MESSAGE PASSING PROGRAMS

Message Passing Model

- Multiple “threads” of execution
  - Do not share address space
  - Processes
  - Each process may further have multiple threads of control

Message Passing Model

- Multiple “threads” of execution
  - Do not share address space
  - Processes
  - Each process may further have multiple threads of control

Shared Memory Model

Read Input
Create Sharing threads:
Process(sharedInput, myID)

Message Passing Model

- Multiple “threads” of execution
  - Do not share address space
  - Processes
  - Each process may further have multiple threads of control

Shared Memory Model

Read Input
Create Sharing threads:
Process(sharedInput, myID)

Message Passing Model

Read Input
Create Remote Processes
Loop: Send data to each process
Wait and collect results

Message Passing Model

- Multiple “threads” of execution
  - Do not share address space
  - Processes
  - Each process may further have multiple threads of control

Shared Memory Model

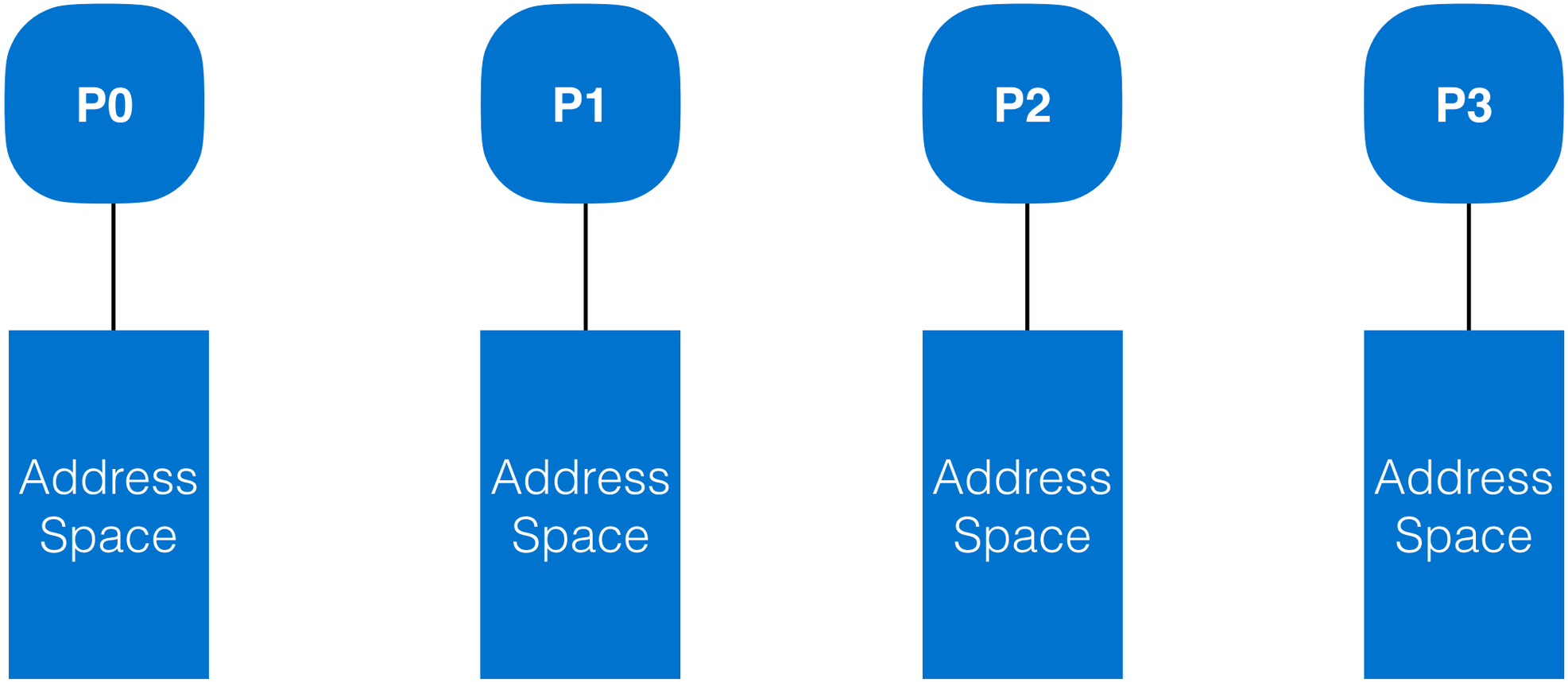
Read Input
Create Sharing threads:
Process(sharedInput, myID)

Message Passing Model

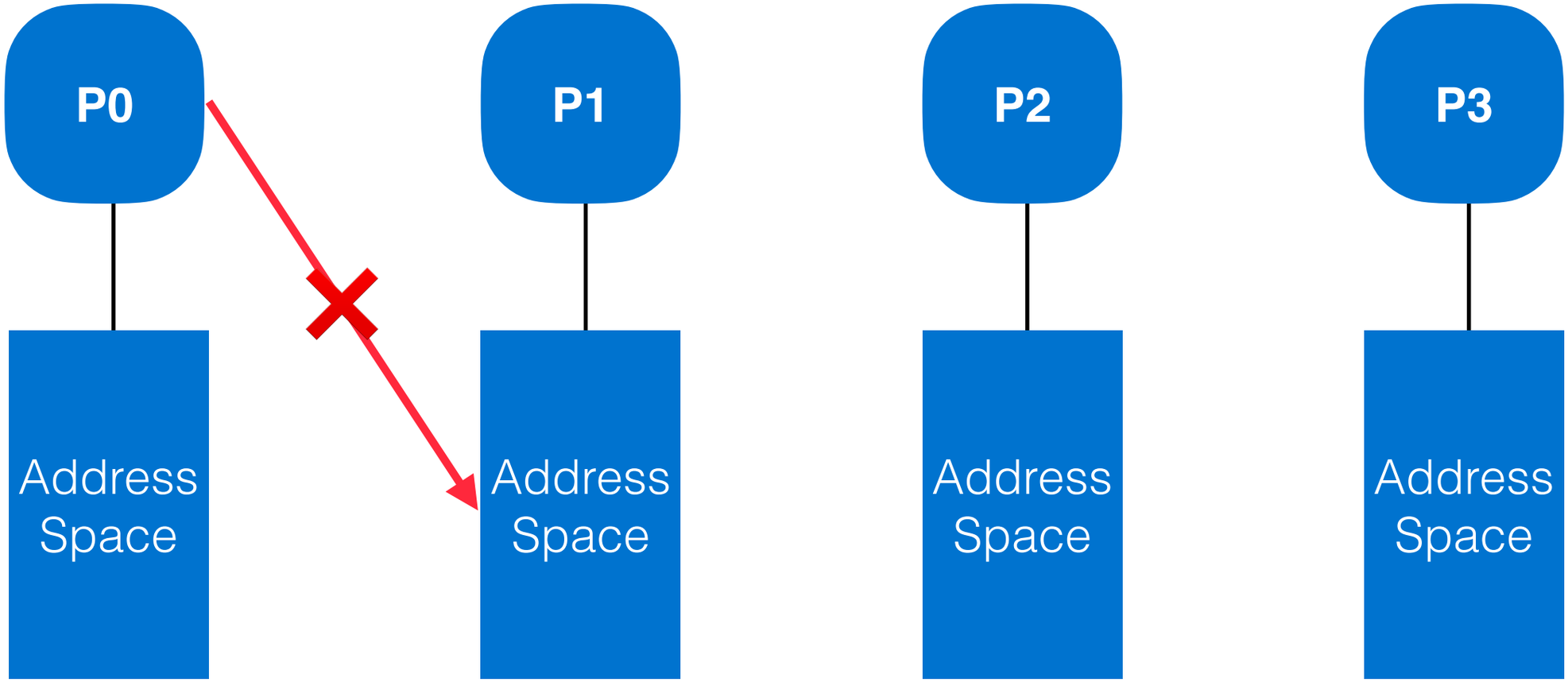
Read Input
Create Remote Processes
Loop: Send data to each p:
Wait and collect results

Recv data
Process(data)
Send results

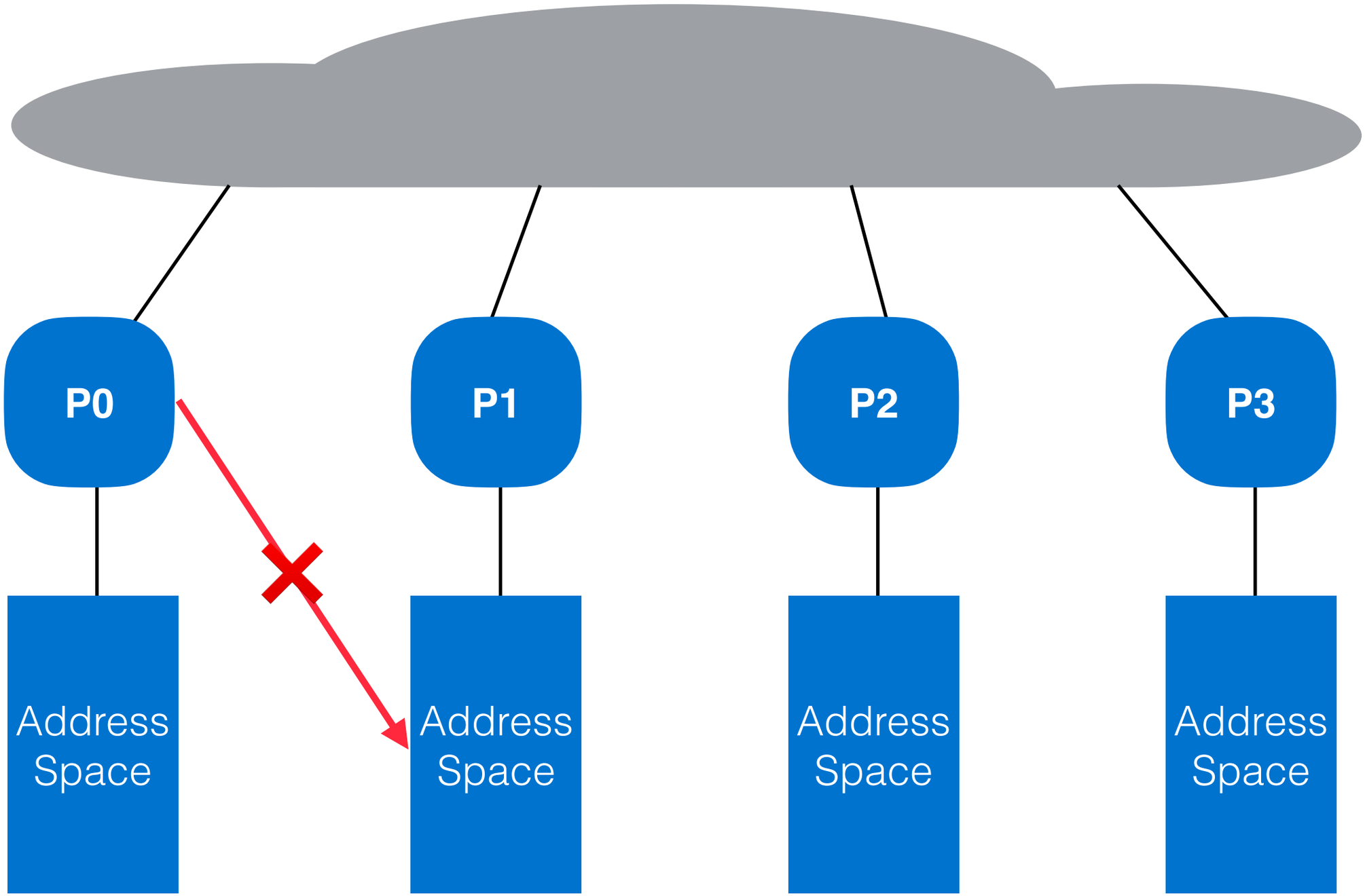
Distributed Memory



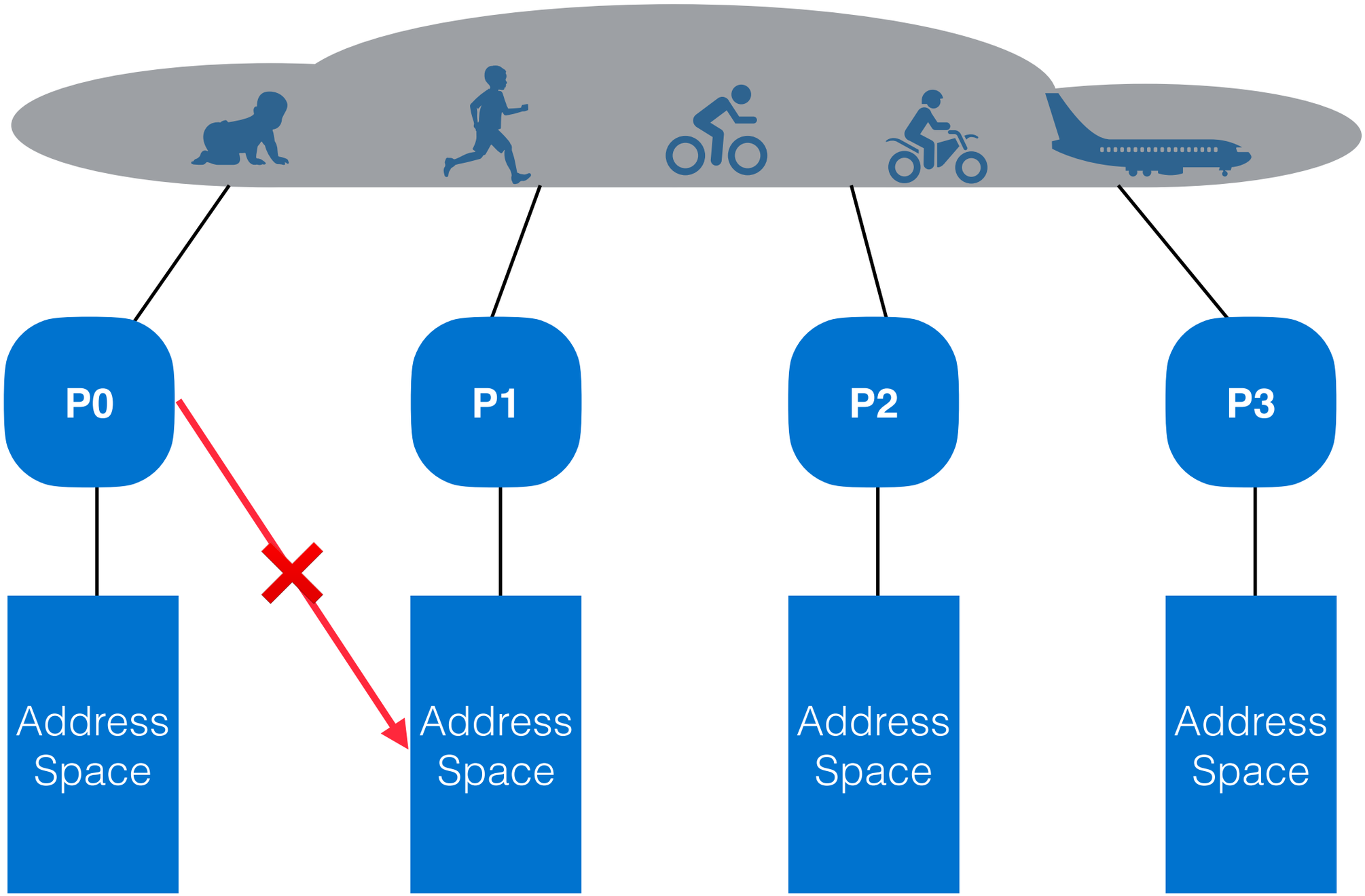
Distributed Memory



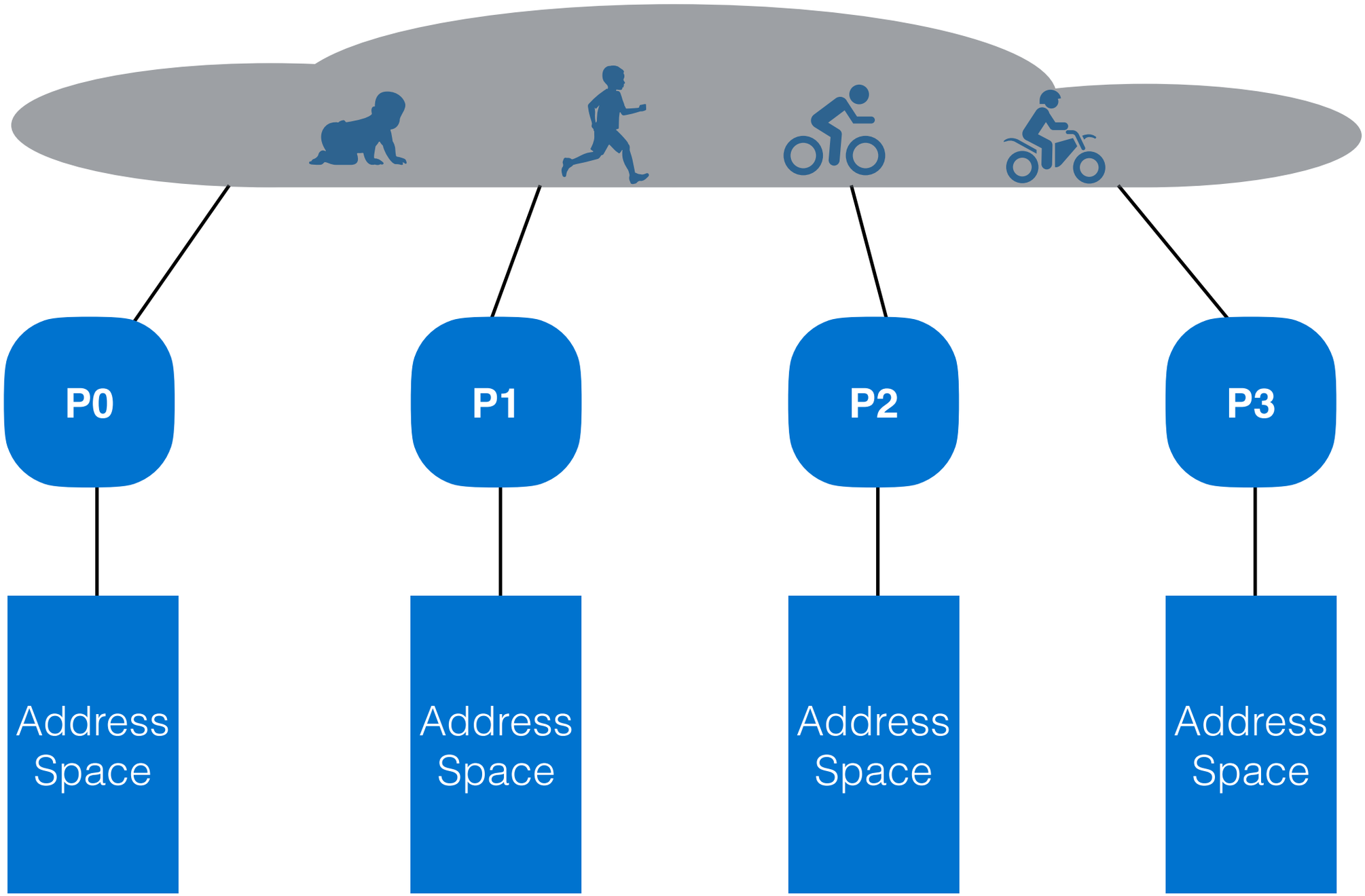
Distributed Memory



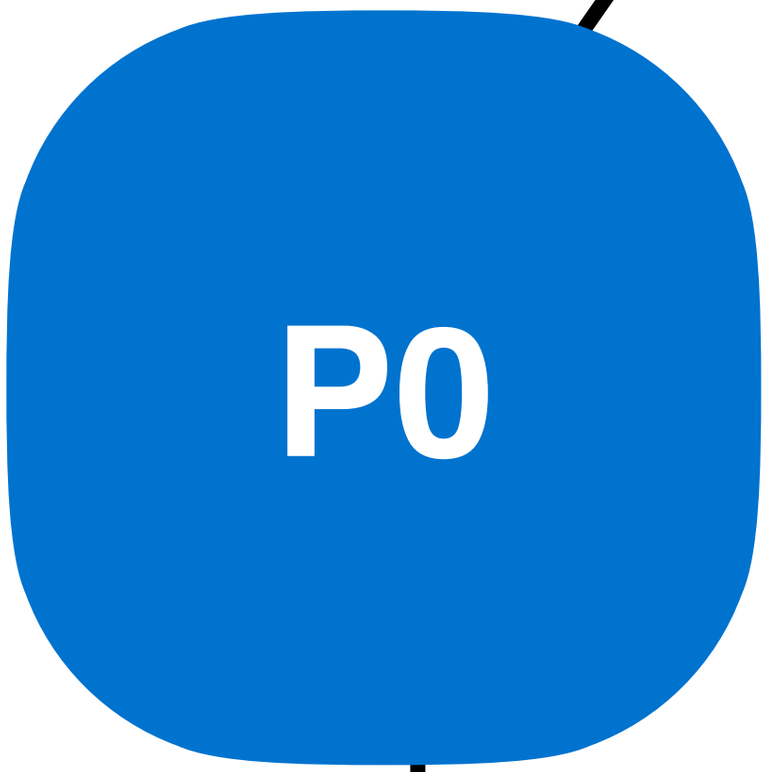
Distributed Memory



Distributed Memory



Distributed Memory

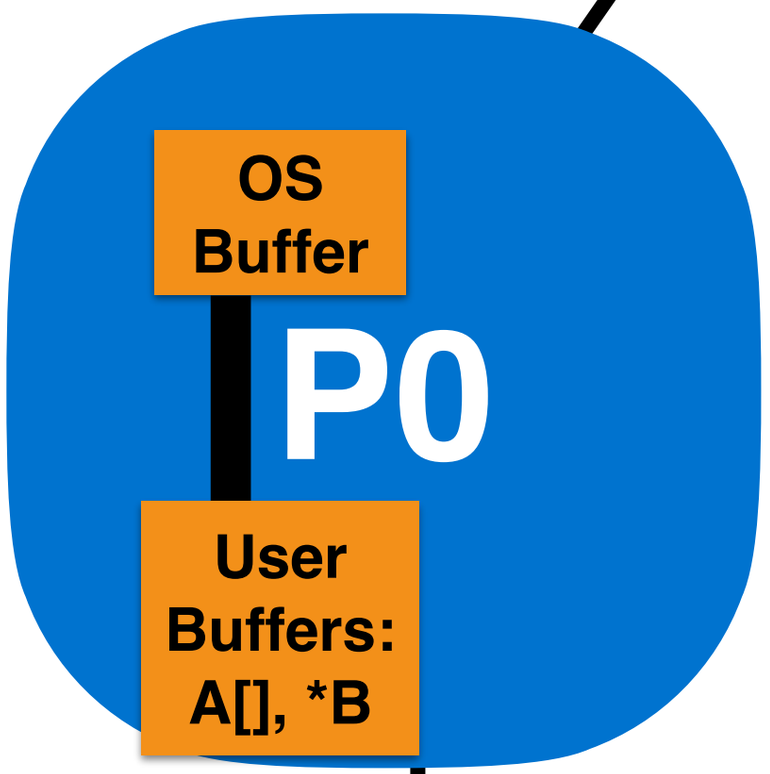


Distributed Memory

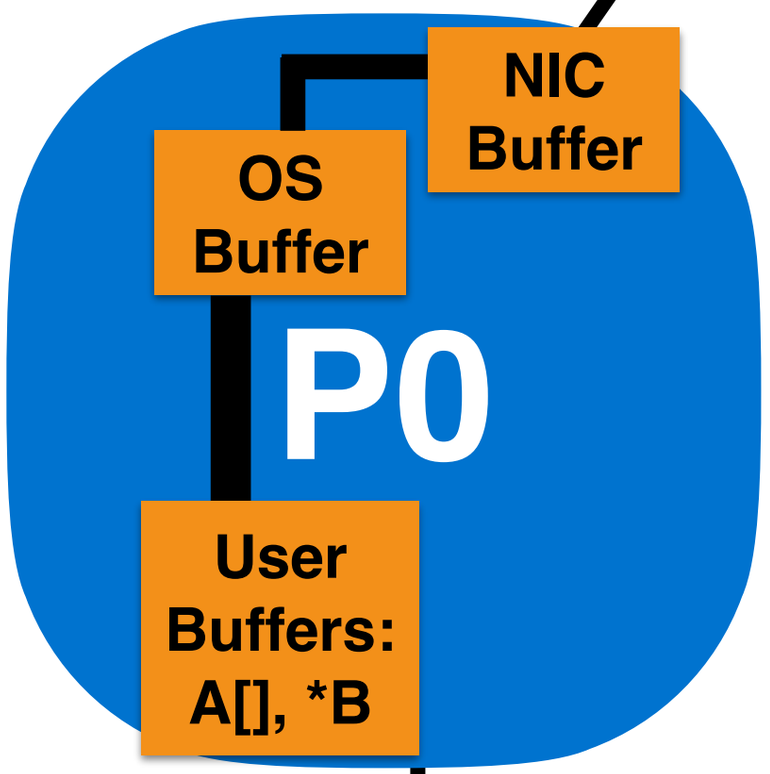
P0

User Buffers: A[], *B

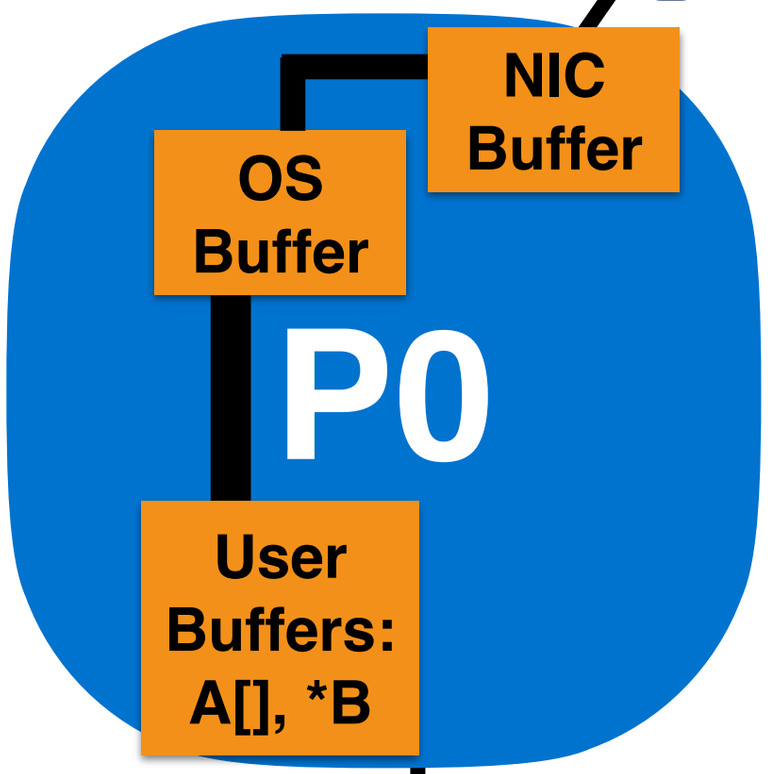
Distributed Memory



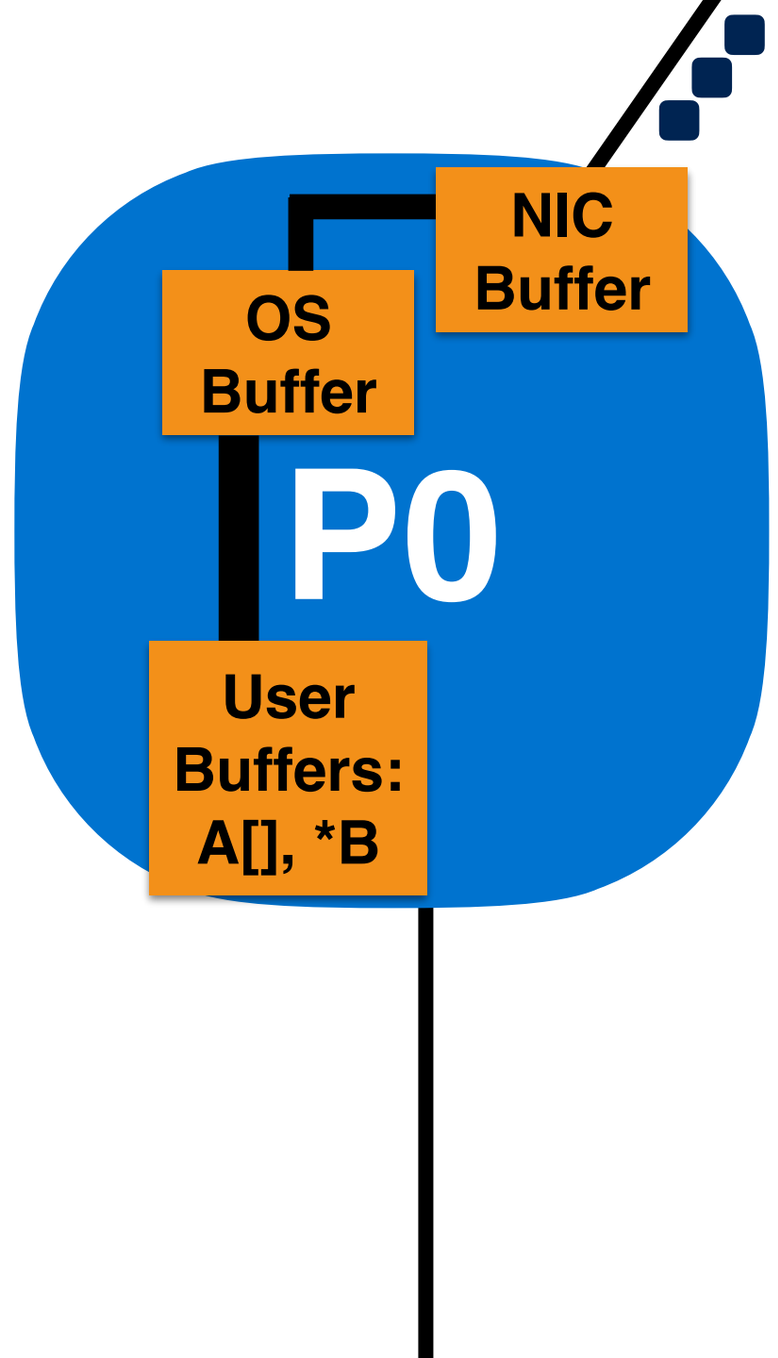
Distributed Memory



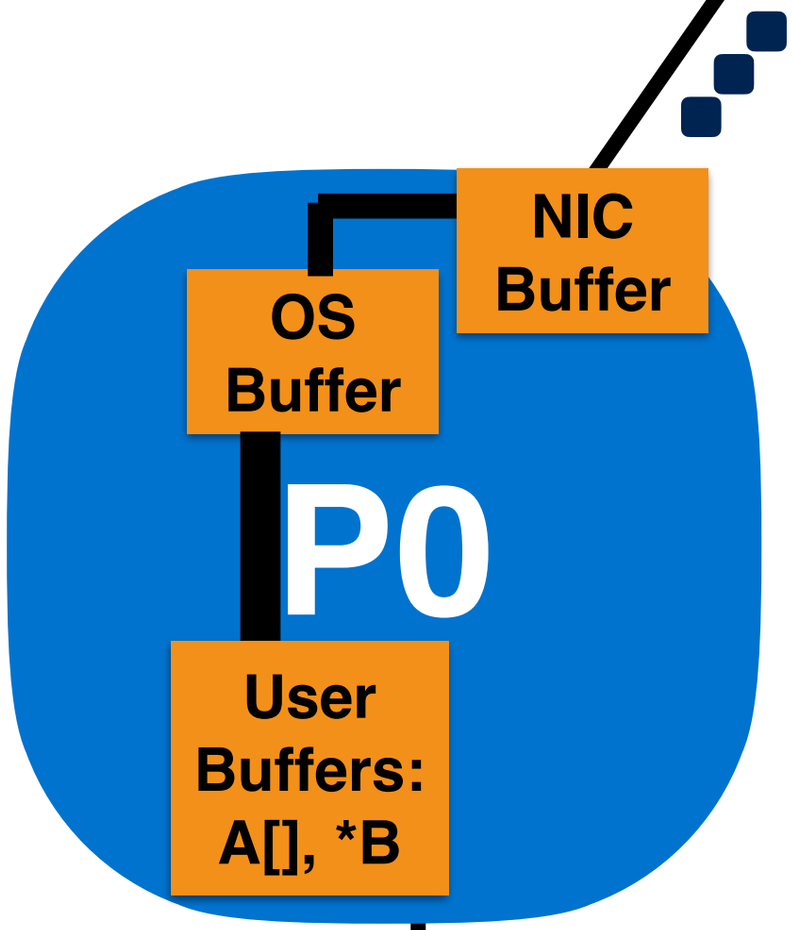
Distributed Memory

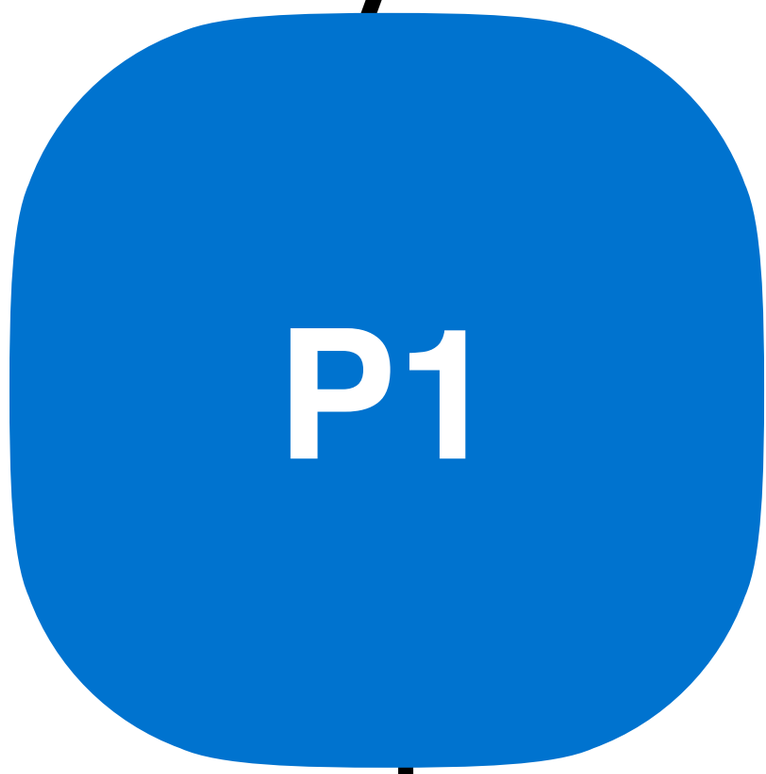


Distributed Memory

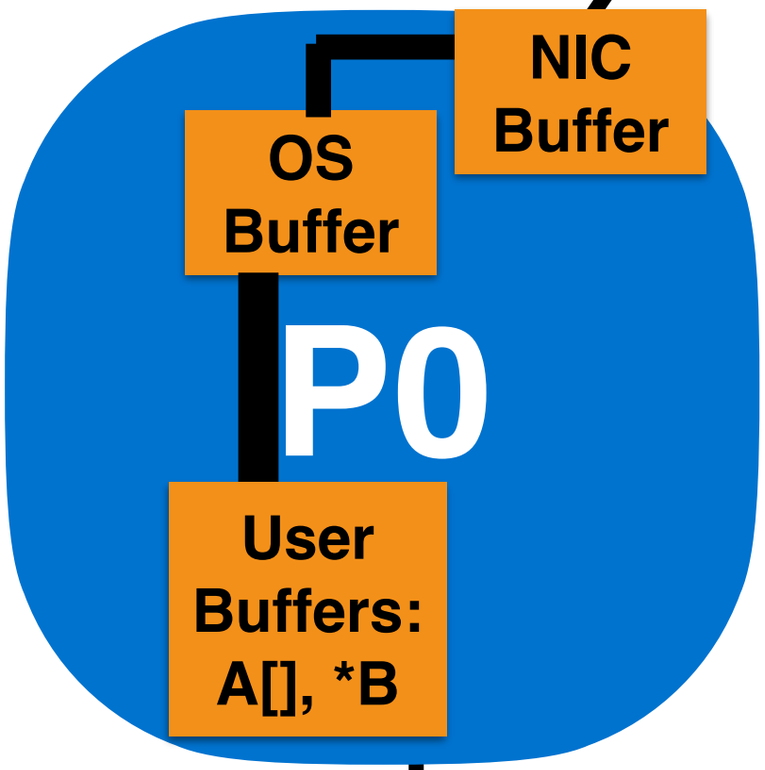


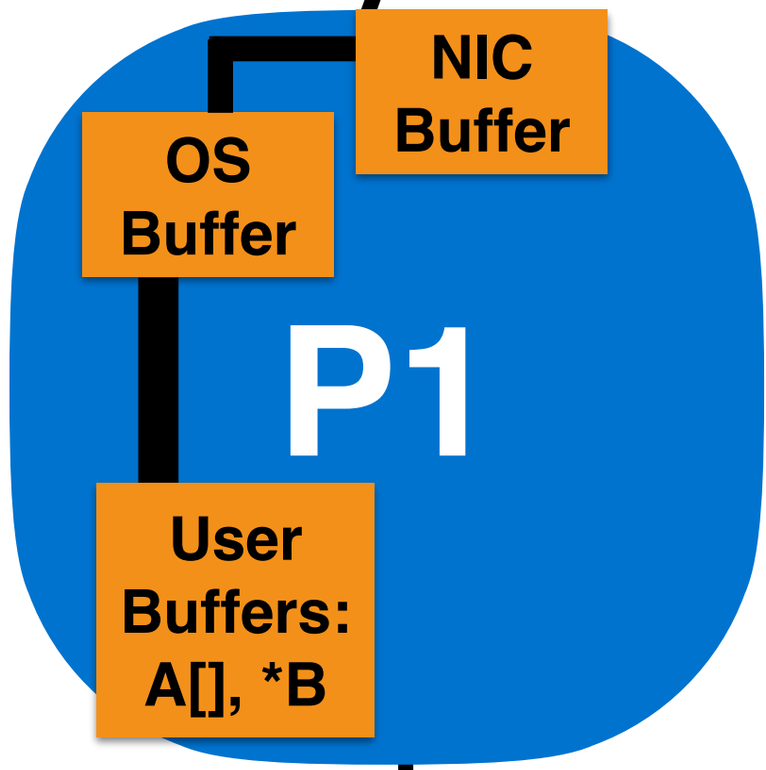
Distributed Memory



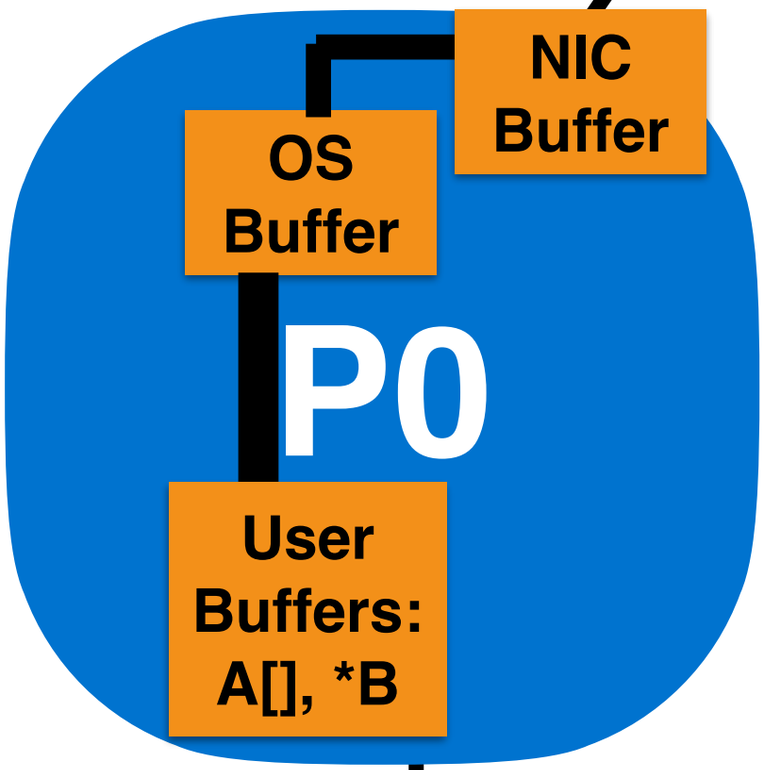


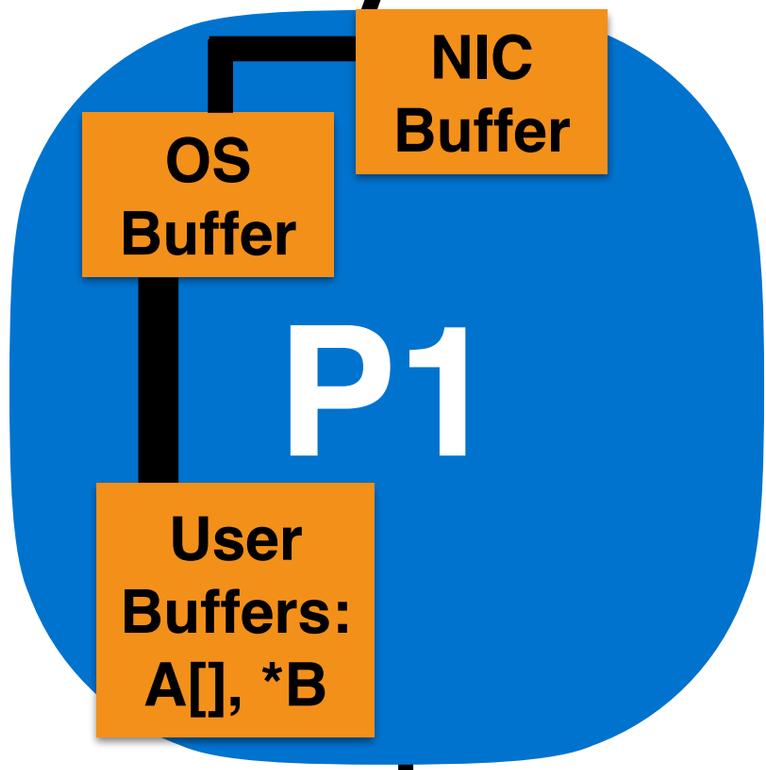
Distributed Memory





Distributed Memory





RDMA solutions also exist

Message Semantics

Message Semantics

- Lossless?

