Intro to MPI programming (II)

Lab session 2

Blocking and non-blocking

Q: When is a SEND instruction complete?

A: When it is safe to change the data that we sent.

Q: When is a RECEIVE instruction complete?

A: When it is safe to access the data we received.

With both communications (send and receive) we have two choices:

- Start a communication and wait for it to complete: BLOCKING approach
- Start a communication and return control to the main program: NON-BLOCKING approach

The Non-Blocking approach REQUIRES us to check for completion before we can modify/access the sent/received data!!!

Pros and cons of non-blocking send/receive

- Non-Blocking communications allows the separation between the initiation of the communication and the completion.
- Advantages:
  - between the initiation and completion the program could do other useful computation (latency hiding).
- Disadvantages:
  - the programmer has to insert code to check for completion.

Communication mode

- 4 different send types:
  - Standard: let MPI decide the best strategy...
  - Synchronous: it is complete when the receiver acknowledged the reception of the message
  - Buffered: it is complete when the data has been copied to a local buffer
  - Ready: requires a receiver to be already waiting for the message
- 1 single receive type

Communication modes and MPI subroutines

Mode	Completion condition	Blocking	Non-blocking
Standard send	Message sent (receive state unknown)	MPI_SEND	MPI_ISEND
Receive	Completes when message arrives	MPI_RECV	MPI_IRECV
Synchronous send	Only completes when the receive has completed	MPI_SSEND	MPI_ISSEND
Buffered send	Always completes, regardless of receiver	MPI_BSEND	MPI_IBSEND
Ready send	Always completes, regardless of receiver	MPI_RSEND	MPI_IRSEND

Waiting and testing for completion

int MPI_Wait(MPI_Request *req, MPI_Status *status);

A call to this subroutine cause the code to wait until the communication pointed by req is complete.

Req INTEGER) input/output, identifier associated to a communications event (initiated by **MPI_ISEND** or **MPI_IRECV**).

Status (INTEGER) array of size

MPI_STATUS_SIZE, if **req** was associated to a call to **MPI_IRECV**, **status** contains informations on the received message, otherwise **status** could contain an error code.

ierr (INTEGER) output, error code (if **ierr=0** no error occours).

Collective operations

- Collective routines provide a higher-level way to organize a parallel program
- Each process executes the same communication operations
- MPI provides a rich set of collective operations…

Collective operations (II)

- Communications involving group of processes in a communicator.
- Groups and communicators can be constructed “by hand” or using topology routines.
- Tags are not used; different communicators deliver similar functionality.
- No non-blocking collective operations.
- Three classes of operations: synchronization, data movement, collective computation.

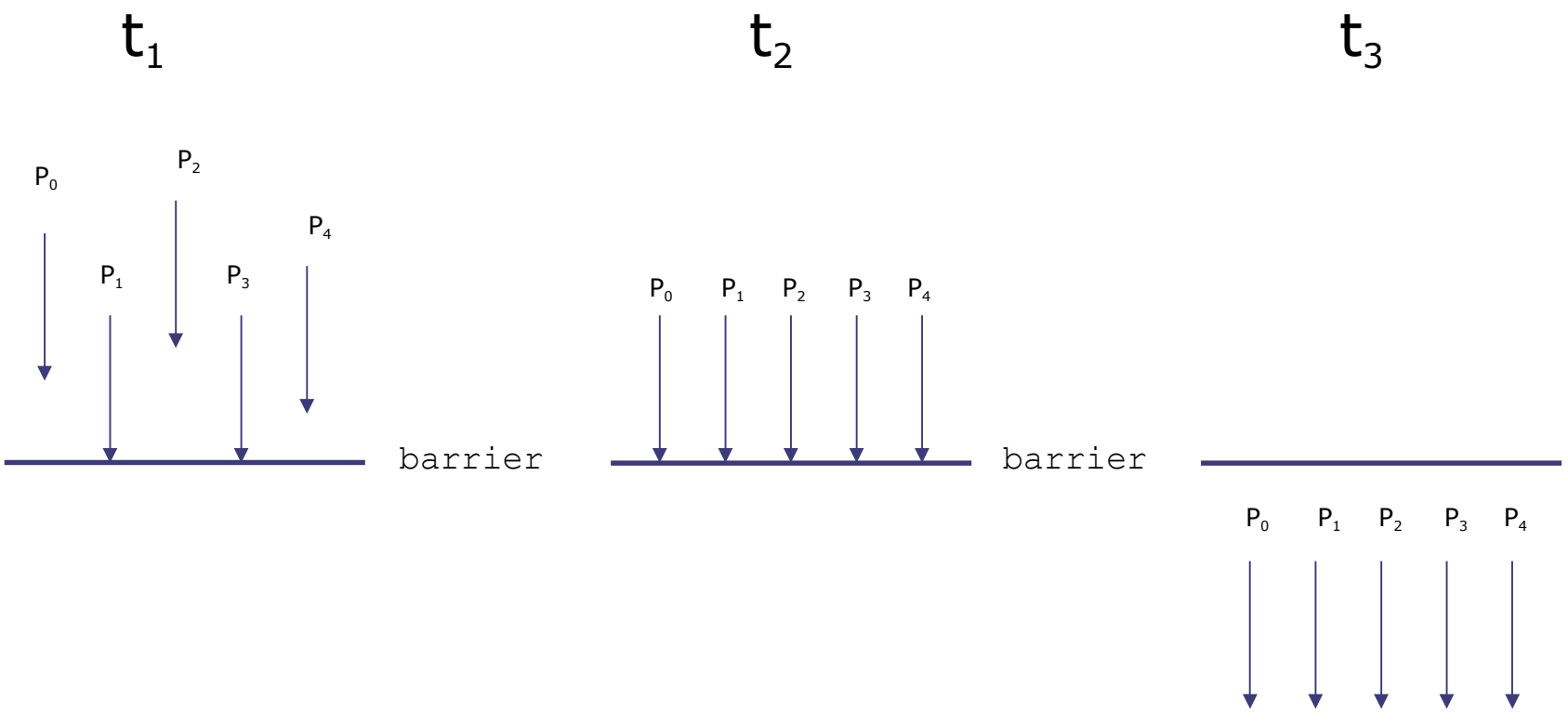
MPI_Barrier

- Stop processes until all processes within a communicator reach the barrier
- Almost never required in a parallel program

