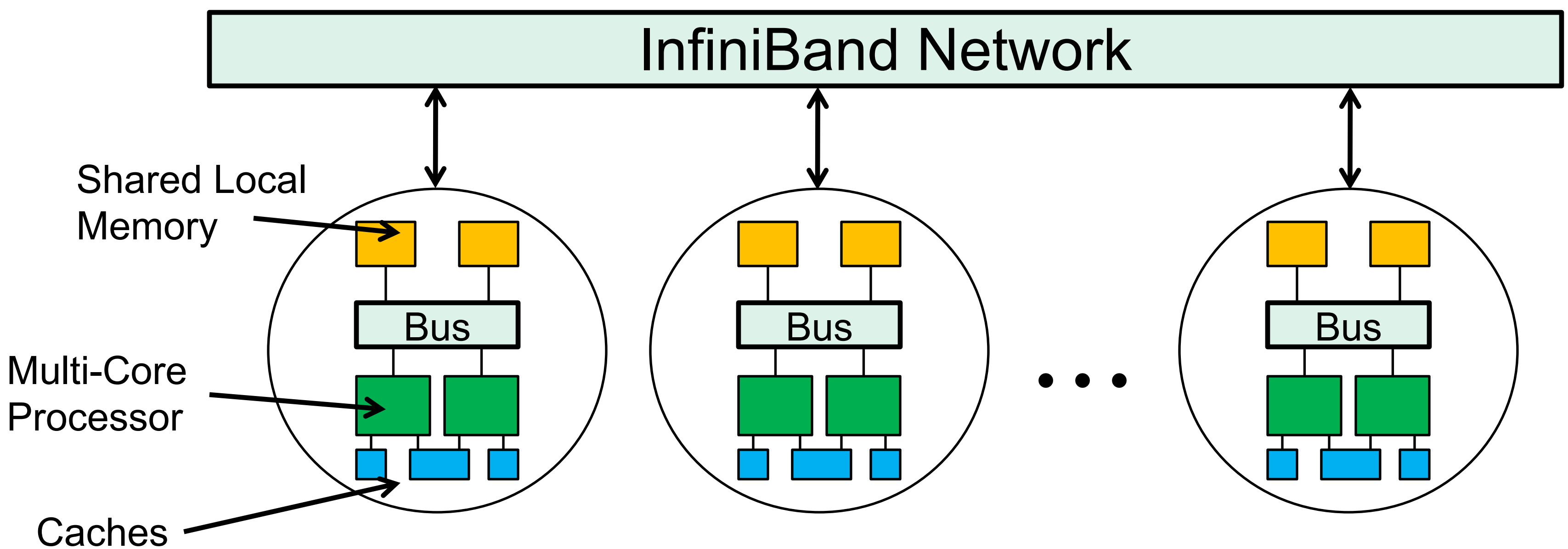
Message-Passing Concepts

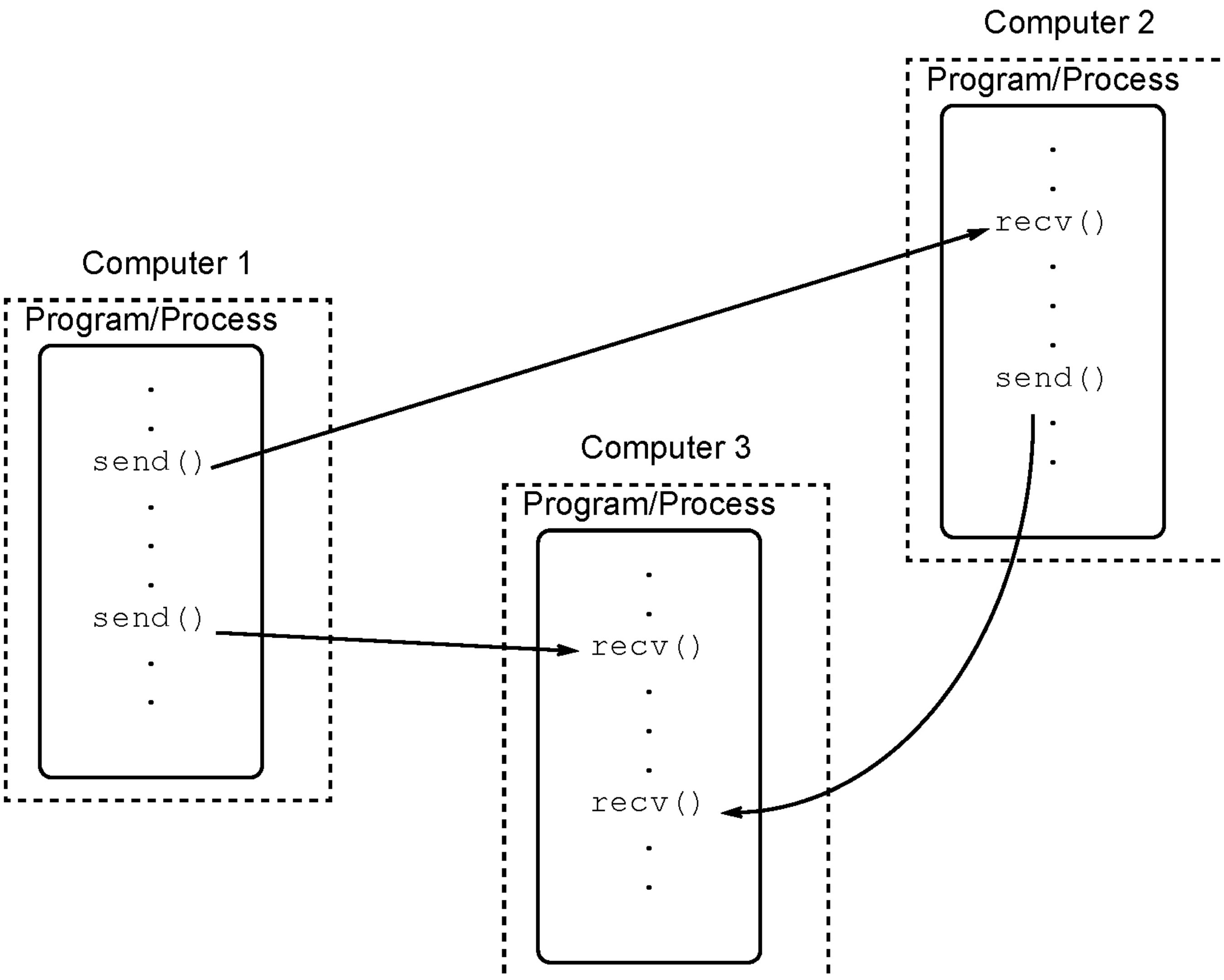
CPSC 424/524
Lecture #8
October 22, 2018

Linux Cluster Parallel Computers

- Yale Clusters (e.g. Omega)
  - Hybrid: Many dual-chip, multi-core shared-memory compute nodes connected by one or more networks
  - Each node computes independently, communicating with other nodes by passing messages as needed.

