High Performance Computing

R. Guivarc'h

Ronan.Guivarch@toulouse-inp.fr

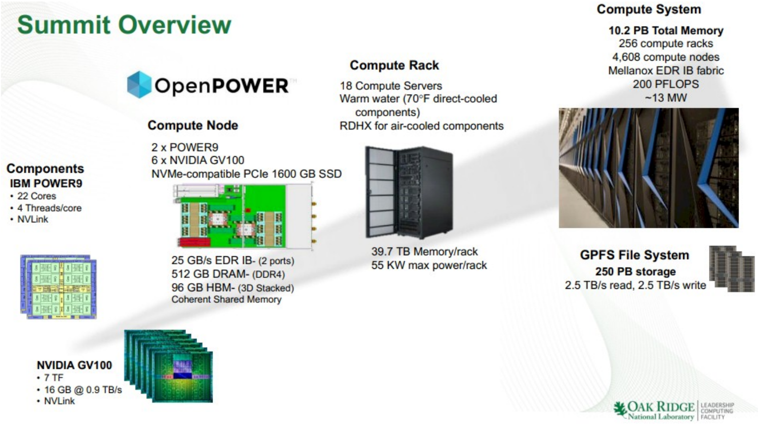
2025 - ModIA

Acknowledgements

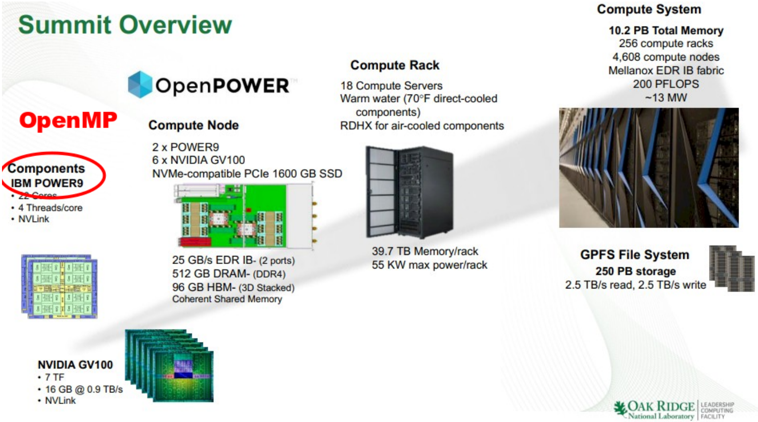
This course uses materials from different sources

- ©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)

