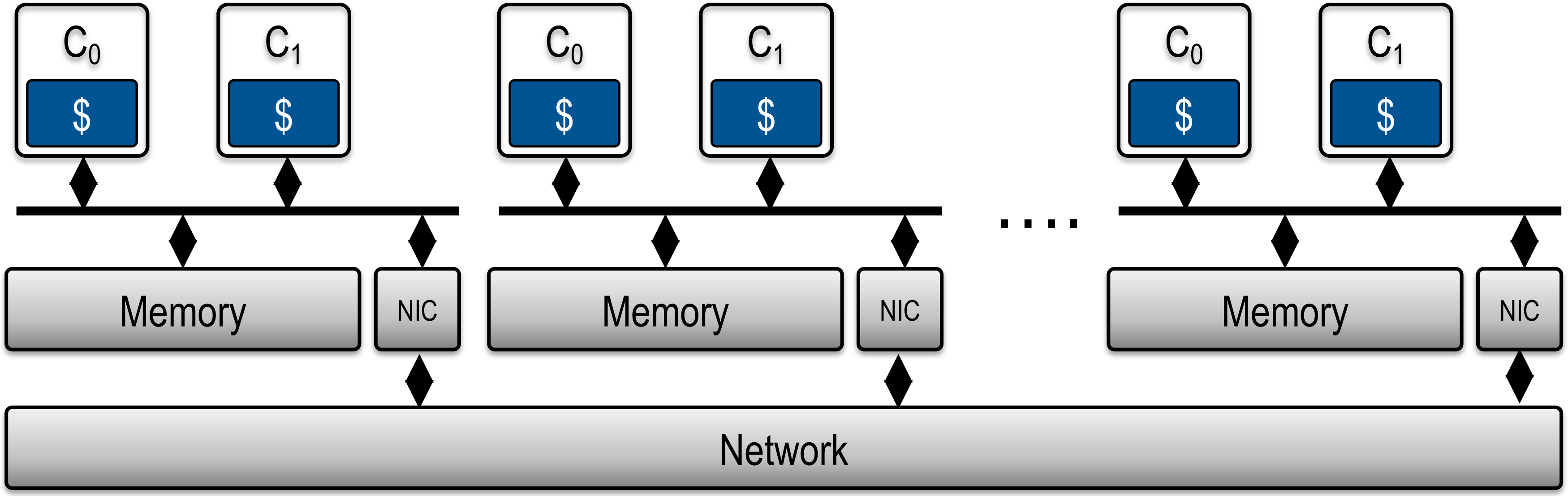
CS302

Message Passing



Spring 2025
Arkaprava Basu & Babak Falsafi
parsa.epfl.ch/course-info/cs302

Where are We?

M	T	W	T	F
17-Feb	18-Feb	19-Feb	20-Feb	21-Feb
24-Feb	25-Feb	26-Feb	27-Feb	28-Feb
3-Mar	4-Mar	5-Mar	6-Mar	7-Mar
	11-Mar	12-Mar	13-Mar	14-Mar
17-Mar	18-Mar	19-Mar	20-Mar	21-Mar
24-Mar	25-Mar	26-Mar	27-Mar	28-Mar
31-Mar	1-Apr	2-Apr	3-Apr	4-Apr
7-Apr	8-Apr	9-Apr	10-Apr	11-Apr
14-Apr	15-Apr	16-Apr	17-Apr	18-Apr
21-Apr	22-Apr	23-Apr	24-Apr	25-Apr
28-Apr	29-Apr	30-Apr	1-May	2-May
5-May	6-May	7-May	8-May	9-May
12-May	13-May	14-May	15-May	16-May
19-May	20-May	21-May	22-May	23-May
26-May	27-May	28-May	29-May	30-May

- Message Passing
  - Message passing model
  - MPI overview
- Exercise session
  - Example MPI programs
- Next Tuesday:
  - Memory Consistency

Overview of Parallel Programming Models

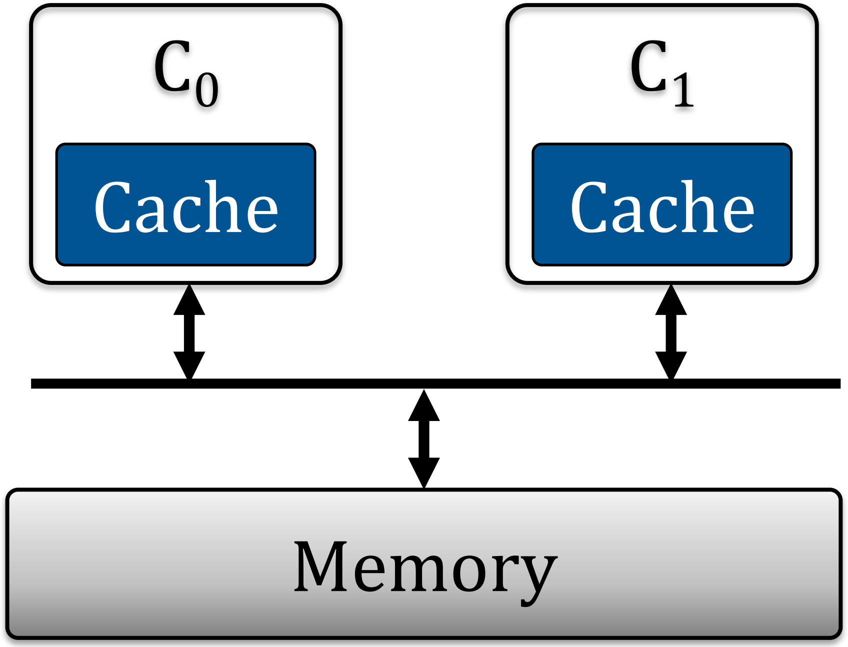
- Two classes of parallel programming:
  - Shared memory
  - Message passing

- So far, we have only looked at the shared memory model
  - OpenMP usage and optimizations

- This lecture will focus on the message passing model
  - Advantages and disadvantages compared to shared memory
  - MPI programming in C
  - Brief intro to hybrid programming models

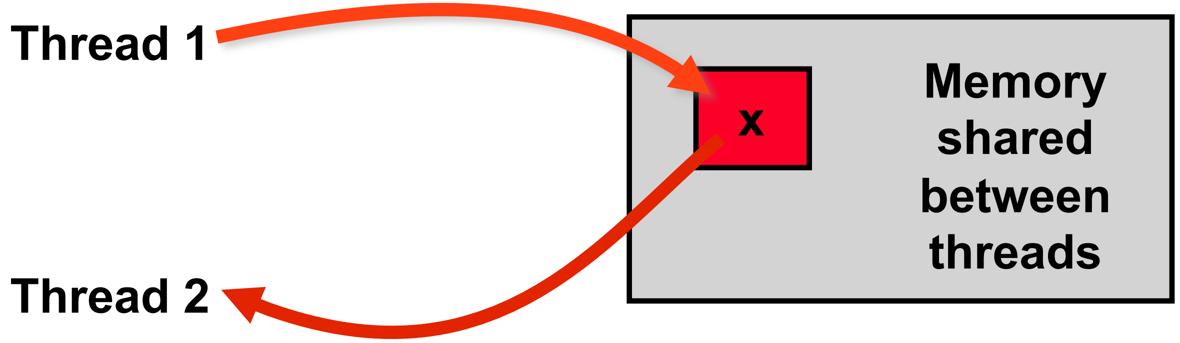
Memory: A Hardware Perspective

- Conventional CPUs use shared physical memory
- Cores use physical addresses to reference memory
- Unified physical address space for all cores
- Coherence and consistency rules apply



Memory: A Programmer's Perspective

- Threads communicate by reading/writing to shared vars
- Shared variables are like a big bulletin board
- Any thread can read or write



