MPI Introduction

The Cyprus Institute

CaSToRC

Outline

- **Overview of MPI**
- **Basics**
  - Concept
  - Starting up
  - Common initialization
- **Synchronization**
- **Collectives**
- **Point-to-point**
- **Advanced Issues**
  - Non-blocking communication
  - Mixing with OpenMP

The Message Passing Interface

- **MPI: An Application Programmer Interface (API)**
- **A *de facto* standard for programming distributed memory systems**
- **Current specification is version 2 (MPI-2)**
- **Several free (open) implementations, e.g.:**
  - Mvapich(2)
  - OpenMPI

The Library

- **Includes**
  - Function definitions, types, constants and macros for the MPI library are included in a single include file:

Fortran	C
`include “mpif.h”`	`#include <mpi.h>`

The Library

- **Compiling**
  - Compiling and linking is made easy with a wrapper-compiler which most implementations provide. Invocation is usually via:

Fortran	C
`mpif77 hello_world.f` `mpif90 hello_world.f90`	`mpicc hello_world.c`

Runtime

- **Running**
  - Running an MPI program is usually done via the `mpirun` or `mpiexec` wrapper scripts, which take care of initializing the appropriate environment for the parallel run:

Fortran and C
`mpirun -np 2 ./a.out`

Basic concepts

- **The distributed memory model**
  - Invocation of `mpirun` will run multiple instances of the *same* program in parallel
  - Without calls to MPI, all parallel instances will, ideally, run and terminate identically
  - With calls to MPI, one can:
    - Differentiate between parallel instances (i.e., give each instance, or *process,* a unique ID)
    - Synchronize processes
    - Send messages between processes

Exercises

- **General Notes**
  - The exercises are placed in `MPI_Intro/Exercises/`
  - There are eight directories: `Ex1/, …, Ex8/`
  - Each directory contains:
    - A Makefile
    - A job-script
    - A C and Fortran 90 sources code file (except for Ex8)
  - Load module `openmpi` before compiling
- **In Ex1, the job script is incomplete. In Ex2 to Ex8, the source codes are incomplete**
- **You can choose to carry out the exercise in either C or Fortran:**
  - `make C/F` will compile the C/Fortran program
  - `qsub ex[12345678].job` will submit the script to run your program

Exercise 1

- **In this exercise you will run a non-MPI program with `mpiexec`**
  - First of all, study the source code `main.c` or `main.f90`
  - Compile either the C version or the Fortran version with:
    - `make C` or
    - `make F`
  - Under `Ex1/`., read the comments in `ex1.job`
  - After completing the missing line in `ex1.job` submit the job
    - `qsub ex1.job`
  - Once run, the output will be in `Ex1.out`

Basics

- **Initialization**
  - All MPI programs must begin with a call to `MPI_Init()` and close with a call to `MPI_Finalize()`.

Fortran	C
`CALL MPI_INIT(IERROR)` `CALL MPI_FINALIZE(IERROR)`	`ierror = MPI_Init(&argc, &argv);` `ierror = MPI_Finalize();`

  - In Fortran, an integer error code is always passed as the last argument
  - In C, this integer error code is the return value
  - Not invoking `MPI_Finalize()` at the end may raise a false error
  - In C, the command line arguments must be passed to `MPI_Init()`.

Basics

- **Size and rank**
  - Get how many processes are running in a given *communicator*, and the rank of the calling process within that communicator.

Fortran	C
`CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NPROC, IERR)` `CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, IERR)`	`ierr = MPI_Comm_size(MPI_COMM_WORLD, &nproc);` `ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);`

  - The communicator `MPI_COMM_WORLD` is set to contain all processes available, after invocation of `MPI_Init()`
  - The integer `nproc` will be the number of processes within the communicator (should be the same as what was specified with `mpirun`)
  - Note that in C pointers to integers are passed
  - Here, `MPI_Comm_rank()` is our first example where an MPI function gives a different result depending on the calling process. `rank` will be the rank of the calling process within the communicator: a number from 0 to `nproc`-1.