Message Semantics

- Lossless?
- Ack-based?

Message Semantics

- Lossless?
- Ack-based?
- Buffered?

Message Semantics

- Lossless?
- Ack-based?
- Buffered?
- FIFO?

Message Semantics

- Lossless?
- Ack-based?
- Buffered?
- FIFO?
- Streamed?

Message Semantics

- Lossless?
- Ack-based?
- Buffered?
- FIFO?
- Streamed?
- Point to Point vs Collective?

Message Semantics

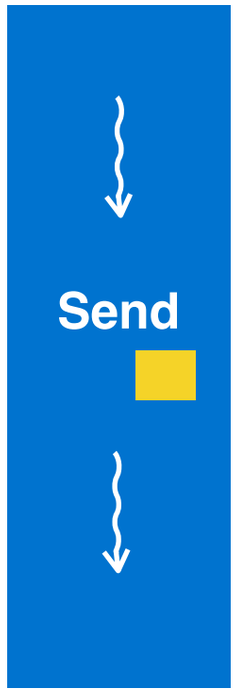
- Lossless?
- Ack-based?
- Buffered?
- FIFO?
- Streamed?
- Point to Point vs Collective?
- Addressing?

Message Semantics

- Lossless?
- Ack-based?
- Buffered?
- FIFO?
- Streamed?
- Point to Point vs Collective?
- Addressing?
- Application to application?

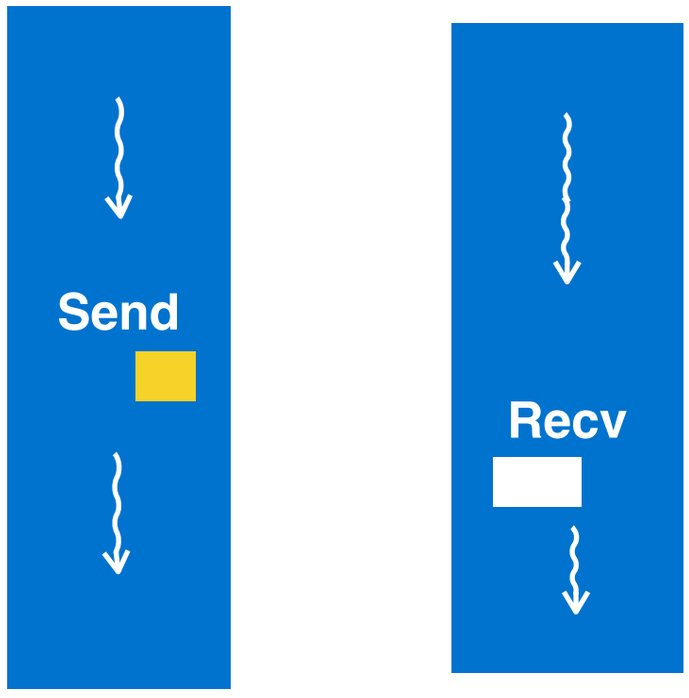
Message Semantics

- Lossless?
- Ack-based?
- Buffered?
- FIFO?
- Streamed?
- Point to Point vs Collective?
- Addressing?
- Application to application?



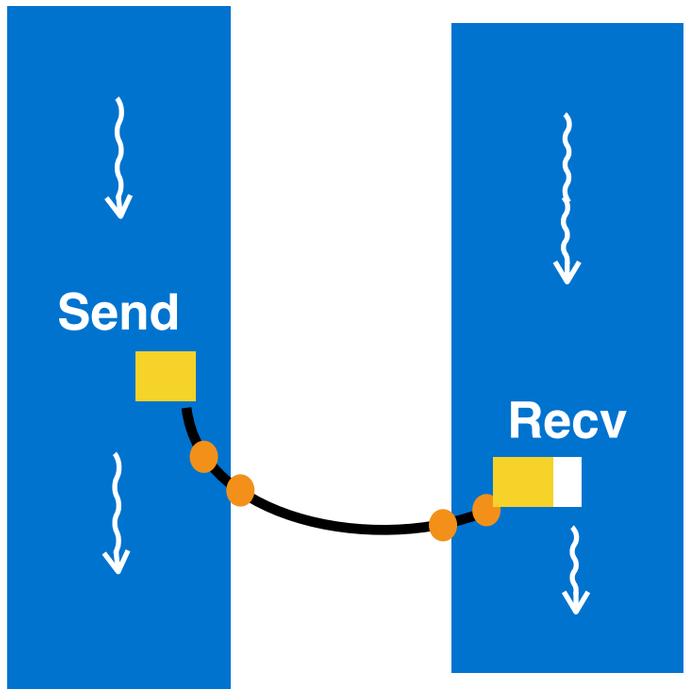
Message Semantics

- Lossless?
- Ack-based?
- Buffered?
- FIFO?
- Streamed?
- Point to Point vs Collective?
- Addressing?
- Application to application?



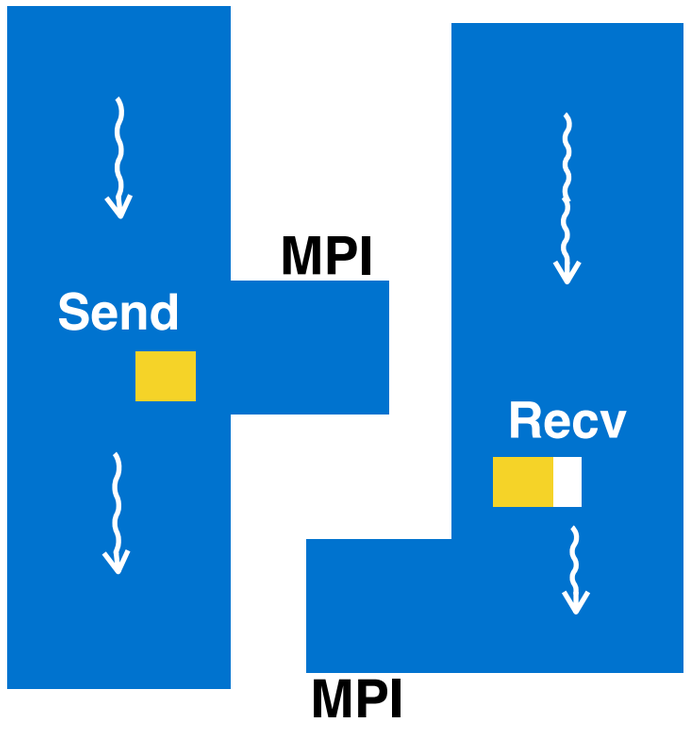
Message Semantics

- Lossless?
- Ack-based?
- Buffered?
- FIFO?
- Streamed?
- Point to Point vs Collective?
- Addressing?
- Application to application?



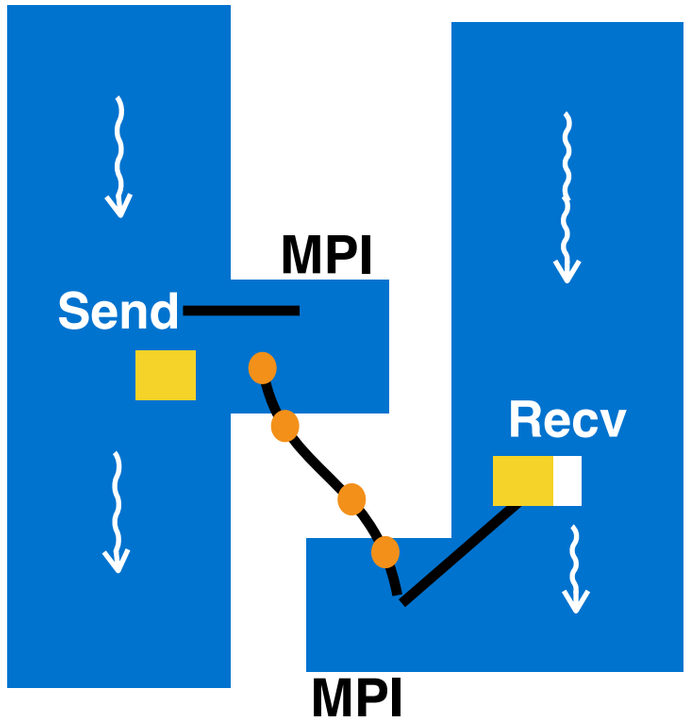
Message Semantics

- Lossless?
- Ack-based?
- Buffered?
- FIFO?
- Streamed?
- Point to Point vs Collective?
- Addressing?
- Application to application?



Message Semantics

- Lossless?
- Ack-based?
- Buffered?
- FIFO?
- Streamed?
- Point to Point vs Collective?
- Addressing?
- Application to application?



MPI: MESSAGE PASSING INTERFACE

MPI

- MPI is for inter-process communication
  - Process creation
  - Data communication
  - Synchronization
- Allows
  - Synchronous communication
  - Asynchronous communication
    - compare to shared memory
- Handles Nitty-gritty of communication and book-keeping

Process Organization

- Context
  - "Communication universe"
  - Messages across context have no 'interference'
- Groups
  - Collection of processes
  - Can create arbitrary grouping
- Communicator
  - Groups of processes that share a context
  - Notion of inter-communicator
  - Default: MPI_COMM_WORLD
- Rank
  - In the group associated with a communicator

Run Time Environment

- Launch processes
- Directory of ways to connect/communicate with processes
  – All processes need to exchange parts of this directory
- Input/Output
  – printf, scanf
  – exit status
- Control
  – Clean up resources on closure/crash

MPI Basics

- Communicator
  - Collection of processes
  - Determines scope to which messages are relative
  - Identity of process (rank) is relative to communicator
  - Scope of global communications (broadcast, etc.)
- Query:

```
MPI_Comm_size (MPI_COMM_WORLD, &p);
MPI_Comm_rank (MPI_COMM_WORLD, &id);
```

Starting and Ending

```
MPI_Init(&argc, &argv);
```

– Needed before any other MPI call

```
MPI_Finalize();
```

– Required

Send/Receive

Send/Receive

```
int MPI_Send(void* buf, int count,
  MPI_Datatype datatype, int dest,
  int tag, MPI_Comm comm)
```

Send/Receive

```
int MPI_Send(void* buf, int count,
  MPI_Datatype datatype, int dest,
  int tag, MPI_Comm comm)
```

```
void MPI::Comm::Send(const void* buf,
  int count, const MPI::Datatype&
  datatype, int dest, int tag) const
```

Send/Receive

```
int MPI_Send(void* buf, int count,
  MPI_Datatype datatype, int dest,
  int tag, MPI_Comm comm)
```

```
void MPI::Comm::Send(const void* buf,
  int count, const MPI::Datatype&
  datatype, int dest, int tag) const
```

Send/Receive

```
int MPI_Send(void* buf, int count,
  MPI_Datatype datatype, int dest,
  int tag, MPI_Comm comm)

void MPI::Comm::Send(const void* buf,
  int count, const MPI::Datatype&
  datatype, int dest, int tag) const

int MPI_Recv(void* buf, int count,
  MPI_Datatype datatype, int source,
  int tag, MPI_Comm comm, MPI_Status
  *status)
```

Send/Receive

```
int MPI_Send(void* buf, int count,
  MPI_Datatype datatype, int dest,
  int tag, MPI_Comm comm)
```

```
void MPI::Comm::Send(const void* buf,
  int count, const MPI::Datatype&
  datatype, int dest, int tag) const
```

```
int MPI_Recv(void* buf, int count,
  MPI_Datatype datatype, int source,
  int tag, MPI_Comm comm, MPI_Status
  *status)
```

Blocking calls

Send

- message contents — block of memory
- count — number of items in message
- message type — MPI TYPE of each item
- destination — rank of recipient
- tag — integer “message type”
- communicator

```
int MPI_Send(void* buf, int count, MPI_Datatype
  datatype, int dest, int tag, MPI_Comm comm)
```

Receive

- message contents — memory buffer to store received message
- count — space in buffer, overflow error if too small
- message type — type of each item
- source — sender's rank (can be wild card)
- tag — type (can be wild card)
- communicator
- status — information about message received

```
int MPI_Recv(void* buf, int count, MPI_Datatype
  datatype, int source, int tag, MPI_Comm comm,
  MPI_Status *status)
```

Example

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"     /* includes MPI library code specs */

#define MAXSIZE 100

int main(int argc, char* argv[])
{
  MPI_Init(&argc, &argv);                          // start MPI
  int numProc, myRank;
  MPI_Comm_size(MPI_COMM_WORLD, &numProc);// Group size
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank); // get my rank
  doProcessing(myRank, numProc);

  MPI_Finalize();                                  // stop MPI
}
```

Example

```
doProcessing(int myRank, int nProcs)
  {                       /* I am ID myRank of nProcs          */
   int  source;           /* rank of sender                    */
   int  dest;             /* rank of destination               */
   int  tag = 0;          /* tag to distinguish messages       */
   char mesg[MAXSIZE];/* message (other types possible) */
   int  count;            /* number of items in message        */
   MPI_Status status; /* status of message received        */
```

Example

```
if (myRank != 0){  // all other than master send to P0
    // create message and send
    sprintf(message, "Hello from %d", myRank);
    dest  = 0;
    MPI_Send(mesg, strlen(mesg)+1, MPI_CHAR,
             dest, tag, MPI_COMM_WORLD);
 }
  else{ // P0 receives from everyone else in order
    for(source = 1; source < nProcs; source++){
      if(MPI_Recv(mesg, MAXSIZE, MPI_CHAR, source,
              tag, MPI_COMM_WORLD, &status) == MPI_SUCCESS)
         printf("Received from %d: %s\n", source, mesg);
      else
         printf("Receive from %d failed\n", source);
    }
  }
}
```

Example

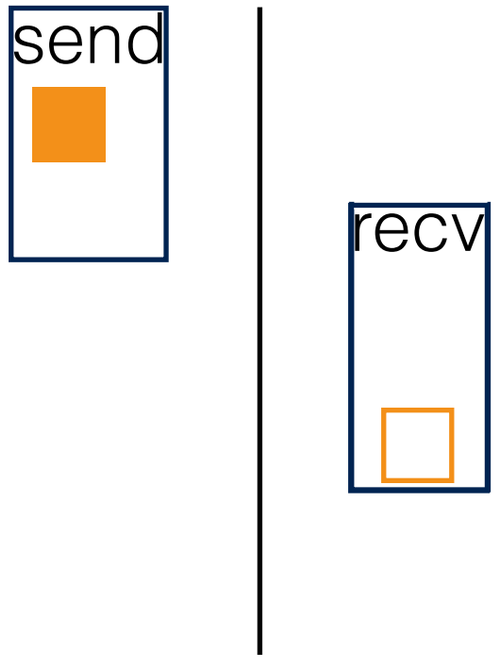
```
if (myRank != 0){  // all other than master send to P0
    // create message and send
    sprintf(message, "Hello from %d", myRank);
    dest  = 0;
    MPI_Send(mesg, strlen(mesg)+1, MPI_CHAR,
             dest, tag, MPI_COMM_WORLD);
 }
  else{ // P0 receives from everyone else in order
    for(source = 1; source < nProcs; source++){
      if(MPI_Recv(mesg, MAXSIZE, MPI_CHAR, MPI_ANY_SOURCE,
              tag, MPI_COMM_WORLD, &status) == MPI_SUCCESS)
         printf("Received from %d: %s\n", source, mesg);
      else
         printf("Receive from %d failed\n", source);
    }
  }
}
```

Send, Receive = “Synchronization”

- Fully Synchronized (Rendezvous)
  - Send and Receive complete simultaneously
    - whichever code reaches the Send/Receive first waits
  - provides synchronization point (up to network delays)
- Asynchronous
  - Sending process may proceed immediately
    - does not need to wait until message is copied to buffer
    - must check for completion before using message memory
  - Receiving process may proceed immediately
    - will not have message to use until it is received
    - must check for completion before using message

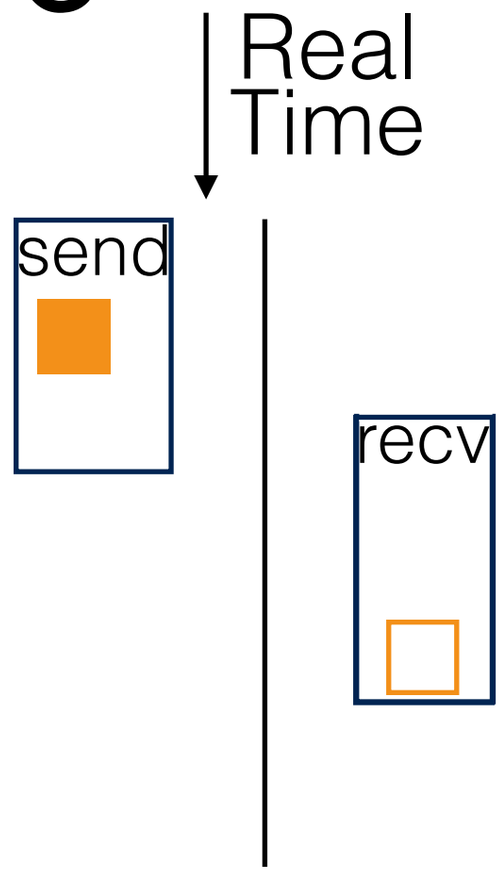
MPI Send and Receive

- `MPI_Send/MPI_Recv` is blocking
  - MPI_Recv blocks until message is received
  - MPI_Send may be synchronous or buffered
- *Standard* mode:
  - implementation dependent
  - Buffering improves performance, but requires sufficient resources
- *Buffered* mode
  - If no receive posted, system must buffer
  - User specified buffer size
- *Synchronous* mode
  - Will complete only if receive operation has accepted
  - send can be started whether or not a matching receive was posted
- *Ready* mode
  - Send may start only if receive has been posted
  - Buffer may be re-used
  - Like standard, but helps performance



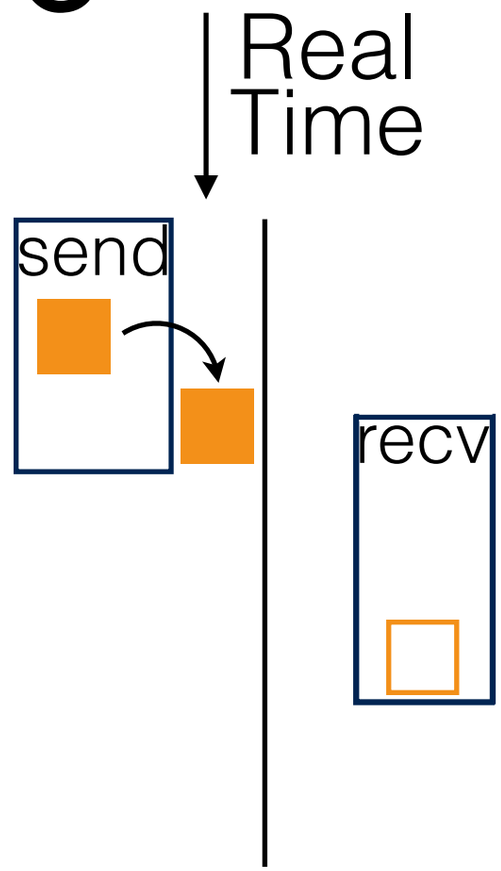
MPI Send and Receive

- `MPI_Send/MPI_Recv` is blocking
  - MPI_Recv blocks until message is received
  - MPI_Send may be synchronous or buffered
- *Standard* mode:
  - implementation dependent
  - Buffering improves performance, but requires sufficient resources
- *Buffered* mode
  - If no receive posted, system must buffer
  - User specified buffer size
- *Synchronous* mode
  - Will complete only if receive operation has accepted
  - send can be started whether or not a matching receive was posted
- *Ready* mode
  - Send may start only if receive has been posted
  - Buffer may be re-used
  - Like standard, but helps performance



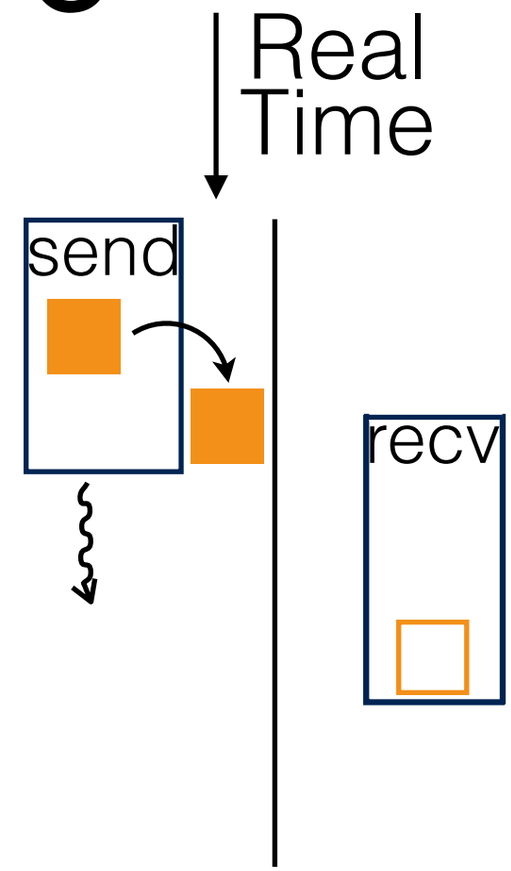
MPI Send and Receive

- `MPI_Send/MPI_Recv` is blocking
  - MPI_Recv blocks until message is received
  - MPI_Send may be synchronous or buffered
- *Standard* mode:
  - implementation dependent
  - Buffering improves performance, but requires sufficient resources
- *Buffered* mode
  - If no receive posted, system must buffer
  - User specified buffer size
- *Synchronous* mode
  - Will complete only if receive operation has accepted
  - send can be started whether or not a matching receive was posted
- *Ready* mode
  - Send may start only if receive has been posted
  - Buffer may be re-used
  - Like standard, but helps performance



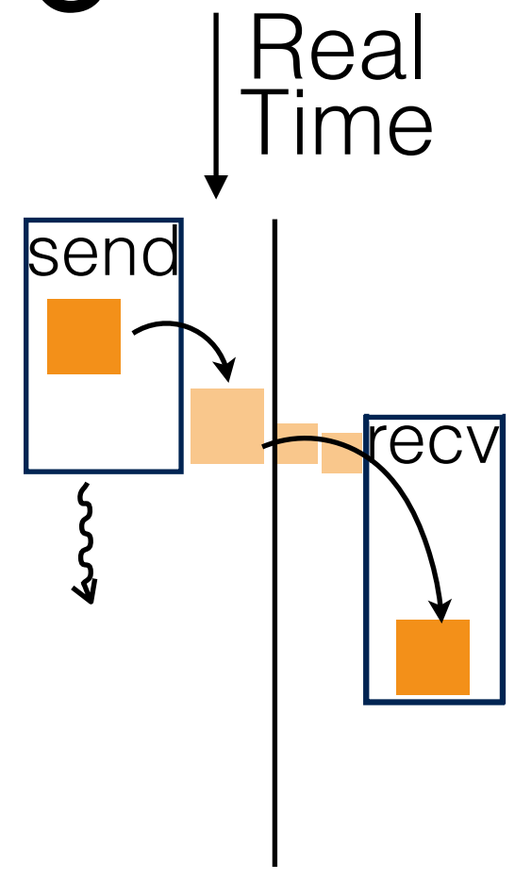
MPI Send and Receive

- `MPI_Send/MPI_Recv` is blocking
  - MPI_Recv blocks until message is received
  - MPI_Send may be synchronous or buffered
- *Standard* mode:
  - implementation dependent
  - Buffering improves performance, but requires sufficient resources
- *Buffered* mode
  - If no receive posted, system must buffer
  - User specified buffer size
- *Synchronous* mode
  - Will complete only if receive operation has accepted
  - send can be started whether or not a matching receive was posted
- *Ready* mode
  - Send may start only if receive has been posted
  - Buffer may be re-used
  - Like standard, but helps performance



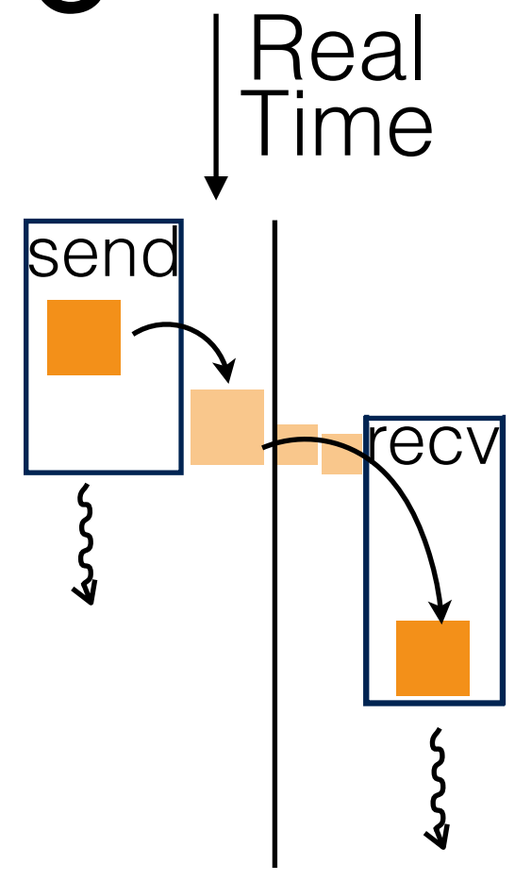
MPI Send and Receive

- `MPI_Send/MPI_Recv` is blocking
  - MPI_Recv blocks until message is received
  - MPI_Send may be synchronous or buffered
- *Standard* mode:
  - implementation dependent
  - Buffering improves performance, but requires sufficient resources
- *Buffered* mode
  - If no receive posted, system must buffer
  - User specified buffer size
- *Synchronous* mode
  - Will complete only if receive operation has accepted
  - send can be started whether or not a matching receive was posted
- *Ready* mode
  - Send may start only if receive has been posted
  - Buffer may be re-used
  - Like standard, but helps performance



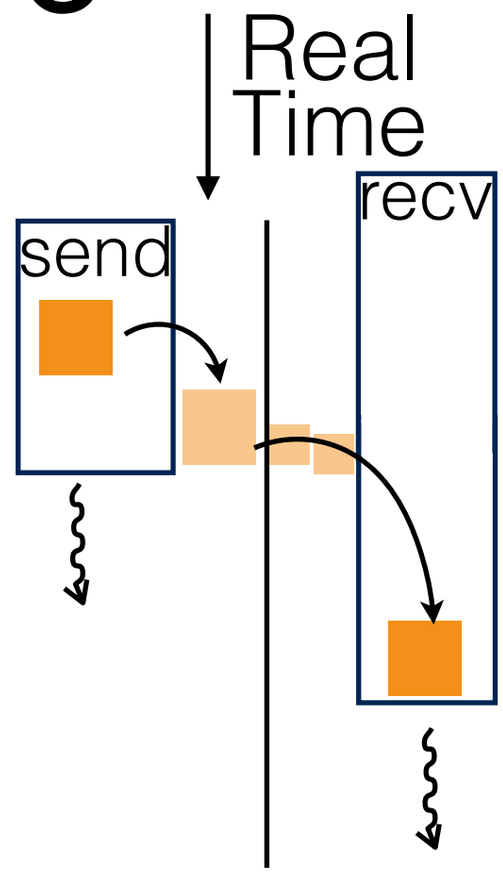
MPI Send and Receive

- `MPI_Send/MPI_Recv` is blocking
  - MPI_Recv blocks until message is received
  - MPI_Send may be synchronous or buffered
- *Standard* mode:
  - implementation dependent
  - Buffering improves performance, but requires sufficient resources
- *Buffered* mode
  - If no receive posted, system must buffer
  - User specified buffer size
- *Synchronous* mode
  - Will complete only if receive operation has accepted
  - send can be started whether or not a matching receive was posted
- *Ready* mode
  - Send may start only if receive has been posted
  - Buffer may be re-used
  - Like standard, but helps performance



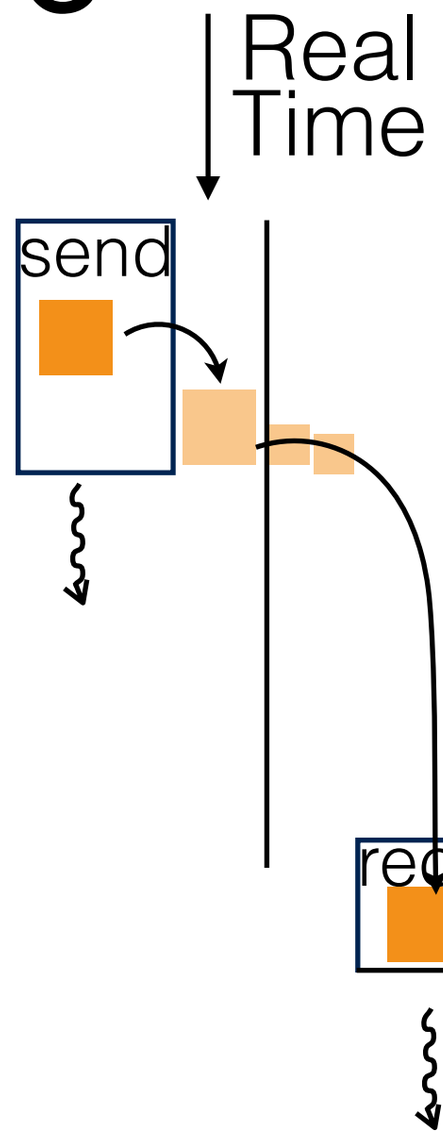
MPI Send and Receive

- `MPI_Send/MPI_Recv` is blocking
  - MPI_Recv blocks until message is received
  - MPI_Send may be synchronous or buffered
- *Standard* mode:
  - implementation dependent
  - Buffering improves performance, but requires sufficient resources
- *Buffered* mode
  - If no receive posted, system must buffer
  - User specified buffer size
- *Synchronous* mode
  - Will complete only if receive operation has accepted
  - send can be started whether or not a matching receive was posted
- *Ready* mode
  - Send may start only if receive has been posted
  - Buffer may be re-used
  - Like standard, but helps performance



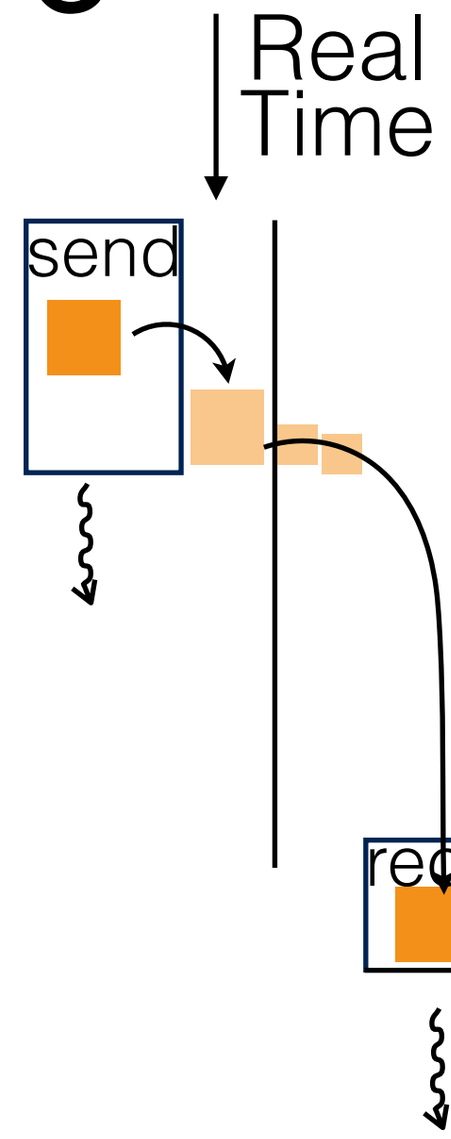
MPI Send and Receive

- `MPI_Send/MPI_Recv` is blocking
  - MPI_Recv blocks until message is received
  - MPI_Send may be synchronous or buffered
- *Standard* mode:
  - implementation dependent
  - Buffering improves performance, but requires sufficient resources
- *Buffered* mode
  - If no receive posted, system must buffer
  - User specified buffer size
- *Synchronous* mode
  - Will complete only if receive operation has accepted
  - send can be started whether or not a matching receive was posted
- *Ready* mode
  - Send may start only if receive has been posted
  - Buffer may be re-used
  - Like standard, but helps performance



MPI Send and Receive

- `MPI_Send/MPI_Recv` is blocking
  - MPI_Recv blocks until message is received
  - MPI_Send may be synchronous or buffered
- *Standard* mode:
  - implementation dependent
  - Buffering improves performance, but requires sufficient resources
- *Buffered* mode
  - If no receive posted, system must buffer
  - User specified buffer size
- *Synchronous* mode
  - Will complete only if receive operation has accepted
  - send can be started whether or not a matching receive was posted
- *Ready* mode
  - Send may start only if receive has been posted
  - Buffer may be re-used
  - Like standard, but helps performance

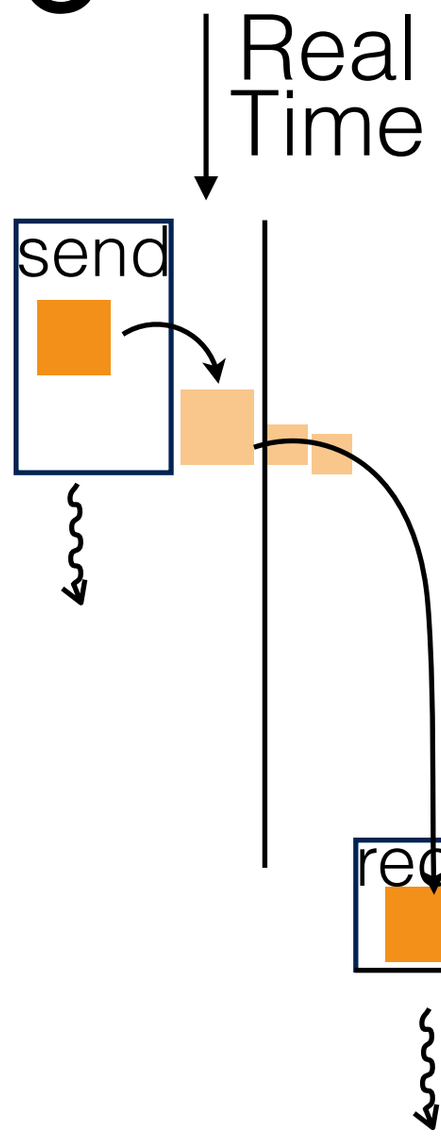


Buffered

MPI Send and Receive

- `MPI_Send/MPI_Recv` is blocking
  - MPI_Recv blocks until message is received
  - MPI_Send may be synchronous or buffered
- *Standard* mode:
  - implementation dependent
  - Buffering improves performance, but requires sufficient resources
- *Buffered* mode
  - If no receive posted, system must buffer
  - User specified buffer size
- *Synchronous* mode
  - Will complete only if receive operation has accepted
  - send can be started whether or not a matching receive was posted
- *Ready* mode
  - Send may start only if receive has been posted
  - Buffer may be re-used
  - Like standard, but helps performance

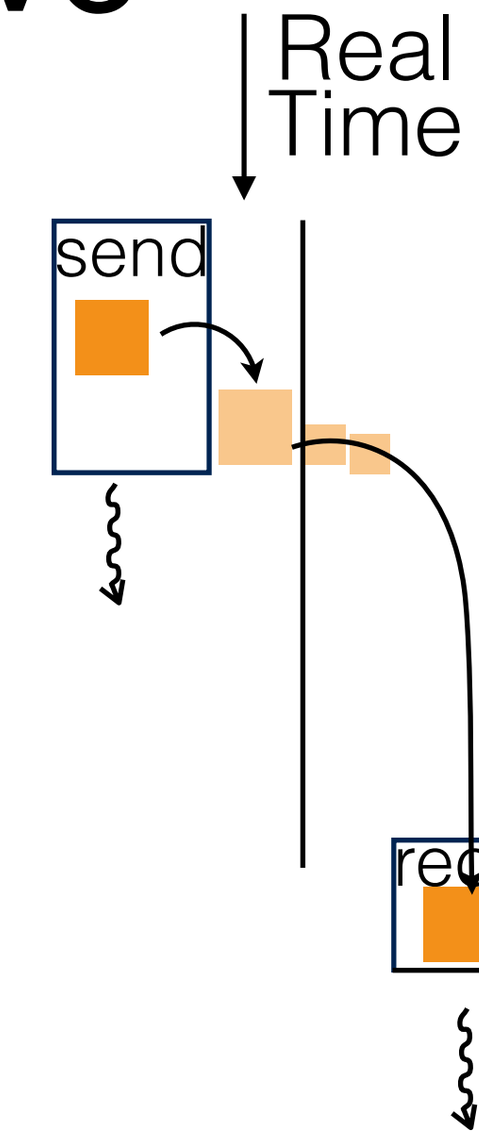
see: MPI_Buffer_attach



Buffered

MPI Send and Receive

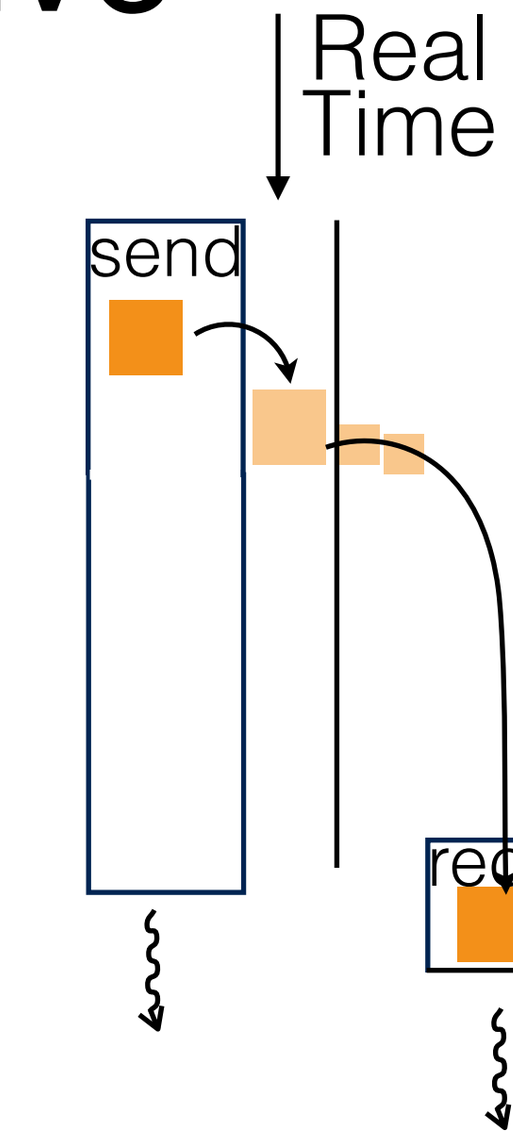
- `MPI_Send/MPI_Recv` is blocking
  - MPI_Recv blocks until message is received
  - MPI_Send may be synchronous or buffered
- *Standard* mode:
  - implementation dependent
  - Buffering improves performance, but requires sufficient resources
- *Buffered* mode
  - If no receive posted, system must buffer
  - User specified buffer size
- *Synchronous* mode
  - Will complete only if receive operation has accepted
  - send can be started whether or not a matching receive was posted
- *Ready* mode
  - Send may start only if receive has been posted
  - Buffer may be re-used
  - Like standard, but helps performance



Synchronous

MPI Send and Receive

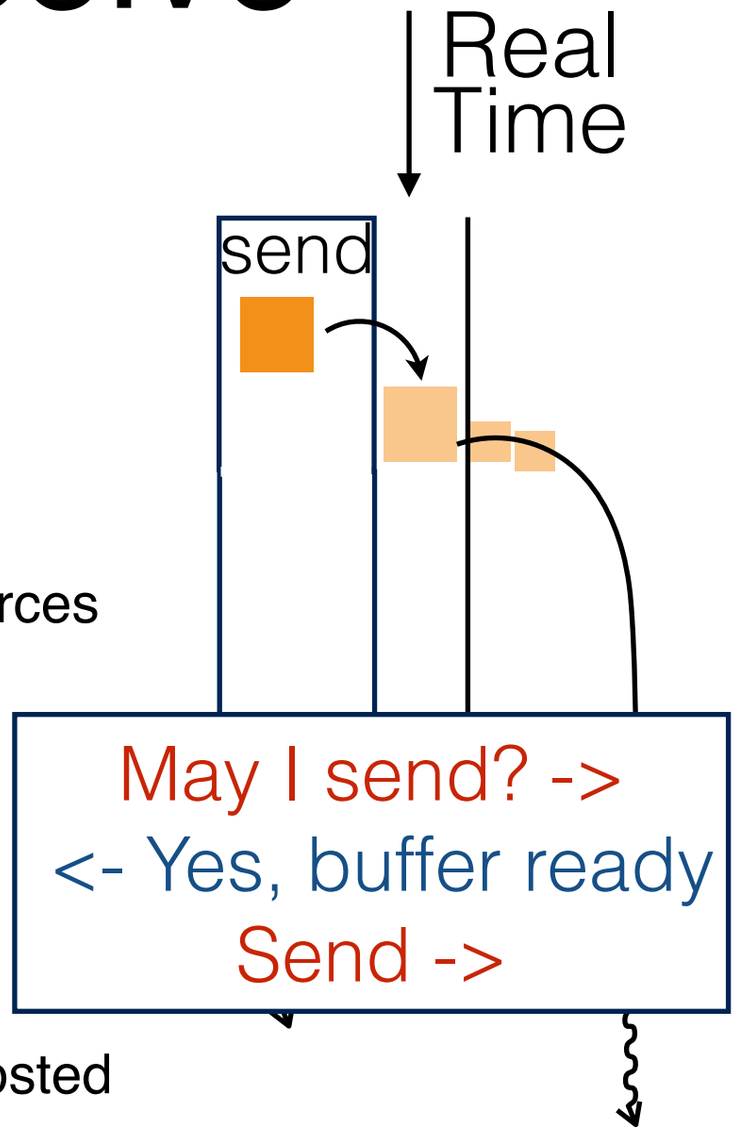
- `MPI_Send/MPI_Recv` is blocking
  - MPI_Recv blocks until message is received
  - MPI_Send may be synchronous or buffered
- *Standard* mode:
  - implementation dependent
  - Buffering improves performance, but requires sufficient resources
- *Buffered* mode
  - If no receive posted, system must buffer
  - User specified buffer size
- *Synchronous* mode
  - Will complete only if receive operation has accepted
  - send can be started whether or not a matching receive was posted
- *Ready* mode
  - Send may start only if receive has been posted
  - Buffer may be re-used
  - Like standard, but helps performance



Synchronous

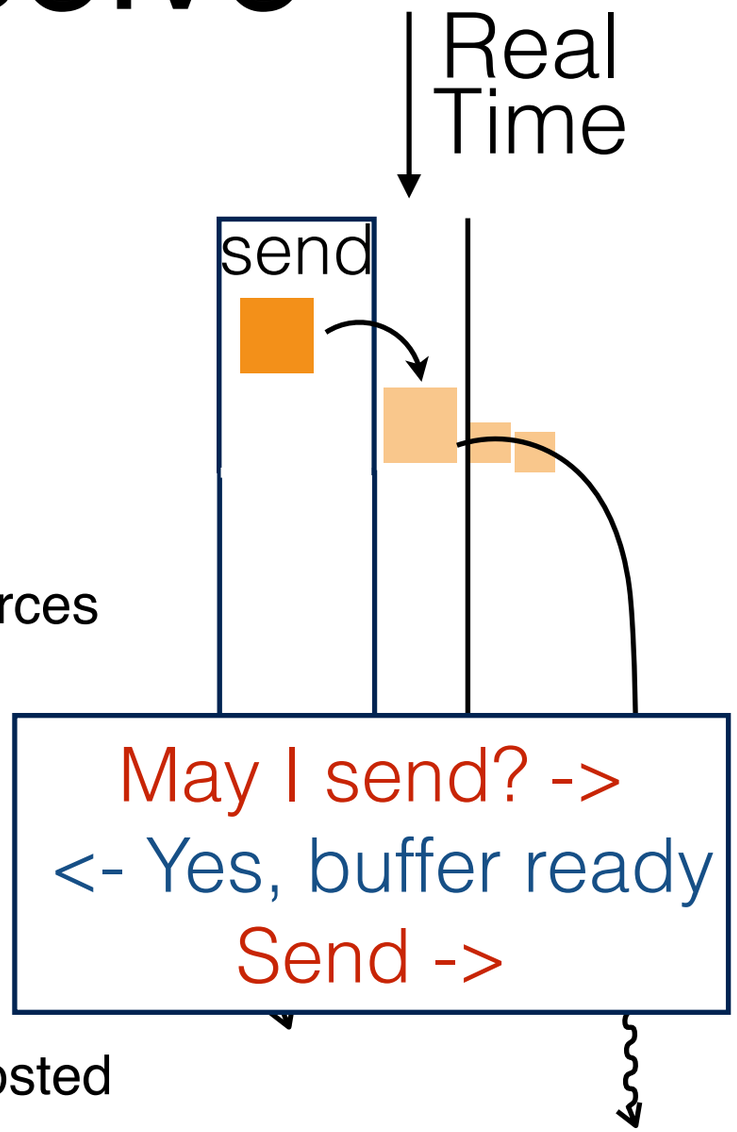
MPI Send and Receive

- `MPI_Send/MPI_Recv` is blocking
  - MPI_Recv blocks until message is received
  - MPI_Send may be synchronous or buffered
- *Standard* mode:
  - implementation dependent
  - Buffering improves performance, but requires sufficient resources
- *Buffered* mode
  - If no receive posted, system must buffer
  - User specified buffer size
- *Synchronous* mode
  - Will complete only if receive operation has accepted
  - send can be started whether or not a matching receive was posted
- *Ready* mode
  - Send may start only if receive has been posted
  - Buffer may be re-used
  - Like standard, but helps performance



MPI Send and Receive

- `MPI_Send/MPI_Recv` is blocking
  - MPI_Recv blocks until message is received
  - MPI_Send may be synchronous or buffered
- *Standard* mode:
  - implementation dependent
  - Buffering improves performance, but requires sufficient resources
- *Buffered* mode
  - If no receive posted, system must buffer
  - User specified buffer size
- *Synchronous* mode
  - Will complete only if receive operation has accepted
  - send can be started whether or not a matching receive was posted
- *Ready* mode
  - Send may start only if receive has been posted
  - Buffer may be re-used
  - Like standard, but helps performance



Function Names for Different Modes

- MPI_Send
- MPI_Bsend
- MPI_Ssend
- MPI_Rsend
- Only one MPI_Recv mode

Messages

- Messages comprise envelope and data
  - Envelope contains tag, communicator, length, source information
  - Other implementation-specific information
- Short message
  - Data sent along with envelope
- Eager
  - Message is sent proactively, assuming destination can accept into a local buffer
- Rendezvous
  - Message is not sent until destination indicates ‘ready’ to accept
  - Recipient must wait for buffer availability before sending ‘ready’ message

Messages

- Messages comprise envelope and data
  - Envelope contains tag, communicator, length, source information
  - Other implementation-specific information
- Short message
  - Data sent along with envelope
- Eager
  - Message is sent proactively, assuming destination can accept into a local buffer
- Rendezvous
  - Message is not sent until destination indicates ‘ready’ to accept
  - Recipient must wait for buffer availability before sending ‘ready’ message

NOT≡ MPI_Rsend

Messages

- Messages comprise envelope and data
  - Envelope contains tag, communicator, length, source information
  - Other implementation-specific information
- Short message
  - Data sent along with envelope
- Eager **NOT≡ MPI_Rsend**
  - Message is sent proactively, assuming destination can accept into a local buffer
- Rendezvous **NOT≡ MPI_Ssend**
  - Message is not sent until destination indicates ‘ready’ to accept
  - Recipient must wait for buffer availability before sending ‘ready’ message

Eager

- "Just send" when user calls MPI_Send
- Low synchronization overhead
- Recipient must buffer (no matching MPI_Recv call may exist).
  - If recipient NIC (+ DMA) out of space, CPU must be interrupted
  - More buffer requested and data copied
- Much unused buffer
- Smart allocation required for scalability
  - May not be implemented

Rendezvous

- Send Envelope
  - Still need to buffer envelopes
- Wait for “Ready”
- Higher synchronization overhead
  - Recipient must ensure availability of buffer
- Extra buffer copies can be eliminated
  - User buffer $\longleftrightarrow$ User buffer
- More robust due to lower buffer assumptions

Implementing Send

- Can select protocol based on message size and buffer availability
  - Short and/or eager for small messages
  - Rendezvous for longer messages
- Rsend could always use eager
  - Some Rsend implementations employ Send
- Ssend may always use rendezvous

RDMA

- Transfer between the address spaces of two processes across a network
  - May be two-sided send/receive or one-sided put/get
  - Queued directly from the user program to the NIC (HCA) without involving CPU
- Driven by the initiator of the operation
  - Peer must send its local addresses/registrations to the initiator
  - Memory used in RDMA operations are ‘pinned’ and registered with the interconnect
    - Can be expensive, Cache/Reuse registrations
- Pipe-lined RDMA
  - Initially send non RDMA eager payload
  - Recipient does registration in pieces
  - Overlap the registration with communication

MPI Message Semantics

Process may have multiple computational threads of control

- In order
  - Multi-threaded applications need to be careful about order
- Progress
  - For a matching send/Recv pair, at least one of these two operations will complete
- Fairness not guaranteed
  - A Send or a Recv may starve because all matches are satisfied by others
- Resource limitations
  - Can lead to deadlocks
- Synchronous sends rely the least on resources
  - May be used as a debugging tool

Q3: What is Output