Thread 1:

```
int x = 0;
x = 1;
```

Thread 2:

```
int x;
while (x == 0) {}

print x;
```

Memory: A Systems Perspective

Option 1: threads share an address space

- All data is sharable

Option 2: each thread has its own virtual address space

- Shared part maps to the same physical location

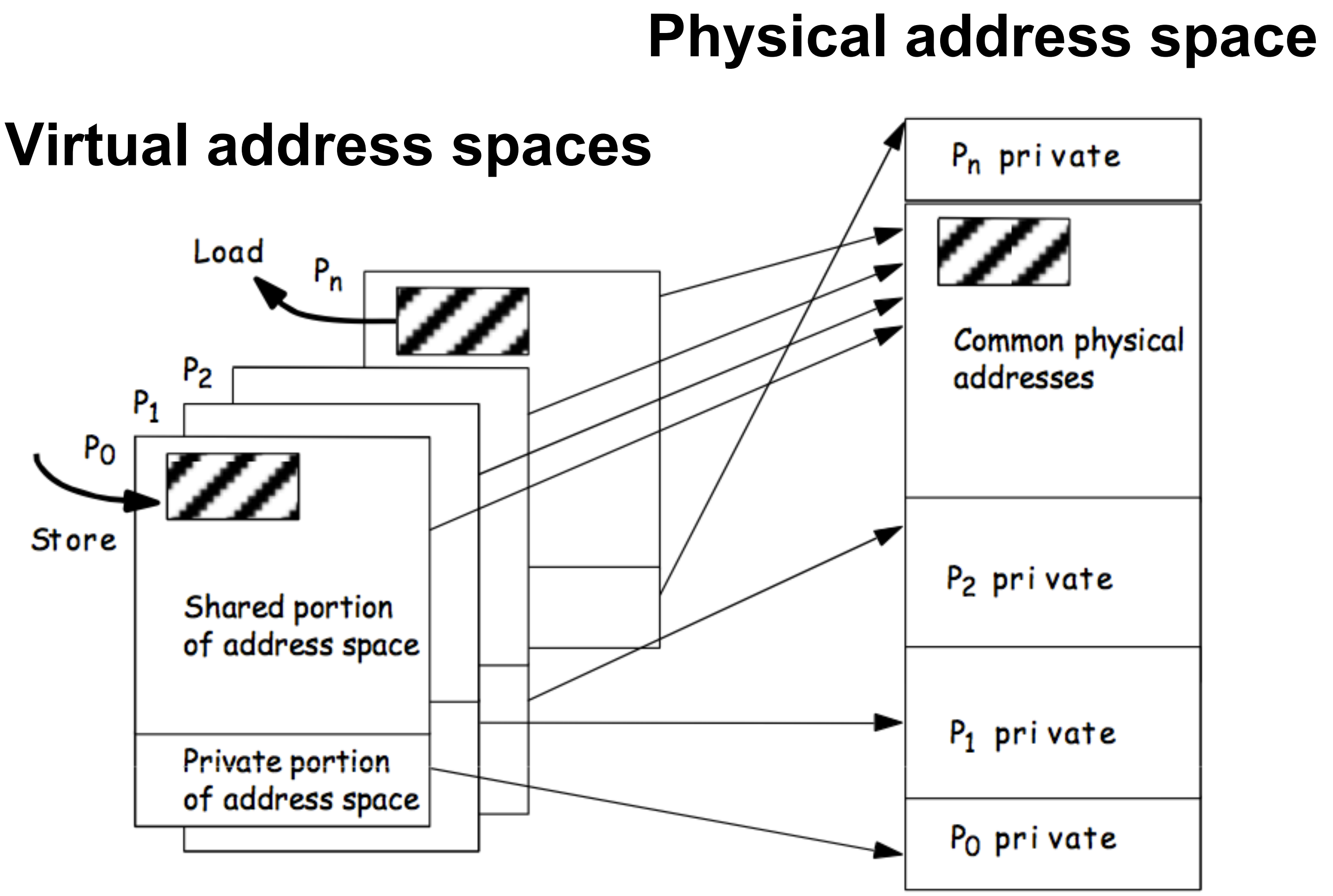
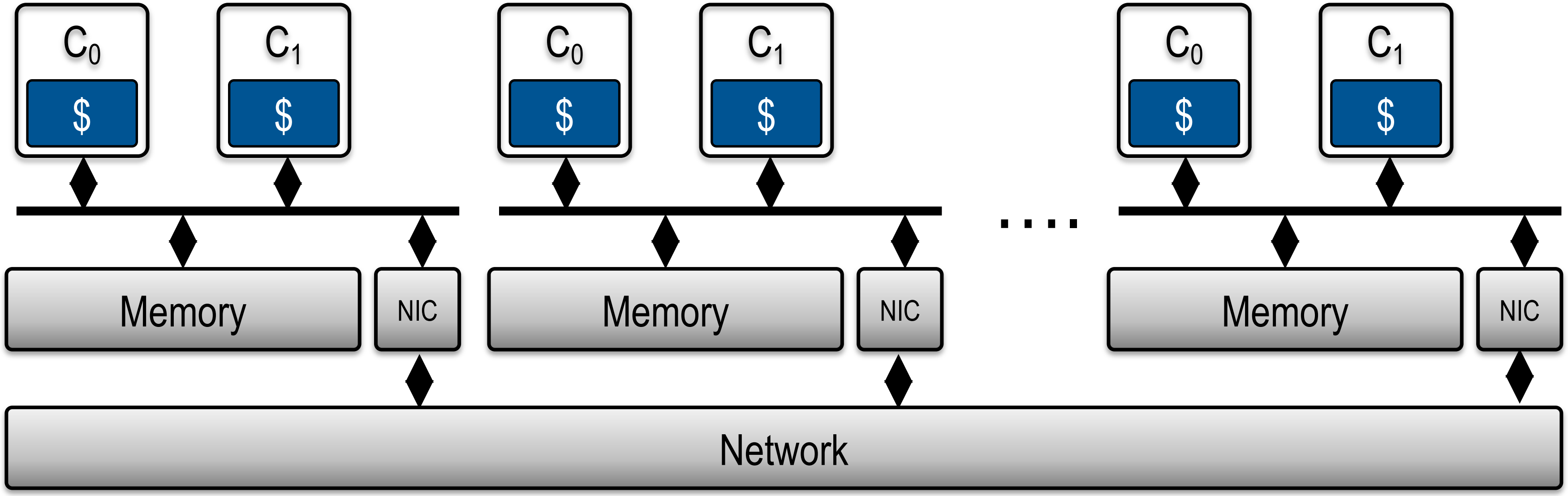


