Introduction to MPI

Part2. *Intermediate Topics* （A）

人工智能学院
缪青海
miaoqh@ucas.ac.cn

Content

- **Non-blocking communication**
- Group (collective) communication
- MPI Datatypes
- Virtual Topology

Mpi buffer

- MPI定义三种缓冲区buffer:
  - 应用缓冲区a_buffer
    - 将要发送/接收数据的地址空间，在消息格式中定义;
  - 系统缓冲区s_buffer
    - MPI环境为通信所准备的存储空间，a_buffer在此出入复制;
  - 用户定义缓冲区u_buffer
    - 用户在使用某些api时在程序中显式申请的用于通信的空间，注册到MPI环境中;

Blocking Communication

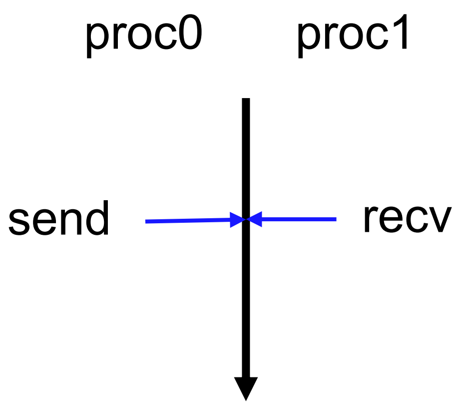
- In blocking communication:
  - **MPI_SEND** does not return until buffer is empty (available for reuse)
  - **MPI_RECV** does not return until buffer is full (available for use)

- A process sending data will be <u>blocked</u> until data in the send buffer is emptied.
- A process receiving data will be <u>blocked</u> until the receive buffer is filled.

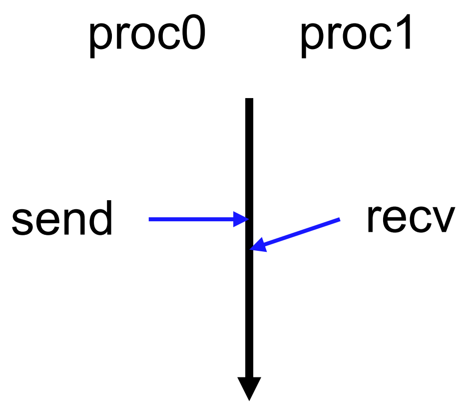
- Exact completion semantics of communication generally depends on the message size and the system buffer size.

Blocking Communication

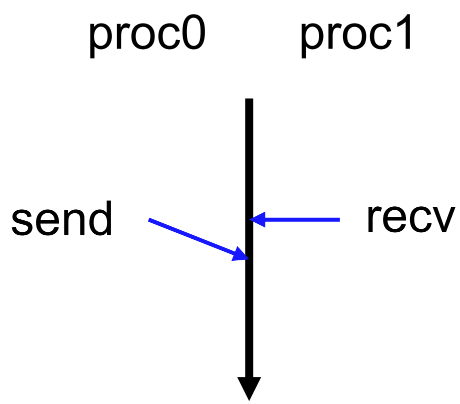
标准阻塞通信的状态：



立即发送



消息小于系统缓冲区：
缓存并返回
消息大于系统缓冲区：
等待接收



立即发送

Blocking Communication

阻塞通信的类型：

类型	发送	接收	注释
标准模式	MPI_Send	MPI_Recv MPI_Irecv MPI_Recv_Init	
缓冲模式	MPI_Bsend		wait
就绪模式	MPI_Rsend		test
同步模式	MPI_Ssend		

Blocking Communication

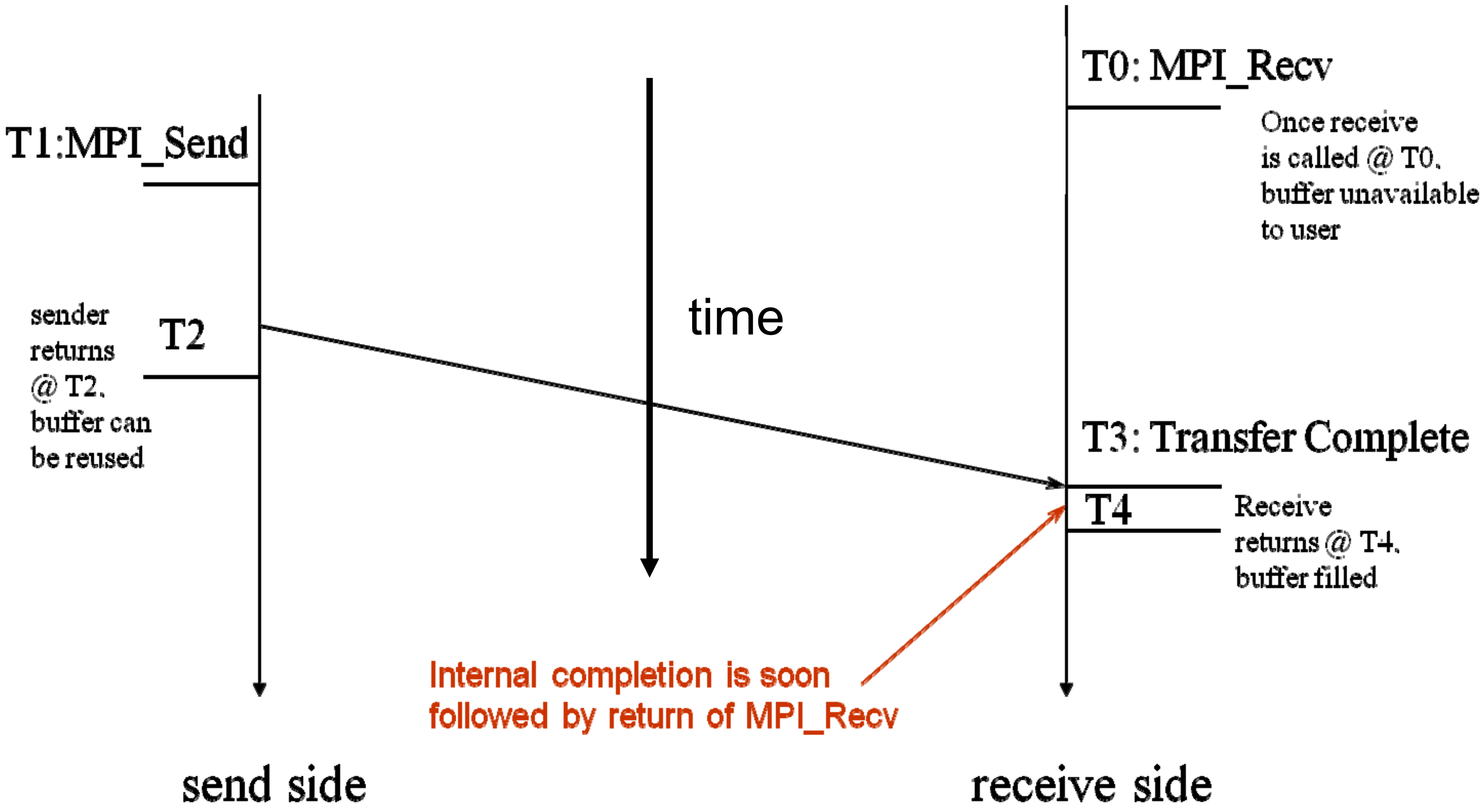
- In blocking communication.
  - **`MPI_SEND`** does not return until buffer is empty (available for reuse)
  - **`MPI_RECV`** does not return until buffer is full (available for use)

- Blocking communication is simple to use but can be prone to ***deadlocks***

Usually deadlocks →

```
if (rank == 0) {
        MPI_SEND(..to rank 1..)
        MPI_RECV(..from rank 1..)
else if (rank == 1) {
        MPI_SEND(..to rank 0..)      ← reverse send/recv
        MPI_RECV(..from rank 0..)
}
```

Blocking Send-Receive Diagram



Blocking vs. Non-blocking

- **MPI_SEND/MPI_RECV** are blocking communication calls
  - Return of the routine implies completion
  - When these calls return the memory locations used in the message transfer can be safely accessed for reuse
  - For "send" completion implies variable sent can be reused/modified
  - Modifications will not affect data intended for the receiver
  - For "receive" variable received can be read
- **MPI_ISEND/MPI_IRECV** are non-blocking variants
  - Routine returns immediately
    - – completion has to be separately tested for
  - These are primarily used to overlap computation and communication to improve performance

