UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Computing

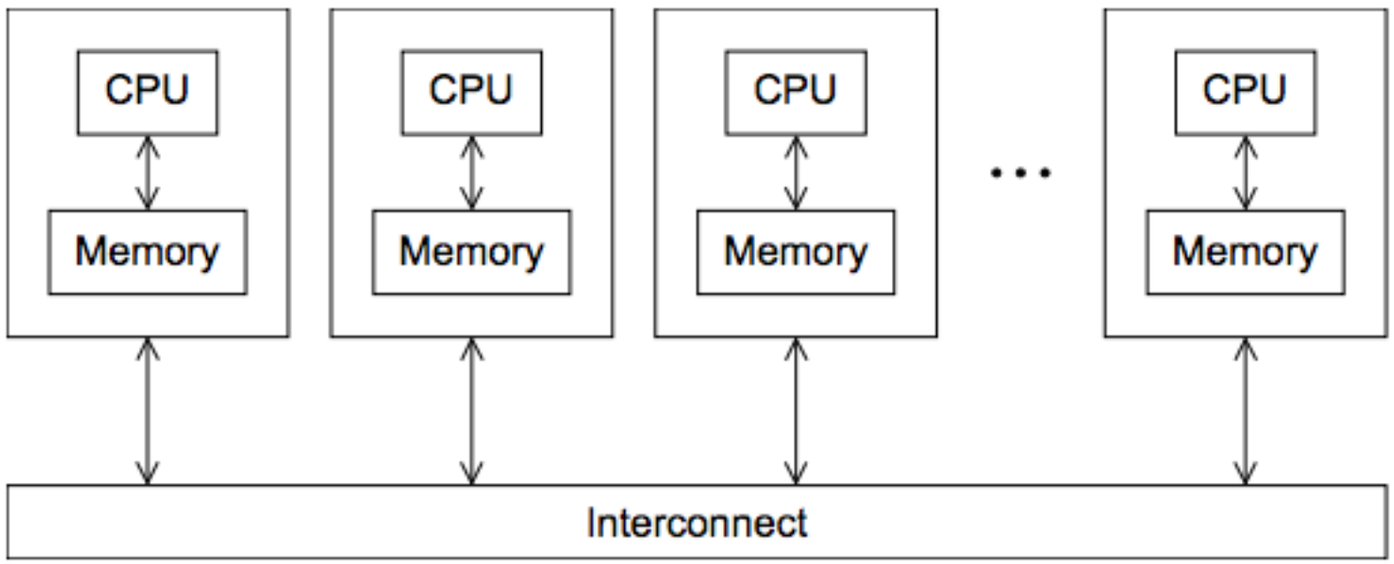
Prof. Marco Bertini

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Distributed memory: message passing

Distributed memory systems

- From a programmer's point of view, a distributed-memory system consists of a collection of core-memory pairs connected by a network, and the memory associated with a core is directly accessible only to that core.
  - Information is passed between nodes using the network
- No cache coherence and no need for special cache coherency hardware



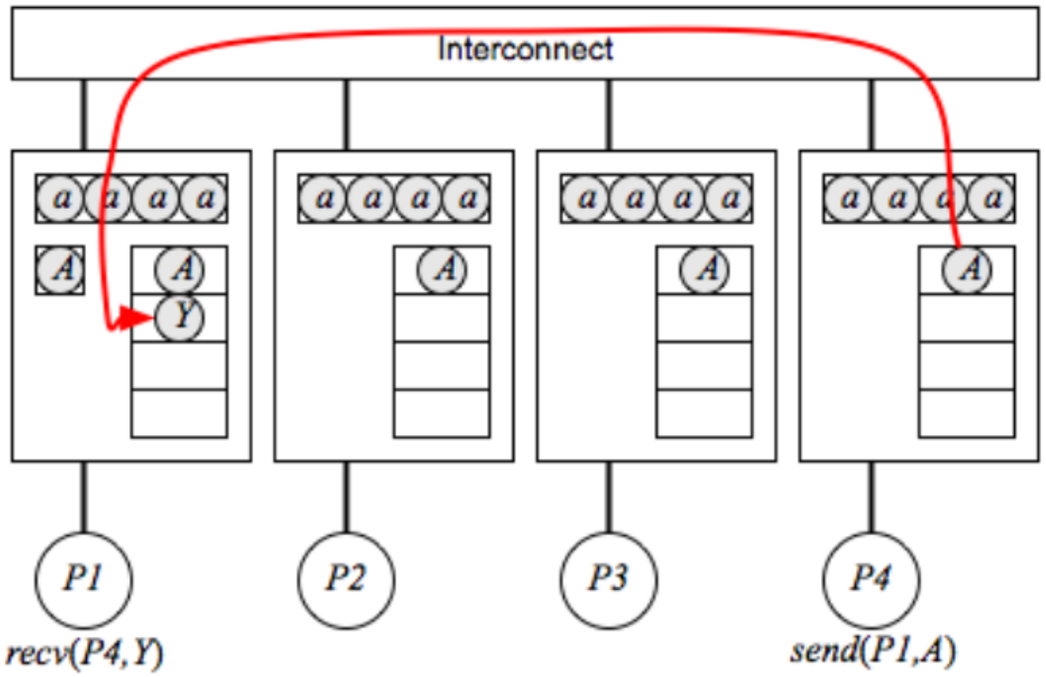
Parallel Computing Model

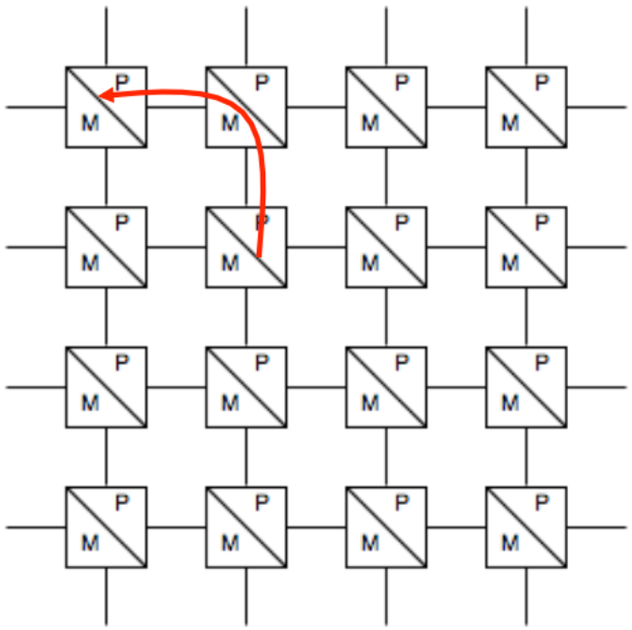
- In this context SPMD is the typical computing model
  - Each process has its own exclusive address space. Typical 1 process per processor. The same code is executed by every process.
    - Only supports explicit parallelization
      - Adds complexity to programming
      - Encourages locality of data access
- The program running on one of the core-memory pairs needs to communicate with other SPMDs
  - we need a method to let processes communicate

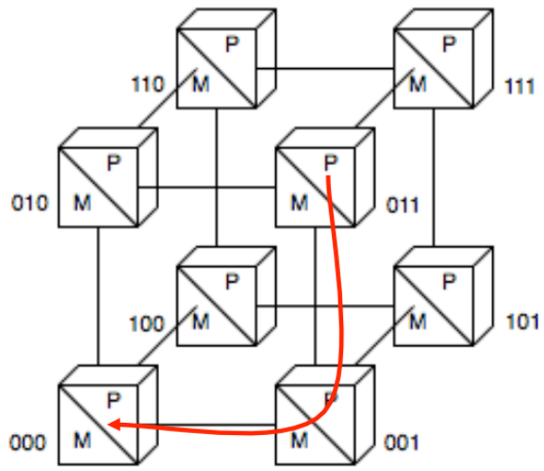
Message passing programming

- Program is a set of named processes
  - Process has thread of control and local memory with local address space
- Processes communicate via explicit data transfers
  - Messages between source and destination, where source and destination are named processors $P_0 ... P_n$ (or compute nodes)
  - Logically shared data is explicitly partitioned over local memories
  - Communication with send/recv via standard message passing libraries, such as MPI and its many “open” variants
  - Each node has a network interface
    - Communication and synchronization via network
    - Message latency and bandwidth is dependent on network topology and routing algorithms

Message passing programming







- Programming model vs. machine models

MPI

- The Message-Passing Interface (MPI) is a standardization of a message-passing library interface specification.
- MPI defines the syntax and semantics of library routines for standard communication patterns
- It's a library, with bindings for C and Fortran. C++ library from Boost. Java versions are also available.
- Many open source and public implementations
  - you need to make no code changes when moving your code between implementations (thanks to the standardized API)
- Supports static (# processes specified before execution - MPI-1) and dynamic (processes created during execution - MPI-2) process creation

MPI

- The Message-Passing Interface (MPI) is a standardization of a message-passing library interface specification.
- MPI defines the syntax and semantics of library routines for standard communication patterns
- It's a library, with bindings for C and Fortran. C++ library from Boost. Java versions are also available. Open MPI has Java bindings.
- Many open source and public implementations
  - you need to make no code changes when moving your code between implementations (thanks to the standardized API)
- Supports static (# processes specified before execution - MPI-1) and dynamic (processes created during execution - MPI-2) process creation

Open source implementations

- Two important open source implementations are:
  - MPICH - http://www.mpich.org
  - Open MPI - https://www.open-mpi.org/
- They both support the latest MPI standard: MPI-3.1

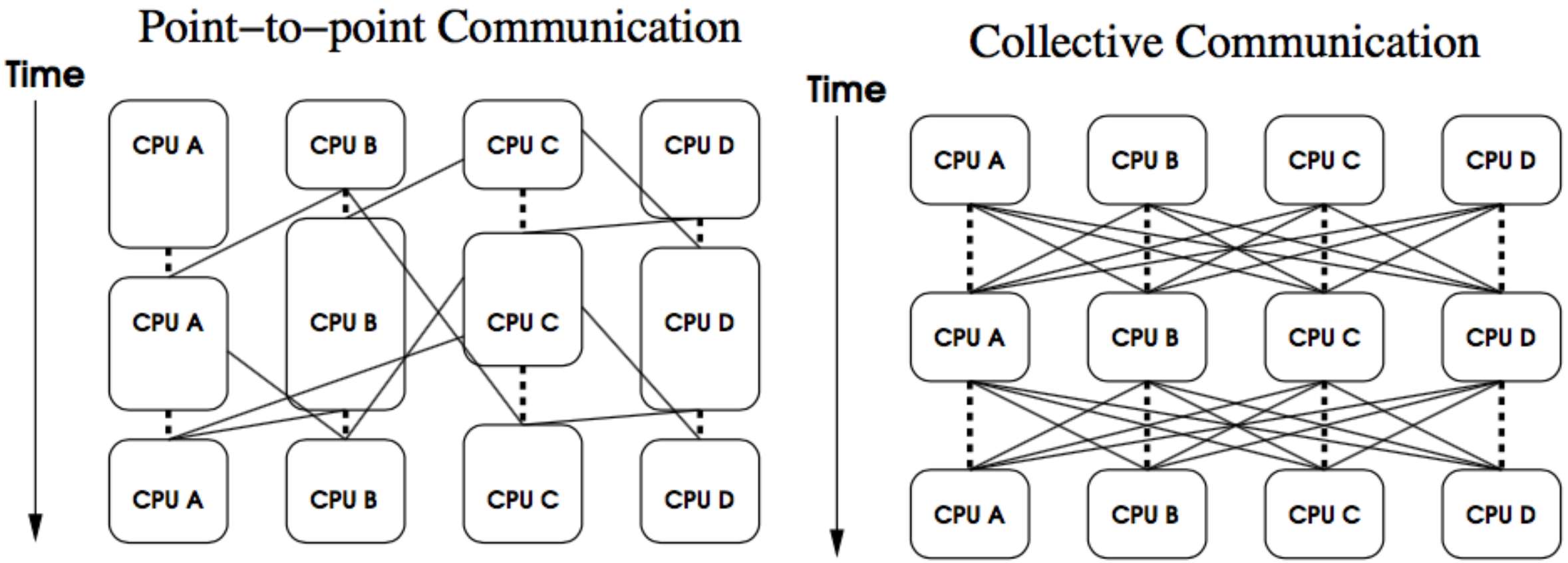
MPI: how does it work

- The same program is launched for execution independently on a collection of cores
- Each core executes the program. To support portability, MPI programs should be written for an arbitrary number of processes. The actual number of processes used for a specific program execution is set when starting the program.
- What differentiates processes is their **rank**: processes with different ranks do different things (“branching based on the process rank”)
  - Process 0 is often treated as “master”
- Communications between process may be “point-to-point” (pairwise), where only two communicating processes are involved, or they may be “collective”, where all of the processes are involved.