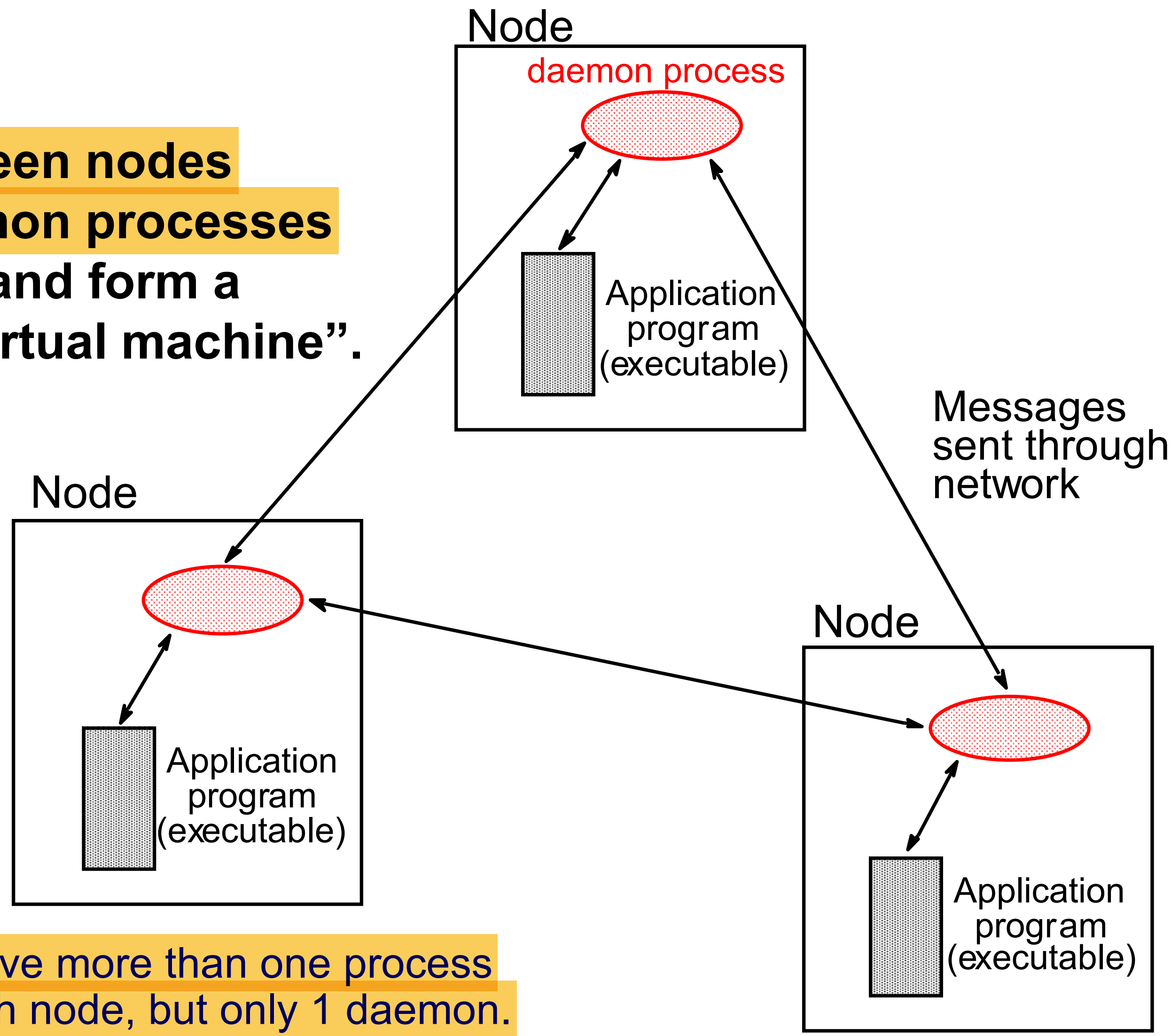


Message-Passing Concepts – Application View



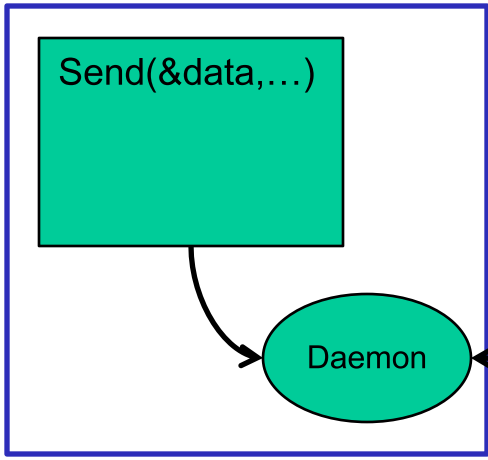
Message-Passing Concepts – System View

Message routing between nodes typically done by daemon processes that run on the nodes and form a "network-connected virtual machine".

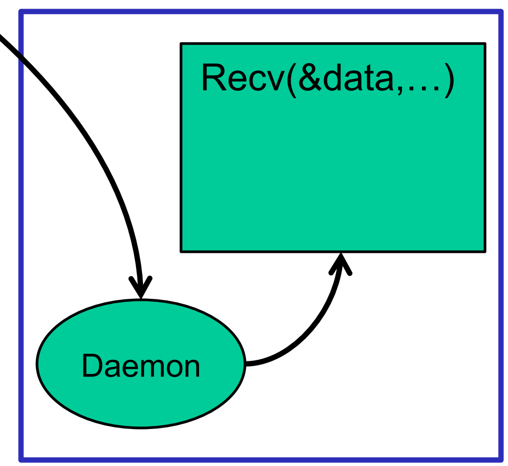


Can have more than one process on each node, but only 1 daemon.

Messaging: One possibility for what really happens



1. Local daemon makes copy of data. (`Send()` returns)
2. Local daemon communicates with remote daemon to set up actual data transfer.
3. Local daemon transfers the data.