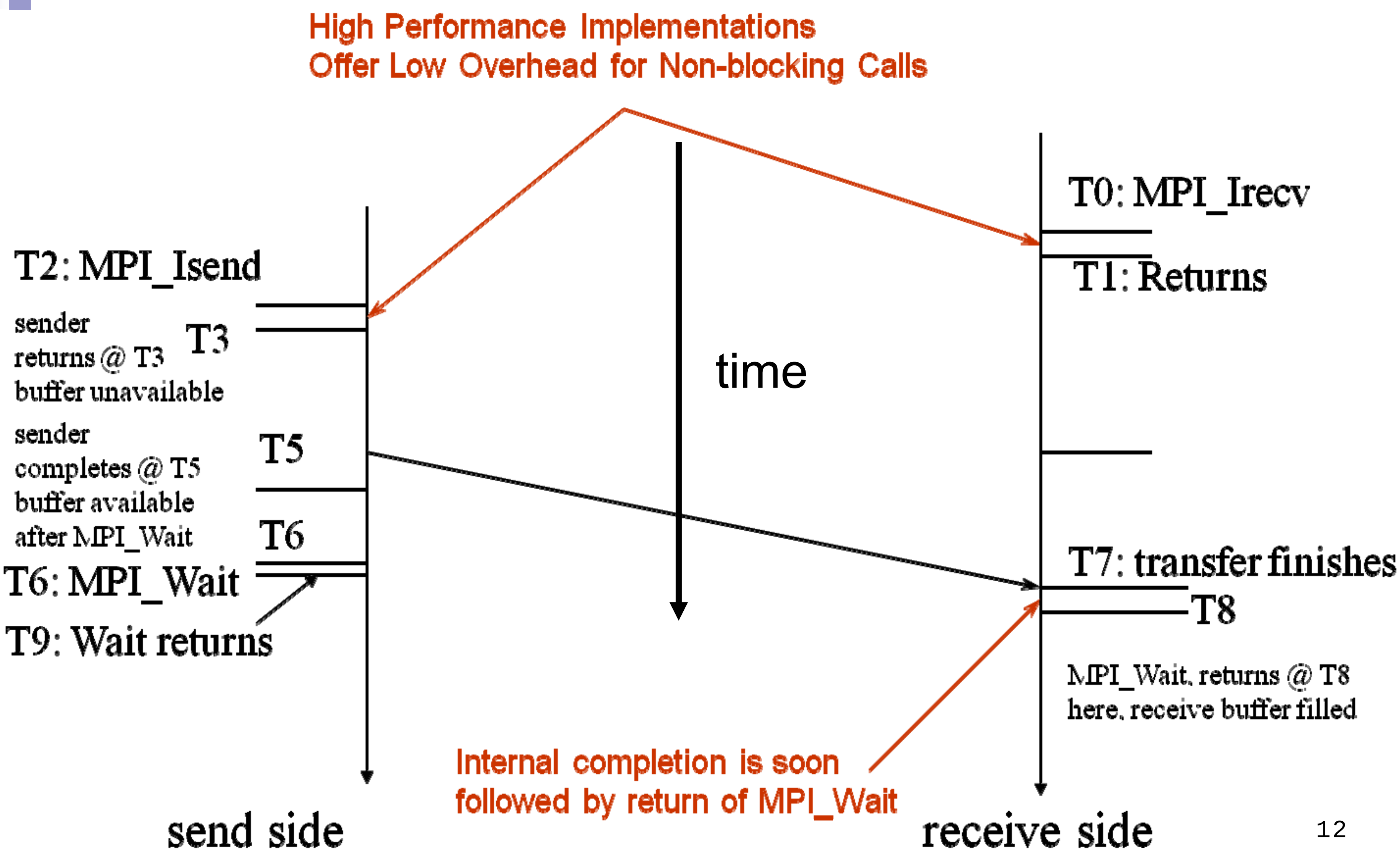
Non-Blocking Communication

- Non-blocking (asynchronous) operations return (immediately) ‘‘request handles” that can be waited on and queried
  - **`MPI_ISEND(buf, count, datatype, dest, tag, comm, request)`**
  - **`MPI_IRECV(buf, count, datatype, src, tag, comm, request)`**
  - **`MPI_WAIT(request, status)`**
- Anywhere you use **MPI_SEND** or **MPI_RECV**, you can use the pair of **MPI_ISEND/MPI_WAIT** or **MPI_IRECV/MPI_WAIT**
- One can also test without waiting using **MPI_TEST**
  - **MPI_TEST(request, flag, status)**

Multiple Completions

- It is sometimes desirable to wait on multiple requests:
  - `MPI_Waitall(count, array_of_requests, array_of_statuses)`
  - `MPI_Waitany(count, array_of_requests, &index, &status)`
  - `MPI_Waitsome(incount, array_of_requests, outcount,`
    `array_of_indices, array_of_statuses)`
- There are corresponding versions of **TEST** for each of these

Non-Blocking Send-Receive Diagram



Message Completion and Buffering

- A send has completed when the user supplied buffer can be reused

```
*buf =3;
MPI_Send(buf, 1, MPI_INT …)
*buf = 4; /* OK, receiver will
always receive 3 */
```

```
*buf =3;
MPI_Isend(buf, 1, MPI_INT …)
*buf = 4; /*Not certain if receiver
gets 3 or 4 or anything else */
MPI_Wait(…);
```

- Just because the send completes does not mean that the receive has completed
  - Message may be buffered by the system
  - Message may still be in transit

A Non-Blocking example

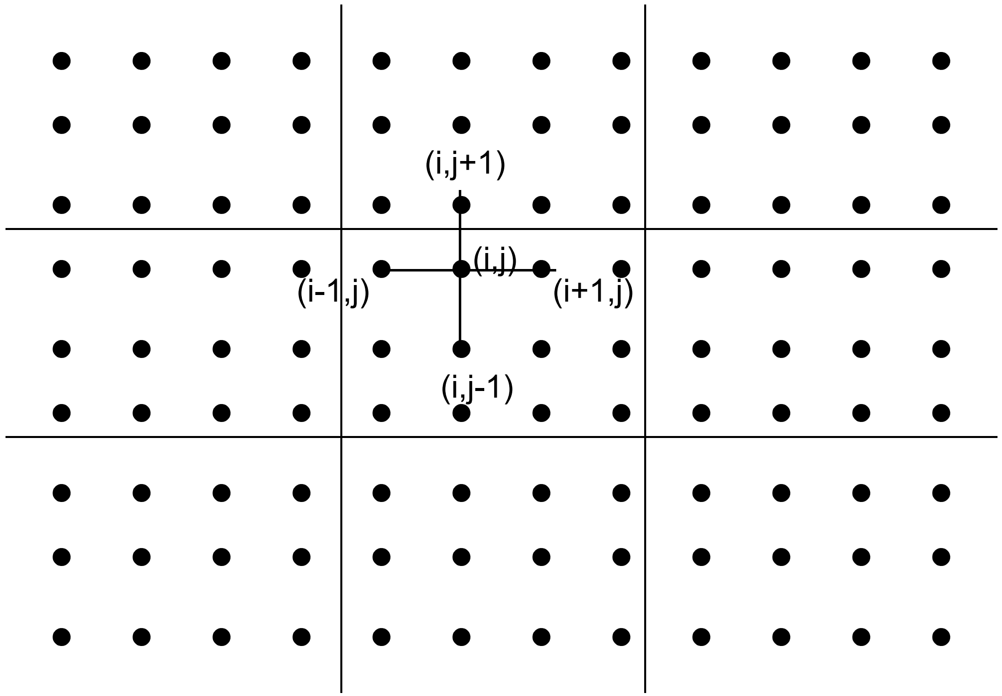
```
int main(int argc, char ** argv)
{
    [...snip...]
    if (rank == 0) {
        for (i=0; i< 100; i++) {
            /* Compute each data element and send it out */
            data[i] = compute(i);
            MPI_Isend(&data[i], 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
                      &request[i]);
        }
        MPI_Waitall(100, request, MPI_STATUSES_IGNORE)
    }
    else {
        for (i = 0; i < 100; i++)
            MPI_Recv(&data[i], 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                     MPI_STATUS_IGNORE);
    }
    [...snip...]
}
```