Exercise 2

- **In this exercise you will invoke the appropriate MPI functions to get the number of processes and the process MPI rank and print to screen**
  - You may choose to modify either `main.c` or `main.f90`
  - Read the comments. They will instruct you as to where the modifications need to be made
  - You can check the syntax of MPI functions on the command line using `man`
    - E.g, `` `man MPI_Init` ``
  - Compile either the C version or the Fortran version with:
    - `make C` or
    - `make F`
  - Once you have compiled and are confident that the program is correct, submit the job with: `qsub ex2.job`
  - Once run, the output will be in `Ex2.out`

Synchronization

- **Explicit synchronization**
  - Explicitly synchronizing all processes in a given communicator requires invoking a barrier:

Fortran	C
`CALL MPI_BARRIER(MPI_COMM_WORLD, IERR)`	`ierr = MPI_Barrier(MPI_COMM_WORLD);`

  - It is guaranteed that any process which calls this function will return from it only if all processes have entered.
- **Implicit synchronization**
  - Processes are implicitly synchronized when *collectives* are invoked (note: `MPI_Barrier()` is also a collective operation)

Exercise 3

- **In this exercise you will invoke the an appropriately placed barrier**
  - Most lines of the exercise in `Ex3/` are identical to those in `Ex2/`.
  - Study the comments in either the Fortran or C code
  - You need to add a barrier, such that the processes print to screen in order of ascending rank
  - Once compiled and run successfully, you should see one line from each rank, in order, in file `Ex3.out`

Collectives

- **Collective operations**
  - Are operations which involve all processes in a given communicator
  - It is the user's responsibility to ensure that all processes in a given communicator enter the collective function
  - Collective operations are *blocking*; the calling process waits (blocks) until the operation has completes

Collectives – some examples

- **Broadcast**
  - Send a message from one process to all processes

Fortran	`CALL MPI_BCAST(ARR, N, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, IERR)`
C	`ierr = MPI_Bcast(arr, N, MPI_DOUBLE, 0, MPI_COMM_WORLD)`

  - Sends the `N` double-precision elements in array `arr` from process with rank `0` to all processes in `MPI_COMM_WORLD`
  - The third argument (`MPI_DOUBLE, MPI_DOUBLE_PRECISION`) are data type handles
  - The broadcast operation causes synchronization of all processes in the communicator

Collectives – some examples

- **Reduction**
  - Perform an operation over data on all processes and store the result in one process