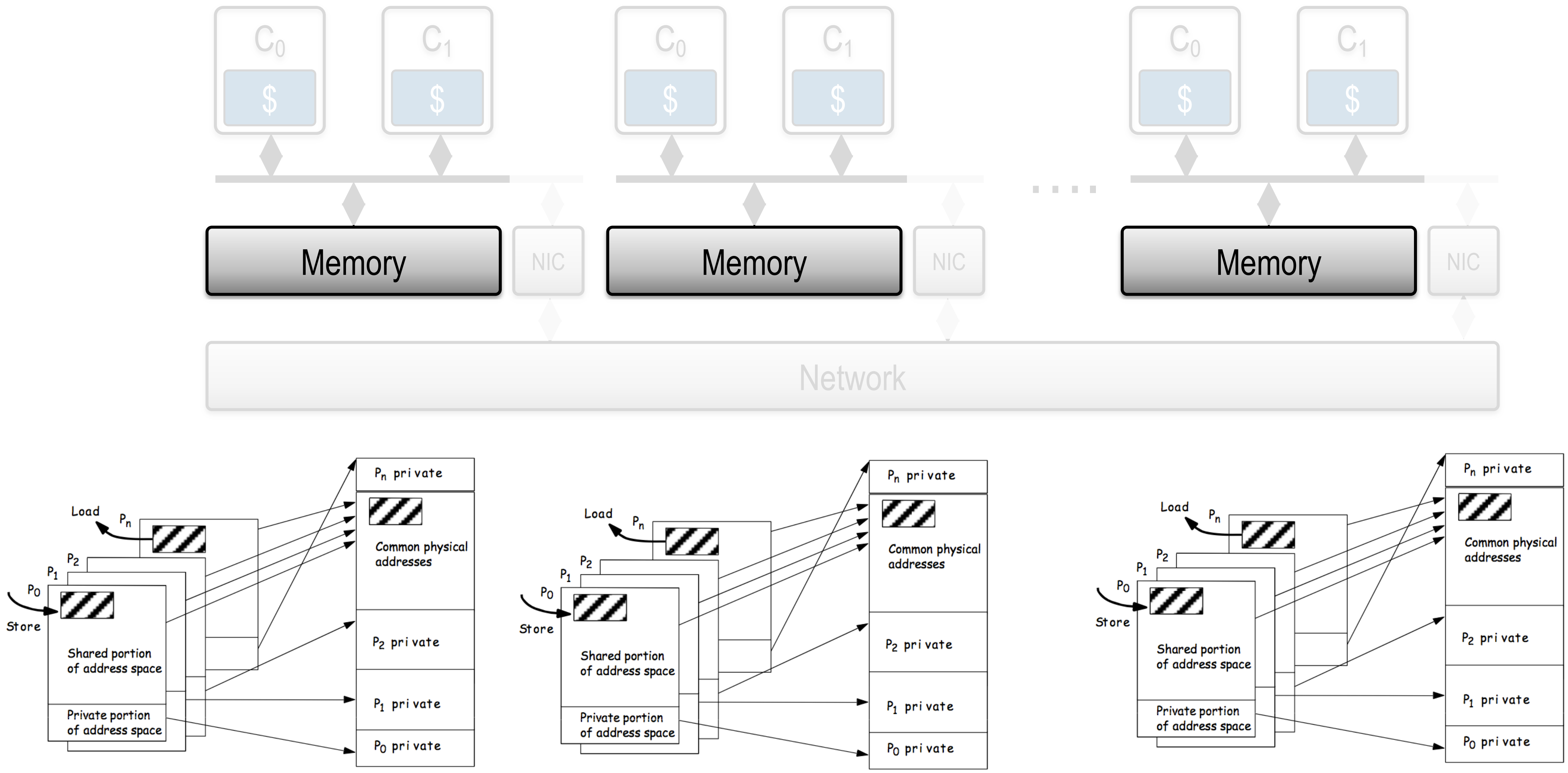
Image credit: Culler, Singh, and Gupta

Beyond a Few CPUs Memory is Distributed

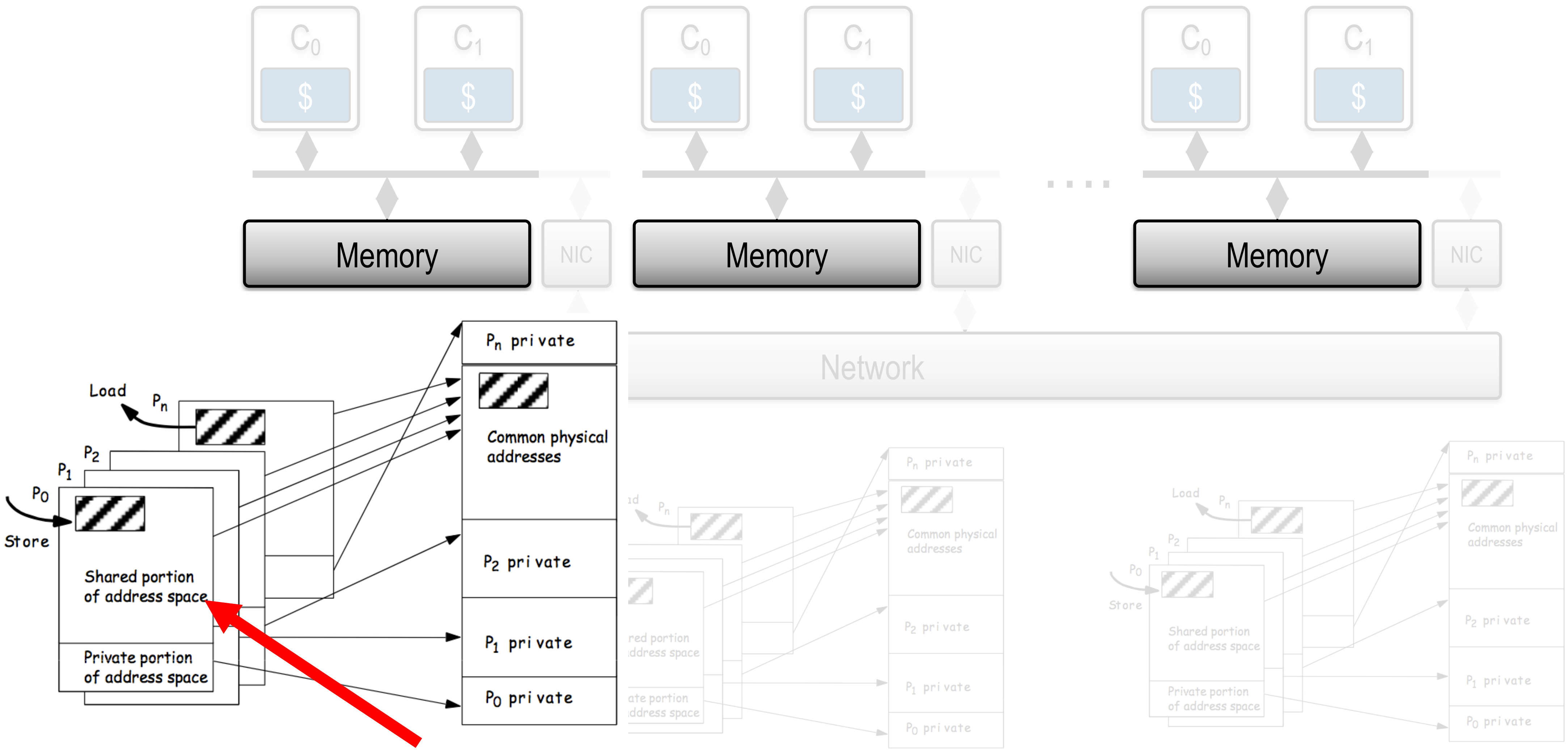


- Each node has its own memory
- A Network Interface Card (NIC) connects nodes to a network
- Nodes communicate through message passing

Distributed Memory: Multiple Physical Address Spaces

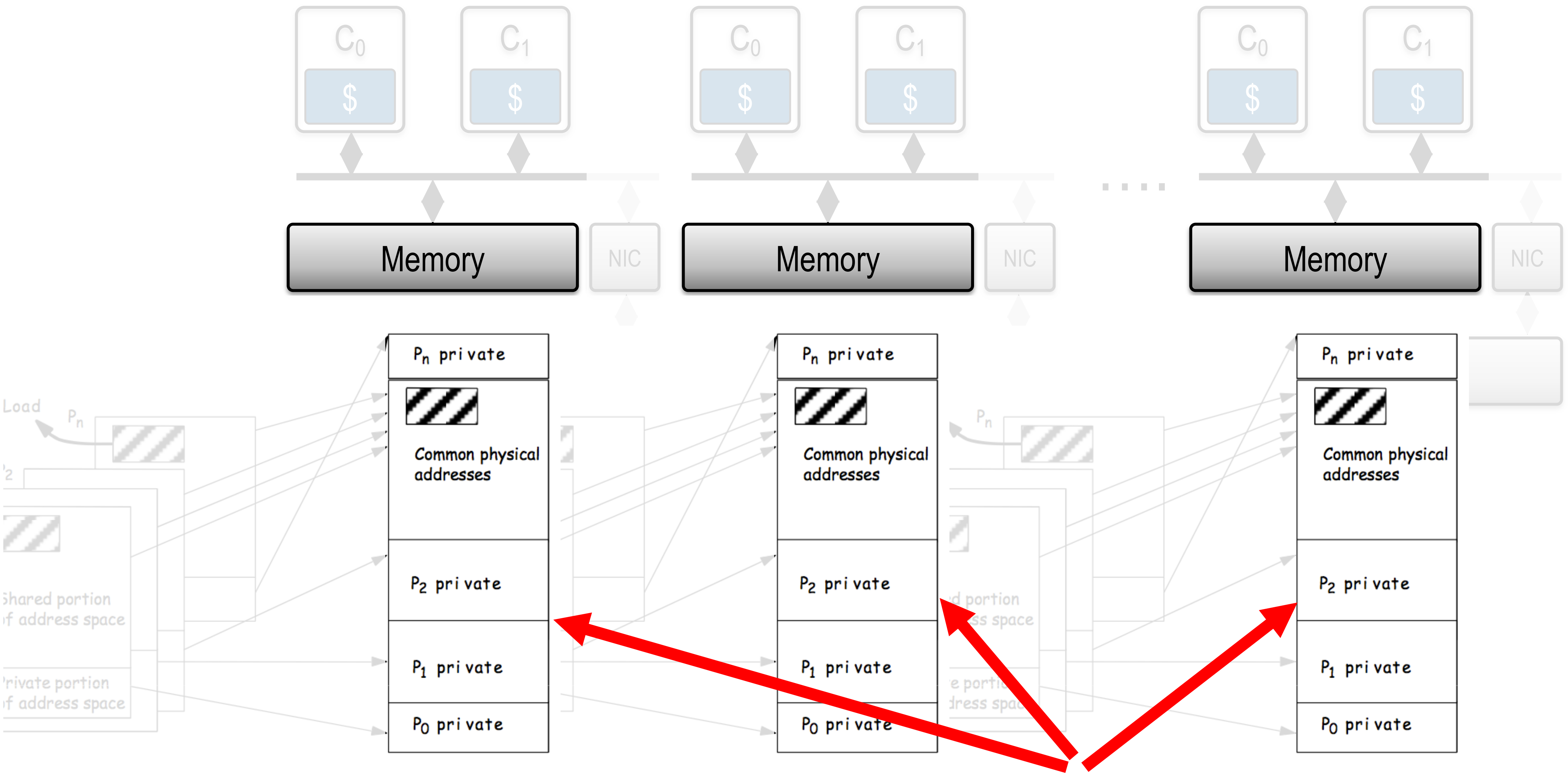


Distributed Memory: Multiple Physical Address Spaces



Multiple processes share memory on a single node

Distributed Memory: Multiple Physical Address Spaces



Multiple nodes have disjoint physical memory

Message Passing Advantages

- Cost scalability
  - Cache-coherent (distributed) shared memory is possible but expensive
  - E.g., HPE Superdome
  - Distributed memory w/o cache coherence is common
  - Requires message passing for software
- Performance transparency
  - Communication is explicit (send/receive messages)
- Fault tolerance
  - A single process failure does not crash the entire system

Message Passing Disadvantages

- Higher communication overhead
  - User-level library, syscalls, network communication software for send/receive
- Increased code complexity
  - Programmer must explicitly manage data movement
  - Bloated in terms of LOC
- Difficult to overlap communication with computation

Supercomputers use Message Passing

- World’s second fastest supercomputer (present in the US)
- Uses a hybrid programming model based on message passing (MPI)

What is MPI?

- MPI: Message Passing Interface
- A standardized API for parallel programming using message passing

- The MPI effort involved about 80 people from 40 organizations
- Incorporated the most useful features of several systems
- MPI is the de-facto standard for all high-performance systems

- Code is highly portable
- Write code once, run on any MPI compatible system

What is MPI?

- MPI is designed for running multiple instances of the same program
- Best for Single Program Multiple Data (SPMD) execution model

- MPI is NOT designed to be used for:
  - Communication between different programs
  - Client-server communications
  - Concurrent execution

- Coroutines and RPCs used for these purposes
  - To be covered in later lectures

MPI Programming Overview

- Creating parallelism
  - SPMD Model
- Communication between processes
  - Basic
  - Non-blocking
  - Collective
- Synchronization
  - Point-to-point synchronization by message passing
  - Global synchronization by collective communication

SPMD Model

- Single Program Multiple Data model of programming:
  - Each process has a copy of the same program
  - All run them at their own rate
  - May take different paths through the code
- Process-specific control through variables like:
  - Unique process number
  - Total number of processes
- Processes may synchronize, but none is implicit

MPI Libraries

- MPI is programming language and implementation independent
    - Just a standard API specification

- Vendors have various MPI libraries
    - MPICH/Open MPI are popular open source and free implementations
    - Vendors add features and optimizations for their systems
        - Intel MPI, Microsoft MPI, etc.
    - Bindings exist for high level languages such as Java, Python, etc

- In this course, we focus on MPI programming using C and C++
- Programming using MPI is independent of the library used

MPI Hello World (Trivial)

- A simple, but not very interesting, SPMD program

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv);
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```

MPI Hello World (Trivial)