Explicit parallelism

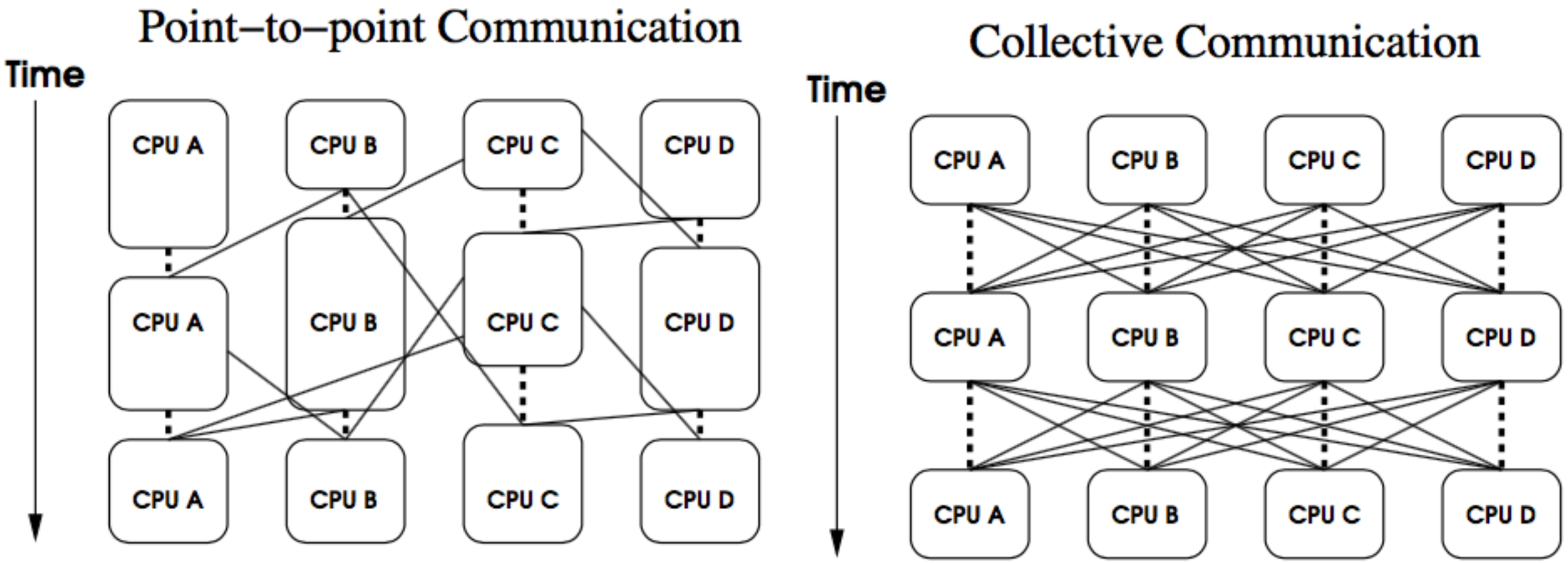
- **All parallelism is explicit**: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.

MPI communications



- Communicators let you specify groups of processes that can intercommunicate
  - Default is `MPI_COMM_WORLD`. Can create groups and hierarchies of processes.

MPI communications



For two different communicators, the same process can have two different ranks: so the meaning of a “rank” is only defined when you specify the communicator

- Communicators let you specify groups of processes that can intercommunicate

- Default is `MPI_COMM_WORLD`. Can create groups and hierarchies of processes.

MPI_Comm_create

- We can create other communicators with:

```
int MPI_Comm_create(MPI_Comm comm,
MPI_Group group, MPI_Comm *newcomm);
```

- Input Parameters
  comm - communicator (handle)
  group - subset of the family of processes making up the comm (handle)

- Output Parameter
  comm_out - new communicator (handle)

Minimal set of MPI routines

- Example of MPI C routines
- `MPI_Init` - Initializes MPI.
- `MPI_Finalize` - Terminates MPI.
- `MPI_Comm_size` - Determines the number of processes.
- `MPI_Comm_rank` - Determines the label of calling process.
- `MPI_Send` - Sends a message.
- `MPI_Recv` - Receives a message.
- `MPI_Probe` - Test for message (returns Status object).

Minimal set of MPI routines

```
#include "mpi.h"
#include <iostream>
int main(int argc, char **argv) {
  int my_rank, n;
  char hostname[128];

  MPI_Init(&argc,&argv); // Has to be called first, and once
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &n);
  gethostname(hostname, 128);
  if (my_rank == 0) { /* master */
    printf("I am the master: %s\n", hostname);
  }
  else { /* worker */
    printf("I am a worker: %s (rank=%d/%d)\n", hostname, my_rank, n-1);
  }
  MPI_Finalize(); // has to be called, and once
  return 0;
}
```

Run the program with the appropriate MPI command, e.g. using 4 instances:

```
mpiexec -np 4 ./openmpi_hello_world

I am a worker: hostname.local (rank=3/3)
I am a worker: hostname.local (rank=1/3)
I am a worker: hostname.local (rank=2/3)
I am the master: hostname.local
```

```
mpiexec –host h1,h2,h3,h4 –np 16 ./test
```

- Runs the first four processes on h1, the next four on h2, etc.

```
mpiexec -hostfile nodeconfig -npernode 16 ~/test
```

- Runs 16 processes per node, using the nodes listed in **nodeconfig** file

```
#inclu
#inclu
int ma
  int
  char

  MPI_
  MPI_
  MPI_
  geth
  if (my_rank == 0) { /* master */
    printf("I am the master: %s\n", hostname);
  }
  else { /* worker */
    printf("I am a worker: %s (rank=%d/%d)\n", hostname, my_rank, n-1);
  }
  MPI_Finalize(); // has to be called, and once
  return 0;
}
```

Run the program with the appropriate MPI command, e.g. using 4 instances:

```
mpiexec -np 4 ./openmpi_hello_world

I am a worker: hostname.local (rank=3/3)
I am a worker: hostname.local (rank=1/3)
I am a worker: hostname.local (rank=2/3)
I am the master: hostname.local
```

```
mpiexec –host h1,h2,h3,h4 –np 16 ./test
```

- Runs the first four processes on h1, the next four on h2, etc.

```
mpiexec -hostfile nodeconfig -npernode 16 ~/test
```

- Runs 16 processes per node, using the nodes listed in `nodeconfig` file

- In general, starting an MPI program is dependent on the implementation of MPI you are using, and might require various scripts, program arguments, and/or environment variables.
- `mpiexec <args>` is part of MPI-2, as a recommendation, but not a requirement, for implementors.
- Compilation is often carried on using a driver like `mpicc` that manages all the `-L`, `-l`, and `-I` parameters

Types of MPI operations

- **Blocking operation**: an MPI communication operation is blocking, if return of control to the calling process indicates that all resources, such as buffers, specified in the call can be reused, e.g., for other operations. In particular, all state transitions initiated by a blocking operation are completed before control returns to the calling process.

- **Non-blocking operation**: an MPI communication operation is non-blocking, if the corresponding call may return before all effects of the operation are completed and before the resources used by the call can be reused. Thus, a call of a non-blocking operation only starts the operation. The operation itself is completed not before all state transitions caused are completed and the resources specified can be reused.

The terms *blocking* and *non-blocking* describe the behavior of operations from the local view of the executing process, without taking the effects on other processes into account.

- **Blocking operation**: an MPI communication operation is blocking, if return of control to the calling process indicates that all resources, such as buffers, specified in the call can be reused, e.g., for other operations. In particular, all state transitions initiated by a blocking operation are completed before control returns to the calling process.

- **Non-blocking operation**: an MPI communication operation is non-blocking, if the corresponding call may return before all effects of the operation are completed and before the resources used by the call can be reused. Thus, a call of a non-blocking operation only starts the operation. The operation itself is completed not before all state transitions caused are completed and the resources specified can be reused.

Types of MPI operations

- **Synchronous communication**: the communication operation between sender and receiver does not complete before both processes have started their communication operation. The completion of a synchronous send indicates not only that the send buffer can be reused, but also that the receiving process has started the execution of the corresponding receive operation.

- **Asynchronous communication**: using asynchronous communication, the sender can execute its communication operation without any coordination with the receiving process.

These types of operations consider the effect of communication operations from a global viewpoint.

- **Synchronous communication**: the communication operation between sender and receiver does not complete before both processes have started their communication operation. The completion of a synchronous send indicates not only that the send buffer can be reused, but also that the receiving process has started the execution of the corresponding receive operation.

- **Asynchronous communication**: using asynchronous communication, the sender can execute its communication operation without any coordination with the receiving process.

Data communication

- Data communication in MPI is like email exchange
  - One process sends a copy of the data to another process (or a group of processes), and the other process receives it
- Communication requires the following information:
  - Sender has to know:
    - Whom to send the data to (receiver's process rank)
    - What kind of data to send (100 integers or 200 characters, etc)
    - A user-defined "tag" for the message (think of it as an email subject; allows the receiver to understand what type of data is being received)
  - Receiver "might" have to know:
    - Who is sending the data (OK if the receiver does not know; in this case anyone can send)
    - What kind of data is being received (partial information is OK: I might receive up to 1000 integers)
    - What the user-defined "tag" of the message is (OK if the receiver does not know)

Tags: why ?

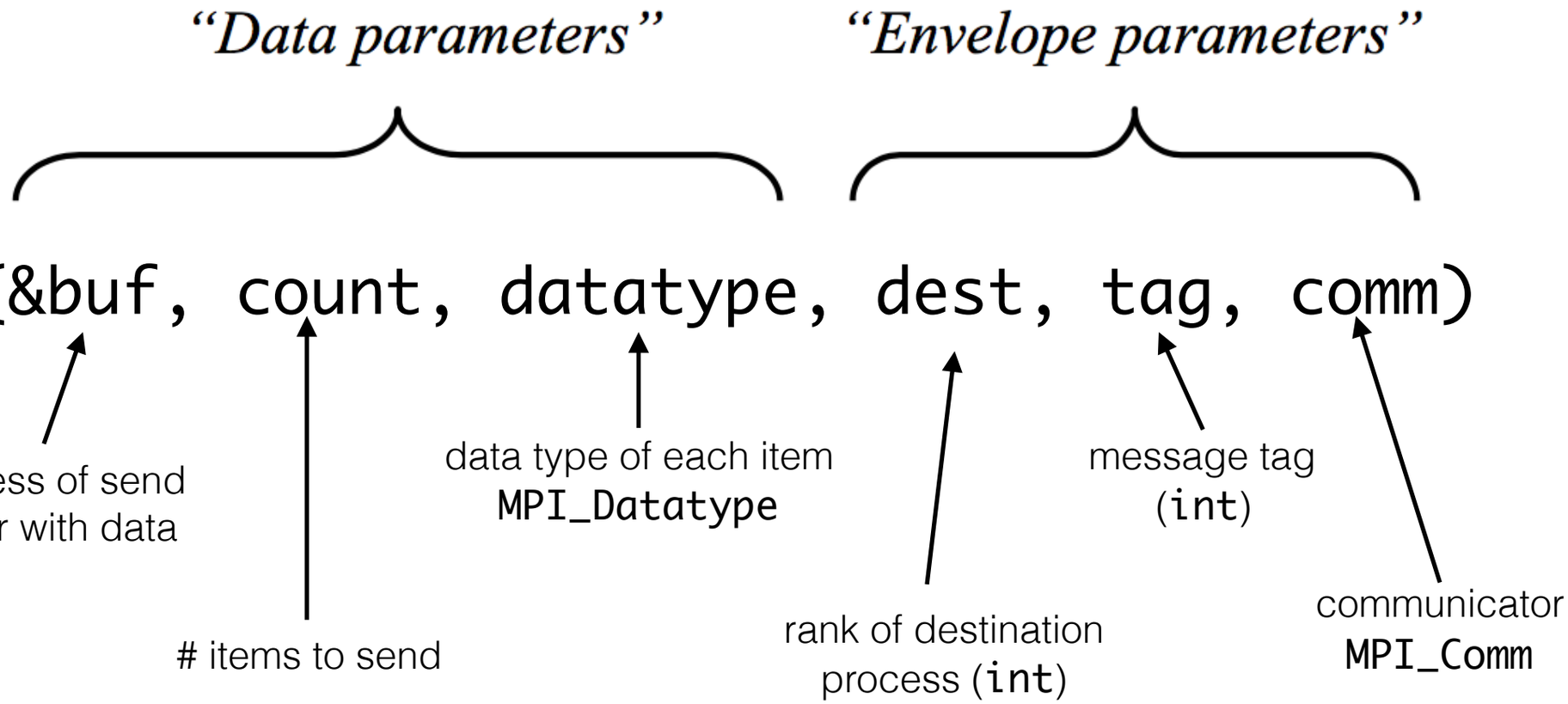
- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message.