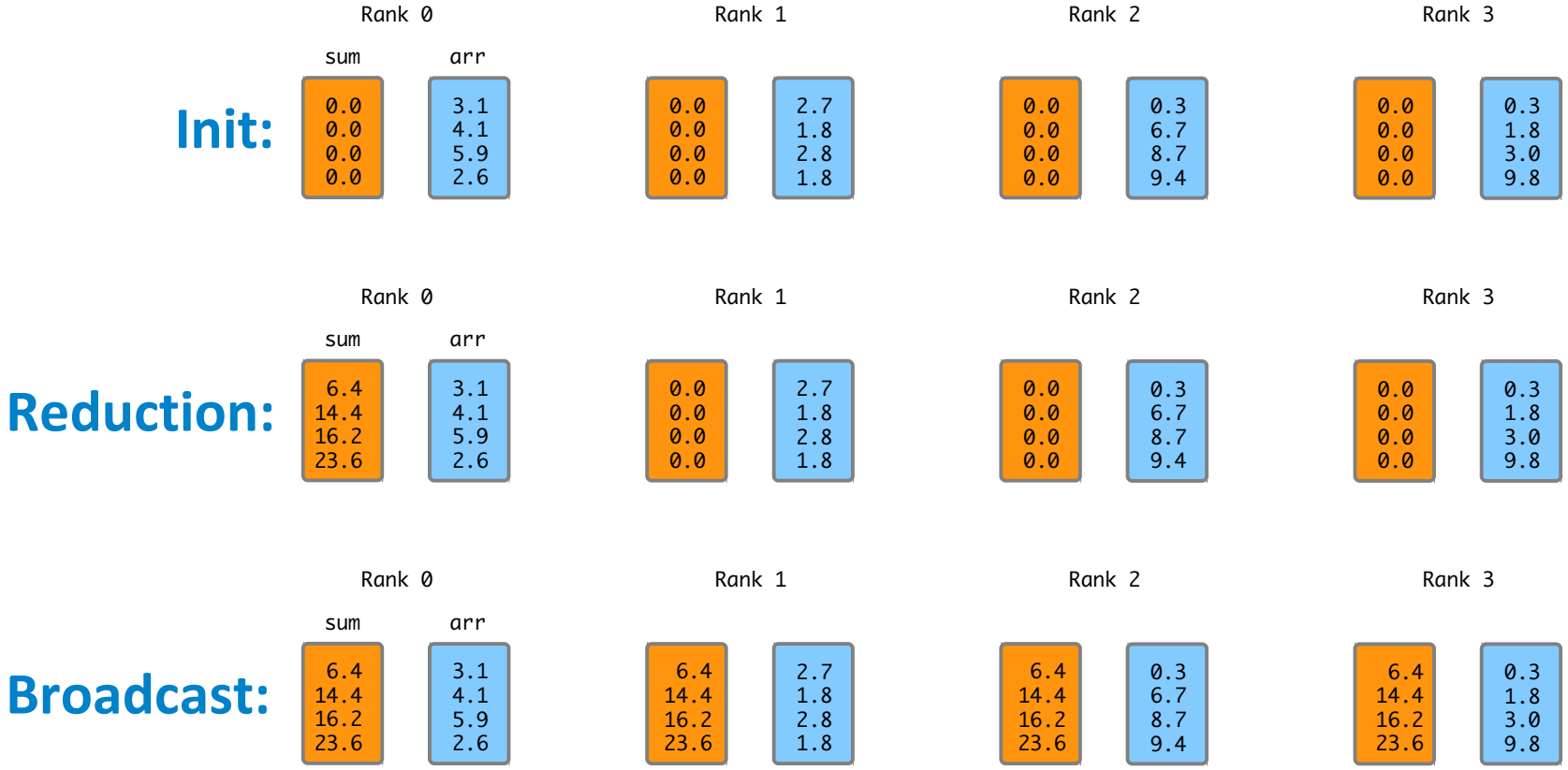
Fortran	`CALL MPI_REDUCE(A, B, 1, MPI_DOUBLE_PRECISION, MPI_SUM, 0, MPI_COMM_WORLD, IERR)`
C	`ierr = MPI_Reduce(&a, &b, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD)`

  - Perform `a` sum over the double-precision variable a over all processes and place the result into `b` on process `0`.
  - The fifth argument (`MPI_SUM`) is an MPI handle to the operation (can e.g. be sum, prod, sub, or, etc.)

Exercise 4

- **This example demonstrates the reduction and broadcast operations**
  - Once the program is complete, it should:
    - Initialize two arrays of double precision values on each process
    - Set the first array to random numbers and the second to zeros
    - Print the sum of the elements of the first arrays in each process
    - Sum element-wise the arrays and put the result in a different array on process 0
    - Print the sum of the elements of the second array in each process. This should be zero in all processes but 0
    - Broadcast the sum-array, from process zero to all processes
    - Print the sum of the elements of the second array in each process. This should now be non-zero for all processes

Exercise 4


Rank 0
Rank 1
Rank 2
Rank 3
sum
arr
Init:
0.0 0.0 0.0 0.0
3.1 4.1 5.9 2.6
0.0 0.0 0.0 0.0
2.7 1.8 2.8 1.8
0.0 0.0 0.0 0.0
0.3 6.7 8.7 9.4
0.0 0.0 0.0 0.0
0.3 1.8 3.0 9.8
Rank 0
Rank 1
Rank 2
Rank 3
sum
arr
Reduction:
6.4 14.4 16.2 23.6
3.1 4.1 5.9 2.6
0.0 0.0 0.0 0.0
2.7 1.8 2.8 1.8
0.0 0.0 0.0 0.0
0.3 6.7 8.7 9.4
0.0 0.0 0.0 0.0
0.3 1.8 3.0 9.8
Rank 0
Rank 1
Rank 2
Rank 3
sum
arr
Broadcast:
6.4 14.4 16.2 23.6
3.1 4.1 5.9 2.6
6.4 14.4 16.2 23.6
2.7 1.8 2.8 1.8
6.4 14.4 16.2 23.6
0.3 6.7 8.7 9.4
6.4 14.4 16.2 23.6
0.3 1.8 3.0 9.8

Point-to-point

- **Point-to-point communication**
  - Communications that involve transfer of data between two processes
  - Integers (tags) are used to match send and receive operations
  - Point-to-point communications may be non-blocking
  - Asynchronous in nature; extra caution for preventing deadlocks