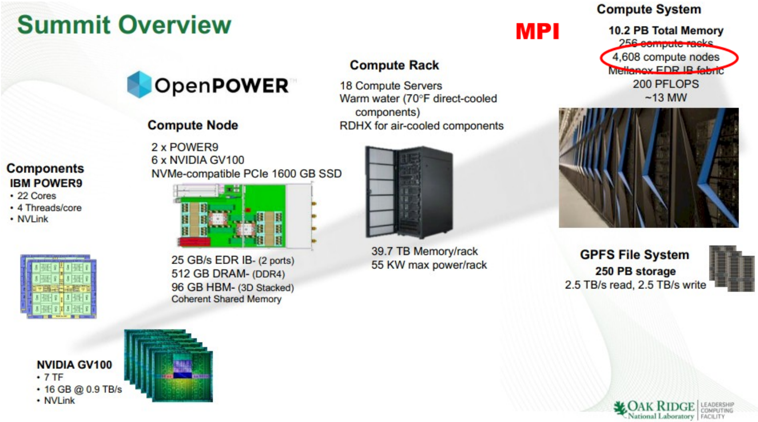
Introduction



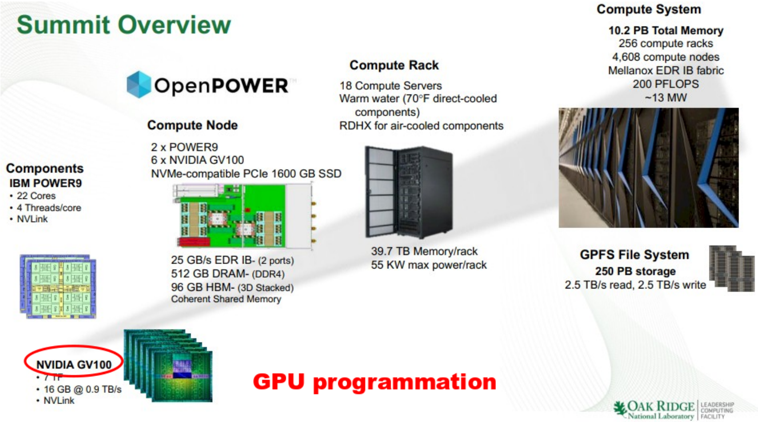
Introduction



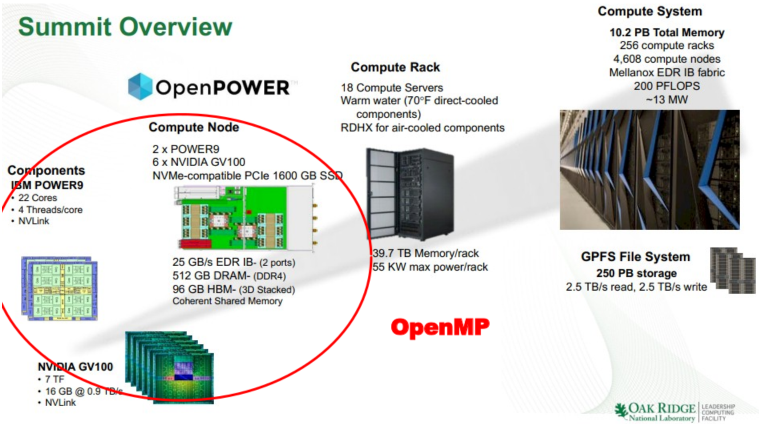
Introduction



Introduction



Introduction



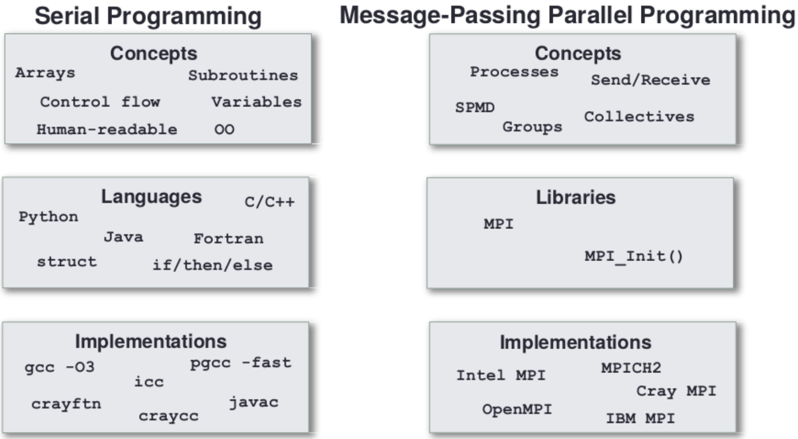
Part I - MPI

1. Presentation of the SimGrid Environment
2. Codes that illustrate some MPI concepts on Moodle

1. Message-Passing Concepts

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

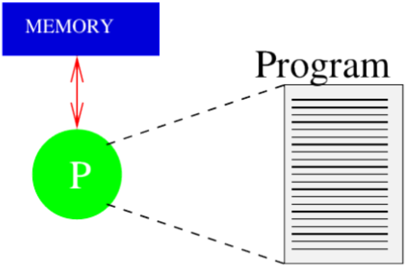
Programming Models



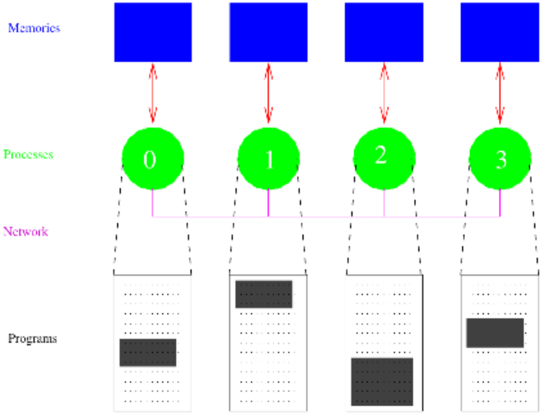
Message Passing Model