- For example, if an application is expecting two types of messages from a peer, tags can help distinguish these two types.

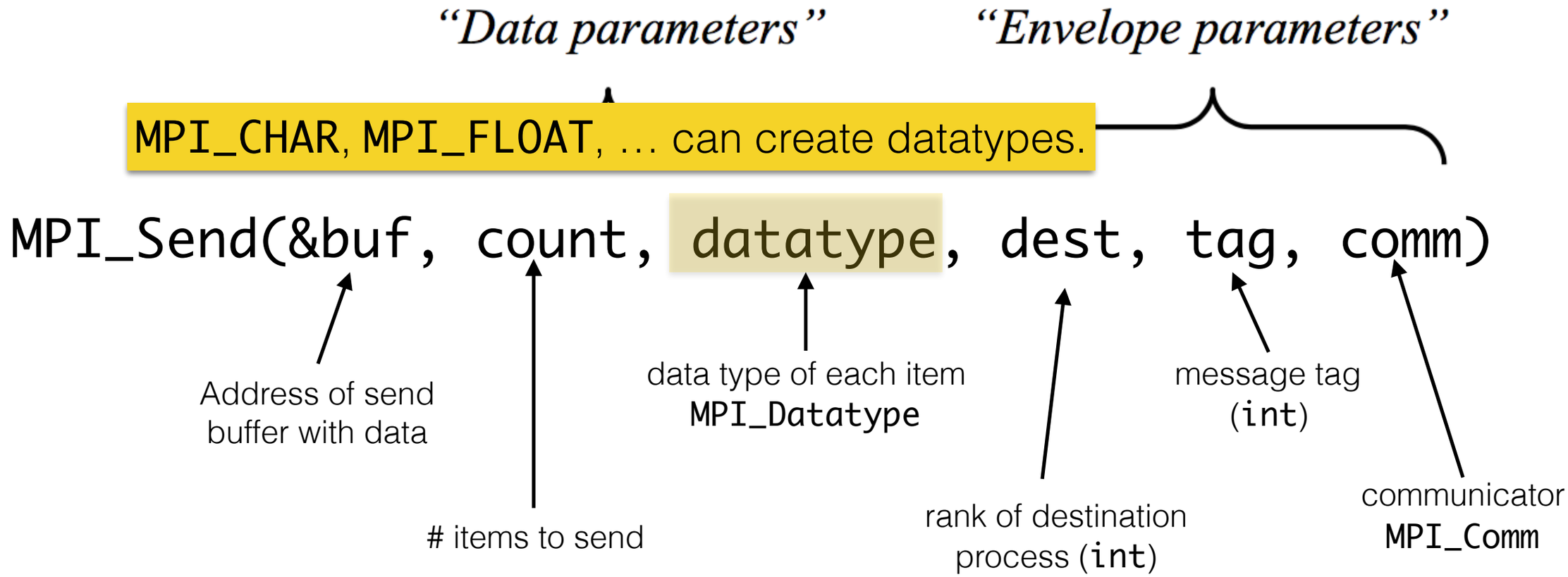
- Messages can be screened at the receiving end by specifying a specific tag.

- `MPI_ANY_TAG` is a special “wild-card” tag that can be used by the receiver to match any tag.

