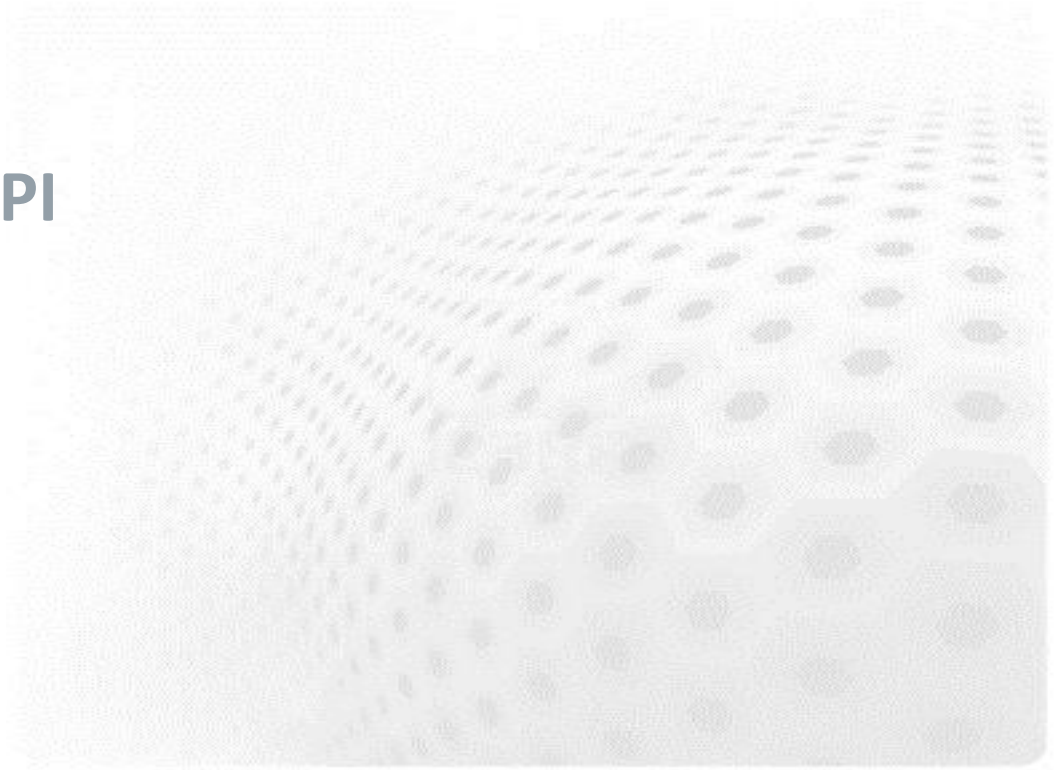
INTRODUCTION TO MPI



Message-passing interface

- MPI is an application programming interface (API) for communication between separate processes
  - The most widely used approach for *distributed* parallel computing
- MPI programs are portable and scalable
- MPI is flexible and comprehensive
  - Large (hundreds of procedures)
  - Concise (often only 6 procedures are needed)
- MPI standardization by MPI Forum

Execution model

- Parallel program is launched as set of independent, identical processes
- The same program code and instructions
- Can reside in different nodes
  - or even in different computers
- The way to launch parallel program is implementation dependent
  - mpirun, mpiexec, srun, aprun, poe, ...

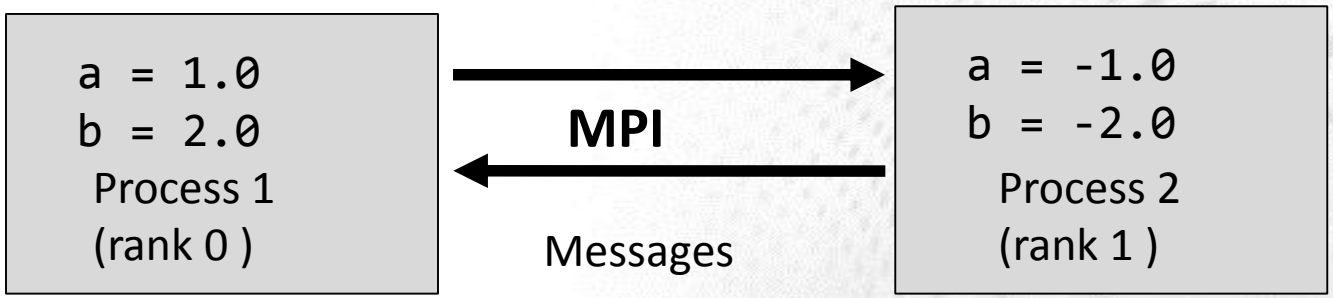
MPI ranks

- MPI runtime assigns each process a rank
  - identification of the processes
  - ranks start from 0 and extent to N-1
- Processes can perform different tasks and handle different data basing on their rank

```
...
if ( rank == 0 ) {
    ...
    }
if ( rank == 1) {
    ...
    }
...
```

Data model

- All variables and data structures are local to the process
- Processes can exchange data by sending and receiving messages



MPI communicator

- Communicator is an object connecting a group of processes
- Initially, there is always a communicator MPI_COMM_WORLD which contains all the processes
- Most MPI functions require communicator as an argument
- Users can define own communicators

Routines of the MPI library

- Information about the communicator
  - number of processes
  - rank of the process
- Communication between processes
  - sending and receiving messages between two processes
  - sending and receiving messages between several processes
- Synchronization between processes
- Advanced features

Programming MPI

- MPI standard defines interfaces to C and Fortran programming languages
  - There are unofficial bindings to Python, Perl and Java
- C call convention

  rc = MPI_Xxxx(parameter,...)
  - some arguments have to be passed as pointers
- Fortran call convention

  CALL MPI_XXXX(parameter,..., rc)
  - return code in the last argument

First five MPI commands

- Set up the MPI environment

  `MPI_Init()`

- Information about the communicator

  `MPI_Comm_size(comm, size)`

  `MPI_Comm_rank(comm, rank)`

  – Parameters

  comm communicator

  size number of processes in the communicator

  rank rank of this process

First five MPI commands

- Synchronize processes

  `MPI_Barrier(comm)`

- Finalize MPI environment

  `MPI_Finalize()`

Writing an MPI program

- Include MPI header files

```
C:          #include <mpi.h>
Fortran:    use mpi
```

- Call MPI_Init
- Write the actual program
- Call MPI_Finalize before exiting from the main program

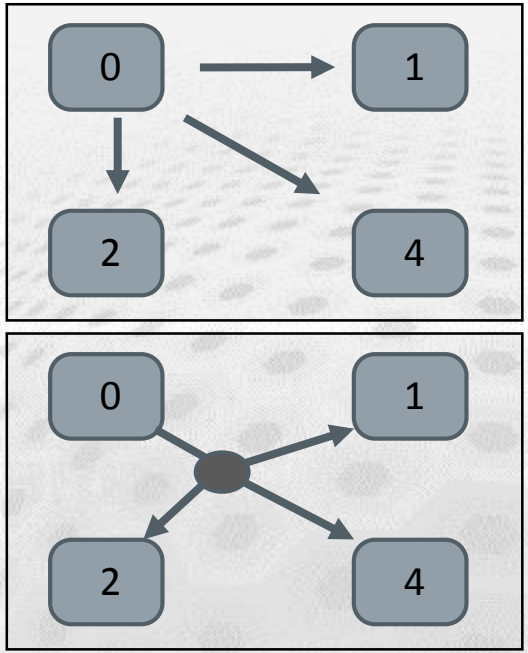
Summary

- In MPI, a set of independent processes is launched
  - Processes are identified by ranks
  - Data is always local to the process
- Processes can exchange data by sending and receiving messages
- MPI library contains functions for
  - Communication and synchronization between processes
  - Communicator manipulation

POINT-TO-POINT COMMUNICATION

Introduction

- MPI processes are independent, they communicate to coordinate work
- Point-to-point communication
  - Messages are sent between two processes
- Collective communication
  - Involving a number of processes at the same time



MPI point-to-point operations

- One process *sends* a message to another process that *receives* it
- Sends and receives in a program should match – one receive per send

MPI point-to-point operations

- Each message (envelope) contains
  - The actual *data* that is to be sent
  - The *datatype* of each element of data.
  - The *number of elements* the data consists of
  - An identification number for the message (*tag*)
  - The ranks of the *source* and *destination* process

Presenting syntax

Operations presented in pseudocode, C and Fortran bindings presented in extra material slides.