- 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

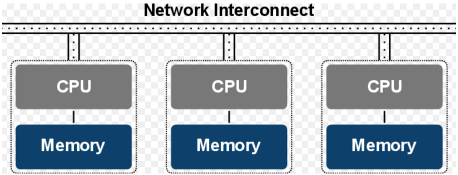
Sequential Paradigm



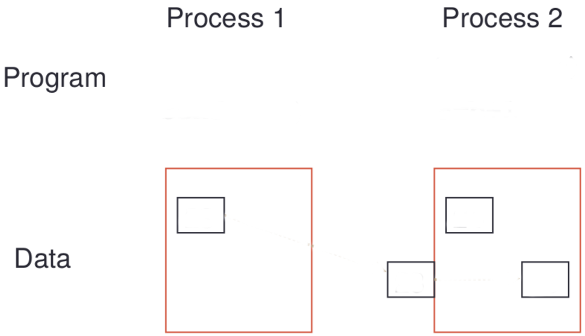
Parallel Paradigm



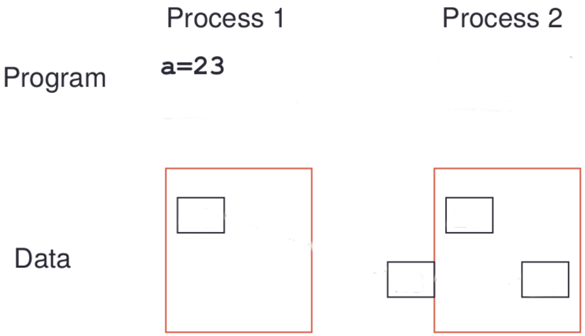
Distributed-Memory Architectures



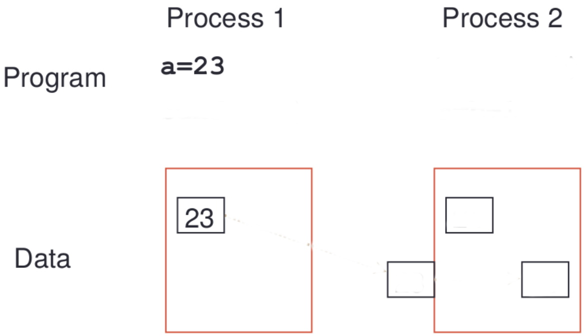
Process Communication



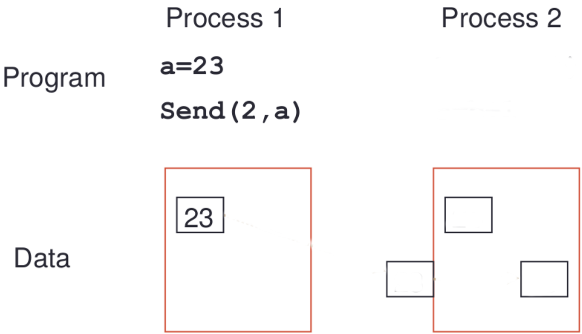
Process Communication



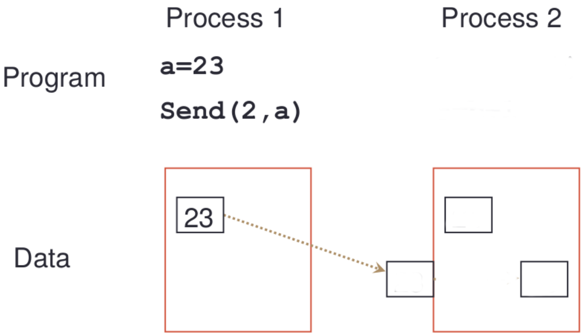
Process Communication



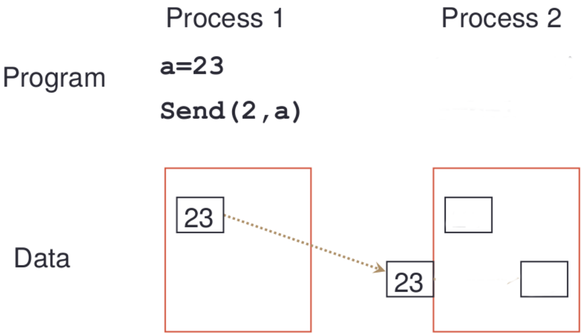
Process Communication



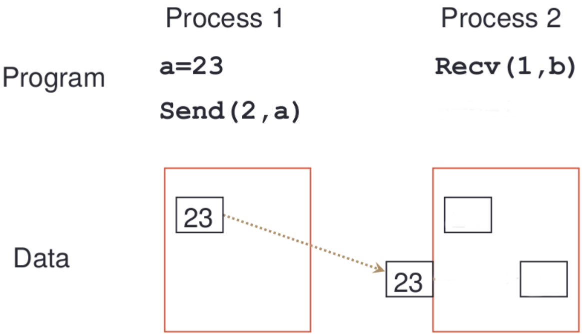
Process Communication



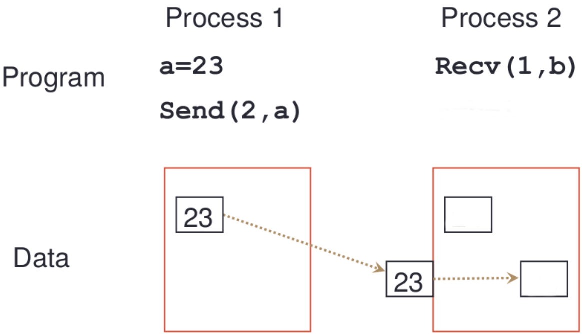