1. Local daemon hears from remote daemon and negotiates the data transfer.
2. Local daemon receives data from remote daemon.
3. Local daemon stores the data in proper location. (`Recv()` returns)

Message-Passing Software APIs

Message-passing Application Programming Interfaces (APIs) are sets of library calls or similar operations added to base serial programming languages to support:

1. Process management: Starting, stopping, and controlling programs (processes) running on the nodes. (More than one process per node may be allowed.)
2. Communication: One or more methods for sending data from one process to another.
3. Collective operations: Coordinated communication among multiple processes, often combined with arithmetic (e.g., reduction operations, barriers, etc.)

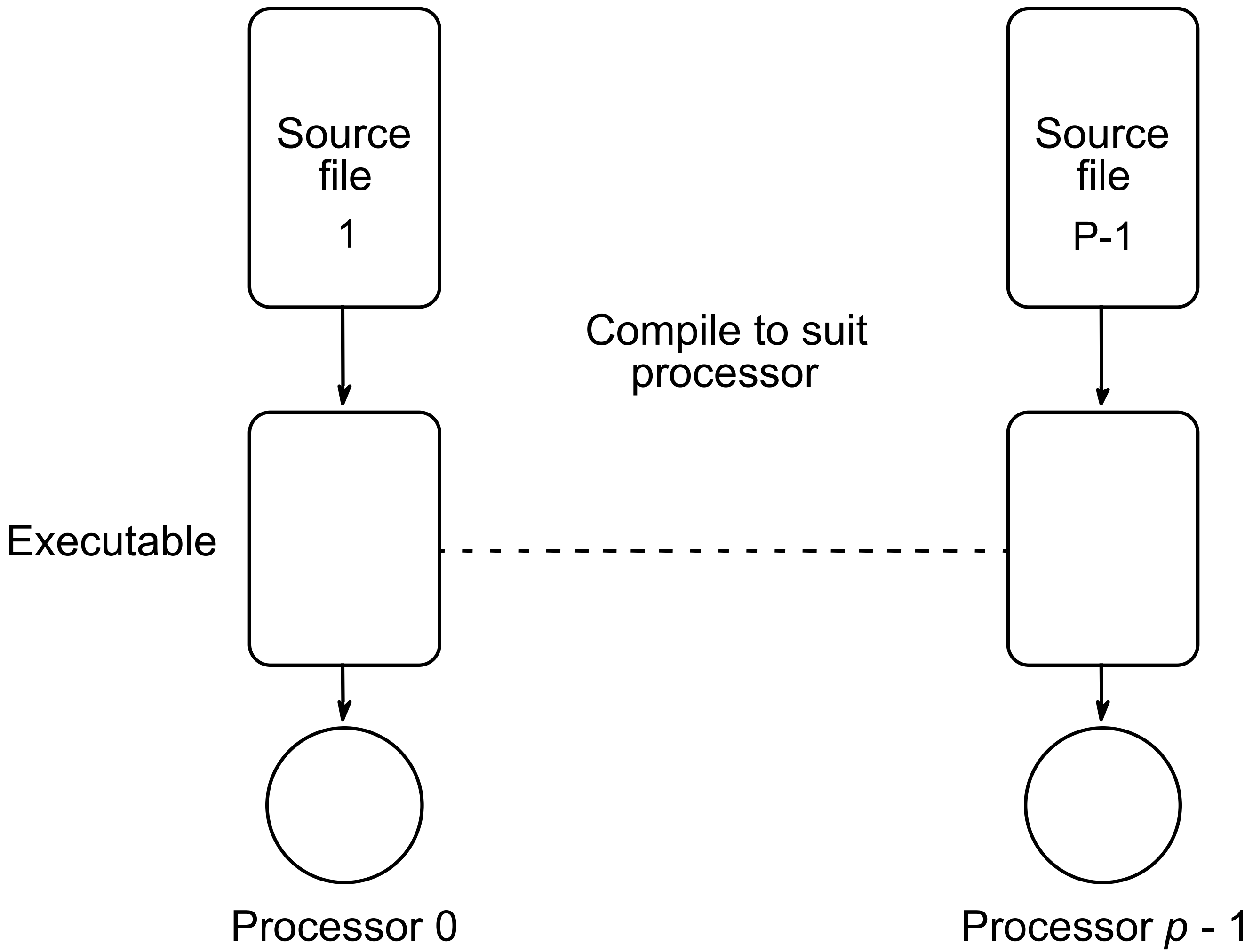
The actual definition of the integration with a particular language is often called a "binding"

MPI (Message Passing Interface) Standard

- Message passing standard developed to foster more widespread use through standardization and (logical) portability
- Maintained by the MPI Forum (www.mpi-forum.org).
- Current version: 3.1 (released in 2015); Working on 4.0
- Based on runtime libraries capable of working with various serial programming languages (e.g. C, C++, F77, F90)
- Defines interfaces & functional semantics, not implementation.
  - What should programmer expect when calling MPI_Send?
  - What does it mean when MPI_Send returns?
  - But NOT: How fast or efficient is MPI_Send?
  - Or even: Must daemons make copies of messages before sending?

Multiple Program, Multiple Data Model (MPMD)