Code Example - 2

- *code/test2/mpi_helloNBsend.c*
- *code/test2/mpi_send_recv_nonblocking.c*

  - *mpicc mpi_xxx.c –o test*
  - *mpiexec –n 2 ./test*

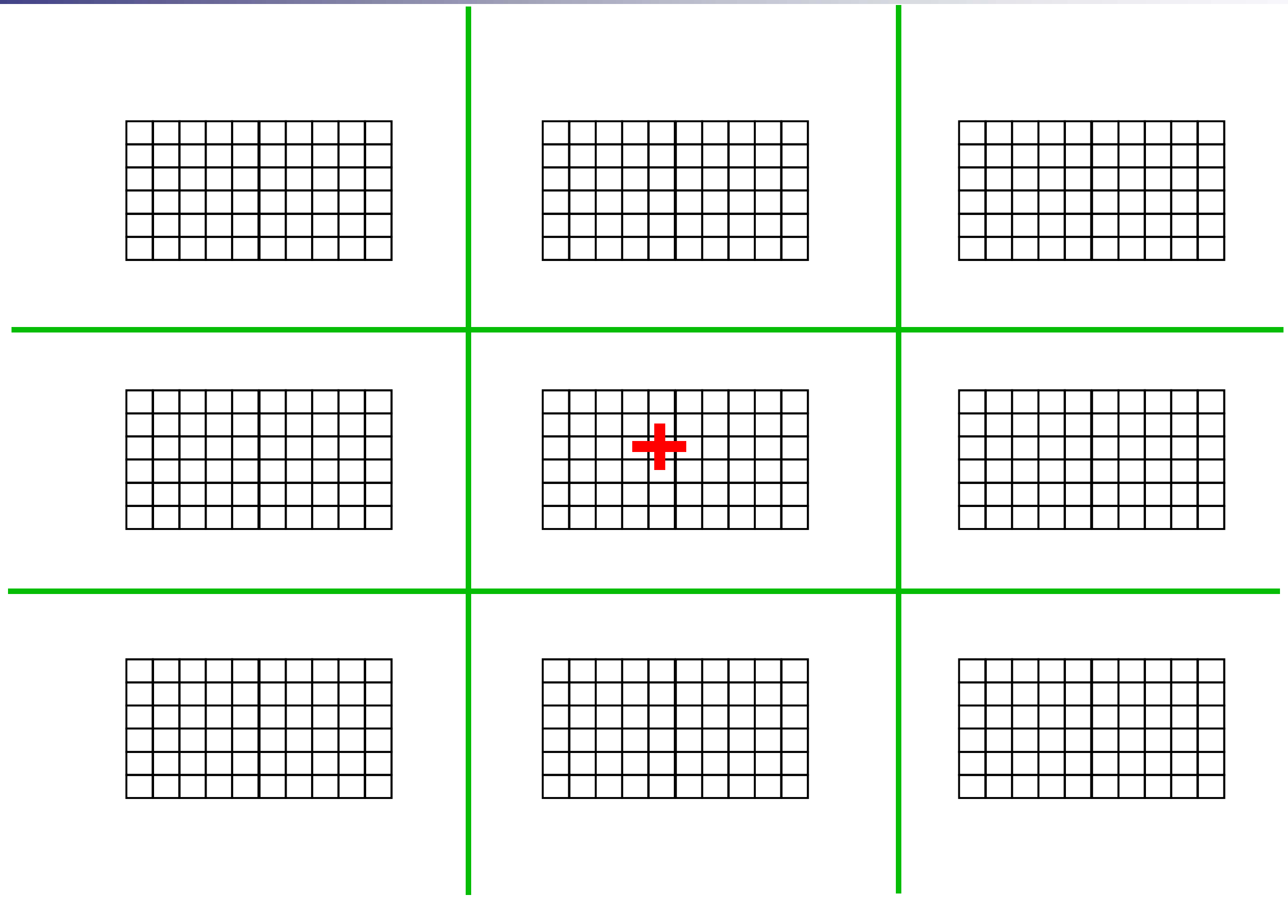
2D Poisson Problem



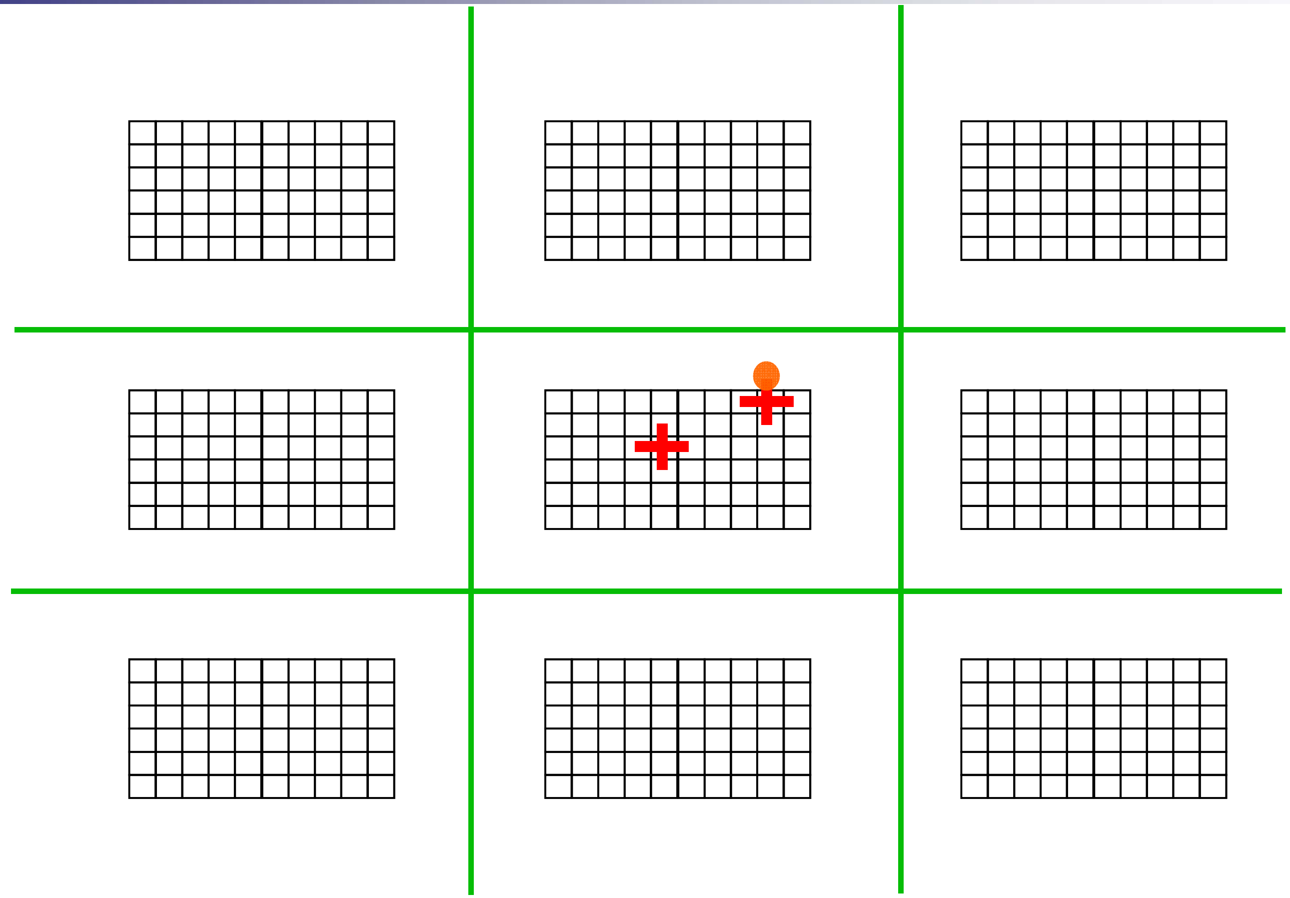
Regular Mesh Algorithms

- Many scientific applications involve the solution of partial differential equations (PDEs)
- Many algorithms for approximating the solution of PDEs rely on forming a set of difference equations
  - Finite difference, finite elements, finite volume
- The exact form of the differential equations depends on the particular method
  - From the point of view of parallel programming for these algorithms, the operations are the same
- Five-point stencil is a popular approximation solution