Point-to-point

- **Send/receive**
  - Send/receive a buffer from one process to another

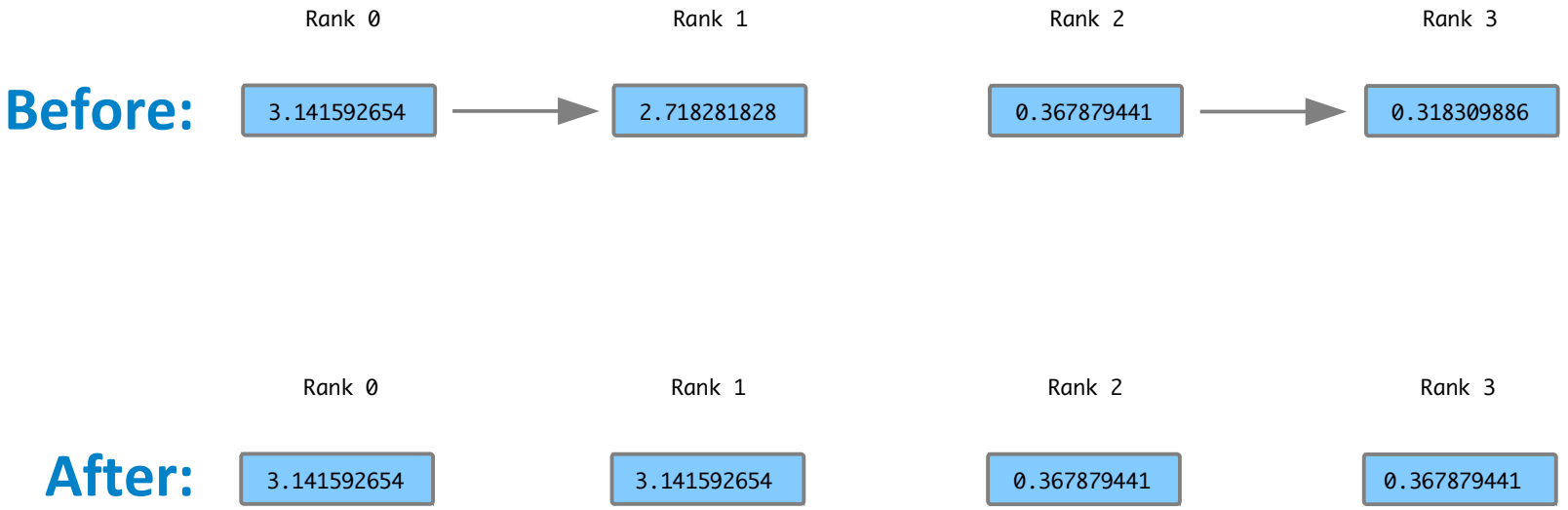
Fortran	`CALL MPI_SEND(ARR, N, MPI_DOUBLE_PRECISION, DEST_RANK, TAG, MPI_COMM_WORLD, IERR)` `CALL MPI_RECV(ARR, N, MPI_DOUBLE_PRECISION, SRCE_RANK, TAG, MPI_COMM_WORLD, STATUS, IERR)`
C	`ierr = MPI_Send(arr, N, MPI_DOUBLE, dest_rank, tag, MPI_COMM_WORLD)` `ierr = MPI_Recv(arr, N, MPI_DOUBLE, srce_rank, tag, MPI_COMM_WORLD, &status)`

  - Send/Receive an array of `N` double-precision numbers
  - `dest_rank/srce_rank`, is the rank of the destination/origin process
  - Tag is an integer chosen by the user. It is used to match a send operation with its corresponding receive
  - Blocking operations:
    - The receive will only return after the receive buffer has been updated
    - The send will only return after it is safe to modify the send buffer without corrupting the message

Exercise 5

- **This exercise demonstrates the Send and the Receive operations**
  - You must modify the source code, such that the even ranks send their `random_double` variables to their neighbouring odd rank
  - The even ranks need to call a “send” and the odd ranks need to call a “receive”
  - You need to specify a tag to match send with receives. As a convention, let the tag be the *sending processes rank*

