

High Performance Computing

R. Guivarc'h

Ronan.Guivarch@toulouse-inp.fr

2024 - ModIA

Acknowledgements

This course uses materials from different sources

- ©EPCC, The University of Edinburgh, www.epcc.ed.ac.uk, under a Creative Commons Attribution- NonCommercial- ShareAlike 4.0 International License.
<http://creativecommons.org/licenses/by-nc-sa/4.0/>
deed.en_US



- IDRIS (Institut du développement et des ressources en informatique scientifique)
<http://www.idris.fr/formations/mpi/>
- "Calcul Réparti et Grid Computing", Patrick Amestoy and Michel Daydé (Toulouse INP - ENSEEIHT - IRIT) with J.-Y. L'Excellent (INRIA/LIP-ENS Lyon)

High Performance Computing

- ① Algorithmes Parallèles [Alfredo Buttari], 6 séances de CTD
- ② Paradigmes de Programmation Parallèle
 - Message Passing Interface (MPI) [Ronan Guivarch],
2 CTD + 4 TP + 2 TP éval
 - OpenMP [Alfredo Buttari & Ronan Guivarch], 2 CTD + 2 TP
 - GPU [Ronan Guivarch], 1 CTD + 1 TP

Évaluation

- Partiel sur la partie Algorithmes Parallèles
- Développement et Analyse de Performance MPI
(2 derniers TP réservés, possibilité de commencer avant)

Introduction

Summit Overview

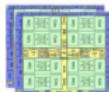


Compute Node

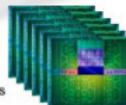
- 2 x POWER9
- 6 x NVIDIA GV100
- NVMe-compatible PCIe 1600 GB SSD



- 25 GB/s EDR IB- (2 ports)
- 512 GB DRAM- (DDR4)
- 96 GB HBM- (3D Stacked)
- Coherent Shared Memory



- NVIDIA GV100**
- 7 TF
 - 16 GB @ 0.9 TB/s
 - NVLink



Compute Rack

- 18 Compute Servers
- Warm water (70°F direct-cooled components)
- RDHX for air-cooled components



- 39.7 TB Memory/rack
55 KW max power/rack



GPFS File System

- 250 PB storage
2.5 TB/s read, 2.5 TB/s write



Introduction

Summit Overview

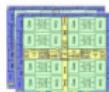


OpenMP

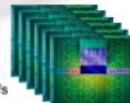
Components

IBM POWER9

- 24 Cores
- 4 Threads/core
- NVLink



- NVIDIA GV100
- 7 TF
 - 16 GB @ 0.9 TB/s
 - NVLink



Compute Node

- 2 x POWER9
6 x NVIDIA GV100
NVMe-compatible PCIe 1600 GB SSD



- 25 GB/s EDR IB- (2 ports)
512 GB DRAM- (DDR4)
96 GB HBM- (3D Stacked)
Coherent Shared Memory

Compute Rack

- 18 Compute Servers
Warm water (70°F direct-cooled
components)
RDHX for air-cooled components



- 39.7 TB Memory/rack
55 KW max power/rack

Compute System

- 10.2 PB Total Memory**
256 compute racks
4,608 compute nodes
 Mellanox EDR IB fabric
200 PFLOPS
~13 MW



GPFS File System

- 250 PB storage**
2.5 TB/s read, 2.5 TB/s write



Introduction

Summit Overview

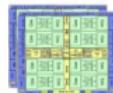


Compute Node

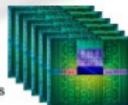
- 2 x POWER9
- 6 x NVIDIA GV100
- NVMe-compatible PCIe 1600 GB SSD



- 25 GB/s EDR IB- (2 ports)
- 512 GB DRAM- (DDR4)
- 96 GB HBM- (3D Stacked)
- Coherent Shared Memory



- NVIDIA GV100
- 7 TF
- 16 GB @ 0.9 TB/s
- NVLink



- Components**
- IBM POWER9**
 - 22 Cores
 - 4 Threads/core
 - NVLink

Compute Rack

- 18 Compute Servers
- Warm water (70°F direct-cooled components)
- RDHX for air-cooled components



- 39.7 TB Memory/rack
- 55 KW max power/rack

MPI

Compute System

- 10.2 PB Total Memory**
- 256 compute racks
- 4,608 compute nodes
- Mellanox EDR IB fabric
- 200 PFLOPS
- ~13 MW



GPFS File System

- 250 PB storage**
- 2.5 TB/s read, 2.5 TB/s write



Introduction

Summit Overview

**OpenPOWER™**

Compute Node

IBM POWER9
• 22 Cores
• 4 Threads/core
• NVLink

Components

IBM POWER9

- 2 x POWER9
- 6 x NVIDIA GV100
- NVMe-compatible PCIe 1600 GB SSD



25 GB/s EDR IB- (2 ports)
512 GB DRAM- (DDR4)
96 GB HBM- (3D Stacked)
Coherent Shared Memory

Compute Rack

18 Compute Servers
Warm water (70°F direct-cooled components)
RDHX for air-cooled components



39.7 TB Memory/rack
55 KW max power/rack

Compute System

10.2 PB Total Memory
256 compute racks
4,608 compute nodes
Mellanox EDR IB fabric
200 PFLOPS
~13 MW



GPFSS File System

250 PB storage
2.5 TB/s read, 2.5 TB/s write



GPU programmation



NVIDIA GV100

- 7 TF
- 16 GB @ 0.9 TB/s
- NVLink



OAK RIDGE
National Laboratory | LEADERSHIP COMPUTING FACILITY

Introduction

Summit Overview

The diagram illustrates the architecture of the Summit supercomputer, showing the flow from individual components to a full system.

Components

- IBM POWER9**
 - 22 Cores
 - 4 Threads/core
 - NVLink
- NVIDIA GV100**
 - 7 TF
 - 16 GB @ 0.9 TB/s
 - NVLink

Compute Node

- 2 x POWER9
- 6 x NVIDIA GV100
- NVMe-compatible PCIe 1600 GB SSD
- 25 GB/s EDR IB- (2 ports)
- 512 GB DRAM- (DDR4)
- 96 GB HBM- (3D Stacked)
- Coherent Shared Memory

Compute Rack

- 18 Compute Servers
- Warm water (70°F direct-cooled components)
- RDHX for air-cooled components
- 39.7 TB Memory/rack
- 55 KW max power/rack

Compute System

- 10.2 PB Total Memory**
- 256 compute racks
- 4,608 compute nodes
- Mellanox EDR IB fabric
- 200 PFLOPS
- ~13 MW

GPFSS File System

- 250 PB storage**
- 2.5 TB/s read, 2.5 TB/s write

OpenMP

Part I - MPI

- ① Presentation of the SimGrid Environment
- ② Codes on Moodle

1. Message-Passing Concepts

- message passing model
- SPMD
- communication modes
- collective communications

Programming Models

Serial Programming

Concepts

Arrays	Subroutines
Control flow	Variables
Human-readable	OO

Message-Passing Parallel Programming

Concepts

Processes	Send/Receive
SPMD	Collectives
Groups	

Languages

Python	C/C++
Java	Fortran
struct	if/then/else

Libraries

MPI
<code>MPI_Init()</code>

Implementations

gcc -O3	pgcc -fast
	icc
crayftn	javac
	craycc

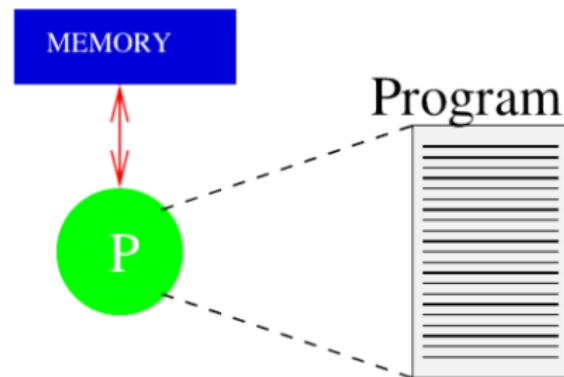
Implementations

Intel MPI	MPICH2
OpenMPI	Cray MPI
	IBM MPI

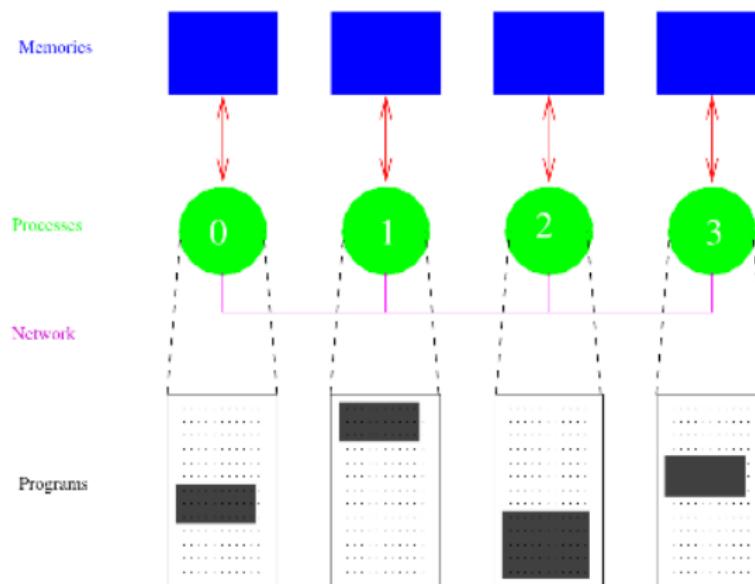
Message Passing Model

- The message passing model is based on the notion of processes - can think of a process as an instance of a running program, together with the program's data
- In the message passing model, parallelism is achieved by having many processes co-operate on the same task
- Each process has access only to its own data - ie all variables are private
- Processes communicate with each other by sending and receiving messages - typically library calls from a conventional sequential language