Occasionally useful in measuring performance and load balancing

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

Barrier



Boradcast: MPI_Bcast

- One-to-all communication: same data sent from root process to all others in the communicator
- All processes must specify same root, rank and communicator

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

Reduction

- The reduction operation allows to:
  - Collect data from each process
  - Reduce the data to a single value
  - Store the result on the root process
  - Store the result on all processes

Reduction primitives: MPI_Reduce & MPI_Allreduce

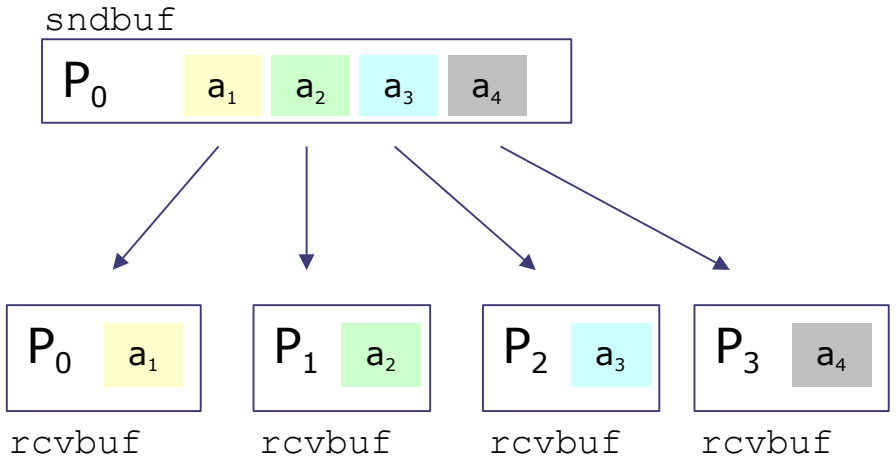
- snd_buf input array of type type containing local values.
- rcv_buf output array of type typecontaining global results
- count number of element of snd_buf and rcv_buf
- typeMPI type of snd_buf and rcv_buf
- op parallel operation to be performed
- rootMPI id of the process storing the result
- commcommunicator of processes involved in the operation
- ierr output, error code (if ierr=0no error occours)

Predefined reduction operations

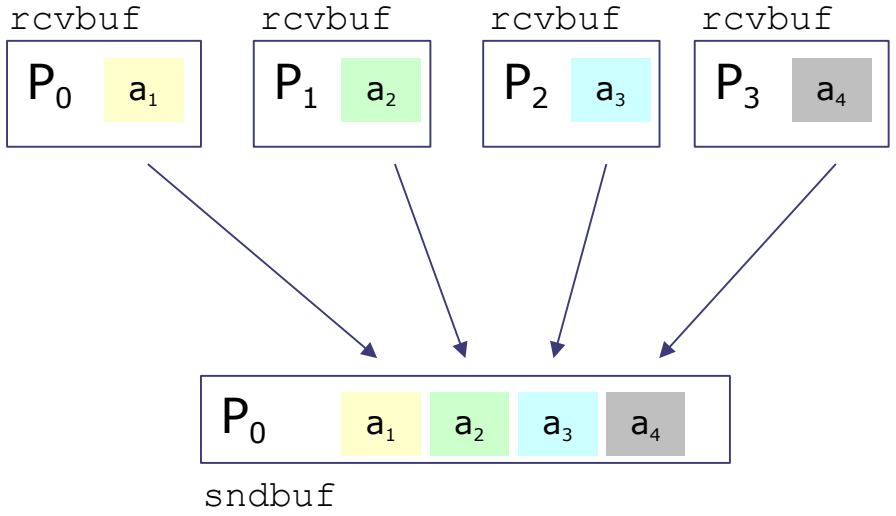
MPI op	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

Scatter-gather

Scatter



Gather



MPI_Scatter

- One-to-all communication: different data sent from root process to all others in the communicator
- Arguments definition are like other MPI subroutine but:
  - sndcount is the number of elements sent to each process, not the size of the send buffer
  - The sender arguments are significant only at root process

MPI_Scatter(sndbuf, sndcount, sndtype, rcvbuf, rcvcount, rcvtype, root, comm, ierr)

MPI_Gather

- One-to-all communication: different data colleted by the root process, from all others in the communicator.
- Arguments definition are like other MPI subroutine but:
  - rcvcount is the number of elements collected from each process, not the size of the receive buffer
  - The receiver arguments are significant only at root process

MPI_Gather(sndbuf, sndcount, sndtype, rcvbuf, rcvcount, rcvtype, root, comm, ierr)

Exercises

- Array decomposition: the master task initialize an array and distributes an equal portion of that array to the other tasks
  - Each task (including master) performs some kind of operation on the elements.
  - Each task also maintain a sum (for example) of the elements of their portion of the array.
  - Once finished, each task sends the updated portion of the array to the master.
  - The master collects the updated sub-arrays and the local sums of each process
- Download, compile and execute “bug” examples. Try to find what’s wrong in the source code.

Exercises

- Re-visit the MPI version of our ‘pi’ calculator, and use collective operations instead of a naïve communication algorithm.