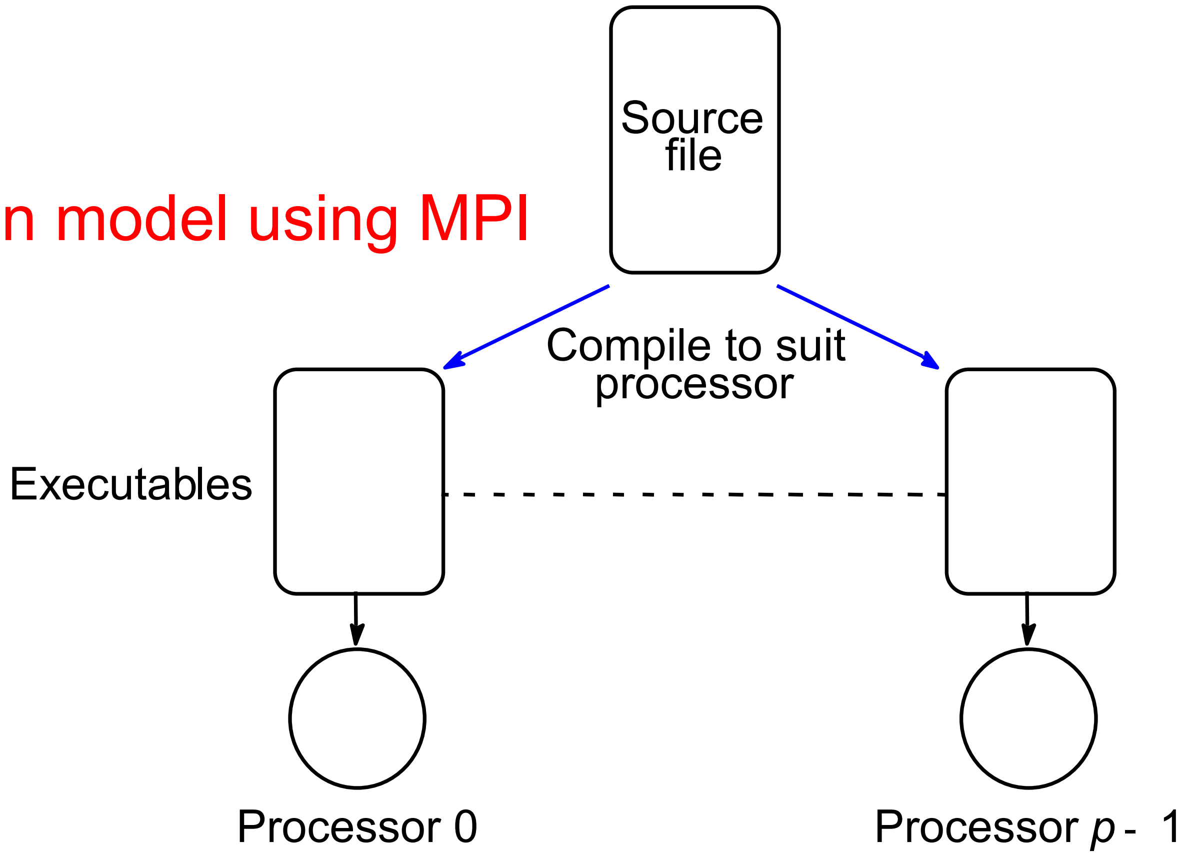
- MPI supports execution of different programs by each processor



Single Program, Multiple Data Model (SPMD)

- Almost always, though, all processors execute the same MPI program
- Base-language control statements are used to select what instructions execute on each processor.

Most common model using MPI



Static Process Creation

- MPI execution command starts all executables together.

```
mpiexec -n 4 program
```

- This starts daemons on the nodes and instructs each one to start MPI processes, each running a local copy of your executable
- Once started, each process calls `MPI_Init()`, which initializes communication protocols among the entire group of processes.
- MPI supports multiple groupings of the processes (communicators) to better control interprocess communication.

  `MPI_COMM_WORLD` is a special communicator containing the entire group of processes; often, programs use other communicators that are subsets of `MPI_COMM_WORLD`.

Communicators and Processes

In MPI, each process is a member of one or more communicators. All processes are members of `MPI_COMM_WORLD`.

Within each communicator, each process is given a process number called a rank, assigned sequentially starting from zero.

Programs use control constructs, typically `if` statements, to direct specific processes to perform specific actions. E.g.:

```
if (rank == 0) ...     ;/* rank 0 do this */
if (rank == 1) ...     ;/* rank 1 do this */
        .
        .
        .
```

MPI Communicators

- Define communication domains: sets of processes that can communicate among themselves. Processes generally have different ranks in different communicators.

- Communication domains of libraries can be separated from that of a user program.

- Used in all point-to-point and collective MPI message-passing communications.

Master-Worker Organization

Often, one process (the master) performs management (and possibly some computation), while other processes (the workers) perform only computation.

Master responsibilities: I/O, problem decomposition, worker control, error handling, etc.

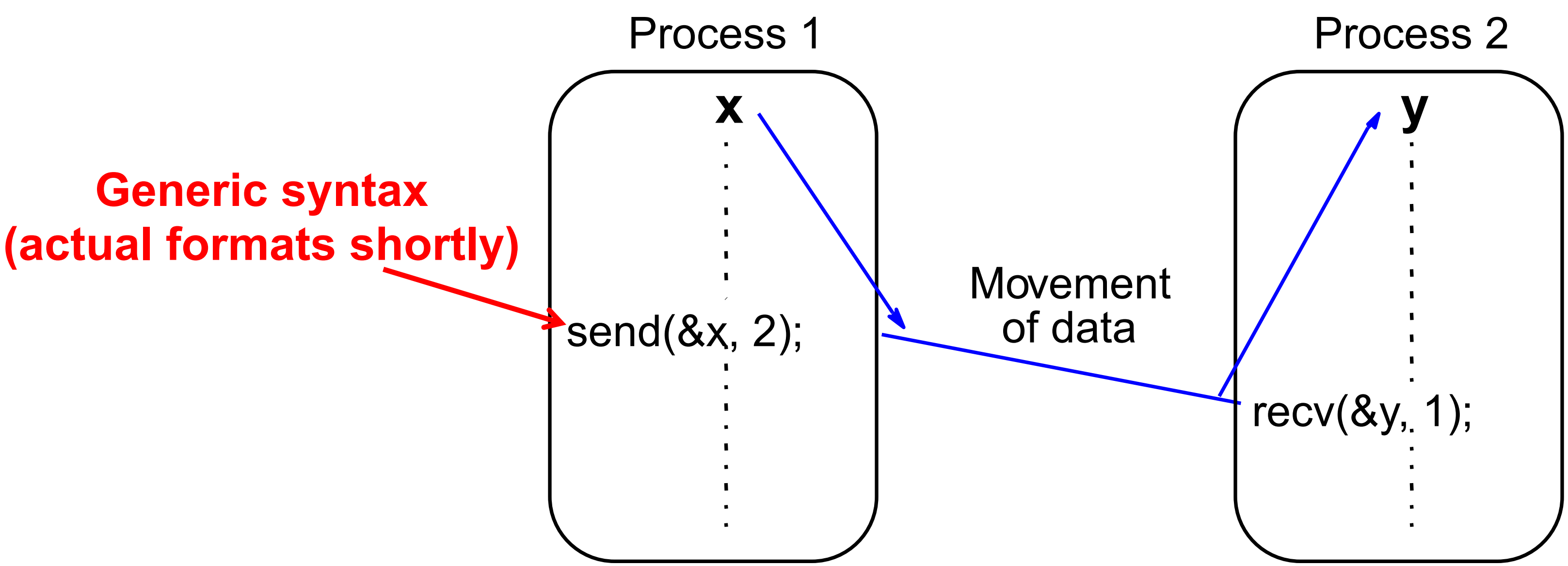
Worker responsibilities: Computation

SPMD Implementation:

```
if (rank == 0) ...   ;// master does this
else ...          ; // all workers do this
```

Point-to-Point Messaging

Passing a message between processes using generic send() and recv():



Some semantic questions about messaging:

1. What should be expected when an operation RETURNS?
2. What does it mean for an operation to COMPLETE? (Must that always occur?)
3. How should send & recv operations be MATCHED?
4. How should operations be ORDERED? How is ordering controlled?
5. Must all operations make PROGRESS? (If so, must progress be FAIR?)