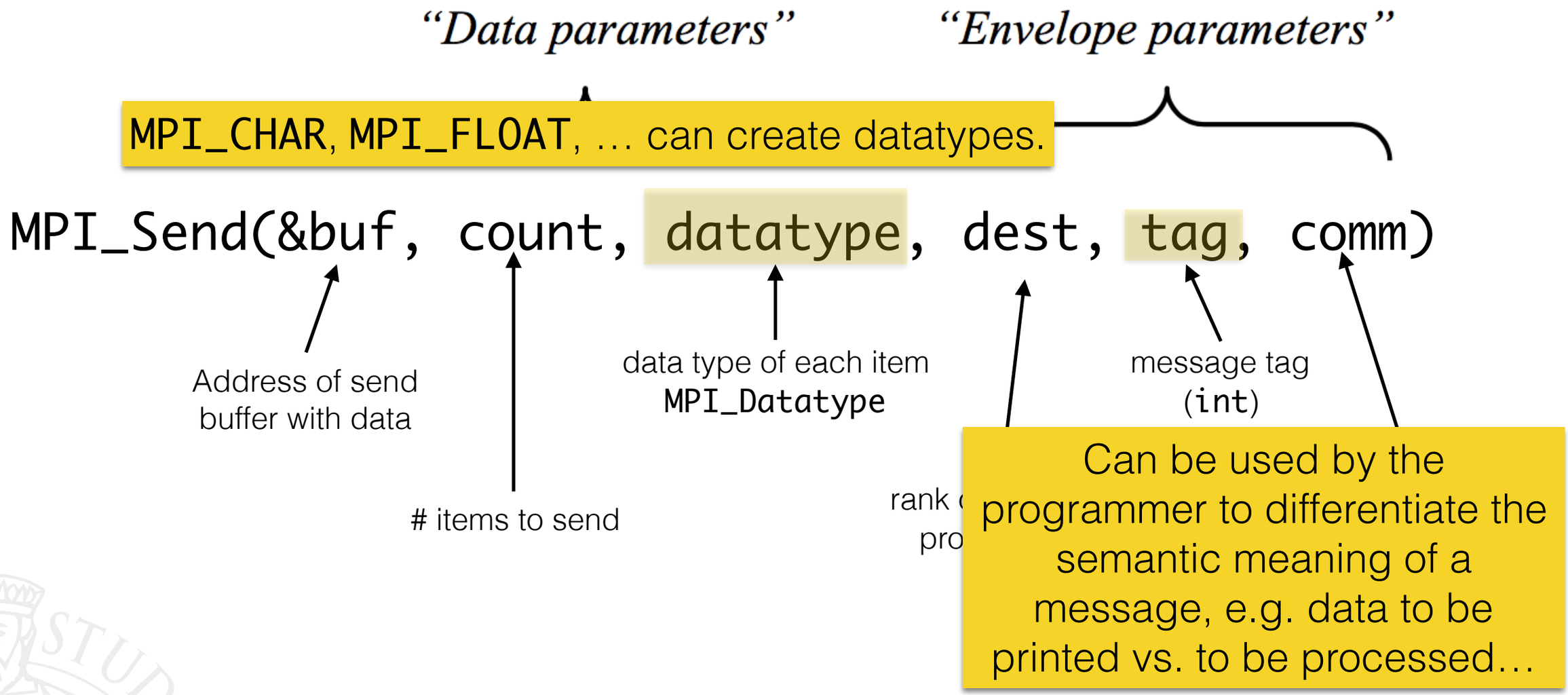
MPI Point-to-point Send Format



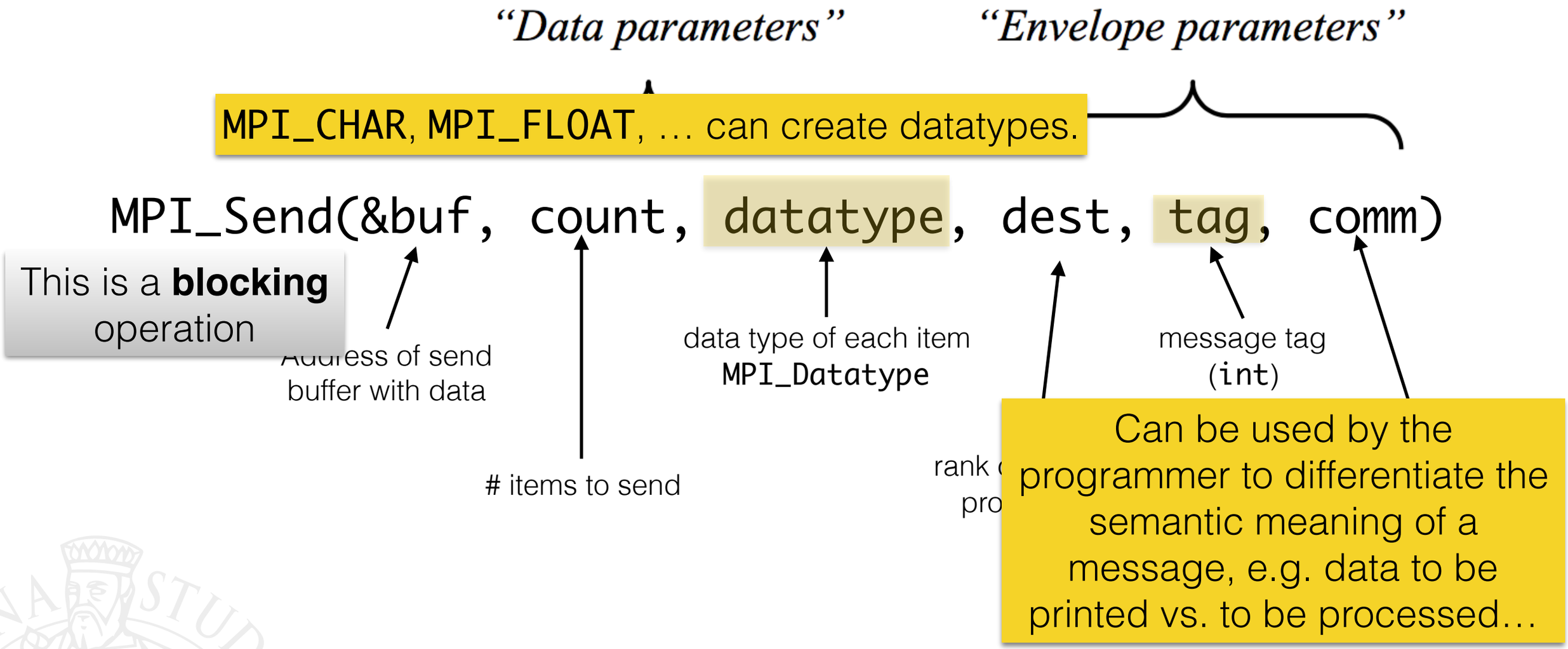
MPI Point-to-point Send Format



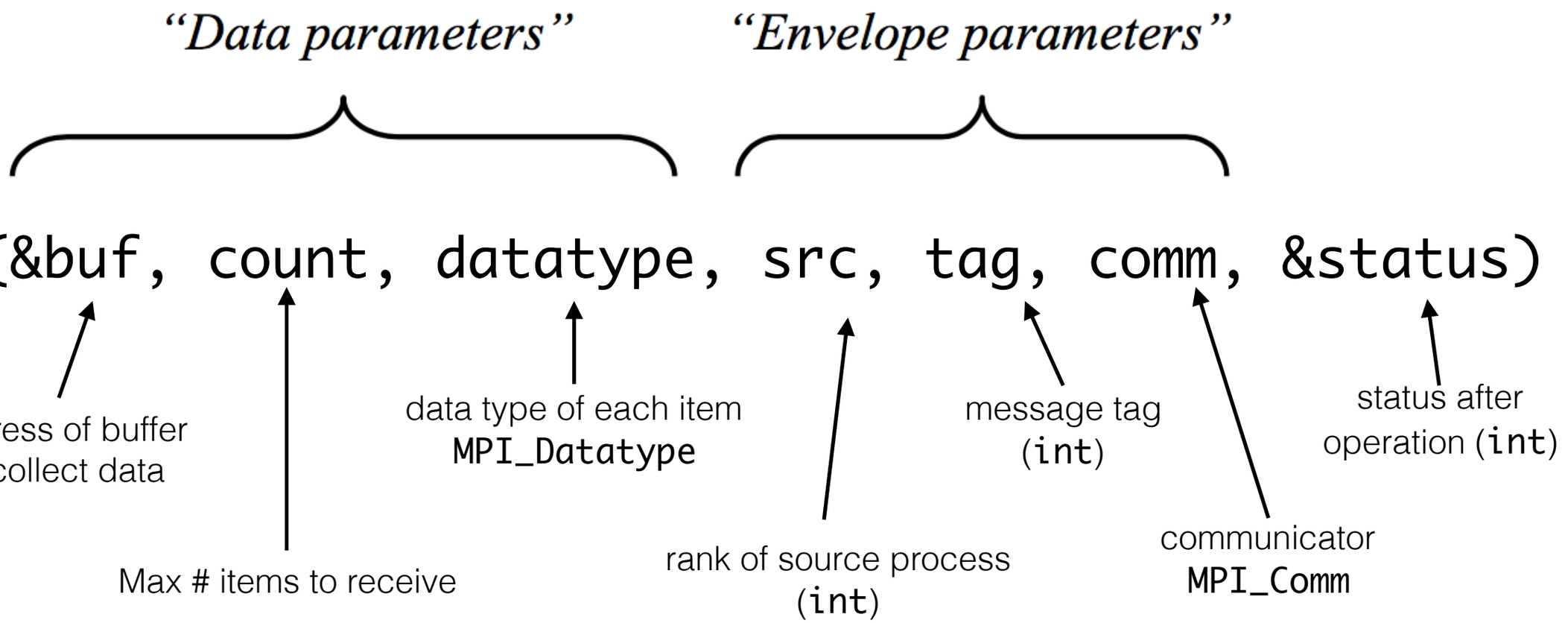
MPI Point-to-point Send Format



MPI Point-to-point Send Format



MPI Point-to-point Recv Format



MPI Point-to-point Recv Format

"Data parameters" *"Envelope parameters"*

a special constant `MPI_ANY_SOURCE` means that any source process can send

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

- `&buf` — Address of buffer to collect data
- `count` — Max # items to receive
- `datatype` — data type of each item `MPI_Datatype`
- `src` — rank of source process (`int`)
- `tag` — message tag (`int`)
- `comm` — communicator `MPI_Comm`
- `&status` — status after operation (`int`)

MPI Point-to-point Recv Format

"Data parameters" *"Envelope parameters"*

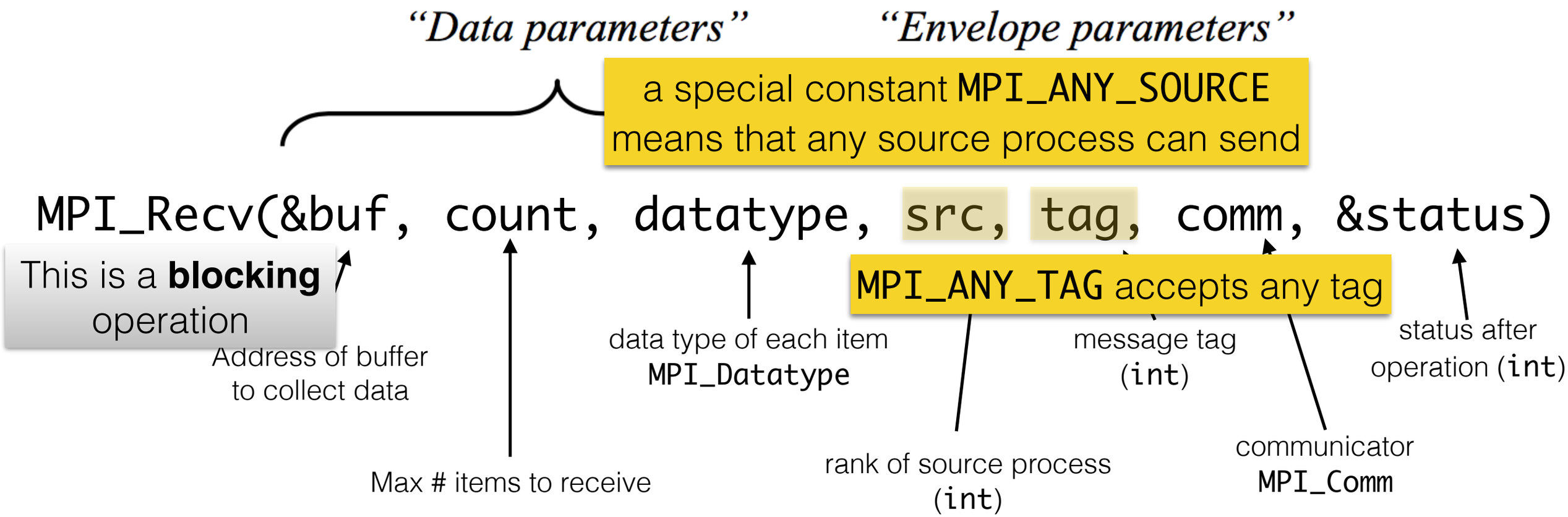
a special constant `MPI_ANY_SOURCE` means that any source process can send

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

`MPI_ANY_TAG` accepts any tag

- Address of buffer to collect data
- Max # items to receive
- data type of each item `MPI_Datatype`
- rank of source process (`int`)
- message tag (`int`)
- communicator `MPI_Comm`
- status after operation (`int`)

MPI Point-to-point Recv Format



MPI Point-to-point Recv Format

"Data parameters" *"Envelope parameters"*

a special constant `MPI_ANY_SOURCE` means that any source process can send

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

`MPI_ANY_TAG` accepts any tag

This is a **blocking** operation

Address of buffer to collect data

Max # items to receive

data type of each item `MPI_Datatype`

rank of source process (`int`)

message tag (`int`)

communicator `MPI_Comm`

status after operation (`int`)

```
// waiting for all the messages from comm_sz sources in no specific order
for (i = 1; i < comm_sz; i++) {
  MPI_Recv(result, result_sz, result_type, MPI_ANY_SOURCE,
                result_tag, comm, MPI_STATUS_IGNORE);
  Process_result(result);
}
```

Send/receive example

```
int source; /* rank of sender */
int dest; /* rank of receiver */
int tag = 0; /* tag for messages */
char message[100]; /* storage for message */
MPI_Status status; /* return status for receive */


MPI_Init(&argc, &argv); // start MPI library
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // Find out process rank
MPI_Comm_size(MPI_COMM_WORLD, &p); // Find out number of processes

if (my_rank != 0) {
    /* Create message */
    sprintf(message, "Greetings from process %d!", my_rank);
    dest = 0;
    /* Use strlen+1 so that '\0' gets transmitted */
    MPI_Send(message, strlen(message) + 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
}
else { /* my_rank == 0 */
    for (source = 1; source < p; source++) {
        MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
        printf("Master program received msg: %s\n", message);
    }
}

MPI_Finalize(); // Shut down MPI
```

Send/receive example

```
int source; /* rank of sender */
int dest; /* rank of receiver */
int tag = 0; /* t
char message[100]
MPI_Status status
```

tells the MPI system to do all of the necessary setup, like allocating buffers for communications, assign ranks to processes, etc. No other MPI functions should be called before.

```
MPI_Init(&argc, &argv); // start MPI library
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // Find out process rank
MPI_Comm_size(MPI_COMM_WORLD, &p); // Find out number of processes

if (my_rank != 0) {
    /* Create message */
    sprintf(message, "Greetings from process %d!", my_rank);
    dest = 0;
    /* Use strlen+1 so that '\0' gets transmitted */
    MPI_Send(message, strlen(message) + 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
}
else { /* my_rank == 0 */
    for (source = 1; source < p; source++) {
        MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
        printf("Master program received msg: %s\n", message);
    }
}

MPI_Finalize(); // Shut down MPI
```

Send/receive example

```
int source; /* rank of sender */
int dest; /* rank of receiver */
int tag = 0; /* t
char message[100]
MPI_Status status
```

tells the MPI system to do all of the necessary setup, like allocating buffers for communications, assign ranks to processes, etc. No other MPI functions should be called before.

```
MPI_Init(&argc, &argv); // start MPI library
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // Find out process rank
MPI_Comm_size(MPI_COMM_WORLD, &p); // Find out number of processes

if (my_rank != 0) {
    /* Create message */
    sprintf(message, "Greetings from process %d!", my_rank);
    dest = 0;
    /* Use strlen+1 so that '\0' gets transmitted */
    MPI_Send(message, strlen(message) + 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
}
else { /* my_rank == 0 */
    for (source = 1; source < p; source++) {
        MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
        printf("Master program received msg: %s\n", message);
    }
}

MPI_Finalize(); // Shut down MPI
```

Frees resources. No MPI functions should be called after.

Send/receive example

```
int source; /* rank of sender */
int dest; /* rank of receiver */
int tag = 0; /* t
char message[100]
MPI_Status status
```

tells the MPI system to do all of the necessary setup, like allocating buffers for communications, assign ranks to processes, etc. No other MPI functions should be called before.

```
MPI_Init(&argc, &argv); // start MPI library
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // Find out process rank
MPI_Comm_size(MPI_COMM_WORLD, &p); // Find out number of processes
```

```
if (my_rank != 0) {
    /* Create message */
    MPI_Send(message, strlen(message) + 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
}
else { /* my_rank == 0 */
    for (source = 1; source < p; source++) {
        MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
        printf("Master program received msg: %s\n", message);
    }
}
```

If the type of the message is the same and the size of the receiving buffer is >= of the sending buffer then the message can be successfully received

Frees resources. No MPI functions should be called after.

```
MPI_Finalize(); // Shut down MPI
```

Message status

- A receiver can receive a message without knowing:
  - the amount of data in the message
  - the sender of the message (is using `MPI_ANY_SOURCE`)
  - the tag of the message (if using `MPI_ANY_TAG`).

- Using a `MPI_Status status_variable` to get the status of a received message is possible to obtain `status_variable.MPI_SOURCE` and `status_variable.MPI_TAG`.

- To get message size use: `MPI_Get_count(&status_variable, recv_type, &count)` to store in the `int count` variable the size of the message.

- `status_variable.MPI_ERROR` contains an error code.

Communication

- Internally a message transfer in MPI is usually performed in three steps:

1. The message is assembled by adding a header with information on the sending process, the receiving process, the tag, and the communicator used.

2. The message is sent via the network from the sending process to the receiving process.

3. At the receiving side, the data entries of the message are copied from the system buffer into the receive buffer specified by `MPI_Recv()`.