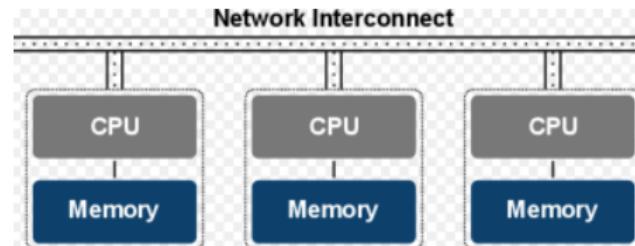
Sequential Paradigm



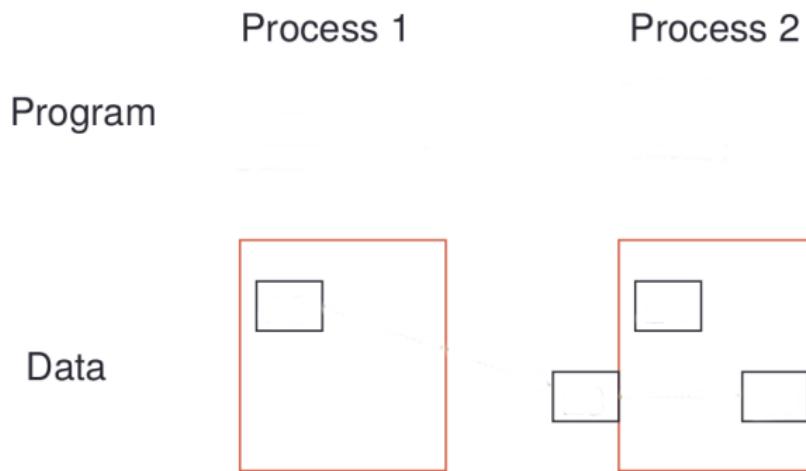
Parallel Paradigm



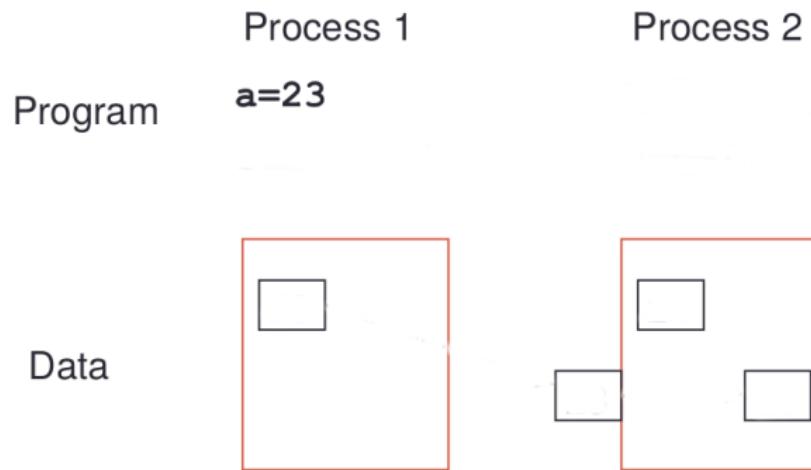
Distributed-Memory Architectures



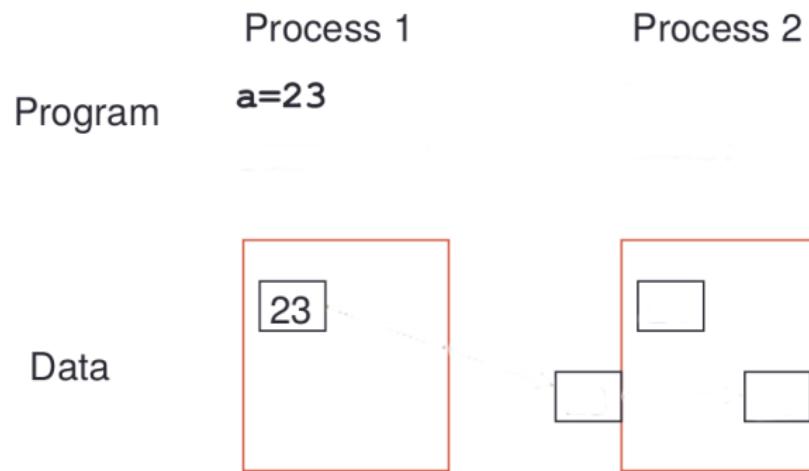
Process Communication



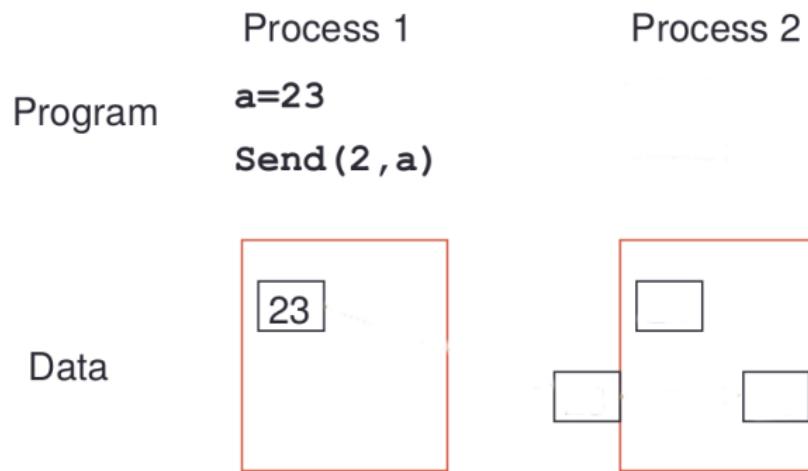
Process Communication



Process Communication



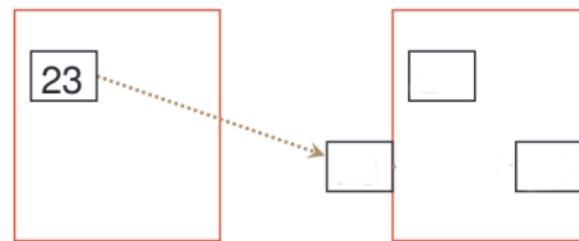
Process Communication



Process Communication

	Process 1	Process 2
Program	a=23	
	Send(2,a)	

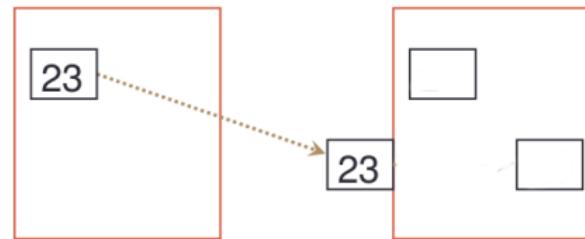
Data



Process Communication

	Process 1	Process 2
Program	a=23	
		Send(2, a)

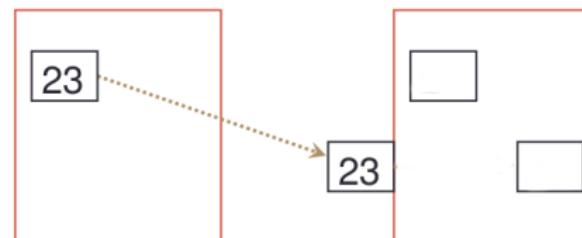
Data



Process Communication

	Process 1	Process 2
Program	<code>a=23</code>	<code>Recv(1,b)</code>
	<code>Send(2,a)</code>	

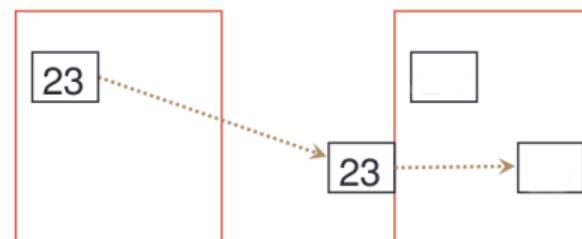
Data



Process Communication

	Process 1	Process 2
Program	$a=23$ Send(2, a)	Recv(1, b)

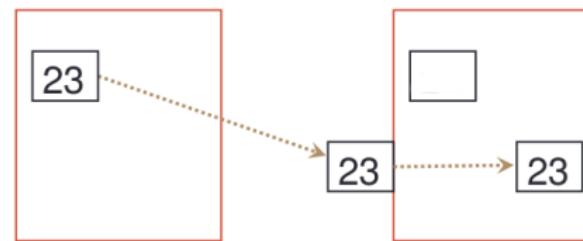
Data



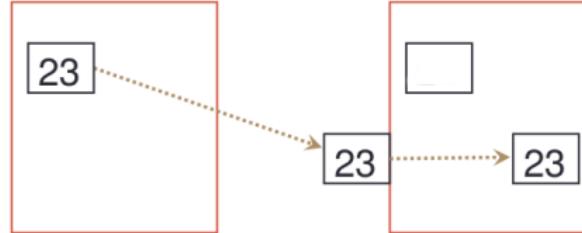
Process Communication

	Process 1	Process 2
Program	$a=23$ Send(2, a)	Recv(1, b)

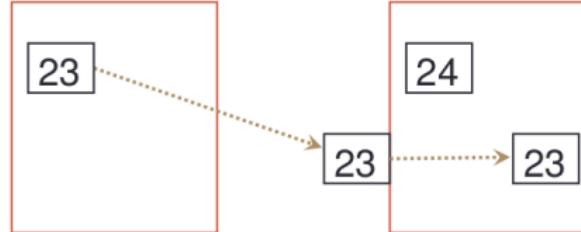
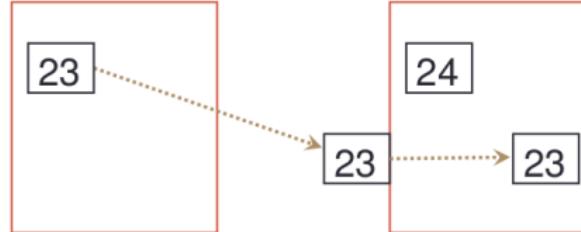
Data