Process Communication



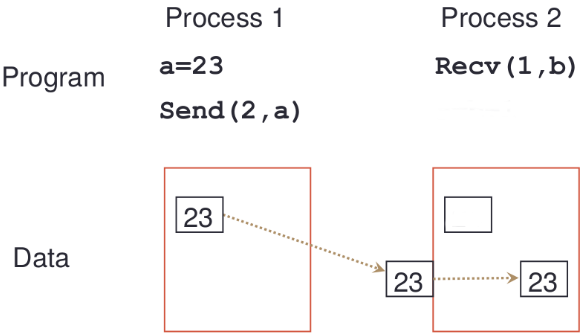
Process Communication



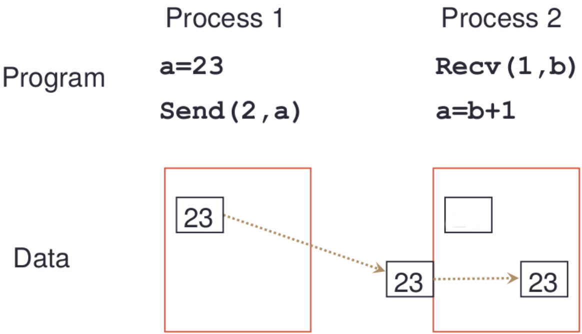
Process Communication



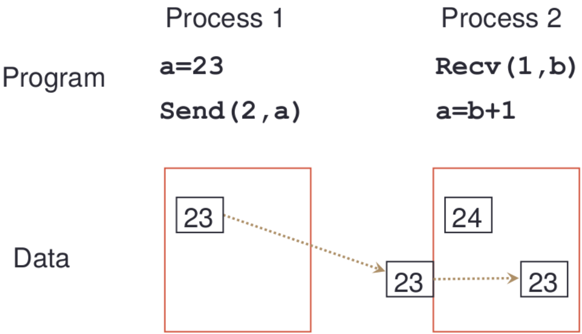
Process Communication



Process Communication



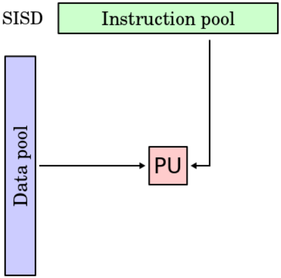
Process Communication



Flynn's taxonomy

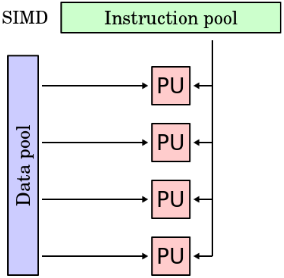
Flynn's taxonomy is a classification of computer architectures, proposed by Michael J. Flynn in 1966 and extended in 1972. (source: wikipedia)

- Single instruction stream, single data stream (SISD)