```
// Rank 0
MPI_Send(mesg, 1024, MPI_INT, 1, 99, MPI_COMM_WORLD);
MPI_Send(mesg, 1025, MPI_INT, 2, 99, MPI_COMM_WORLD);

// Rank 1
MPI_Recv(mesg0, 2000, MPI_INT, 2, 99
                    MPI_COMM_WORLD, &status0);
MPI_Get_count(&status0, MPI_INT, &count0);
printf("Received: %d\n", count0);
MPI_Recv(mesg1, 2000, MPI_INT, 0, 99
                    MPI_COMM_WORLD, &status1);
MPI_Get_count(&status1, MPI_INT, &count1);
printf("Received: %d\n", count1);


// Rank 2
MPI_Recv(mesg, 2000, MPI_INT, 0, 99,
                   MPI_COMM_WORLD, &status0);
MPI_Send(mesg, 1026, MPI_INT, 1, 99, MPI_COMM_WORLD);
```

Asynchronous Send and Receive

- **`MPI_Isend() / MPI_Irecv()`**
  - Non-blocking: Control returns after setup
  - Blocking and non-blocking Send/Recv match
  - Still lower Send overhead if Recv has been posted
- All four modes are applicable
  - Limited impact for buffered and ready modes
- Syntax is similar to Send and Recv
  - MPI_Request* parameter is added to Isend and replaces the MPI_Status* for receive.

No blocking Send/Receive

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

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

Non-blocking calls

Detecting Completion

- **`MPI_Wait(&request, &status)`**
  - **`status`** returns status similar to **`Recv`**
  - Blocks for send until safe to reuse buffer
    - Means message was copied out, or Recv was started
  - Blocks for receive until message is in the buffer
    - Call to Send may not have returned yet
  - Request is de-allocated
- **`MPI_Test(&request, &flag, &status)`**
  - does not block
  - flag indicates whether operation is complete
  - Poll
- **`MPI_Request_get_status(&request, &flag, &status)`**
  - This variant does not de-allocate request
- **`MPI_Request_free(&request)`**
  - Free the request

Non-blocking Batch Communication

- Ordering is by the initiating call
- There is provision for

  `MPI_Waitany(count, requestsarray, &whichReady, &status)`

  – If no active request:
    - whichReady = **`MPI_UNDEFINED`**, and empty status returned
- Also:
  – **`MPI_Waitall, MPI_Testall`**
  – **`MPI_Waitsome, MPI_Testsome`**

Receiver Message Peek

- **`MPI_Probe(source, tag, comm, &status)`**
- **`MPI_Iprobe(source, tag, comm, &flag, &status)`**
  - Check information about incoming messages without actually receiving them
  - Eg., useful to know message size and allocate buffers
  - Next (matching) Recv will receive it
- **`MPI_Cancel(&request)`**
  - Request cancellation of a non-blocking request (no de-allocation)
  - Itself non-blocking: marks for cancellation and returns
  - Must still complete communication (or deallocate request) with **`MPI_Wait/MPI_Test/MPI_Request_free`**
- The operation that 'completes' the request returns status
  - One can test with **`MPI_Test_Cancelled(&status, &flag)`**

Persistent Send/Recv

```
MPI_Send_init(buf, count, datatype,
dest, tag, comm, &request);
MPI_Start(&request);
```

- MPI_Start is non-blocking
  - blocking versions do not exist (but see MPI_Wait)
- There is also MP_Start_all
  - And **`MPI_Recv_init`**
  - And **`MPI_Bsend_init`** etc.
- Reduces Process interaction time with the Communication system

Send and Recv

```
MPI_Sendrecv(sendbuf, sendcount,
sendDataType, dest, sendtag,
recvbuf, recvcount, recvtype, source,
recvtag, comm, &status)
```

- Does both
- Semantics:
  - Fork, Send and Recv, Join
- Non-blocking
  - Blocking variant: **`MPI_Sendrecv_replace`**

Review Basics

- Send - Recv is point-to-point
  - Can Recv from any source using **MPI_ANY_SOURCE**
- Buffer in Recv must contain space for entire message
  - Count parameter is the capacity of buffer
  - Can query the actual count received, e.g.,

```
MPI_Get_count(&status, MPI_CHAR, &count); // int count
```

- Count parameter in Send determines the number
- Type parameter determines the exact number of bytes
  - Must use MPI_Datatype // e.g. MPI_INIT is MPI_Datatype so is MPI_CHAR
- Integer tag to distinguish message streams
  - Can Recv any stream using **MPI_ANY_TAG**
- Variants are: Buffered, Synchronous, Ready
  - Only one Recv variant
  - Corresponding, Non-blocking variants

Simple Example - II

```
int rank, size, st_source, st_tag, st_count;
MPI_Status status;
double data[10];

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
int dest = size - 1

if (rank == 0) {
    for(i=0; i<10; i++)
       data[i] = i
    MPI_Send(data, 10, MPI_DOUBLE, dest, 2001, MPI_COMM_WORLD);
 } else if (rank == dest) {
    MPI_Recv(data, 10, MPI_DOUBLE, MPI_ANY_SOURCE,
                                MPI_ANY_TAG, MPI_COMM_WORLD,
                &status);
    MPI_Get_count(&status, MPI_DOUBLE, &st_count);
     st_source = status.MPI_SOURCE;
     st_tag = stat.MPI_TAG;
}
MPI_Finalize();
```

Collective MPI Communication

- MPI_Barrier
  - Barrier synchronization across all members of a group
- MPI_Bcast
  - Broadcast from one member to all members of a group
- MPI_Scatter, MPI_Gather, MPI_Allgather
  - Gather data from all members of a group to one
- MPI_Alltoall
  - complete exchange or all-to-all
- MPI_Allreduce, MPI_Reduce
  - Reduction operations
- MPI_Reduce_Scatter
  - Combined reduction and scatter operation
- MPI_Scan, MPI_Exscan
  - Prefix

Barrier

- Synchronization of the calling processes
  - the call blocks until all of the processes in the group have *called* Barrier
  - No time guarantee on their exit

```
MPI_Barrier(comm);
```

Barrier

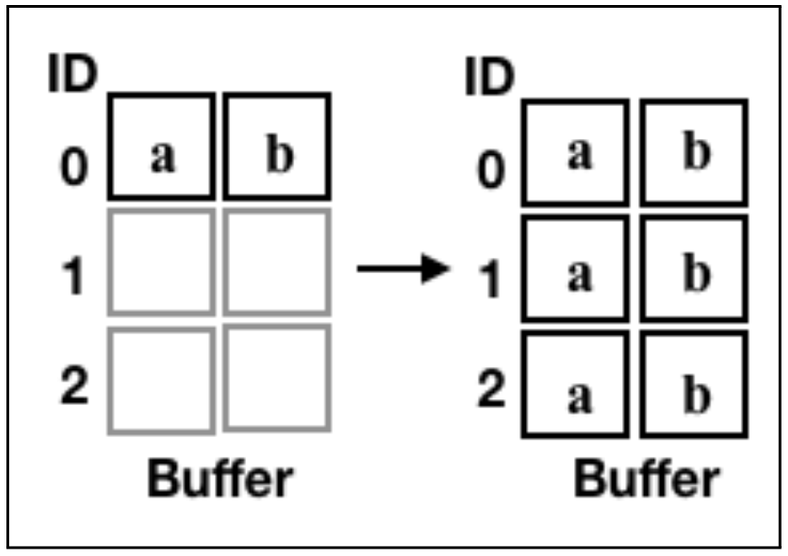
- Synchronization of the calling processes
  - the call blocks until all of the processes in the group have *called* Barrier
  - No time guarantee on their exit

```
MPI_Barrier(comm);
```

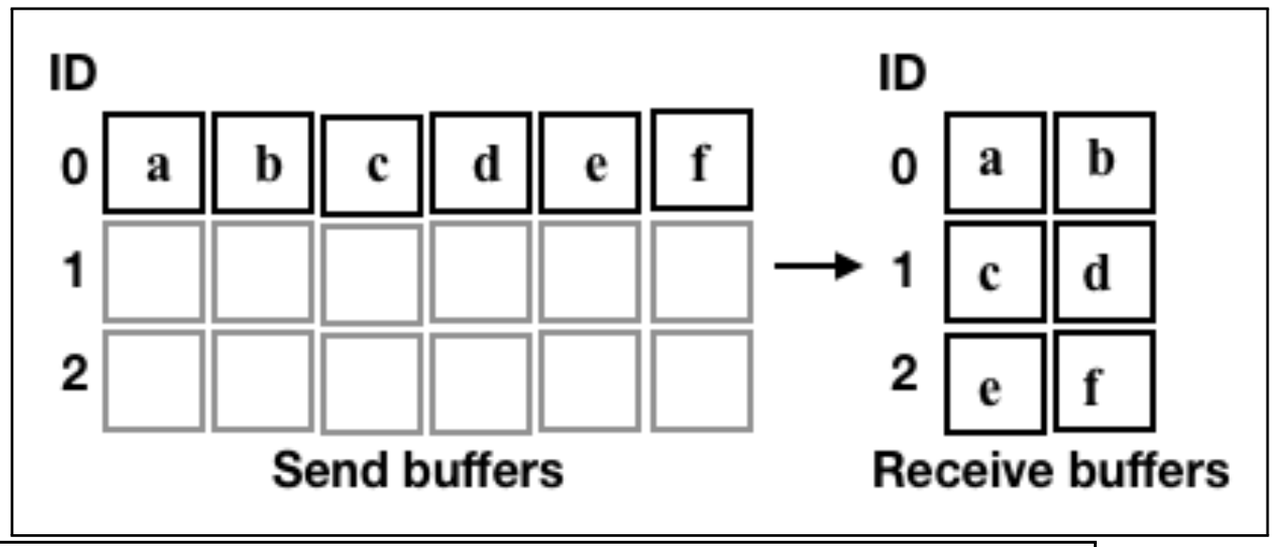
Avoid Using

Collective Communication

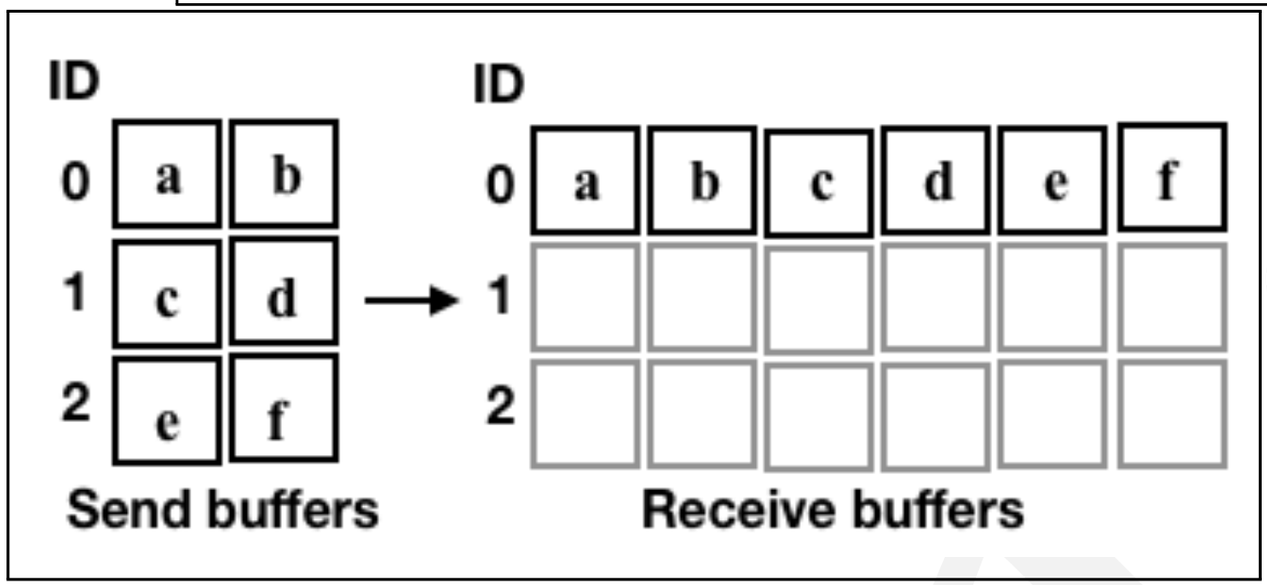
Broadcast



Scatter

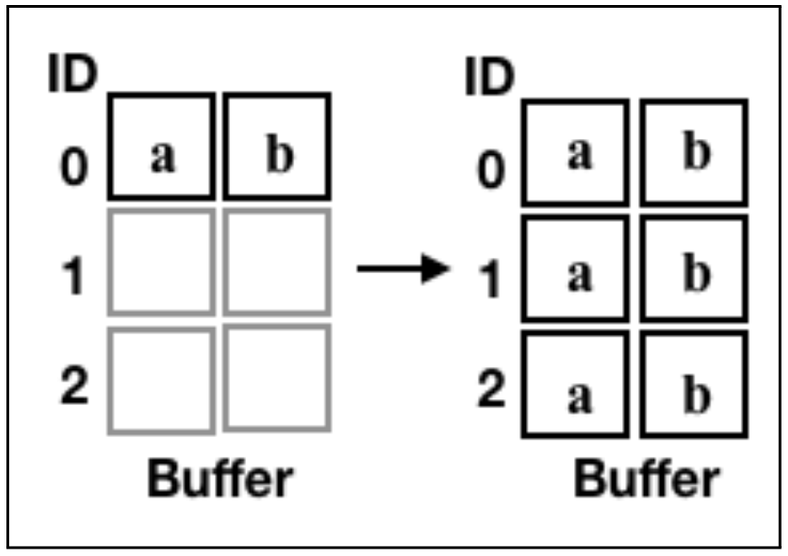


Gather

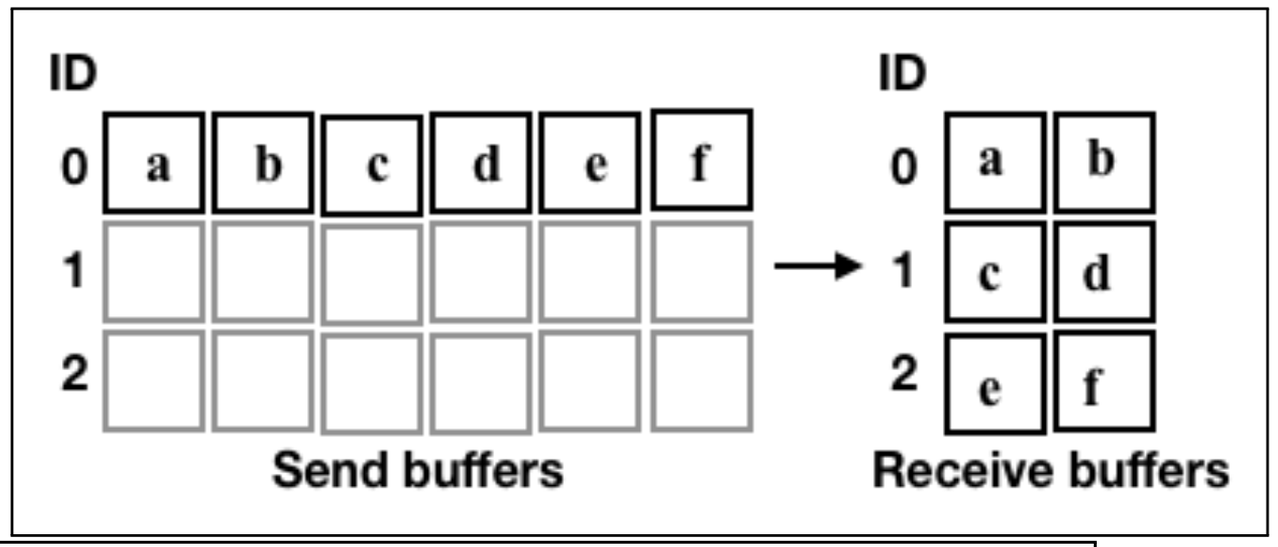


Collective Communication

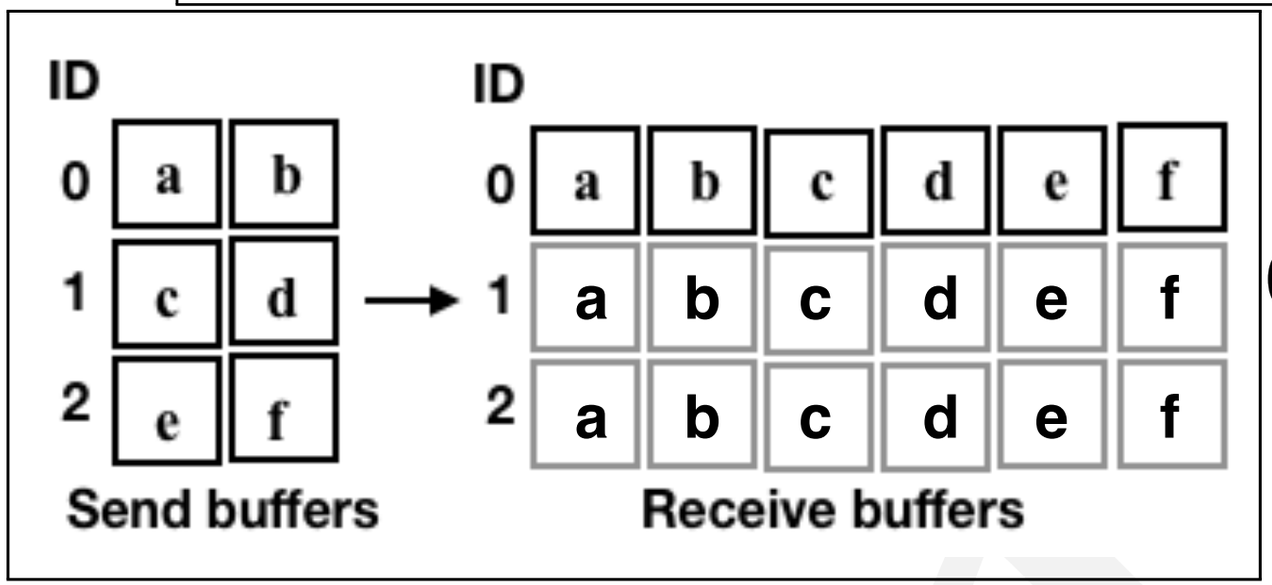
Broadcast



Scatter

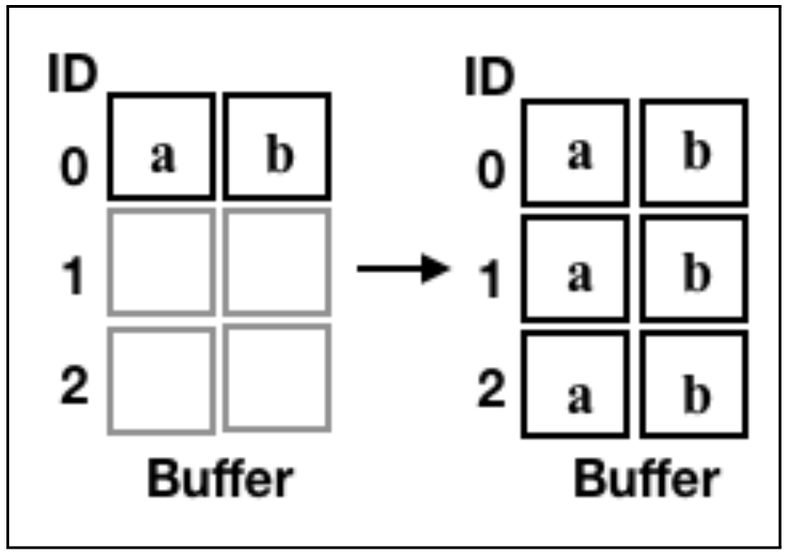


All Gather

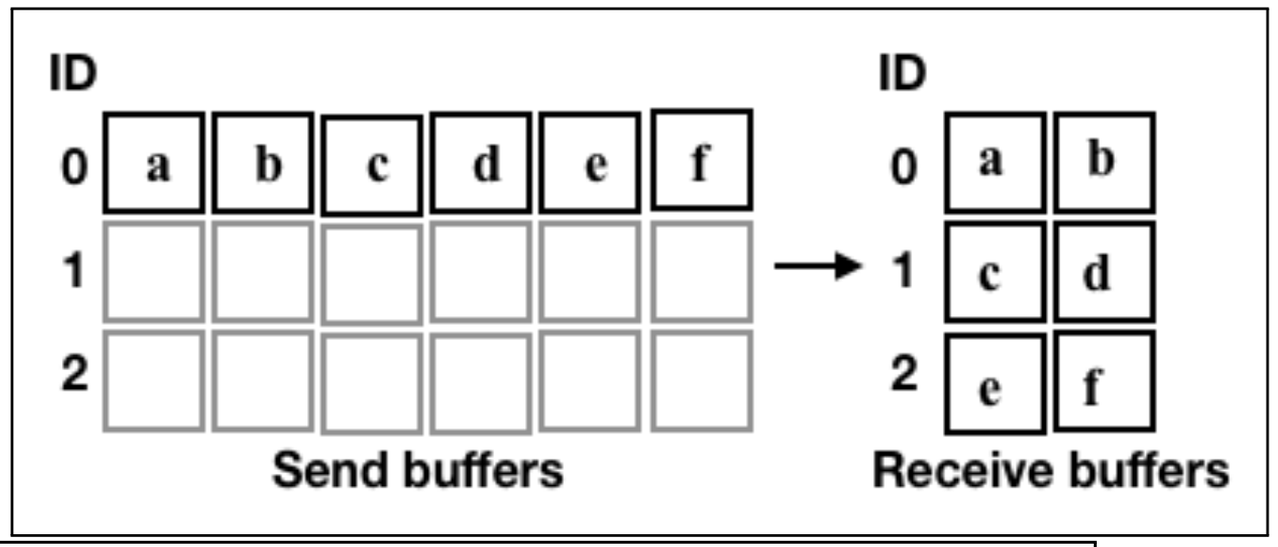


Collective Communication

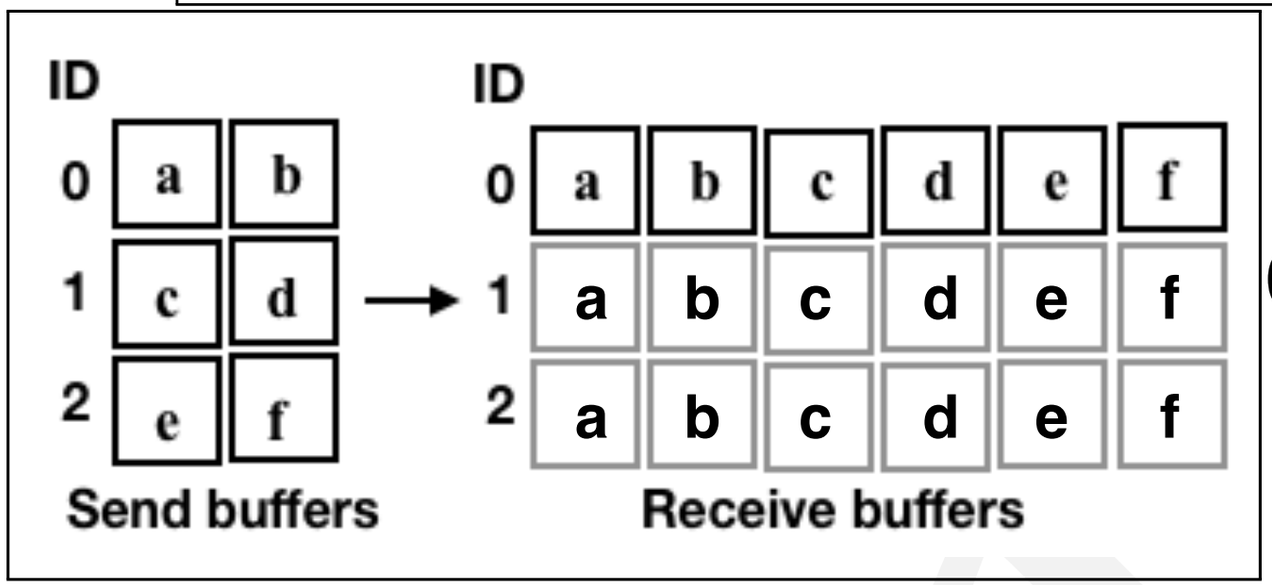
Broadcast



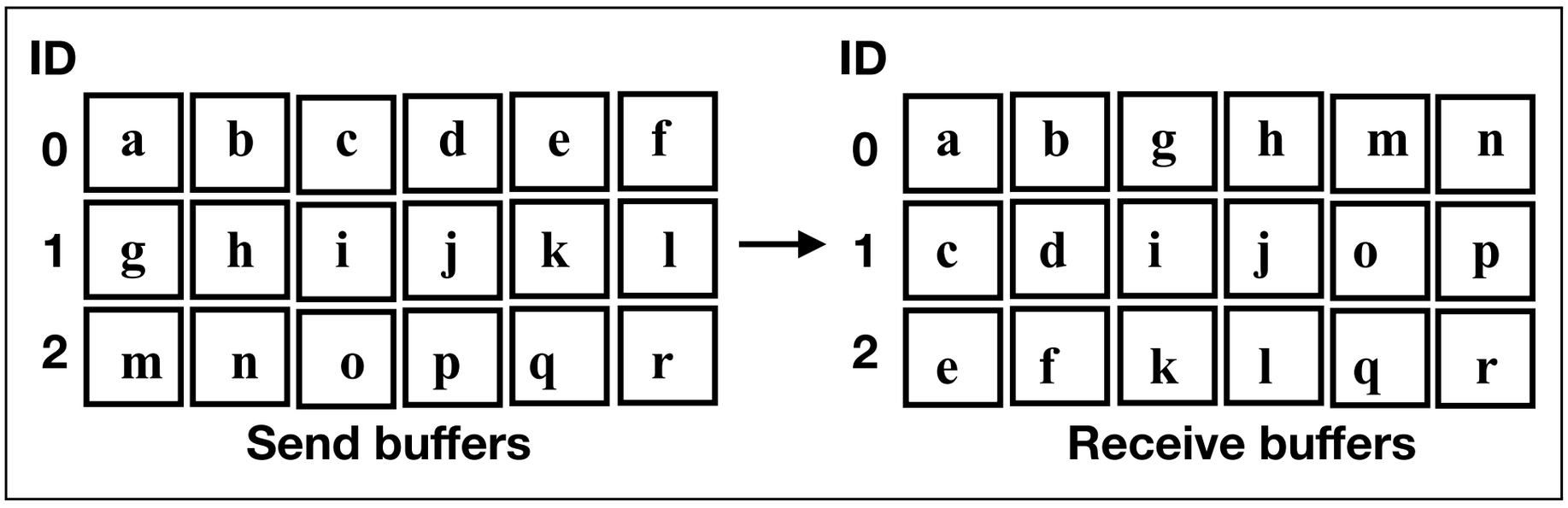
Scatter



All Gather



All to all



Broadcast

```
MPI_Bcast(mesg, count, type, root, comm);
```

`mesg`	pointer to message buffer
`count`	number of items sent
`type`	type of item sent
`root`	sending processor

- All participants must call
- count and type should be compatible everywhere
- Can broadcast on “inter-communicators” also

Broadcast

```
MPI_Bcast(mesg, count, type, root, comm);
```

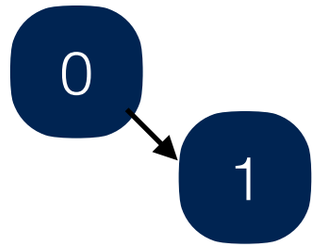
`mesg`	pointer to message buffer
`count`	number of items sent
`type`	type of item sent
`root`	sending processor

- All participants must call
- count and type should be compatible everywhere
- Can broadcast on “inter-communicators” also

Broadcast

```
MPI_Bcast(mesg, count, type, root, comm);
```

`mesg`	pointer to message buffer
`count`	number of items sent
`type`	type of item sent
`root`	sending processor

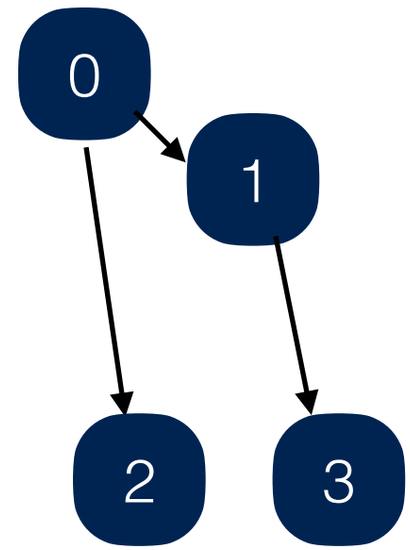


- All participants must call
- count and type should be compatible everywhere
- Can broadcast on “inter-communicators” also

Broadcast

