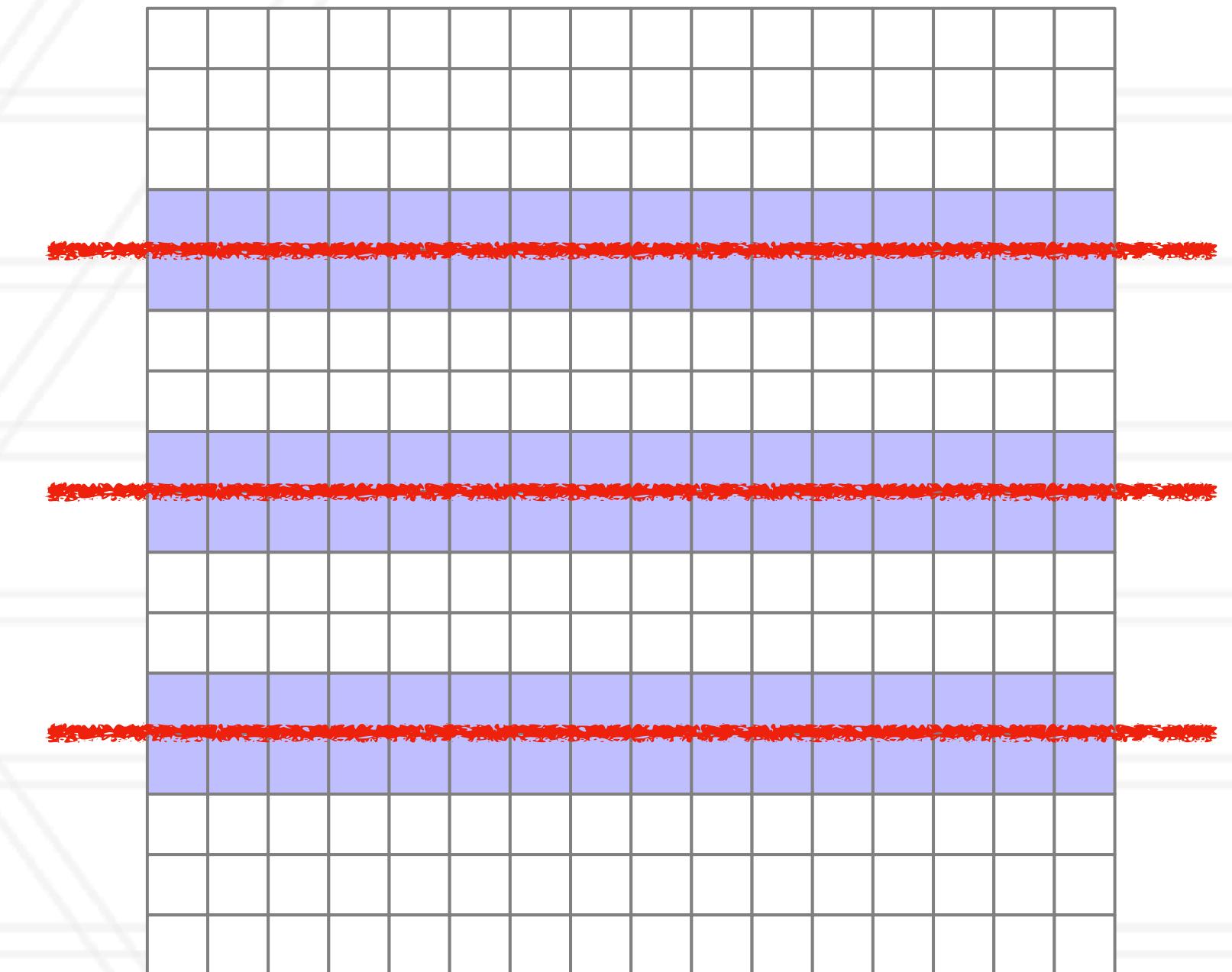
```
// copy contents of A_new into A  
  
for(i ...)   
    for(j ...)   
        A_new[i, j] = (A[i, j] + A[i-1, j] + ...
```



2D stencil computation

```
int main(int argc, char *argv) {  
    ...  
  
    for(int t=0; t<num_steps; t++) {  
  
        MPI_Irecv(&data1, 16, MPI_DOUBLE, (myrank-1)%4, 0, ...);  
        MPI_Irecv(&data2, 16, MPI_DOUBLE, (myrank+1)%4, 0, ...);  
  
        MPI_Isend(&data3, 16, MPI_DOUBLE, (myrank-1)%4, 0, ...);  
        MPI_Isend(&data4, 16, MPI_DOUBLE, (myrank+1)%4, 0, ...);  
  
        MPI_Waitall(...);  
  
        compute(); // copy contents of A_new into A  
    }  
    ...  
}
```