Syntax of MPI_Send (in C)

```
MPI_Send(buf, count, datatype, dest, tag, comm);
```

- buf: Address of send (message) buffer
- count: Number of items to send
- datatype: Datatype of each item
- dest: Rank of destination process
- tag: Message tag
- comm: Communicator

Message Envelope

Fortran:

```
CALL MPI_SEND(buf, count, datatype, dest, tag, comm, ierr)
```

Notes:

1. For no-op: Set destination process to **MPI_PROC_NULL**
2. Returns **int** error code in C

Syntax of MPI_Recv (in C)

```
MPI_Recv(buf, count, datatype, src, tag, comm, status);
```

- `buf`: Address of receive buffer
- `count`: Maximum number of items to receive
- `datatype`: Datatype of each item
- `src`: Rank of source process
- `tag`: Message tag
- `comm`: Communicator
- `status`: Address of status

Message Envelope

Fortran:

```
CALL MPI_RECV(buf, count, datatype, dest, tag, comm, status, ierr)
```

Notes:

1. Source process can be `MPI_ANY_SOURCE` or `MPI_PROC_NULL`
2. Tag can be `MPI_ANY_TAG`
3. Returns `int` error code in C
4. `status` variable returns info about `src`, `tag`, `error`, and other info

Semantics of Blocking Communication

Blocking Communication: MPI routines do not <u>return</u> until at least all *<u>local</u>* actions have completed.

After return, all local variables can be used or altered safely.

MPI_Send() - Message may or may not have reached its destination, but no longer depends on sending process's data.

MPI_Recv() – Message has been received and data is available.

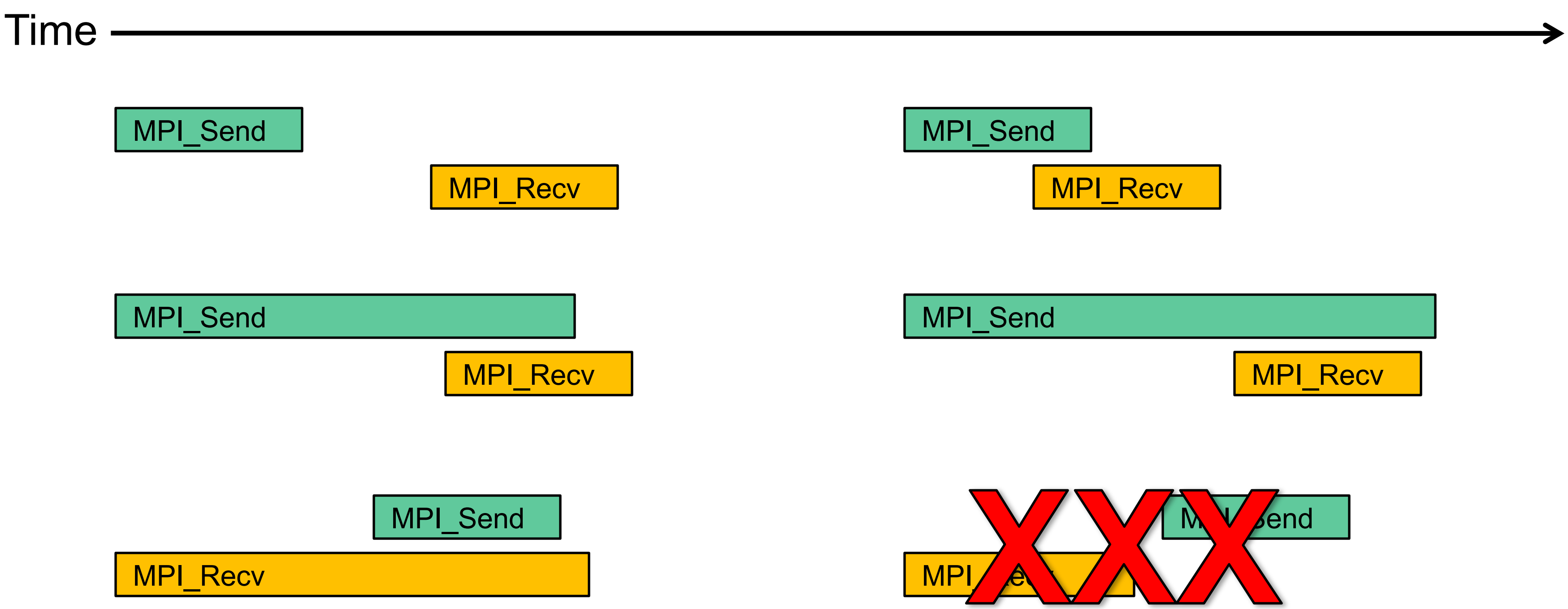
MPI includes variants of MPI_Send() and MPI_Recv() with different semantics.

NOTE: MPI implementations may be more restrictive, but not less!