Process Communication

	Process 1	Process 2
Program	$a=23$ Send(2, a)	Recv(1, b) $a=b+1$
Data		

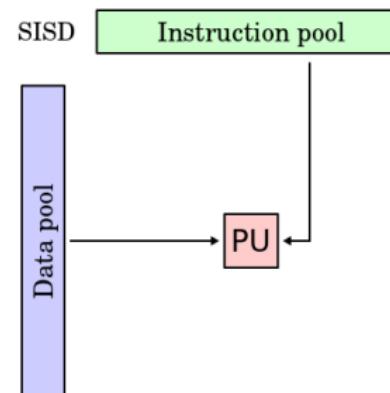
Process Communication

	Process 1	Process 2
Program	$a=23$ Send(2, a)	Recv(1, b) $a=b+1$
Data		

Flynn's taxonomy

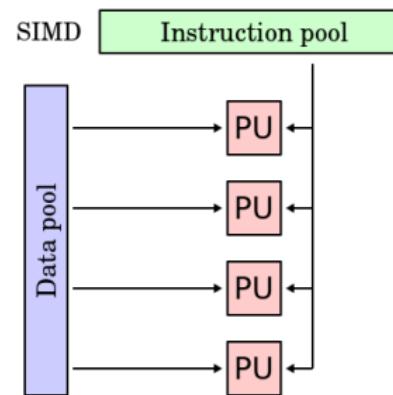
Flynn's taxonomy is a classification of computer architectures, proposed by Michael J. Flynn in 1966 and extended in 1972.
(source: wikipedia)

- Single instruction stream, single data stream (SISD)



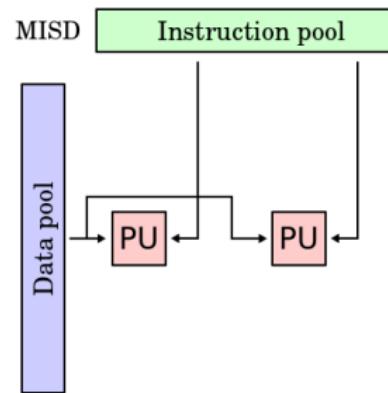
Flynn's taxonomy

- Single instruction stream, single data stream (SISD)
- Single instruction stream, multiple data streams (SIMD)



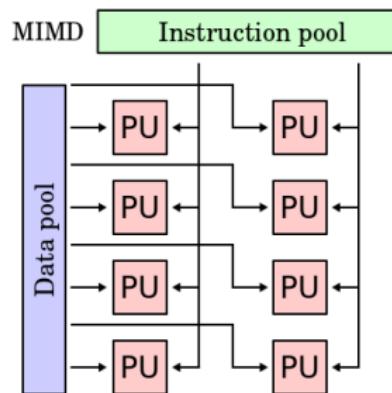
Flynn's taxonomy

- Single instruction stream, single data stream (SISD)
- Single instruction stream, multiple data streams (SIMD)
- Multiple instruction streams, single data stream (MISD)



Flynn's taxonomy

- Single instruction stream, single data stream (SISD)
- Single instruction stream, multiple data streams (SIMD)
- Multiple instruction streams, single data stream (MISD)
- Multiple instruction streams, multiple data streams (MIMD)



Flynn's taxonomy

Although these are not part of Flynn's work, some further divide the MIMD category into the two categories:

- SPMD, Single Program (Process), Multiple Data.
- MPMD, Multiple programs, multiple data:
"Manager/Worker" strategy, ...

SPMD

- Most message passing programs use the Single-Program-Multiple-Data (SPMD) model
- All processes run (their own copy of) the same program
- Each process has a separate copy of the data or a portion of the data
- To make this useful, each process has a unique identifier
- Processes can follow different control paths through the program, depending on their process ID
- Usually run one process per processor / core

From SPMD to MPMD

```
int main (int argc, char *argv[]) {  
  
    if (manager_process) {  
        Manager( /* Arguments */ );  
    } else {  
        Worker( /* Arguments */ );  
    }  
}
```

Messages

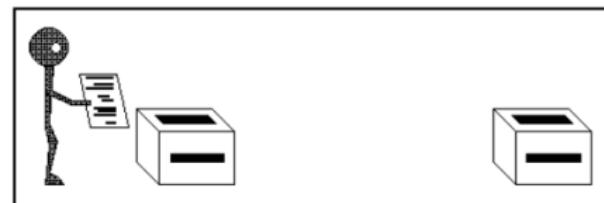
- A message transfers a number of data items of a certain type from the memory of one process to the memory of another process
- A message typically contains
 - the ID of the sending process
 - the ID of the receiving process
 - the type of the data items
 - the number of data items
 - the data itself
 - a message type identifier

Communication modes

- Sending a message can either be synchronous or asynchronous
- A synchronous send is not completed until the message has started to be received
- An asynchronous send completes as soon as the message has gone
- Receives are usually synchronous - the receiving process must wait until the message arrives

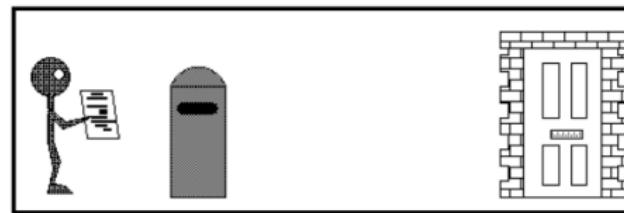
Synchronous send

- Analogy with faxing a letter (or a phone call).
- Know when fax (phone call) has started to be received.



Asynchronous send

- Analogy with posting a letter.
- Only know when letter has been posted, not when it has been received.



Point-to-Point Communications

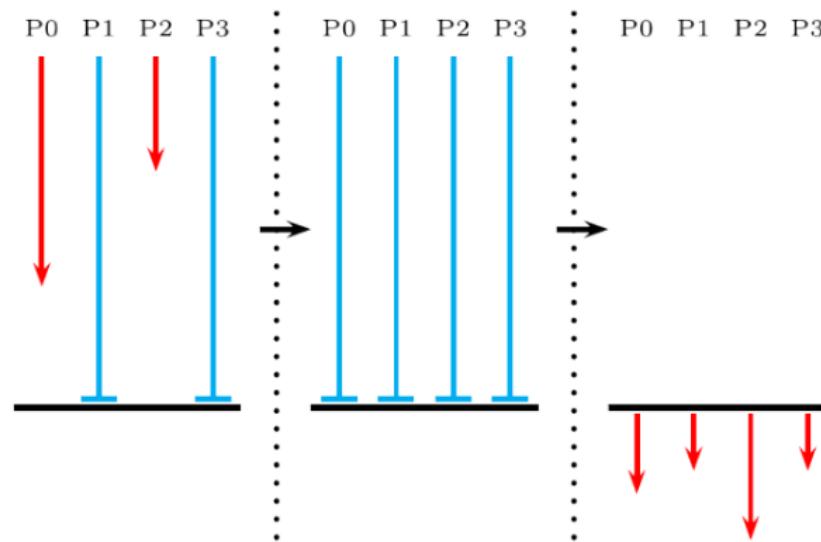
- We have considered two processes
 - one sender
 - one receiver
- This is called point-to-point communication
 - simplest form of message passing
 - relies on matching send and receive

Collective Communications

- There are many instances where communication between groups of processes is required
- Can be built from simple messages, but often implemented separately, for efficiency

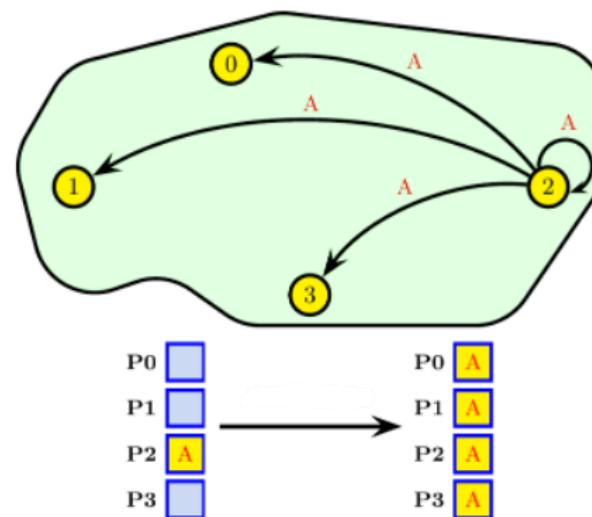
Barrier

- Global synchronisation



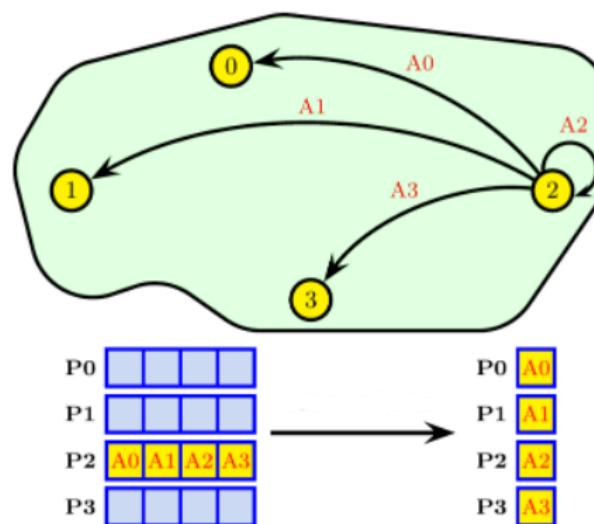
Broadcast

- From one process to all others



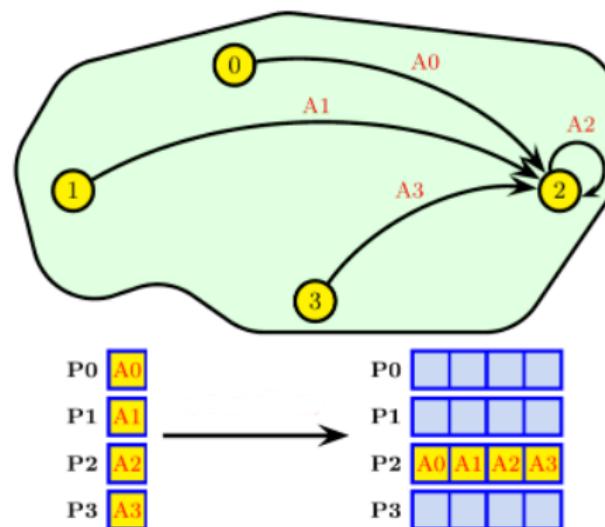
Scatter

- Information scattered to many processes



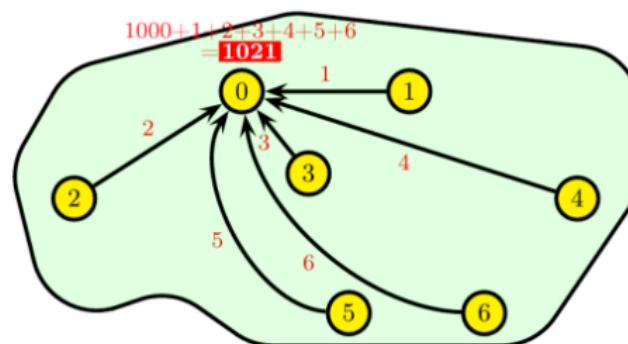
Gather

- Information gathered onto one process



Reduction

- Combine data from several processes to form a single result
- Form a global sum, product, max, min, etc.