Wraparound



Other MPI calls

- `int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)`
 - `flag` is set to true if the operation has completed
- `int MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status *array_of_statuses[])`
 - Waits for all given MPI requests to complete
- `MPI_Waitany`
 - Waits for any specified MPI request to complete

Collective operations

- All processes in the communicator participate in the operation

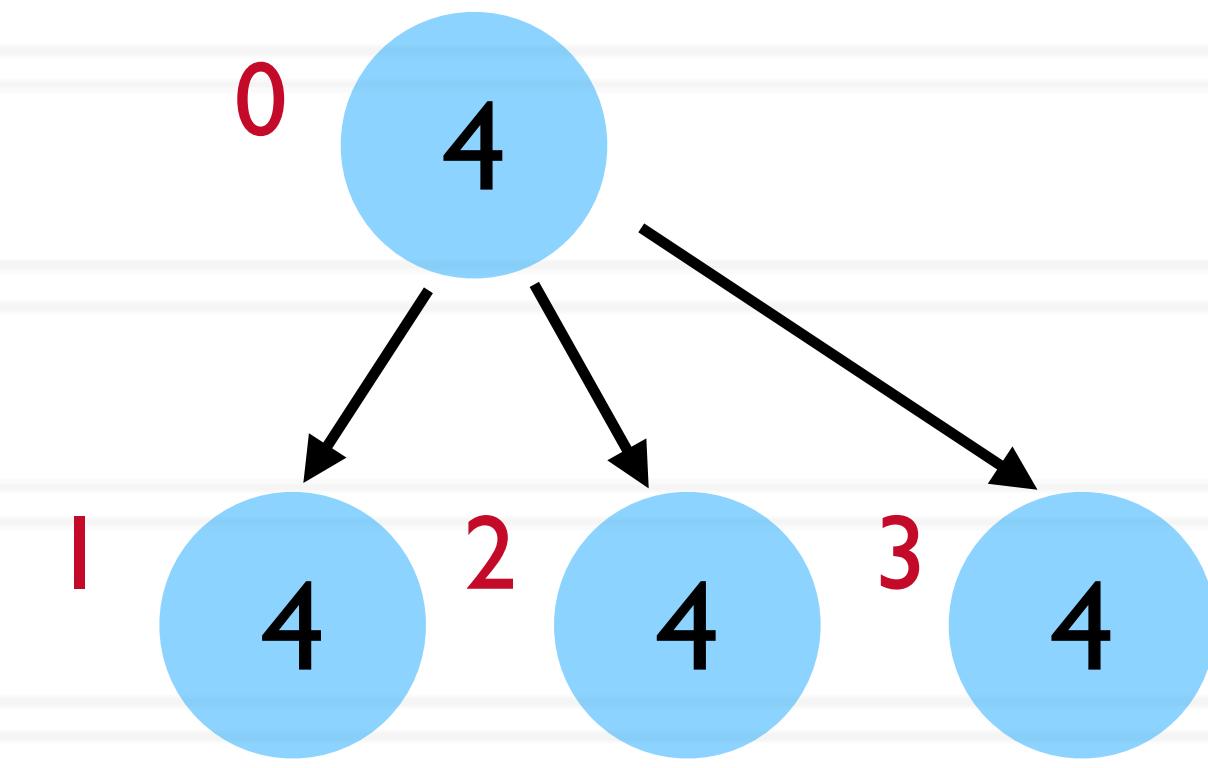
Collective operations

Collective operations

- `int MPI_Barrier(MPI_Comm comm)`
 - Blocks until all processes in the communicator have reached this routine

Collective operations

- `int MPI_Barrier(MPI_Comm comm)`
 - Blocks until all processes in the communicator have reached this routine
- `int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`
 - Send data from root to all processes of the communicator
 - Buffer represents what is being sent on root but where things will be written on other processes



Announcements

- Assignment 2 sneak peek
 - Official post date: Sept 25, Due on: Oct 9 11:59 pm
- Expect Quiz I to be posted next week