INPUT arguments in red

OUTPUT arguments in blue

Note! Extra error parameter for Fortran

Slide with extra material included in handouts

Send operation

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

- buf — The data that is sent
- count — Number of elements in buffer
- datatype — Type of each element in buf (see later slides)
- dest — The rank of the receiver
- tag — An integer identifying the message
- comm — A communicator
- error — Error value; in C/C++ it's the return value of the function, and in Fortran an additional output parameter

Send operation

```
int MPI_Send(void *buffer, int count, MPI_Datatype datatype,int
dest, int tag, MPI_Comm comm)
```

- The return value of the function is the error value

Fortran binding

```
MPI_SEND(buffer, count, datatype,
   dest,tag, comm, ierror)
<type> buf(*)
integer count, datatype, dest, tag, comm, ierror
```

- **ierror:** the error value

Send operation

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

buf	The data that is sent
count	Number of elements in buffer
datatype	Type of each element in buf (see later slides)
dest	The rank of the receiver
tag	An integer identifying the message
comm	A communicator
error	Error value; in C/C++ it's the return value of the function, and in Fortran an additional output parameter

Receive operation

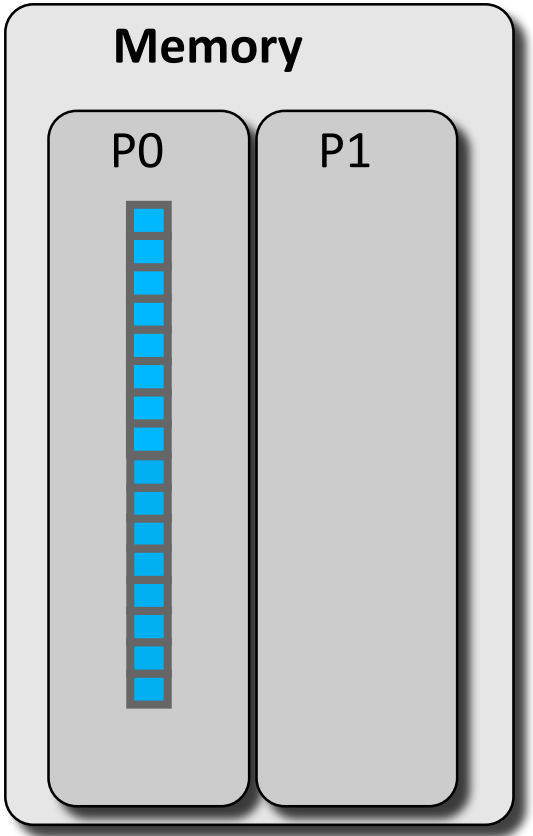
```
MPI_Recv(buf, count, datatype, source, tag, comm, status)
```

buf	Buffer for storing received data
count	Number of elements in buffer, not the number of element that are actually received
datatype	Type of each element in buf
source	Sender of the message
tag	Number identifying the message
comm	Communicator
status	Information on the received message
error	As for send operation

MPI datatypes

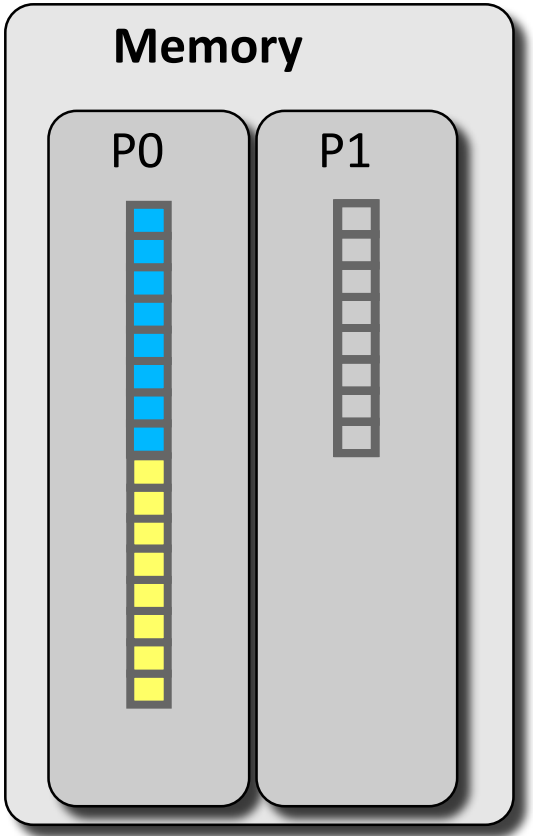
- MPI has a number of predefined datatypes to represent data
- Each C or Fortran datatype has a corresponding MPI datatype
  - C examples: MPI_INT for int and MPI_DOUBLE for double
  - Fortran example: MPI_INTEGER for integer
- One can also define custom datatypes

Case study: parallel sum

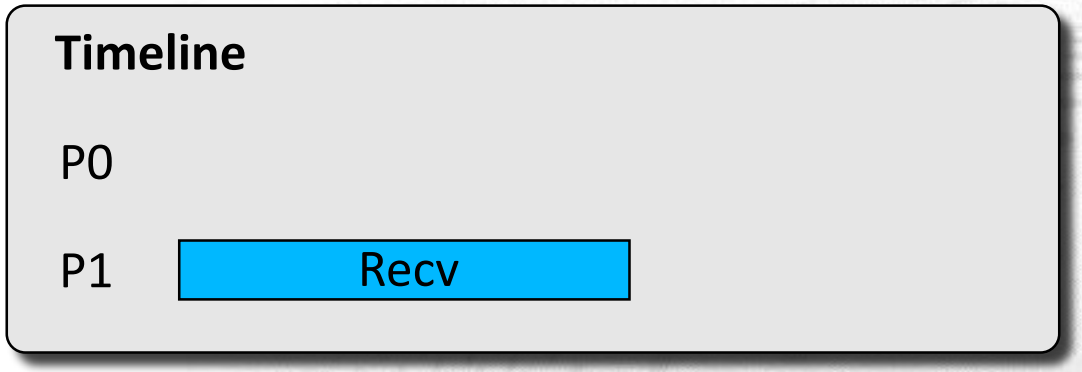


- Array originally on process #0 (P0)
- Parallel algorithm
  - Scatter
    - Half of the array is sent to process 1
  - Compute
    - P0 & P1 sum independently their segments
  - Reduction
    - Partial sum on P1 sent to P0
    - P0 sums the partial sums