Semantics of Blocking Communication

- **Blocking**:
  - MPI_Send: Returns only when it is safe to modify the send buffer
  - MPI_Recv: Returns only when the buffer contains the message
  - From system view: these may not be complete when they return

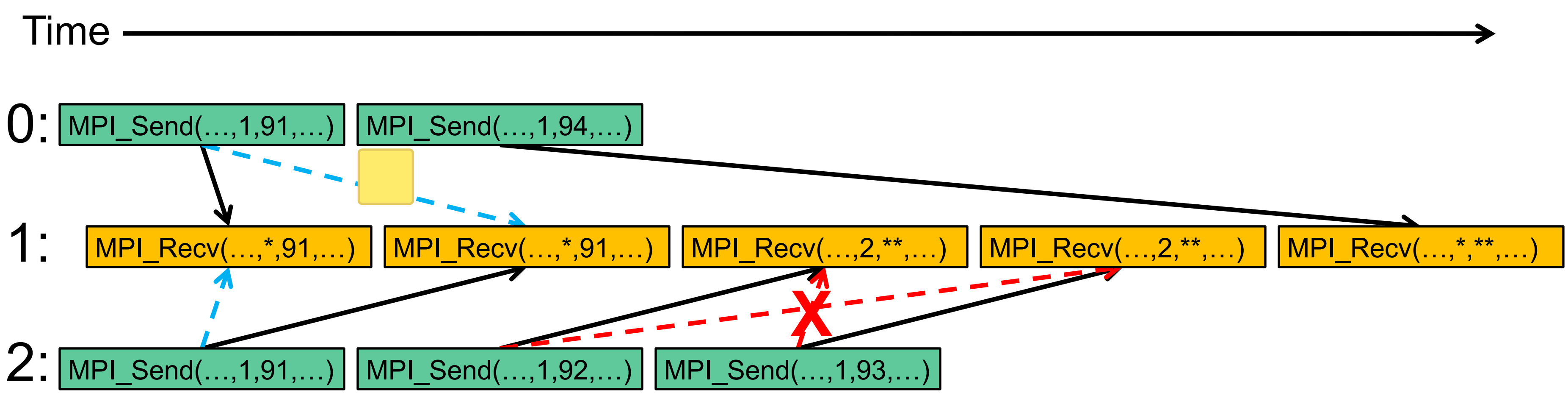
Many possible timing scenarios for pairwise completion:



Semantics of Blocking Communication (cont.)

- **Locally-ordered**. In any given single-threaded process:
  - MPI_Send: "Non-overtaking". If there are two active sends to a single destination matching a given recv operation, the second send cannot be received while the first one is still pending.
  - MPI_Recv: Sequentialized. Incoming msgs match earliest recv

Possible timing scenarios:



*=MPI_ANY_SOURCE **=MPI_ANY_TAG

More on Message Envelope

Message contents are described by a Message Envelope that contains all the data used to match sends and receives.

What's on the "outside" of the envelope?

- Rank of message destination
- Name of communicator
- Message tag
- [Implicitly] Rank of message source

Notice what's not:

- Data type
- Number of items

What Happens in this Example?

```
#include "mpi.h"
  int tag=1, count1, count2, rank;
  float *buf1, *buf2;
  . . .
  MPI_Get_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0)
    MPI_Send(buf1, count1, MPI_FLOAT, 2, tag, MPI_COMM_WORLD);
    MPI_Send(buf2, count2, MPI_FLOAT, 1, tag, MPI_COMM_WORLD);
  else if (rank == 1)
    MPI_Send(buf2, count2, MPI_FLOAT, 2, tag, MPI_COMM_WORLD);
    MPI_Recv(buf2, count2, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, &status);
  else if (rank == 2)
    MPI_Recv(buf1, count1, MPI_FLOAT, *, tag, MPI_COMM_WORLD, &status);
    MPI_Recv(buf2, count2, MPI_FLOAT, *, tag, MPI_COMM_WORLD, &status);
  }
  . . .
```

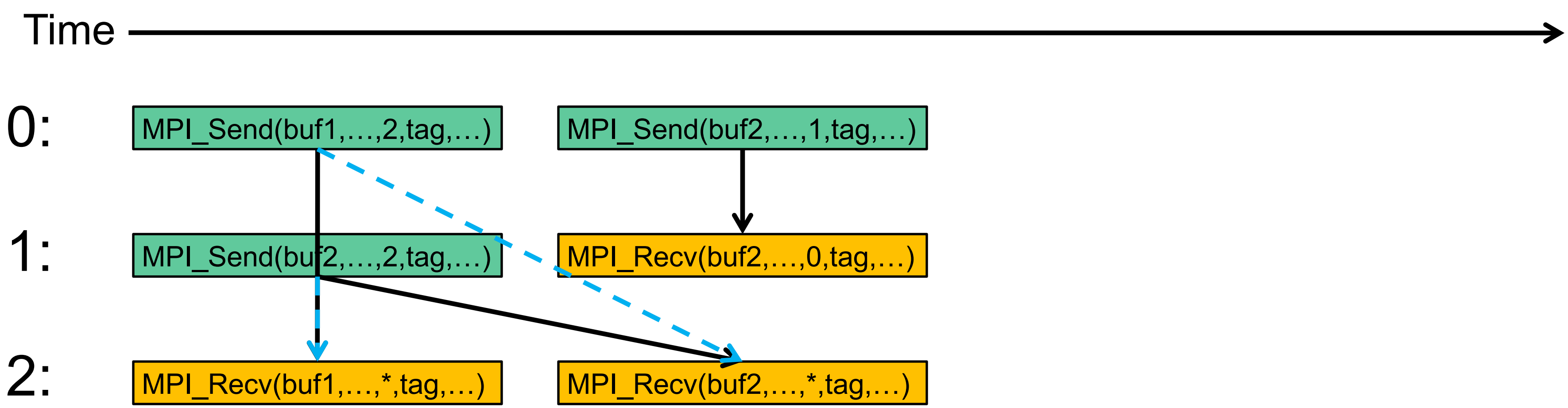
*=MPI_ANY_SOURCE

Semantics of Blocking Communication (cont.)

Ordering is a partial ordering, and it IS NOT transitive!! (Note assumption about single-threaded processes.)