Communication

- Both `MPI_Send()` and `MPI_Recv()` are blocking, asynchronous operations.
- This means that an `MPI_Recv()` operation can also be started when the corresponding `MPI_Send()` operation has not yet been started. The process executing the `MPI_Recv()` operation is blocked until the specified receive buffer contains the data elements sent.
- Regarding sending it may depends on the size of the message and the MPI implementation:
  - If the message is sent directly from the send buffer specified without using an internal system buffer, then the `MPI_Send()` operation is blocked until the entire message has been copied into a receive buffer at the receiving side.
  - If the message is first copied into an internal system buffer of the runtime system, the sender can continue its operations as soon as the copy operation into the system buffer is completed.
- When using `MPI_Send`, when the function returns, we don't actually know whether the message has been transmitted. We only know that the storage we used for the message, the send buffer, is available for reuse by our program.

Communication

- Both `MPI_Send()` and `MPI_Recv()` are blocking, asynchronous operations.

- This means that an `MPI_Recv()` operation can also be started when the corresponding `MPI_Send()` operation has not yet been started. The process executing the `MPI_Recv()` operation is blocked until the specified receive buffer contains the data elements sent.

- Regarding sending it may depends on the size of the message and the MPI implementation:

  - If the message is sent directly from the send buffer specified without using an internal system buffer, then the `MPI_Send()` operation is blocked until the entire message has been copied into a receive buffer at the receiving side.

  - If the message is first copied into an internal system buffer of the runtime system, the sender can continue its operations as soon as the copy operation into the system buffer is completed.

Many implementations of MPI set a threshold at which the system switches from buffering to blocking. That is, messages that are relatively small will be buffered by `MPI_Send`, but for larger messages, it will block.

Delivery order

- An important property to be fulfilled by any MPI library is that messages are delivered in the order in which they have been sent (**nonovertaking**).

- If a sender sends two messages one after another to the same receiver and both messages fit to the first `MPI_Recv()` called by the receiver, the MPI runtime system ensures that the first message sent will always be received first.

- However, **there is no restriction** on the arrival of messages sent from different processes.

Delivery order

- An important property to be fulfilled by any MPI library is that messages are delivered in the order in which they have been sent (**nonovertaking**).

- If a sender sends two messages one after another to the same receiver and both messages fit to the first `MPI_Recv()` called by the receiver, the MPI runtime system ensures that the first message sent will always be received first.

- However, **there is no restriction** on the arrival of messages sent from different processes.

This is essentially because MPI can't impose performance on a network…

Problems

- Note that the semantics of `MPI_Recv` suggests a potential pitfall in MPI programming: if a process tries to receive a message and there's no matching send, then the process will block forever.
- Similarly, if a call to `MPI_Send` blocks and there's no matching receive, then the sending process can hang.
- We need to be very careful when we're coding that there are no mistakes in our calls to `MPI_Send` and `MPI_Recv`. For example, if the tags don't match, or if the rank of the destination process is the same as the rank of the source process, the receive won't match the send, and either a process will hang, or the receive may match another send.

Deadlock

```
MPI_Comm_rank (comm, &my_rank);

if (my_rank == 0) {

  MPI_Recv(recvbuf, count, MPI_INT, 1, tag, comm, &status);

  MPI_Send(sendbuf, count, MPI_INT, 1, tag, comm);

} else if (my_rank == 1) {

  MPI_Recv(recvbuf, count, MPI_INT, 0, tag, comm, &status);

  MPI_Send(sendbuf, count, MPI_INT, 0, tag, comm);

}
```

Deadlock

```
MPI_Comm_rank (comm, &my_rank);

if (my_rank == 0) {

  MPI_Recv(recvbuf, count, MPI_INT, 1, tag, comm, &status);

  MPI_Send(sendbuf, count, MPI_INT, 1, tag, comm);

} else if (my_rank == 1) {

  MPI_Recv(recvbuf, count, MPI_INT, 0, tag, comm, &status);

  MPI_Send(sendbuf, count, MPI_INT, 0, tag, comm);

}
```

Both processes 0 and 1 execute an `MPI_Recv()` operation before an `MPI_Send()` operation. This leads to a deadlock because of mutual waiting

Deadlock

```
/* program fragment that does not cause a deadlock */

MPI_Comm_rank (comm, &my_rank);

if (my_rank == 0) {

  MPI_Send(sendbuf, count, MPI_INT, 1, tag, comm);
  MPI_Recv(recvbuf, count, MPI_INT, 1, tag, comm, &status);

} else if (my_rank == 1) {

  MPI_Recv(recvbuf, count, MPI_INT, 0, tag, comm, &status);
  MPI_Send(sendbuf, count, MPI_INT, 0, tag, comm);

}
```

Secure program

- An MPI program is called secure if the correctness of the program does not depend on assumptions about specific properties of the MPI runtime system, like the existence of system buffers or the size of system buffers.

- If more than two processes exchange messages such that each process sends and receives a message, the program must exactly specify in which order the send and receive operations are to be executed to avoid deadlocks.

- Using `MPI_Sendrecv()`, the programmer does not need to worry about the order of the send and receive operations. This function carries out a blocking send and a receive in a single call.
  - the function uses two disjoint, non-overlapping buffers to send and receive messages.
  - What makes it especially useful is that the MPI implementation schedules the communications so that the program won't hang or crash.
  - If it happens that the send and the receive buffers should be the same, MPI provides the alternative int `MPI_Sendrecv_replace`.

Non-blocking operations

- The use of blocking communication operations can lead to waiting times in which the blocked process does not perform useful work.

- A non-blocking send operation initiates the sending of a message and returns control to the sending process as soon as possible. Upon return, the send operation has been started, but the send buffer specified cannot be reused safely, i.e., the transfer into an internal system buffer may still be in progress. A separate completion operation is provided to test whether the send operation has been completed locally.

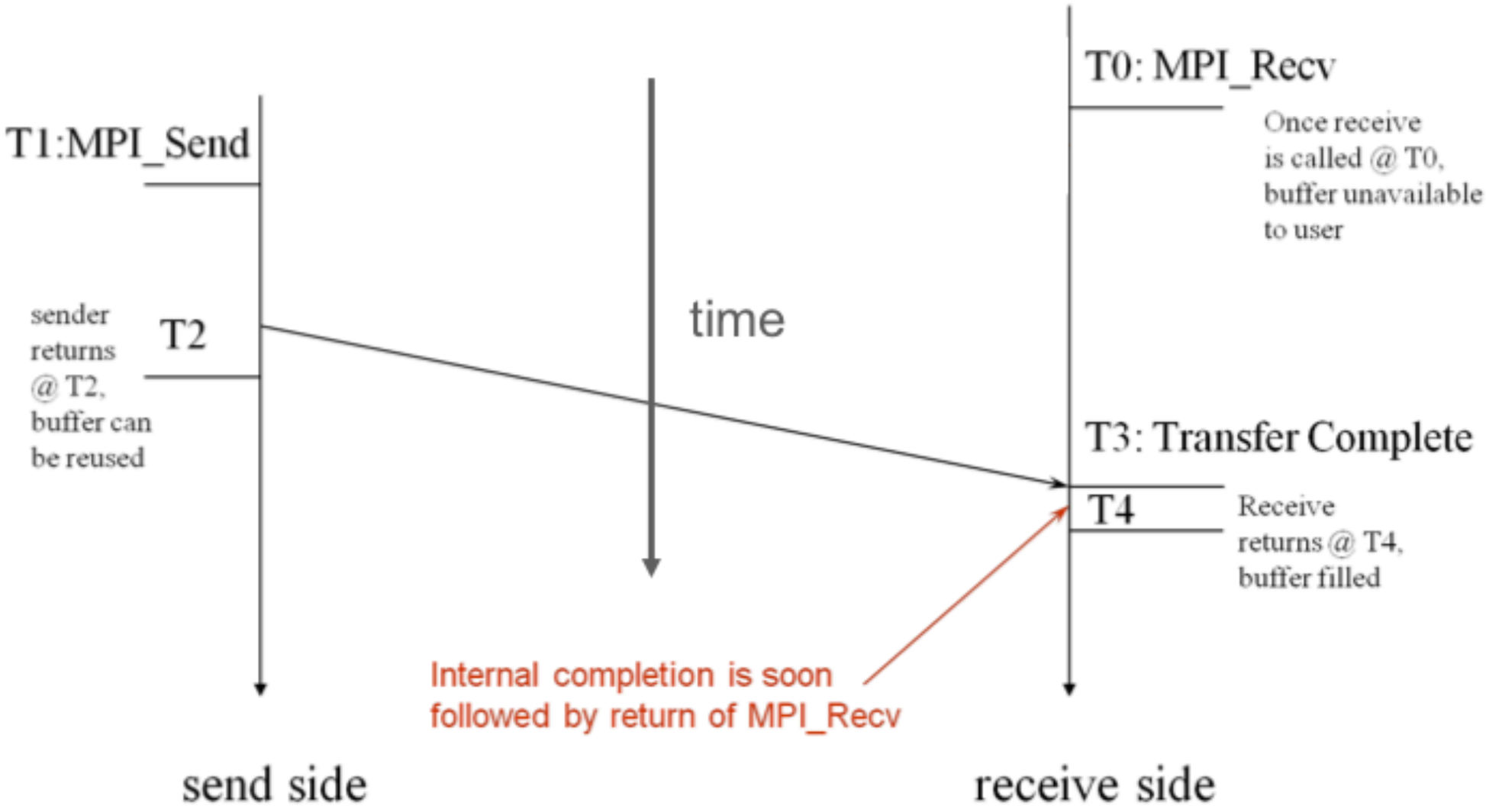
- `MPI_Isend` has the same interface, adding an opaque structure `MPI_Request *request` that can be used for the identification of a specific communication operation.

- Similarly `MPI_Irecv`, initiates the receiving of a message and returns control to the receiving process as soon as possible. Upon return, the receive operation has been started and the runtime system has been informed that the receive buffer specified is ready to receive data. But the return of the call does not indicate that the receive buffer already contains the data

Blocking vs. Non-blocking Communication

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

Blocking Send-Receive Diagram



Completion and waiting

- `int MPI_Test (MPI Request *request, int *flag, MPI_Status *status)`

- test for the completion of a non-blocking communication operation.

- The call returns `flag = 1` (true), if the communication operation identified by request has been completed. Otherwise, `flag = 0`. The parameter `status` contains information on the message received, as seen before.

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

- can be used to wait for the completion of a non-blocking communication operation. When calling this function, the calling process is blocked until the operation identified by request has been completed.