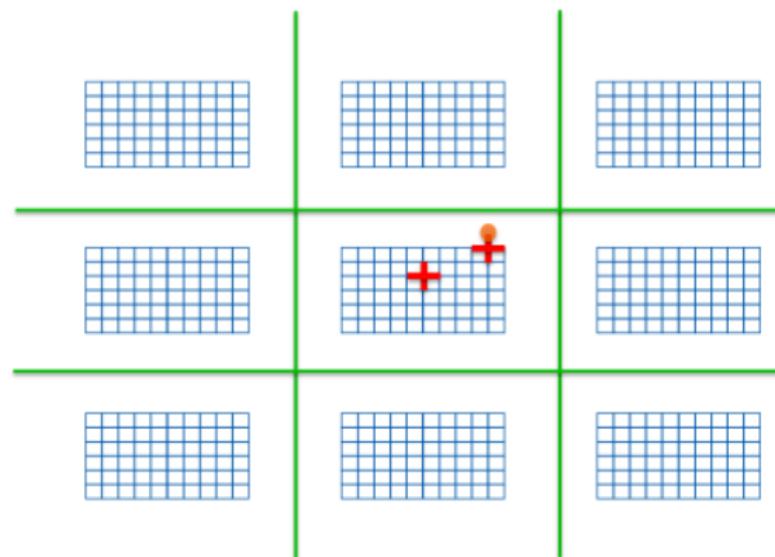
Development of a Message-Passing Program

- Write a **single piece** of source code
 - with calls to message-passing functions such as send / receive
- Compile with a **standard compiler** and link to a **message-passing library** provided for you
 - both open-source and vendor-supplied libraries exist
- Run **multiple copies** of **same executable** on parallel machine
 - each copy is a separate process
 - each has its own private data completely distinct from others
 - each copy can be at a completely different line in the program
- Running is usually done via a launcher program
 - *"please run N copies of my executable called program.exe"*

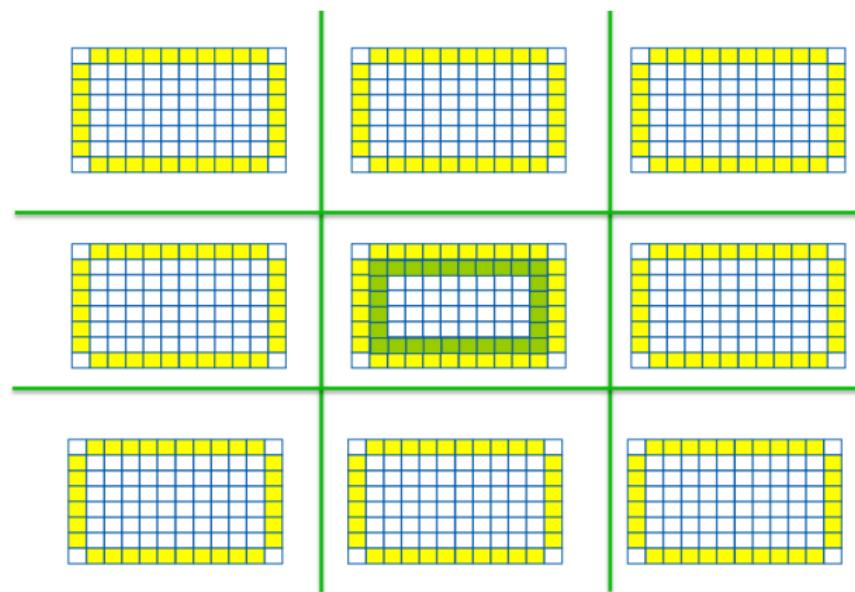
Points of focus

- Sends and receives must match
 - danger of deadlock
 - program will stall (forever!)
- Possible to write very complicated programs, but ...
 - most scientific codes have a simple structure
 - often results in simple communications patterns
- Use collective communications where possible
 - may be implemented in efficient ways

Example of Data transfer



Example of Data transfer



2. Introduction to MPI

What is MPI?

MPI Forum

- First message-passing interface standard.
- Sixty people from forty different organisations.
- Users and vendors represented, from the US and Europe.
- Two-year process of proposals, meetings and review.
- **Message Passing Interface** document produced in 1993

History (source IDRIS)

- **Version 1.0:** June 1994, the MPI (Message Passing Interface) Forum, with the participation of about forty organisations, developed the definition of a set of subroutines concerning the MPI library.
- **Version 1.1:** June 1995, only minor changes.
- **Version 1.2:** 1997, minor changes for more consistency in the names of some subroutines.
- **Version 1.3:** September 2008, with clarifications of the MPI 1.2 version which are consistent with clarifications made by MPI-2.1.
- **Version 2.0:** Released in July 1997, important additions which were intentionally not included in MPI 1.0 (process dynamic management, one-sided communications, parallel I/O, etc.).
- **Version 2.1:** June 2008, with clarifications of the MPI 2.0 version but without any changes.
- **Version 2.2:** September 2009, with only "small" additions.

History (cont.)

- **Version 3.0:** September 2012 Changes and important additions compared to version 2.2:
 - Nonblocking collective communications
 - Revised implementation of one-sided communications
 - Fortran (2003-2008) bindings
 - C++ bindings removed
 - Interfacing of external tools (for debugging and performance measurements)
- **Version 3.1:** June 2015
 - Correction to the Fortran (2003-2008) bindings
 - New nonblocking collective I/O routines
- **Version 4.0:** MPI-4.0 was approved by the MPI Forum on June 9, 2021.
<https://www.mpi-forum.org>

Implementation

- MPI is a library of function/subroutine calls
- MPI is not a language
- There is no such thing as an MPI compiler
- The C or Fortran compiler you invoke knows nothing about what MPI actually does
 - only knows prototype/interface of the function/subroutine calls

Goals and Scope of MPI

- MPI's prime goals are:
 - To provide source-code portability.
 - To allow efficient implementation.
- It also offers:
 - A great deal of functionality.
 - Support for heterogeneous parallel architectures.

Header files

- C/C++:

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

- Fortran 77:

```
#include 'mpif.h'
```

- Fortran 90:

```
use mpi
```

- Fortran 2008:

```
use mpi_f08
```

MPI Function Format

- C:

```
error = MPI_Xxxxx(parameter1, ...);
```

```
MPI_Xxxxx(parameter1, ...);
```

Handles

- MPI controls its own internal data structures.
- MPI defines some new types.
- MPI releases ‘handles’ to allow programmers to refer to these.
- C handles are of defined typedefs.

Initialising MPI

- C:

```
int MPI_Init(int *argc, char ***argv);
```

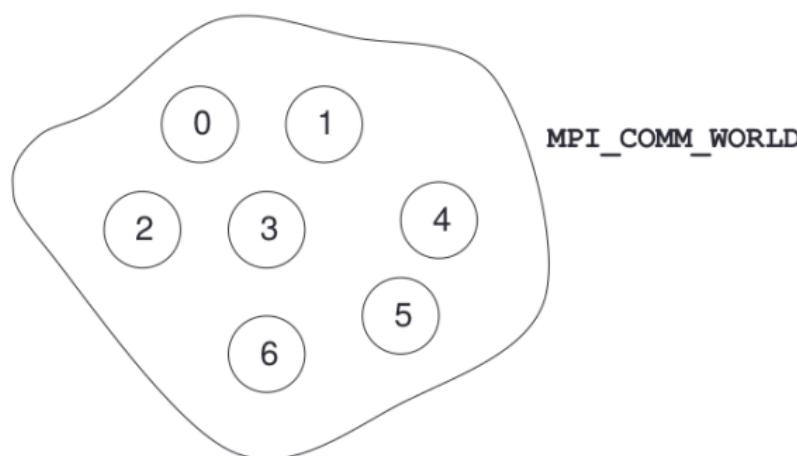
- Must be the first MPI procedure called.
 - but multiple processes are already running before MPI_Init

MPI_Init

```
int main(int argc, char *argv[]) {  
    ...  
    MPI_Init(&argc, &argv);  
    ...  
}  
  
int main(void) {  
    ...  
    MPI_Init(NULL, NULL);  
    ...  
}
```

MPI_COMM_WORLD

- Communicators



Rank

- How do you identify different processes in a communicator?

```
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

- The rank is not the physical processor number.
 - numbering is always $\{0, 1, 2, \dots, N - 1\}$
- N is the total number of MPI process; it is given by

```
int MPI_Comm_size (MPI_Comm comm, int *size);
```

MPI_Comm_rank

```
int rank, size;  
...  
MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
MPI_Comm_size(MPI_COMM_WORLD, &size);  
printf("Hello from process %d of %d\n",  
      rank, size);
```

Exiting MPI

- C:

```
int MPI_Finalize(void);
```

- Must be the last MPI procedure called.

What machine am I on?

- Can be useful on a cluster (e.g. to confirm mapping of processes to nodes/processors/core)