Case study: parallel sum

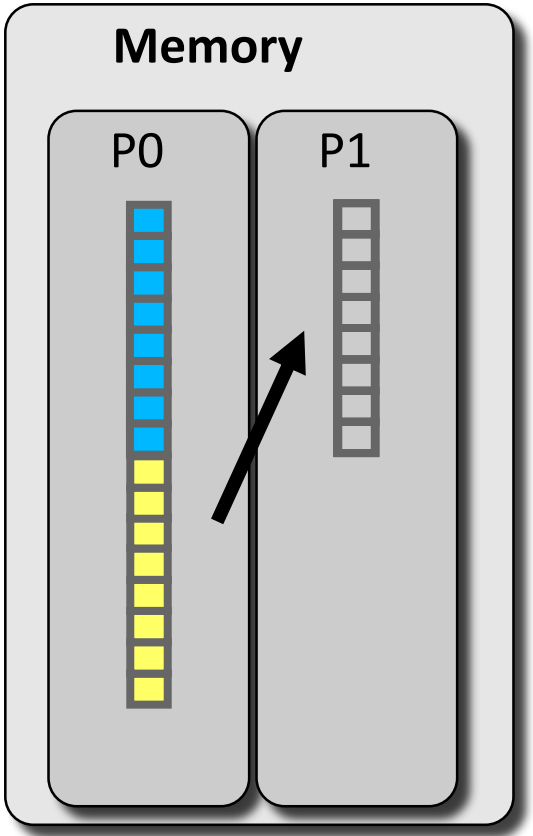


Step 1.1: Receive operation in scatter

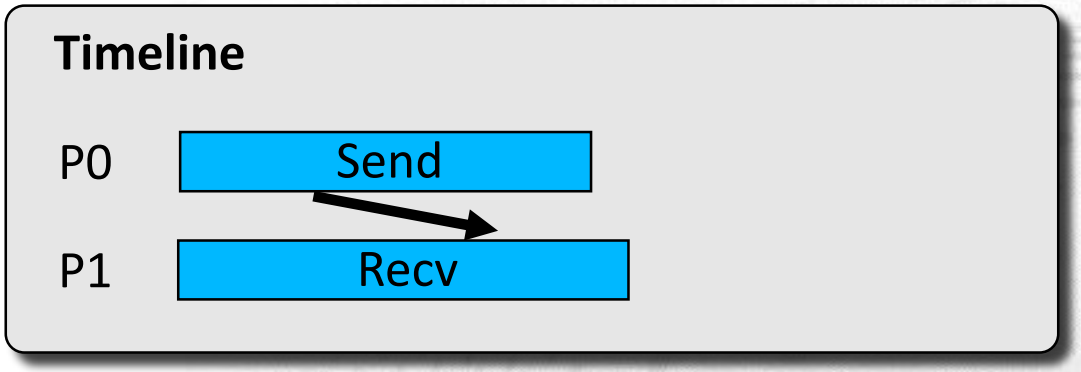


P1 posts a receive to receive half of the array from P0

Case study: parallel sum

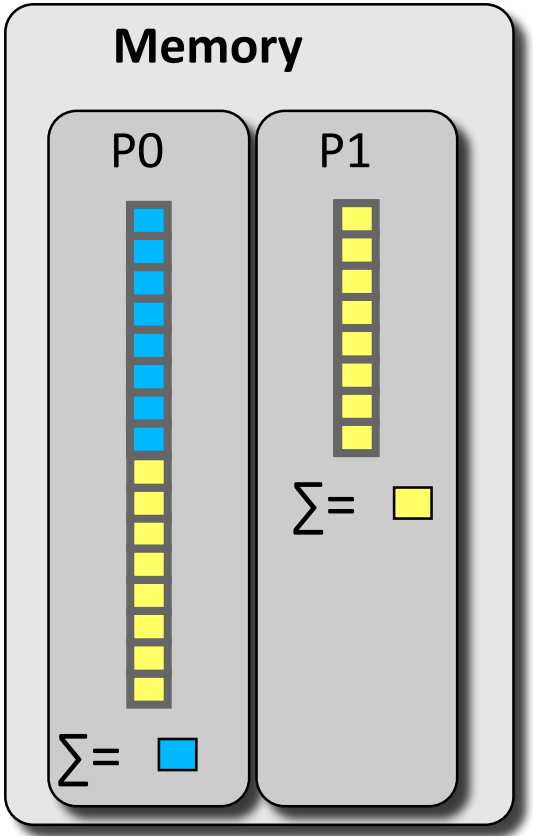


Step 1.2: Send operation in scatter

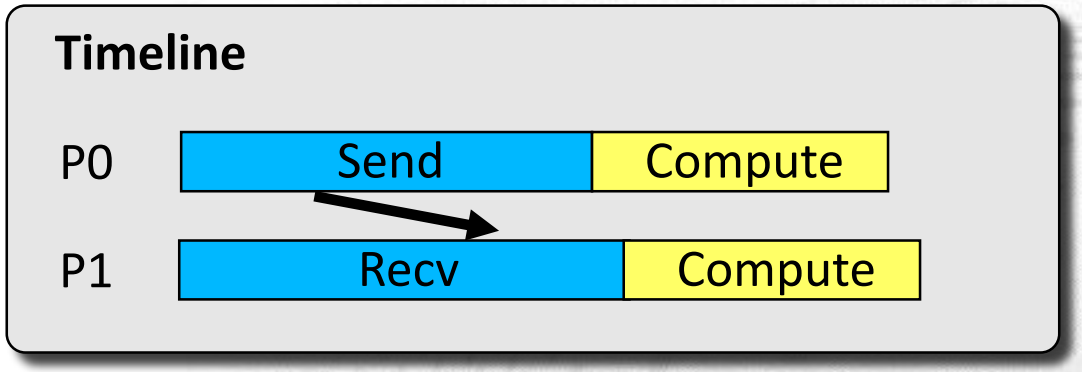


P0 posts a send to send the lower part of the array to P1

Case study: parallel sum

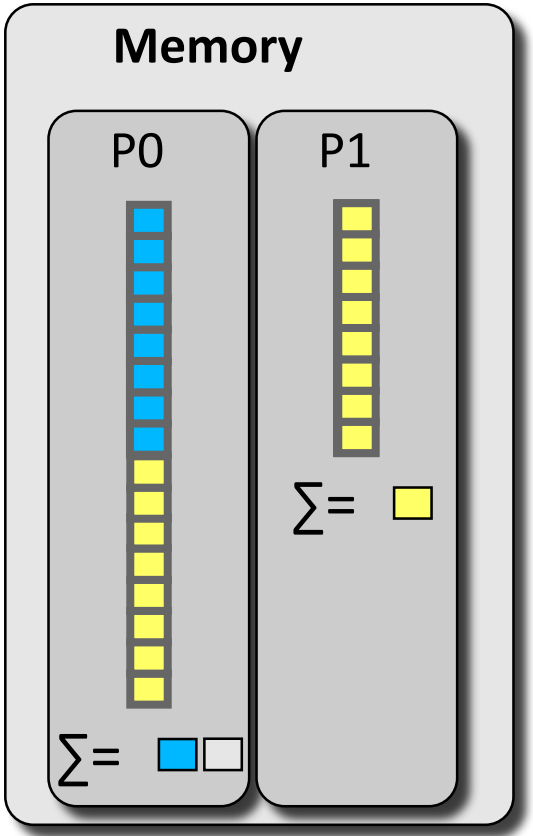


Step 2: Compute the sum in parallel

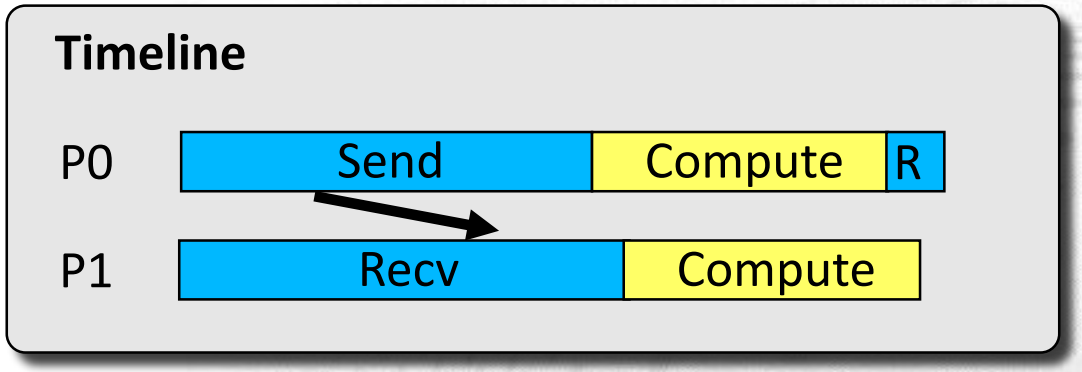


P0 & P1 computes their parallel sums and store them locally

Case study: parallel sum

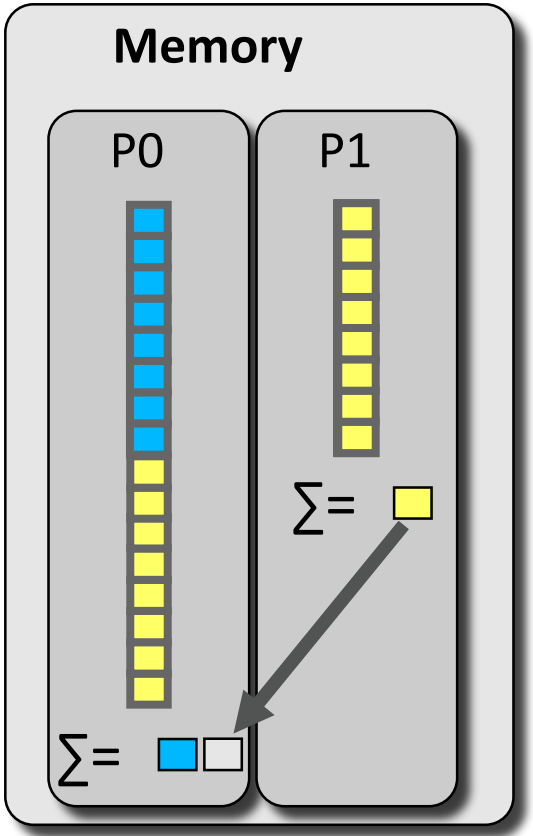


Step 3.1: Receive operation in reduction

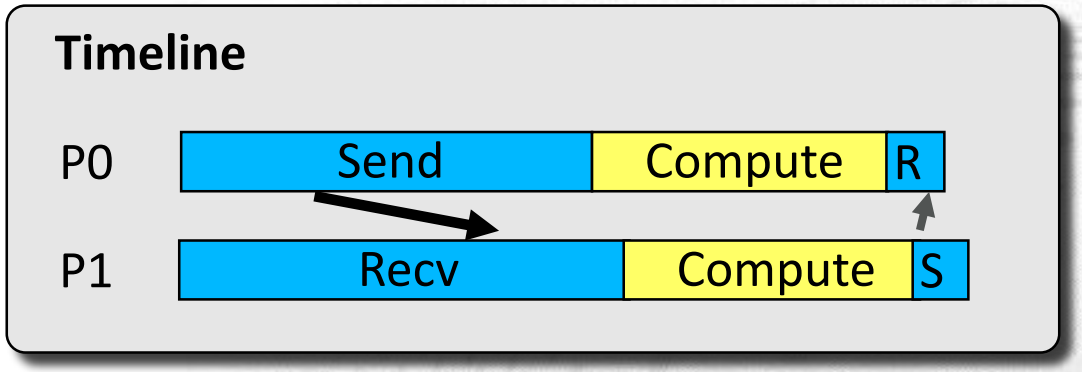


P0 posts a receive to receive partial sum

Case study: parallel sum

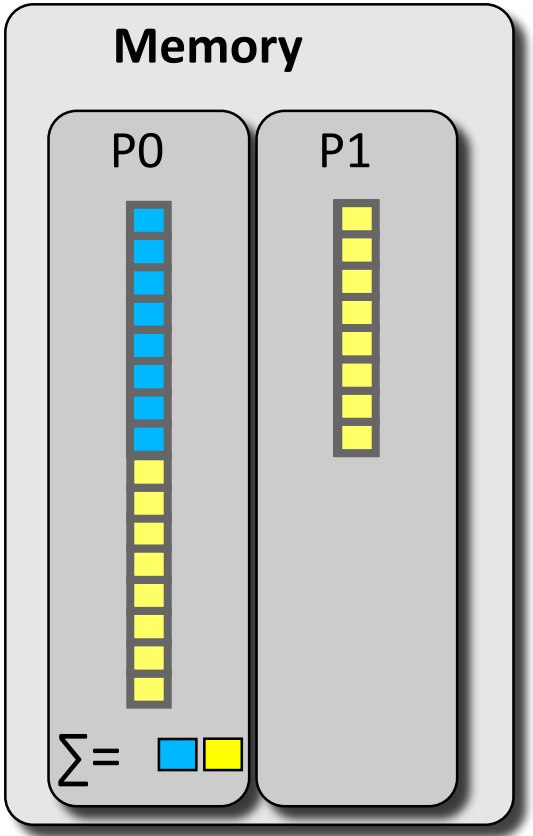


Step 3.2: send operation in reduction

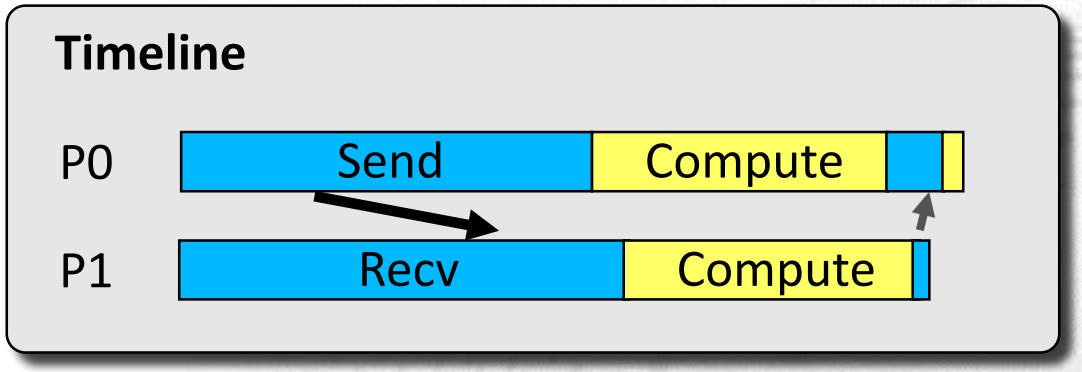


P1 posts a send with partial sum

Case study: parallel sum



Step 4: Compute final answer



P0 sums the partial sums

MORE ABOUT POINT-TO-POINT COMMUNICATION