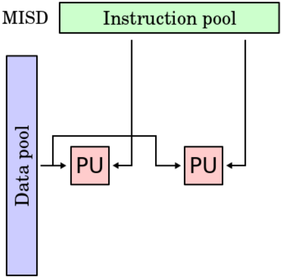
Flynn's taxonomy

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



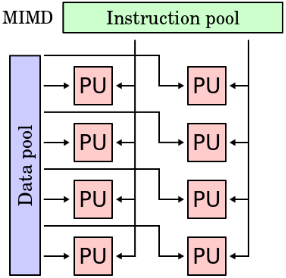
Flynn's taxonomy

- 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)



By I, Cburnett, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2233537

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

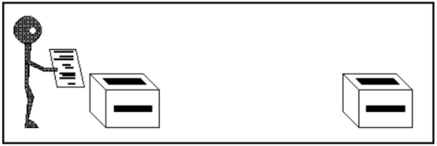
- A message transfers a number of data items of a certain type from the memory of one process to the memory of another process
- A message typically contains
  - the ID of the sending process
  - the ID of the receiving process
  - the type of the data items
  - the number of data items
  - the data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

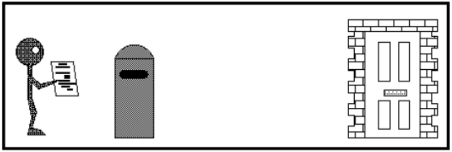
Synchronous send

- 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

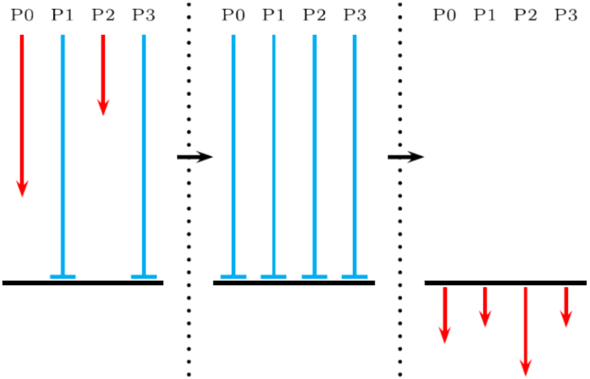
- We have considered two processes
  - one sender
  - one receiver
- This is called point-to-point communication
  - simplest form of message passing
  - relies on matching send and receive