- A simple, but not very interesting, SPMD program

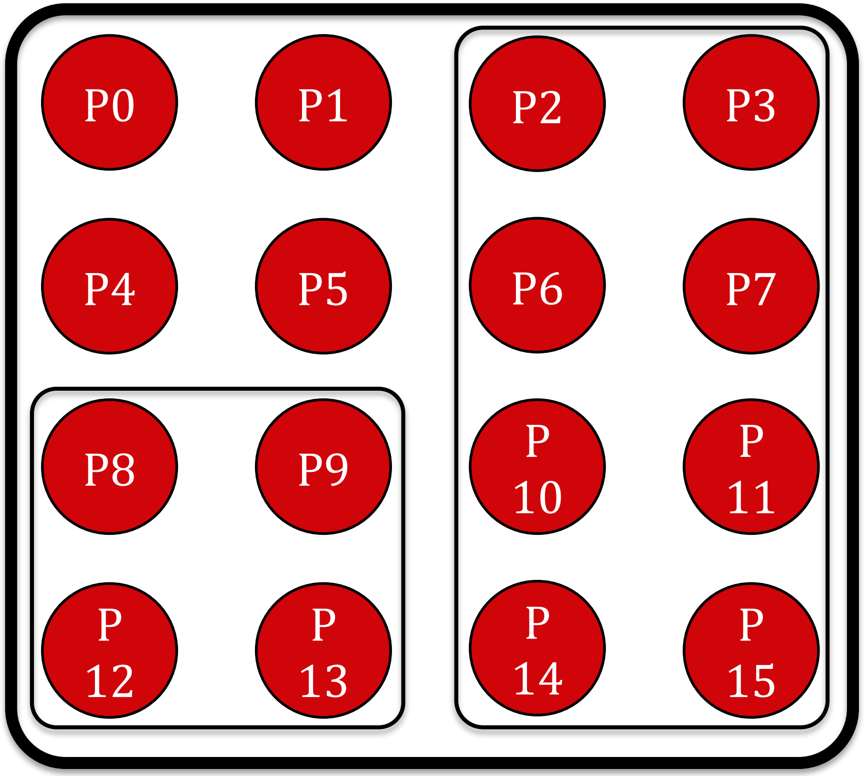
```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv);
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```

- `#include "mpi.h"` → Include the MPI Libraries
- `MPI_Init( &argc, &argv);` → Initializes the MPI environment
- `MPI_Finalize();` → End of MPI section (no MPI calls allowed after this line)

- Each process simply prints “Hello, world!” and exits

MPI Communicators

- MPI processes can be collected into groups
- MPI identifies groups by their context
- A communicator is a combination of a group and its context



MPI Communicators

- On starting an MPI program there is one predefined communicator
  - **`MPI_COMM_WORLD`**
  - Contains the group of all processes

- A process is identified by a unique number within each communicator
  - This is the rank of a process
  - The same process can have different ranks in different communicators

- Communicators are used to minimize unnecessary communications
- However, simple programs generally only use **`MPI_COMM_WORLD`**
- More complex use-cases to be discussed later

MPI Hello World (Process specific)

- Individual processes can be made to process-specific tasks

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf("I am process %d of %d.\n", rank, size);
    MPI_Finalize();
    return 0;
}
```

MPI Hello World (Process specific)

- Individual processes can be made to process-specific tasks

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf("I am process %d of %d.\n", rank, size);
    MPI_Finalize();
    return 0;
}
```

- `int rank, size;` → Variables to identify a process
- `MPI_Comm_rank( MPI_COMM_WORLD, &rank );` → Get the rank of the process
- `MPI_Comm_size( MPI_COMM_WORLD, &size );` → Get the # of processes

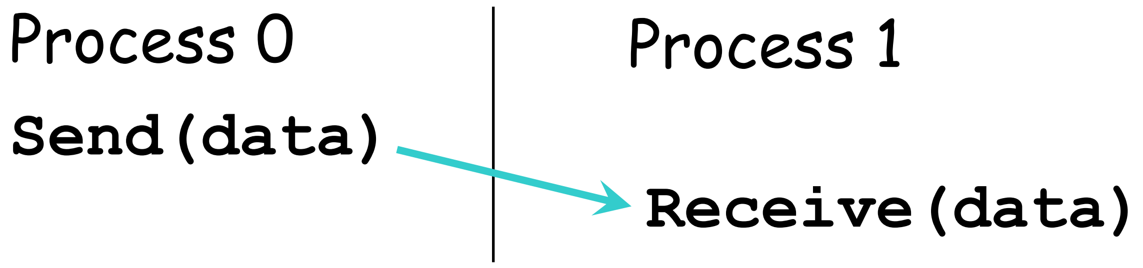
- Each process can now identify itself and print a custom hello world statement

Components of a MPI program

- Sender and receiver processes
- Messages

MPI Basic Send/Receive

- "Two sided" – both sender and receiver must take action



- Things that need specifying:
  - How will processes be identified? (through rank inside the communicator)
  - How will "data" be described?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?

MPI Send

- Function signature to send data:

```
MPI_Send(
    void* message,
    int count,
    MPI_Datatype datatype,
    int dest,
    int tag,
    MPI_Comm communicator);
```

MPI Send

- Function signature to send data:

```
MPI_Send(
    void* message,                    → The buffer of data to send
    int count,                        → Number of elements in the buffer
    MPI_Datatype datatype,            → Datatype of the elements in the buffer
    int dest,                         → Rank of the receiver
    int tag,                          → An identifying number like the subject of an email
    MPI_Comm communicator);           → Communicator of the receiver
```

Things to note

- **`MPI_Send`** is a blocking function call
- When **`MPI_Send`** returns, message has been delivered to the system
  - The buffer can be reused
  - Does not imply that the message has been received by the target process

- Three other variants of **`MPI_Send`**:
  - **`MPI_Bsend`**: Returns when the message is buffered in an application buffer
  - **`MPI_Ssend`**: Returns only when receiver has started to receive the message
  - **`MPI_Rsend`**: Assuming receiver is ready, message is sent as soon as possible

MPI Receive

- Function signature to receive data:

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

MPI Receive

- Function signature to receive data:

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

- `void* data` → Buffer to store the received data in
- `int count` → Receive at most this many elements
- `MPI_Datatype datatype` → Datatype of the received elements
- `int source` → Rank of the sender from whom it expects data
- `int tag` → The expected tag of the message
- `MPI_Comm communicator` → Communicator of the sender
- `MPI_Status* status` → Metadata assosciated with the message

Things to note

- **`MPI_Recv`** is also a blocking function call
- **`source`** can be **`MPI_ANY_SOURCE`** to accept data from any sender
- **`tag`** can be **`MPI_ANY_TAG`** to accept messages with any tag
- **`status`** contains further information:
  - Who sent the message (can be used with **`MPI_ANY_SOURCE`**)
  - How much data was actually received (using **`MPI_GET_COUNT`**)
  - Tag of the message (can be used with **`MPI_ANY_TAG`**)
  - **`MPI_STATUS_IGNORE`** can be used to ignore these additional information

Using Message Status

- Status is a data structure allocated in the user's program.
- Especially useful with wild-cards to find out what matched:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

MPI Datatypes

- An MPI datatype is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., **`MPI_INT`**)
  - a contiguous array of MPI datatypes
  - a strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes

- There are MPI functions to construct custom datatypes, such an array of (int, float) pairs, or a row of a matrix stored columnwise.

Point-to-Point Communication Example

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] )
{
   int rank, size;
   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   MPI_Comm_size( MPI_COMM_WORLD, &size );



   MPI_Finalize();
   return 0;
}
```

Standard MPI Boilerplate

Point-to-Point Communication Example (1)

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] )
{
   ...

   int data, tag = 1;
   if(rank == 0){
       data = 42;
       MPI_Send(&data, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
       printf("Message Sent: %d\n", data);
   } else if(rank == 1){
       MPI_Recv(&data, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
       printf("Message Received: %d\n", data);
   }

   ...
}
```

Point-to-Point Communication Example (2)

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] )
{
   ...

   int data, tag = 1;
   if(rank == 0){
       data = 42;
       MPI_Send(&data, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
       printf("Message Sent: %d\n", data);
   } else if(rank == 1){
       MPI_Recv(&data, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                       MPI_COMM_WORLD, MPI_STATUS_IGNORE);
       printf("Message Received: %d\n", data);
   }
   ...
}
```

Compiling and Running MPI Programs

- Compilation:
  - Regular applications: `gcc prog.c -o prog`
  - MPI applications: `mpicc prog.c -o prog`

- Execution:
  - Regular applications: `./prog`
  - MPI applications: `mpiexec -np <nprocs> ./prog`

- `mpiexec` acts as runtime system coordinating amongst all processes
- Exact commands will differ depending on the compiler used
- Examples on SCITAS during exercise session

Communicating Arrays

- Approach 1:

```
int A[1000];
for(int i = 0; i < 1000; i++) {
        MPI_Send(&A[i], 1, MPI_INT, rank, tag, MPI_COMM_WORLD);
}
```

- Approach 2:

```
int A[1000];
MPI_Send(A, 1000, MPI_INT, rank, tag, MPI_COMM_WORLD);
```

- Which approach is better?

Communicating Arrays

- Approach 1: Sending elements one by one
  - Each message has to be acknowledged by the receiver
  - Overheads associated with each function call

- Approach 2: Sending entire array at once
  - Overheads amortized over a single large function call
  - However, large amount of data needs to be buffered and transmitted

- A hybrid approach is often best (e.g, chunks of 100 elements)
- Optimization knob in programs

Latency and Bandwidth

- For short messages, latency dominates transfer time
- For long messages, the bandwidth term dominates transfer time
- Latency * bandwidth is the amount of buffer that exists in the system → also referred to as **critical message size**
- Example: 50 us * 50 MB/s = 2500 bytes
  - messages > 2500 bytes are bandwidth dominated
  - messages < 2500 bytes are latency dominated