Blocking routines & deadlocks

- Blocking routines
  - Completion depends on other processes
  - Risk for deadlocks – the program is stuck forever
- MPI_Send exits once the send buffer can be safely read and written to
- MPI_Recv exits once it has received the message in the receive buffer

Point-to-point communication patterns

Pairwise exchange

Pipe, a ring of processes exchanging data

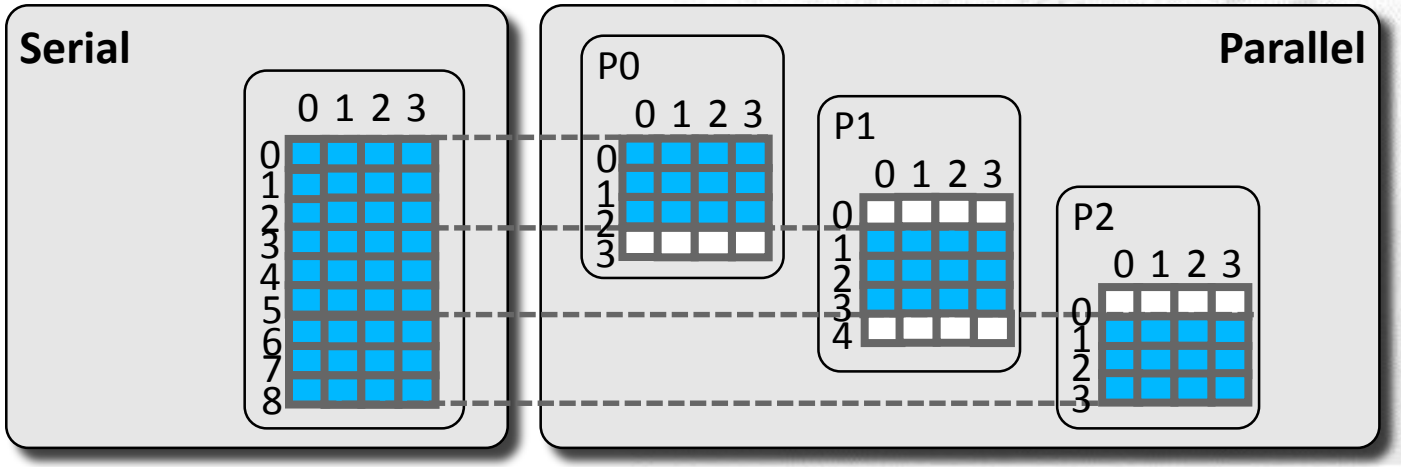
Combined send & receive

```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest,
   sendtag, recvbuf, recvcount, recvtype, source,
   recvtag, comm, status)
```

- Parameters as for MPI_Send and MPI_Recv combined
- Sends one message and receives another one, with one single command
  - Reduces risk for deadlocks
- Destination rank and source rank can be same or different

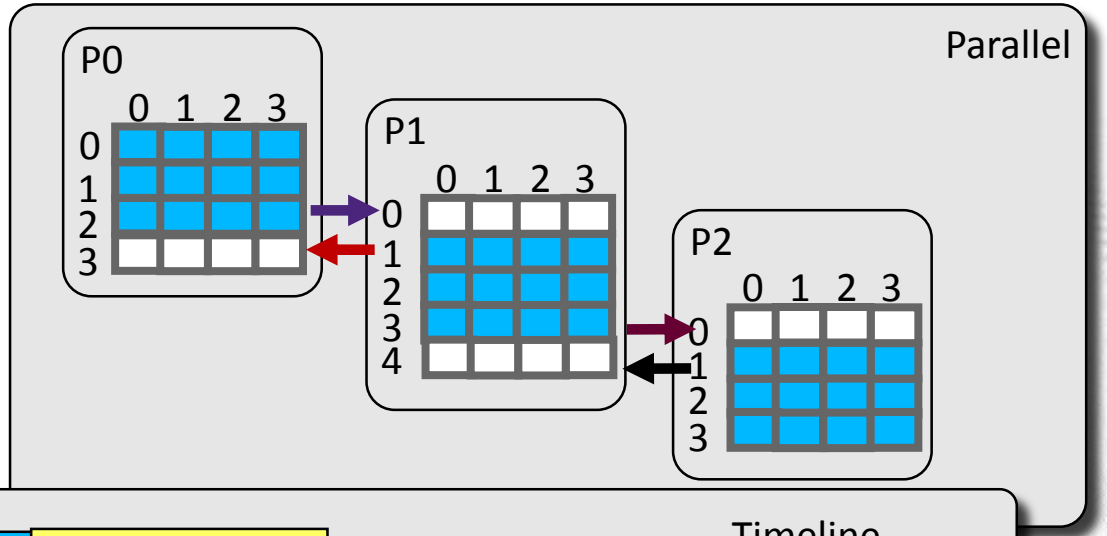
Case study 2: Domain decomposition

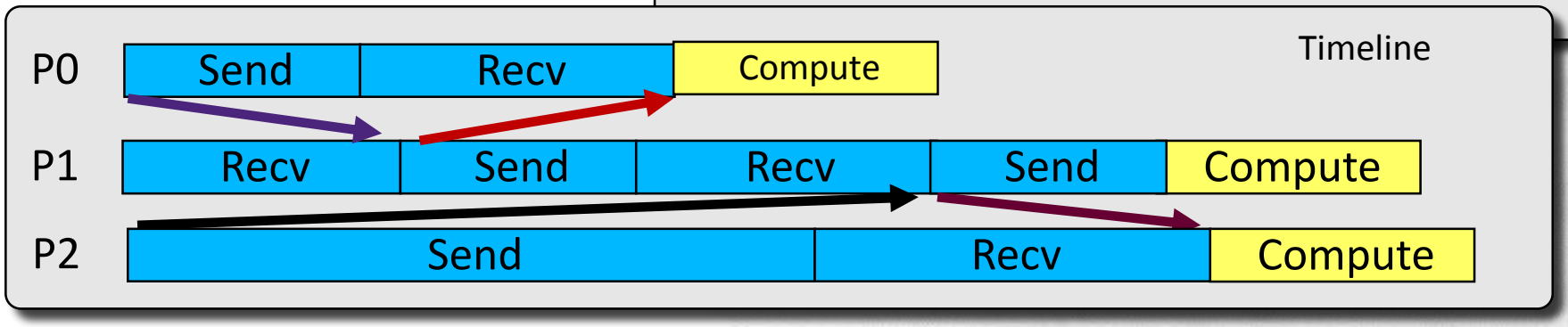
- Computation inside each domain can be carried out independently; hence in parallel
- *Ghost layer* at boundary represent the value of the elements of the other process



