

Exercises for MPI Intro Course

Exercise 1: Hello World

LEARNING AIM: To learn how to compile and run MPI programs on a local machine and to understand that multiple versions of a program can run on different processes. Each process has a unique rank in the range of 0 to n-1, where n is the total number of processes. Using this information it is possible to control which processes execute certain instructions.

1. Using the program skeleton provided, write a program that causes each MPI process to print

Hello world from process i of n

using the rank in `MPI_COMM_WORLD` for i and the size of `MPI_COMM_WORLD` for n.

Hint: Use `MPI_Init()`, `MPI_Comm_rank()`, `MPI_Comm_size()` and `MPI_Finalize()`

2. Compile and run it on a single processor.
3. Run it on several processors in parallel.
4. Modify your program so that Process 0 also prints "I am Process 0"

Exercise 2: Ping Pong

LEARNING AIM: To experiment with point-to-point communications and introduce the MPI timing routines. You will be able to see the effects of message size on how long communications take, and as a result, learn how to write more efficient programs.

1. Using the program skeleton provided, write a program in which 2 processes repeatedly pass a message back and forth.

Hints:

- Use `MPI_Recv()` and `MPI_Ssend()`
- Remember to declare any extra parameters which are required

Exercise 3: Timing

LEARNING AIM: To learn how to submit jobs to a cluster and to introduce the MPI timing routines. You will be able to see the effects of message size on how long communications take, and as a result, learn how to write more efficient programs.

1. Using the program you wrote in Exercise 2, insert timing calls to measure the time taken for one message. Test your program locally before running it on the cluster.

Hint:

- Use `MPI_Wtime()`

2. Modify your program to investigate how the time taken varies with the size of the message. Is it more efficient to send data as many small messages, or one large one?

Exercise 4: Using gdb in Parallel

LEARNING AIM: To test how gdb can be used to debug MPI programs. This will help you sort out bugs and crashes for your own code in the future.

Your task is to use gdb to look for the value of the variables **loop** and **value** on each processor.

1. Compile the code Exercise3.c as ex3, remembering to include the -g flag for debugging
2. Open another terminal window
3. On the first terminal, mpirun the program with gdb and 2 processors

```
# mpirun -gdb -np 2 ./ex3
```
4. Set an appropriate break point using the *break* command. Use *z* to select processors and *next* or *step* commands to go through the code line-by-line. Then use *print* command to display the value of a variable.

Exercise 5: The Cooperation/Competition (“Prisoner's Dilemma”) Game

LEARNING AIM: To get a hands-on feel for how non-blocking communications work in practice.

Description: This exercise is a classic game-theory scenario, often run with humans as the players. However, in this example you will set up a parallel architecture using MPI such that we can run automated experiments with computer agents as the players.

In this game for 2 players, we play ten rounds, with the goal to maximise earnings by the end of the ten rounds. In each round, each player must make an independent decision to either co-operate, or compete (another version of this game sees two prisoner's either cooperating or “turning” on the other, hence the alternative title).

The payoffs to each player in various scenarios is as follows. If both cooperate, they receive \$2 each. If both compete, they receive \$1 each. If one cooperates and the other competes, then the competitor gets \$3, and the cooperator gets \$0.

The following diagram summarises the payoffs.

	Cooperate	Compete
Cooperate	\$2, \$2	\$0, \$3
Compete	\$3, \$0	\$1, \$1

See http://en.wikipedia.org/wiki/Prisoner's_dilemma for more information.

The Code, Architecture, & Your Task

You have been given a serial version of the code to start with, where the players are queried for their decisions in sequence, then the result of their decisions is determined, and they are both told the results. Each Player's decision making process is encoded in a function that you don't need to modify. PlayerA is a player who favours cooperation, and PlayerB is a more “sneaky” player who goes for the competition option more often.

Your job is to complete the parallel version of the code provided. Each process must run one of the players decision-making, then they both have to transmit their decision to the other, and then update their own tally of the result.

Hint: Use `MPI_Send()`, `MPI_Irecv()`, and `MPI_Wait()`

Optional task: If you finish this task early, you could attempt to generalise the code to include a third player. You will have to invent a new payoff matrix as appropriate, and this time each player will need to communicate their decision to both other players. Perhaps your third player could be a moderate, somewhere between the other two in terms of its strategy. What happens to the chance for cooperation as more players are added?