```
MPI_Bcast(mesg, count, type, root, comm);
```

`mesg`	pointer to message buffer
`count`	number of items sent
`type`	type of item sent
`root`	sending processor

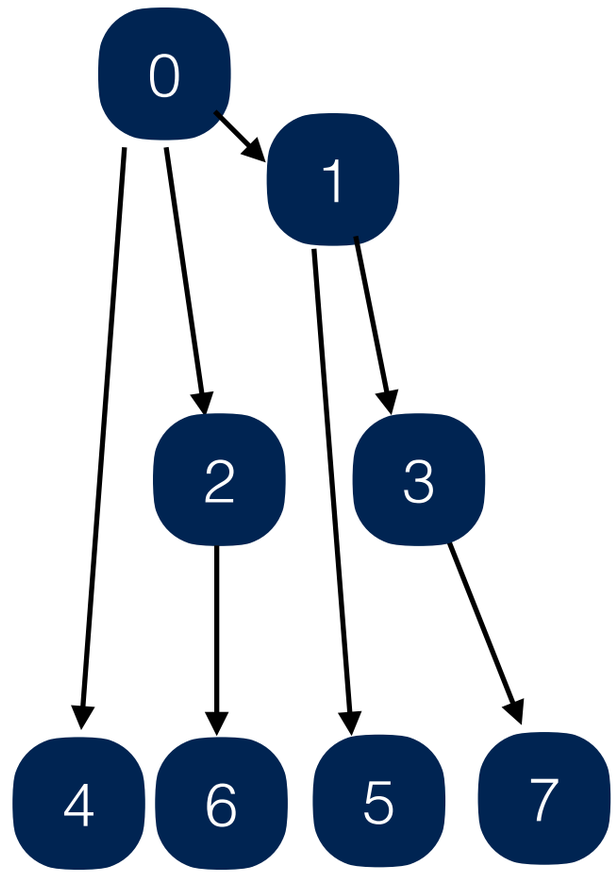


- All participants must call
- count and type should be compatible everywhere
- Can broadcast on “inter-communicators” also

Broadcast

```
MPI_Bcast(mesg, count, type, root, comm);
```

`mesg`	pointer to message buffer
`count`	number of items sent
`type`	type of item sent
`root`	sending processor

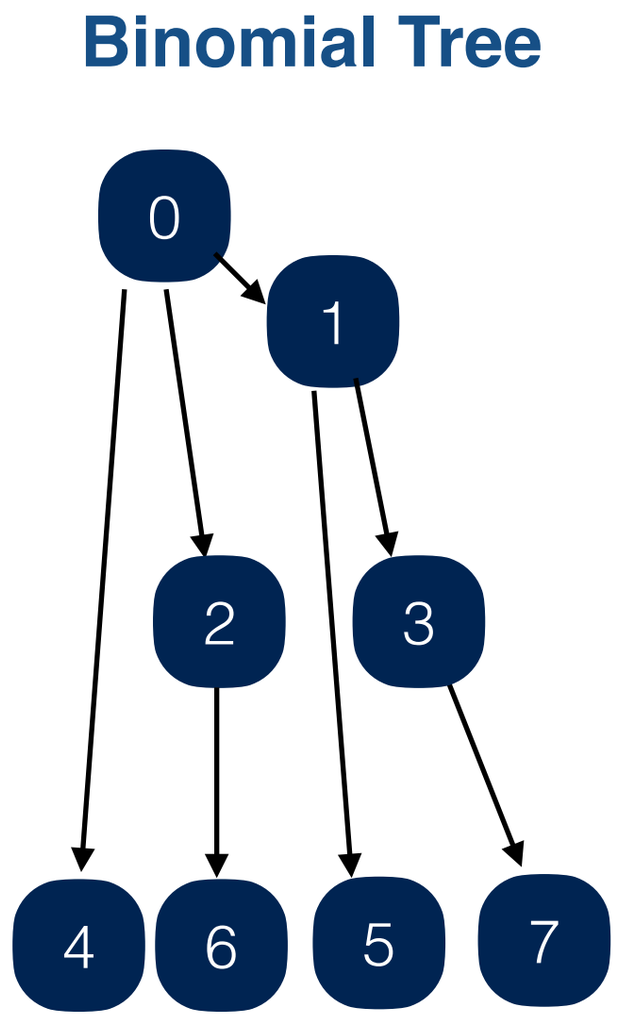


- All participants must call
- count and type should be compatible everywhere
- Can broadcast on “inter-communicators” also

Broadcast

```
MPI_Bcast(mesg, count, type, root, comm);
```

`mesg`	pointer to message buffer
`count`	number of items sent
`type`	type of item sent
`root`	sending processor

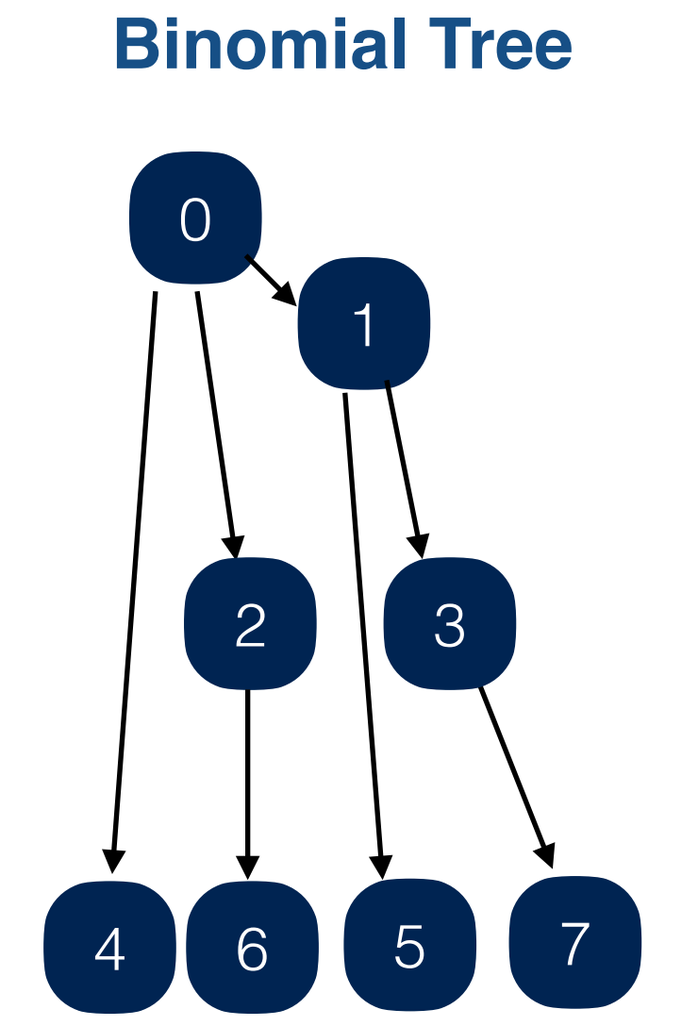


- All participants must call
- count and type should be compatible everywhere
- Can broadcast on “inter-communicators” also

Broadcast

```
MPI_Bcast(mesg, count, type, root, comm);
```

`mesg`	pointer to message buffer
`count`	number of items sent
`type`	type of item sent
`root`	sending processor



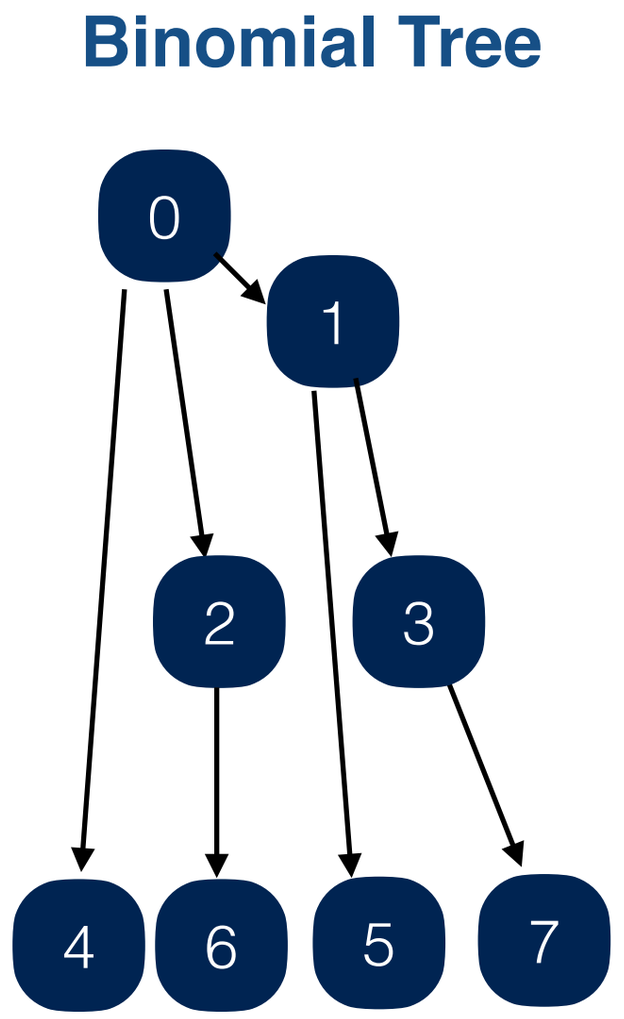
- All participants must call
- count and type should be compatible everywhere
- Can broadcast on “inter-communicators” also

Hardware multicast

Broadcast

```
MPI_Bcast(mesg, count, type, root, comm);
```

`mesg`	pointer to message buffer
`count`	number of items sent
`type`	type of item sent
`root`	sending processor



- All participants must call
- count and type should be compatible everywhere
- Can broadcast on “inter-communicators” also

Hardware multicast

+ Local updates

Broadcast

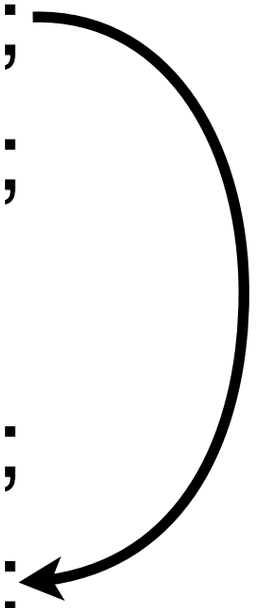
- Broadcast: one sender, many receivers
- Includes all processes in communicator
  - all processes must make a call to MPI_Bcast
  - Must agree on sender/root
- Broadcast does not require synchronization
  - Call may return before other have called
  - Some implementations may incur synchronization cost
  - Different from MPI_Barrier(communicator)

What might be wrong?

- Thread 0:
  - MPI_Bcast(buf1, count, type, 0, comm01);
  - MPI_Bcast(buf2, count, type, 2, comm20);
- Thread 1:
  - MPI_Bcast(buf1, count, type, 1, comm12);
  - MPI_Bcast(buf2, count, type, 0, comm01);
- Thread 2:
  - MPI_Bcast(buf1, count, type, 2, comm20);
  - MPI_Bcast(buf2, count, type, 1, comm12);

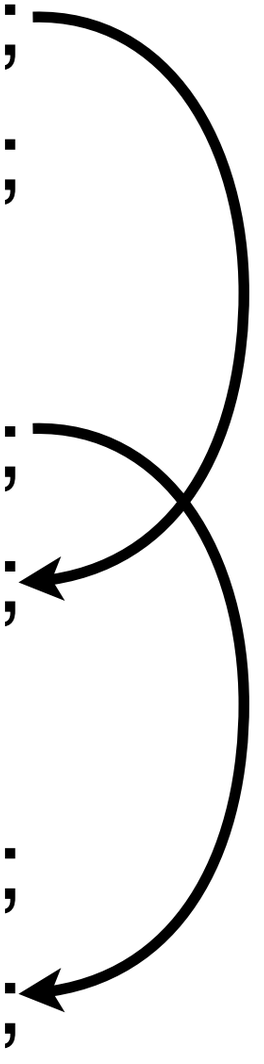
What might be wrong?

- Thread 0:
  - MPI_Bcast(buf1, count, type, 0, comm01);
  - MPI_Bcast(buf2, count, type, 2, comm20);
- Thread 1:
  - MPI_Bcast(buf1, count, type, 1, comm12);
  - MPI_Bcast(buf2, count, type, 0, comm01);
- Thread 2:
  - MPI_Bcast(buf1, count, type, 2, comm20);
  - MPI_Bcast(buf2, count, type, 1, comm12);



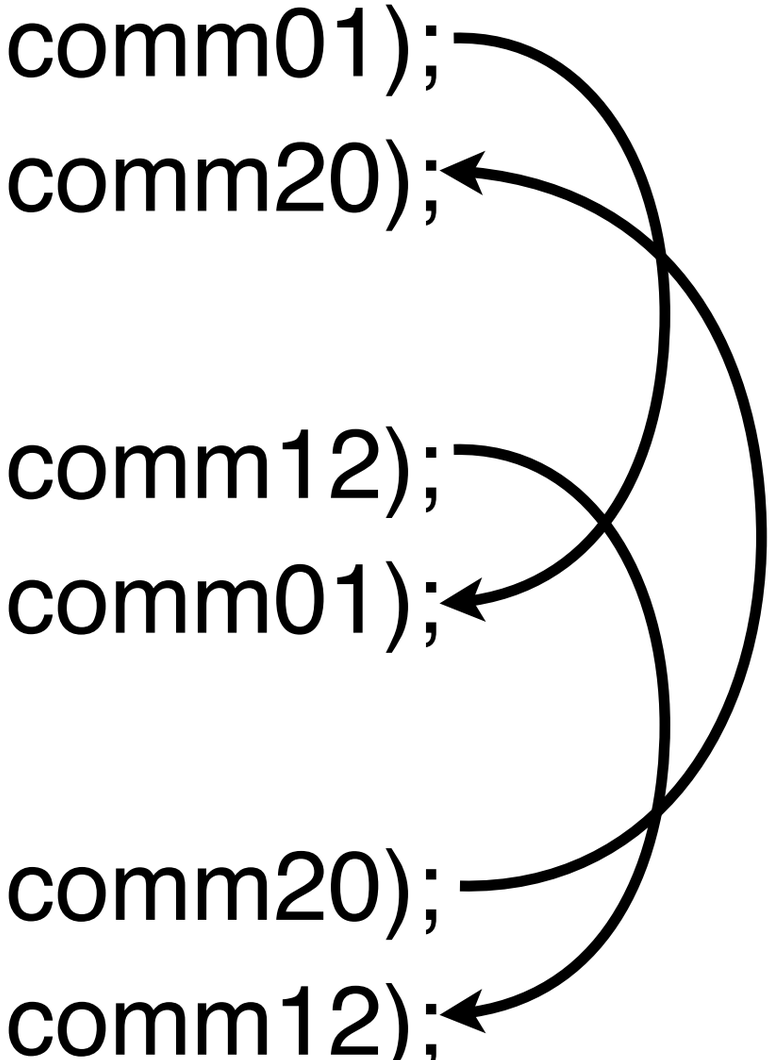
What might be wrong?

- Thread 0:
  - MPI_Bcast(buf1, count, type, 0, comm01);
  - MPI_Bcast(buf2, count, type, 2, comm20);
- Thread 1:
  - MPI_Bcast(buf1, count, type, 1, comm12);
  - MPI_Bcast(buf2, count, type, 0, comm01);
- Thread 2:
  - MPI_Bcast(buf1, count, type, 2, comm20);
  - MPI_Bcast(buf2, count, type, 1, comm12);



What might be wrong?

- Thread 0:
  - MPI_Bcast(buf1, count, type, 0, comm01);
  - MPI_Bcast(buf2, count, type, 2, comm20);
- Thread 1:
  - MPI_Bcast(buf1, count, type, 1, comm12);
  - MPI_Bcast(buf2, count, type, 0, comm01);
- Thread 2:
  - MPI_Bcast(buf1, count, type, 2, comm20);
  - MPI_Bcast(buf2, count, type, 1, comm12);

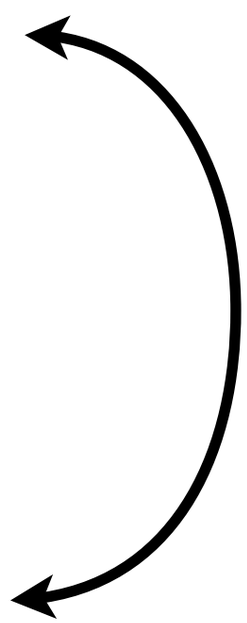


This too is wrong

- Thread 0:
  - MPI_Bcast(buf1, count, type, 0, comm);
  - MPI_Bcast(buf2, count, type, 1, comm);
- Thread 1:
  - MPI_Bcast(buf1, count, type, 1, comm);
  - MPI_Bcast(buf2, count, type, 0, comm);

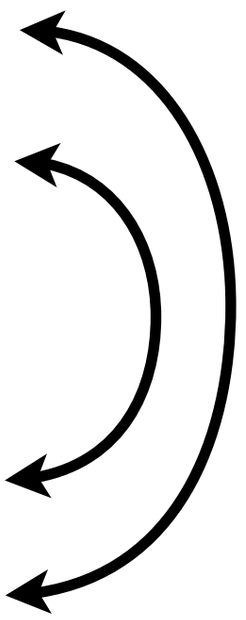
This too is wrong

- Thread 0:
  - MPI_Bcast(buf1, count, type, 0, comm);
  - MPI_Bcast(buf2, count, type, 1, comm);
- Thread 1:
  - MPI_Bcast(buf1, count, type, 1, comm);
  - MPI_Bcast(buf2, count, type, 0, comm);



This too is wrong

- Thread 0:
  - MPI_Bcast(buf1, count, type, 0, comm);
  - MPI_Bcast(buf2, count, type, 1, comm);
- Thread 1:
  - MPI_Bcast(buf1, count, type, 1, comm);
  - MPI_Bcast(buf2, count, type, 0, comm);



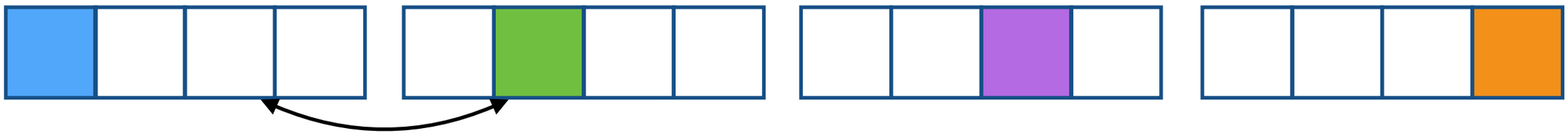
MPI_Gather

```
MPI_Gather(sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, root, comm);
```

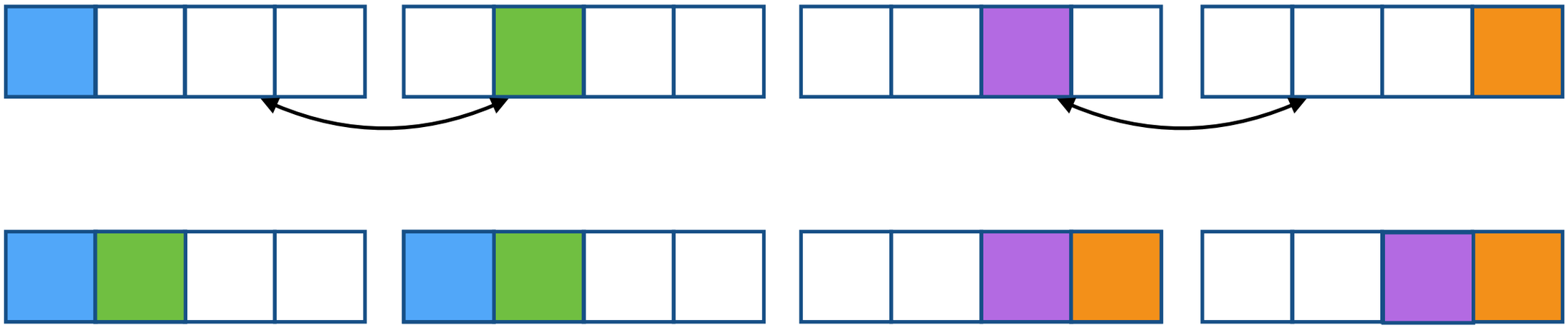
- Similar to non-roots sending:
  - MPI_Send(sendbuf, sendcount, sendtype, root, ...),
- and the root receiving n times:
  - MPI_Recv(recvbuf + i * recvcount *extent(recvtype), recvcount, recvtype, i, ...)
- **`MPI_Gatherv`** allows different size data to be gathered
- **`MPI_Allgather`** has No root, all nodes get result

MPI_Gather

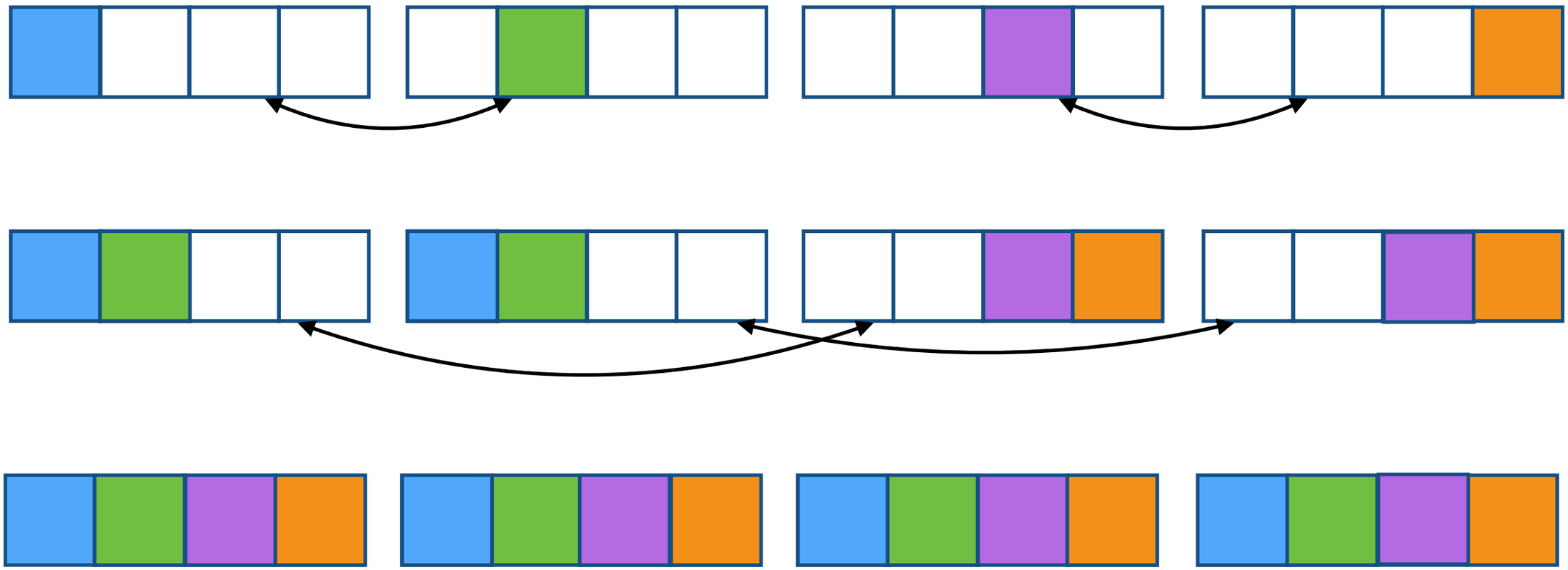
MPI_Gather



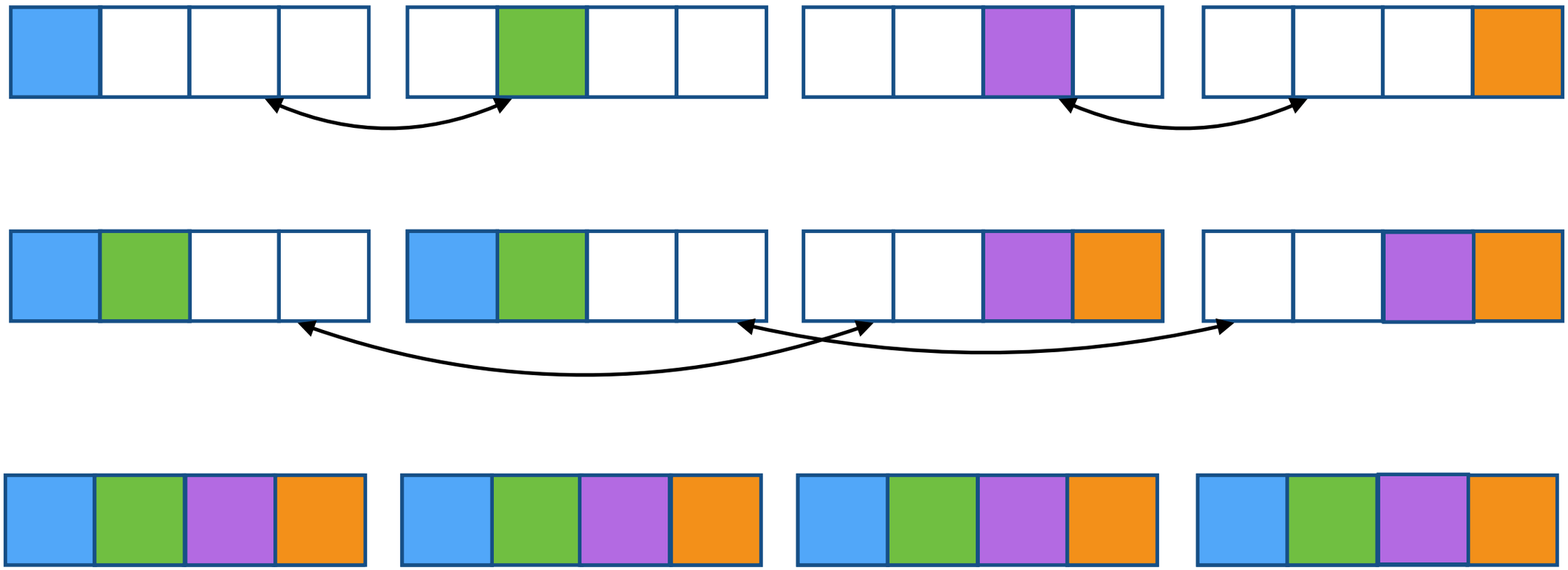
MPI_Gather



MPI_Gather

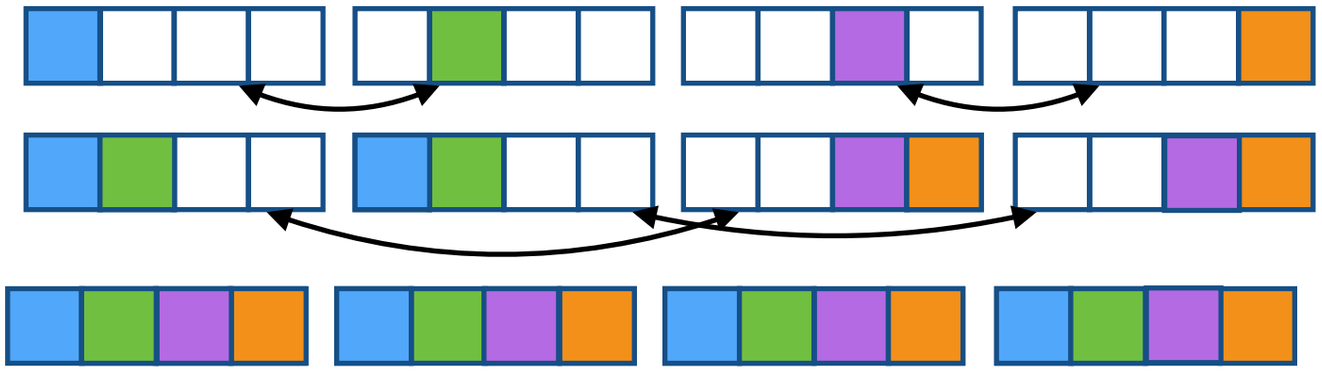


MPI_Gather

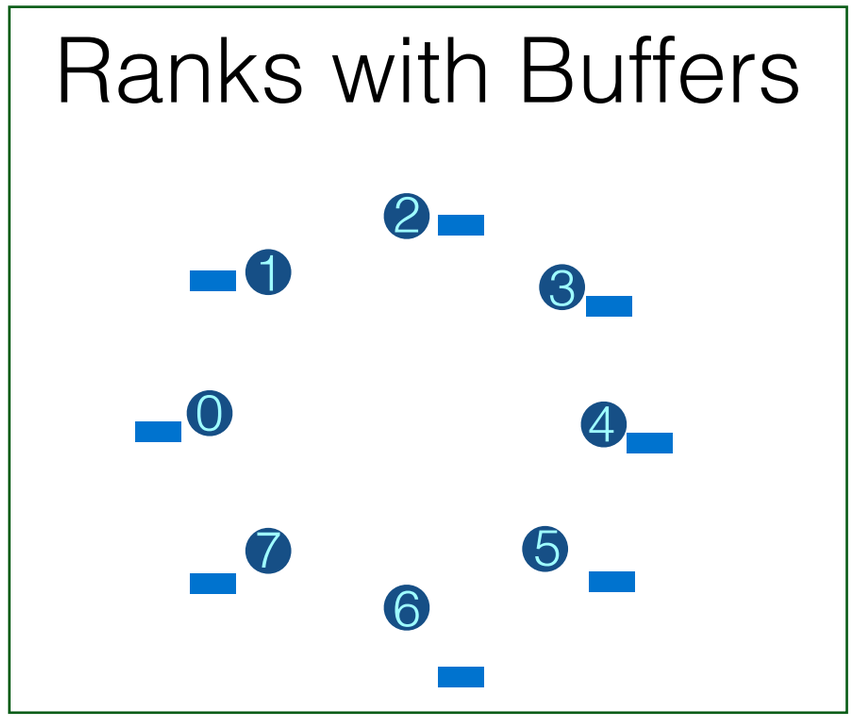


Recursive doubling

MPI_Alltoall



Recursive doubling

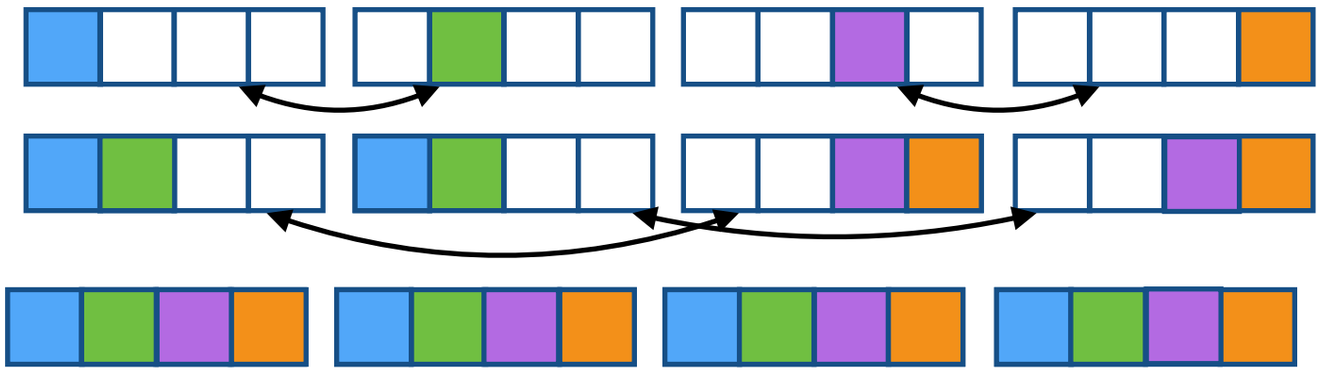



```
MPI_Alltoall(sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, comm);
```

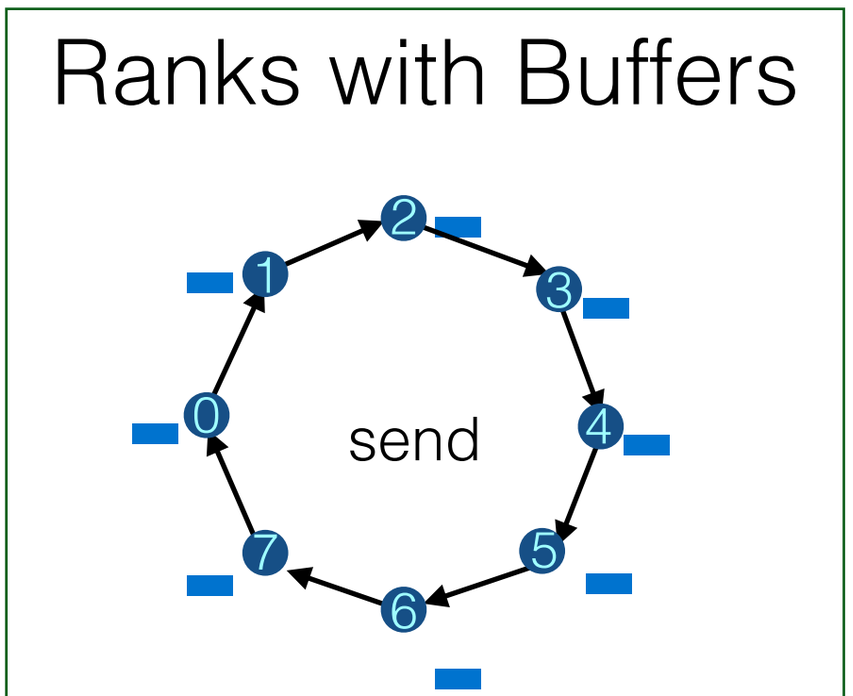
Similar to:

```
for (i = 0, i < n; i++)
        MPI_Send(sendbuf + i*sendcount*extent(sendtype), sendcount, sendtype, i, ...);
for (i = 0, i < n; i++)
        MPI_Recv(recvbuf + i*recvcount*extent(recvtype), recvcount, recvtype, i, ...,);
```

MPI_Alltoall



Recursive doubling

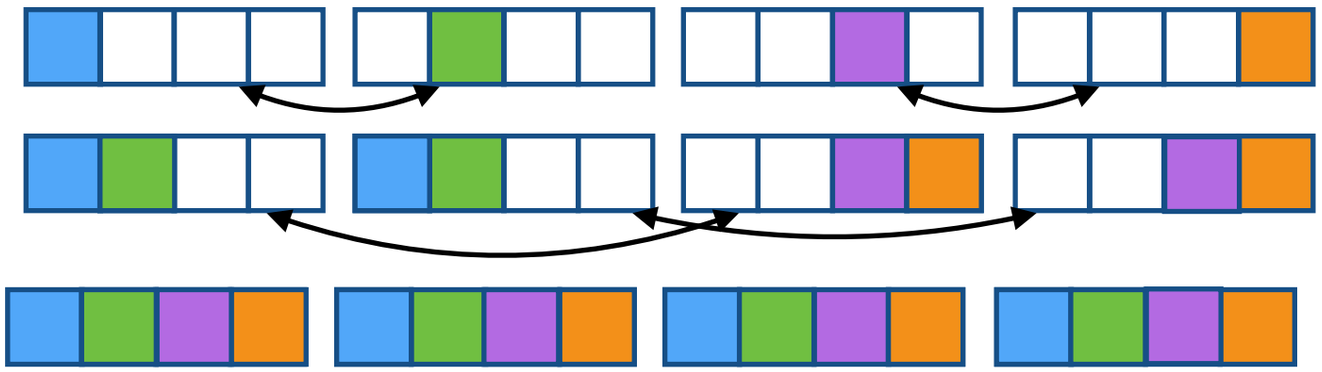



```
MPI_Alltoall(sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, comm);
```

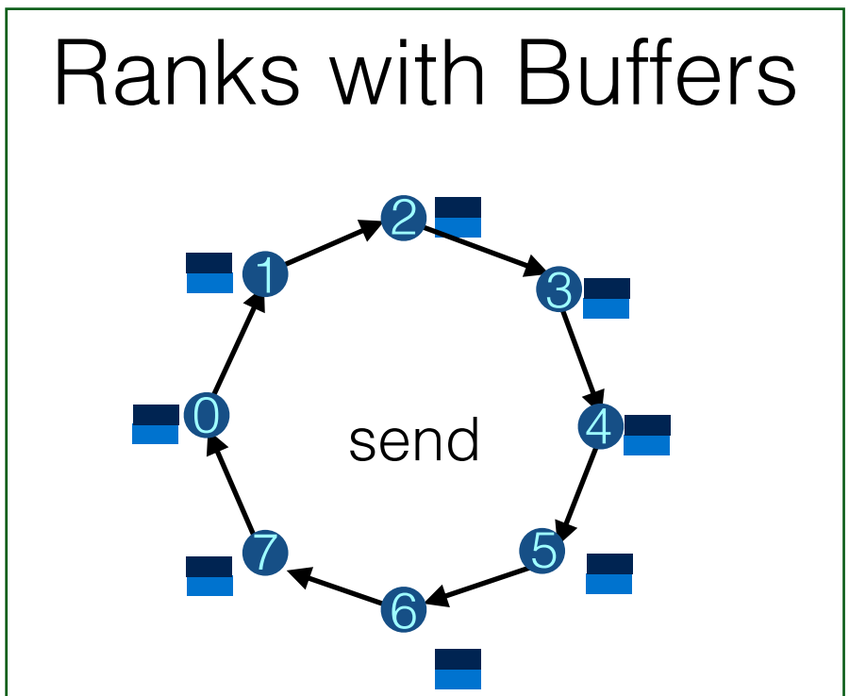
Similar to:

```
for (i = 0, i < n; i++)
        MPI_Send(sendbuf + i*sendcount*extent(sendtype), sendcount, sendtype, i, ...);
for (i = 0, i < n; i++)
        MPI_Recv(recvbuf + i*recvcount*extent(recvtype), recvcount, recvtype, i, ...,);
```

MPI_Alltoall



Recursive doubling

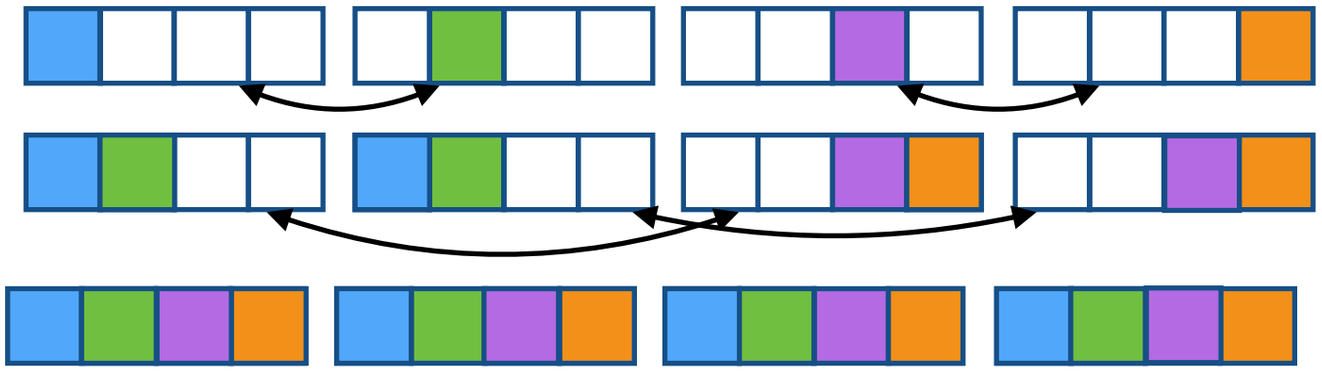


MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm);

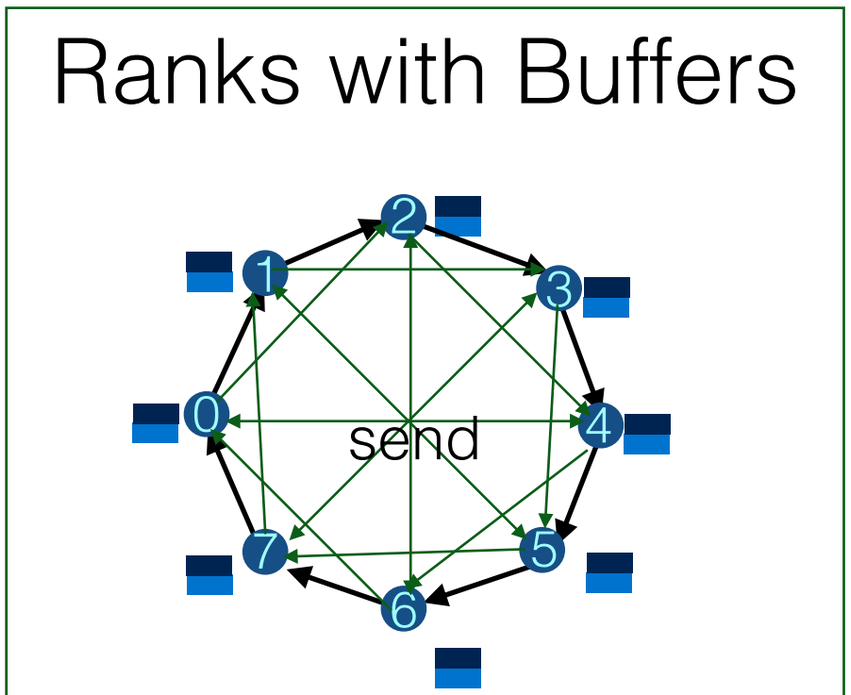
Similar to:

```
for (i = 0, i < n; i++)
        MPI_Send(sendbuf + i*sendcount*extent(sendtype), sendcount, sendtype, i, ...);
for (i = 0, i < n; i++)
        MPI_Recv(recvbuf + i*recvcount*extent(recvtype), recvcount, recvtype, i, ...,);
```

MPI_Alltoall



Recursive doubling

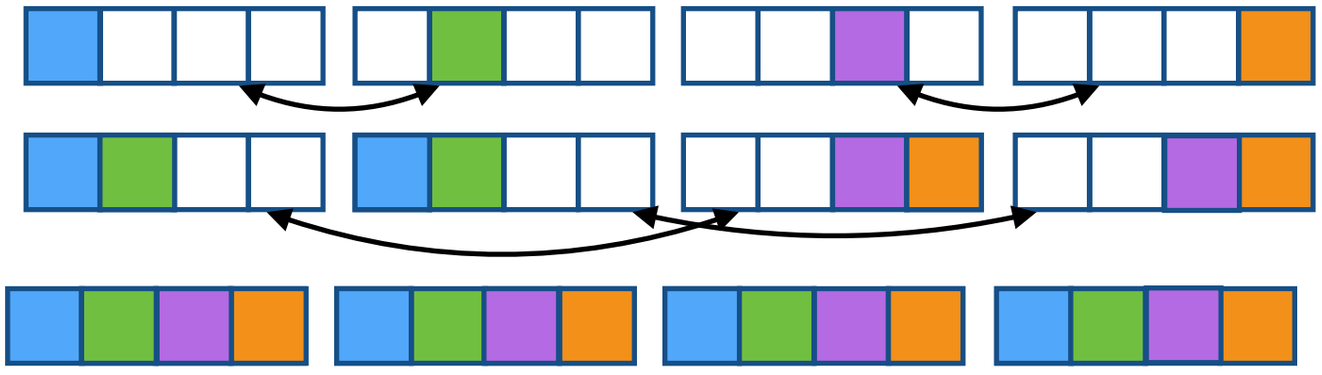



```
MPI_Alltoall(sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, comm);
```

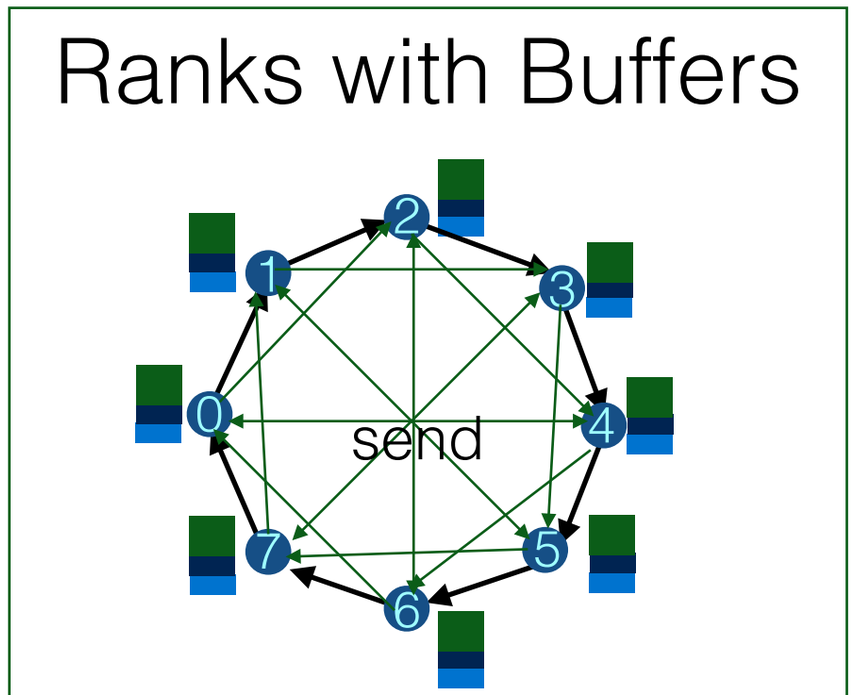
Similar to:

```
for (i = 0, i < n; i++)
        MPI_Send(sendbuf + i*sendcount*extent(sendtype), sendcount, sendtype, i, ...);
for (i = 0, i < n; i++)
        MPI_Recv(recvbuf + i*recvcount*extent(recvtype), recvcount, recvtype, i, ...,);
```

MPI_Alltoall



Recursive doubling




```
MPI_Alltoall(sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, comm);
```

Similar to:

```
for (i = 0, i < n; i++)
        MPI_Send(sendbuf + i*sendcount*extent(sendtype), sendcount, sendtype, i, ...);
for (i = 0, i < n; i++)
        MPI_Recv(recvbuf + i*recvcount*extent(recvtype), recvcount, recvtype, i, ...,);
```

Gather Example

```
int gsize, myrank, sendarray[100];
int root, *recvbuf;
MPI_Datatype rtype;
  ...
MPI_Comm_rank( comm, &myrank);
MPI_Comm_size( comm, &gsize);
MPI_Type_contiguous( 100, MPI_INT, &rtype );
MPI_Type_commit( &rtype );
if(myrank == root)
   recvbuf = (int *) malloc(gsize * 100*sizeof(int));
MPI_Gather(sendarray, 100, MPI_INT,
                recvbuf, 1, rtype, root, MPI_COMM_WORLD);
```

Gather Example

```
int gsize, myrank, sendarray[100];
int root, *recvbuf;
MPI_Datatype rtype;
  ...
MPI_Comm_rank( comm, &myrank);
MPI_Comm_size( comm, &gsize);
MPI_Type_contiguous( 100, MPI_INT, &rtype );
MPI_Type_commit( &rtype );
if(myrank == root)
   recvbuf = (int *) malloc(gsize * 100*sizeof(int));
MPI_Gather(sendarray, 100, MPI_INT,
                recvbuf, 1, rtype, root, MPI_COMM_WORLD);
```

Recv 1 rtype from each member
(Recv args ignored at non-root)

Gather Example

```
int gsize, myrank, sendarray[100];
int root, *recvbuf;
MPI_Datatype rtype;
  ...
MPI_Comm_rank( comm, &myrank);
MPI_Comm_size( comm, &gsize);
MPI_Type_contiguous( 100, MPI_INT, &rtype );
MPI_Type_commit( &rtype );
if(myrank == root)
   recvbuf = (int *) malloc(gsize * 100*sizeof(int));
MPI_Gather(sendarray, 100, MPI_INT,
           recvbuf, 1, rtype, root, MPI_COMM_WORLD);
```

Use MPI_IN_PLACE on root to disable local copy to recvbuf

Recv 1 rtype from each member (Recv args ignored at non-root)

MPI_Reduce()

```
MPI_Reduce(dataArray, resultArray, count,
                type, MPI_SUM, root, com);
```

dataArray	data sent from each processor
Result	stores result of combining operation
count	number of items in each of dataArray, result
MPI_SUM	combining operation, one of a predefined set
root	rank of processor receiving data

- Multiple elements can be reduced in one shot
- Illegal to alias input and output arrays

MPI_Reduce variants

- MPI_Reduce: result is sent out to the root
  - the operation is applied element-wise for each element of the input arrays on each processor
- MPI_Allreduce: result is sent out to everyone
- MPI_Reduce_scatter: functionality equivalent to a reduce followed by a scatter
- User defined operations

User-defined reduce operations

```
void rfunction(void *invec,
               void *inoutvec, int *len,
               MPI_Datatype *datatype);

MPI_Op op;
MPI_Op_create(rfunction, commute, &op);
```

…

Later:

```
MPI_op_free(&op);
```

Prefix Scan

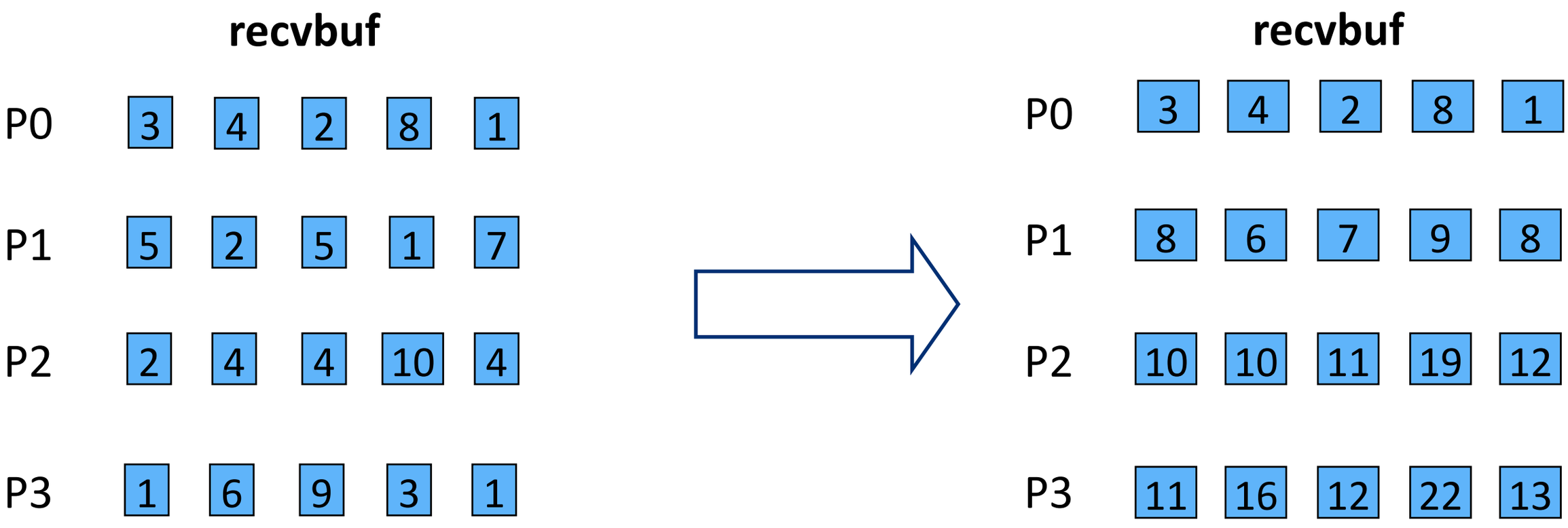
```
MPI_Scan(sendbuf, recvbuf, count,
  datatype, op, comm);
```

- Performs a prefix reduction on data in sendbuf
  - Multiple prefix ops in one shot
- Returns in the receive buffer of the process i:
  - reduction of the values in the send buffers of processes 0,...,i (inclusive)
- All must agree on op, datatype, count
- There is also exclusive scan:

```
MPI_Escan(sendbuf, recvbuf, count,
datatype, op, comm);
```

In-place MPI_Scan

process i receives data reduced on process 0 to i




```
MPI_Scan(MPI_IN_PLACE, recvbuf, 5, MPI_INT, MPI_SUM,
         MPI_COMM_WORLD)
```

Non-blocking Collectives

- MPI_Ireduce(sendbuf, recvbuf, count, type, op,
  root, comm, &request);

- MPI_Igather(sendbuf, sendcount, sendtype,
  recvbuf, recvcount, recvtype, root, comm, &request);

- MPI_Ialltoall(sendbuf, sendcount, sendtype,
  recvbuf, recvcount, recvtype, comm, &request);

Process Start

`MPI_Comm_Spawn(command, argv, maxprocs, info, root, comm, intercomm, array_of_errcodes)`

- The children have their own MPI_COMM_WORLD
- May not return until MPI_INIT has been called in the children
- More efficient to start all processes at once

Process Start

```
MPI_Comm_Spawn(command, argv, maxprocs,
  info, root, comm, intercomm,
  array_of_errcodes)
```

- The children have their own MPI_COMM_WORLD
- May not return until MPI_INIT has been called in the children
- More efficient to start all processes at once

Also see:

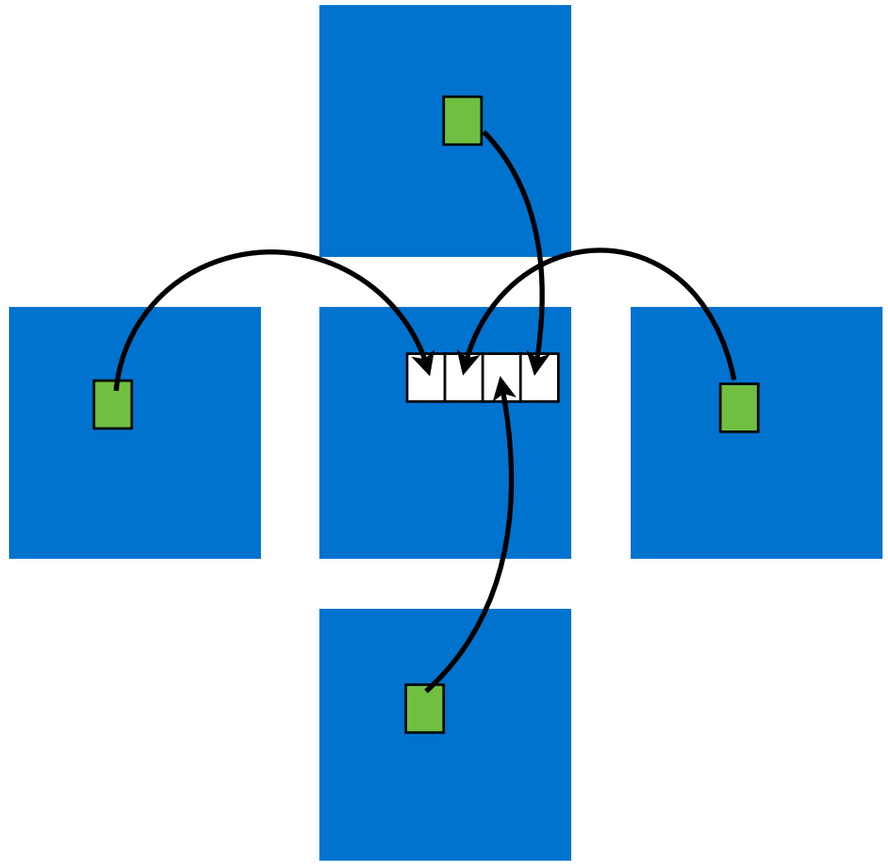
```
MPI_Comm_group(comm,&grp);
MPI_Group_incl(grp, newgrpcount,
            array_of_ranks_to_include, &newgrp);
MPI_Comm_create(comm, newgrp, &newcomm);
```

Process Topology

- Define neighbor
- Send to/Receive from neighbor

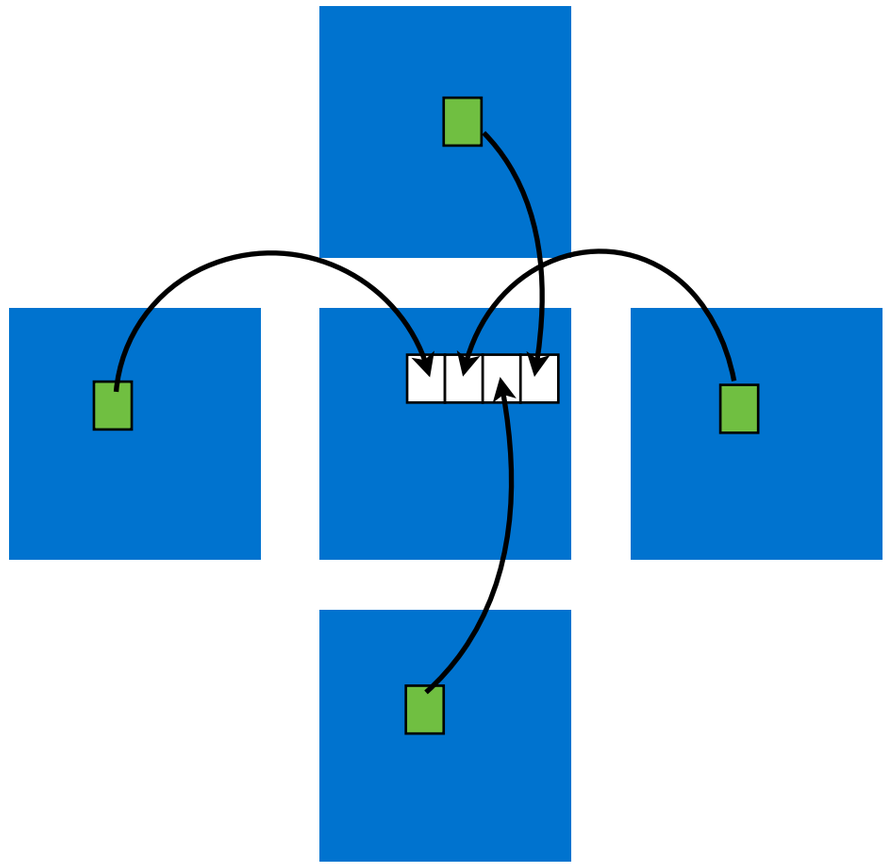
Process Topology

- Define neighbor
- Send to/Receive from neighbor



Process Topology

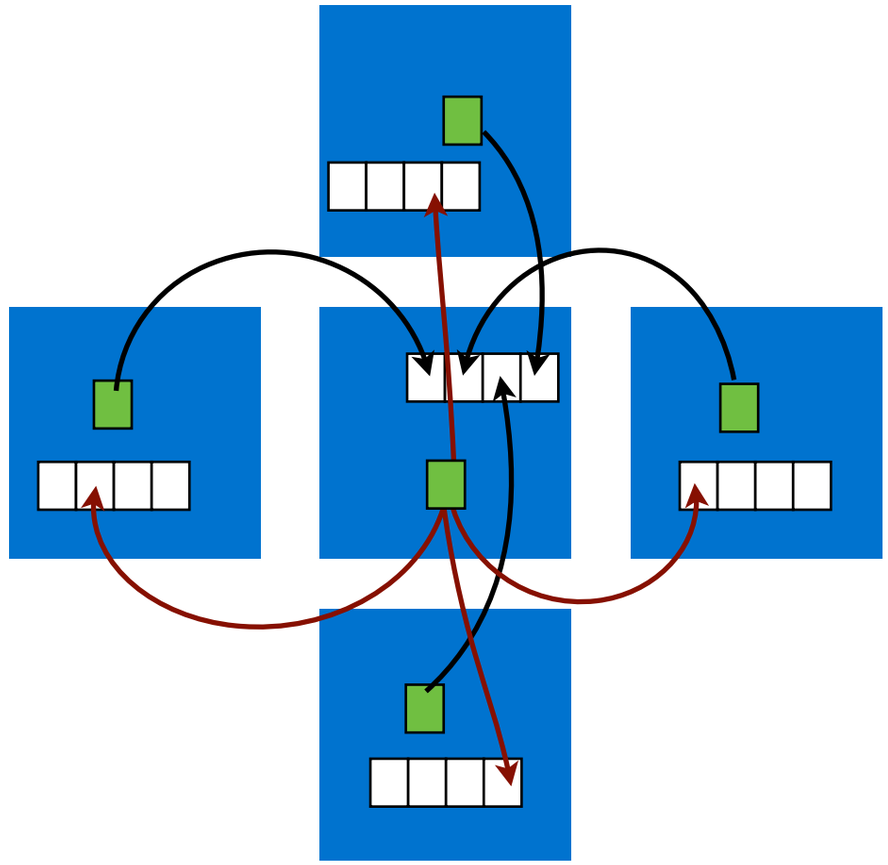
- Define neighbor
- Send to/Receive from neighbor



```
int ID0;
MPI_Comm_rank(MPI_COMM_WORLD, &ID0);
MPI_Communicator gridcomm;
int mxm[] = {3, 3}, periodic = {1, 1}, rerank = 1;

MPI_Cart_create(MPI_COMM_WORLD, 2, mxm, periodic, rerank, &gridcomm);
                        // initial comm, dimension, Wrap-around?, rank rename?, new comm

int recv[4];
MPI_Neighbor_allgather(&ID0, 1, MPI_INT, recv, 4, MPI_INT, gridcomm);
```

Process Topology

- Define neighbor
- Send to/Receive from neighbor



```
int ID0;
MPI_Comm_rank(MPI_COMM_WORLD, &ID0);
MPI_Communicator gridcomm;
int mxm[] = {3, 3}, periodic = {1, 1}, rerank = 1;

MPI_Cart_create(MPI_COMM_WORLD, 2, mxm, periodic, rerank, &gridcomm);
                        // initial comm, dimension, Wrap-around?, rank rename?, new comm

int recv[4];
MPI_Neighbor_allgather(&ID0, 1, MPI_INT, recv, 4, MPI_INT, gridcomm);
```

Remote Memory

```
MPI_Win win;
MPI_Info info;
MPI_Win_create(basemem, size, disp_unit,
info, MPI_COMM_WORLD, &win);
…
MPI_Win_free(&win);
```

- Weak synchronization
- Collective call
- Info specifies system-specific information
  - For example, memory locking
  - Designed for optimized performance
- Using **`MPI_Alloc_mem`** for basemem could be faster

MPI_Put, MPI_Get

```
MPI_Put(src_addr, src_count,
src_datatype, dest_rank, dest_disp,
dest_count, dest_datatype, win);
```

- Written in the dest window-buffer at address
  - window_base + disp×disp_unit
- Must fit in the target buffer
- dest_datatype defined on the src
  - should match definition on dest
- **`MPI_Get`** reverses
  - For the same call: data will be retrieved from ‘dest’ to ‘src’
- **`MPI_Accumulate`** performs an “op” instead of replacing at dest

Remote Memory Synchronization

- MPI_Win_fence
- MPI_Win_lock
- MPI_Win_unlock
- MPI_Win_start
- MPI_Win_complete
- MPI_Win_post
- MPI_Win_Wait
- MPI_Win_Test

Remote Memory Synchronization

- MPI_Win_fence
- MPI_Win_lock
- MPI_Win_unlock
- MPI_Win_start
- MPI_Win_complete
- MPI_Win_post
- MPI_Win_Wait
- MPI_Win_Test

Look it up

Derived Datatypes

- MPI sends typed data
  - but it does not understand your data structures
    - Too system architecture dependent
    - Need to tell MPI in an abstract, platform independent fashion

- Typemap:
  - $(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})$
  - i^{th} entry is of $type_i$ and starts at byte base + $disp_i$

e.g.,

```
MPI_Datatype newtype;
MPI_Type_contiguous(count, aknowntype, &newtype)
```

Blocks

```
MPI_Type_vector(blockcount, blocklength,
  blockstride, knowntype, &newtype);
```

- Equally-spaced blocks of the known datatype
- Stride between blocks specified in units of *knowntype*
- All blocks are of the same length
- Contiguous copies

```
MPI_Type_create_hvector(blockcount, blk_len,
  bytestride, knowntype, &newtype);
```

- Gap between blocks is in bytes

Blocks

```
MPI_Type_vector(blockcount, blocklength,
  blockstride, knowntype, &newtype);
```

- Equally-spaced blocks of the known datatype
- Stride between blocks specified in units of *knowntype*
- All blocks are of the same length
- Contiguous copies

```
MPI_Type_create_hvector(blockcount, blk_len,
  bytestride, knowntype, &newtype);
```

- Gap between blocks is in bytes

Generalized Blocks

```
MPI_Type_indexed(count, //3
    array_of_blocklengths, // 3, 2, 1
    array_of_displacements, // 0, 5, 10
    knowntype, &newtype);
```

- Replication into a sequence of blocks
- Blocks can contain different number of copies
- And may have different strides
- But the same data type

Generalized Blocks

```
MPI_Type_indexed(count, //3
    array_of_blocklengths, // 3, 2, 1
    array_of_displacements, // 0, 5, 10
    knowntype, &newtype);
```

- Replication into a sequence of blocks
- Blocks can contain different number of copies
- And may have different strides
- But the same data type

Struct

```
MPI_Type_create_struct(count,
   array_of_blocklengths,
   array_of_bytedisplacements,
   array_of_knowntypes, &newtype)
```

- e.g., If
  - Type0 = {(double, 0), (char, 8)};
  - int B[] = {2, 1, 3}, D[] = {0, 16, 26};
  - MPI_Datatype T[] = {MPI_FLOAT, Type0, MPI_CHAR};
- **MPI_Type_create_struct(3, B, D, T, &newtype)** returns
  - (float, 0), (float, 4), (double, 16), (char, 24), (char, 26), (char, 27), (char, 28)
- Other functions for structs distributed across processors

f4 f4 G8 d8 c1 c1 c1 c1

Struct

```
MPI_Type_create_struct(count,
    array_of_blocklengths,
    array_of_bytedisplacements,
    array_of_knowntypes, &newtype)
```

- e.g., If
  - Type0 = {(double, 0), (char, 8)};
  - int B[] = {2, 1, 3}, D[] = {0, 16, 26};
  - MPI_Datatype T[] = {MPI_FLOAT, Type0, MPI_CHAR};
- **MPI_Type_create_struct(3, B, D, T, &newtype)** returns
  - (float, 0), (float, 4), (double, 16), (char, 24), (char, 26), (char, 27), (char, 28)
- Other functions for structs distributed across processors

f4 f4 G8 d8 c1 c1 c1 c1

Data Type Functions

`MPI_Type_size(datatype, &size)`

- Actual data size in bytes

`MPI_Type_commit(&datatype)`

- A datatype object must be committed before communication

`MPI_Type_get_extent(datatype, &lb, &ext)`

- Lower bound and extent

`MPI_Type_create_resized(oldtype, lb, extent, &newtype)`

- Create Gap

```
MPI_Datatype type0;
MPI_Type_contiguous(1, MPI_CHAR, &type0);
int size;
MPI_Type_size(type0, &size);
MPI_Type_commit(&type0);
MPI_Send(buf, nItems, type0, dest, tag, MPI_COMM_WORLD);
```

Data Type Functions

`MPI_Type_size(datatype, &size)`

- Actual data size in bytes

`MPI_Type_commit`

- A datatype object

`MPI_Type_get_ex`

- Lower bound and

`MPI_Type_create`

- Create Gap

```
MPI_Datatype type0;
MPI_Type_contiguous
int size;
MPI_Type_size(type0,
MPI_Type_commit(&t
MPI_Send(buf, nItems
```

```
struct {
        float value;
        char flag;
} data[10];
int blocklengths[] = {1, 1};
MPI_Datatype types[] = {MPI_FLOAT, MPI_CHAR};
MPI_Datatype newtype, tmptype;
MPI_Aint extent, base, displacements[2];

MPI_Get_address(data, &base);
MPI_Get_address(&data[0].value, &displacements[0]);
MPI_Get_address(&data[0].flag, &displacements[1]);
MPI_Get_address(&data[1].value, &extent);
displacements[0] = MPI_Aint_diff(displacements[0], base);
displacements[1] = MPI_Aint_diff(displacements[1], base);
extent = MPI_Aint_diff(extent, base);

MPI_Type_create_struct(2, blocklengths, displacements, types, &tmptype );
MPI_Type_create_resized( tmptype, lb, extent, &newtype );
MPI_Type_commit( &newtype );
```

Data Type Functions

`MPI_Type_size(datatype, &size)`

- Actual data size in bytes

`MPI_Type_commit`

- A datatype object

`MPI_Type_get_ex`

- Lower bound and

`MPI_Type_create`

- Create Gap

```
MPI_Datatype type0;
MPI_Type_contiguous
int size;
MPI_Type_size(type0,
MPI_Type_commit(&t
MPI_Send(buf, nItems
```

```
struct {
      float value;
      char flag;
} data[10];
int blocklengths[] = {1, 1};
MPI_Datatype types[] = {MPI_FLOAT, MPI_CHAR};
MPI_Datatype newtype, tmptype;
MPI_Aint extent, base, displacements[2];

MPI_Get_address(data, &base);
MPI_Get_address(&data[0].value, &displacements[0]);
MPI_Get_address(&data[0].flag, &displacements[1]);
MPI_Get_address(&data[1].value, &extent);
displacements[0] = MPI_Aint_diff(displacements[0], base);
displacements[1] = MPI_Aint_diff(displacements[1], base);
extent = MPI_Aint_diff(extent, base);

MPI_Type_create_struct(2, blocklengths, displacements, types, &tmptype );
MPI_Type_create_resized( tmptype, lb, extent, &newtype );
MPI_Type_commit( &newtype );
```

Data Type Functions

`MPI_Type_size(datatype, &size)`

- Actual data size in bytes

`MPI_Type_commit`

- A datatype object

`MPI_Type_get_ex`

- Lower bound and

`MPI_Type_create`

- Create Gap

```
MPI_Datatype type0;
MPI_Type_contiguous
int size;
MPI_Type_size(type0,
MPI_Type_commit(&t
MPI_Send(buf, nItems
```

```
struct {
        float value;
        char flag;
} data[10];
int blocklengths[] = {1, 1};
MPI_Datatype types[] = {MPI_FLOAT, MPI_CHAR};
MPI_Datatype newtype, tmptype;
MPI_Aint extent, base, displacements[2];

MPI_Get_address(data, &base);
MPI_Get_address(&data[0].value, &displacements[0]);
MPI_Get_address(&data[0].flag, &displacements[1]);
MPI_Get_address(&data[1].value, &extent);
displacements[0] = MPI_Aint_diff(displacements[0], base);
displacements[1] = MPI_Aint_diff(displacements[1], base);
extent = MPI_Aint_diff(extent, base);

MPI_Type_create_struct(2, blocklengths, displacements, types, &tmptype );
MPI_Type_create_resized( tmptype, lb, extent, &newtype );
MPI_Type_commit( &newtype );
```

Data Type Functions

`MPI_Type_size(datatype, &size)`

- Actual data size in bytes

`MPI_Type_commit`

- A datatype object

`MPI_Type_get_ex`

- Lower bound and

`MPI_Type_create`

- Create Gap

```
MPI_Datatype type0;
MPI_Type_contiguous
int size;
MPI_Type_size(type0,
MPI_Type_commit(&t
MPI_Send(buf, nItems
```

```
struct {
        float value;
        char flag;
} data[10];
int blocklengths[] = {1, 1};
MPI_Datatype types[] = {MPI_FLOAT, MPI_CHAR};
MPI_Datatype newtype, tmptype;
MPI_Aint extent, base, displacements[2];

MPI_Get_address(data, &base);
MPI_Get_address(&data[0].value, &displacements[0]);
MPI_Get_address(&data[0].flag, &displacements[1]);
MPI_Get_address(&data[1].value, &extent);
displacements[0] = MPI_Aint_diff(displacements[0], base);
displacements[1] = MPI_Aint_diff(displacements[1], base);
extent = MPI_Aint_diff(extent, base);

MPI_Type_create_struct(2, blocklengths, displacements, types, &tmptype );
MPI_Type_create_resized( tmptype, lb, extent, &newtype );
MPI_Type_commit( &newtype );
```

Data Type Functions

`MPI_Type_size(datatype, &size)`

- Actual data size in bytes

`MPI_Type_commit`

- A datatype object

`MPI_Type_get_ex`

- Lower bound and

`MPI_Type_create`

- Create Gap

```
MPI_Datatype type0;
MPI_Type_contiguous
int size;
MPI_Type_size(type0,
MPI_Type_commit(&t
MPI_Send(buf, nItems
```

```
struct {
        float value;
        char flag;
} data[10];
int blocklengths[] = {1, 1};
MPI_Datatype types[] = {MPI_FLOAT, MPI_CHAR};
MPI_Datatype newtype, tmptype;
MPI_Aint extent, base, displacements[2];

MPI_Get_address(data, &base);
MPI_Get_address(&data[0].value, &displacements[0]);
MPI_Get_address(&data[0].flag, &displacements[1]);
MPI_Get_address(&data[1].value, &extent);
displacements[0] = MPI_Aint_diff(displacements[0], base);
displacements[1] = MPI_Aint_diff(displacements[1], base);
extent = MPI_Aint_diff(extent, base);

MPI_Type_create_struct(2, blocklengths, displacements, types, &tmptype );
MPI_Type_create_resized( tmptype, lb, extent, &newtype );
MPI_Type_commit( &newtype );
```

Data Type Functions

`MPI_Type_size(datatype, &size)`

- Actual data size in bytes

`MPI_Type_commit`

- A datatype object

`MPI_Type_get_ex`

- Lower bound and

`MPI_Type_create`

- Create Gap

```
MPI_Datatype type0;
MPI_Type_contiguous
int size;
MPI_Type_size(type0,
MPI_Type_commit(&t
MPI_Send(buf, nItems
```

```
struct {
        float value;
        char flag;
} data[10];
int blocklengths[] = {1, 1};
MPI_Datatype types[] = {MPI_FLOAT, MPI_CHAR};
MPI_Datatype newtype, tmptype;
MPI_Aint extent, base, displacements[2];

MPI_Get_address(data, &base);
MPI_Get_address(&data[0].value, &displacements[0]);
MPI_Get_address(&data[0].flag, &displacements[1]);
MPI_Get_address(&data[1].value, &extent);
displacements[0] = MPI_Aint_diff(displacements[0], base);
displacements[1] = MPI_Aint_diff(displacements[1], base);
extent = MPI_Aint_diff(extent, base);

MPI_Type_create_struct(2, blocklengths, displacements, types, &tmptype );
MPI_Type_create_resized( tmptype, lb, extent, &newtype );
MPI_Type_commit( &newtype );
```

Data Type Functions

`MPI_Type_size(datatype, &size)`

- Actual data size in bytes

`MPI_Type_commit(&datatype)`

- A datatype object must be committed before communication

`MPI_Type_get_extent(datatype, &lb, &ext)`

- Lower bound and extent

`MPI_Type_create_resized(oldtype, lb, extent, &newtype)`

- Create Gap

```
MPI_Datatype type0;
MPI_Type_contiguous(1, MPI_CHAR, &type0);
int size;
MPI_Type_size(type0, &size);
MPI_Type_commit(&type0);
MPI_Send(buf, nItems, type0, dest, tag, MPI_COMM_WORLD);
```

Derived Datatype Example Usage

```
struct Partstruct
{
  int class; /* particle class */
  double d[6]; /* particle coordinates */
  char b[7]; /* some additional information */
};

void useStruct() {

  struct Partstruct particle[1000];
  int i, dest, rank;
  MPI_Comm comm;
```

```
/* build datatype describing structure */
  MPI_Datatype Particletype;
  MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
  int blocklen[3] = {1, 6, 7};
  MPI_Aint disp[3];
  MPI_Aint base;

  /* compute displacements of structure components */
  MPI_Get_address( particle, disp);
  MPI_Get_address( particle[0].d, disp+1);
  MPI_Get_address( particle[0].b, disp+2);
  base = disp[0];
  for (i=0; i <3; i++) disp[i] -= base;

  MPI_Type_struct( 3, blocklen, disp, type, &Particletype);
  MPI_Type_commit( &Particletype);
  MPI_Send( particle, 1000, Particletype, dest, tag, comm);
}
```

MPI_Pack()

```
MPI_Pack(dataPtr, count, type,
         packedbuffer, buffersize,
         &bufferposition, communicator);
```

dataPtr	pointer to data that needs to be packed
count	number of items to pack
type	type of items to pack
buffer	buffer to pack into
size	size of buffer (in bytes) – must be large enough
position	starting position in buffer (in bytes), updated to the end of the packed area

MPI_Pack()

```
MPI_Pack(dataPtr, count, type,
         packedbuffer, buffersize,
         &bufferposition, communicator);
```

dataPtr	pointer to data that needs to be packed
count	number of items to pack
type	type of items to pack
buffer	buffer to pack into
size	size of buffer (in bytes) – must be large enough
position	starting position in buffer (in bytes), updated to the end of the packed area

Also see:
MPI_Pack_external

MPI_Unpack()

- Can first check packed output size:

```
MPI_Pack_size(incount, datatype,
comm, &size)
```

```
MPI_Unpack(packedbuffer, size, &position,
  dataPtr, count, type, communicator);
```

count actual number of items to unpack
must have enough space

More Examples

Example: PI Computation

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
   int done = 0, n, myid, numprocs, i, rc;
   double PI25DT = 3.141592653589793238462643;
   double mypi, pi, h, sum, x, a;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
while (!done)  { // Repeatedly compute PI
  if (myid == 0) {
    printf("Enter the number of intervals: (0 quits) ");
    scanf("%d",&n);
  }
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
  if (n == 0) break;
```

Example: PI Computation page-2

```
 h   = 1.0 / (double) n;
sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs) { // some terms
  x = h * ((double)i - 0.5);
  sum += 4.0 / (1.0 + x*x);
}
mypi = h * sum;
   MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
              MPI_COMM_WORLD);
   if (myid == 0)
     printf("pi is approximately %.16f, Error is %.16f\n",
            pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();
 return 0;
}
```

Data types

```
struct {
   int a;
  double b
} value;
MPI_Datatype  mystruct;
int                rank, blocklens[2];
MPI_Aint          indices[2];
MPI_Datatype old_types[2];

MPI_Comm_rank( MPI_COMM_WORLD, &rank );

/* One value of each type */
blocklens[0] = blocklens[1] = 1;

/* Base types */
old_types[0] = MPI_INT;
old_types[1] = MPI_DOUBLE;

/* The locations of each element */
MPI_Address( &value.a, &indices[0] );
MPI_Address( &value.b, &indices[1] );

/* Make relative */
indices[1] = indices[1] - indices[0];
indices[0] = 0;
MPI_Type_struct(2, blocklens, indices, old_types, &mystruct );
MPI_Type_commit( &mystruct );
```

Using Data type

```
do {
    if (rank == 0)
        scanf( "%d %lf", &value.a, &value.b );

    MPI_Bcast( &value, 1, mystruct, 0, MPI_COMM_WORLD );

    printf( "Process %d got %d and %lf\n", rank, value.a, value.b );
} while (value.a >= 0);

/* Clean up the type */
MPI_Type_free( &mystruct );
```

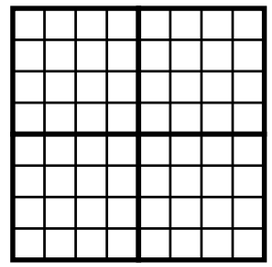
Jacobi

```
int rank, value, size, errcnt, toterr, i, j;
MPI_Status status;
double xlocal[(MAXN/MAXP)+2][MAXN];

MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
if (size != MAXP) MPI_Abort( MPI_COMM_WORLD, 1 );

/* Fill the data: xlocal[][0] is lower ghostpoints, xlocal[][MAXN+2] is upper */

for (i=1; i<=MAXN/size; i++) // Initialize data
    for (j=0; j<MAXN; j++)
        xlocal[i][j] = rank;

 if (rank < size - 1)  // Send up, unless I'm at the top (size-1)
    MPI_Send( xlocal[MAXN/size], MAXN, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD );
 if (rank > 0)  // Receive from below unless I am bottom (0)
    MPI_Recv(xlocal[0], MAXN, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD, &status );

if (rank > 0) // Send down
    MPI_Send(xlocal[1], MAXN, MPI_DOUBLE, rank - 1, 1, MPI_COMM_WORLD);
 if (rank < size - 1)  // Receive from above
    MPI_Recv(xlocal[MAXN/size+1], MAXN, MPI_DOUBLE, rank+1, 1,  MPI_COMM_WORLD,
                                                                 &status);
```

Scatter Matrix

```
double A[8][8], alocal[4][4];
int i, j, r, rank, size;
MPI_Datatype stype, vtype;
int   sendcount[4], sdispls[4];

MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
if (size != 4)  MPI_Abort( MPI_COMM_WORLD, 1 );
if (rank == 0) {

   // Initialize 8x8 matrix A here ..

   MPI_Type_vector(4, 4, 8, MPI_DOUBLE, &vtype );  // 4 sets of 4 doubles, each offset by 8
   MPI_Type_create_resized(vtype, 0, 4*sizeof(double), &stype); // Artificial type for scatter
   MPI_Type_commit( &stype );

   /* Setup the Scatter values for the send buffer */
   sendcount[0] =  sendcount[1] = sendcount[2] = sendcount[3] = 1; // Send one to each
   sdispls[0] = 0; // Starting locations in A of the four sub matrices in terms of stype extent
   sdispls[1] = 1;
   sdispls[2] = 8;
   sdispls[3] = 9;
   MPI_Scatterv(A, sendcount, sdispls, stype,
                     alocal, 4*4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
} else {
   MPI_Scatterv( NULL, NULL, NULL, MPI_DATATYPE_NULL,
                                    alocal, 4*4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

Scatter Matrix

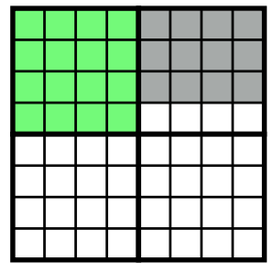


```
double A[8][8], alocal[4][4];
int i, j, r, rank, size;
MPI_Datatype stype, vtype;
int   sendcount[4], sdispls[4];

MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
if (size != 4)  MPI_Abort( MPI_COMM_WORLD, 1 );
if (rank == 0) {

   // Initialize 8x8 matrix A here ..