Case study 2: One iteration step

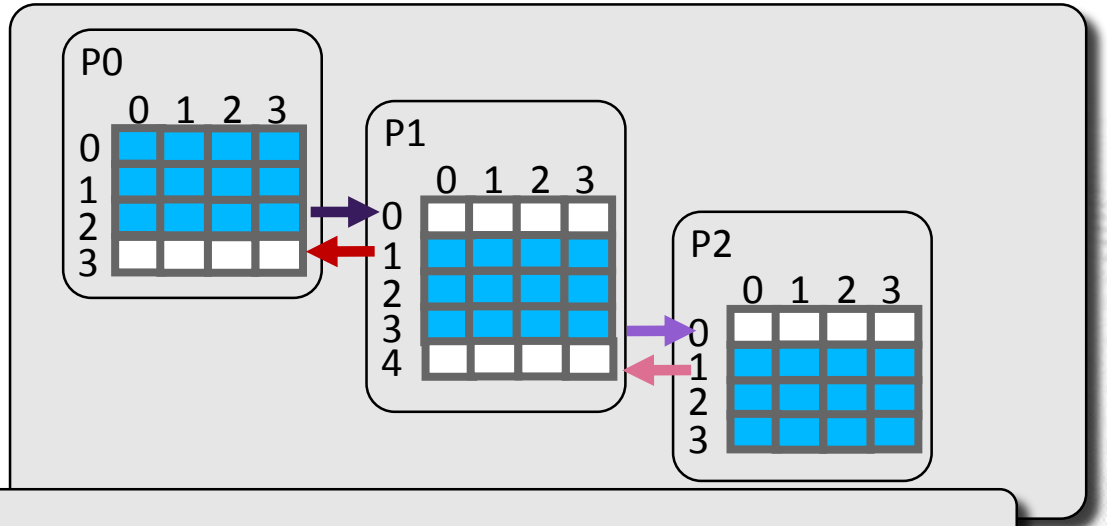
- Have to carefully schedule the order of sends and receives in order to avoid deadlocks

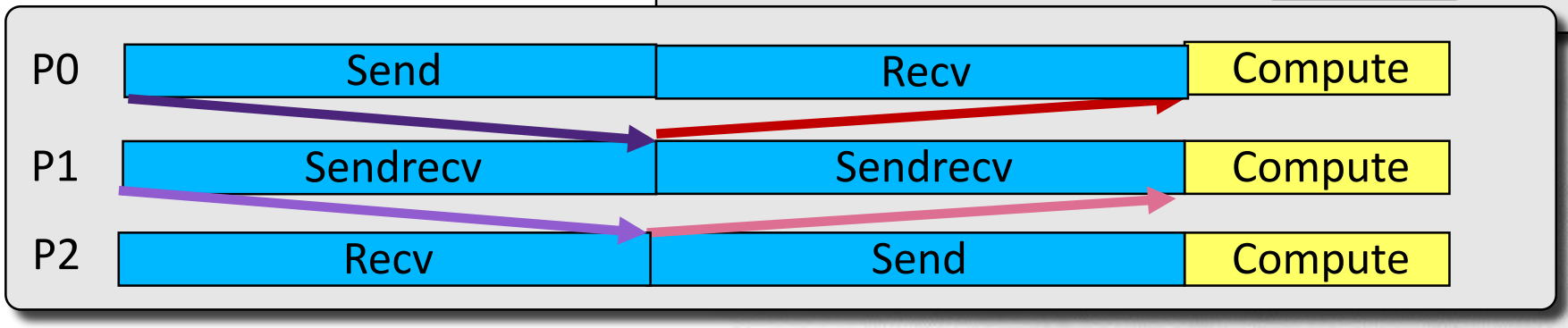




Case study 2: MPI_Sendrecv

- MPI_Sendrecv
  - Sends and receives with one command
  - No risk of deadlocks





Special parameter values

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

parameter	*value*	*function*
dest	*MPI_PROC_NULL*	*Null destination, no operation takes place*
comm	*MPI_COMM_WORLD*	*Includes all processes*
error	*MPI_SUCCESS*	*Operation successful*

Special parameter values

```
MPI_Recv(buf, count, datatype, source, tag, comm,
   status)
```

parameter	*value*	*function*
source	*MPI_PROC_NULL*	*No sender, no operation takes place*
	MPI_ANY_SOURCE	*Receive from any sender*
tag	*MPI_ANY_TAG*	*Receive messages with any tag*
comm	*MPI_COMM_WORLD*	*Includes all processes*
status	*MPI_STATUS_IGNORE*	*Do not store any status data*
error	*MPI_SUCCESS*	*Operation successful*

Status parameter

- The status *parameter* in MPI_Recv contains information on how the receive succeeded
  - Number and datatype of received elements
  - Tag of the received message
  - Rank of the sender
- In C the status parameter is a struct, in Fortran it is an integer array

Status parameter

- Received elements

  Use the function

  `MPI_Get_count(status, datatype, count)`

- Tag of the received message

  C: **`status.MPI_TAG`**

  Fortran: **`status(MPI_TAG)`**

- Rank of the sender

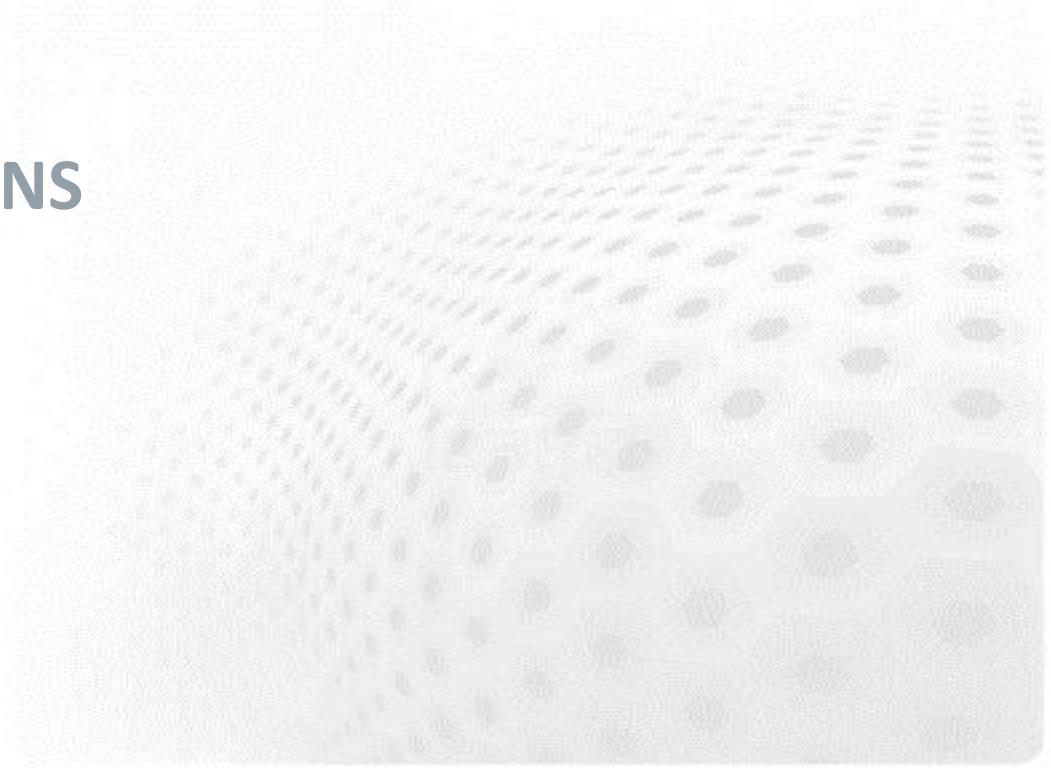
  C: **`status.MPI_SOURCE`**

  Fortran: **`status(MPI_SOURCE)`**

Summary

- Point-to-point communication
  - Messages are sent between two processes
- We discussed send and receive operations enabling any parallel application
  - MPI_Send & MPI_Recv
  - MPI_Sendrecv
- Special argument values
- Status parameter