Collective operations

Collective operations

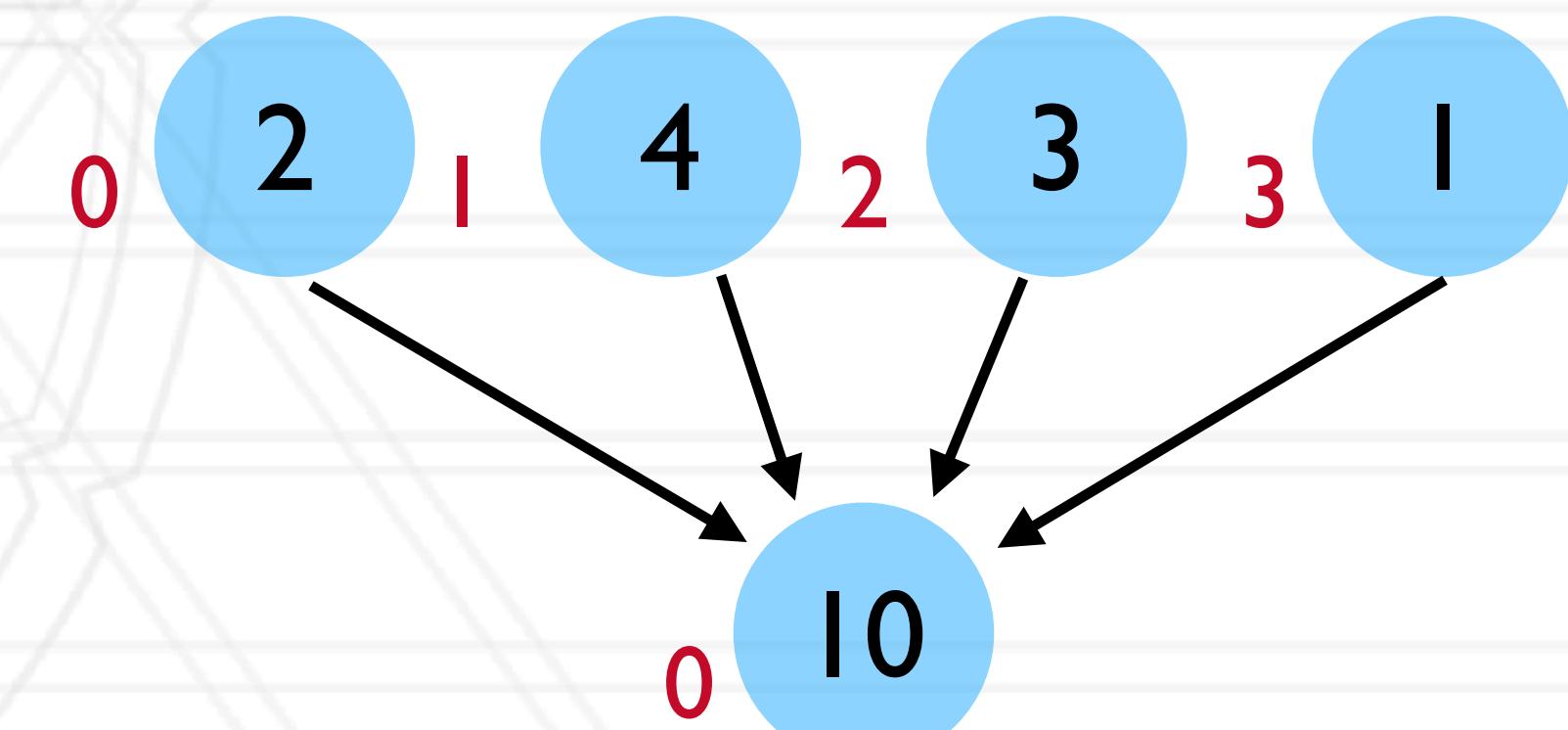
- ```
int MPI_Reduce(const void *sendbuf, void
 *recvbuf, int count, MPI_Datatype
 datatype, MPI_Op op, int root, MPI_Comm
 comm)
```
- Reduce data from all processes to the root
- sendbuf should be valid on all processes
- Recvbuf only needs to exist on root

# Collective operations

---

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

 - Reduce data from all processes to the root
 - sendbuf should be valid on all processes
 - Recvbuf only needs to exist on root
- **MPI_Allreduce**
 - Can be used to send the result back to **all** processes