Results may depend on the implementation and external effects

Possible timing scenarios:



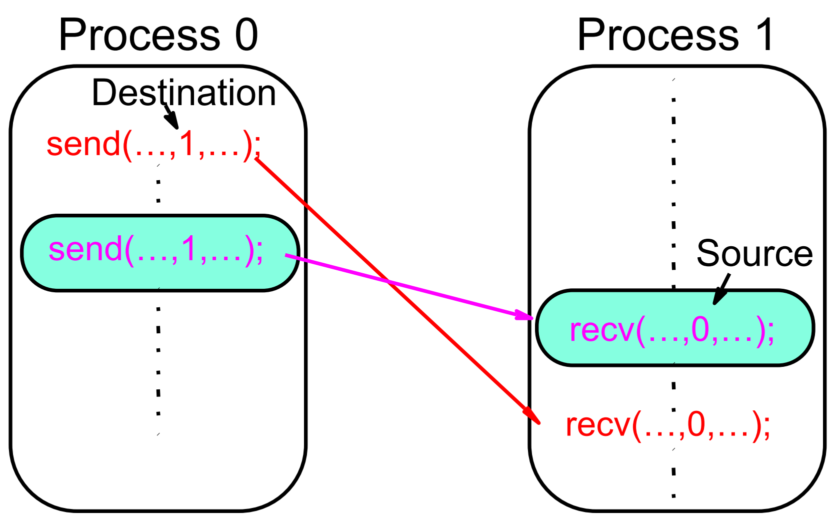
*=MPI_ANY_SOURCE

Message Tags

- Used to distinguish between different messages. The tag is an integer (which may be 16 bits on some machines!). Maximum tag value is given by the attribute **`MPI_TAG_UB`** (implementation dependent, but must be at least 32,767).
- Message tag is carried with message. (May be used as the entire message content in some cases.)
- If specific matching is not required, a wild card message tag may be used (**`MPI_ANY_TAG`**). Then **`MPI_Recv()`** will match any **`MPI_Send()`** if the source and communicator match. However, this may lead to unexpected behavior in some cases.

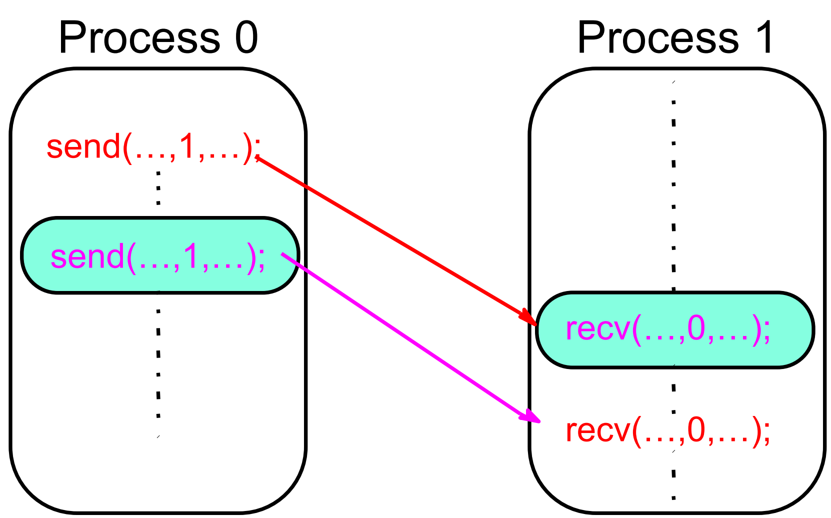
Tags Crucial for Libraries

Intended behavior



op in library

Possible behavior (without tags)



Sample MPI Datatypes

C		Fortran	
signed int	MPI_INT	integer	MPI_INTEGER
signed long	MPI_LONG	integer	MPI_INTEGER
float	MPI_FLOAT	real	MPI_REAL
double	MPI_DOUBLE	double precision	MPI_DOUBLE_PRECISION
float _Complex	MPI_C_COMPLEX	complex	MPI_COMPLEX
signed char	MPI_CHAR	character(1)	MPI_CHARACTER
Matches any bytes	MPI_BYTE		MPI_BYTE
“Packed” buffers	MPI_PACKED		MPI_PACKED

Type Matching

Message-passing has 3 phases:

1. Copy from send buffer to assemble a message (may be "on the fly" directly onto the wire)
2. Message transfer from send daemon to recv daemon
3. Disassemble message, copying into recv buffer

In correct programs, types must match in each phase:

- For phases 1 & 3: Datatypes specified to send/recv must match data in the send/recv buffer
  - This is a match between MPI type and language type
  - **`MPI_BYTE`** and **`MPI_PACKED`** are special
- For Phase 2: MPI datatypes provided to send and recv must match (in meaning). (E.g., **`MPI_INT != MPI_LONG`**)
- Unmatched datatypes make the operation erroneous, with unpredictable results (but not necessarily a crash)

Status Argument

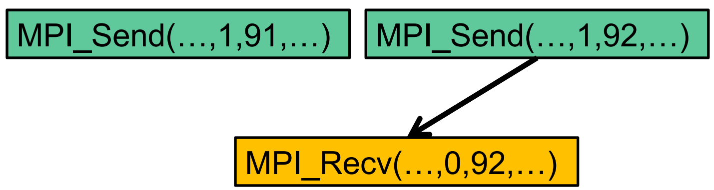
- `status` returns information about Recv operations
  - Fortran: Integer array of size `MPI_STATUS_SIZE`
  - C: Structure of type `MPI_Status`
  - C++: Object of type `MPI::Status`
- "Visible" information: source, tag, error
- "Invisible" information: count of items received

Item	C	Fortran	C++ Member Functions
Source	`status.MPI_SOURCE`	`status(MPI_SOURCE)`	`Get_source()`
Tag	`status.MPI_TAG`	`status(MPI_TAG)`	`Get_tag()`
Error	`status.MPI_ERROR`	`status(MPI_ERROR)`	`Get_error()`
Count	`MPI_GET_count(*status, datatype, *count)`	`MPI_GET_COUNT(status, datatype, count)`	`Get_count(datatype)`

Note: Error is the same as the returned error code

Semantics of Blocking Communication (cont.)

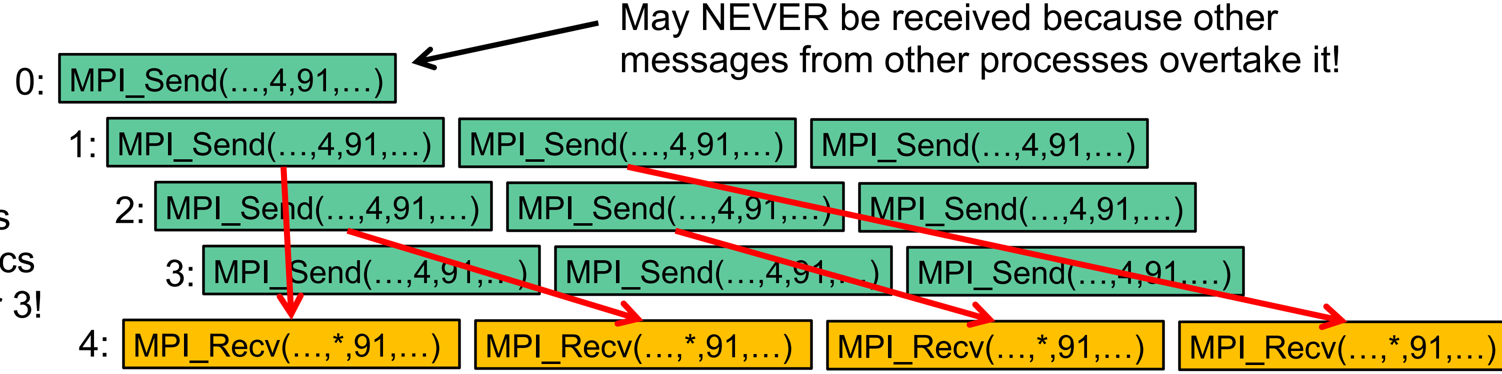
- **Progress**
  - If a matching pair of send & recv operations have been initiated on two processes, then implementations must ensure that at least one of them completes, independent of other actions in the MPI system.