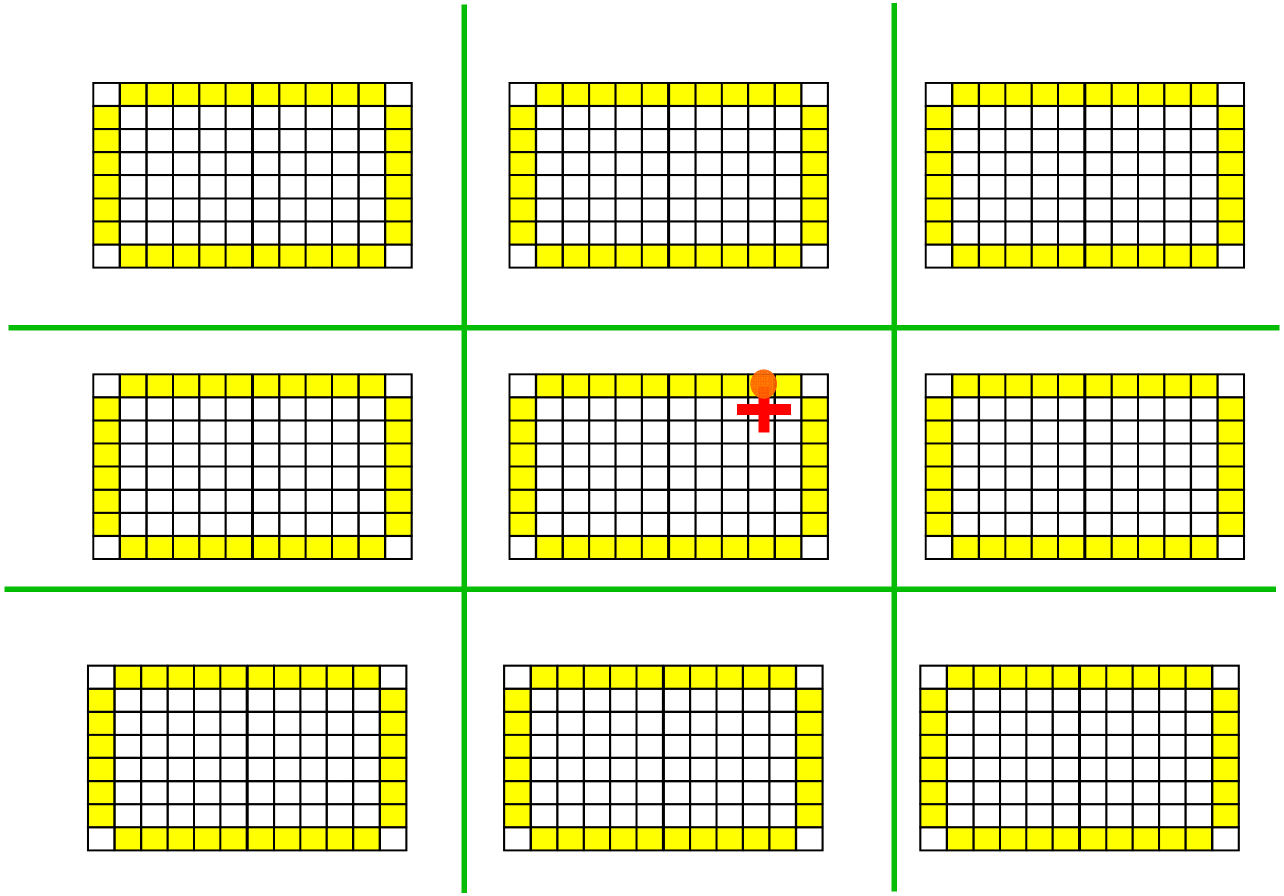
Necessary Data Transfers



Necessary Data Transfers

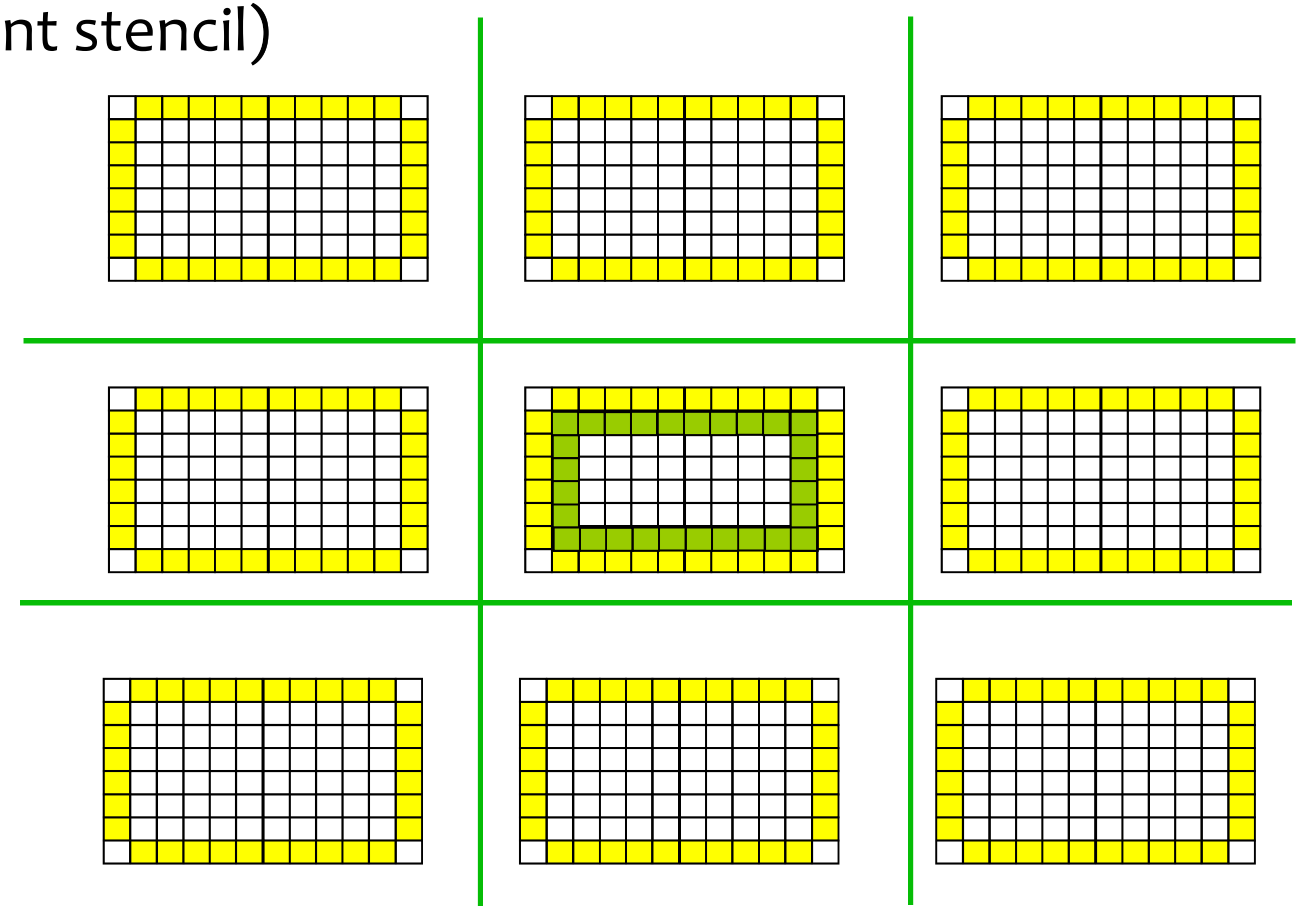


Necessary Data Transfers



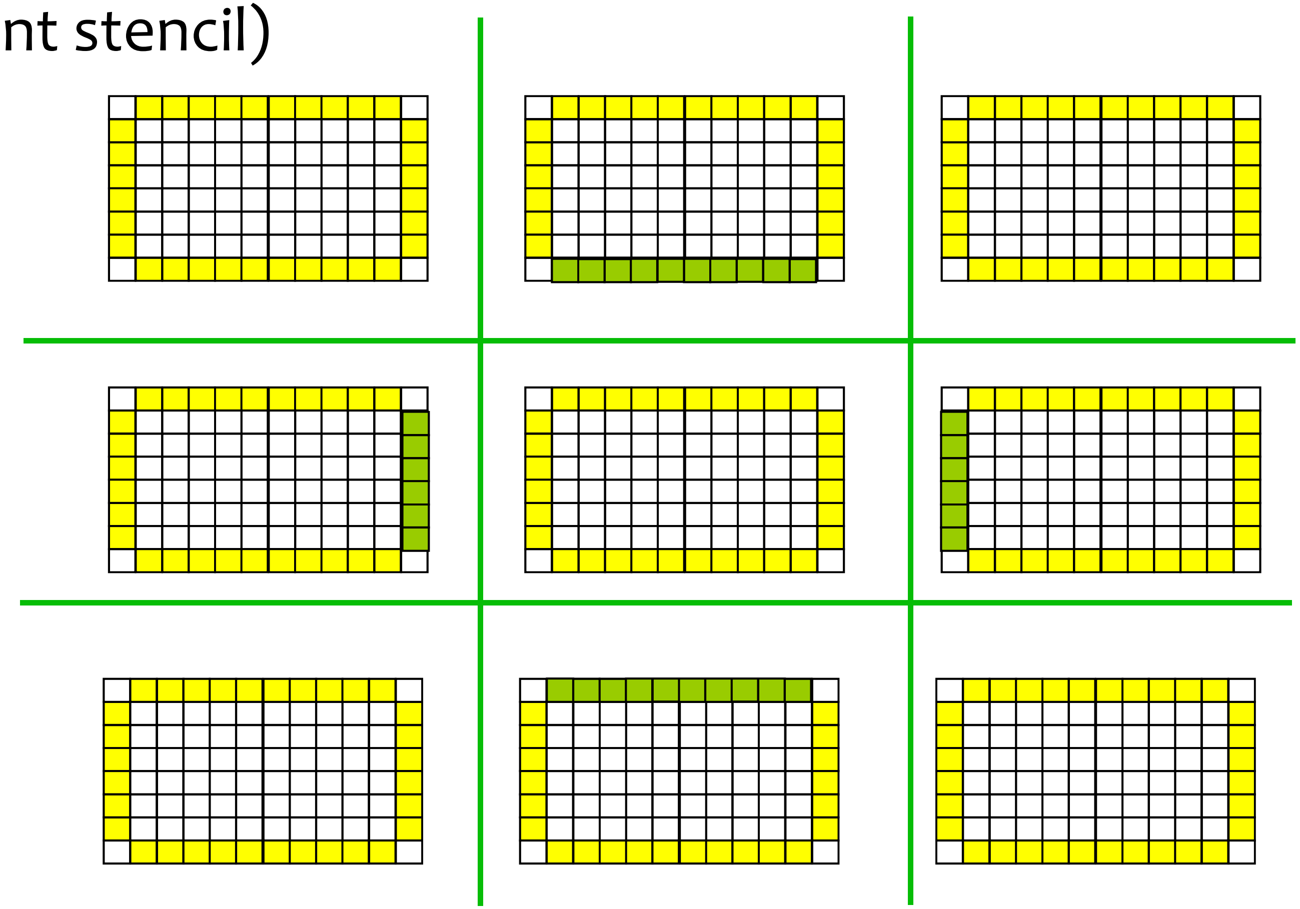
Necessary Data Transfers

- Provide access to remote data through a *halo* exchange (5 point stencil)



Necessary Data Transfers

- Provide access to remote data through a *halo* exchange (5 point stencil)

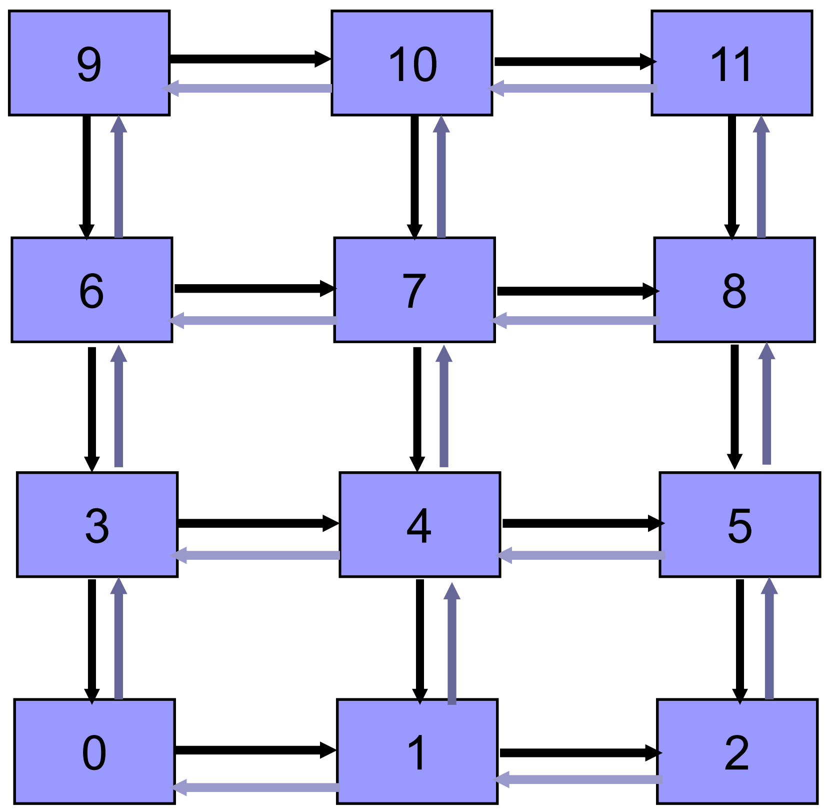


Understanding Performance:

- Unexpected Hot Spots
  - Basic performance analysis looks at two-party exchanges
  - Real applications involve many simultaneous communications
  - Performance problems can arise even in common grid exchange patterns
  - Message passing illustrates problems present even in shared memory
    - Blocking operations may cause unavoidable memory stalls

Mesh Exchange

- Exchange data on a mesh



Sample Code

```
for (i = 0; i < n_neighbors; i++) {
  MPI_Send(edge, len, MPI_DOUBLE, nbr[i], tag, comm);
}
for (i = 0; i < n_neighbors; i++) {
  MPI_Recv(edge, len, MPI_DOUBLE, nbr[i], tag, comm, status);
}
```

- What is wrong with this code?