   MPI_Type_vector(4, 4, 8, MPI_DOUBLE, &vtype );  // 4 sets of 4 doubles, each offset by 8
   MPI_Type_create_resized(vtype, 0, 4*sizeof(double), &stype); // Artificial type for scatter
   MPI_Type_commit( &stype );

   /* Setup the Scatter values for the send buffer */
   sendcount[0] =  sendcount[1] = sendcount[2] = sendcount[3] = 1; // Send one to each
   sdispls[0] = 0; // Starting locations in A of the four sub matrices in terms of stype extent
   sdispls[1] = 1;
   sdispls[2] = 8;
   sdispls[3] = 9;
   MPI_Scatterv(A, sendcount, sdispls, stype,
                     alocal, 4*4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
} else {
   MPI_Scatterv( NULL, NULL, NULL, MPI_DATATYPE_NULL,
                                    alocal, 4*4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

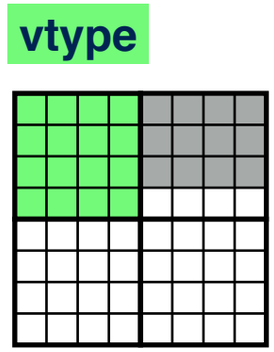
Scatter Matrix

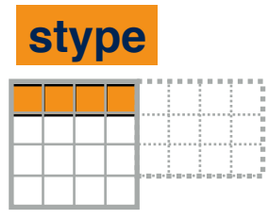
```
double A[8][8], alocal[4][4];
int i, j, r, rank, size;
MPI_Datatype stype, vtype;
int   sendcount[4], sdispls[4];

MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
if (size != 4)  MPI_Abort( MPI_COMM_WORLD, 1 );
if (rank == 0) {

   // Initialize 8x8 matrix A here ..

   MPI_Type_vector(4, 4, 8, MPI_DOUBLE, &vtype );  // 4 sets of 4 doubles, each offset by 8
   MPI_Type_create_resized(vtype, 0, 4*sizeof(double), &stype); // Artificial type for scatter
   MPI_Type_commit( &stype );

   /* Setup the Scatter values for the send buffer */
   sendcount[0] =  sendcount[1] = sendcount[2] = sendcount[3] = 1; // Send one to each
   sdispls[0] = 0; // Starting locations in A of the four sub matrices in terms of stype extent
   sdispls[1] = 1;
   sdispls[2] = 8;
   sdispls[3] = 9;
   MPI_Scatterv(A, sendcount, sdispls, stype,
                     alocal, 4*4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
} else {
   MPI_Scatterv( NULL, NULL, NULL, MPI_DATATYPE_NULL,
                                    alocal, 4*4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

vtype



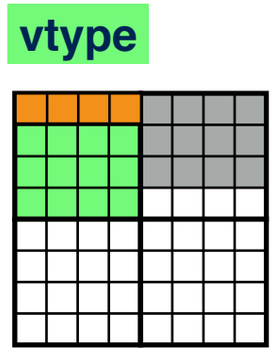
Scatter Matrix

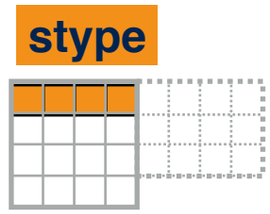





```
double A[8][8], alocal[4][4];
int i, j, r, rank, size;
MPI_Datatype stype, vtype;
int   sendcount[4], sdispls[4];

MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
if (size != 4)  MPI_Abort( MPI_COMM_WORLD, 1 );
if (rank == 0) {

   // Initialize 8x8 matrix A here ..

   MPI_Type_vector(4, 4, 8, MPI_DOUBLE, &vtype );  // 4 sets of 4 doubles, each offset by 8
   MPI_Type_create_resized(vtype, 0, 4*sizeof(double), &stype); // Artificial type for scatter
   MPI_Type_commit( &stype );

   /* Setup the Scatter values for the send buffer */
   sendcount[0] =  sendcount[1] = sendcount[2] = sendcount[3] = 1; // Send one to each
   sdispls[0] = 0; // Starting locations in A of the four sub matrices in terms of stype extent
   sdispls[1] = 1;
   sdispls[2] = 8;
   sdispls[3] = 9;
   MPI_Scatterv(A, sendcount, sdispls, stype,
                     alocal, 4*4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
} else {
   MPI_Scatterv( NULL, NULL, NULL, MPI_DATATYPE_NULL,
                                    alocal, 4*4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

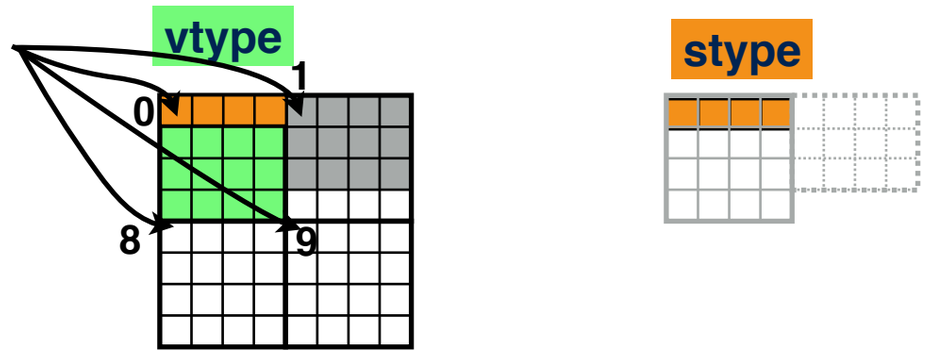
Scatter Matrix






```
double A[8][8], alocal[4][4];
int i, j, r, rank, size;
MPI_Datatype stype, vtype;
int   sendcount[4], sdispls[4];

MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
if (size != 4)  MPI_Abort( MPI_COMM_WORLD, 1 );
if (rank == 0) {

   // Initialize 8x8 matrix A here ..

   MPI_Type_vector(4, 4, 8, MPI_DOUBLE, &vtype );  // 4 sets of 4 doubles, each offset by 8
   MPI_Type_create_resized(vtype, 0, 4*sizeof(double), &stype); // Artificial type for scatter
   MPI_Type_commit( &stype );

   /* Setup the Scatter values for the send buffer */
   sendcount[0] =  sendcount[1] = sendcount[2] = sendcount[3] = 1; // Send one to each
   sdispls[0] = 0; // Starting locations in A of the four sub matrices in terms of stype extent
   sdispls[1] = 1;
   sdispls[2] = 8;
   sdispls[3] = 9;
   MPI_Scatterv(A, sendcount, sdispls, stype,
                     alocal, 4*4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
} else {
   MPI_Scatterv( NULL, NULL, NULL, MPI_DATATYPE_NULL,
                                    alocal, 4*4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

Scatter Matrix

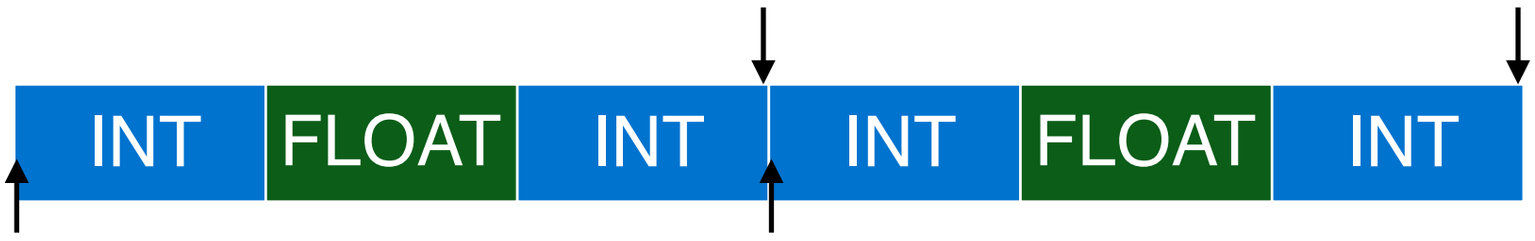



```
double A[8][8], alocal[4][4];
int i, j, r, rank, size;
MPI_Datatype stype, vtype;
int   sendcount[4], sdispls[4];

MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
if (size != 4)  MPI_Abort( MPI_COMM_WORLD, 1 );
if (rank == 0) {

   // Initialize 8x8 matrix A here ..

   MPI_Type_vector(4, 4, 8, MPI_DOUBLE, &vtype );  // 4 sets of 4 doubles, each offset by 8
   MPI_Type_create_resized(vtype, 0, 4*sizeof(double), &stype); // Artificial type for scatter
   MPI_Type_commit( &stype );

   /* Setup the Scatter values for the send buffer */
   sendcount[0] =  sendcount[1] = sendcount[2] = sendcount[3] = 1; // Send one to each
   sdispls[0] = 0; // Starting locations in A of the four sub matrices in terms of stype extent
   sdispls[1] = 1;
   sdispls[2] = 8;
   sdispls[3] = 9;
   MPI_Scatterv(A, sendcount, sdispls, stype,
                     alocal, 4*4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
} else {
   MPI_Scatterv( NULL, NULL, NULL, MPI_DATATYPE_NULL,
                                   alocal, 4*4, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

MPI Type Resizing

- Bounds control replication: where the next item begins
  - each item remains the same as before

MPI Type Resizing

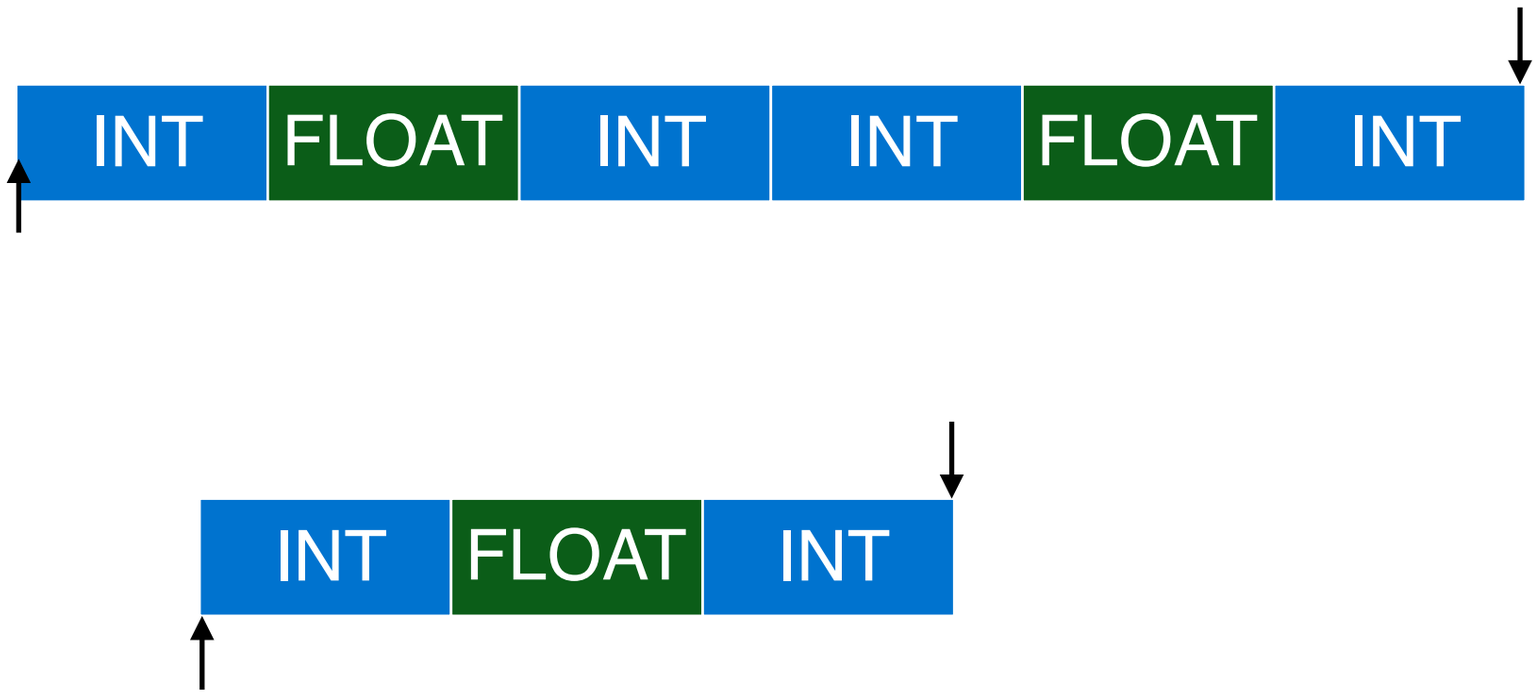


- Bounds control replication: where the next item begins
  - each item remains the same as before

MPI Type Resizing

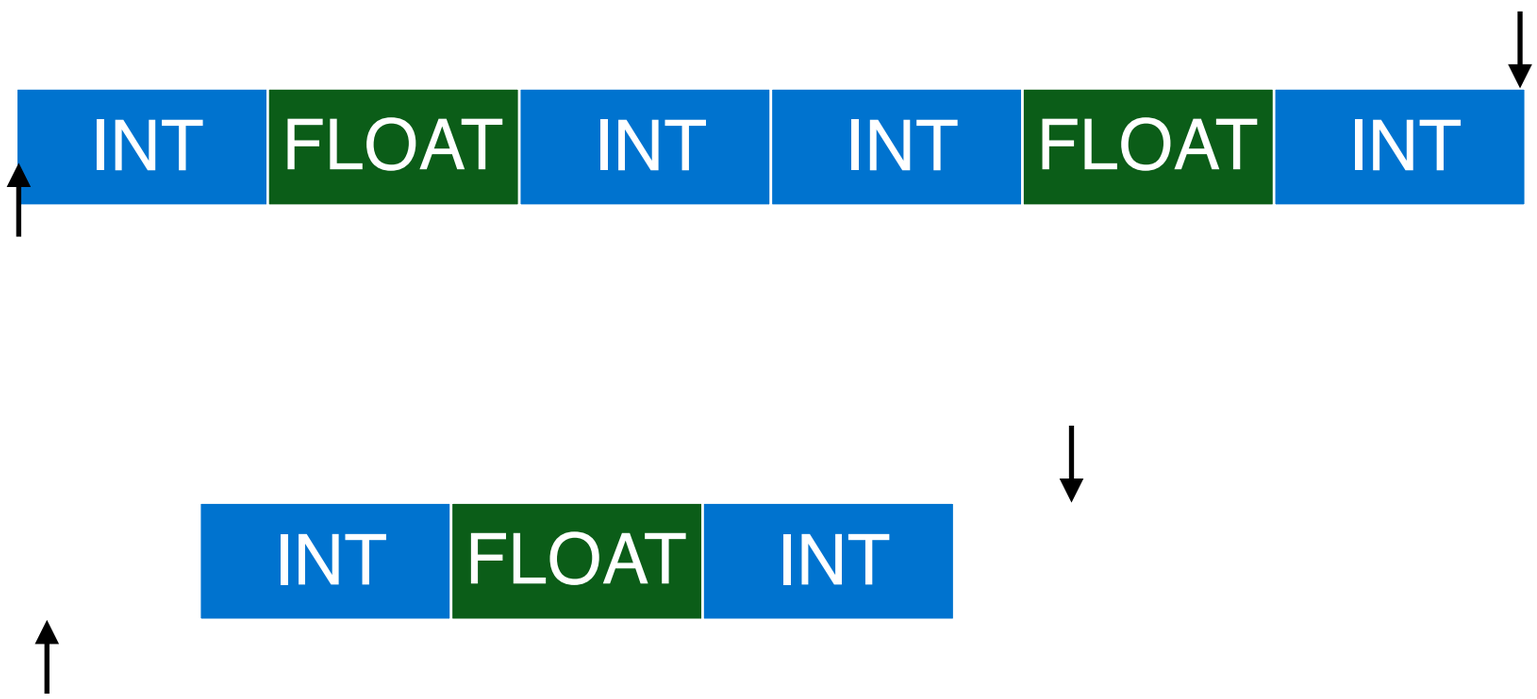
- Bounds control replication: where the next item begins
  - each item remains the same as before

MPI Type Resizing



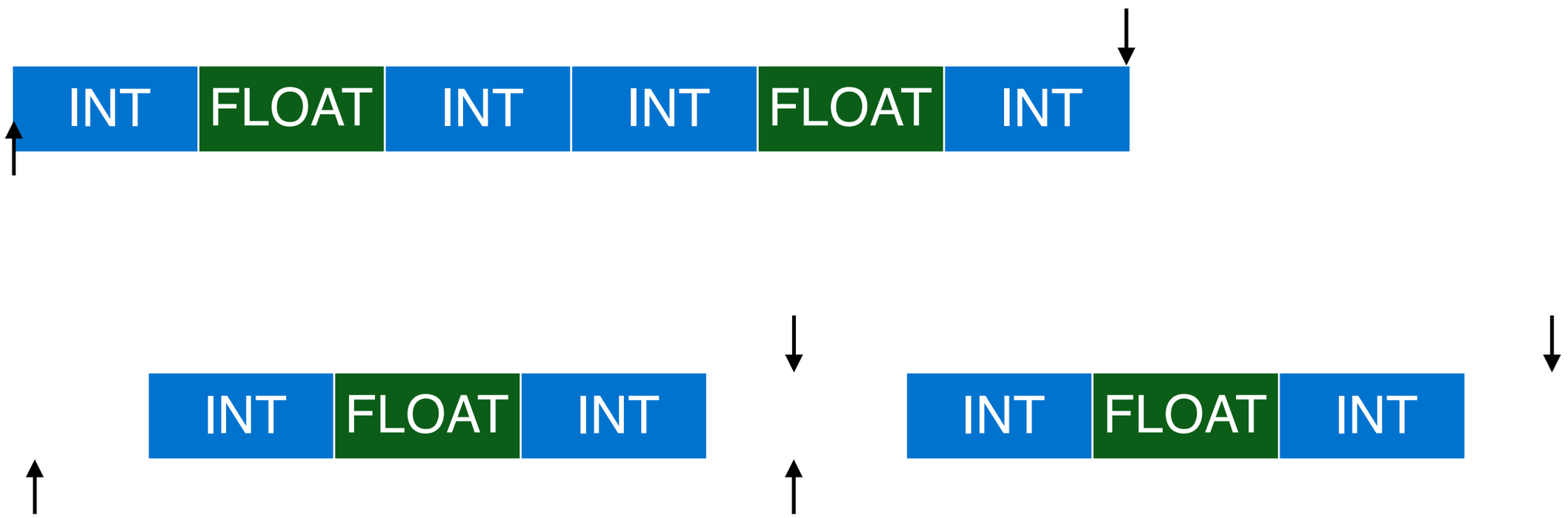
- Bounds control replication: where the next item begins
  - each item remains the same as before

MPI Type Resizing



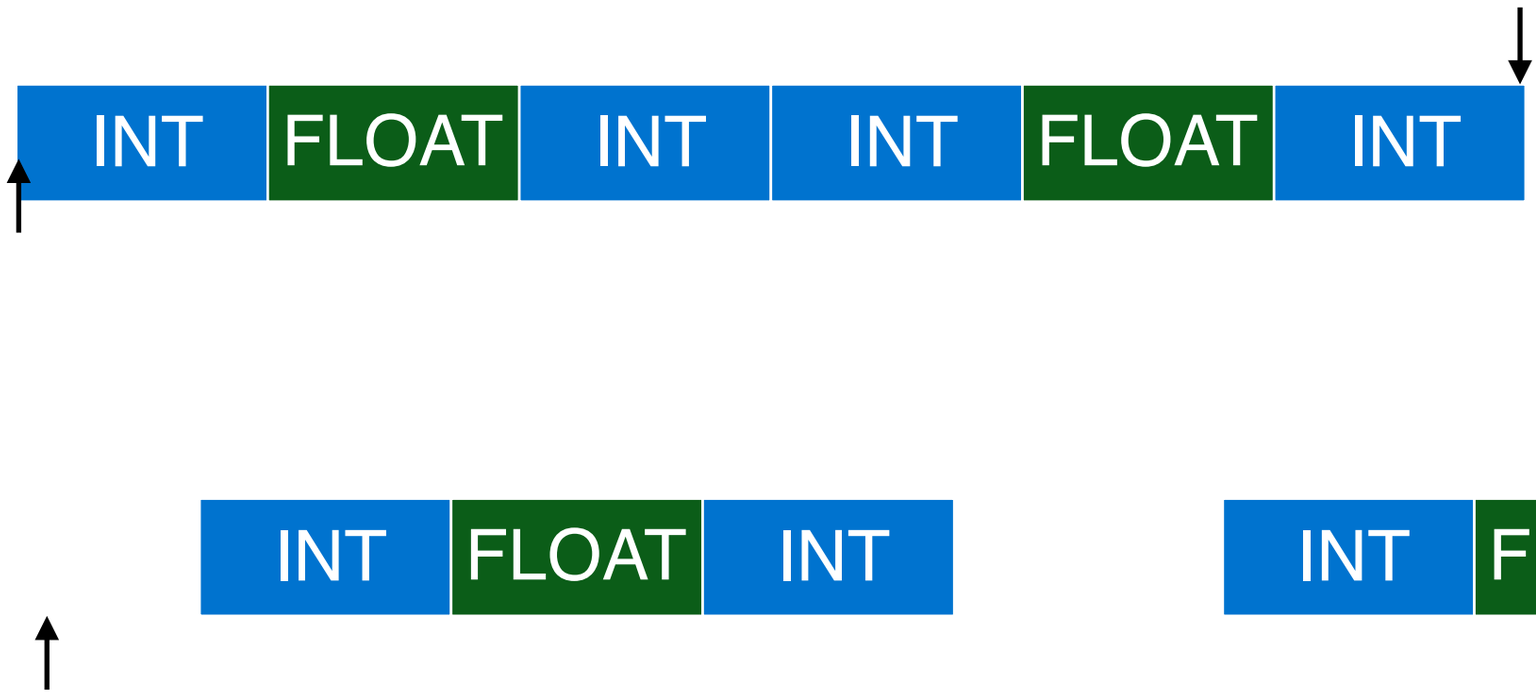
- Bounds control replication: where the next item begins
  - each item remains the same as before

MPI Type Resizing



- Bounds control replication: where the next item begins
  - each item remains the same as before

MPI Type Resizing



- Bounds control replication: where the next item begins
  - each item remains the same as before

MPI-I/O

```
MPI_File fh;
MPI_Status status;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
bufsize = FILESIZE/nprocs;
nints = bufsize/sizeof(int);
MPI_File_open(MPI_COMM_WORLD, “/file", MPI_MODE_RDONLY,
        MPI_INFO_NULL, &fh);
MPI_File_seek(fh, rank * bufsize, MPI_SEEK_SET);
MPI_File_read(fh, buf, nints, MPI_INT, &status);
MPI_File_close(&fh);
```

MPI-I/O

```
MPI_File fh;
MPI_Status status;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
bufsize = FILESIZE/nprocs;
nints = bufsize/sizeof(int);
MPI_File_open(MPI_COMM_WORLD, “/file", MPI_MODE_RDONLY,
        MPI_INFO_NULL, &fh);
MPI_File_seek(fh, rank * bufsize, MPI_SEEK_SET);
MPI_File_read(fh, buf, nints, MPI_INT, &status);
MPI_File_close(&fh);
```

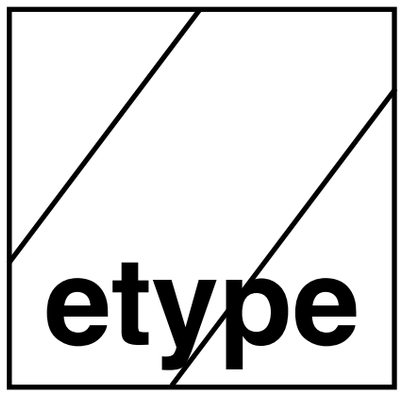
File View

- 3-tuple: <displacement, etype, filetype>
  - byte displacement from the start of the file
  - etype: data unit type
  - filetype: portion of the file visible to the process
- MPI_File_set_view

File View

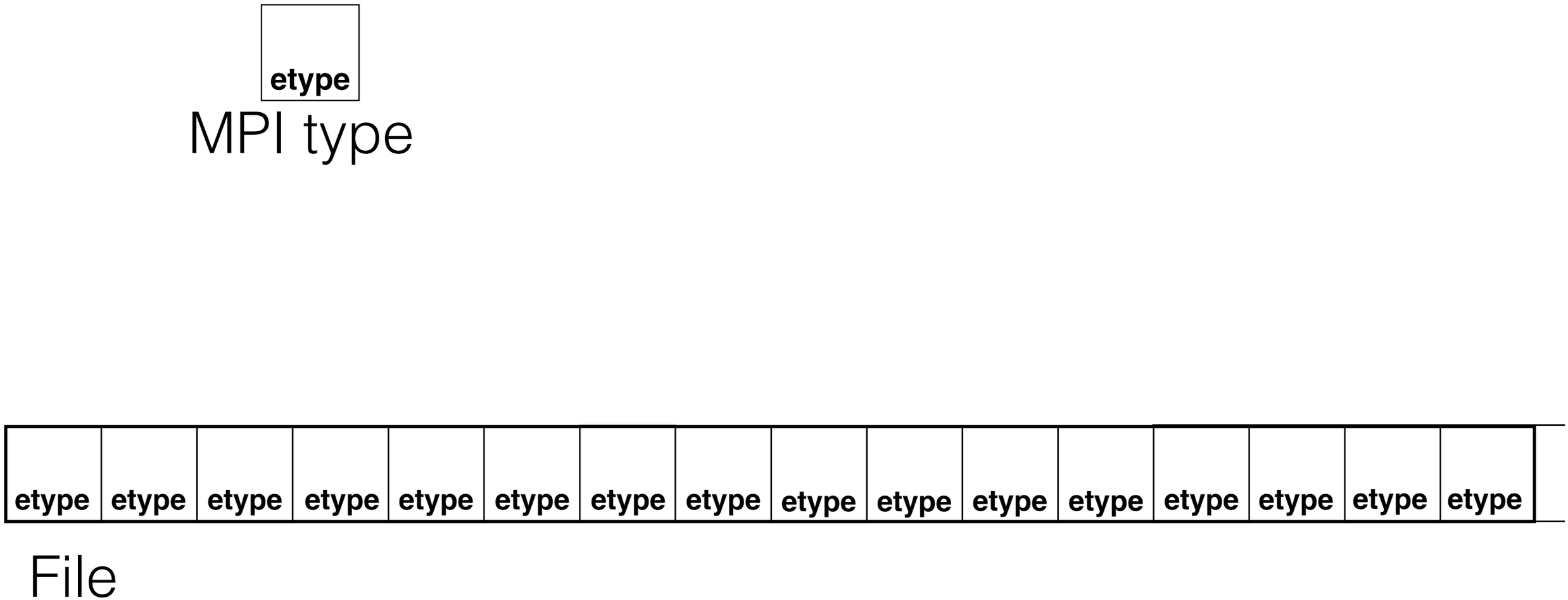
```
int MPI_File_set_view(
      MPI_File fh,
      MPI_Offset disp,              // in bytes
      MPI_Datatype etype,    // file's a sequence of etypes
      MPI_Datatype filetype, // interpret as filetypes
      char *datarep,                // e.g., native, internal, external
      MPI_Info info)
```

File View

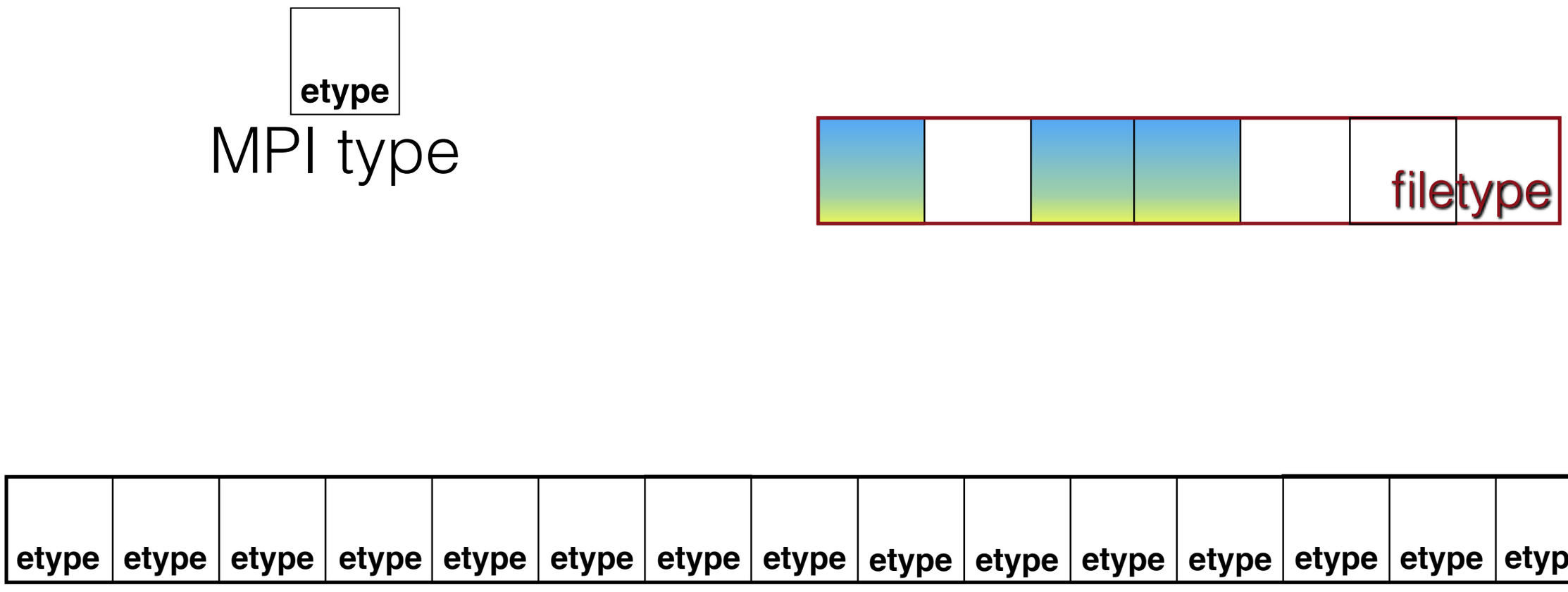


MPI type

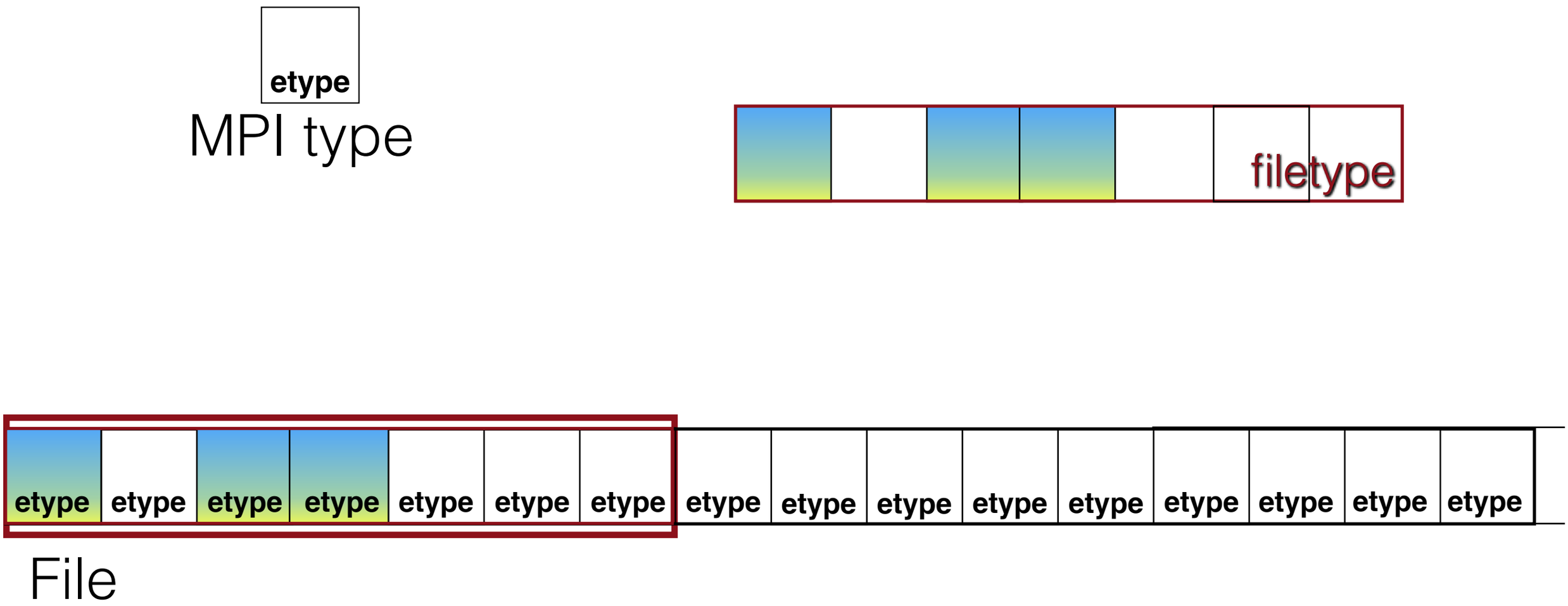
File View


etype
MPI type
etype etype etype etype etype etype etype etype etype etype etype etype etype etype etype etype
File

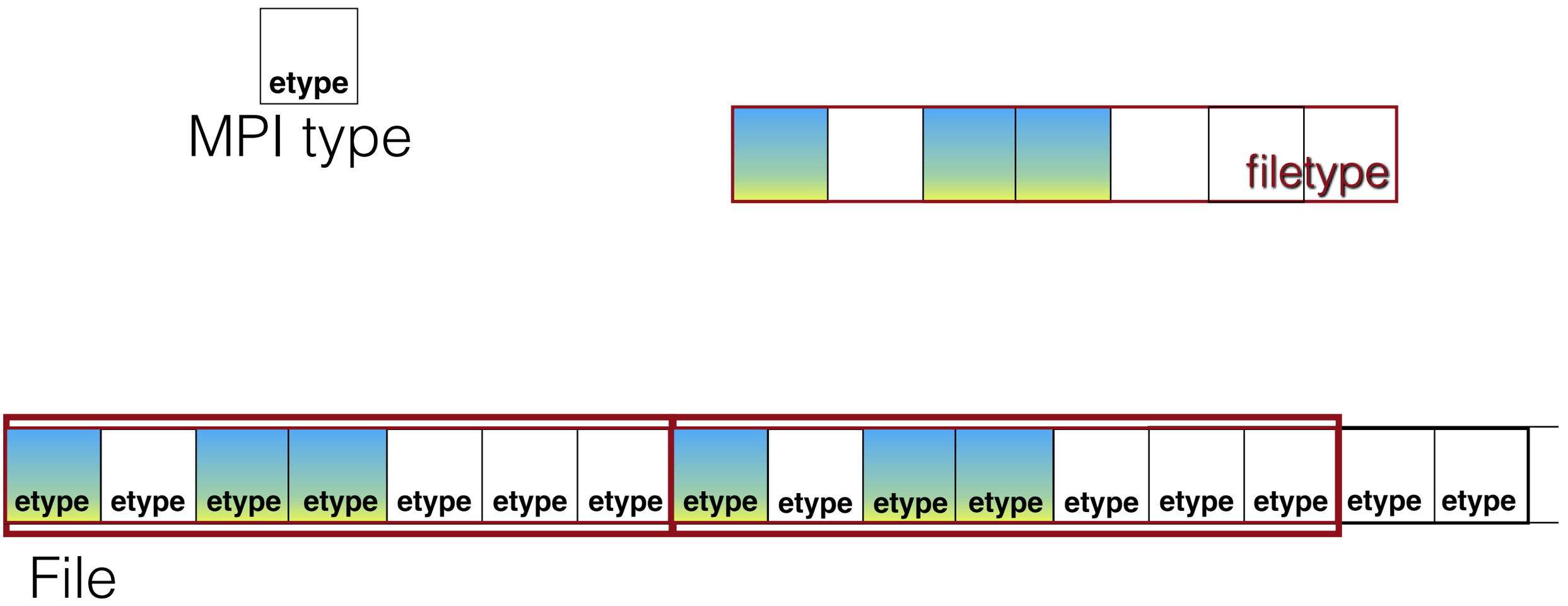
File View


etype
MPI type
filetype
etype etype etype etype etype etype etype etype etype etype etype etype etype etype etype etype
File

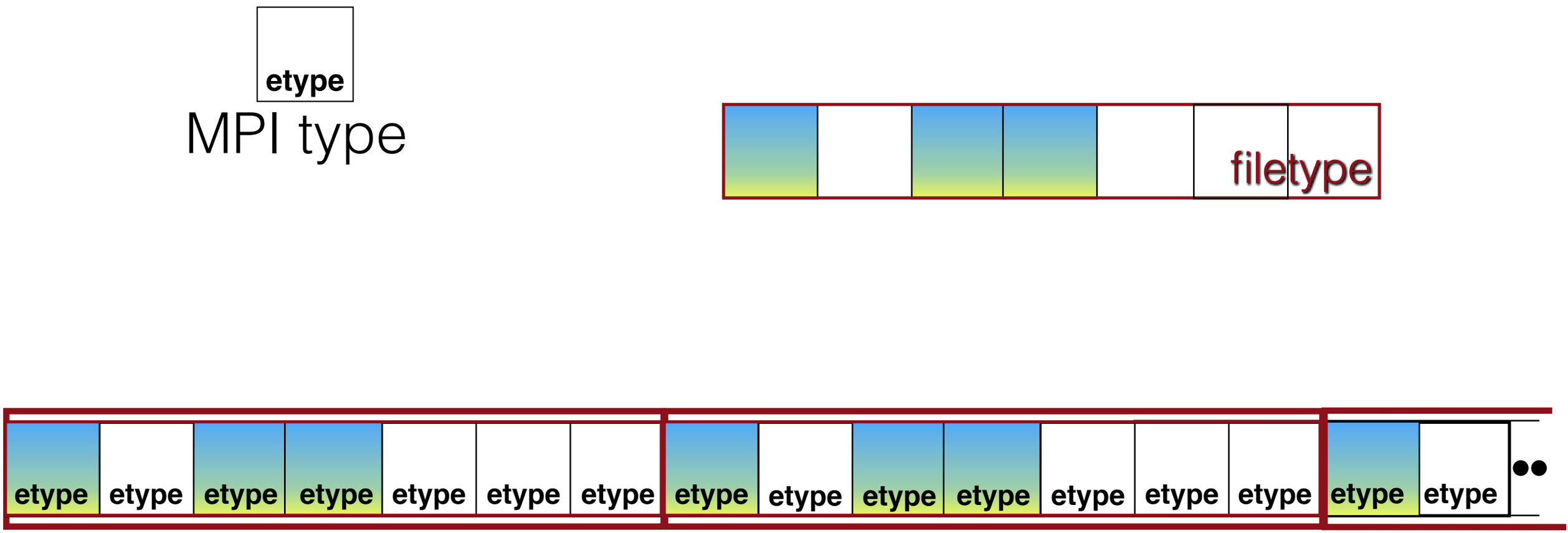
File View


etype
MPI type
filetype
etype etype etype etype etype etype etype etype etype etype etype etype etype etype etype etype etype etype
File

File View


etype
MPI type
filetype
etype etype etype etype etype etype etype etype etype etype etype etype etype etype etype etype etype
File

File View


etype
MPI type
filetype
etype etype etype etype etype etype etype etype etype etype etype etype etype etype etype etype etype etype
File