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

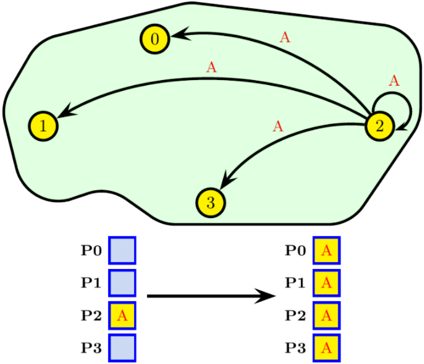
Barrier

- Global synchronisation



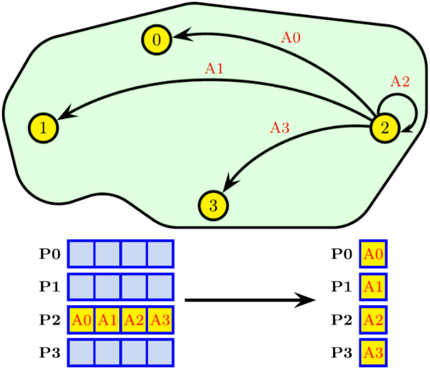
Broadcast

- From one process to all others



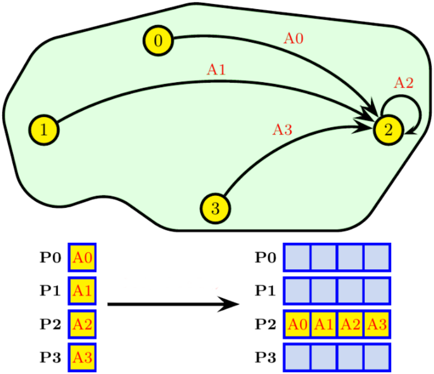
Scatter

- Information scattered to many processes



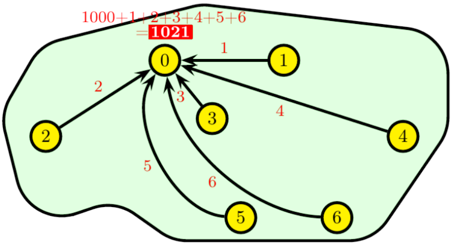
Gather

- Information gathered onto one process



Reduction

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



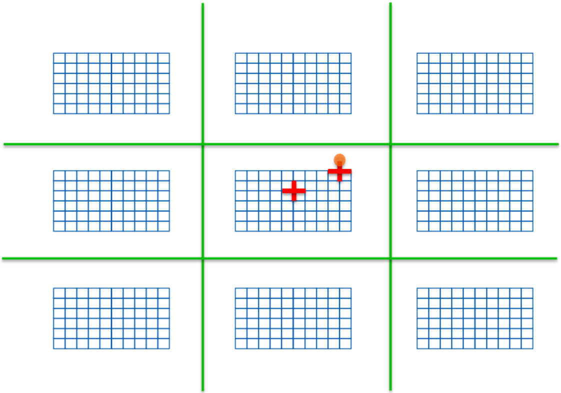
Development of a Message-Passing Program

- 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"*

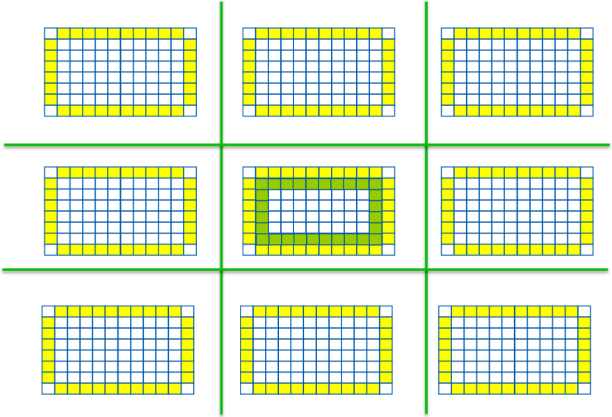
Points of focus

- 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** was approved by the MPI Forum on June 9, 2021.
- **Version 4.1** was approved by the MPI Forum on November 2, 2023. `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/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`

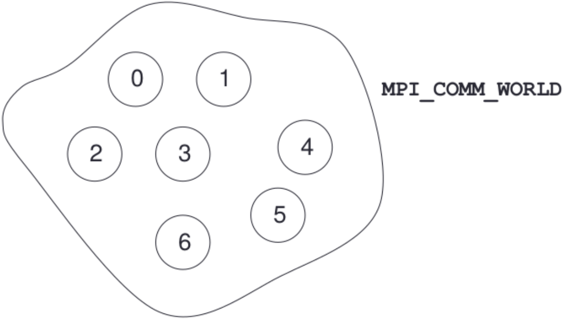
MPI_Init

```
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 $\{0, 1, 2, \ldots, N-1\}$
- 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:

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

- Must be the last MPI procedure called.