COLLECTIVE OPERATIONS



Outline

- Introduction to collective communication
- One-to-many collective operations
- Many-to-one collective operations
- Many-to-many collective operations
- Non-blocking collective operations
- User-defined communicators

Introduction

- Collective communication transmits data among all processes in a process group
  - These routines must be called by all the processes in the group
- Collective communication includes
  - data movement
  - collective computation
  - synchronization

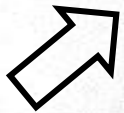
Example
MPI_Barrier
makes each task hold until all tasks have called it

```
int MPI_Barrier(comm)
MPI_BARRIER(comm, rc)
```

Introduction

- Collective communication outperforms normally point-to-point communication
- Code becomes more compact and easier to read:

```
if (my_id == 0) then
  do i = 1, ntasks-1
    call mpi_send(a, 1048576, &
        MPI_REAL, i, tag, &
        MPI_COMM_WORLD, rc)
  end do
else
  call mpi_recv(a, 1048576, &
      MPI_REAL, 0, tag, &
      MPI_COMM_WORLD, status, rc)
end if
```



```
call mpi_bcast(a, 1048576, &
    MPI_REAL, 0, &
    MPI_COMM_WORLD, rc)
```

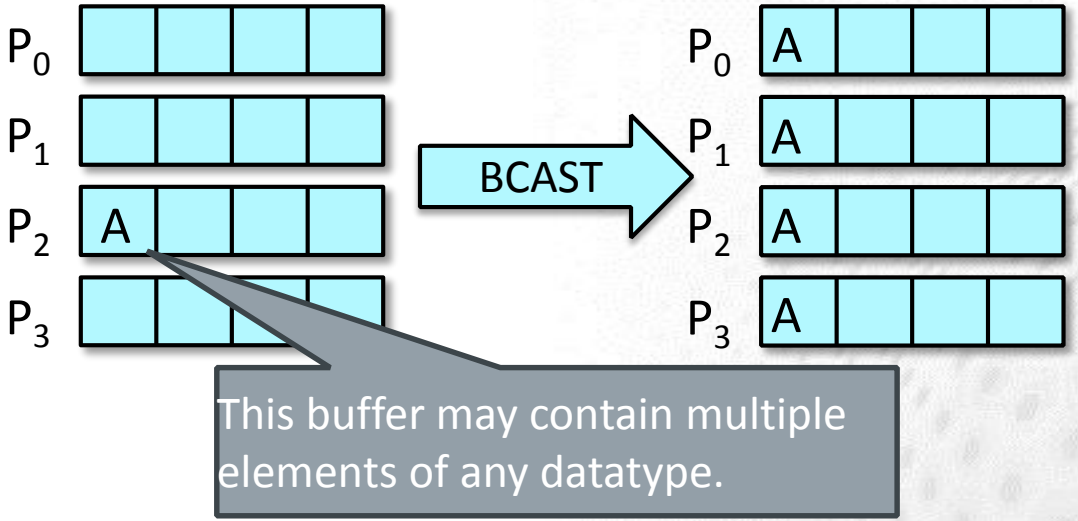
Communicating a vector a consisting of 1M float elements from the task 0 to all other tasks

Introduction

- Amount of sent and received data must match
- Non-blocking routines are available in the MPI 3 standard
  - Older libraries do not support this feature
- No tag arguments
  - Order of execution must coincide across processes

Broadcasting

- Send the same data from one process to all the other



Broadcasting

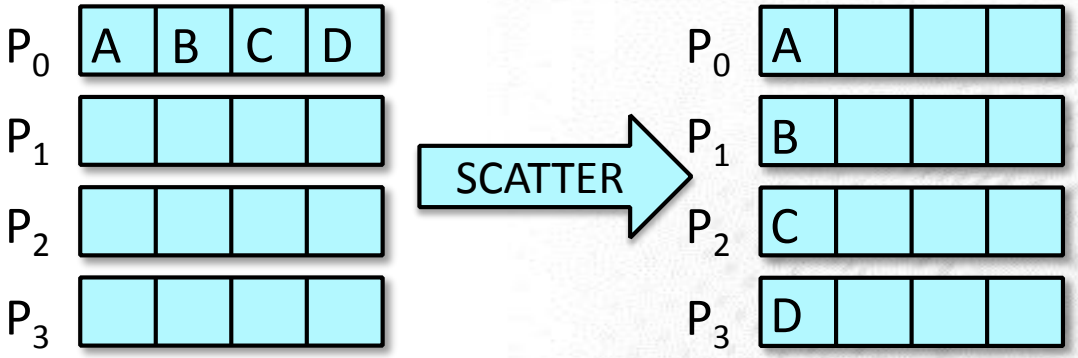
- With MPI_Bcast, the task *root* sends a *buffer* of data to all other tasks

```
MPI_Bcast(buffer, count, datatype, root, comm)
```

buffer	data to be distributed
count	number of entries in buffer
datatype	data type of buffer
root	rank of broadcast root
comm	communicator

Scattering

- Send equal amount of data from one process to others



- Segments A, B, … may contain multiple elements

Scattering

- MPI_Scatter: Task *root* sends an equal share of data (*sendbuf*) to all other processes

```
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf,
            recvcount, recvtype, root, comm)
```

sendbuf	send buffer (data to be scattered)
sendcount	number of elements sent to each process
sendtype	data type of send buffer elements
recvbuf	receive buffer
recvcount	number of elements in receive buffer
recvtype	data type of receive buffer elements
root	rank of sending process
comm	communicator

One-to-all example

```
if (my_id==0) then
  do i = 1, 16
    a(i) = i
  end do
end if

call mpi_bcast(a,16,MPI_INTEGER,0, &
               MPI_COMM_WORLD,rc)
if (my_id==3) print *, a(:)
```

```
if (my_id==0) then
  do i = 1, 16
    a(i) = i
  end do
end if
call mpi_scatter(a,4,MPI_INTEGER, &
                 aloc,4,MPI_INTEGER, &
                 0,MPI_COMM_WORLD,rc)
if (my_id==3) print *, aloc(:)
```

Assume 4 MPI tasks. What would the (full) program print?

```
A.  1 2 3 4
B.  13 14 15 16
C.  1 2 3 4
    5 6 7 8
    9 10 11 12
    13 14 15 16
```

```
A.  1 2 3 4
B.  13 14 15 16
C.  1 2 3 4
    5 6 7 8
    9 10 11 12
    13 14 15 16
```

Varying-sized scatter

- Like MPI_Scatter, but messages can have different sizes and displacements

```
MPI_Scatterv(sendbuf, sendcounts, displs, sendtype,
             recvbuf, recvcount, recvtype, root, comm)
```

sendbuf	send buffer
sendcounts	array (of length ntasks) specifying the number of elements to send to each processor
displs	array (of length ntasks). Entry i specifies the displacement (relative to sendbuf)
sendtype	data type of send buffer elements
recvbuf	receive buffer
recvcount	number of elements in receive buffer
recvtype	data type of receive buffer elements
root	rank of sending process
comm	communicator

Scatterv example

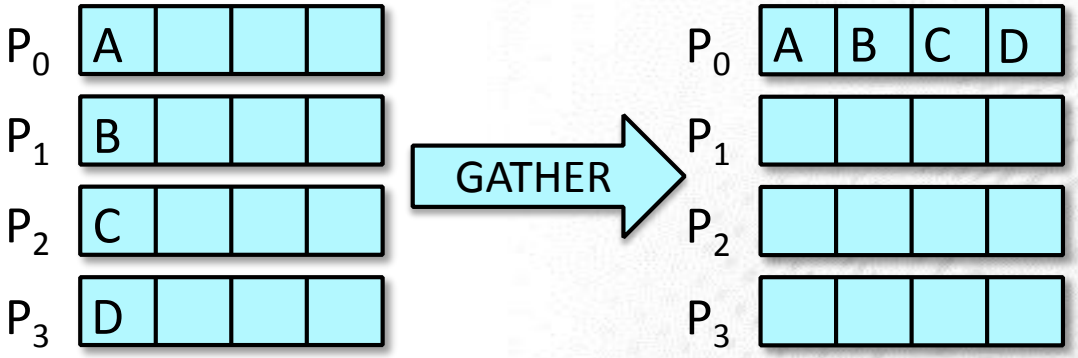
```
if (my_id==0) then
  do i = 1, 10
    a(i) = i
  end do
  sendcnts = (/ 1, 2, 3, 4 /)
  displs = (/ 0, 1, 3, 6 /)
end if

call mpi_scatterv(a, sendcnts, &
    displs, MPI_INTEGER,&
    aloc, 4, MPI_INTEGER, &
    0, MPI_COMM_WORLD, rc)
```

Assume 4 MPI tasks. What are the values in aloc in the last task (#3)?