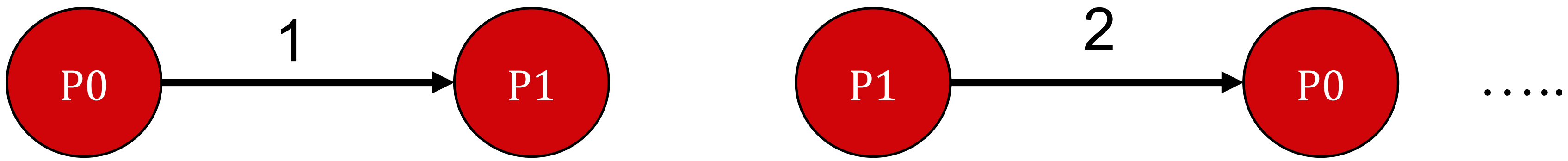
Example: ping-pong (mpitutorial.com) 2 processes

```
int ping_pong_count = 0;
int partner_rank = (rank + 1) % 2;
while (ping_pong_count < PING_PONG_LIMIT) {
    if (rank == ping_pong_count % 2) {
        // Increment the ping pong count before you send it
        ping_pong_count++;
        MPI_Send(&ping_pong_count, 1, MPI_INT, partner_rank, 0, MPI_COMM_WORLD);
        printf("%d sent and incremented ping_pong_count %d to %d\n", rank,
               ping_pong_count, partner_rank);
    } else {
        MPI_Recv(&ping_pong_count, 1, MPI_INT, partner_rank, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        printf("%d received ping_pong_count %d from %d\n", rank, ping_pong_count,
                 partner_rank);
    }
}
```

Example: ping-pong (mpitutorial.com) 2 processes

first iteration

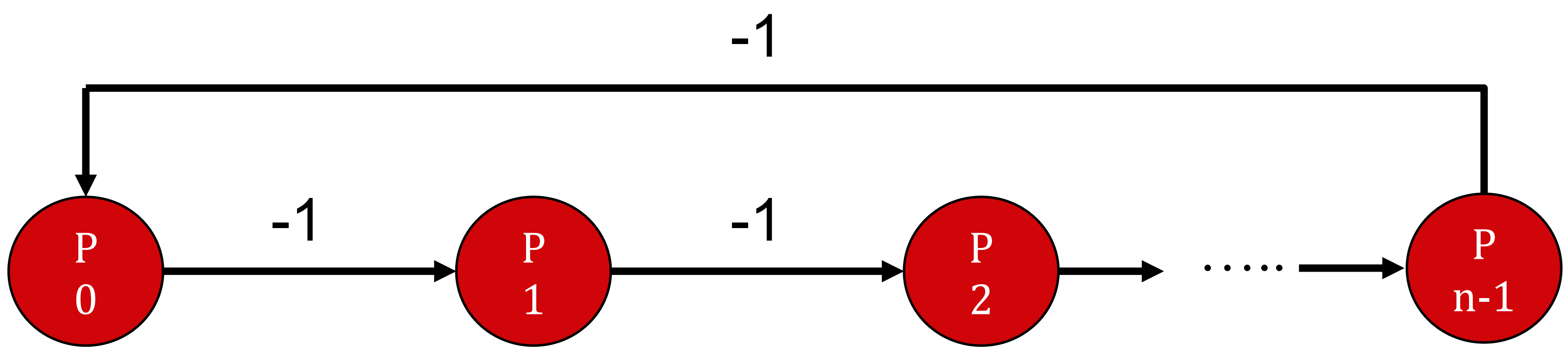
second iteration



Example: ring (mpitutorial.com) n processes

```
int token;
if (rank != 0) {
    MPI_Recv(&token, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process %d received token %d from process %d\n", rank, token, rank - 1);
} else {
    token = -1; // Set the token's value if you are process 0
}

MPI_Send(&token, 1, MPI_INT, (rank + 1) % size, 0, MPI_COMM_WORLD);

// Now process 0 can receive from the last process.
if (rank == 0) {
    MPI_Recv(&token, 1, MPI_INT, size - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process %d received token %d from process %d\n", rank, token, size - 1);
}
```

Example: ring (mpitutorial.com) n processes



MPI Functions So Far

- `MPI_Init()`
- `MPI_Comm_rank()`
- `MPI_Comm_size()`
- `MPI_Send()`
- `MPI_Recv()`
- `MPI_Get_count()`
- `MPI_Finalize()`

- These functions are enough to write most MPI programs
- But need to consider other features for extracting performance

Revisiting MPI_Send and MPI_Recv

- **`MPI_Send(void* message, ...)`**
- **`MPI_Recv(void* data, ...)`**

- **`MPI_Send`** is a blocking function call
  - Does not return until the data in the **`message`** buffer has been copied
  - Return of the call implies completion of the send procedure
  - The memory location referenced by **`message`** cannot be reused until return

- **`MPI_Recv`** is also a blocking function call
  - Execution cannot continue until **`data`** is completely received

Problems with Blocking Function Calls

- Problem 1: Deadlocks can happen if program not written correctly
  - This is a correctness problem, not a performance problem
- For example, consider two processes communicating as follows:

```
...
if(rank == 0){
   MPI_Send(...);
   MPI_Recv(...);
} else if(rank == 1){
   MPI_Send(...);
   MPI_Recv(...);
}
...
```

Deadlocks due to Blocking Function Calls

- Consider two processes communicating as follows:

```
...
if(rank == 0){
    MPI_Send(...);
    MPI_Recv(...);
} else if(rank == 1){
    MPI_Send(...);
    MPI_Recv(...);
}
...
```

Process 0 stuck sending data

Process 1 also stuck sending data

- Both processes stuck sending data to each other
- Neither process can invoke **`MPI_Recv`** to receive the data

Simple Fix

- If order is known, simply order MPI_Send and MPI_Recv correctly

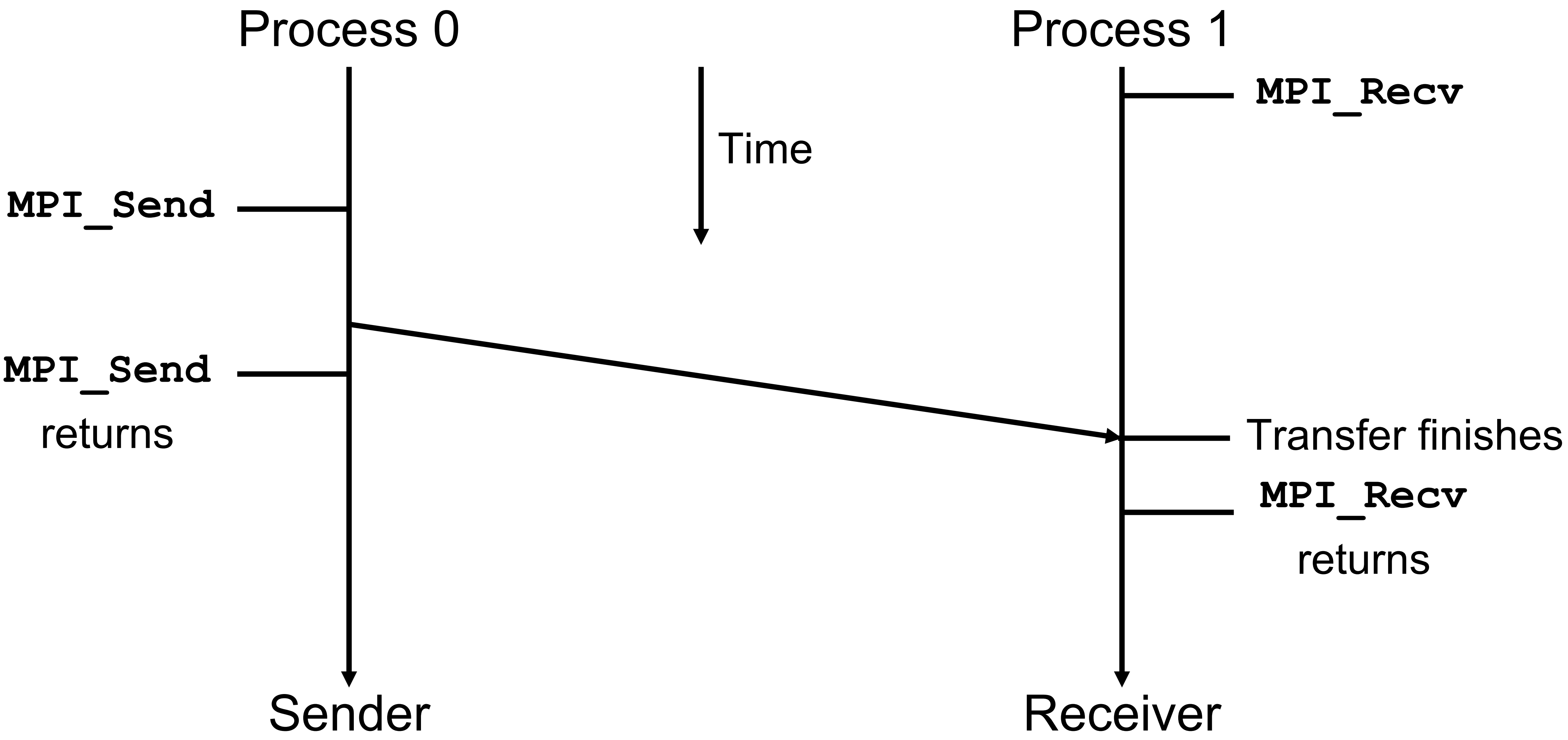
```
...
if(rank == 0){
    MPI_Send(...);
    MPI_Recv(...);
} else if(rank == 1){
    MPI_Recv(...);
    MPI_Send(...);
}
...
```

- Not applicable to scenarios with variable communication patterns
- Requires programmer effort to ensure deadlocks do not arise