```
int namelen;
char procname[MPI_MAX_PROCESSOR_NAME];
...
MPI_Get_processor_name(procname, &namelen);
printf("rank %d is on machine %s\n",
       rank,           procname);
```

A first example

- ① (Presentation of the SimGrid Environment)
- ② (Codes on Moodle)
- ③ Fill the exemple "who_am_i.c" with the 5 previous MPI functions
- ④ Compile It
- ⑤ Run the code with different numbers of (simulated) processors

documentation and examples: <https://rookiehpc.org>

Open source MPI implementations (IDRIS)

These can be installed on a large number of architectures but their performance results are generally inferior to the implementations of the constructors.

- **MPICH:** <http://www.mpich.org/>
- **Open MPI:** <http://www.open-mpi.org/>

MPI manpages

- documentation of distributions: mpich, open-mpi
- documentation and examples:
<https://rookiehpc.org>
[example of MPI_Comm_size]

References (IDRIS)

- MPI : A Message-Passing Interface Standard, Version 3.1.
High-Performance Computing Center Stuttgart (HLRS), University of Stuttgart, 2015.
<https://fs.hlrs.de/projects/par/mpi/mpi31/>
- William Gropp, Ewing Lusk, and Anthony Skjellum. Using MPI, third edition Portable Parallel Programming with the Message-Passing Interface, MIT Press, 2014.
- William Gropp, Torsten Hoefler, Rajeev Thakur and Erwing Lusk : Using Advanced MPI Modern Features of the Message-Passing Interface, MIT Press, 2014.
- Additional references:
 - Message Passing Interface Forum
<http://www.mpi-forum.org>
 - <http://www.mcs.anl.gov/research/projects/mpi/learning.html>

Tools (IDRIS)

- Debuggers
 - **Totalview:** <https://totalview.io/>
 - **Linaro Forge ex-DDT:**
<https://www.linaroforge.com/>
- Performance measurement
 - **MPE MPI Parallel Environment:**
<http://www.mcs.anl.gov/research/projects/perfvis/download/index.htm>
 - **FFMPI Fast Profiling library for MPI:** <http://www.mcs.anl.gov/research/projects/fppmpi/WWW/>
 - **Scalasca Scalable Performance Analysis of Large-Scale Applications:** <http://www.scalasca.org/>
- Simulation: **SIMGRID**
 - <https://simgrid.org>
 - https://simgrid.github.io/SMPI_CourseWare/

Open source parallel scientific libraries

- **ScaLAPACK** Linear algebra problem solvers using direct methods:
<http://www.netlib.org/scalapack/>
- **PETSc** Linear and non-linear algebra problem solvers using iterative methods: <https://petsc.org/release/>
- **PaStiX** Parallel sparse direct Solvers (Bordeaux):
<https://solverstack.gitlabpages.inria.fr/pastix/>
- **FFTW** Fast Fourier Transform: <http://www.fftw.org>
- **FEAST** Eigenvalue Solver: <https://arxiv.org/abs/2002.04807>
- **FreeFem++** Partial Differential Equation solver
<https://freefem.org/>
- **MUMPS** Parallel sparse direct Solver (Toulouse-Lyon):
<https://mumps-solver.org/>

3. Point-to-Point Communication

Point-to-Point Communication

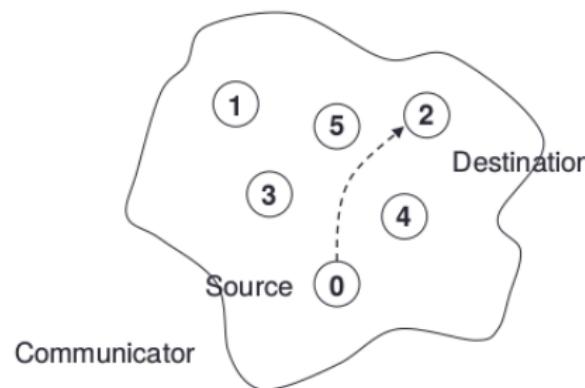
MPI Messages

- A message contains a number of elements of some particular datatype
- MPI datatypes:
 - Basic types
 - Derived types
- Derived types can be built up from basic types.

MPI Basic Datatypes - C

MPI datatypes	C datatypes
MPI_CHAR	signed char
MPI_INT	signed int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_BYTE	char
...	...

Point-to-Point Communication



- Communication between two processes
- Source process sends message to destination process
- Communication takes place within a communicator
- Source and Destination processes are identified by their rank in the communicator

Point-to-point messaging in MPI

- Sender calls a SEND routine
 - specifying the data that is to be sent
 - this is called the ***send buffer***
- Receiver calls a RECEIVE routine
 - specifying where the incoming data should be stored
 - this is called the ***receive buffer***
- Data goes into the receive buffer
- Metadata describing message also transferred
 - this is received into separate storage
 - this is called the ***status***

Communication modes

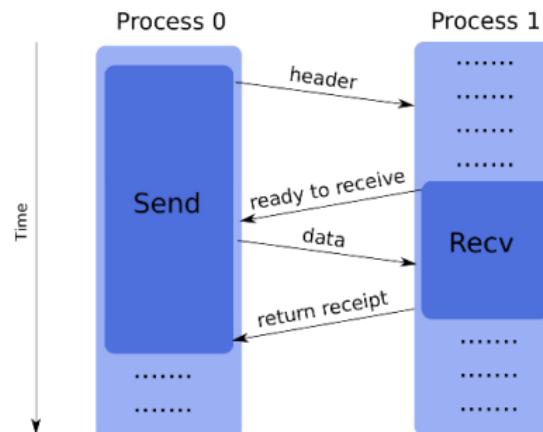
Mode	Notes
Synchronous send	Only completes when the receive has completed
Buffered send	Always completes (unless an error occurs), irrespective of receiver ≡ Asynchronous
Standard send	Either synchronous or buffered
Receive	Completes when a message has arrived, always synchronous

MPI Communication Modes

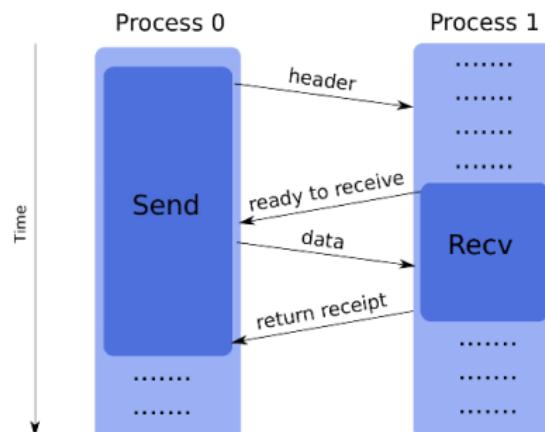
Operation	MPI Call
Synchronous send	<code>MPI_Ssend</code>
Buffered send	<code>MPI_Bsend</code>
Standard send	<code>MPI_Send</code>
Receive	<code>MPI_Recv</code>

Synchronous Blocking Message-Passing

- Processes synchronise
- Sender process specifies the synchronous mode
- Blocking: both processes wait until the transaction has completed



Synchronous Blocking Message-Passing



- The rendezvous protocol is generally the protocol used for synchronous sends (implementation-dependent). The return receipt is optional

Sending a message with synchronous mode

- C:

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

Send data from rank 1 to rank 3

```
// Array of ten integers
int x[10];
...
if (rank == 1) {
    MPI_Ssend(x, 10, MPI_INT, 3, 0, MPI_COMM_WORLD);
}

// Integer scalar
int x;
...
if (rank == 1) {
    MPI_Ssend(&x, 1, MPI_INT, 3, 0, MPI_COMM_WORLD);
}
```

Receiving a message

- C:

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

Receive data from rank 1 on rank 3

```
// Array of ten integers
int y[10];
MPI_Status status;
...
if (rank == 3) {
    MPI_Recv(y, 10, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
}

// Integer scalar
int y;
MPI_Status status;
...
if (rank == 3) {
    MPI_Recv(&y, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
}
```

Implementation of synchronous send and receive

- ① synchronous send and receive of one integer
(02_Ssend_Recv)
- ② synchronous send and receive of a vector of integers
(03_Ssend_RecvV)

For a communication to succeed:

Synchronous or Asynchronous (cf slide 23 and later for precision on asynchronous mode)

- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
- The communicator must be the same
- Tags must match
- Message types must match
- Receiver's buffer must be large enough

Wildcarding

- Receiver can wildcard
- To receive from any source MPI_ANY_SOURCE
- To receive with any tag MPI_ANY_TAG
- Actual source and tag are returned in the receiver's status parameter

Communication Envelope Information

- Envelope information is returned from MPI_RECV as *status*
- Information includes:
 - Source: status.MPI_SOURCE
 - Tag: status.MPI_TAG
 - Count: MPI_Get_count

Examples with 03_Ssend_RecvV

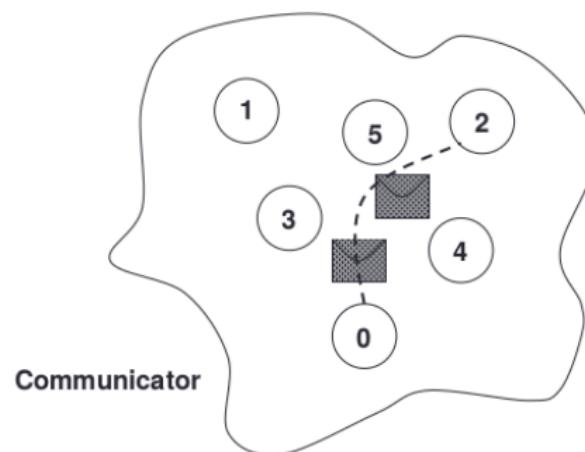
Received Message Count

- C:

```
int MPI_Get_count( MPI_Status *status ,  
                   MPI_Datatype datatype ,  
                   int *count );
```

- Example with 03_Ssend_RecvV [same size, less data, too much data]

Message Order Preservation



- Messages do not overtake each other
- This is true even for non-synchronous sends

Message Matching (i)

Rank 0:

```
Ssend(msg1, dest=1, tag=1)  
Ssend(msg2, dest=1, tag=2)
```

Rank 1:

```
Recv(buf1, src=0, tag=1)  
Recv(buf2, src=0, tag=2)
```

- buf1 = msg1; buf2 = msg2
- Sends and receives correctly matched
- example:** 04_Message_matchind_Ssend

Message Matching (ii)

Rank 0:

```
Ssend(msg1, dest=1, tag=1)  
Ssend(msg2, dest=1, tag=2)
```

Rank 1:

```
Recv(buf2, src=0, tag=2)  
Recv(buf1, src=0, tag=1)
```

- Deadlock (due to synchronous send)
- Sends and receives incorrectly matched

Message Matching (iii)

Rank 0:

```
Bsend(msg1, dest=1, tag=1)  
Bsend(msg2, dest=1, tag=2)
```

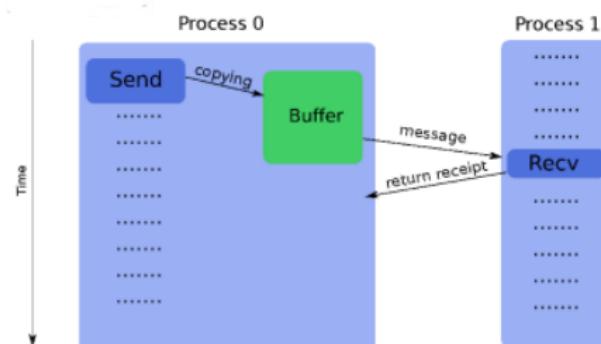
Rank 1:

```
Recv(buf2, src=0, tag=2)  
Recv(buf1, src=0, tag=1)
```

- buf1 = msg1; buf2 = msg2
 - No Deadlock
 - example: 05_Message_matchind_Bsend
- => communication layer on the receiver side stores messages

(MPI Buffered Send)

- A buffered send implies the copying of data into an intermediate memory space
- No coupling between the two processes of communication (asynchronous communication mode)
- Return of this type of send does not mean that the receive has occurred



Message Order Preservation

- If a receive matches multiple messages in the “inbox”
 - then the messages will be received in the order they were sent
- Only relevant for multiple messages from the same source
(examples with : Ssend and Bsend)

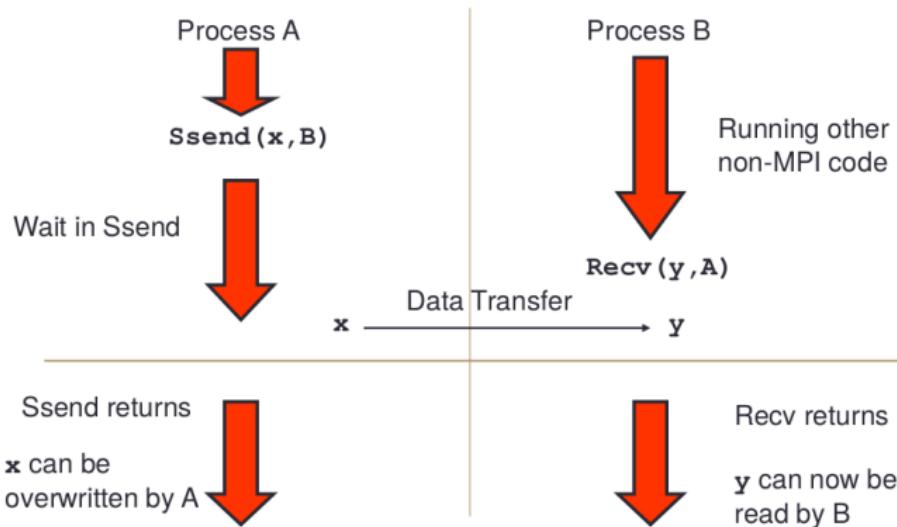
4. Modes, Tag, Communicators

Modes, Tag, Communicators

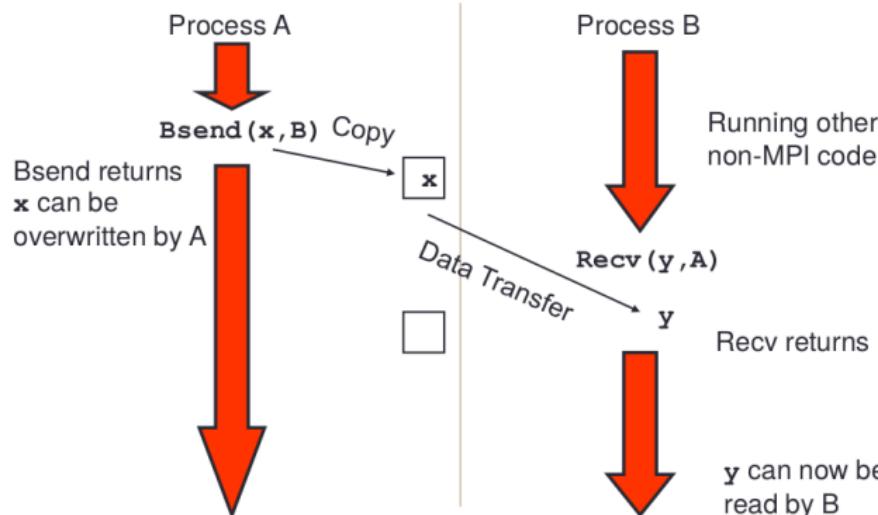
MPI Modes

- MPI_SSEND (Synchronous Send)
 - guaranteed to be synchronous
 - routine will not return until message has been delivered
- MPI_BSEND (Buffered Send)
 - guaranteed to be asynchronous
 - routine returns before the message is delivered
 - system copies data into a buffer and sends it later on
- MPI_SEND (standard Send)
 - may be implemented as synchronous or asynchronous send
 - this causes a lot of confusion (see later)

MPI_Ssend



MPI_Bsend



Notes

- `Recv` is always synchronous
 - if process B issued `Recv` before the `Bsend` from process A, then B would wait in the `Recv` until `Bsend` was issued
- Where does the buffer space come from?
 - for `Bsend`, the user provides a single large block of memory
 - make this available to MPI using `MPI_Buffer_attach`
- If A issues another `Bsend` before the `Recv`
 - system tries to store message in free space in the buffer
 - if there is not enough space then `BSEND` will FAIL!

MPI_Send

- Problems
 - Ssend runs the risk of deadlock
 - Bsend less likely to deadlock, and your code may run faster, but
 - the user must supply the buffer space
 - the routine will FAIL if this buffering is exhausted
- MPI_Send tries to solve these problems
 - buffer space is provided by the system
 - Send will normally be asynchronous (like Bsend)
 - if buffer is full, Send becomes synchronous (like Ssend)
- MPI_Send routine is unlikely to fail
 - but could cause your program to deadlock if buffering runs out

MPI_Send

Process A



Send(x,B)

Recv(x,B)

Process B



Send(y,A)

Recv(y,A)

- This code is **NOT** guaranteed to work
 - will deadlock if `Send` is synchronous
 - is guaranteed to deadlock if you use `Ssend!`

Examples with 06_Send_Recv

Solutions

- To avoid deadlock
 - either match sends and receives explicitly
 - e.g. for ping-pong
 - process A sends then receives
 - process B receives then sends
- For a more general solution use non-blocking communications (see later)

Checking for Messages

- MPI allows you to check if any messages have arrived
 - you can "probe" for matching messages
 - same syntax as receive except no receive buffer specified
- C:

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

- Status is set as if the receive took place
 - e.g. you can find out the size of the message and allocate space prior to receive (see RookieHPC example)
- Be careful with wildcards
 - you can use, e.g., MPI_ANY_SOURCE in call to probe
 - but must use specific source in receive to guarantee matching same message
 - MPI_Recv(buff, count, datatype, status.MPI_SOURCE, ...)

Tags

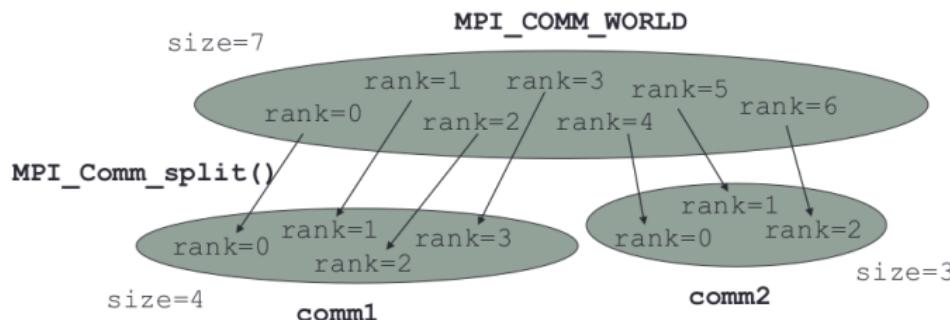
- Every message can have a tag
 - this is a non-negative integer value
 - not everyone uses them; many MPI programs set all tags to zero
- Tags can be useful in some situations
 - can choose to receive messages only of a given tag
- Most commonly used with MPI_ANY_TAG
 - receives the most recent message regardless of the tag
 - user then finds out the actual value by looking at the status

Communicators

- All MPI communications take place within a communicator
 - a communicator is fundamentally a group of processes
 - here is a pre-defined communicator: `MPI_COMM_WORLD` which contains ALL the processes
 - also `MPI_COMM_SELF` which contains only one process
- A message can ONLY be received within the same communicator from which it was sent
 - unlike tags, it is not possible to wildcard on `comm`

Uses of Communicators (i)

- Can split `MPI_COMM_WORLD` into pieces (example: `07_MPI_Comm_Split()`)
 - each process has a new rank within each sub-communicator
 - guarantees messages from the different pieces do not interact
 - can attempt to do this using tags but there are no guarantees



Uses of Communicators (ii)

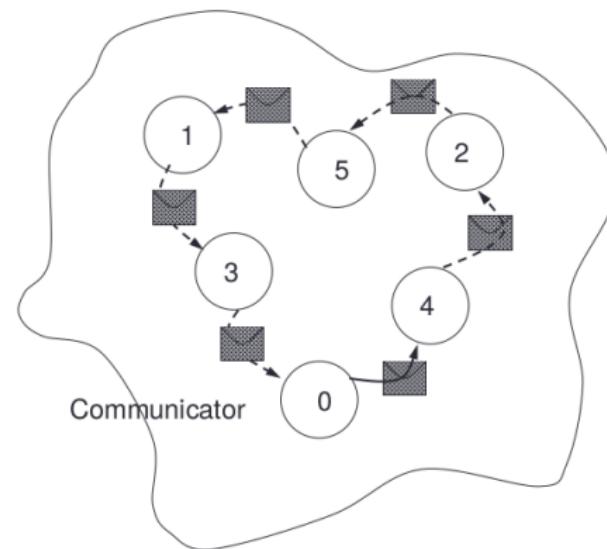
- Can make a copy of MPI_COMM_WORLD
 - e.g. call the MPI_Comm_dup routine
 - containing all the same processes but in a new communicator
- Enables processes to communicate with each other safely within a piece of code
 - guaranteed that messages cannot be received by other code
 - this is **essential** for people writing parallel libraries (e.g. a Fast Fourier Transform, Eigenvalues computation) to stop library messages becoming mixed up with user messages
 - user cannot intercept the library messages if the library keeps the identity of the new communicator a secret
 - not safe to simply try and reserve tag values due to wildcarding

5. Non Blocking Communications

Non Blocking Communications

Deadlock