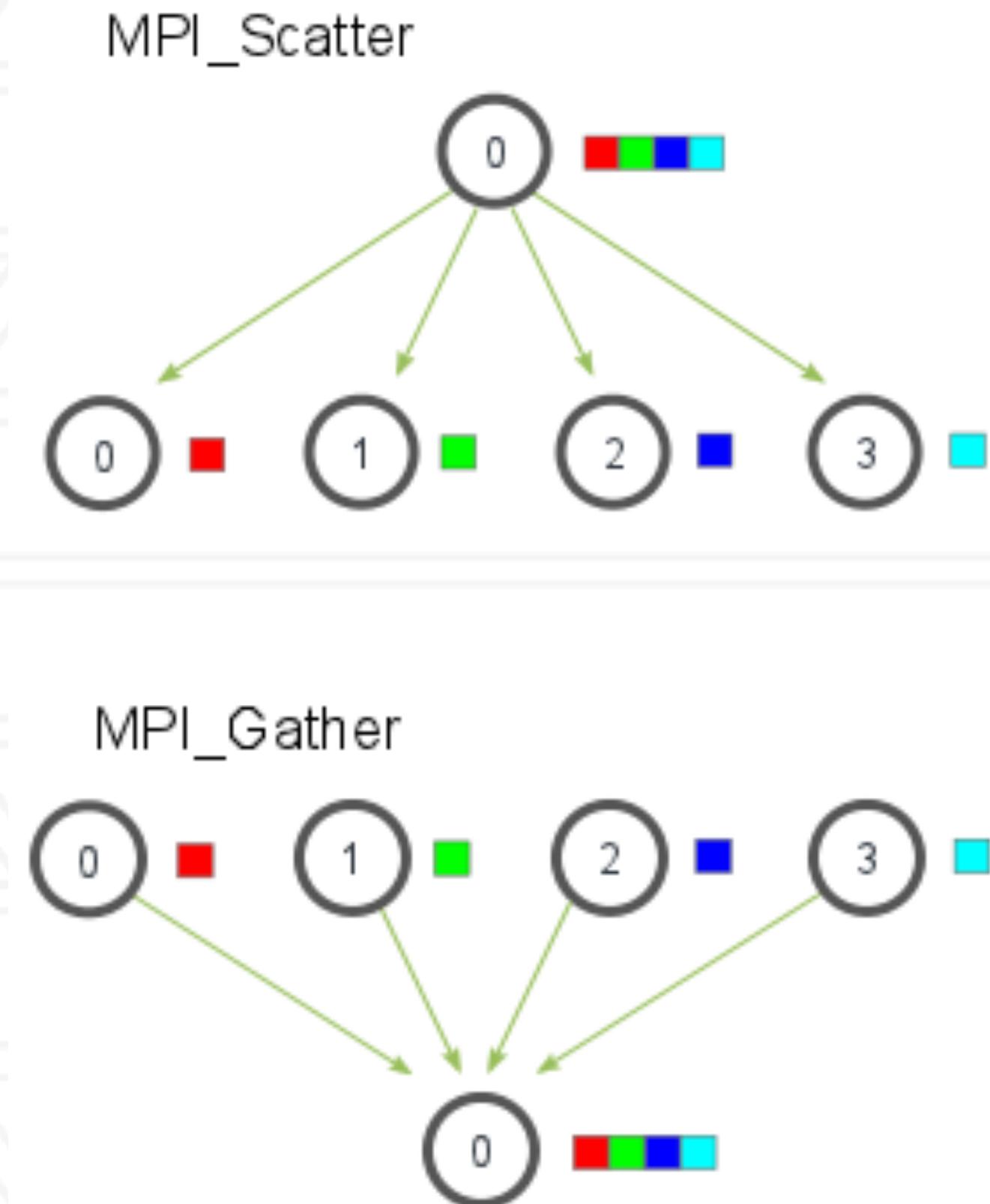


Collective operations

- ```
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

  - Send distinct data from root to all processes
- ```
int MPI_Gather( const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

 - Gather distinct data from all processes to the root
- **MPI_Scan:** Computes prefix sum



<https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/>

Other MPI calls

- double MPI_Wtime (void)
 - Returns elapsed time in seconds since an arbitrary time in the past

```
{  
    double starttime, endtime;  
    starttime = MPI_Wtime();  
  
    .... code region to be timed ....  
  
    endtime = MPI_Wtime();  
    printf("Time %f seconds\n", endtime-starttime);  
}
```

Calculate the value of

$$\pi = \int_0^1 \frac{4}{1+x^2}$$

```
int main(int argc, char *argv[ ])
{
    ...
    n = 10000;
    h = 1.0 / (double) n;
    sum = 0.0;

    for (i = 1; i <= n; i += 1) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x * x));
    }
    pi = h * sum;

    ...
}
```