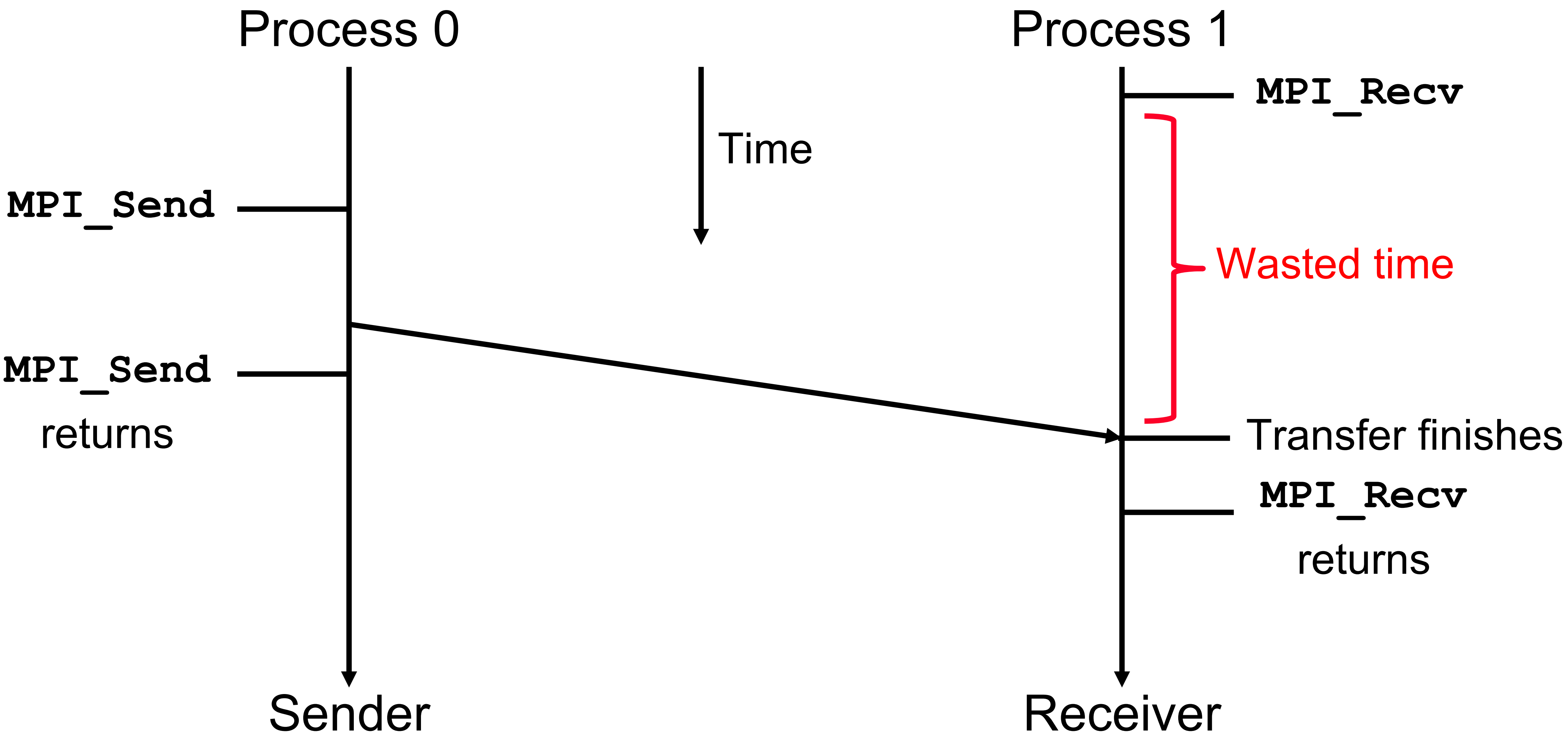
Problems with Blocking Function Calls

- Problem 2: Performance penalty
  - Note that this is not a correctness problem
- On calling **`MPI_Send/MPI_Recv`**, a process cannot do anything else
- Need to wait before transfer finishes before doing useful work
- Big performance penalty for programs with sparse communication

Performance Problem due to Blocking



Performance Problem due to Blocking



Non-Blocking Calls

- Non-blocking function calls return immediately after invocation
  - Does not provide any guarantee on the status of the message
  - Programmer needs to manually check for completion of mesage transfer

- Requires more programmer effort BUT
  - Allows to overlap communication with computation
  - Increase the efficiency and performance of the entire program

- This mode of communication is called asynchronous communication
- Critical to improve the performance of any message passing system

Non-Blocking Send

```
MPI_Send(
        void* message,
        int count,
        MPI_Datatype datatype,
        int dest,
        int tag,
        MPI_Comm communicator)
```

```
MPI_Isend(
        void* message,
        int count,
        MPI_Datatype datatype,
        int dest,
        int tag,
        MPI_Comm communicator,
        MPI_Request* req)
```

Non-Blocking Send

```
MPI_Send(
      void* message,
      int count,
      MPI_Datatype datatype,
      int dest,
      int tag,
      MPI_Comm communicator)
```

```
MPI_Isend(
      void* message,
      int count,
      MPI_Datatype datatype,
      int dest,
      int tag,
      MPI_Comm communicator,
      MPI_Request* req)
```

The current sending status is available through the request variable

Non-Blocking Receive

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

```
MPI_Irecv(
        void* data,
        int count,
        MPI_Datatype datatype,
        int source,
        int tag,
        MPI_Comm communicator,
        MPI_Request* req)
```

Non-Blocking Receive

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

```
MPI_Irecv(
        void* data,
        int count,
        MPI_Datatype datatype,
        int source,
        int tag,
        MPI_Comm communicator,
        MPI_Request* req)
```

Along with status, contains information on completion

Status of Non-blocking Calls

- Non blocking calls return immediately
  - Do not wait for completion of data transfers
  - Allows processes to make forward progress

- While data transfer is going on,
  - Safe to compute on data not being used in the asynchronous communication
  - Unsafe to compute on data being communicated

- Only compute on communicated data once transfer is complete
- We need a way to know if the data transfer has completed

Example Benefit with Asynchronous Transfers

```
*buf1 = 3;
MPI_Send(buf1, 1, MPI_INT, …)
*buf2 = 4;

/* Process waits before
operating on different data */
```

```
*buf1 = 3;
MPI_Isend(buf1, 1, MPI_INT, …)
*buf2 = 4;

/* Process does not wait before
moving on to different data */
```

Example Problem with Asynchronous Transfers

```
*buf1 = 3;
MPI_Send(buf1, 1, MPI_INT, …)
*buf1 = 4;

/* This is ok, receiver will
always receive 3 */
```

```
*buf1 = 3;
MPI_Isend(buf1, 1, MPI_INT, …)
*buf1 = 4;

/* This is non-deterministic,
receiver can get either 3 or 4
*/
```

Testing the Status of Non-Blocking Calls

```
MPI_Test(
        MPI_Request* request,
        int* flag,
        MPI_Status* status)
```

- `request` → The request variable returned by a non-blocking call
- `flag` → Whether operation is complete or not
- `status` → The associated metadata of the operation, same as seen before

- Returns (immediately) the status of a non-blocking function call
- If **`flag == 0`**, then the operation is not yet complete
- If **`flag == 1`**, then the operation is complete
- **`status`** contains valid information only when **`flag == 1`**

Testing the Status of Non-Blocking Calls

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

- `MPI_Request* request` → The request variable returned by a non-blocking function
- `MPI_Status* status` → The assosciated metadata of the operation, same as seen before

- This function waits until the operation is completed
- Once complete, the function returns the **`status`** of the operation
- Note that a non-blocking function followed by **`MPI_Wait`** is equivalent to its blocking variant.

Testing Multiple Completions

- It is sometimes desirable to wait on multiple requests:
  - **`MPI_Waitall(count, array_of_requests, array_of_statuses)`**
  - **`MPI_Waitany(count, array_of_requests, &index, &status)`**
  - **`MPI_Waitsome(count, array_of_requests, array_of_indices, array_of_statuses)`**

- There are corresponding versions of **`MPI_Test`** for each of these.

Example Program Using Non-Blocking Calls

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] )
{
    ...
    if(rank == 0){
        for(i = 0; i < 100; i++) {
            data[i] = compute(i);
            MPI_Isend(&data[i], 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request[i]);
        }
        MPI_Waitall(100, request, MPI_STATUSES_IGNORE);
    } else {
        for(i = 0; i < 100; i++)
            MPI_Irecv(&data[i], 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &resp[i]);
        MPI_Waitall(100, resp, MPI_STATUSES_IGNORE);
    }
    ...
}
```

Example Program Using Non-Blocking Calls

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] )
{
   ...
   if(rank == 0){
       for(i = 0; i < 100; i++) {
              data[i] = compute(i);
              MPI_Isend(&data[i], 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request[i]);
       }
       MPI_Waitall(100, request, MPI_STATUSES_IGNORE);
   } else {
       for(i = 0; i < 100; i++)
              MPI_Irecv(&data[i], 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &resp[i]);
       MPI_Waitall(100, resp, MPI_STATUSES_IGNORE);
   }
   ...
}
```