A. 1 2 3
B. 7 8 9 10
C. 1 2 3 4 5 6 7 8 9 10

Gathering

- Collect data from all the process to one process



- Segments A, B, … may contain multiple elements

Gathering

- MPI_Gather: Collect equal share of data (in *sendbuf*) from all processes to *root*

```
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf,
           recvcount, recvtype, root, comm)
```

sendbuf	send buffer (data to be gathered)
sendcount	number of elements pulled from each process
sendtype	data type of send buffer elements
recvbuf	receive buffer
recvcount	number of elements in any **single** receive
recvtype	data type of receive buffer elements
root	rank of receiving process
comm	communicator

Varying-sized gather

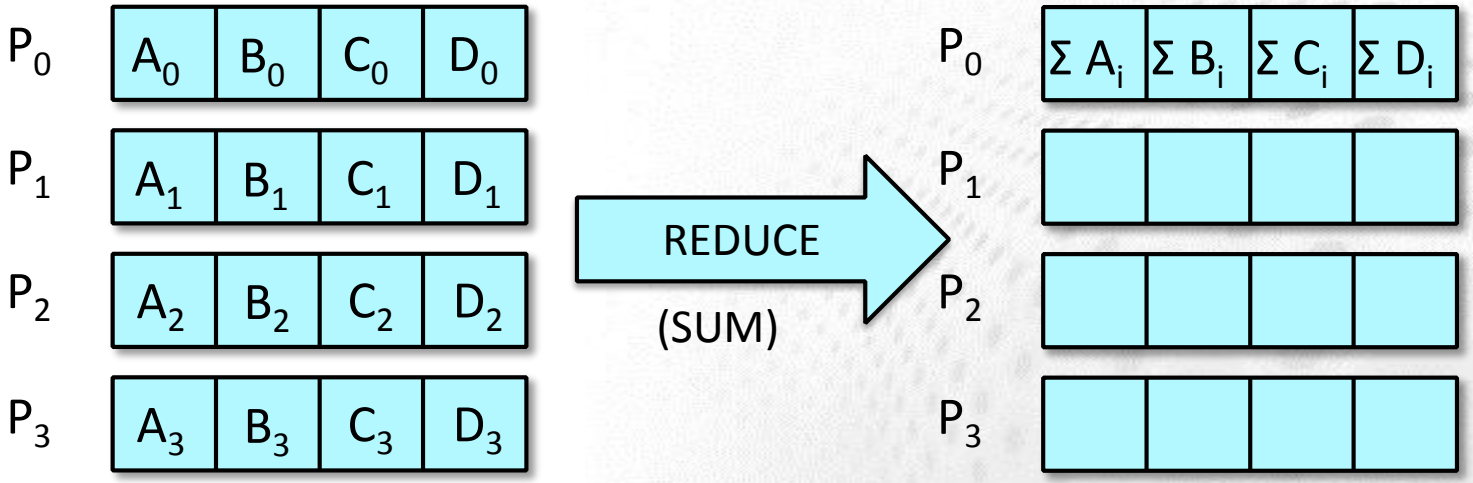
- Like MPI_Gather, but messages can have different sizes and displacements

```
MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf,
            recvcounts, displs, recvtype, root, comm)
```

sendbuf	send buffer
sendcount	the number of elements to send
sendtype	data type of send buffer elements
recvbuf	receive buffer
recvcounts	array (of length ntasks). Entry i specifies how many to receive from that process
displs	array relative to recvcounts, displacement in recvbuf
recvtype	data type of receive buffer elements
root	rank of receiving process
comm	communicator

Reduce operation

- Applies an operation over set of processes and places result in single process



Reduce operation

- Applies a reduction operation *op* to *sendbuf* over the set of tasks and places the result in *recvbuf* on *root*

```
MPI_Reduce(sendbuf, recvbuf, count, datatype, op,
           root, comm)
```

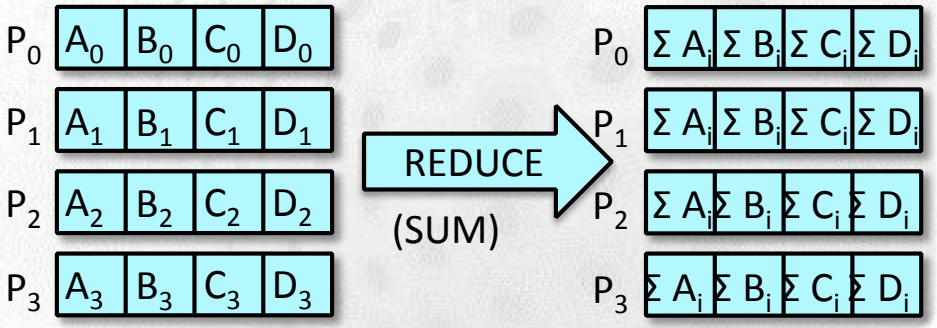
sendbuf	send buffer
recvbuf	receive buffer
count	number of elements in send buffer
datatype	data type of elements in send buffer
op	operation
root	rank of root process
comm	communicator

Global reduce operation

- MPI_Allreduce combines values from all processes and distributes the result back to all processes
  - Compare: MPI_Reduce + MPI_Bcast

```
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm)
```

sendbuf starting address of send buffer
recvbuf starting address of receive buffer
count number of elements in send buffer
datatype data type of elements in send buffer
op operation
comm communicator



Allreduce example: parallel dot product

```
> aprun -n 8 ./mpi_pdot
 id= 6 local= 39.68326  global= 338.8004
 id= 7 local= 39.34439  global= 338.8004
 id= 1 local= 42.86630  global= 338.8004
 id= 3 local= 44.16300  global= 338.8004
 id= 5 local= 39.76367  global= 338.8004
 id= 0 local= 42.85532  global= 338.8004
 id= 2 local= 40.67361  global= 338.8004
 id= 4 local= 49.45086  global= 338.8004
```

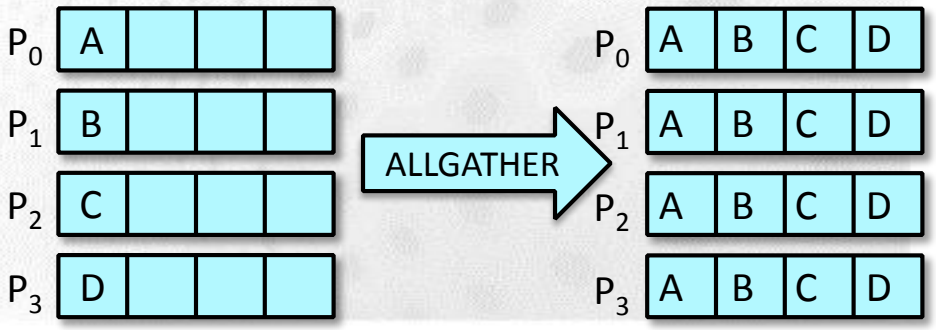
```
real :: a(1024), aloc(128)
...
if (my_id==0) then
    call random_number(a)
end if
call mpi_scatter(a, 128, MPI_INTEGER, &
                 aloc, 128, MPI_INTEGER, &
                 0, MPI_COMM_WORLD, rc)
rloc = dot_product(aloc,aloc)
call mpi_allreduce(rloc, r, 1, MPI_REAL,&
                   MPI_SUM, MPI_COMM_WORLD,
                   rc)
```

All-to-one plus one-to-all

- MPI_Allgather gathers data from each task and distributes the resulting data to each task
  - Compare: MPI_Gather + MPI_Bcast

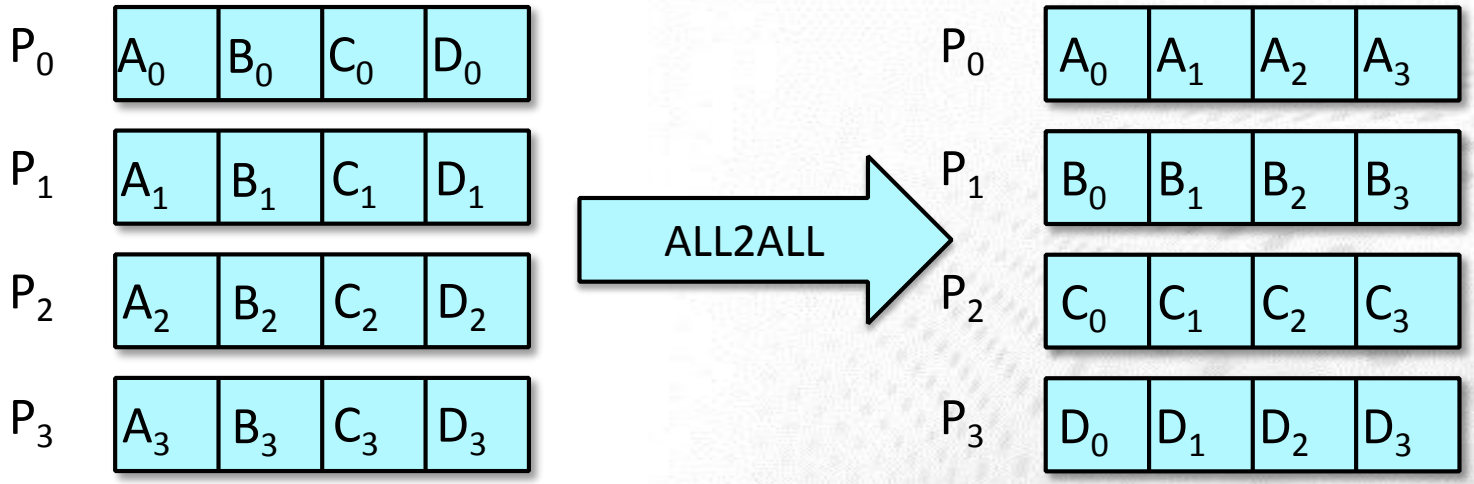
```
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf,
recvcount, recvtype, comm)
```

sendbuf	send buffer
sendcount	number of elements in send buffer
sendtype	data type of send buffer elements
recvbuf	receive buffer
recvcount	number of elements received from any process
recvtype	data type of receive buffer