Example: Write in Set View

```
MPI_File pfile;

for (i=0; i<BUFSIZE; i++)
     buf[i] = rank * BUFSIZE + i;
MPI_File_open(MPI_COMM_WORLD, "/file",
        MPI_MODE_CREATE | MPI_MODE_WRONLY,
        MPI_INFO_NULL, &pfile);
MPI_File_set_view(pfile, myrank * BUFSIZE * sizeof(int),
        MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
MPI_File_write(pfile, buf, BUFSIZE, MPI_INT,
            MPI_STATUS_IGNORE);
MPI_File_close(&pfile);
```

Blocking, Collective

Collective

Blocking, Individual

Example of views

```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, intpair;
MPI_Offset disp;
MPI_Type_contiguous(2, MPI_INT, &intpair);
lb = 0; extent = 6 * sizeof(int);
MPI_Type_create_resized(intpair, lb, extent, &filetype);
MPI_Type_commit(&filetype);
disp = 5 * sizeof(int) + 2 * rank*sizeof(int);
MPI_File_open(MPI_COMM_WORLD, “/scratch/datafile",
      MPI_MODE_CREATE|MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, MPI_INT, filetype, "native",
                        MPI_INFO_NULL);
MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```

Example of views

```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, intpair;
MPI_Offset disp;
MPI_Type_contiguous(2, MPI_INT, &intpair);
lb = 0; extent = 6 * sizeof(int);
MPI_Type_create_resized(intpair, lb, extent, &filetype);
MPI_Type_commit(&filetype);
disp = 5 * sizeof(int) + 2 * rank*sizeof(int);
MPI_File_open(MPI_COMM_WORLD, “/scratch/datafile",
     MPI_MODE_CREATE|MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, MPI_INT, filetype, "native",
                  MPI_INFO_NULL);
MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```

Example of views


```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, intpair;
MPI_Offset disp;
MPI_Type_contiguous(2, MPI_INT, &intpair);
lb = 0; extent = 6 * sizeof(int);
MPI_Type_create_resized(intpair, lb, extent, &filetype);
MPI_Type_commit(&filetype);
disp = 5 * sizeof(int) + 2 * rank*sizeof(int);
MPI_File_open(MPI_COMM_WORLD, “/scratch/datafile",
     MPI_MODE_CREATE|MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, MPI_INT, filetype, "native",
                  MPI_INFO_NULL);
MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```

Example of views


```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, intpair;
MPI_Offset disp;
MPI_Type_contiguous(2, MPI_INT, &intpair);
lb = 0; extent = 6 * sizeof(int);
MPI_Type_create_resized(intpair, lb, extent, &filetype);
MPI_Type_commit(&filetype);
disp = 5 * sizeof(int) + 2 * rank*sizeof(int);
MPI_File_open(MPI_COMM_WORLD, “/scratch/datafile",
     MPI_MODE_CREATE|MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, MPI_INT, filetype, "native",
                  MPI_INFO_NULL);
MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```

Example of views


```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, intpair;
MPI_Offset disp;
MPI_Type_contiguous(2, MPI_INT, &intpair);
lb = 0; extent = 6 * sizeof(int);
MPI_Type_create_resized(intpair, lb, extent, &filetype);
MPI_Type_commit(&filetype);
disp = 5 * sizeof(int) + 2 * rank*sizeof(int);
MPI_File_open(MPI_COMM_WORLD, “/scratch/datafile",
      MPI_MODE_CREATE|MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, MPI_INT, filetype, "native",
                  MPI_INFO_NULL);
MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```

Example of views


```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, intpair;
MPI_Offset disp;
MPI_Type_contiguous(2, MPI_INT, &intpair);
lb = 0; extent = 6 * sizeof(int);
MPI_Type_create_resized(intpair, lb, extent, &filetype);
MPI_Type_commit(&filetype);
disp = 5 * sizeof(int) + 2 * rank*sizeof(int);
MPI_File_open(MPI_COMM_WORLD, “/scratch/datafile",
      MPI_MODE_CREATE|MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, MPI_INT, filetype, "native",
                  MPI_INFO_NULL);
MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```

Example of views


```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, intpair;
MPI_Offset disp;
MPI_Type_contiguous(2, MPI_INT, &intpair);
lb = 0; extent = 6 * sizeof(int);
MPI_Type_create_resized(intpair, lb, extent, &filetype);
MPI_Type_commit(&filetype);
disp = 5 * sizeof(int) + 2 * rank*sizeof(int);
MPI_File_open(MPI_COMM_WORLD, “/scratch/datafile",
      MPI_MODE_CREATE|MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, MPI_INT, filetype, "native",
                  MPI_INFO_NULL);
MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```

Example of views


```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, intpair;
MPI_Offset disp;
MPI_Type_contiguous(2, MPI_INT, &intpair);
lb = 0; extent = 6 * sizeof(int);
MPI_Type_create_resized(intpair, lb, extent, &filetype);
MPI_Type_commit(&filetype);
disp = 5 * sizeof(int) + 2 * rank*sizeof(int);
MPI_File_open(MPI_COMM_WORLD, “/scratch/datafile",
      MPI_MODE_CREATE|MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, MPI_INT, filetype, "native",
                  MPI_INFO_NULL);
MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```

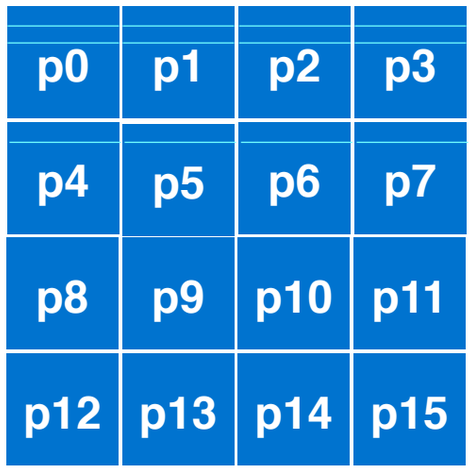
Example of views


```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, intpair;
MPI_Offset disp;
MPI_Type_contiguous(2, MPI_INT, &intpair);
lb = 0; extent = 6 * sizeof(int);
MPI_Type_create_resized(intpair, lb, extent, &filetype);
MPI_Type_commit(&filetype);
disp = 5 * sizeof(int) + 2 * rank*sizeof(int);
MPI_File_open(MPI_COMM_WORLD, “/scratch/datafile",
      MPI_MODE_CREATE|MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, MPI_INT, filetype, "native",
                  MPI_INFO_NULL);
MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```

Multiple Reads *vs* Single Read

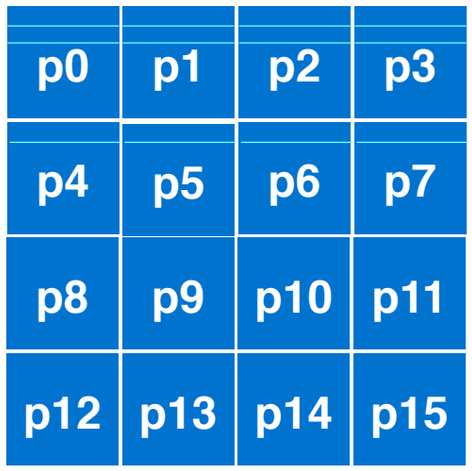
p0	p1	p2	p3
p4	p5	p6	p7
p8	p9	p10	p11
p12	p13	p14	p15

Multiple Reads *vs* Single Read



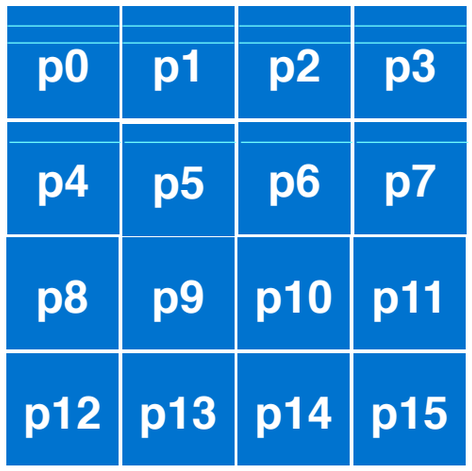
p0 p1 p2 p3 p0 p1 p2 p3 … p4 p5 p6 p7 p4 p5 p6 p7 …

Multiple Reads *vs* Single Read



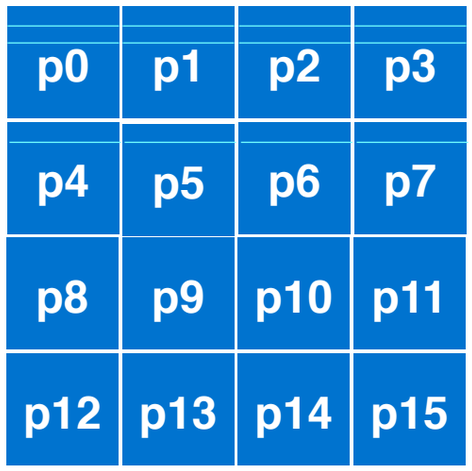
p0 p1 p2 p3 p0 p1 p2 p3 … p4 p5 p6 p7 p4 p5 p6 p7 …

Multiple Reads *vs* Single Read

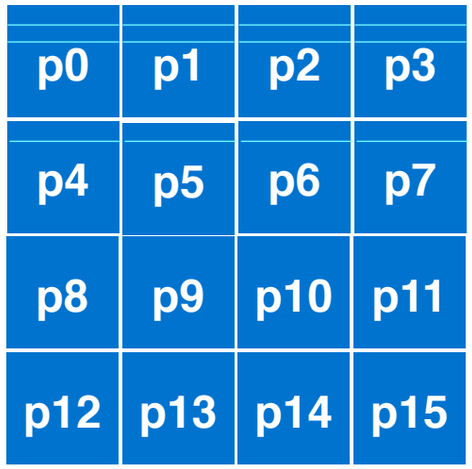


p0 p1 p2 p3 p0 p1 p2 p3 … p4 p5 p6 p7 p4 p5 p6 p7 …

Multiple Reads *vs* Single Read



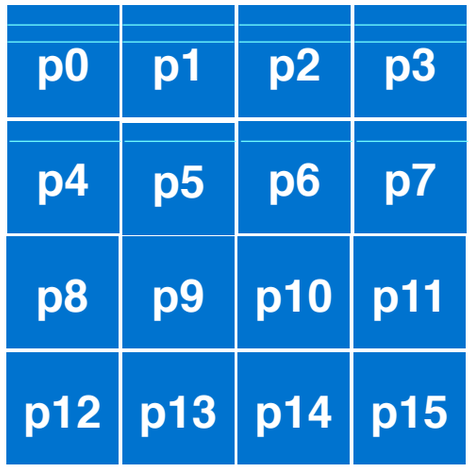
Multiple Reads *vs* Single Read



p0 p1 p2 p3 p0 p1 p2 p3 … p4 p5 p6 p7 p4 p5 p6 p7 …

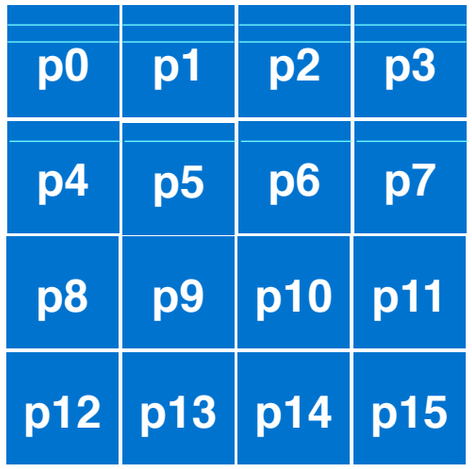
```
MPI_File_open(MPI_COMM_WORLD, file, ..., &fh);
  for (i=0; i<n_local_rows; i++) {
     MPI_File_seek(fh, ...);
     MPI_File_read    (fh, &(A[i][0]), ...);
}
MPI_File_close(&fh);
```

Multiple Reads *vs* Single Read




```
MPI_File_open(MPI_COMM_WORLD, file, ..., &fh);
  for (i=0; i<n_local_rows; i++) {
     MPI_File_seek(fh, ...);
     MPI_File_read_all(fh, &(A[i][0]), ...);
}
MPI_File_close(&fh);
```

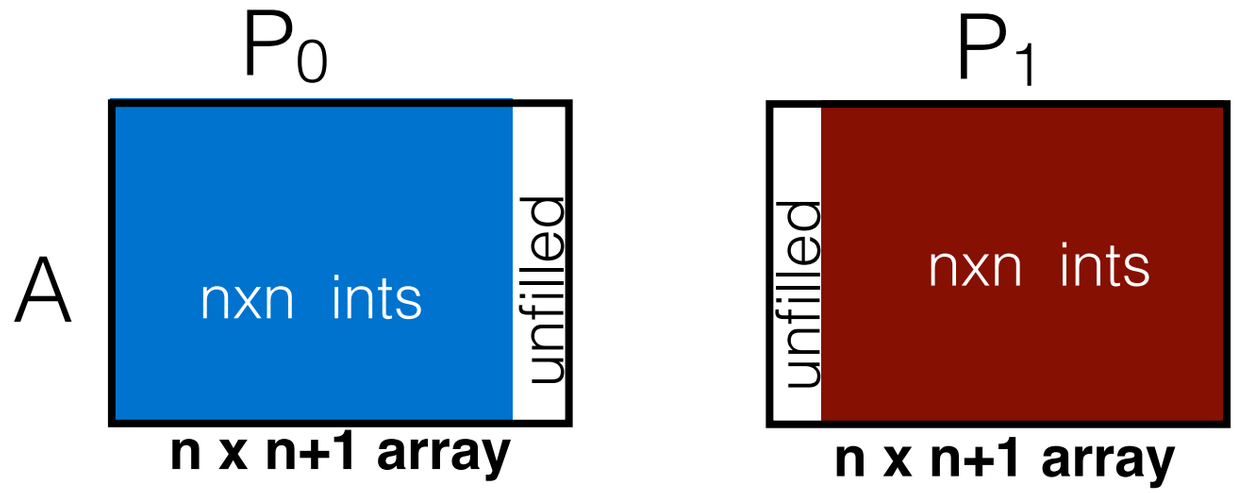
Multiple Reads *vs* Single Read




```
MPI_File_open(MPI_COMM_WORLD, file, ..., &fh);
  for (i=0; i<n_local_rows; i++) {
     MPI_File_seek(fh, ...);
     MPI_File_read_all(fh, &(A[i][0]), ...);
}
MPI_File_close(&fh);
```

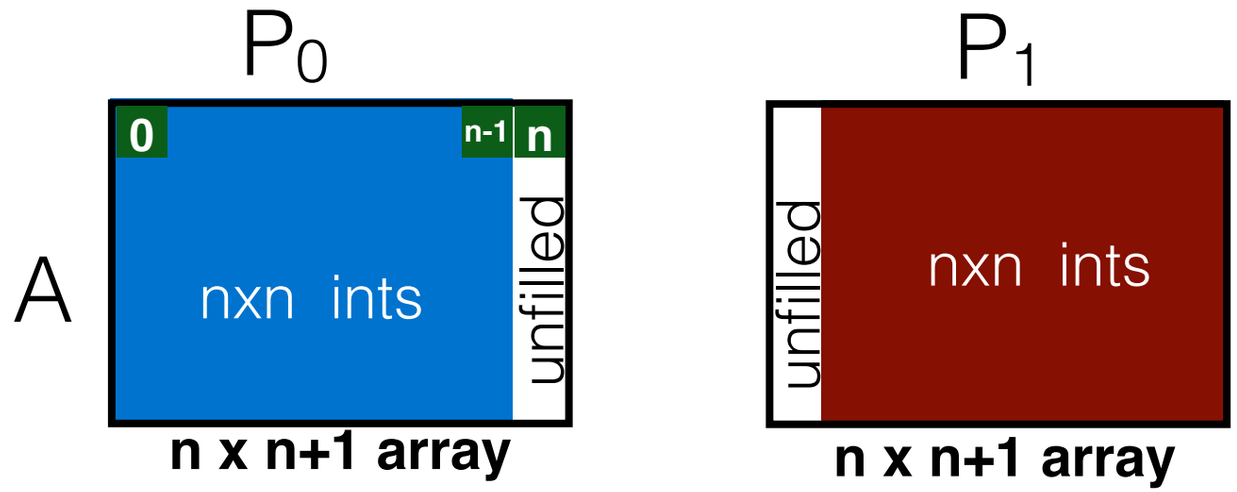
```
MPI_Type_create_subarray(2,..., &subarray);
MPI_Type_commit(&subarray);
MPI_File_open(MPI_COMM_WORLD, file,..., &fh);
MPI_File_set_view(fh, ..., subarray, ...);
MPI_File_read_all(fh, A, ...);
MPI_File_close(&fh);
```

Data Transfer



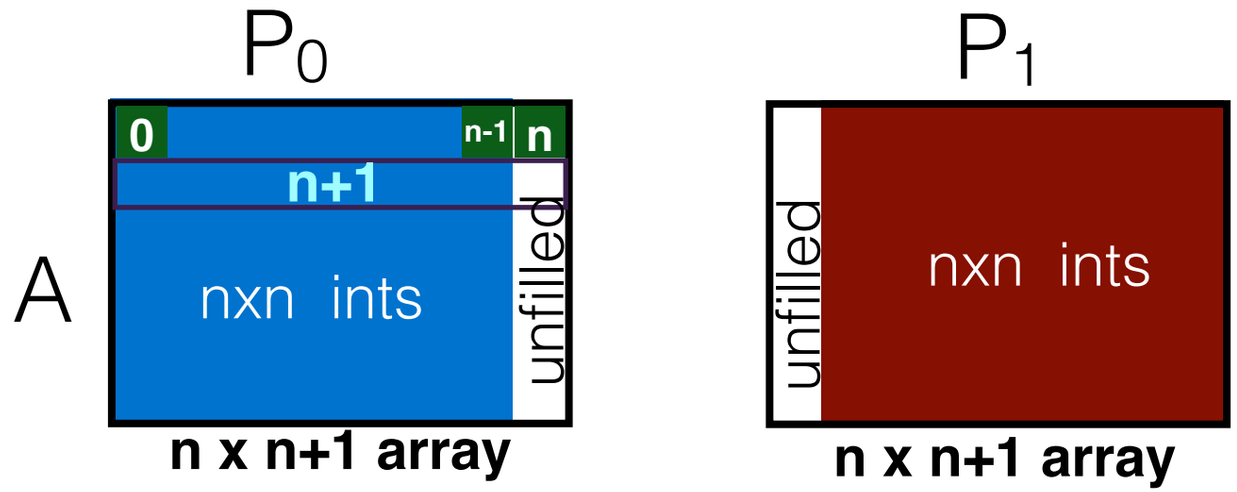

```
MPI_Status status;
MPI_Datatype column;
MPI_Type_vector(n, 1, n+1, MPI_INT, &column);
MPI_Type_commit(&column);
if(rank == 0) {
   MPI_Send(A+n-1, 1, column, 1, 99, MPI_COMM_WORLD);
   MPI_Recv(A+n, 1, column, 1, 99, MPI_COMM_WORLD, &status)
}
if(rank == 1) {
   MPI_Recv(A, 1, column, 0, 99, MPI_COMM_WORLD, &status);
   MPI_Send(A+1, 1, column, 0, 99, MPI_COMM_WORLD);
}
```

Data Transfer



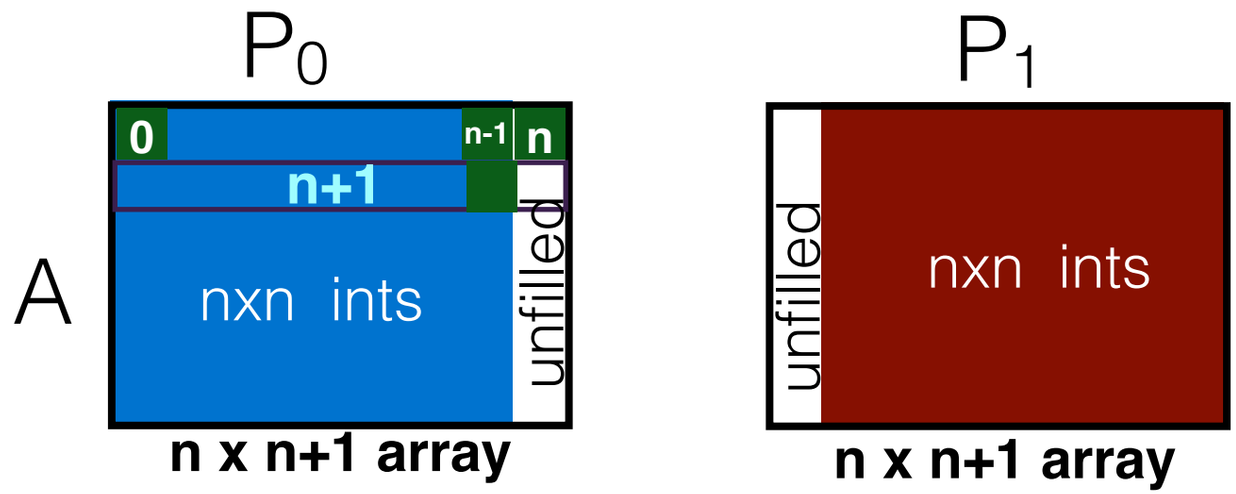

```
MPI_Status status;
MPI_Datatype column;
MPI_Type_vector(n, 1, n+1, MPI_INT, &column);
MPI_Type_commit(&column);
if(rank == 0) {
   MPI_Send(A+n-1, 1, column, 1, 99, MPI_COMM_WORLD);
   MPI_Recv(A+n, 1, column, 1, 99, MPI_COMM_WORLD, &status)
}
if(rank == 1) {
   MPI_Recv(A, 1, column, 0, 99, MPI_COMM_WORLD, &status);
   MPI_Send(A+1, 1, column, 0, 99, MPI_COMM_WORLD);
}
```

Data Transfer



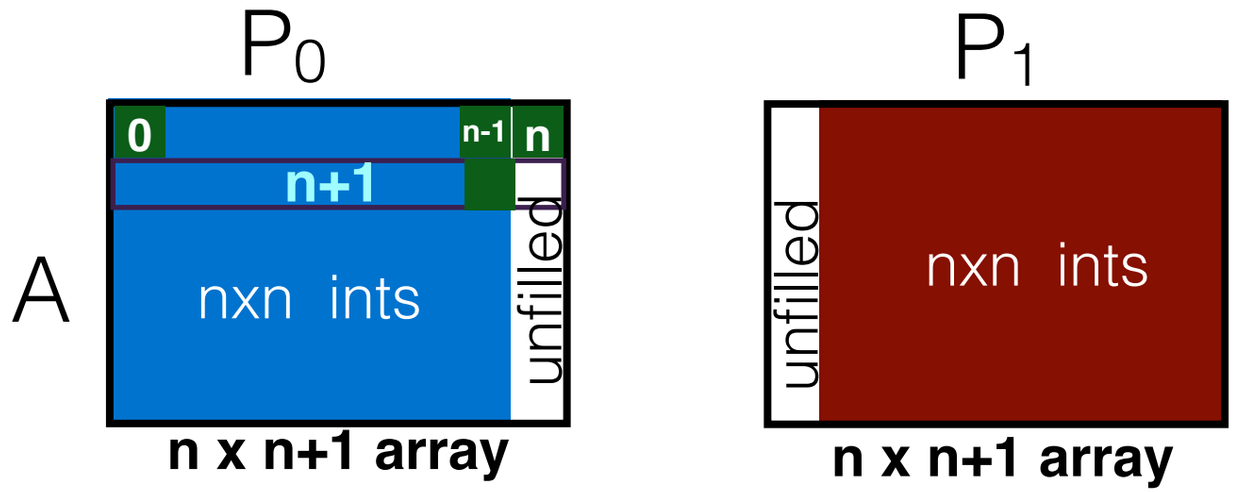

```
MPI_Status status;
MPI_Datatype column;
MPI_Type_vector(n, 1, n+1, MPI_INT, &column);
MPI_Type_commit(&column);
if(rank == 0) {
   MPI_Send(A+n-1, 1, column, 1, 99, MPI_COMM_WORLD);
   MPI_Recv(A+n, 1, column, 1, 99, MPI_COMM_WORLD, &status)
}
if(rank == 1) {
   MPI_Recv(A, 1, column, 0, 99, MPI_COMM_WORLD, &status);
   MPI_Send(A+1, 1, column, 0, 99, MPI_COMM_WORLD);
}
```

Data Transfer



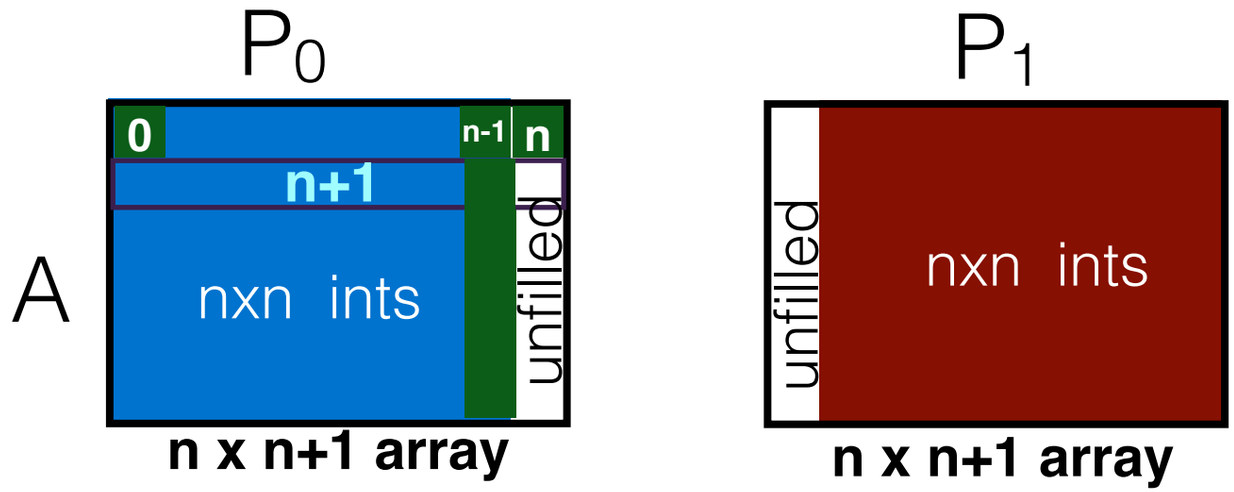

```
MPI_Status status;
MPI_Datatype column;
MPI_Type_vector(n, 1, n+1, MPI_INT, &column);
MPI_Type_commit(&column);
if(rank == 0) {
   MPI_Send(A+n-1, 1, column, 1, 99, MPI_COMM_WORLD);
   MPI_Recv(A+n, 1, column, 1, 99, MPI_COMM_WORLD, &status)
}
if(rank == 1) {
   MPI_Recv(A, 1, column, 0, 99, MPI_COMM_WORLD, &status);
   MPI_Send(A+1, 1, column, 0, 99, MPI_COMM_WORLD);
}
```

Data Transfer



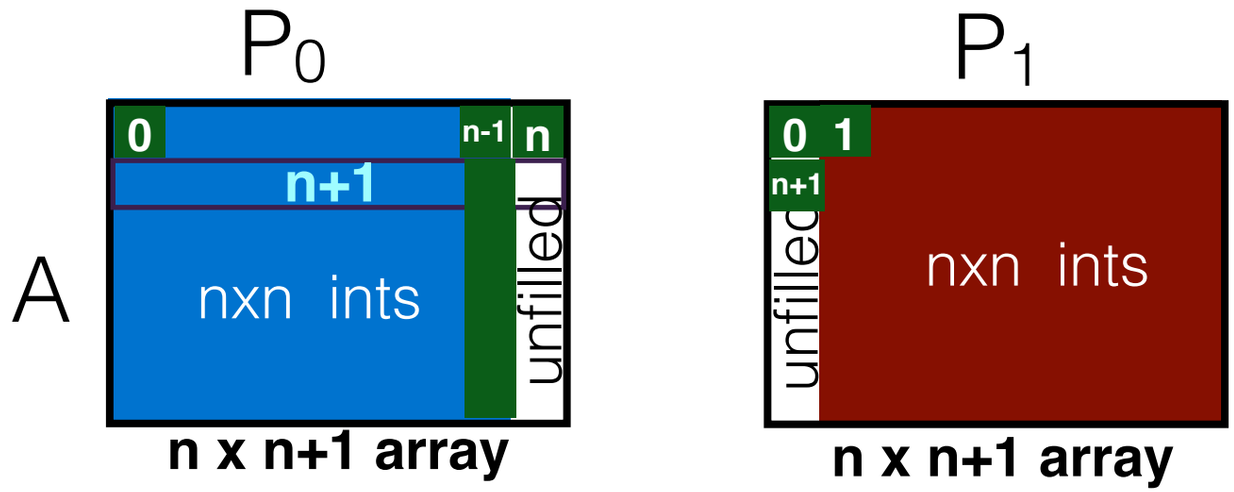

```
MPI_Status status;
MPI_Datatype column;
MPI_Type_vector(n, 1, n+1, MPI_INT, &column);
MPI_Type_commit(&column);
if(rank == 0) {
   MPI_Send(A+n-1, 1, column, 1, 99, MPI_COMM_WORLD);
   MPI_Recv(A+n, 1, column, 1, 99, MPI_COMM_WORLD, &status)
}
if(rank == 1) {
   MPI_Recv(A, 1, column, 0, 99, MPI_COMM_WORLD, &status);
   MPI_Send(A+1, 1, column, 0, 99, MPI_COMM_WORLD);
}
```

Data Transfer



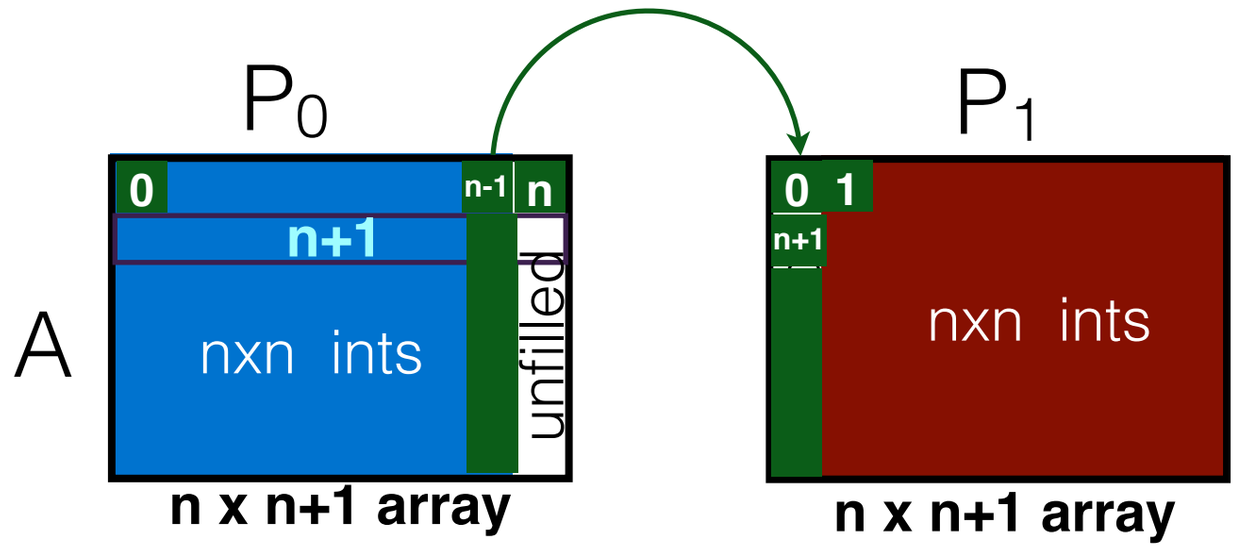

```
MPI_Status status;
MPI_Datatype column;
MPI_Type_vector(n, 1, n+1, MPI_INT, &column);
MPI_Type_commit(&column);
if(rank == 0) {
   MPI_Send(A+n-1, 1, column, 1, 99, MPI_COMM_WORLD);
   MPI_Recv(A+n, 1, column, 1, 99, MPI_COMM_WORLD, &status)
}
if(rank == 1) {
   MPI_Recv(A, 1, column, 0, 99, MPI_COMM_WORLD, &status);
   MPI_Send(A+1, 1, column, 0, 99, MPI_COMM_WORLD);
}
```

Data Transfer



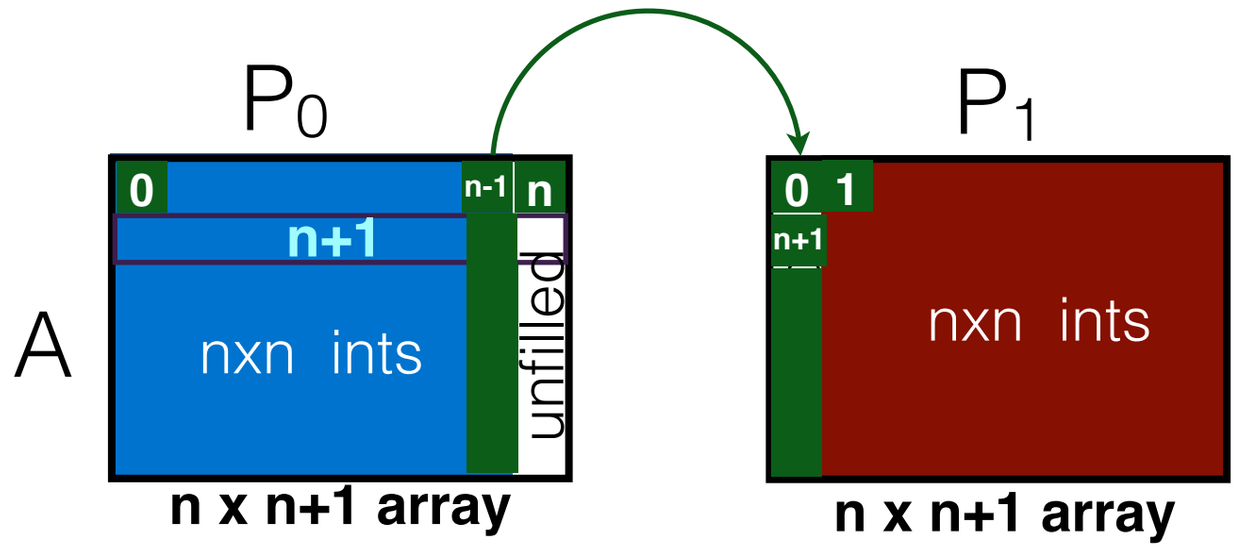

```
MPI_Status status;
MPI_Datatype column;
MPI_Type_vector(n, 1, n+1, MPI_INT, &column);
MPI_Type_commit(&column);
if(rank == 0) {
   MPI_Send(A+n-1, 1, column, 1, 99, MPI_COMM_WORLD);
   MPI_Recv(A+n, 1, column, 1, 99, MPI_COMM_WORLD, &status)
}
if(rank == 1) {
   MPI_Recv(A, 1, column, 0, 99, MPI_COMM_WORLD, &status);
   MPI_Send(A+1, 1, column, 0, 99, MPI_COMM_WORLD);
}
```

Data Transfer



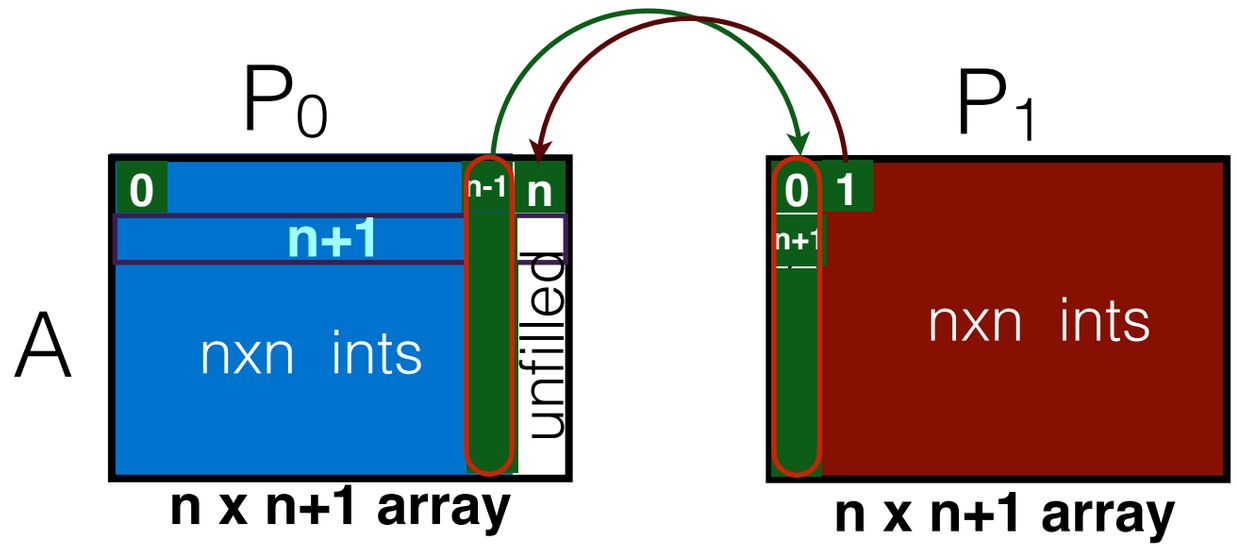

```
MPI_Status status;
MPI_Datatype column;
MPI_Type_vector(n, 1, n+1, MPI_INT, &column);
MPI_Type_commit(&column);
if(rank == 0) {
   MPI_Send(A+n-1, 1, column, 1, 99, MPI_COMM_WORLD);
   MPI_Recv(A+n, 1, column, 1, 99, MPI_COMM_WORLD, &status)
}
if(rank == 1) {
   MPI_Recv(A, 1, column, 0, 99, MPI_COMM_WORLD, &status);
   MPI_Send(A+1, 1, column, 0, 99, MPI_COMM_WORLD);
}
```

Data Transfer



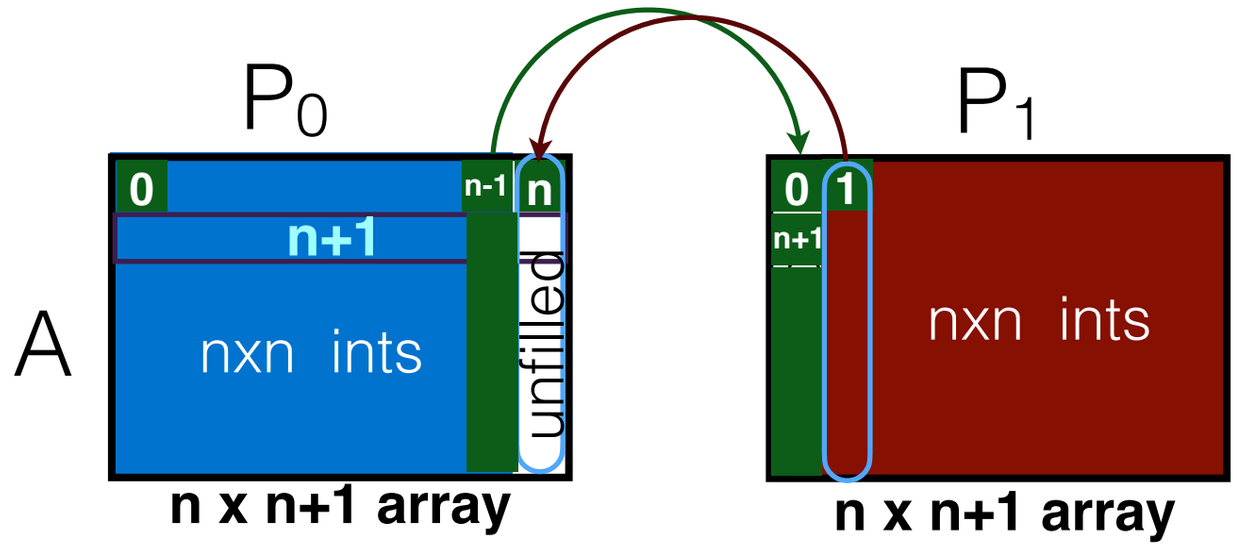