Multiple Completions

- It is sometimes desirable to wait on multiple requests:

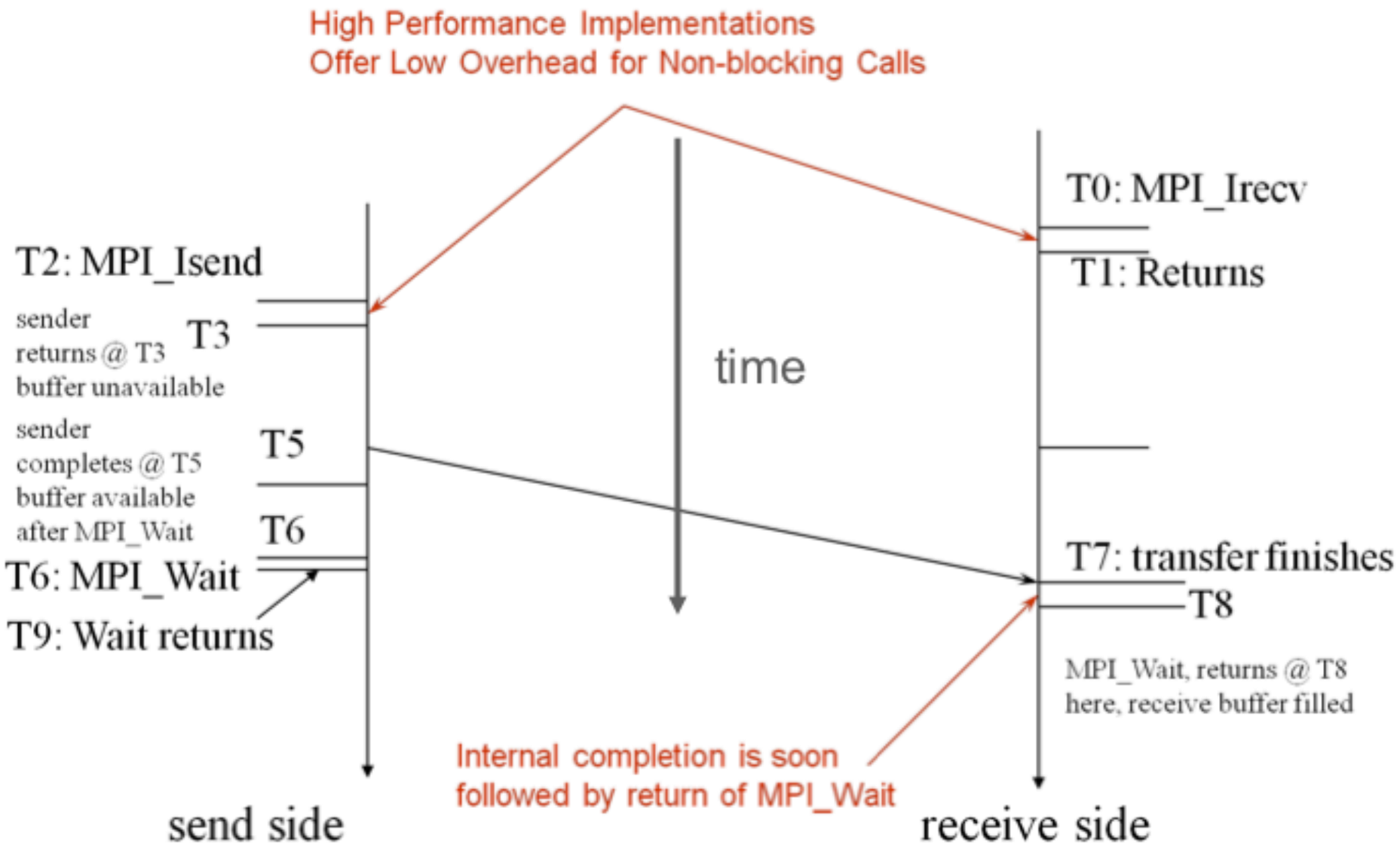
  - `MPI_Waitall(count, array_of_requests, array_of_statuses)`

  - `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 test for each of these

Non-Blocking Send-Receive Diagram



Example: non-Blocking communication example

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

Probe to receive

- Check for existence of data to receive, without actually receiving them
- The user can then decide how to receive them, based on the information returned by the probe (basically, the information returned by status)
- In particular, the user may allocate memory for the receive buffer, according to the length of the probed message
- Blocking Probe, wait till match

```
int MPI_Probe(int source, int tag, MPI_Comm comm,
MPI_Status *status);
```

- Non Blocking Probe, flag true if ready

```
int MPI_Iprobe(int source,int tag,MPI_Comm comm,int
*flag,MPI_Status *status);
```

Synchronous mode

- In the standard mode, a send operation can be completed even if the corresponding receive operation has not yet been started (if system buffers are used).

- In contrast, in **synchronous mode**, a send operation will be completed not before the corresponding receive operation has been started and the receiving process has started to receive the data sent: `MPI_Ssend()` (blocking) and `MPI_Issend()` (non blocking)

- The execution of a send and receive operation in synchronous mode leads to a form of synchronization between the sending and the receiving processes:

  - the return of a send operation in synchronous mode indicates that the receiver has started to store the message in its local receive buffer.

Buffered mode

- In buffered mode, the local execution and termination of a send operation is not influenced by non-local events as is the case for the synchronous mode and can be the case for standard mode if no or too small system buffers are used.
  - Control is returned to the calling process even if the corresponding receive operation has not yet been started.
  - If the corresponding receive operation has not yet been started, the runtime system must buffer the outgoing message.
  - The send buffer (provided by user to runtime system) can be reused immediately after control returns, even if a non-blocking send is used.
- Blocking: `MPI_Bsend()` - same parameters as `MPI_Send()` with the same meaning. Non-blocking send operation in buffered mode is performed by calling `MPI_Ibsend()`, which has the same parameters as `MPI_Isend()`

Buffered mode

- I
  - e
  - too
  - ing

Before sending attach the user provided buffer with
`MPI_Buffer_attach(void *buf, int size)`

Message may remain in the buffer until a matching receive is posted.
`MPI_Buffer_Detach()` will block until all messages are received

  - If the corresponding receive operation has not yet been started, the runtime system must buffer the outgoing message.
  - The send buffer (provided by user to runtime system) can be reused immediately after control returns, even if a non-blocking send is used.
- Blocking: `MPI_Bsend()` - same parameters as `MPI_Send()` with the same meaning. Non-blocking send operation in buffered mode is performed by calling `MPI_Ibsend()`, which has the same parameters as `MPI_Isend()`

Point-to-point communication modes

Communication mode	Advantages	Disadvantages
Synchronous	Safest, and therefore most portable SEND/RECV order not critical Amount of buffer space irrelevant	Can incur substantial synchronization overhead
Buffered	Decouples SEND from RECV No sync overhead on SEND Order of SEND/RECV irrelevant Programmer can control size of buffer space	Additional system overhead incurred by copy to buffer
Standard	Good for many cases	Either uses an internal buffer or buffered: may lead to unexpected deadlock

Collective Communication

- A communication operation is called collective or global if all or a subset of the processes of a parallel program are involved.
  - All processes in a communicator are involved, and all must make the same call at the same time. For use on a subset, you need to create another communicator.
  - All datatypes and counts must be the same in all of the calls that match (i.e. on all processes)
  - Often simplify distribution of data to workers.

Types of collective communication

- Global Synchronization (barrier synchronization)
- Global Communication (broadcast, scatter, gather, etc.)
- Global Operations (sum, global maximum, etc.)
- All collective operations are blocking

Barrier synchronization

- A node invoking the barrier routine will be blocked until all the nodes within the group (communicator) have invoked it

- `MPI_Barrier(MPI_Comm comm)`

- Almost never required in a parallel program

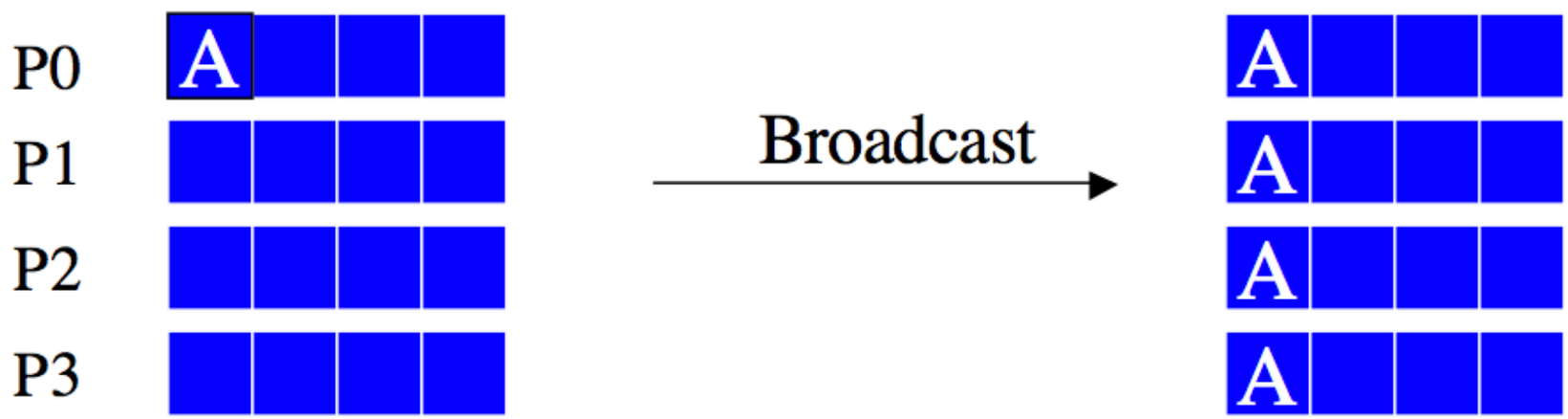
  - Occasionally useful in measuring performance and load balancing. Eliminate once debugging is finished.

Broadcast

- `MPI_Bcast` is called by both the sender (called the root process) and the processes that are to receive the broadcast

  - `MPI_Bcast` is not a “multi-send”

  - “root” argument is the rank of the sender; this tells MPI which process originates the broadcast and which receive

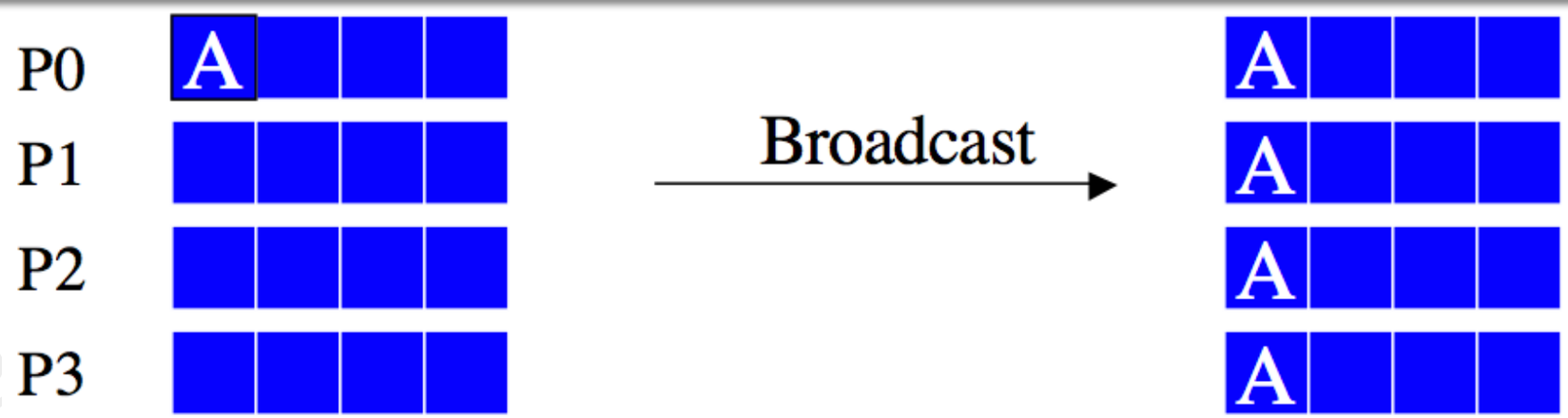


Broadcast

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

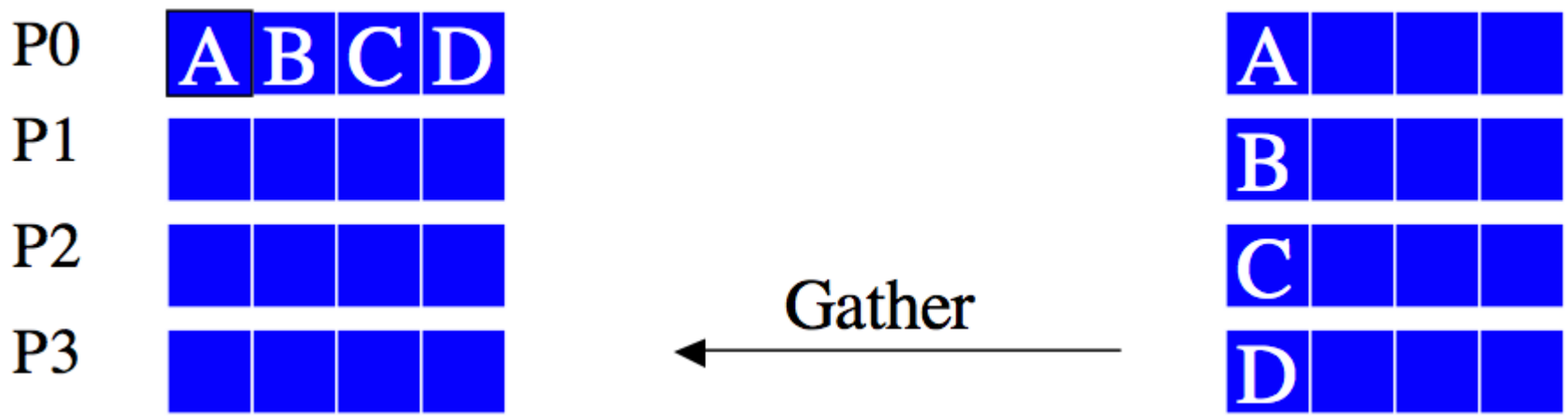
E.g.:

```
float A[N][N], Ap[N/P][N], b[N], c[N], cp[N/P];
//...
root = 0;
MPI_Bcast(b, N, MPI_Float, root, MPI_COMM_WORLD);
```