```c
MPI_Finalize();
```

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/fpmpi/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

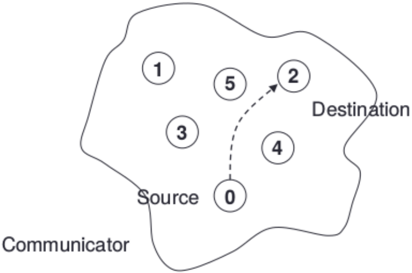
MPI Messages

- 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

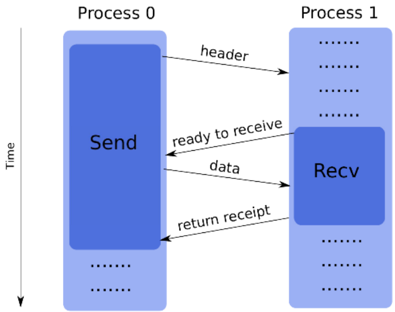
Mode	Notes
Synchronous send	Only completes when the receive has completed
Buffered send	Always completes (unless an error occurs), irrespective of receiver $\equiv$ **Asynchronous**
Standard send	Either synchronous or buffered
Receive	Completes when a message has arrived, always synchronous

MPI Communication Modes

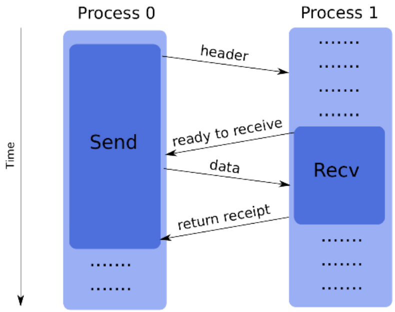
Operation	MPI Call
Synchronous send	MPI_Ssend
Buffered send	MPI_Bsend
Standard send	MPI_Send
Receive	MPI_Recv

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 (see 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

4. Status, Tag, Communicators

Status, Tag, Communicators

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

Commmunication 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]

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

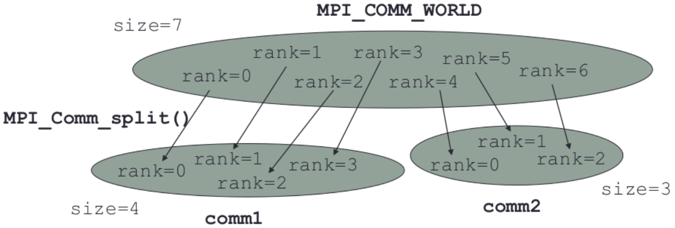
- Every message can have a tag
  - this is a non-negative integer value
  - not everyone uses them; many MPI programs set all tags to zero
- Tags can be useful in some situations
  - can choose to receive messages only of a given tag
- Most commonly used with `MPI_ANY_TAG`
  - receives the most recent 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 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. Collective Communications