```
MPI_Ssend(message, count, MPI_INT, next, 0, communicator);  
MPI_Recv(message, count, MPI_INT, previous, 0, communicator);
```



Completion

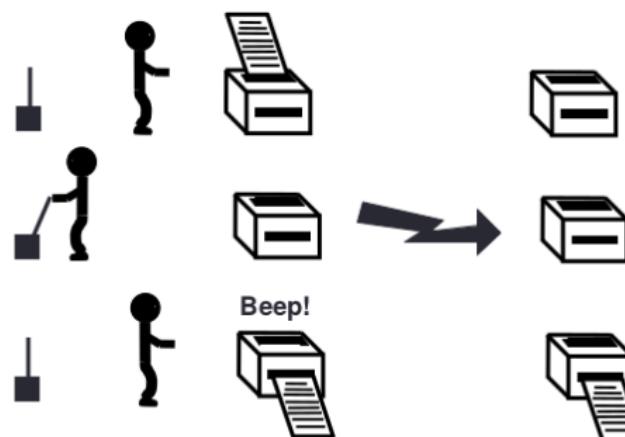
- The **mode** of a communication determines when its constituent operations complete
 - i.e. synchronous / asynchronous
- The **form** of an operation determines when the procedure implementing that operation will return
 - i.e. when control is returned to the user program

Blocking Operations

- Relate to when the operation has completed
- Only return from the subroutine call when the operation has completed
- These are the routines you used thus far
 - MPI_Ssend
 - MPI_Bsend
 - MPI_Recv

Non-Blocking Operations

- Return straight away and allow the user program to continue to perform other work
- At some later time the user program can **test** or **wait** for the completion of the non-blocking operation



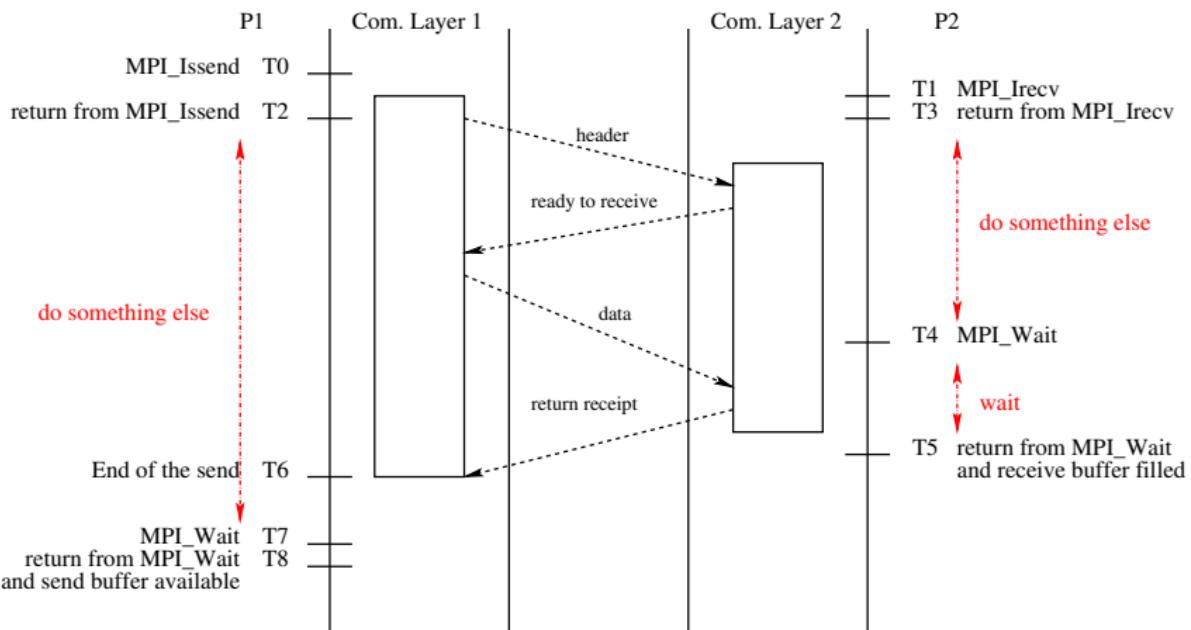
Non-Blocking Operations

- All non-blocking operations should have matching wait operations. Some systems cannot free resources until wait has been called
- A non-blocking operation immediately followed by a matching wait is equivalent to a blocking operation
- Non-blocking operations are not the same as sequential subroutine calls as the operation continues after the call has returned

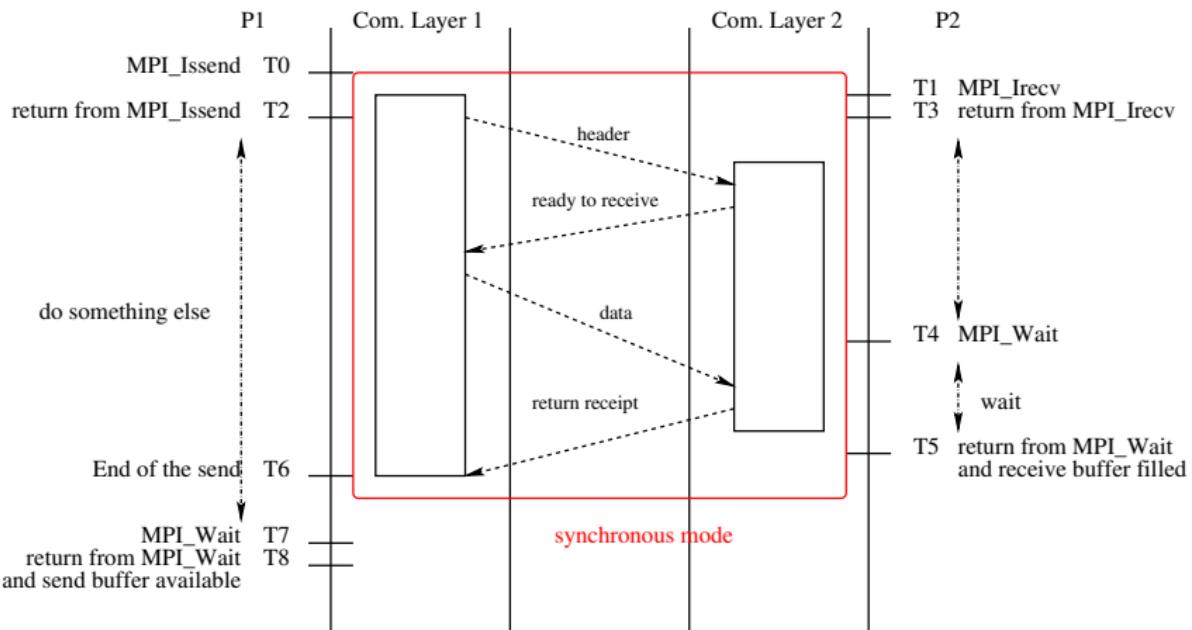
Non-Blocking Communications

- Separate communication into three phases:
 - ① Initiate non-blocking communication
 - ② Do some work (perhaps involving other communications?)
 - ③ Wait for non-blocking communication to complete

Non-Blocking Communication (send and receive)



Mode vs. Form



Handles used for Non-blocking Comms

- datatype same as for blocking (`MPI_Datatype`)
- communicator same as for blocking (`MPI_Comm`)
- A request handle is allocated when a communication is initiated (`MPI_Request`)

Non-blocking Synchronous Send

"I" stands for *Immediate*

- C:

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

Non-blocking Receive

- C:

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

Blocking and Non-Blocking

- Send and receive can be blocking or non-blocking
- A blocking send can be used with a non-blocking receive, and vice-versa
- Non-blocking sends can use any mode - synchronous, buffered or standard
- Synchronous mode affects completion, not initiation

Communication Modes

Non-Blocking Operation	MPI Call
Standard send	<code>MPI_Isend</code>
Synchronous send	<code>MPI_Issend</code>
Buffered send	<code>MPI_Ibsend</code>
Receive	<code>MPI_Irecv</code>

Completion

- C:

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

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

Example (C)

```
MPI_Request request;
MPI_Status status;

if (rank == 0) {
    MPI_Issend(sendarray, 10, MPI_INT, 1, tag,
               MPI_COMM_WORLD, &request);
    Do_something_else_while_Issend_happens ();
    // now wait for send to complete
    MPI_Wait(&request, &status);
} else if (rank == 1) {
    MPI_Irecv(recvarray, 10, MPI_INT, 0, tag,
              MPI_COMM_WORLD, &request);
    Do_something_else_without_data_from_recvarray
        _while_Irecv_happens ();
    // now wait for receive to complete;
    MPI_Wait(&request, &status);
    Do_something_with_the_data_in_recvarray ();
}
```

Multiple Communications

- Test or wait for completion of one message
- Test or wait for completion of all messages
- Test or wait for completion of as many messages as possible

Combined Send and Receive

- Specify all send / receive arguments in one call
 - MPI implementation avoids deadlock
 - useful in simple pairwise communications patterns, but not as generally applicable as non-blocking