```
MPI_Status status;
MPI_Datatype column;
MPI_Type_vector(n, 1, n+1, MPI_INT, &column);
MPI_Type_commit(&column);
if(rank == 0) {
    MPI_Send(A+n-1, 1, column, 1, 99, MPI_COMM_WORLD);
    MPI_Recv(A+n, 1, column, 1, 99, MPI_COMM_WORLD, &status)
}
if(rank == 1) {
    MPI_Recv(A, 1, column, 0, 99, MPI_COMM_WORLD, &status);
    MPI_Send(A+1, 1, column, 0, 99, MPI_COMM_WORLD);
}
```

Data Transfer



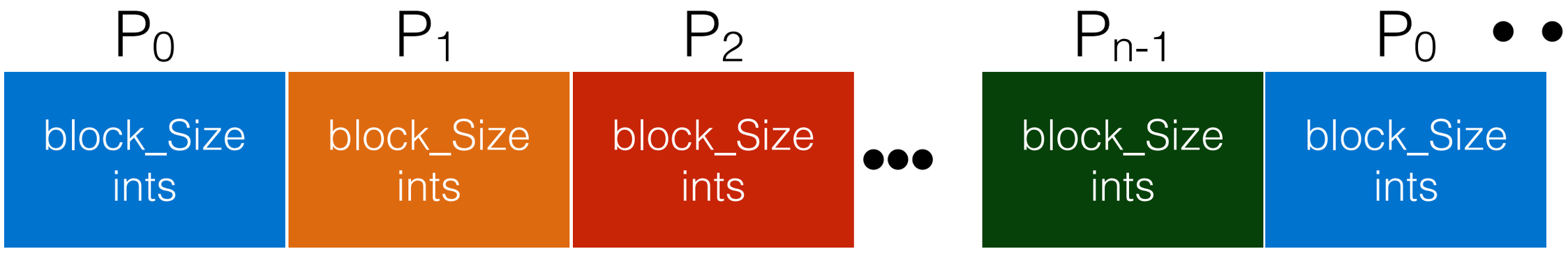

```
MPI_Status status;
MPI_Datatype column;
MPI_Type_vector(n, 1, n+1, MPI_INT, &column);
MPI_Type_commit(&column);
if(rank == 0) {
    MPI_Send(A+n-1, 1, column, 1, 99, MPI_COMM_WORLD);
    MPI_Recv(A+n, 1, column, 1, 99, MPI_COMM_WORLD, &status)
}
if(rank == 1) {
    MPI_Recv(A, 1, column, 0, 99, MPI_COMM_WORLD, &status);
    MPI_Send(A+1, 1, column, 0, 99, MPI_COMM_WORLD);
}
```

Data Transfer



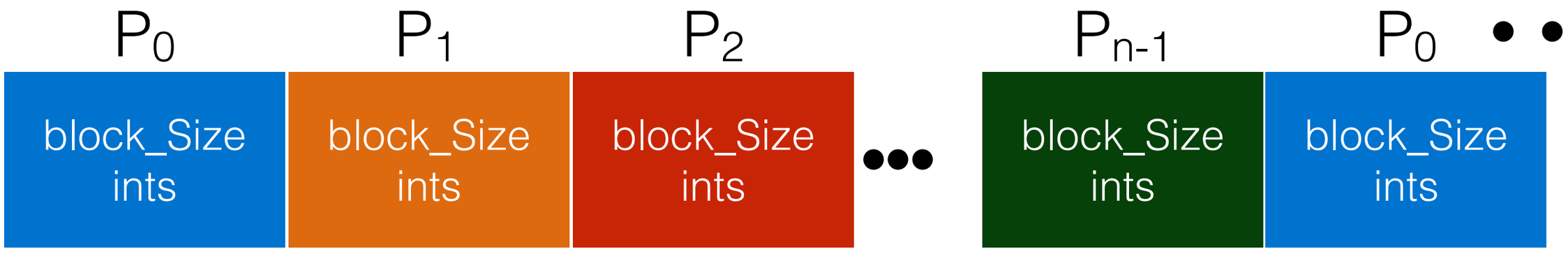

```
MPI_Status status;
MPI_Datatype column;
MPI_Type_vector(n, 1, n+1, MPI_INT, &column);
MPI_Type_commit(&column);
if(rank == 0) {
    MPI_Send(A+n-1, 1, column, 1, 99, MPI_COMM_WORLD);
    MPI_Recv(A+n, 1, column, 1, 99, MPI_COMM_WORLD, &status)
}
if(rank == 1) {
    MPI_Recv(A, 1, column, 0, 99, MPI_COMM_WORLD, &status);
    MPI_Send(A+1, 1, column, 0, 99, MPI_COMM_WORLD);
}
```

Read from each processor in parallel




```
MPI_Aint lb, extent;
MPI_Datatype filetype, block;
MPI_File fh;
MPI_Status status;
MPI_Type_contiguous(block_Size, MPI_INT, &block);
lb = 0; extent = block_Size * sizeof(int);
MPI_Type_create_resized(block, lb, extent, &filetype);
MPI_Type_commit(&filetype);
MPI_Offset disp = block_Size * rank * sizeof(int);
MPI_File_open(MPI_COMM_WORLD,"filename", MPI_MODE_RDONLY,
              MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, MPI_INT, filetype, "native",
                  MPI_INFO_NULL);
MPI_File_read(fh, buf, filesize/n, MPI_INT, &status);
```

Read from each processor in parallel



One Read per processor

```
MPI_Aint lb, extent;
MPI_Datatype filetype, block;
MPI_File fh;
MPI_Status status;
MPI_Type_contiguous(block_Size, MPI_INT, &block);
lb = 0; extent = block_Size * sizeof(int);
MPI_Type_create_resized(block, lb, extent, &filetype);
MPI_Type_commit(&filetype);
MPI_Offset disp = block_Size * rank * sizeof(int);
MPI_File_open(MPI_COMM_WORLD,"filename", MPI_MODE_RDONLY,
              MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, MPI_INT, filetype, "native",
                  MPI_INFO_NULL);
MPI_File_read(fh, buf, filesize/n, MPI_INT, &status);
```

IO Variants

- Location

  MPI_File_read_**at**(fh, offset, buffer, count, datatype, &status)

- Non-blocking

  MPI_File_**i**read(fh, buffer, count, datatype, &request)

- Collective

  MPI_File_read_**all**(fh, buffer, count, datatype, &status)

- Shared File pointer (Common data IO)

  MPI_File_read_**shared**(fh, buffer, count, datatype, &status)

  MPI_File_read_**ordered** (fh, buffer, count, datatype, &status)

MPI Review

- Communication semantics
  - Standard, Buffered, Synchronous, Ready
  - Blocking and Non-blocking
- Collective operations
  - Including Reduce and Scan
- Synchronization
  - Barrier
- Trade-off among various communication modes
- Implementation strategies for different communication modes
- Derived data types
  - Copy-eliminating data transfer through “data holes”
- One-sided communication (Get/Put)
- Parallel file IO

Distributed Computing Frameworks

- Remote function execution
  - launch tasks
- Futures, atomics and synchronization
  - Non-blocking launches
  - Test, Wait
- Parallel loops, reductions, and maps
- Distributed data
  - Communications
  - Active Messages
  - Global addresses space

Chapel

Examples from Cray

Chapel

```
const D = {1..m, 1..n};
forall (i,j) in D
    do A[i,j] = i + j/10.0;
```

Examples from Cray

Chapel

Data Parallelism

```
const D = {1..m, 1..n};
forall (i,j) in D
    do A[i,j] = i + j/10.0;
```

Examples from Cray

Chapel

Data Parallelism

```
const D = {1..m, 1..n};
forall (i,j) in D
    do A[i,j] = i + j/10.0;
```

- Sequential
- Work Sharing
- Task launching

Examples from Cray

Chapel

begin taskFn(arg1, arg2);

Data Parallelism

```
const D = {1..m, 1..n};
forall (i,j) in D
    do A[i,j] = i + j/10.0;
```

- Sequential
- Work Sharing
- Task launching

Examples from Cray

Chapel

Task Parallelism

begin taskFn(arg1, arg2);

Data Parallelism

```
const D = {1..m, 1..n};
forall (i,j) in D
    do A[i,j] = i + j/10.0;
```

- Sequential
- Work Sharing
- Task launching

Examples from Cray

Chapel

Data Parallelism

```
const D = {1..m, 1..n};
forall (i,j) in D
    do A[i,j] = i + j/10.0;
```

- Sequential
- Work Sharing
- Task launching

Task Parallelism

```
begin taskFn(arg1, arg2);

cobegin {
    producerFn(1);
    producerFn(2);
    consumerFn(1);
} //implicit join
```

Examples from Cray

Chapel

Data Parallelism

```
const D = {1..m, 1..n};
forall (i,j) in D
    do A[i,j] = i + j/10.0;
```

- Sequential
- Work Sharing
- Task launching

Task Parallelism

```
begin taskFn(arg1, arg2);

cobegin {
    producerFn(1);
    producerFn(2);
    consumerFn(1);
} //implicit join
```

Remote execution

```
on Locales[1] do Fn(args);
```

Examples from Cray

Chapel

Data Parallelism

```
const D = {1..m, 1..n};
forall (i,j) in D
    do A[i,j] = i + j/10.0;
```

- Sequential
- Work Sharing
- Task launching

Task Parallelism

```
begin taskFn(arg1, arg2);

cobegin {
    producerFn(1);
    producerFn(2);
    consumerFn(1);
} //implicit join
```

Remote execution

```
on Locales[1] do Fn(args);

begin on Locales[1] do Fn1(args);
on Locales[2] do begin Fn2(args);
Fn0(args);
```

Examples from Cray

Chapel

Data Parallelism

```
const D = {1..m, 1..n};
forall (i,j) in D
    do A[i,j] = i + j/10.0;
```

- Sequential
- Work Sharing
- Task launching

Task Parallelism

```
begin taskFn(arg1, arg2);

cobegin {
    producerFn(1);
    producerFn(2);
    consumerFn(1);
} //implicit join
```

Remote execution

```
on Locales[1] do Fn(args);

begin on Locales[1] do Fn1(args);
on Locales[2] do begin Fn2(args);
Fn0(args);
```

Future

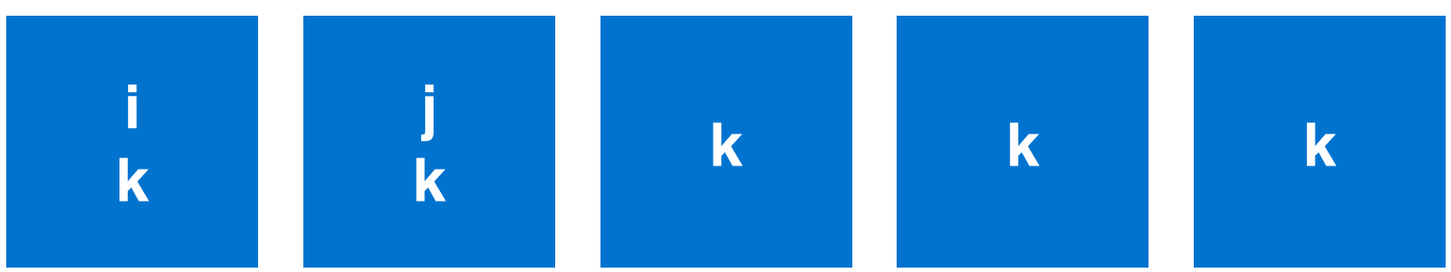
```
begin future$ = compute();
computeSomethingElse();
doSomethingwith(future$);
```

Examples from Cray

Chapel (PGAS)

Partitioned Global Address Space

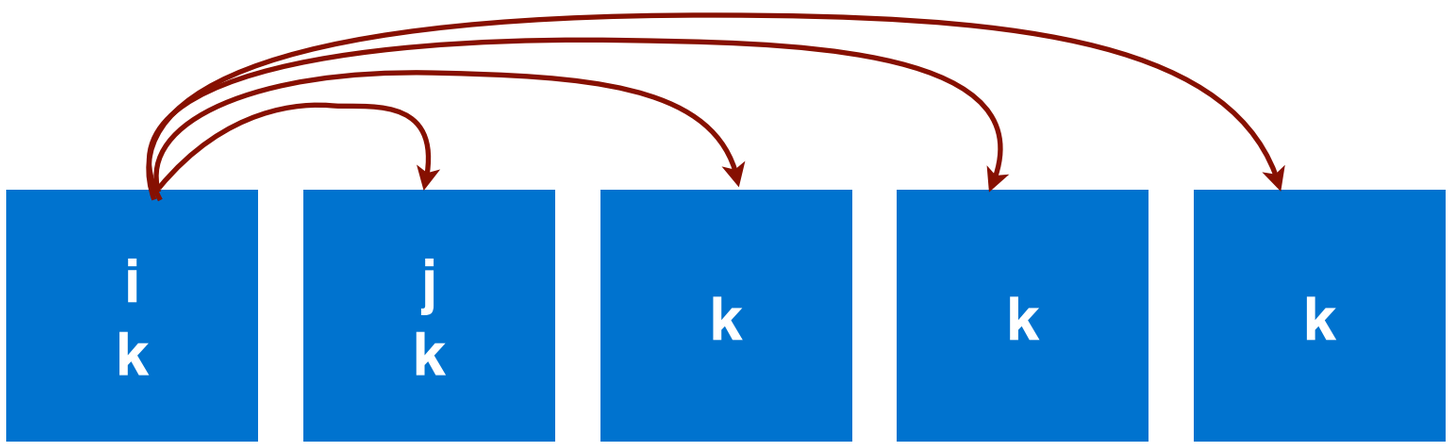
```
var i: int;
on Locales[1] {
   var j: int;
   coforall loc in Locales {
      on loc {
         var k: int;
         k = 2*i + j;
      }
   }
}
```



Chapel (PGAS)

Partitioned Global Address Space

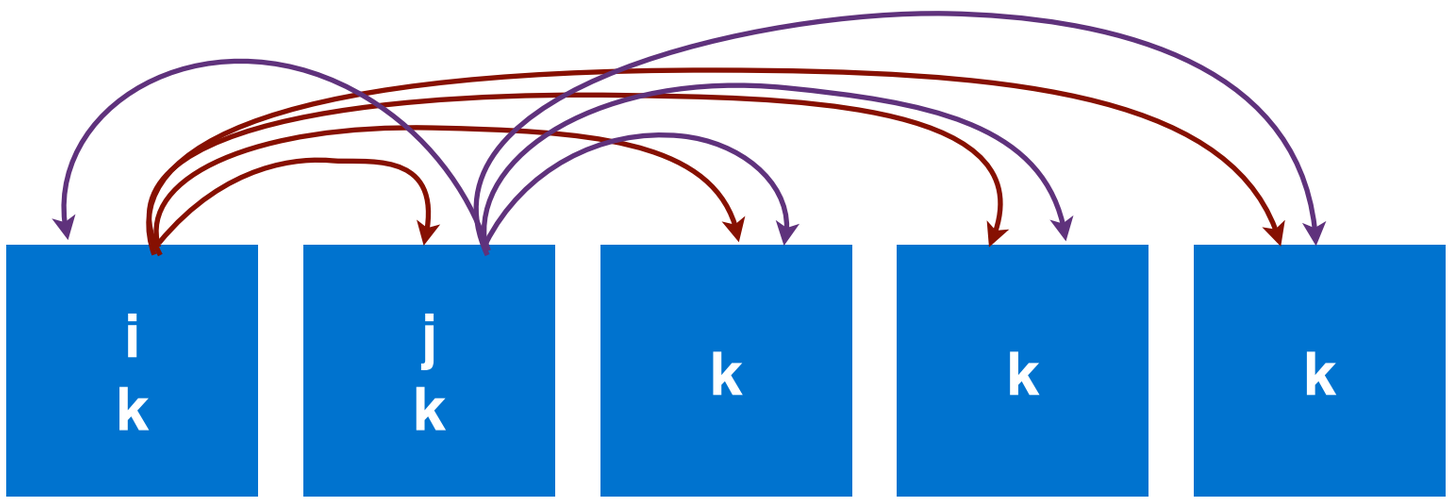
```
var i: int;
on Locales[1] {
   var j: int;
   coforall loc in Locales {
      on loc {
         var k: int;
         k = 2*i + j;
      }
   }
}
```



Chapel (PGAS)

Partitioned Global Address Space

```
var i: int;
on Locales[1] {
   var j: int;
   coforall loc in Locales {
      on loc {
         var k: int;
         k = 2*i + j;
      }
   }
}
```



Map-Reduce Programming Framework

- Data-parallel programming model
- Runtime system
  - Manage parallel processing, data transfers
  - Fault tolerant
- User provides **Map()** and **Reduce()** functions
- Data is mainly represented as <key, value> pairs
- **Map()** processes one *data item*
  - Can produce several data items
- **Reduce()** combines multiple data items
  - Can produce multiple data items
- System determines how and where map() and reduce() are called
  - All to all data exchange between Mappers and Reducers

Map and Reduce

- Map: (K_{in}, V_{in}) ➔ **list**(K_{tmp}, V_{tmp})
- Reduce: (K_{tmp}, **list**(V_{tmp})) ➔ **list**(K_{out}, V_{out})
- Input and output in a set of files

Runtime

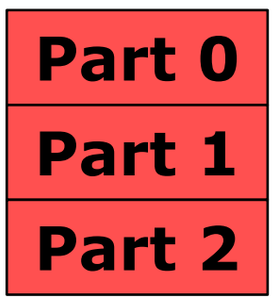
- Allocate a set of input key values (K_{in},V_{in}) to *Map-processors*
  - Furnish a section of the input
- Map-processor calls user's **Map()** function independently of others
  - Once for each K_{in},V_{in}, generating output organized by K_{tmp}
- Allocate intermediate key values (K_{tmp}) to *Reduce-processors*
  - "Alltoall" the intermediate values generated by all mappers
- Reduce-processor calls user-provided **Reduce()** function
  - Once for each K_{tmp} value produced by the Mappers
- Produce the final output
  - Collects the Reduce output, sorted by K_{out} values

Pipeline

Configure nodes, input etc

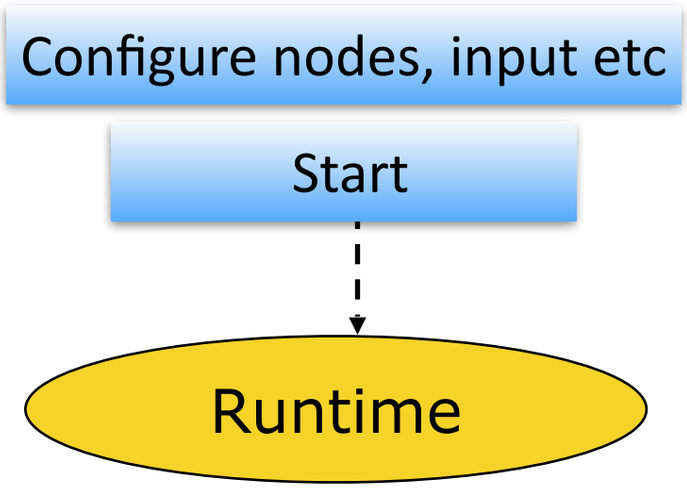
Pipeline

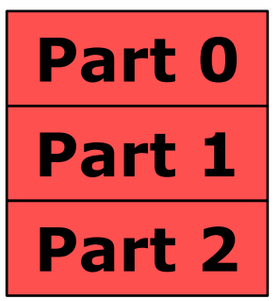
Configure nodes, input etc



Input files

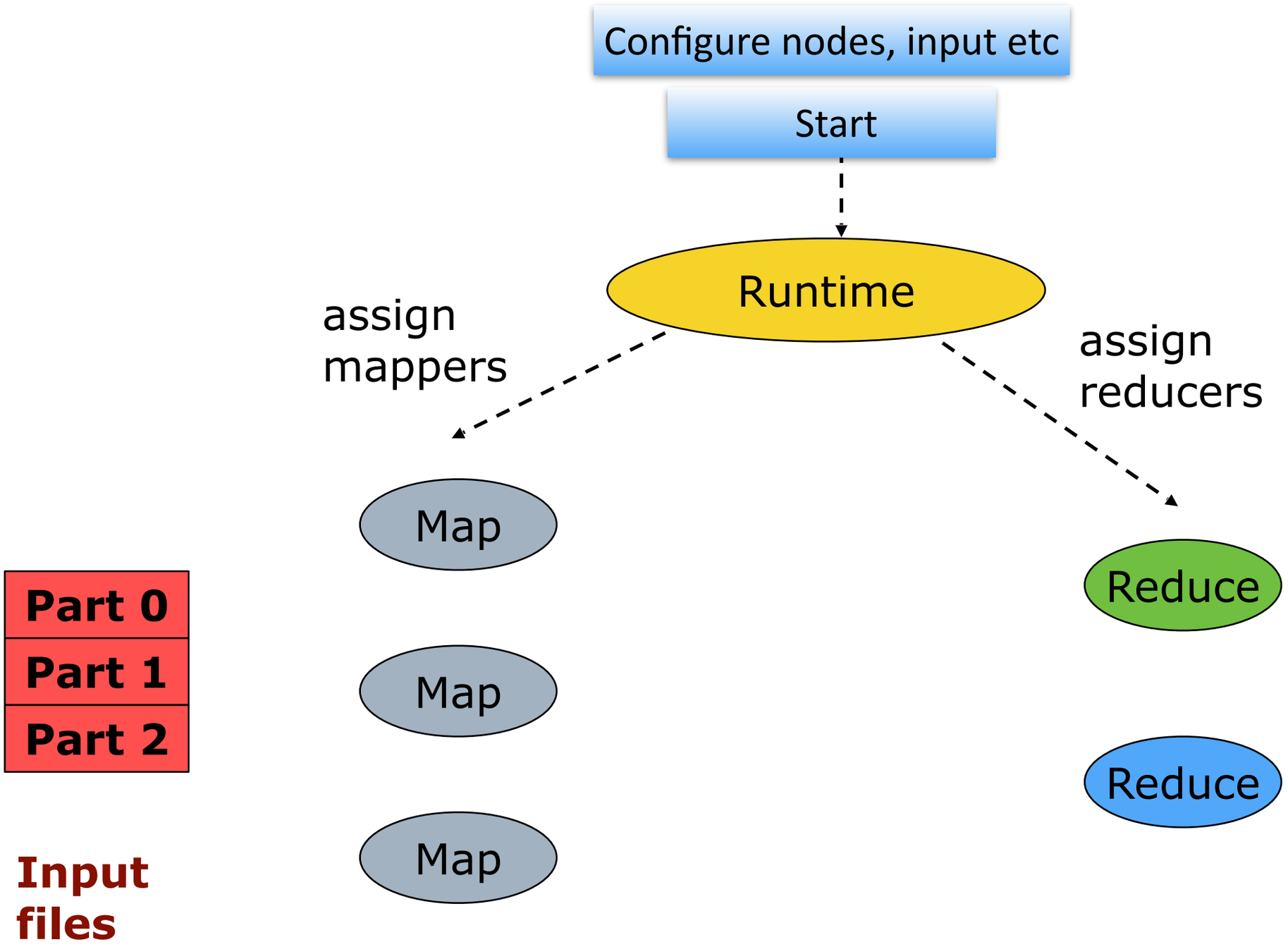
Pipeline



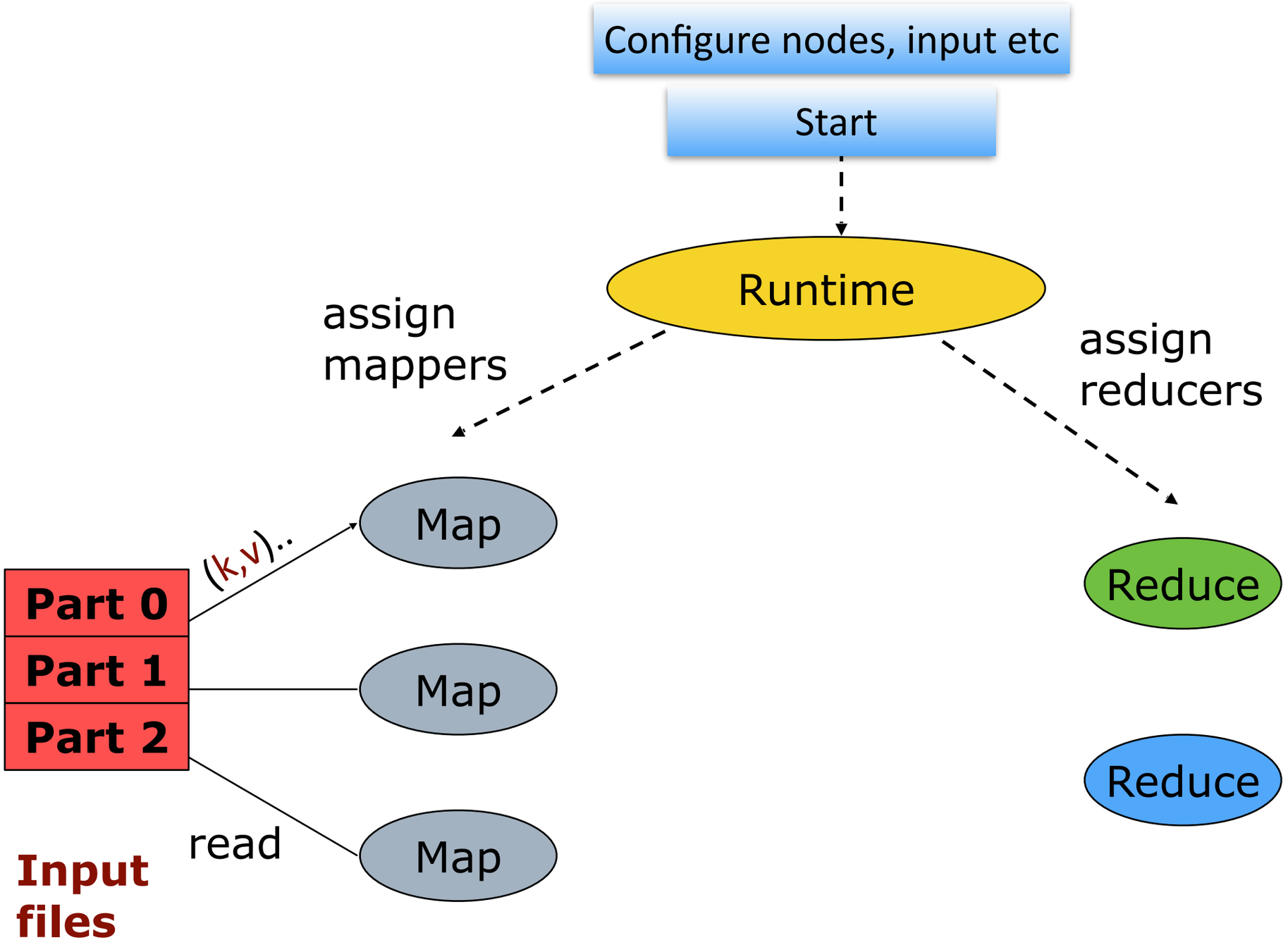


Input files

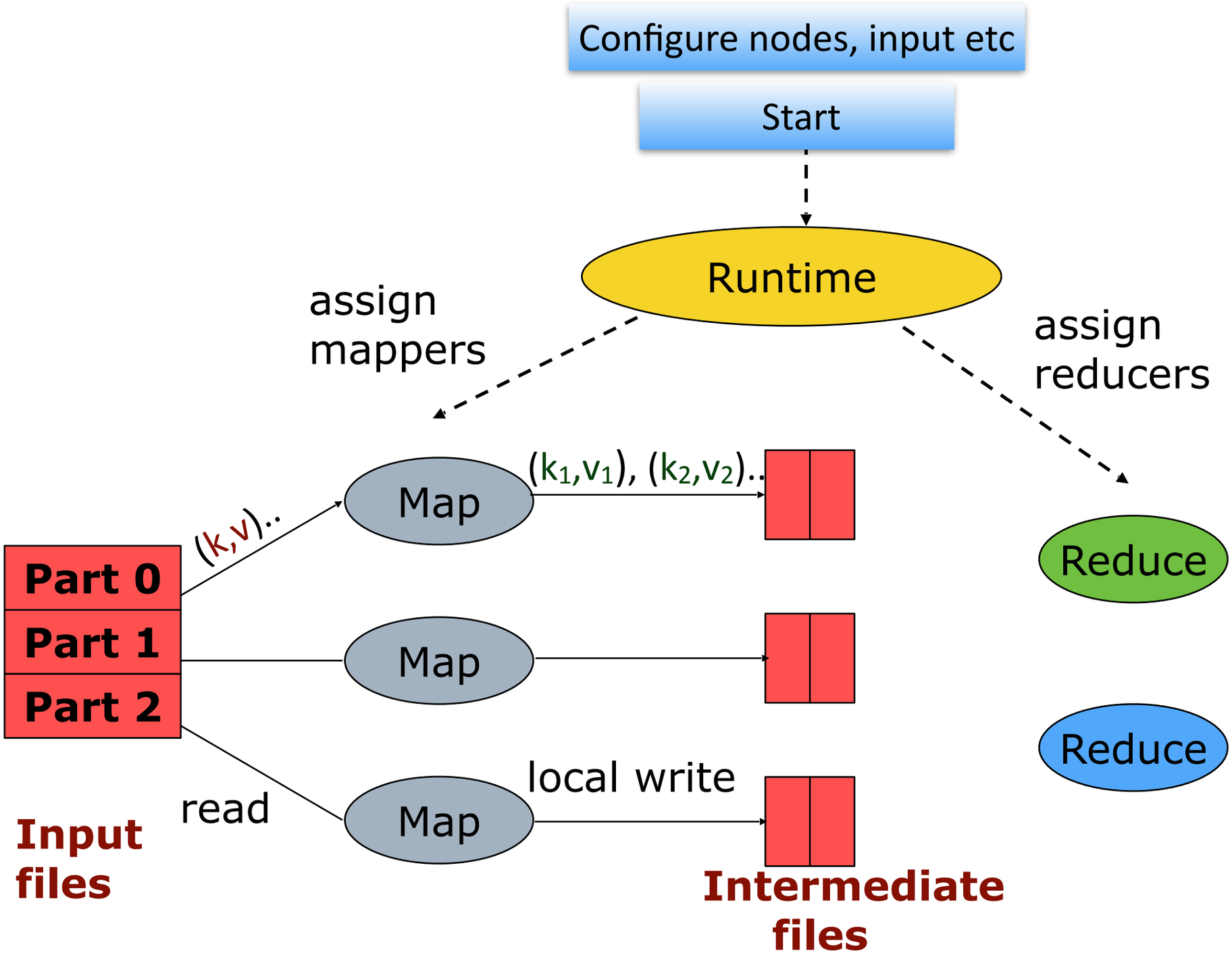
Pipeline


Configure nodes, input etc
Start
Runtime
assign mappers
assign reducers
Map
Map
Map
Reduce
Reduce
Part 0
Part 1
Part 2
Input files

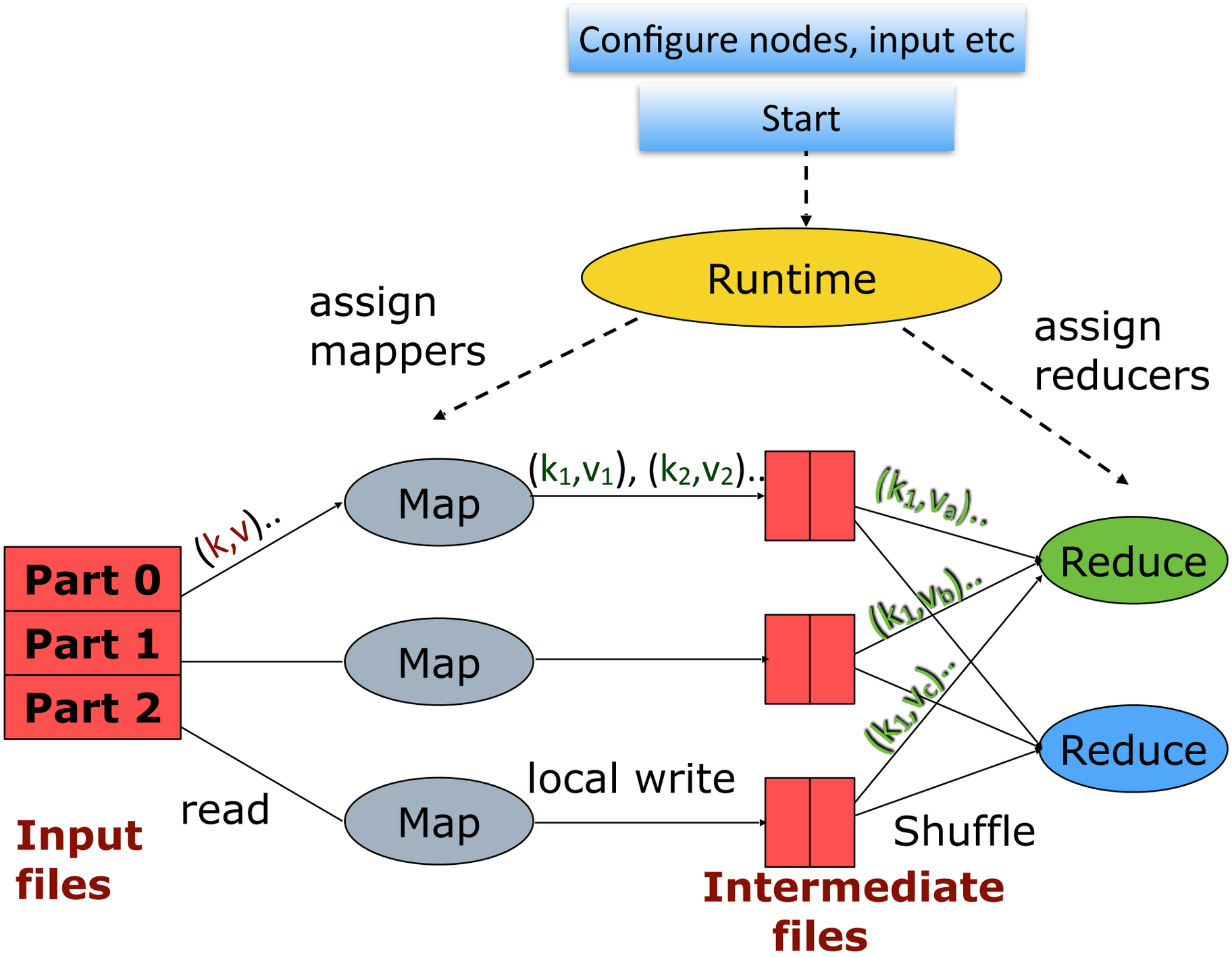
Pipeline


Configure nodes, input etc
Start
Runtime
assign
mappers
assign
reducers
Map
Map
Map
(k,v)..
Part 0
Part 1
Part 2
read
Input
files
Reduce
Reduce

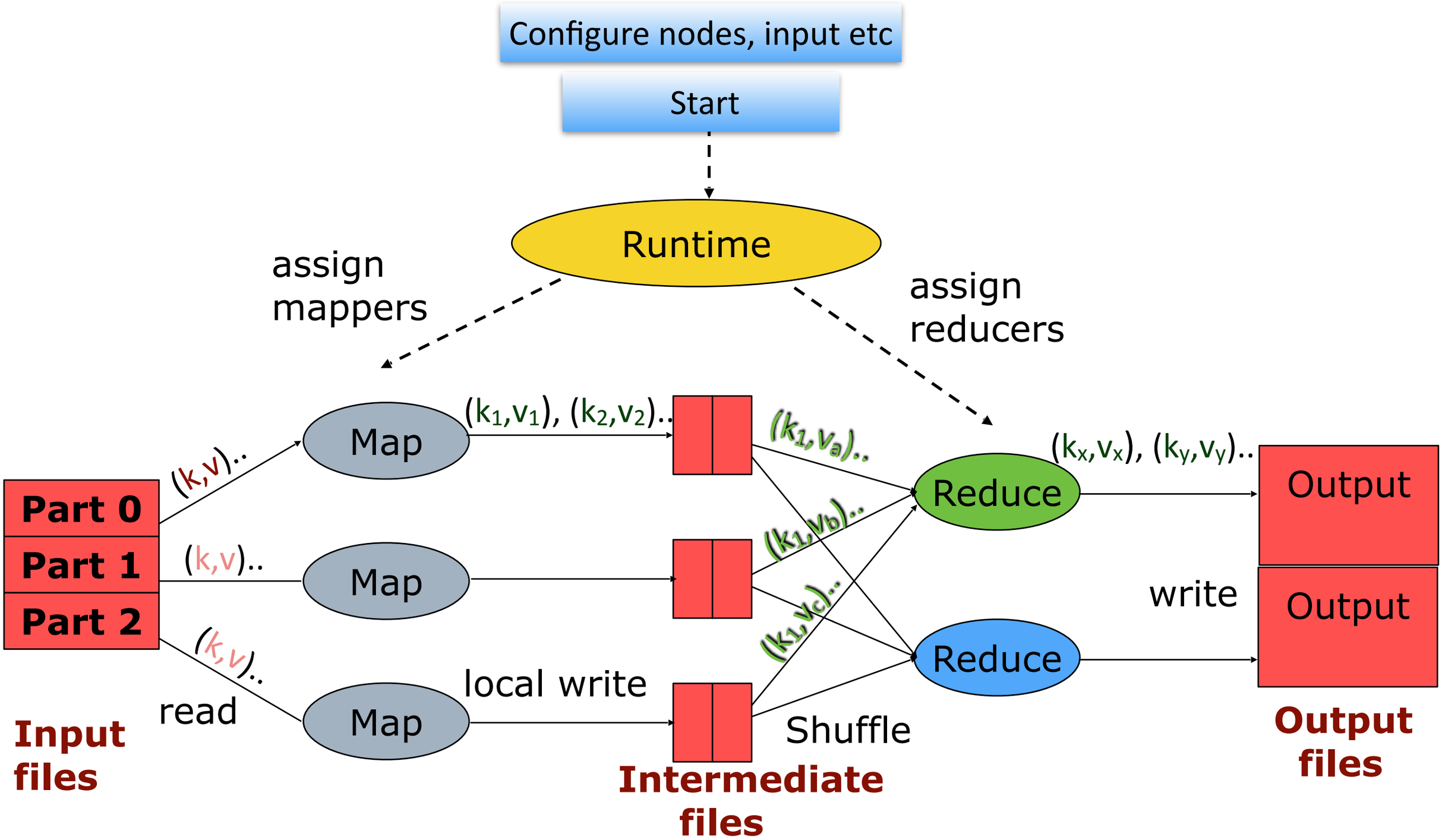
Pipeline


Configure nodes, input etc
Start
Runtime
assign mappers
assign reducers
Map
Map
Map
(k1,v1), (k2,v2)..
(k,v)..
Part 0
Part 1
Part 2
read
local write
Reduce
Reduce
Input files
Intermediate files

Pipeline


Configure nodes, input etc
Start
Runtime
assign mappers
assign reducers
Map
Map
Map
(k,v)..
(k1,v1), (k2,v2)..
(k1,va)..
(k1,vb)..
(k1,vc)..
Reduce
Reduce
Part 0
Part 1
Part 2
read
local write
Shuffle
Input files
Intermediate files

Pipeline


Configure nodes, input etc
Start
Runtime
assign mappers
assign reducers
Map
Map
Map
(k1,v1), (k2,v2)..
(k1,va)..
(k1,vb)..
(k1,vc)..
Reduce
Reduce
(kx,vx), (ky,vy)..
Output
Output
write
Part 0
Part 1
Part 2
(k,v)..
(k,v)..
(k,v)..
read
local write
Shuffle
Input files
Intermediate files
Output files

Map-Reduce Example

Map-Reduce Example

```
public static class Map extends Mapper<LongWritable,Text,Text,IntWritable> {
    public void map(LongWritable key, Text value, Context context) throws
                                                    IOException,InterruptedException{
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line); // Get all words
        while (tokenizer.hasMoreTokens()) { // For each word:
             value.set(tokenizer.nextToken());
            context.write(value, new IntWritable(1)); // produce <value 1>
        }
    }
}
```

Map-Reduce Example

```
public static class Map extends Mapper<LongWritable,Text,Text,IntWritable> {
    public void map(LongWritable key, Text value, Context context) throws
                                                    IOException,InterruptedException{
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line); // Get all words
        while (tokenizer.hasMoreTokens()) { // For each word:
             value.set(tokenizer.nextToken());
            context.write(value, new IntWritable(1)); // produce <value 1>
        }
    }
}
```

```
public static class Reduce extends Reducer<Text,IntWritable,Text,IntWritable> {
    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws
                                                    IOException,InterruptedException {
        int sum=0;
        for(IntWritable x: values)
            sum += x.get();
        context.write(key, new IntWritable(sum)); // Produce one <key, count>
    }
}
```