compute of i=2 can start before sending i=1 completes

Example Program Using Non-Blocking Calls

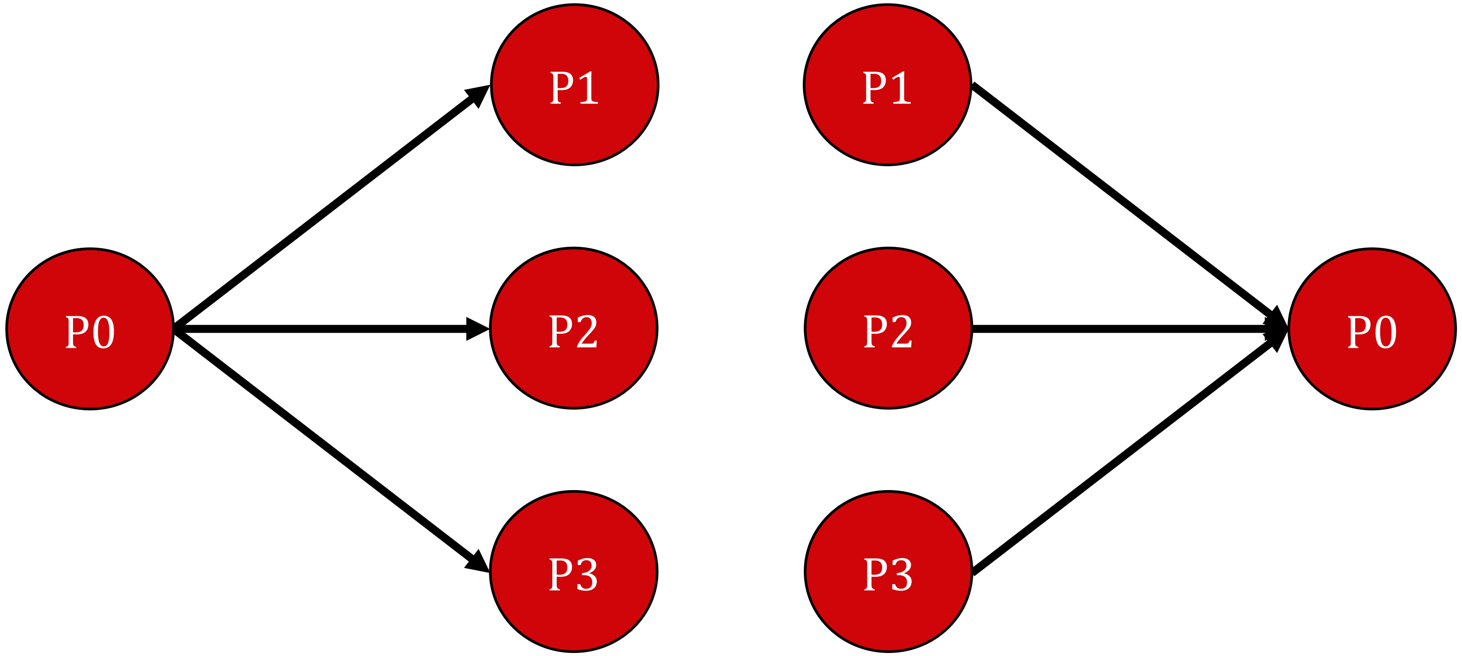
```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] )
{
  ...
  if(rank == 0){
      for(i = 0; i < 100; i++) {
            data[i] = compute(i);
            MPI_Isend(&data[i], 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request[i]);
      }
      MPI_Waitall(100, request, MPI_STATUSES_IGNORE);
  } else {
      for(i = 0; i < 100; i++)
            MPI_Irecv(&data[i], 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &resp[i]);
      MPI_Waitall(100, resp, MPI_STATUSES_IGNORE);
  }
  ...
}
```

Wait once all data has been sent out

Communication Patterns

- So far, we have only looked at point-to-point communication
- What if processes need to communicate to all ?

two processes



all processes

Collective Communications

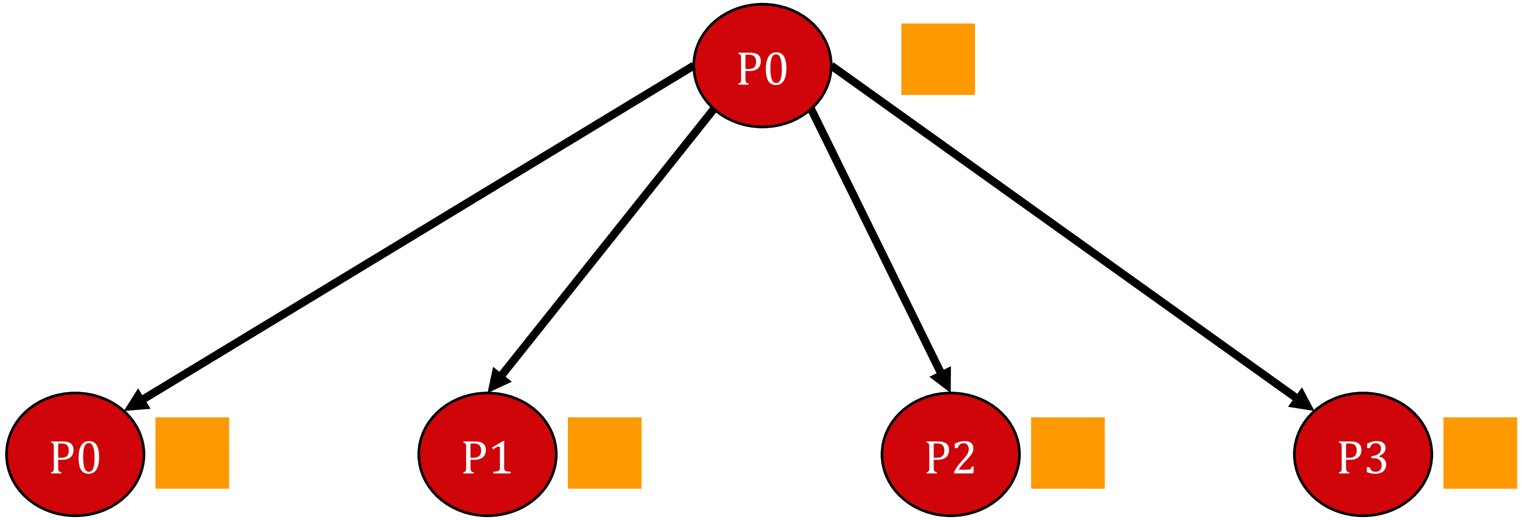
- Communications involving all processes within a communicator
- May involve computation on intermediate communicated data

- Things to consider:
  - Which data to send to which process?
  - How to accumulate multiple sources of data?
  - How to best optimize for performance?

- Some patterns are common for many AI algorithms
- These patterns have dedicated primitives in all MPI systems

Broadcast

- Send the same data to all processes in the group



Broadcast

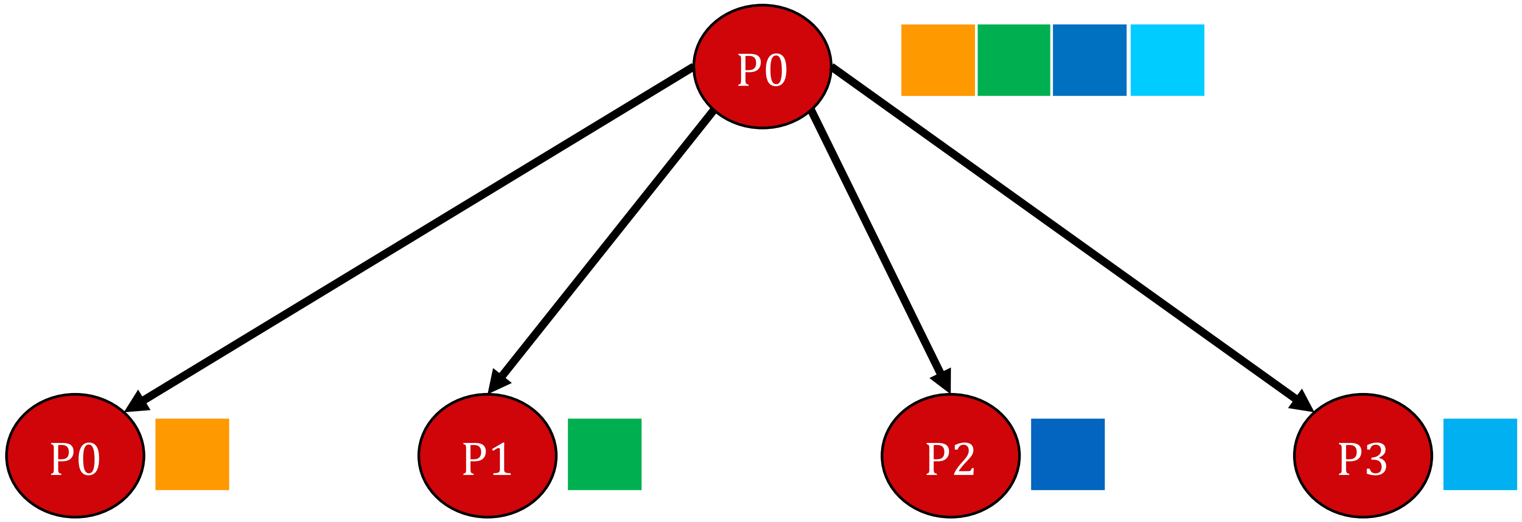
```
int MPI_Bcast(
        void *buffer,
        int count,
        MPI_Datatype datatype,
        int root,
        MPI_Comm comm )
```

- `void *buffer` → Pointer to broadcasted data
- `int count` → Number of broadcasted elements
- `int root` → Rank of process that will broadcast
- `MPI_Comm comm` → Communicator in which to broadcast

- **`root`** process will send the data from **`buffer`**
- Other processes in the communicator will receiver the data in **`buffer`**
  - Note: **`buffer`** for each process are all unique