Calculate the value of

$$\pi = \int_0^1 \frac{4}{1+x^2}$$

```
int main(int argc, char *argv[])
{
    ...

    n = 10000;
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    h = 1.0 / (double) n;
    sum = 0.0;

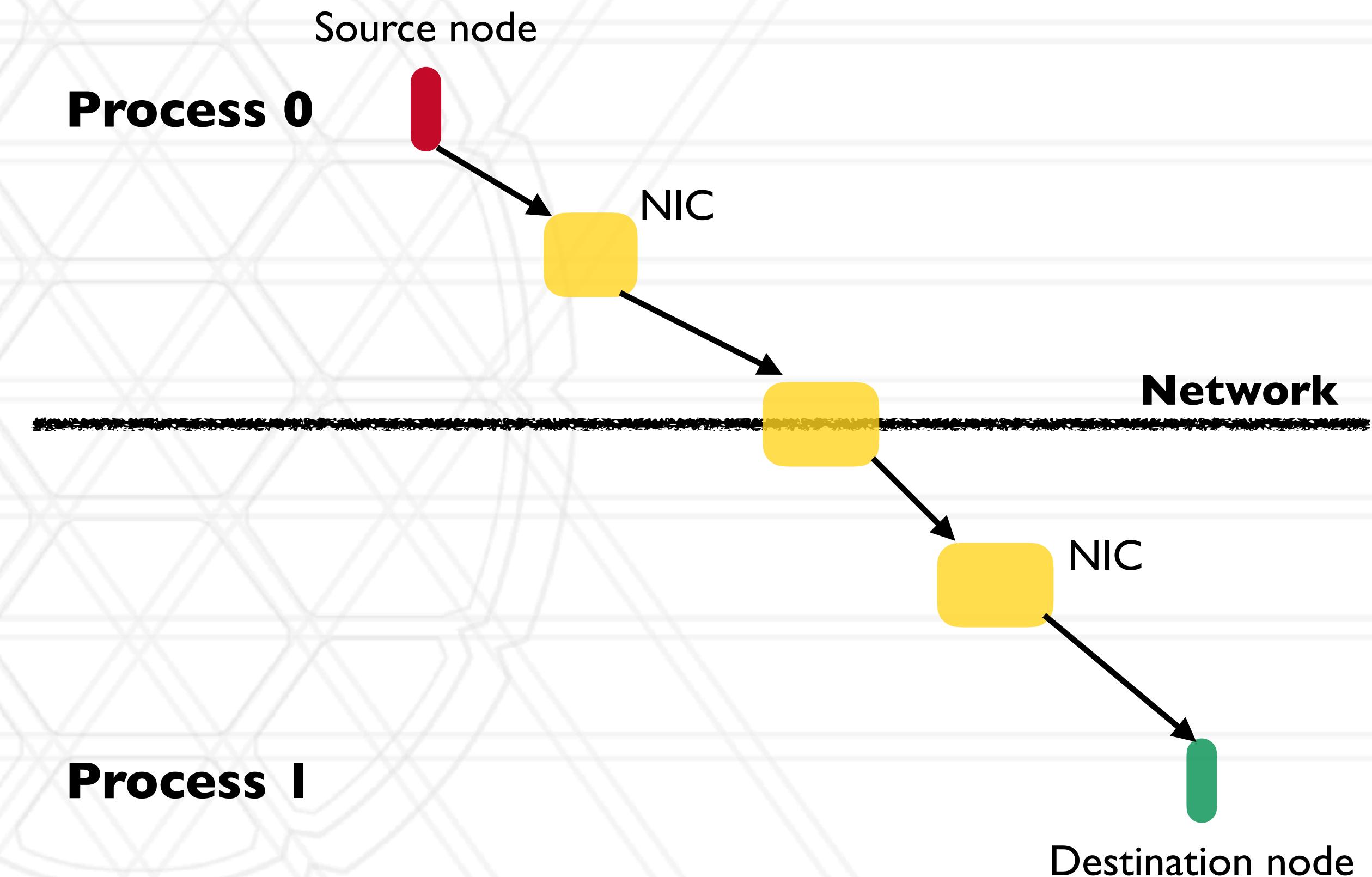
    for (i = myrank + 1; i <= n; i += numpes) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x * x));
    }
    pi = h * sum;

    MPI_Reduce(&pi, &globalpi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    ...
}
```

Protocols for sending message

- Eager
 - Message sent assuming destination can store
- Rendezvous
 - Message only sent after handshake (receiving ack) with destination
- Short
 - Data sent with the message envelope



Other MPI send modes

- Basic mode: `MPI_Send`
- Buffered mode: `MPI_Bsend` — returns immediately and send buffer can be reused
 - Use `MPI_Buffer_attach` to provide space for buffering
- Synchronous mode: `MPI_Ssend` — will not return until matching receive is posted
- Ready mode: `MPI_Rsend` — may be used ONLY if matching receive is already posted
- Non-blocking versions of all of the above also exist

<https://www.mcs.anl.gov/research/projects/mpi/sendmode.html>



UNIVERSITY OF
MARYLAND