Deadlocks!

- All of the sends may block, waiting for a matching receive (will for large enough messages)
- The variation of

if (has up nbr)

```
MPI_Recv( … up … )
```

…

if (has down nbr)

```
MPI_Send( … down … )
```

sequentializes (all except the bottom process blocks)

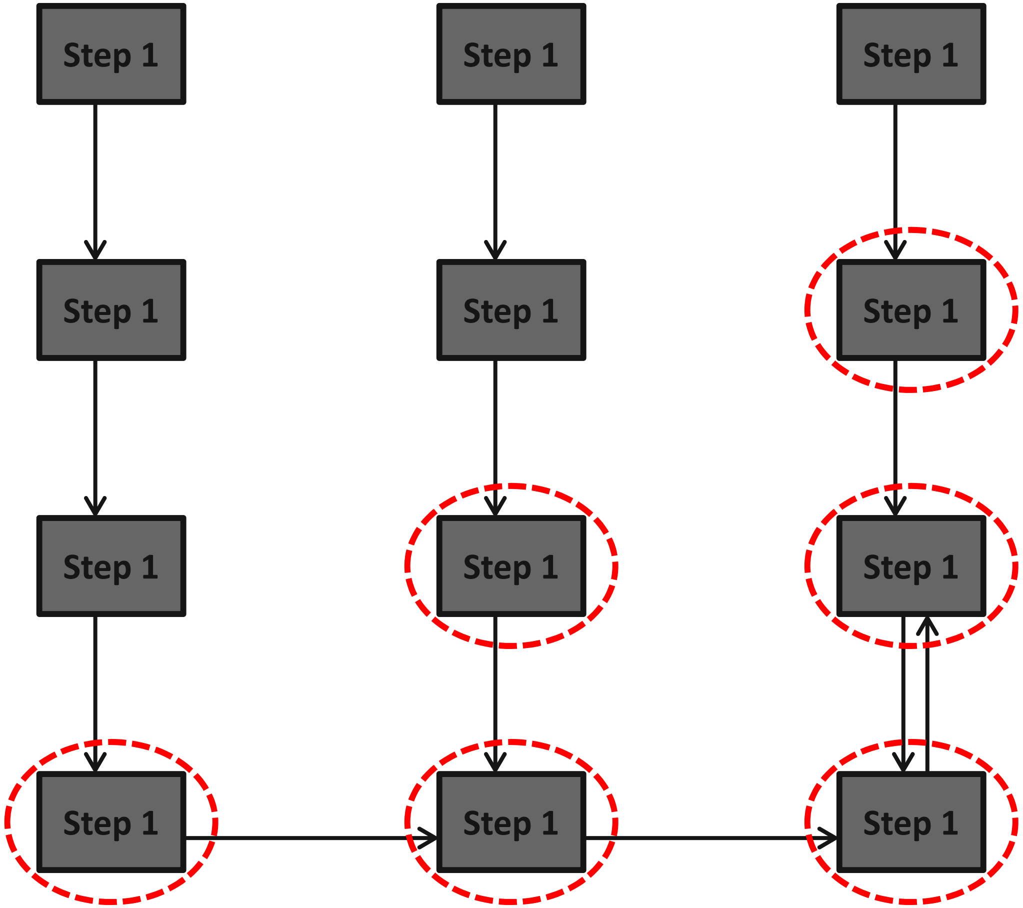
Fix 1: Use Irecv

```
for (i = 0; i < n_neighbors; i++) {
      MPI_Irecv(edge, len, MPI_DOUBLE, nbr[i], tag,
                    comm, requests[i]);
}
for (i = 0; i < n_neighbors; i++) {
      MPI_Send(edge, len, MPI_DOUBLE, nbr[i], tag, comm);
}
MPI_Waitall(n_neighbors, requests, statuses);
```

- Does not perform well in practice. Why?