Scatter

- Send elements in an array to consecutive processes in a group



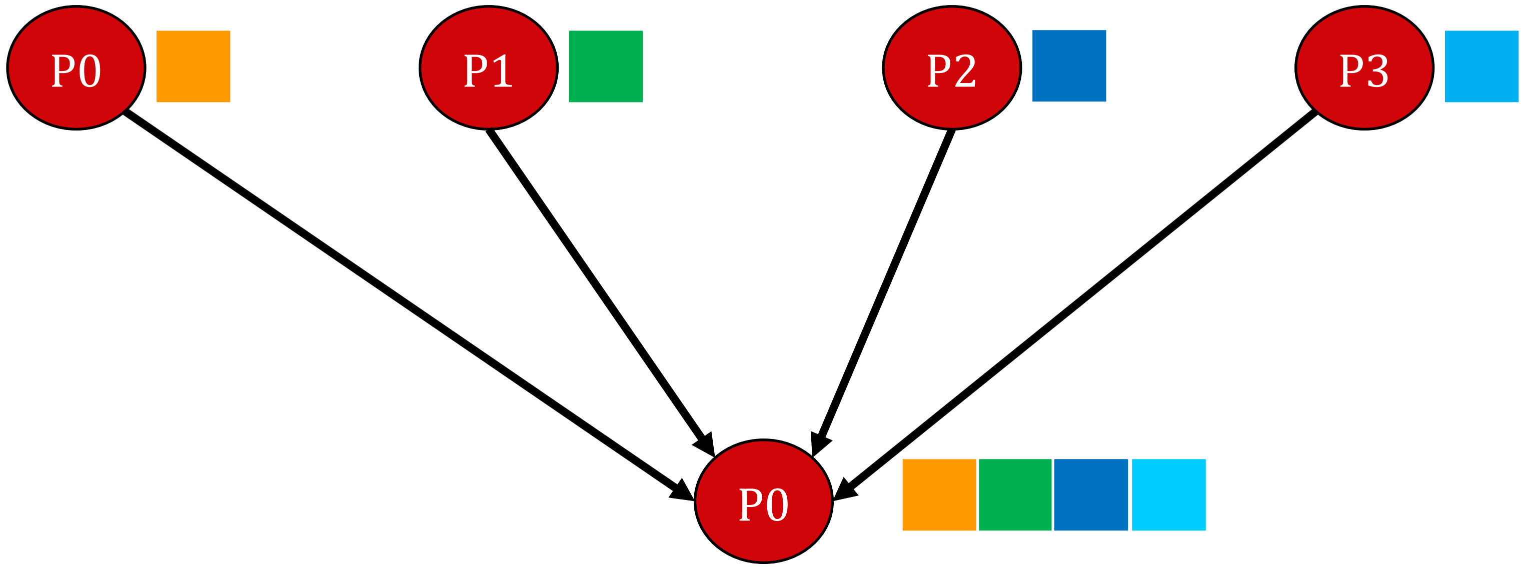
Scatter

```
int MPI_Scatter(
        const void *sendbuf,      → Address of send buffer (for root process)
        int sendcount,            → Number of elements to send to each process
        MPI_Datatype sendtype,
        void *recvbuf,            → Buffer in which to receive data
        int recvcount,            → Number of elements to receive
        MPI_Datatype recvtype,
        int root,                 → Rank of sending process
        MPI_Comm comm)            → Communicator in which to scatter
```

- `sendcount * num_processes` = total count of data to scatter
- Data is transmitted in increasing order of process rank

Gather

- Gathers elements from consecutive processes in group into an array



Gather

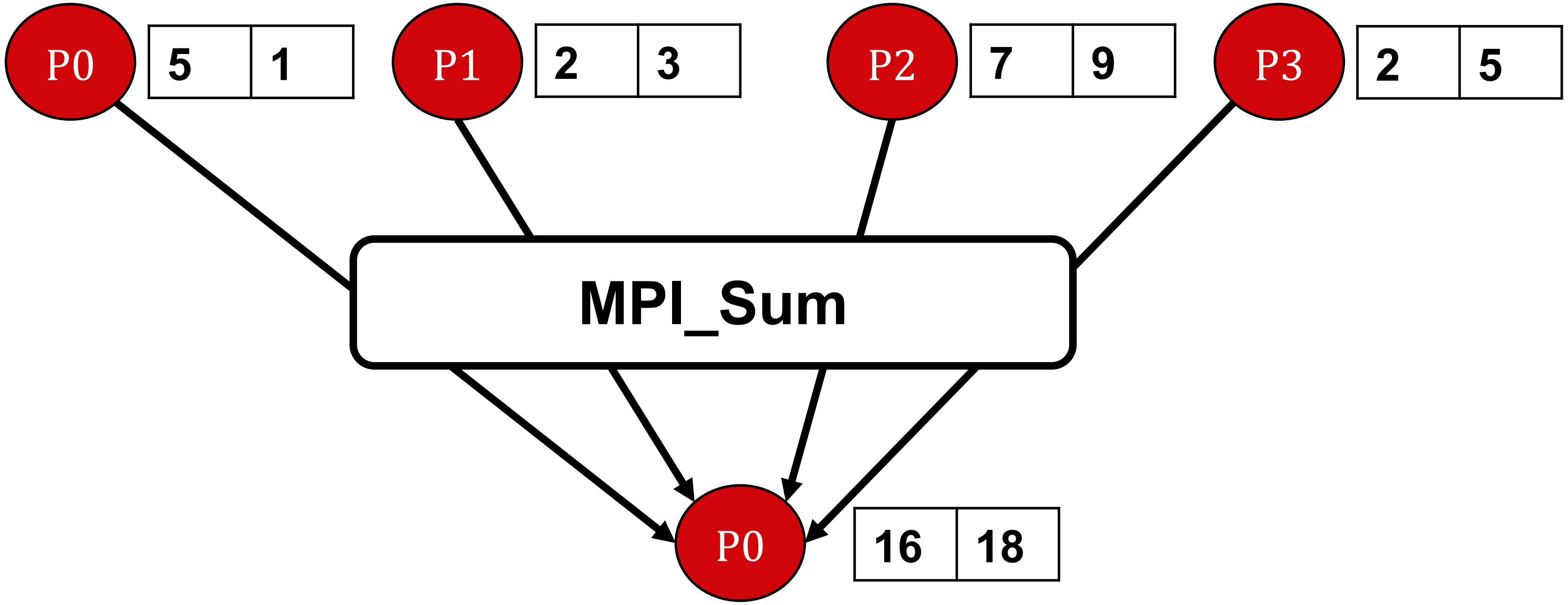
```
int MPI_Gather(
      const void *sendbuf,
      int sendcount,
      MPI_Datatype sendtype,
      void *recvbuf,
      int recvcount,
      MPI_Datatype recvtype,
      int root,
      MPI_Comm comm)
```

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

- Same parameters as scatter but works in reverse
- **`root`** process receives data from all processes in **`recvbuf`**

Reduce

- Takes an array of input elements on each process
- Returns an array of output elements to the root process
  - Given a specific operation



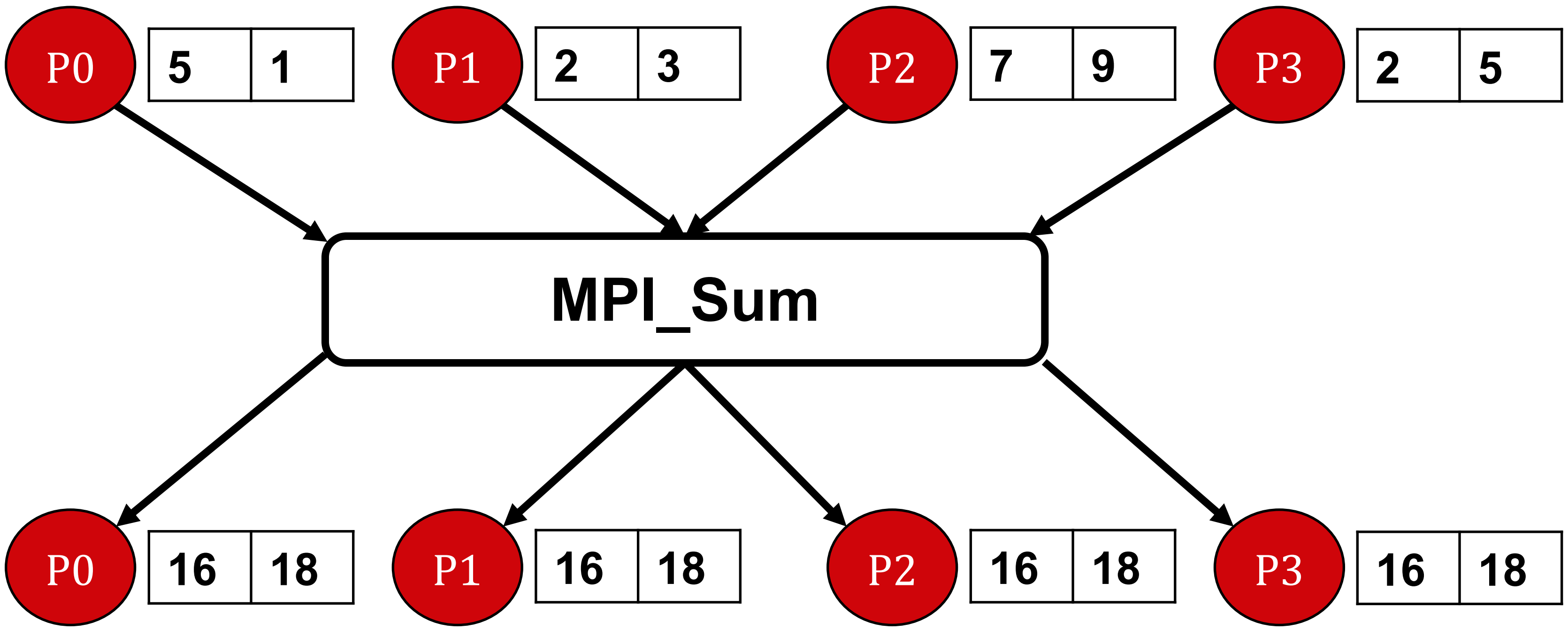
Reduce

```
int MPI_Reduce(
        const void *sendbuf,     → Address of send buffer
        void *recvbuf,           → Address of receive buffer
        int count,               → Number of elements to be sent for reduction by each process
        MPI_Datatype datatype,
        MPI_Op op,               → The reduction operation to perform
        int root,                → The rank of the process that receives the reduced data
        MPI_Comm comm)
```

- **`MPI_Op`** can be many common operations such as **`MPI_MAX`**, **`MPI_MIN`**, **`MPI_SUM`**, **`MPI_PROD`**, etc.
- Custom reduction operations can also be defined

AllReduce

- Similar to reduce but results are distributed to all processes



AllReduce

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

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

- Same parameters as **`MPI_Reduce`** but no **`root`** process
- Equivalent to reducing first and then broadcasting result

Summary

- Message passing model allows multi-node scalability
- MPI provides easy to use functions to parallelize programs
  - Higher programmer effort compared to OpenMP
  - More control and improved scalability compared to OpenMP
- Non blocking functions can be used for asynchronous communication
- Collectives provide a big benefit for common communication patterns