```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 int dest, int sendtag,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int source, int recvtag,
                 MPI_Comm comm, MPI_Status *status);
```

6. Collective Communications

Collective Communications

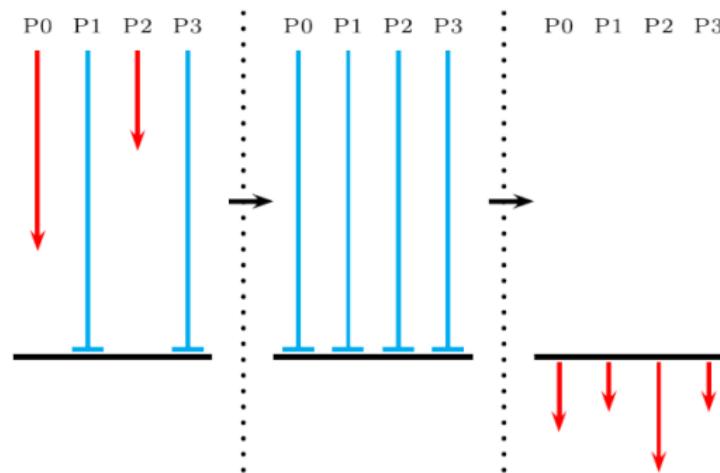
Collective Communication

- Communications involving a group of processes
- Called by all processes in a communicator
- Examples:
 - Barrier synchronisation
 - Broadcast, scatter, gather
 - Global sum, global maximum, etc.

Characteristics of Collective Comms

- Collective action over a communicator
- All processes must communicate
- Standard collective operations are blocking
 - non-blocking versions recently introduced into MPI 3.0
- No tags
- Receive buffers must be exactly the right size

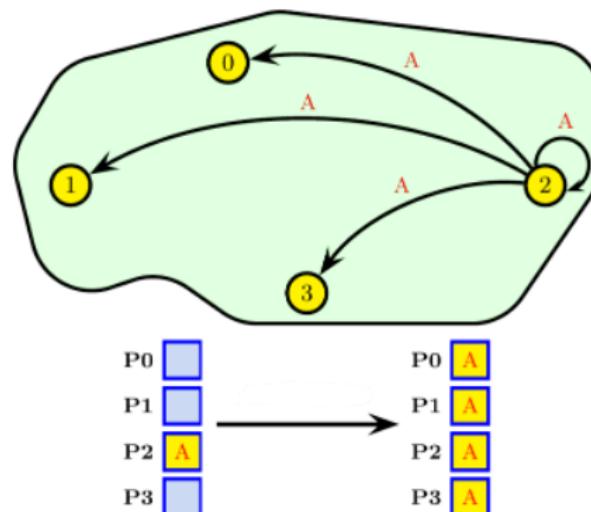
Barrier Synchronisation



C:

```
int MPI_Barrier (MPI_Comm comm);
```

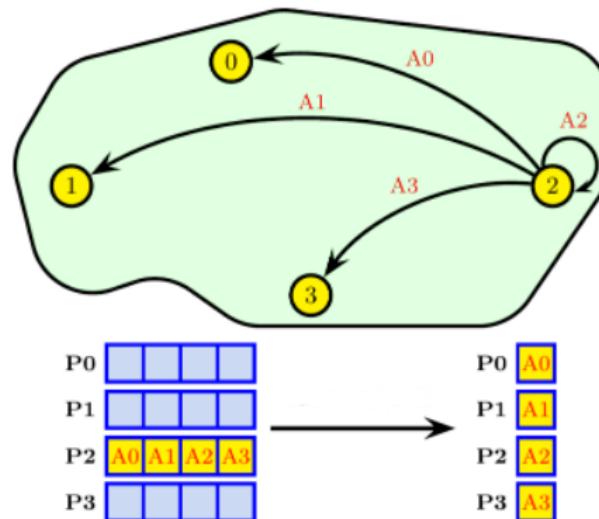
Broadcast



C:

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

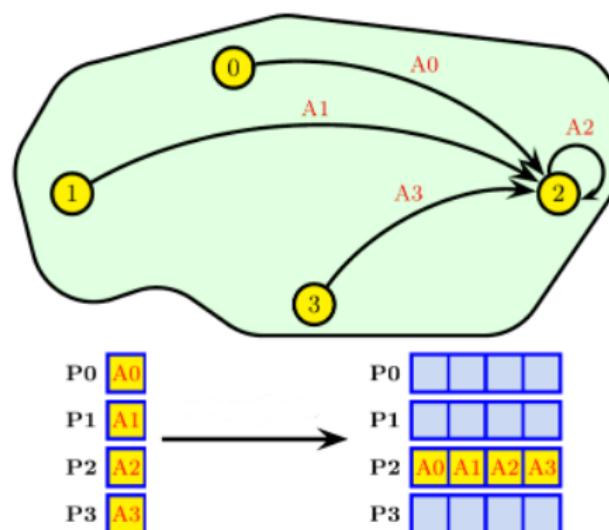
Scatter



C:

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

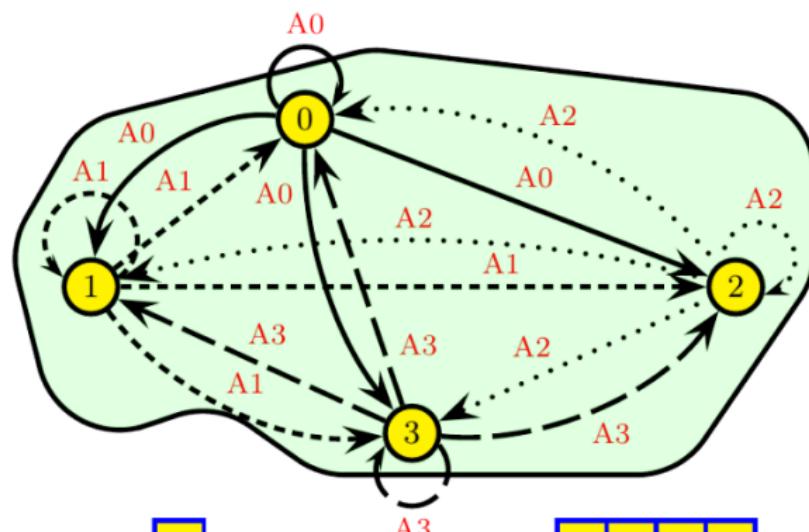
Gather



C:

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

AllGather



P0
A0

P1
A1

P2
A2

P3
A3

MPI_ALLGATHER()

P0
A0 A1 A2 A3

P1
A0 A1 A2 A3

P2
A0 A1 A2 A3

P3
A0 A1 A2 A3

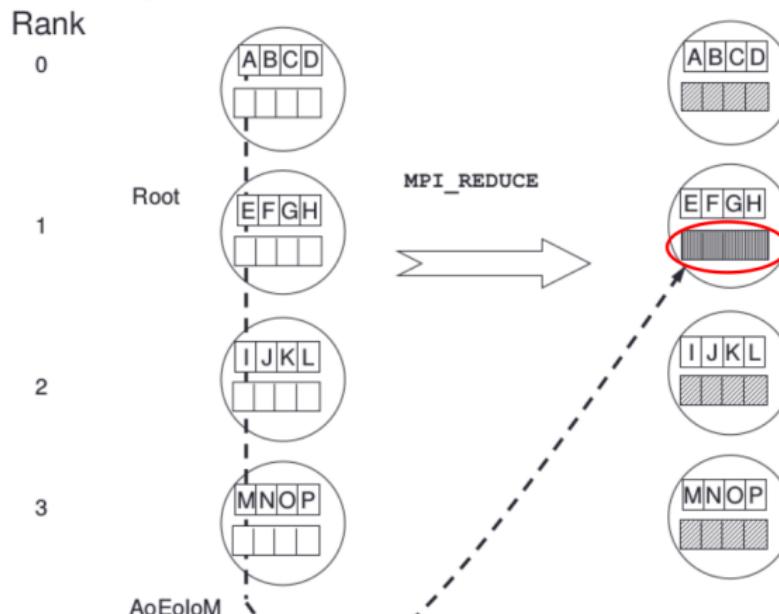
Global Reduction Operations

- Used to compute a result involving data distributed over a group of processes
- Examples:
 - global sum or product
 - global maximum or minimum
 - global user-defined operation

Predefined Reduction Operations

MPI Name	Function
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_MAXLOC	maximum and location
MPI_MINLOC	minimum and location
...	...

Reduce



MPI_Reduce

C:

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

Example of Global Reduction

C:

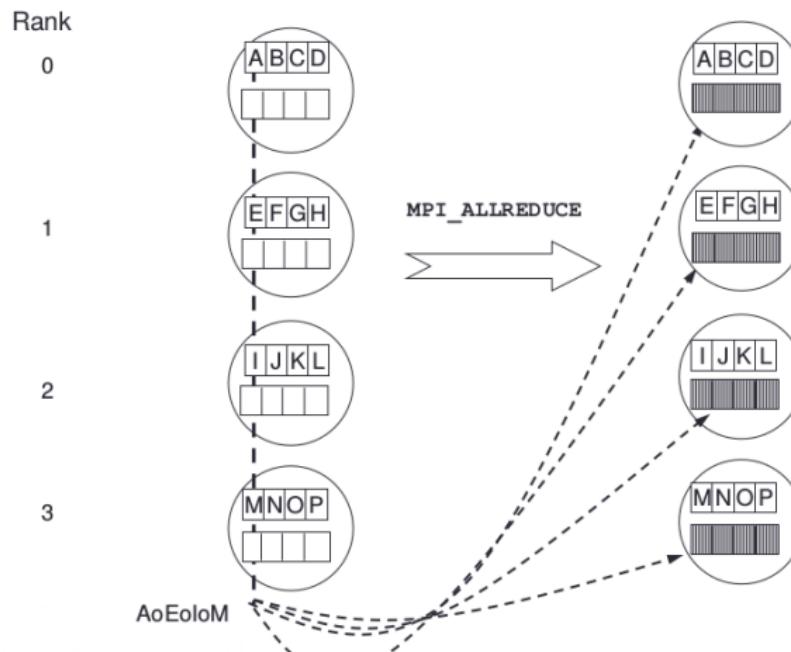
```
int x, result;  
MPI_Reduce(&x, &result, 1, MPI_INT,  
           MPI_SUM, 0, MPI_COMM_WORLD);
```

- Sum of all the x values is placed in result
- The result is only placed there on process 0

Variants of MPI_Reduce

- MPI_Allreduce **no root process**
- MPI_Scan "parallel prefix"

MPI_Allreduce



MPI_Allreduce

C:

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

MPI_Allreduce example

C:

```
int x, result;  
MPI_Allreduce(&x, &result, 1, MPI_INT,  
              MPI_SUM, MPI_COMM_WORLD);
```

- Sum of all the x values is placed in result.
- The result is stored on every process

MPI_Scan