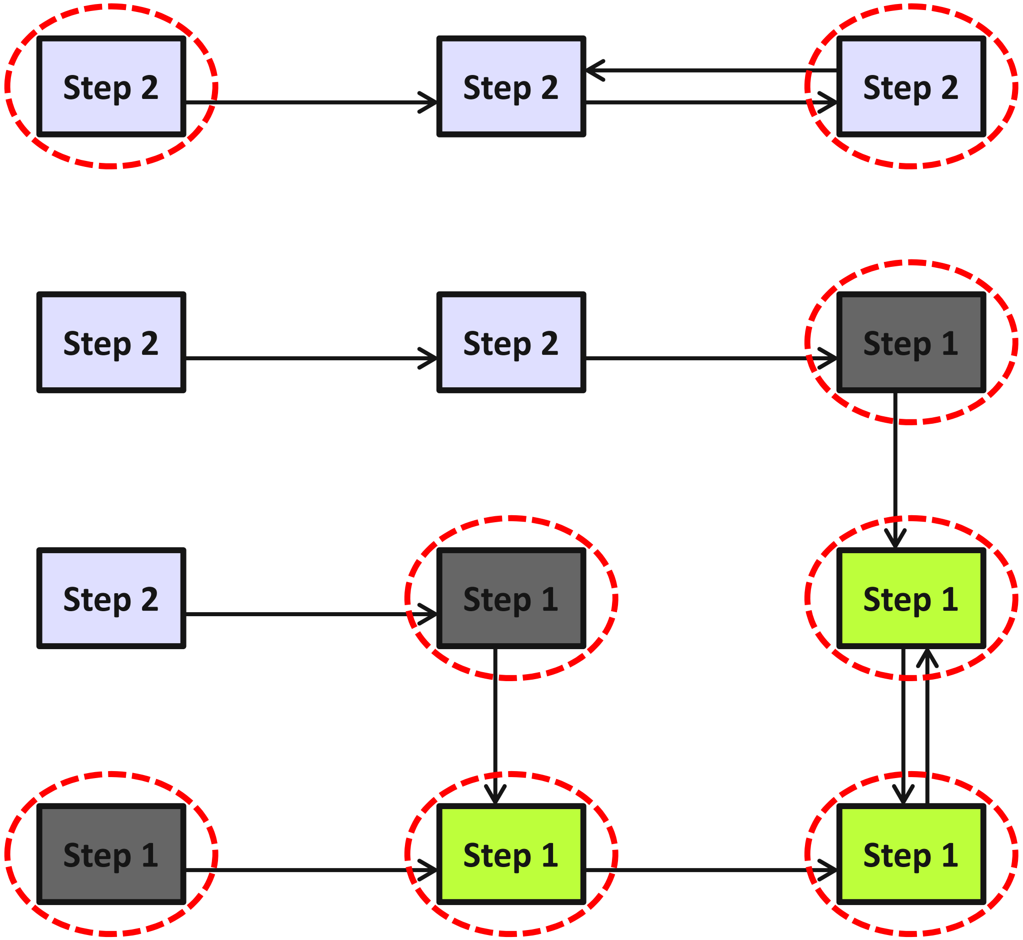
Mesh Exchange

- Exchange data on a mesh



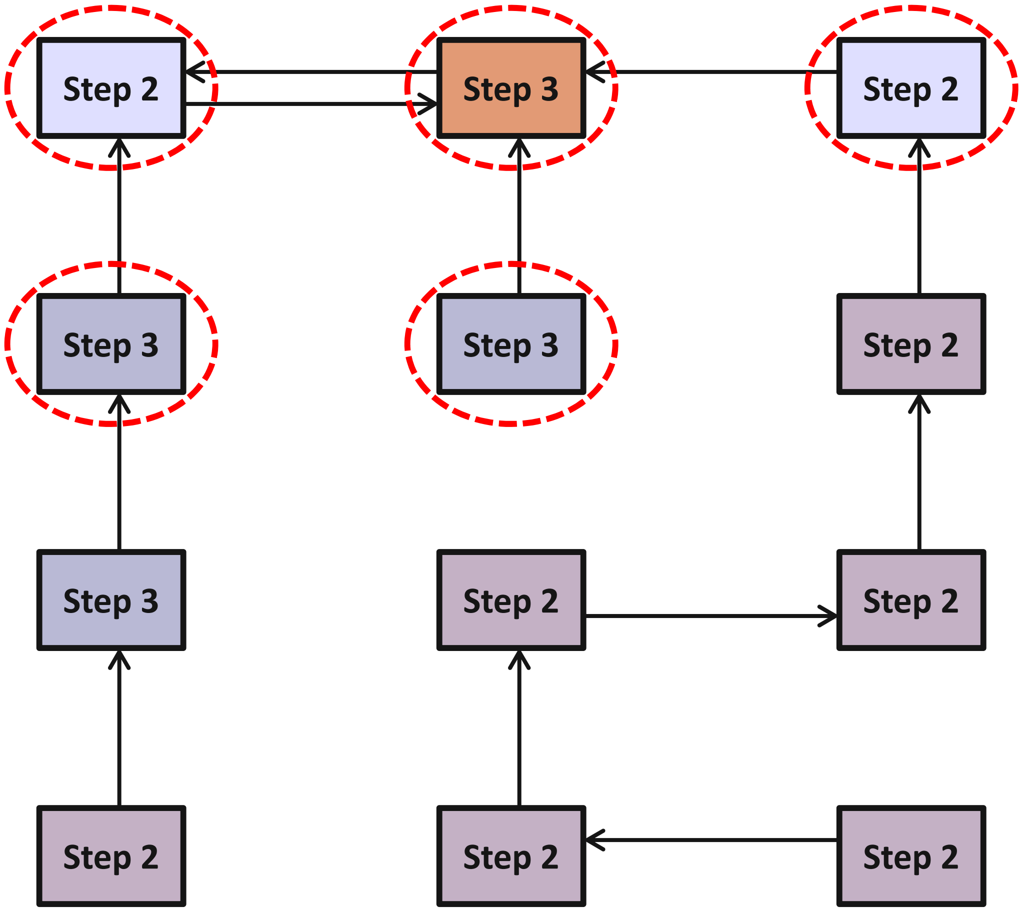
Mesh Exchange

- Exchange data on a mesh



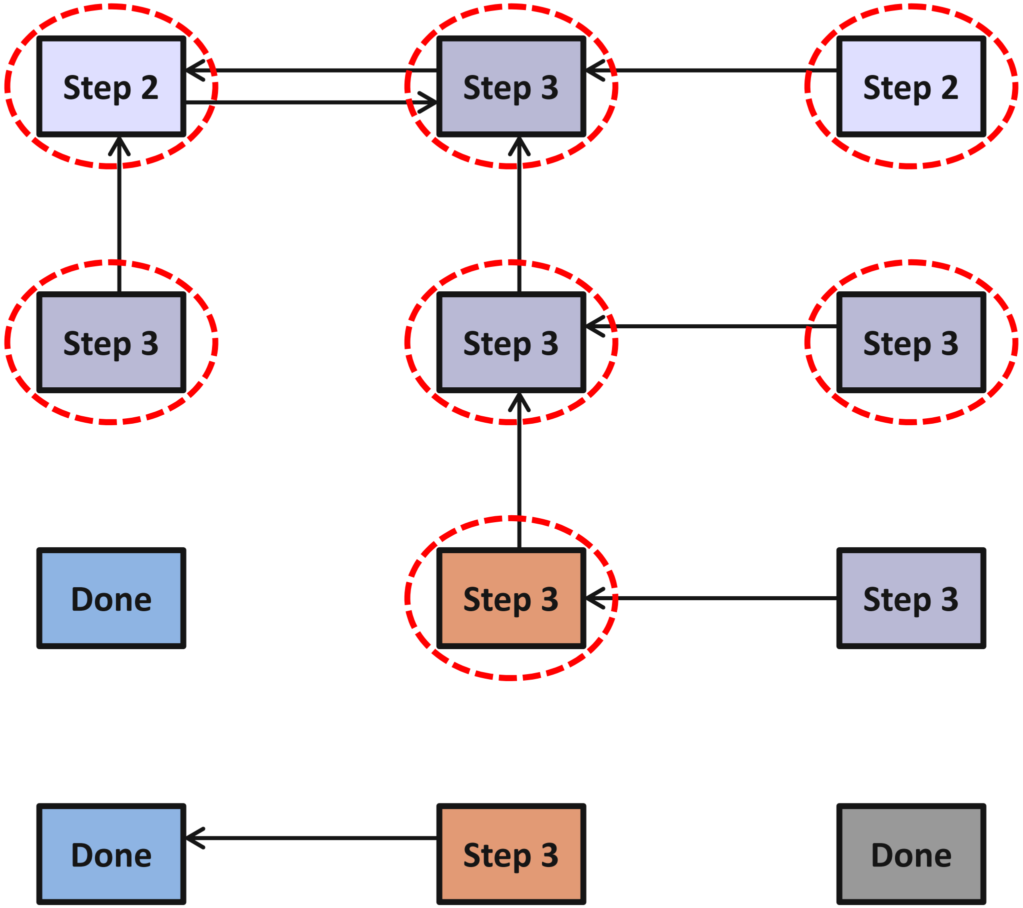
Mesh Exchange

- Exchange data on a mesh



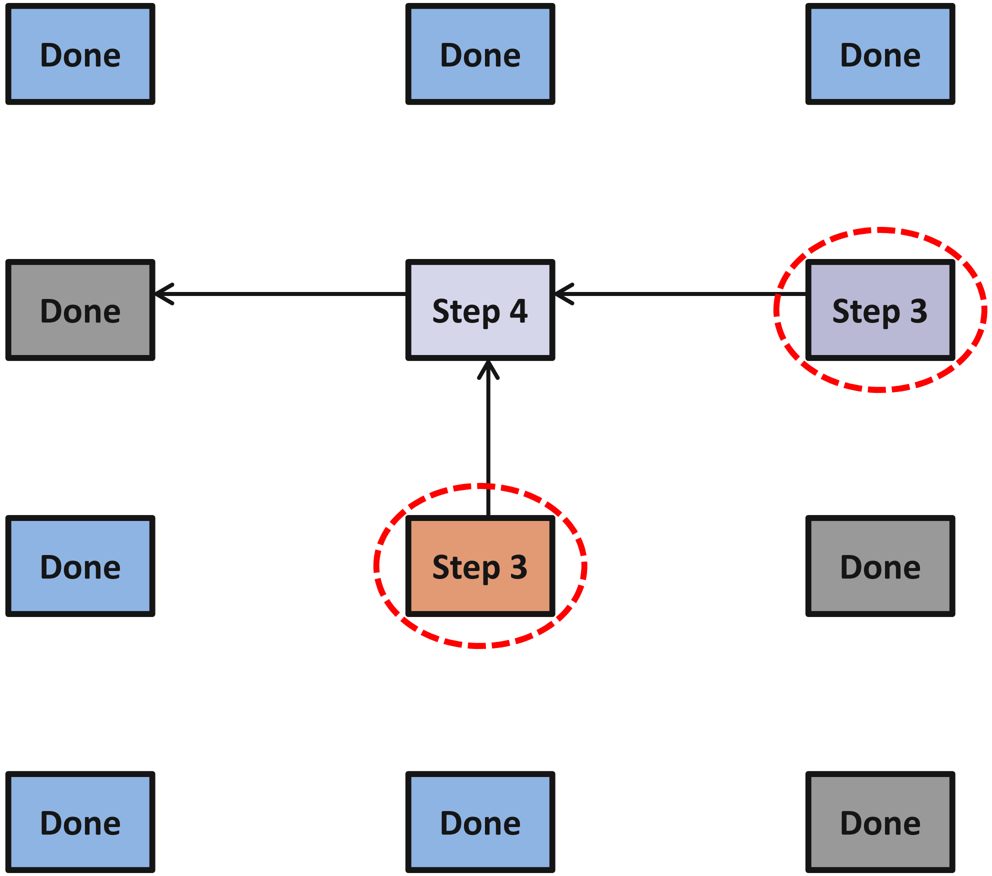
Mesh Exchange

- Exchange data on a mesh



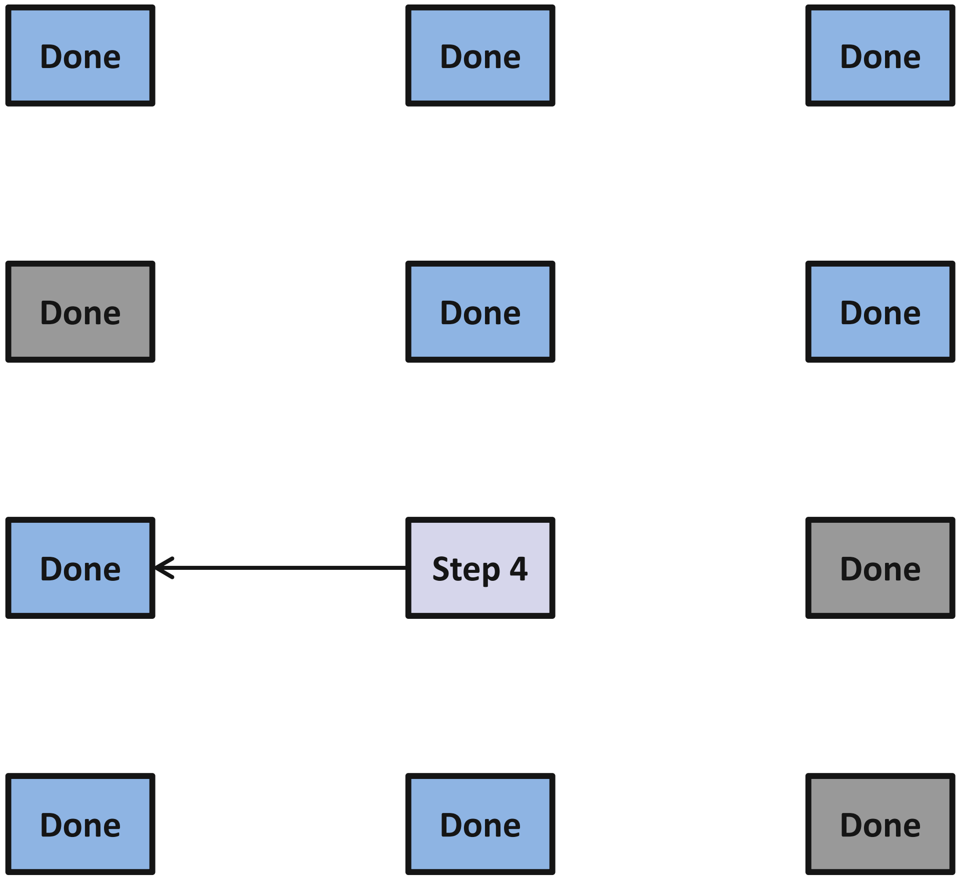
Mesh Exchange

- Exchange data on a mesh



Mesh Exchange

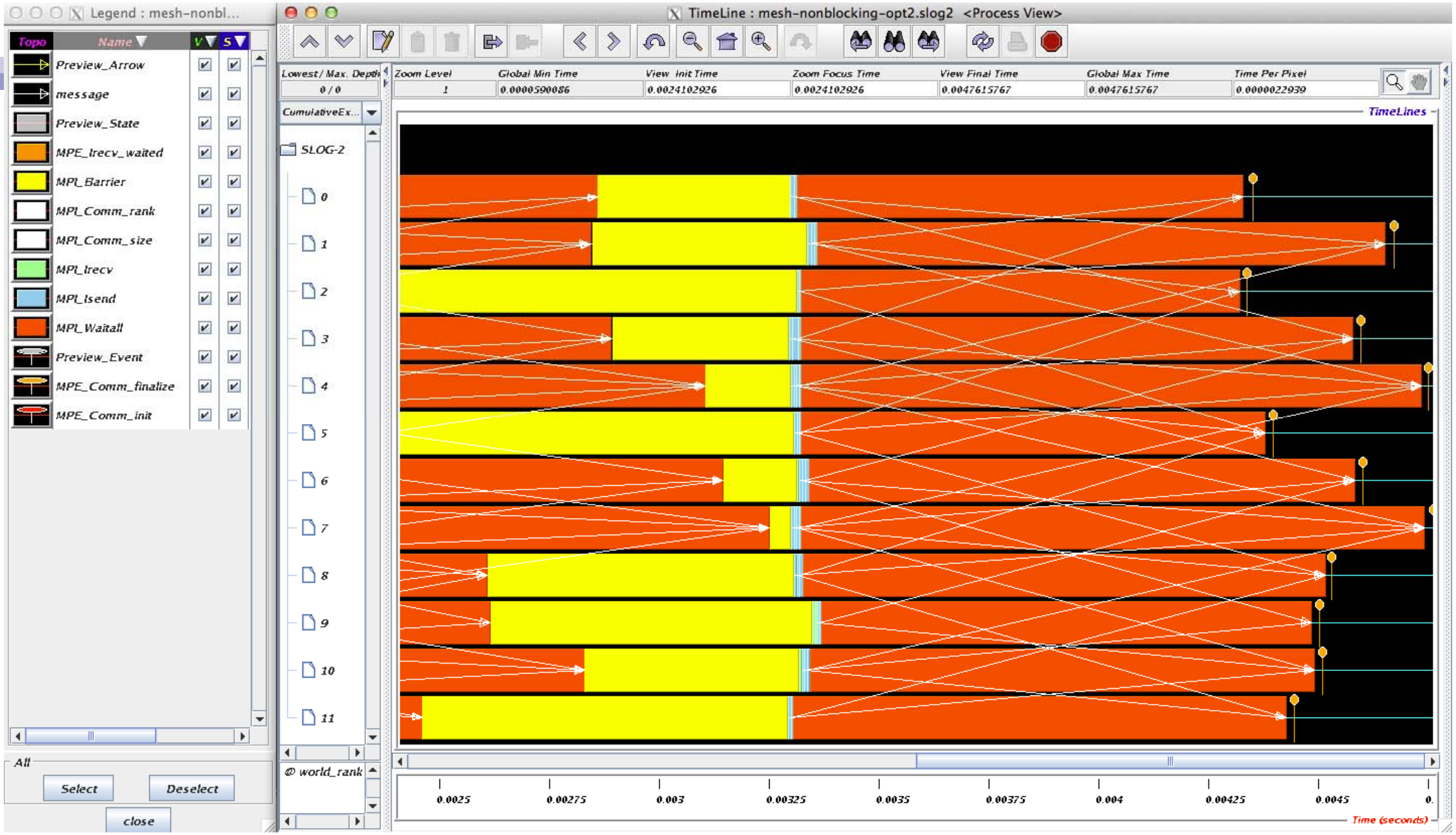
- Exchange data on a mesh



Fix 2: Use Isend and Irecv

```
for (i = 0; i < n_neighbors; i++) {
    MPI_Irecv(edge, len, MPI_DOUBLE, nbr[i], tag,
             comm, requests[i]);
}
for (i = 0; i < n_neighbors; i++) {
    MPI_Isend(edge, len, MPI_DOUBLE, nbr[i], tag, comm,
             requests[n_neighbors + i]);
}
MPI_Waitall(2 * n_neighbors, requests, statuses);
```

Timeline from IB Cluster



Note processes 4 and 7 are the only interior processors; these perform more communication than the other processors

Lesson: Defer Synchronization

- Send-receive accomplishes two things:
  - Data transfer
  - Synchronization
- In many cases, there is more synchronization than required