Exercise 5


Rank 0
Rank 1
Rank 2
Rank 3
Before:
3.141592654
2.718281828
0.367879441
0.318309886
Rank 0
Rank 1
Rank 2
Rank 3
After:
3.141592654
3.141592654
0.367879441
0.367879441

Point-to-point

- **Non-blocking Send/receive**

Fortran	`CALL MPI_ISEND(ARR, N, MPI_DOUBLE_PRECISION, DEST_RANK, TAG, MPI_COMM_WORLD, REQUEST, IERR)`
	`CALL MPI_IRECV(ARR, N, MPI_DOUBLE_PRECISION, SRCE_RANK, TAG, MPI_COMM_WORLD, REQUEST, IERR)`
	`CALL MPI_Wait(REQUEST, STATUS, IERR)`

C	`ierr = MPI_Isend(arr, N, MPI_DOUBLE, dest_rank, tag, &request, MPI_COMM_WORLD)`
	`ierr = MPI_Irecv(arr, N, MPI_DOUBLE, srce_rank, tag, &request, MPI_COMM_WORLD)`
	`ierr = MPI_Wait(&request, &status)`

- The process returns from the Isend/Irecv functions as soon as possible
- A subsequent call to MPI_Wait() will only return after the send or receive operation with the matching request has completed.
- Between an Isend/Irecv and the corresponding Wait, the send/receive buffers are said to be *in-flight*
- Modifying send/receive buffers while in-flight may cause corruption of the messages in an unpredictable way

Exercise 6

- **This exercise demonstrates non-blocking Send and the Receive operations**
  - Each process sets a random double precision number
  - You need to modify the program such that each rank also holds the random number from the next process and the random number from the previous process
  - You can do this by combining a non-blocking send with a blocking receive, or a blocking send with a non-blocking receive

Exercise 6


1:
Rank 0
r_prev
r
r_next
0.0
3.1
0.0
Rank 1
r_prev
r
r_next
0.0
4.1
0.0
Rank 2
r_prev
r
r_next
0.0
5.9
0.0
2:
Rank 0
r_prev
r
r_next
0.0
3.1
4.1
Rank 1
r_prev
r
r_next
0.0
4.1
5.9
Rank 2
r_prev
r
r_next
0.0
5.9
3.1
3:
Rank 0
r_prev
r
r_next
5.9
3.1
4.1
Rank 1
r_prev
r
r_next
3.1
4.1
5.9
Rank 2
r_prev
r
r_next
4.1
5.9
3.1

MPI+OpenMP

- **General Notes**
  - MPI and OpenMP can be used side-by-side in a fairly straight forward way
  - The most direct way is to use all MPI functions outside OpenMP parallel regions, and spawn off OpenMP threads in regions which do not call MPI functions
  - Remember:
    - You need to include <omp.h>
    - You need to use compiler flags. Depends on the compiler:
      - GNU: -fopenmp
    - Some useful OpenMP API functions (they both return integers):
      - `omp_get_num_threads()`
      - `omp_get_thread_num()`

Exercise 7

- **Print rank numbers and OMP thread number**
  - This exercise is similar to Exercise 3, but now we will also report on the OMP thread number
  - You need to call the appropriate functions so that:
    - The program prints the number of threads
    - The program prints the thread number of the calling thread
    - The above are done in thread-order, by placing appropriately an OpenMP barrier
  - **Caution**: do not call an MPI Barrier in an OMP parallel region!
  - You also need to modify the job-script `ex7.job`, to set `OMP_NUM_THREADS` to the number of threads you would like to run for

Exercise 8

- **This example demonstrates a distributed sum**
  - Rank 0 initializes arrays with random numbers, one for each process, and sends to each process
  - Each process sums the elements of the array
  - These partial sums are summed via a reduction operation and stored in rank 0, and subsequently broadcast to all processes
  - To check, rank 0 repeats but without sending the arrays. It performs the sum to check the correctness of the previous process

Exercise 8

- In `./Exercises/Ex8/`
  - Read the comments in `main.c` carefully
    - sorry, no Fortran version :(
  - You need to fill in the lines tagged with:
    - `__MISSING_LINE__` or
    - `__MISSING_LINES__` in the function definitions
  - You need to invoke the appropriate functions where it says:
    - `__MISSING_FUNCTION_CALL__` in the main function