Collective Communications

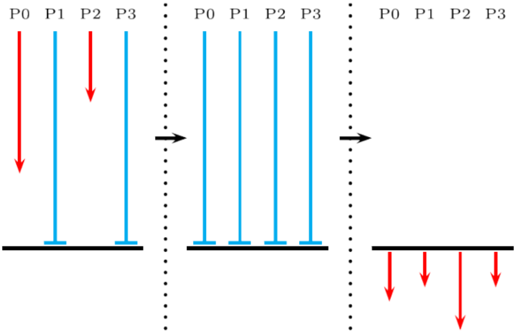
Collective Communication

- 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

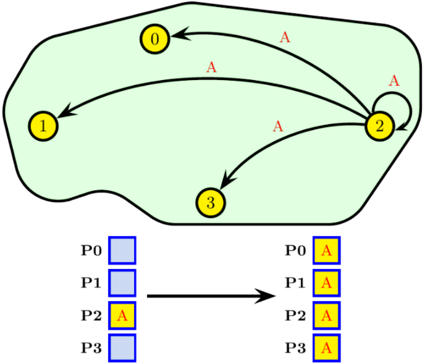
Barrier Synchronisation



C:

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

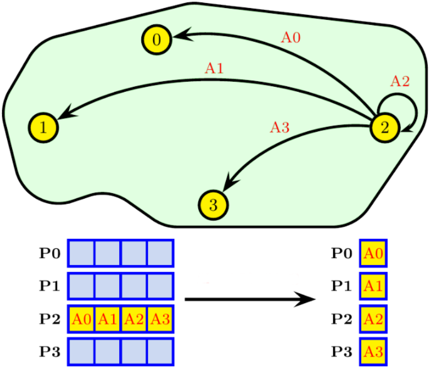
Broadcast



C:

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

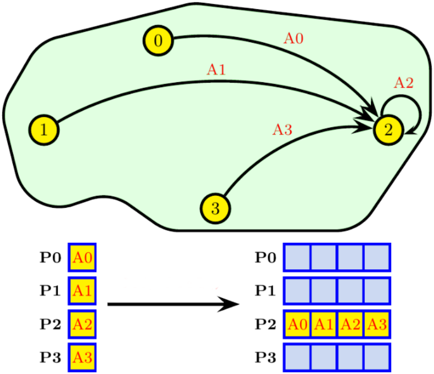
Scatter



C:

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

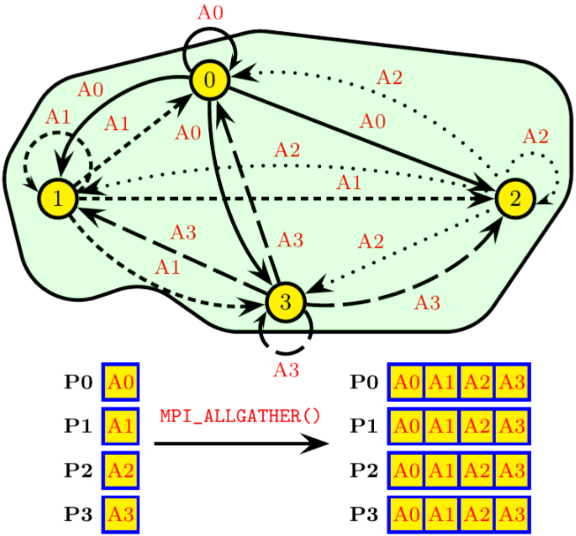
Gather



C:

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

AllGather


A0
A2
A0
A1
A1
A0
A0
A2
A2
A1
A3
A3
A2
A1
A3
A3
0
1
2
3
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

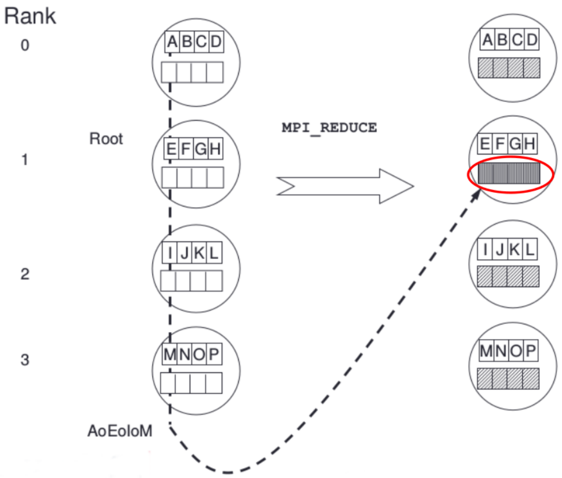
Global Reduction Operations

- 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


Rank
0
1
2
3
Root
A B C D
E F G H
I J K L
M N O P
AoEoIoM
MPI_REDUCE
A B C D
E F G H
I J K L
M N O P

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:

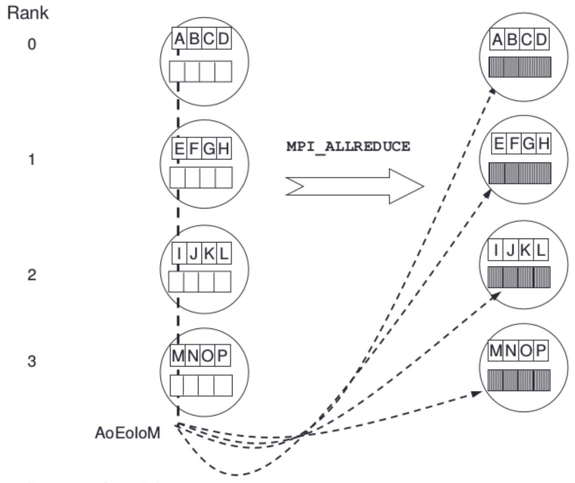
```
int x, result;
MPI_Reduce(&x, &result, 1, MPI_INT,
           MPI_SUM, 0, MPI_COMM_WORLD);
```

- 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);
```

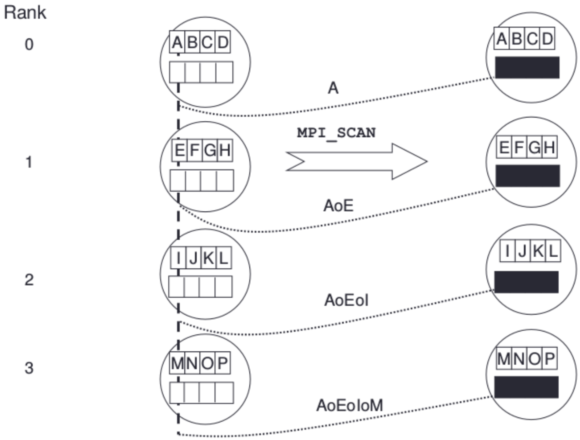
MPI_Allreduce example

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


Rank
0
1
2
3
A B C D
E F G H
I J K L
M N O P
MPI_SCAN
A
AoE
AoEoI
AoEoIoM
A B C D
E F G H
I J K L
M N O P