- Use non-blocking operations and **`MPI_Waitall`** to defer synchronization

- Tools can help out with identifying performance issues:
  - MPE, Tau and HPCToolkit are popular profiling tools
  - Jumpshot tool uses their datasets to show performance problems graphically
  - Display message queue state using Totalview

Content

- Non-blocking communication
- **Group (collective) communication**
- MPI Datatypes
- Virtual Topology

Collective Operations in MPI

- Collective operations are called by all processes in a communicator:
  - **`MPI_BCAST`**
    - distributes data from one process (the root) to all others in a communicator.
  - **`MPI_REDUCE`**
    - combines data from all processes in the communicator and returns it to one process.
- In many numerical algorithms:
  - **`SEND/RECV`** can be replaced by **`BCAST/REDUCE`**,
  - improving both simplicity and efficiency.

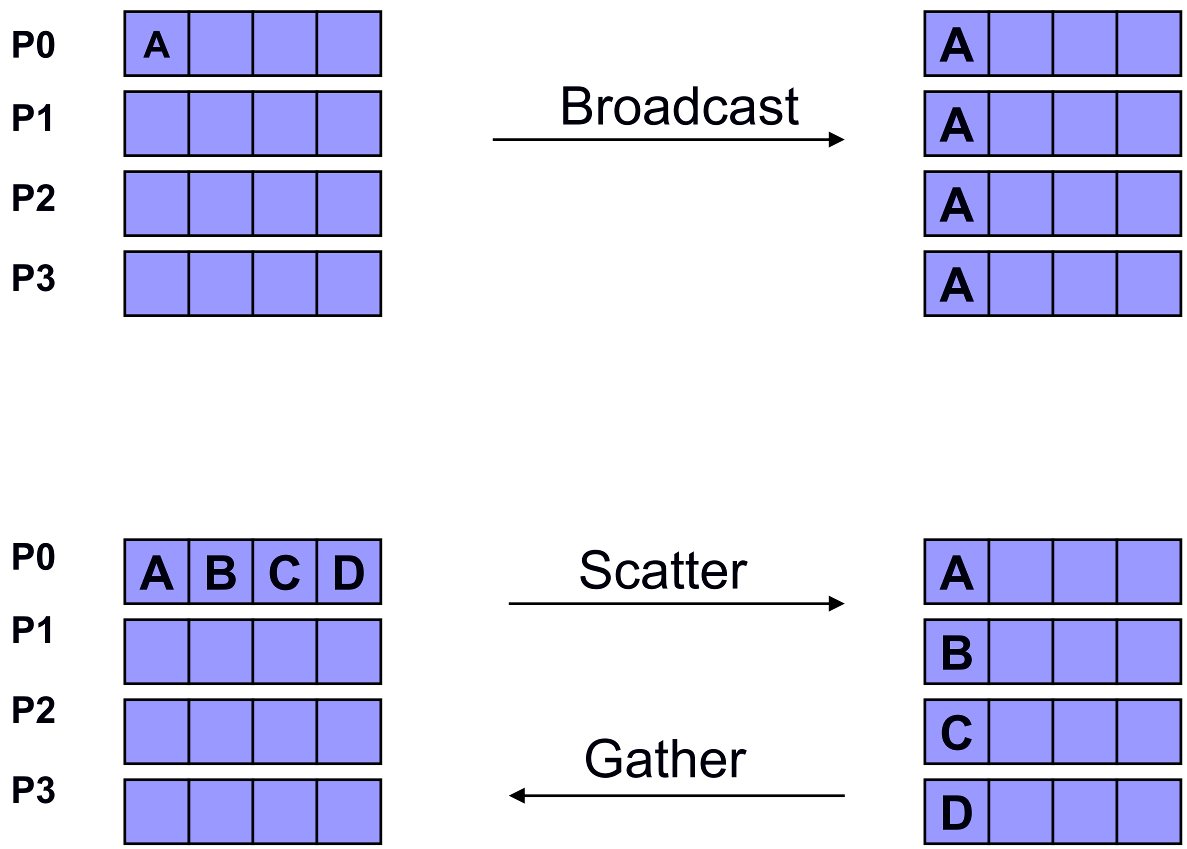
MPI Collective Communication

- Communication and computation is coordinated among a group of processes in a communicator.
  - Tags are not used;
- Three classes of operations:
  - **synchronization**
  - **data movement**
  - **collective computation**
- Non-blocking collective operations in MPI-3