Once 2nd send is initiated, then processing of 1st Send cannot prevent one of the matching Send/Recv operations from making progress. The 2nd Send overtakes the 1st Send. (Does this violate the ordering rule?)

- **Fairness (Do all processes/operations get serviced?)**
  - NO GUARANTEES!

May NEVER be received because other messages from other processes overtake it!



Proc 4 only receives messages from Procs 1-2, never from 0 or 3!

Summary: Semantics of Blocking Communication

- Blocking implementation
  - MPI_Send: Returns only when it is safe to modify buffer
  - MPI_Recv: Returns only when the buffer contains the message
  - Completion may be later than return
- Locally-ordered. In any given single-threaded process:
  - MPI_Send: “Non-overtaking”. when to a single destination with a matching recv operation
  - MPI_Recv: Serialized. Incoming msgs match earliest matching recv
- Progress
  - If a matching send/recv pair has been initiated on two processes, then at least one of these will complete. However, which one is not specified. (Possible race condition!)
- Fairness
  - No guarantees!

Blocking vs. Non-Blocking Communication

- Blocking Communication (`MPI_Send`, `MPI_Recv`)
  - Send:
    - Returns only when it is safe to modify message buffer
    - Completes later (or possibly never)
  - Recv:
    - Completes when the buffer contains message data
    - Returns upon completion (if completion ever takes place)

- Non-Blocking Communication (`MPI_Isend`, `MPI_Irecv`)
  - Send:
    - Returns immediately (may not be safe to modify message buffer)
    - Completes later (or possibly never)
  - Recv:
    - Returns immediately
    - Completes later (or never) at which time buffer contains message data

Non-Blocking Communication

- **Isend/Irecv <u>return</u> immediately**
  - May enhance performance or provide more flexibility
  - Avoids need for local buffering since caller isn't blocked
  - Remember: "returned" does not mean "complete"
  - May be mixed & matched with blocking operations
  - Requires care! Don't modify message buffer too soon!
- **"Request Object" used to identify each posted operation**
  - `MPI_Isend(buf, cnt, type, dest, tag, comm, request)`
  - `MPI_Irecv(buf, cnt, type, source, tag, comm, request)`
- **Completion**
  - Asynchronous with <u>unpredictable delay</u>
  - Independent of whether user checks for completion

Non-Blocking Communication (Cont.)

- **User <u>should</u> check non-blocking operations for completion in most cases**
  - <u>Blocking Checks</u>:

```
MPI_Wait(request, status)
MPI_Waitany(cnt, req_array, indx, status)
MPI_Waitall(cnt, req_array, status_array)
MPI_Waitsome(. . . )
```

  - <u>Non-blocking Checks</u>:

```
MPI_Test(request, flg, status)
MPI_Testany(cnt, req_array, indx, flg, status)
MPI_Testall(cnt, req_array, flg, status)
MPI_Testsome(. . .)
```

Args may be: IN or INOUT or OUT

Summary: Semantics of Non-Blocking Communication

- **Non-Blocking**
  - MPI_Isend and MPI_Irecv operations return immediately. (Use MPI_Wait or MPI_Test to check for completion.)
  - No local system buffering in most implementations (more efficient, no real need since caller can proceed anyway)
- **Local Ordering**
  - "Non-overtaking" property extends to non-blocking Isends
  - As before, this affects matching, not completion
- **Progress**
  - Operations are "active" when processes have posted matching MPI_Isend and MPI_Irecv operations. Then at least one of the two will complete. It is not specified which one(s) will complete.
- **Fairness**
  - No guarantees!

Polling for Incoming Messages (Blocking)

- **`MPI_Probe(source, tag, comm, status`**)
  - Blocks until there is a message with matching envelope
  - **`status`** is the same as **`MPI_Recv`** would have returned
  - Message <u>not</u> received by **`MPI_Probe`**, but will be received by the next **`MPI_Recv`** in the same communicator using the **`source`** and **`tag`** provided in **`status`**.
  - Useful if you need to know about the message before receiving it (e.g., to allocate buffer space).

Polling for Incoming Messages (Non-Blocking)

- **`MPI_Iprobe(source, tag, comm, flag, status`**)
  - Returns TRUE if there is a message with matching envelope that can be received; otherwise returns FALSE.
  - **`status`** is the same as **`MPI_IRecv`** would have returned
  - Message not received by **`MPI_IProbe`**, but will be received by the next **`MPI_Recv`** or **`MPI_IRecv`** in the same process with the same communicator and using the **`source`** and **`tag`** provided in **`status`**.
- **`MPI_Cancel(request`**)
  - Cancels pending non-blocking communication operation
  - Useful if you know that the operation will never complete

MPI Point-to-Point Communication Modes

- **Standard** (`MPI_Send, MPI_Isend`)
  - Send completes when message buffer is reusable (independent of Recv)
  - May or may not use system-level buffering (implementation dependent)
  - Non-local (completion *may* depend on receiving process)
- **Buffered** (`MPI_Bsend, MPI_Ibsend`)
  - Like Standard, but user provides buffer for messages (1 per process)
  - Local (completes when data is in local user-provided buffer)
- **Synchronous** (`MPI_Ssend, MPI_Issend`)
  - Like Standard, but Recv will have started when Send completes
  - Implies synchronization between source and destination
- **Ready** (`MPI_Rsend, MPI_Irsend`)
  - Like Standard, but call asserts that matching Recv has been posted. (Otherwise, operation is erroneous and behavior is undefined)
- `MPI_Recv, MPI_Irecv` are the only Recv operations