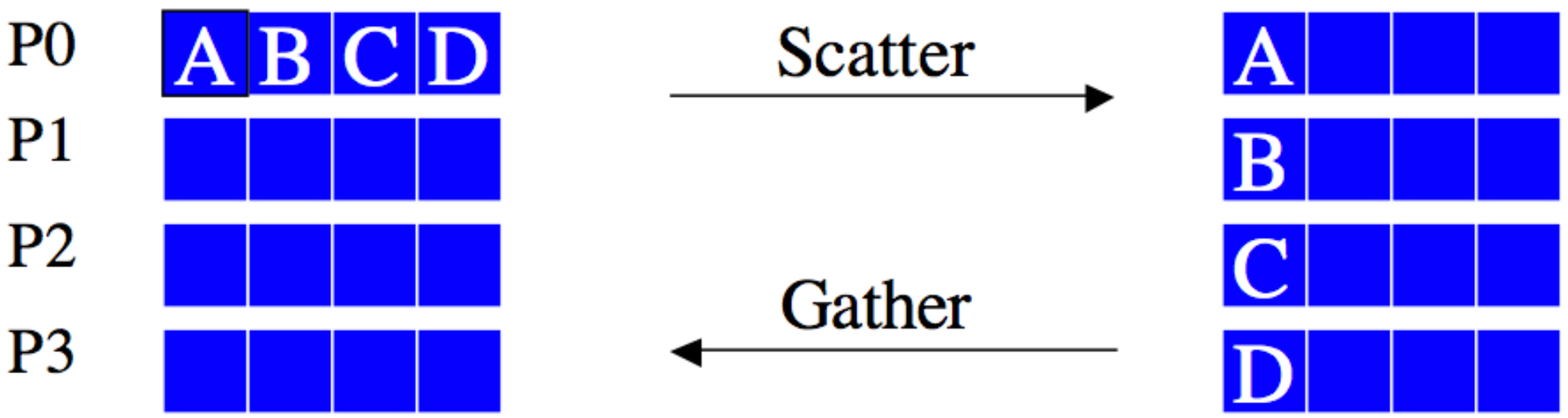
Gather

- Each process sends content of send buffer to the root process
- Root receives and stores in rank order
- `int MPI_Gather (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);`



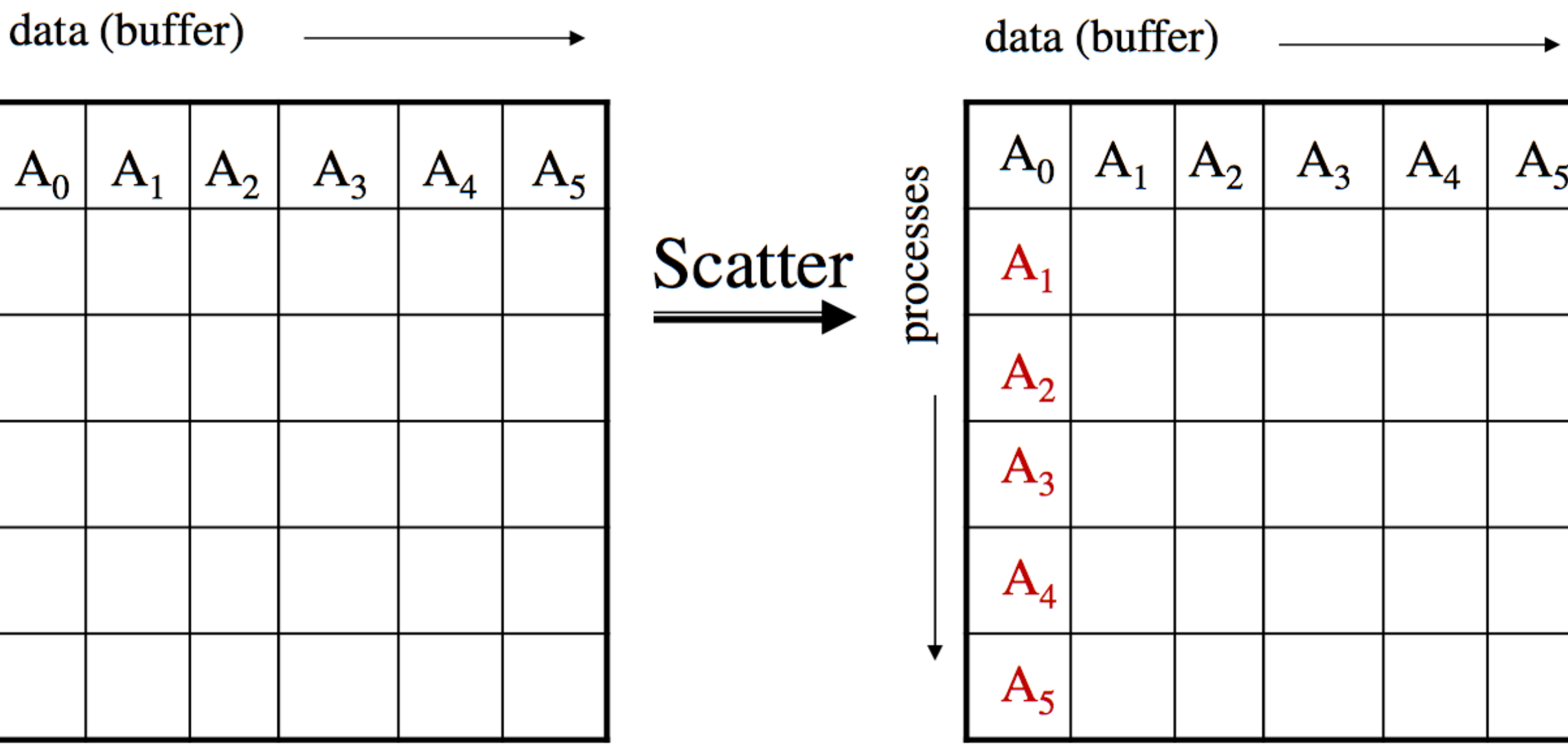
Scatter

- Inverse of MPI_Gather
- Data elements on root listed in rank order – each processor gets corresponding data chunk after call to scatter
- `int MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);`
- All arguments are significant on `root`, while on other processes only `recvbuf`, `recvcount`, `recvtype`, `root`, and `comm` are significant



Scatter

- Inverse of MPI_Gather
- Data elements on root listed in rank order – each processor gets corresponding data chunk after call to scatter



Scatter and gather

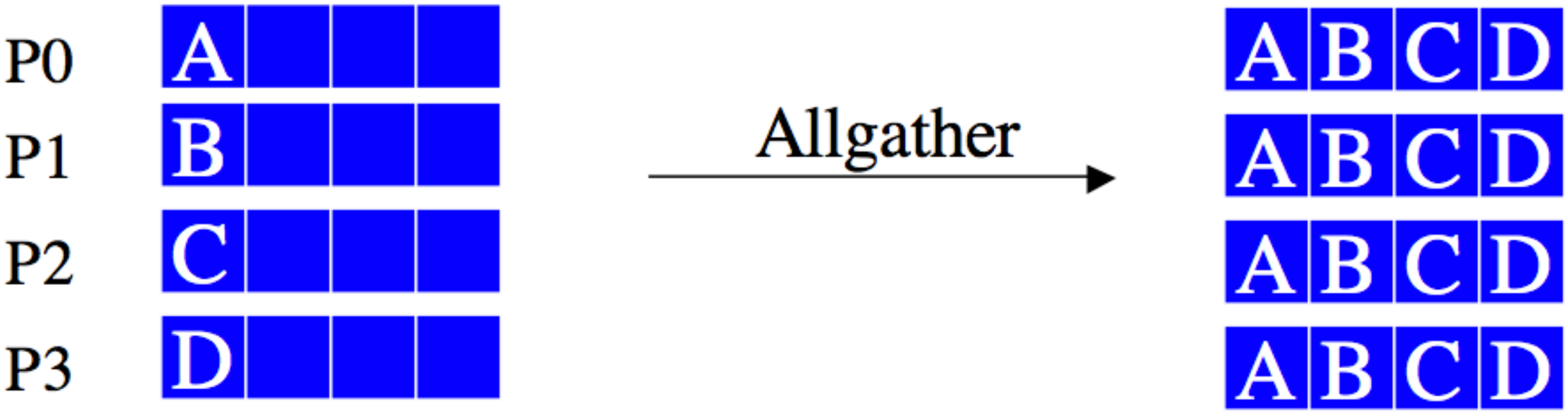
- Gather: you automatically create a serial array from a distributed one

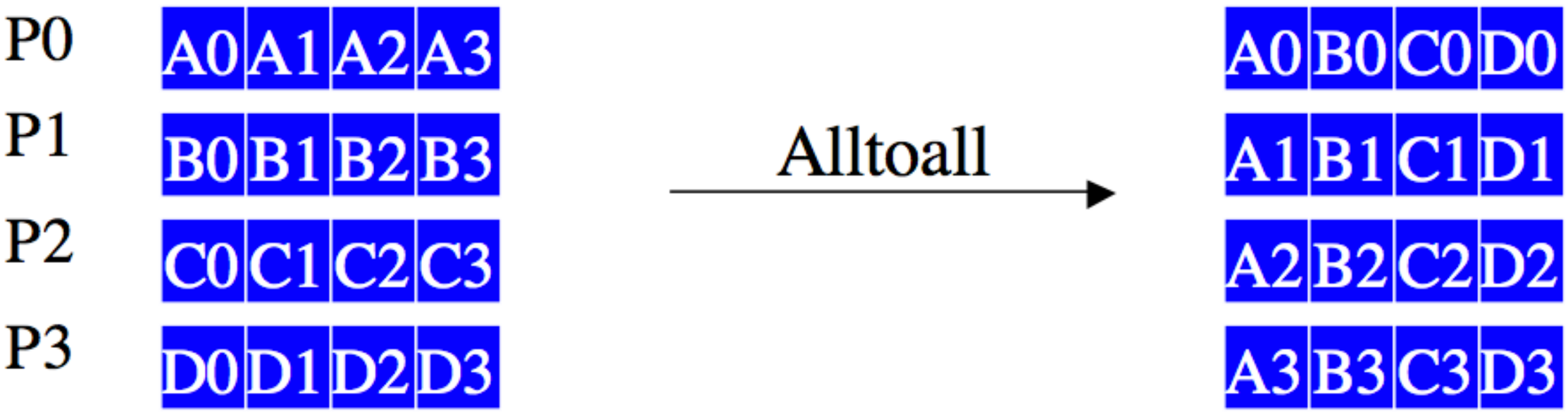
- Scatter: you automatically create a distributed array from a serial one

- Can be used to distribute workload and collect results

Variations

- There are `Allxxxx` and `xxxv` variations of gather
- `All` versions deliver results to all participating processes.
- `v` versions allow the chunks to have different sizes (also for scatter).
- `Alltoall` (with `v` variation) distributes load between all processes