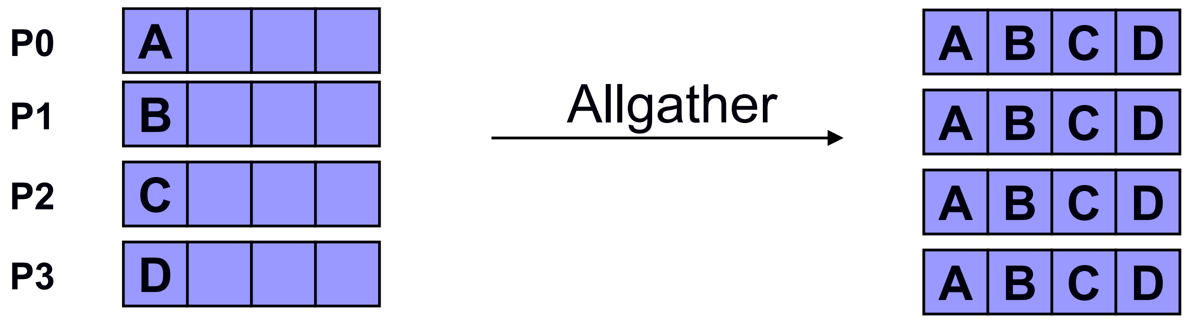
Synchronization

- `MPI_BARRIER(comm)`
  - Blocks until all processes in the group of the communicator **`comm`** call it.
  - A process cannot get out of the barrier until all other processes have reached barrier.

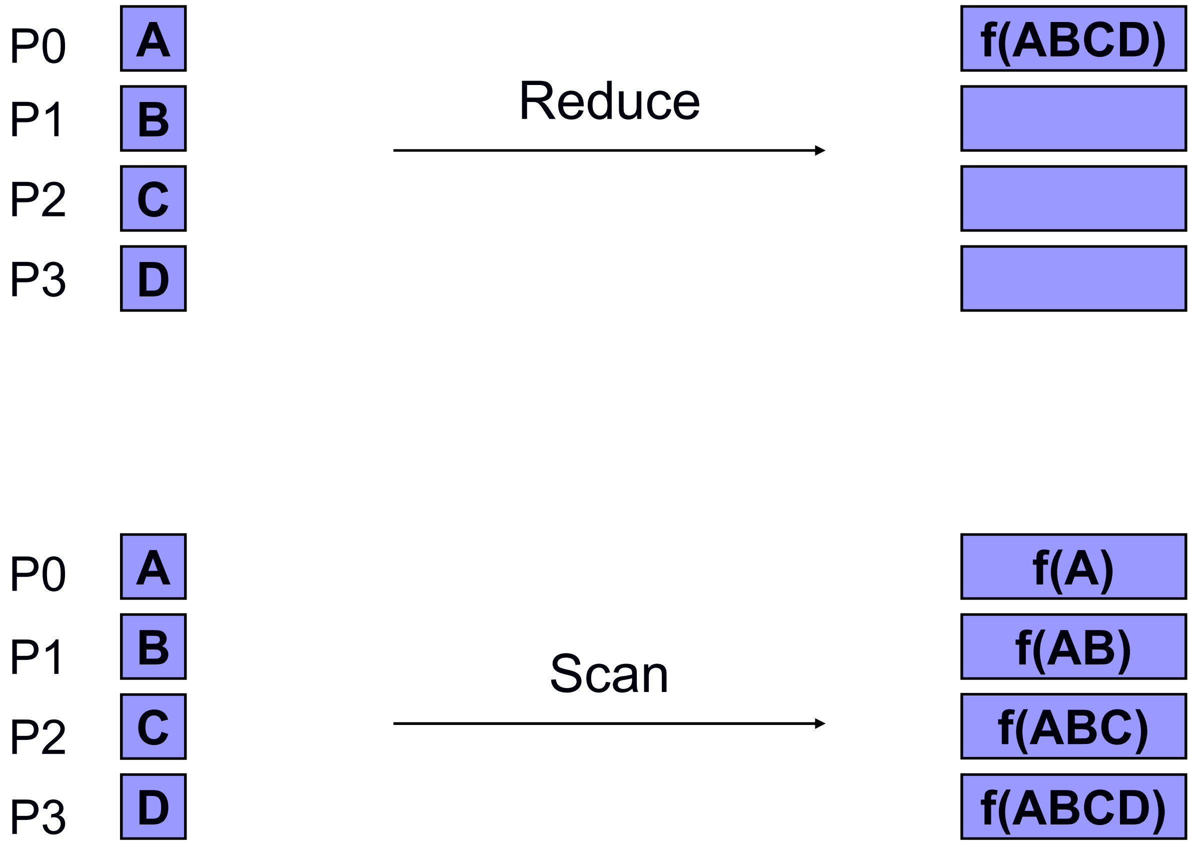
Collective Data Movement



More Collective Data Movement



Collective Computation



MPI Collective Routines

- Many Routines:
  - **`MPI_ALLGATHER, MPI_ALLGATHERV, MPI_ALLREDUCE, MPI_ALLTOALL, MPI_ALLTOALLV, MPI_BCAST, MPI_GATHER, MPI_GATHERV, MPI_REDUCE, MPI_REDUCESCATTER, MPI_SCAN, MPI_SCATTER, MPI_SCATTERV`**
- **“`All`”**
  - versions deliver results to all participating processes
- **“`V`”**
  - versions (stands for vector) allow the chunks to have different sizes

- **`MPI_ALLREDUCE`**, **`MPI_REDUCE`**, **`MPI_REDUCESCATTER`**, and **`MPI_SCAN`** take both built-in and user-defined combiner functions

MPI Built-in Collective Computation

MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_PROD	Product
MPI_SUM	Sum
MPI_LAND	Logical and
MPI_LOR	Logical or
MPI_LXOR	Logical exclusive or
MPI_BAND	Bitwise and
MPI_BOR	Bitwise or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

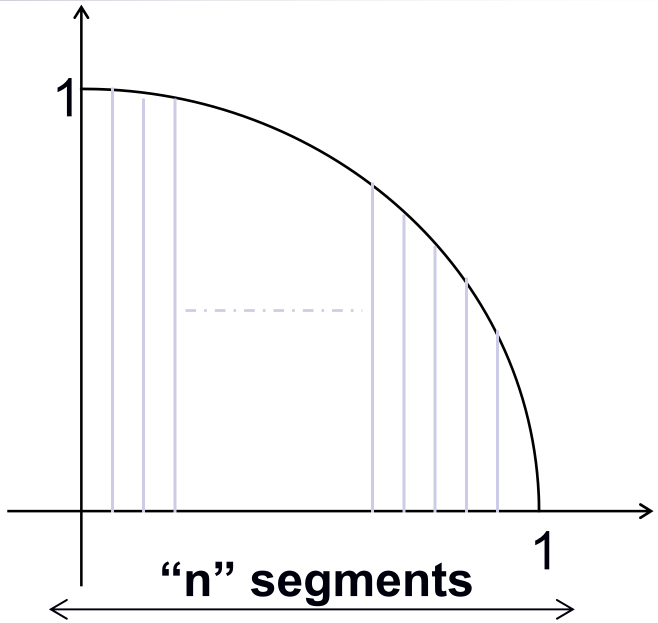
Defining your own Collective Operations

- Create your own collective computations with:
  ```
  MPI_OP_CREATE(user_fn, commutes, &op);
  MPI_OP_FREE(&op);

  user_fn(invec, inoutvec, len, datatype);
  ```
- The user function should perform:
  ```
  inoutvec[i]  =  invec[i]  op  inoutvec[i];
  ```
  for i from 0 to len-1
- The user function:
  - can be non-commutative;
  - but must be associative.

Example: Calculating Pi

- Calculating pi via numerical integration
  - Divide interval up into subintervals
  - Assign subintervals to processes
  - Each process calculates partial sum
  - Add all the partial sums together to get pi



1. Width of each segment (w) will be 1/n
2. Distance (d(i)) of segment "i" from the origin will be "i * w"
3. Height of segment "i" will be sqrt(1 – [d(i)]^2)

Example: PI in C

```
#include <mpi.h>
#include <math.h>
int main(int argc, char *argv[])
{
    [...snip...]
    /* Tell all processes, the number of segments you want */
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    w   = 1.0 / (double) n;
    mypi = 0.0;
    for (i = rank + 1; i <= n; i += size)
        mypi += w * sqrt(1 – (((double) i / n) * ((double) i / n));
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);
    if (rank == 0)
        printf("pi is approximately %.16f, Error is %.16f\n", 4 *
pi, fabs((4 * pi) - PI25DT));
    [...snip...]
}
```

Code Example - 3

- *code/test3/*
  - *cpi.c*
  - *mpi_group_scatter.c*
  - *mpi_group_reduce_pi.c*
  - *mpi_group_communicater.c*

Content

- Non-blocking communication
- Group (collective) communication
- **MPI Datatypes**
- Virtual Topology