From each to every

- Send a distinct message from each task to every task



- ”Transpose” like operation

From each to every

- MPI_Alltoall sends a distinct message from each task to every task
  - Compare: “All scatter”

```
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf,
             recvcount, recvtype, comm)
```

sendbuf	send buffer
sendcount	number of elements to send to each process
sendtype	data type of send buffer elements
recvbuf	receive buffer
recvcount	number of elements received from any process
recvtype	data type of receive buffer elements
comm	communicator

All-to-all example

```
if (my_id==0) then
  do i = 1, 16
    a(i) = i
  end do
end if
call mpi_bcast(a, 16, MPI_INTEGER, 0, &
    MPI_COMM_WORLD, rc)

call mpi_alltoall(a, 4, MPI_INTEGER, &
                  aloc, 4, MPI_INTEGER, &
                  MPI_COMM_WORLD, rc)
```

Assume 4 MPI tasks. What will be the values of aloc in the process #0?

A. 1, 2, 3, 4
B. 1,...,16
C. 1, 2, 3, 4, 1, 2, 3, 4,
   1, 2, 3, 4, 1, 2, 3, 4

Common mistakes with collectives

- ✗ Using a collective operation within one branch of an if-test of the rank

```
IF (my_id == 0) CALL MPI_BCAST(...
```

  - All processes, both the root (the sender or the gatherer) and the rest (receivers or senders), *must* call the collective routine!
- ✗ Assuming that all processes making a collective call would complete at the same time
- ✗ Using the input buffer as the output buffer

```
CALL MPI_ALLREDUCE(a, a, n, MPI_REAL, MPI_SUM, ...
```

Summary

- Collective communications involve all the processes within a communicator
  - All processes must call them
- Collective operations make code more transparent and compact
- Collective routines allow optimizations by MPI library
- Performance consideration:
  - Alltoall is expensive operation, avoid it when possible

USER-DEFINED COMMUNICATORS

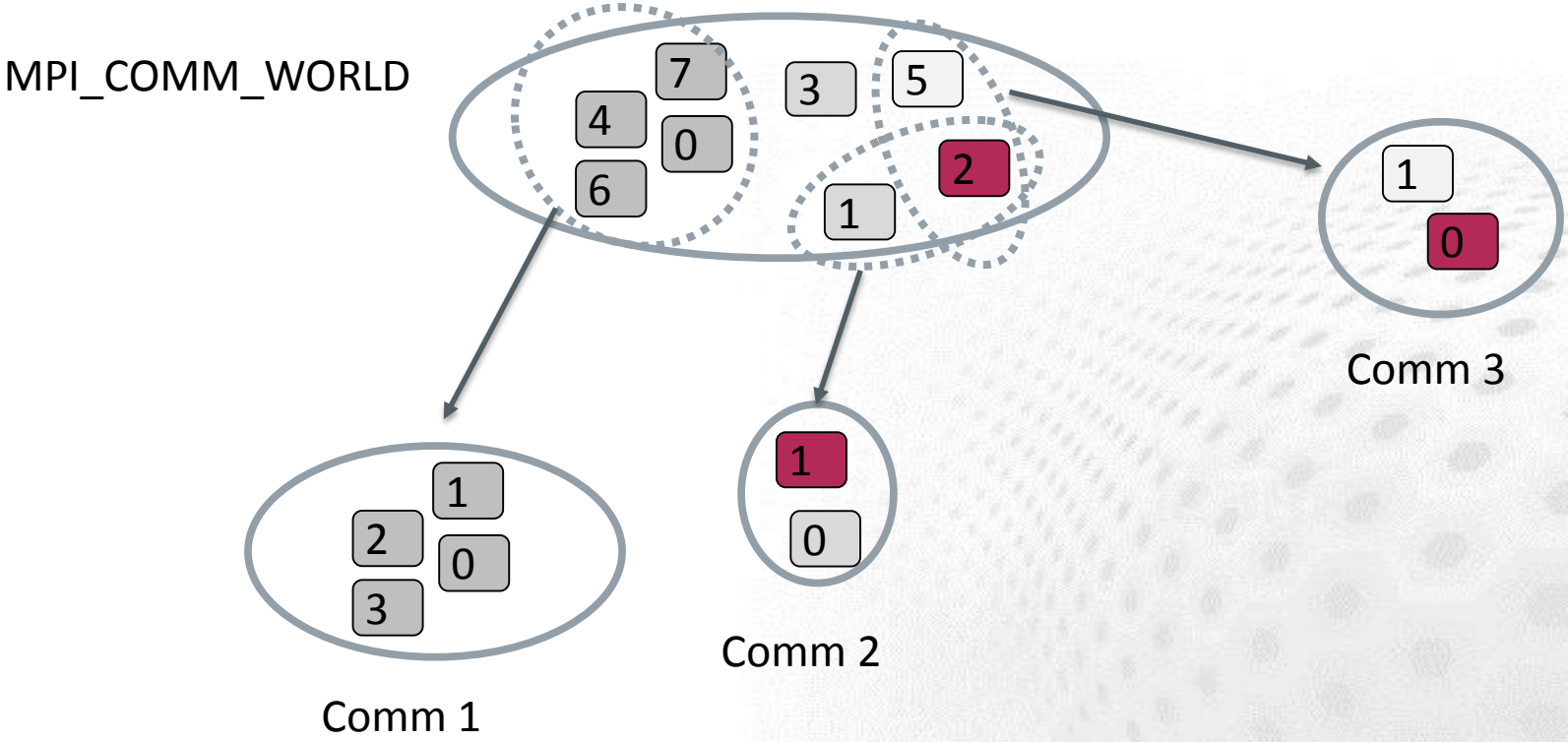
Communicators

- The communicator determines the "communication universe"
  - The source and destination of a message is identified by process rank within the communicator
- So far: MPI_COMM_WORLD
- Processes can be divided into subcommunicators
  - Task level parallelism with process groups performing separate tasks
  - Parallel I/O

Communicators

- Communicators are dynamic
- A task can belong simultaneously to several communicators
  - In each of them it has a unique ID, however
  - Communication is normally within the communicator

Grouping processes in communicators



Creating a communicator

- MPI_Comm_split creates new communicators based on 'colors' and 'keys'

```
MPI_Comm_split(comm, color, key, newcomm)
```

comm	communicator handle
color	control of subset assignment, processes with the same color belong to the same new communicator
key	control of rank assignment
newcomm	new communicator handle

If color = `MPI_UNDEFINED`, a process does not belong to any of the new communicators

Creating a communicator

```
if (myid%2 == 0) {
  color = 1;
} else {
  color = 2;
}
MPI_Comm_split(MPI_COMM_WORLD, color, myid, &subcomm);
MPI_Comm_rank(subcomm, &mysubid);
printf ("I am rank %d in MPI_COMM_WORLD, but %d in
         Comm %d.\n", myid, mysubid, color);
```

```
I am rank 2 in MPI_COMM_WORLD, but 1 in Comm 1.
I am rank 7 in MPI_COMM_WORLD, but 3 in Comm 2.
I am rank 0 in MPI_COMM_WORLD, but 0 in Comm 1.
I am rank 4 in MPI_COMM_WORLD, but 2 in Comm 1.
I am rank 6 in MPI_COMM_WORLD, but 3 in Comm 1.
I am rank 3 in MPI_COMM_WORLD, but 1 in Comm 2.
I am rank 5 in MPI_COMM_WORLD, but 2 in Comm 2.
I am rank 1 in MPI_COMM_WORLD, but 0 in Comm 2.
```

Communicator manipulation

`MPI_Comm_size`	Returns number of processes in communicator's group
`MPI_Comm_rank`	Returns rank of calling process in communicator's group
`MPI_Comm_compare`	Compares two communicators
`MPI_Comm_dup`	Duplicates a communicator
`MPI_Comm_free`	Marks a communicator for deallocation

Basic MPI summary