P0
P1
P2
P3
A
B
C
D
Allgather
A B C D
A B C D
A B C D
A B C D


P0
P1
P2
P3
A0 A1 A2 A3
B0 B1 B2 B3
C0 C1 C2 C3
D0 D1 D2 D3
Alltoall
A0 B0 C0 D0
A1 B1 C1 D1
A2 B2 C2 D2
A3 B3 C3 D3

Reduction operations

- To perform a global reduce operation across all members of a group. d_0 **o** d_1 **o** d_2 **o** d_3 **o** … **o** d_{s-2} **o** d_{s-1}
  - single variable, or
  - vector
- **o** = associative operation
- Example:
  - global sum or product
  - global maximum or minimum
  - global user-defined operation

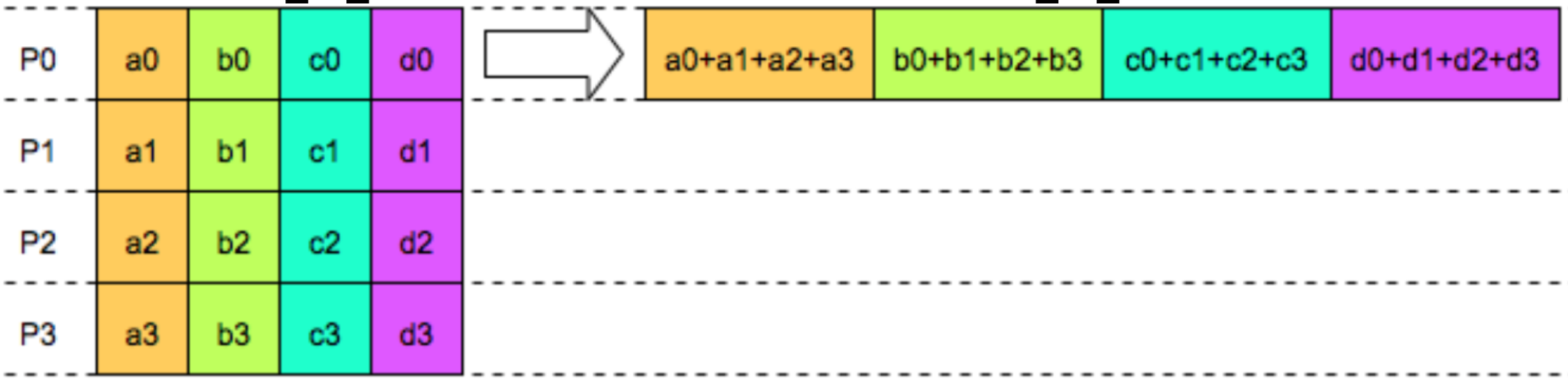
Reduction operations

- To perform a global reduce operation across all members of a group. d_0 **o** d_1 **o** d_2 **o** d_3 **o** … **o** d_{s-2} **o** d_{s-1}

```
MPI_Reduce(void *sbuf,
           void *rbuf,
           int count,
           MPI_Datatype stype,
           MPI_Op op,
           int root,
           MPI_Comm comm)
```

- **o** =
- Ex
  - Es.:
    float abcd[4], sum[4];
  - // ...
    MPI_Reduce(abcd, sum, 4, MPI_Float, MPI_SUM, root,
  - MPI_COMM_WORLD);

Reduction operations



- **o**

- Ex

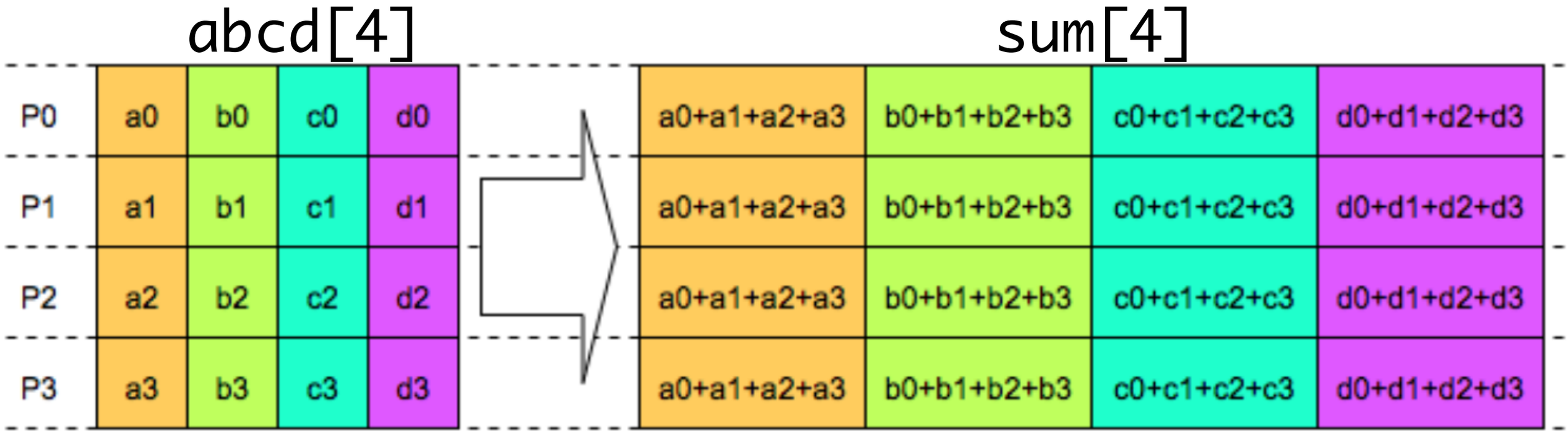
```
                MPI_Op op,
                int root,
                MPI_Comm comm)
```

  - Es.:

```
float abcd[4], sum[4];
// ...
MPI_Reduce(abcd, sum, 4, MPI_Float, MPI_SUM, root,
                MPI_COMM_WORLD);
```

  -

AllReduce variation



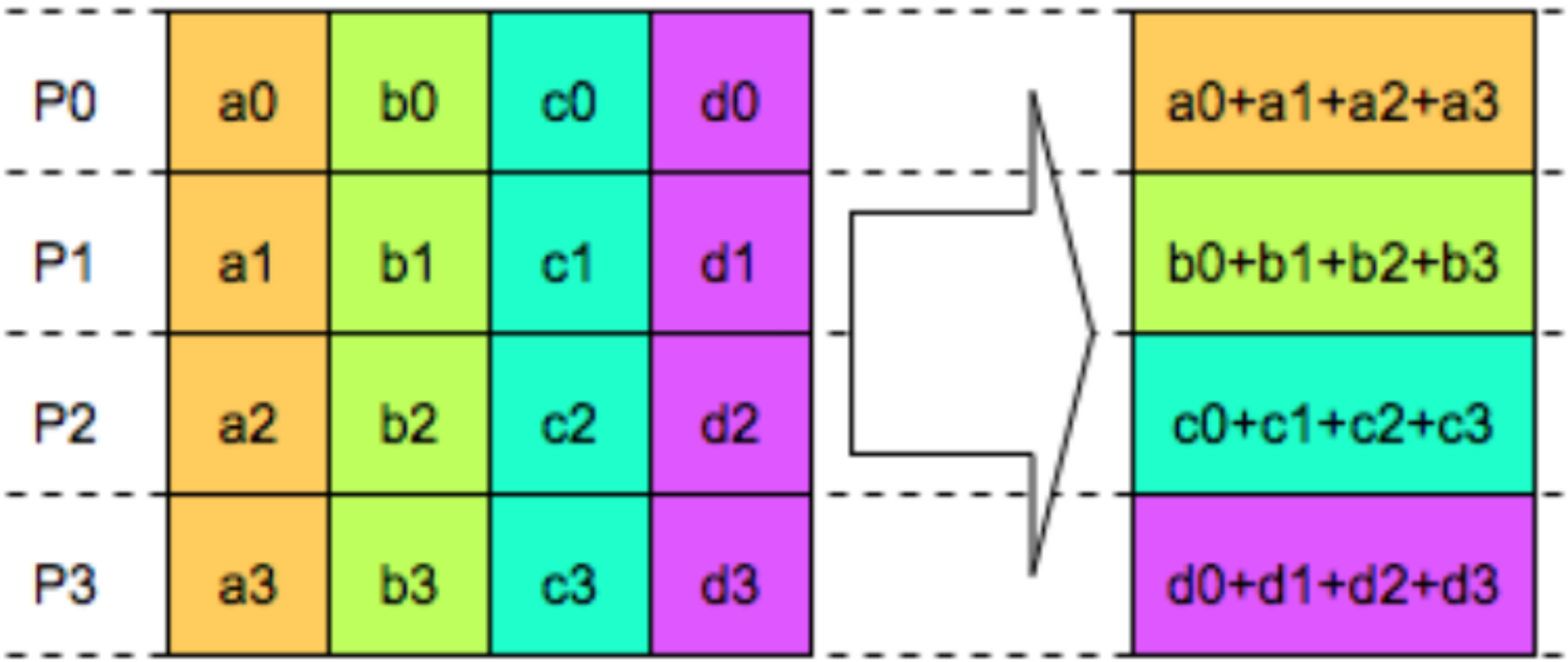

```
float abcd[4], sum[4];

// ...

MPI_AllReduce(abcd, sum, 4, MPI_Float, MPI_SUM, MPI_COMM_WORLD);
```

Reduce_scatter

```
MPI_Reduce_scatter(void *sbuf,
                   void *rbuf,
                   int *rcounts,
                   MPI_Datatype stype,
                   MPI_Op op,
                   MPI_Comm comm)
```

- Same as Reduce followed by Scatter
- Result vector of the reduction operation is scattered to the processes into the real result buffers

Reduce_scatter



- Result vector of the reduction operation is scattered to the processes into the real result buffers

MPI Collective Routines

- Many Routines: MPI_AllGather, MPI_AllGatherV, MPI_AllReduce, MPI_AllToAll, MPI_AllToAllV, MPI_BCast, MPI_Gather, MPI_GatherV, MPI_Reduce, MPI_ReduceScatter, MPI_Scan, MPI_Scatter, MPI_ScatterV
- “All” versions deliver results to all participating processes
- “V” versions (stands for vector) allow the hunks to have different sizes

MPI Built-in Collective Computation Operations

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

User-Defined Reduction Operations

- It's possible to provide user-defined operations
  - should be associative, can be non-commutative
  - must perform the operation on two vectors:
    vecA **o** vecB

- `MPI_Op_create( MPI_User_function *func, int commute, MPI_Op *op);`
  `commute` tells the MPI library whether `func` is commutative or not

- `MPI_Op_free( MPI_User_function *func);`

User-Defined Reduction Operations

- It’s possible to provide user-defined operations
  - should be associative, can be non-commutative
  - must perform the operation on two vectors:
    vecA **o** vecB

```
void oneNorm(float *in, float *inout, int *len, MPI_Datatype *type)  {
  int i;
  for (i=0; i<*len; i++) {
    *inout = fabs(*in) + fabs(*inout); /* one-norm */
    in++;
    inout++;
  }
}
```

```
MPI_Op_free( MPI_User_function *func);
```

Note on communications types

- Use collective communications over point-to-point
  - point-to-point is a bit like assembler programming: low-level, possibly very efficient and RISKY.
  - you are still risking deadlocks, though:
    - the participating processes should call the matching collective communication operations in the same order.
      Also when mixing point-to-point and collective communications

User's datatypes

- MPI has no knowledge of C++ classes or structs
- Must create own datatypes
  - if the struct is made with different types there's an annoying process creating arrays
  - For C++ use Boost.MPI and Boost.Serialization: add serialization code to class then use Boost.MPI to send the messages.

Credits

- These slides report material from:
  - Prof. Robert van Engelen (Florida State University)
  - Prof. Jan Lemeire (Vrjie Universiteit Brussel)
  - Prof. Dan Negrut (Univ. Wisconsin - Madison)
  - William Gropp (Argonne National Lab.)
  - Pavan Balaji and Torsten Hoefler, (ETH)

Books

- Principles of Parallel Programming, Calvin Lyn and Lawrence Snyder, Pearson - Chapt. 7

- Parallel Programming for Multicore and Cluster Systems, Thomas Dauber and Gudula Rünger, Springer - Chapt. 5

- An introduction to parallel programming, Peter S. Pacheco, Morgan Kaufman - Chapt. 3