OpenMP exercises

1. Parallel region and data clauses

Take a look at the exercise skeleton `exercises/ex1_variables.c` (or `ex1_variables.f90`). Add an OpenMP parallel region around the block where the variables Var1 and Var2 are printed and manipulated. What results do you get when you define the variables as **`shared`**, **`private`** or **`firstprivate`**? Explain why do you get different results.

2. Work sharing for a simple loop

The file `exercises/ex2_sum(.c|.f90)` implements a skeleton for the simple summation of two vectors C=A+B. Add the computation loop and add the parallel region with work sharing directives so that the vector addition is executed in parallel.

3. Dot product and race condition

The file `exercises/ex3_dotprod(.c|.f90)` implements a simple dot product of two vectors. Try to parallelize the code by using **`omp parallel`** or **`omp for`** pragmas. Are you able to get same results with different number of threads and in different runs? Explain why the program does not work correctly in parallel. What is needed for correct computation?

4. Reduction and critical

Continue with the previous dot product example and use **`reduction`** clause to compute the sum correctly. Implement also an alternative version where each thread computes its own part to a private variable and then use a **`critical`** section after the loop to compute the global sum. Try to compile and run your code also without OpenMP. Do you get exactly same results in all cases?

5. Using the OpenMP library functions

Write a simple program that uses **`omp_get_num_threads`** and **`omp_get_thread_num`** library functions and prints out the total number of active threads as well as the id of each thread.

6. (Bonus *) Heat equation solver parallelized with OpenMP

a) Parallelize the serial heat equation solver with OpenMP by parallelizing the loops for data initialization and time evolution.

b) Improve the OpenMP parallelization so that the parallel region is opened and closed only once during the program execution.

MPI I: Introduction to MPI

1. Parallel “Hello World”

a) Write a simple program that prints out i.e. “Hello” from multiple processes. Include the MPI headers (C) or use the MPI module (Fortran) and call appropriate initialization and finalization routines.

b) Modify the program so that each process also prints out its rank, and have the rank 0 to print out the total number of MPI processes as well.

2. Simple message exchange

a) Write a simple program where two processes send and receive a message to/from each other using **`MPI_Send`** and **`MPI_Recv`**. The message content is an integer array, where each element is initialized to the rank of the process. After receiving a message, each process should print out the rank of the process and the first element in the received array. You may start from scratch or use as a starting point one of the files `exercises/ex2_ms_exchange(.c|.f90)`.

b) Increase the message size to 100 000 and investigate what happens when reordering the send and receive calls in one of the processes.

3. Message chain

Write a simple program where every processor sends data to the next one. Let **`ntasks`** be the number of the tasks, and **`myid`** the rank of the current process. Your program should work as follows:

- Every task with a rank less than ntasks-1 sends a message to task myid+1. For example, task 0 sends a message to task 1.
- The message content is an integer array where each element is initialized to myid.
- The message tag is the receiver’s id number.
- The sender prints out the number of elements it sends and the tag number.
- All tasks with rank ≥ 1 receive messages.
- Each receiver prints out their myid, and the first element in the received array.

a) Implement the program described above using **`MPI_Send`** and **`MPI_Recv`**. You may start from scratch or use as a starting point on of the files `exercises/ex3_msg_chain(.c|.f90).`

b) (Bonus *) Use the status parameter to find out how much data was received, and print out this piece of information for all receivers
c) (Bonus *) Use **`MPI_ANY_TAG`** when receiving. Print out the tag of the received message based on the status message.
d) Use **`MPI_ANY_SOURCE`** and use the status information to find out the sender. Print out this piece of information.
e) Can the code be simplified using **`MPI_PROC_NULL`**?
f) Use **`MPI_Sendrecv`** instead of **`MPI_Send`** and **`MPI_Recv`**.

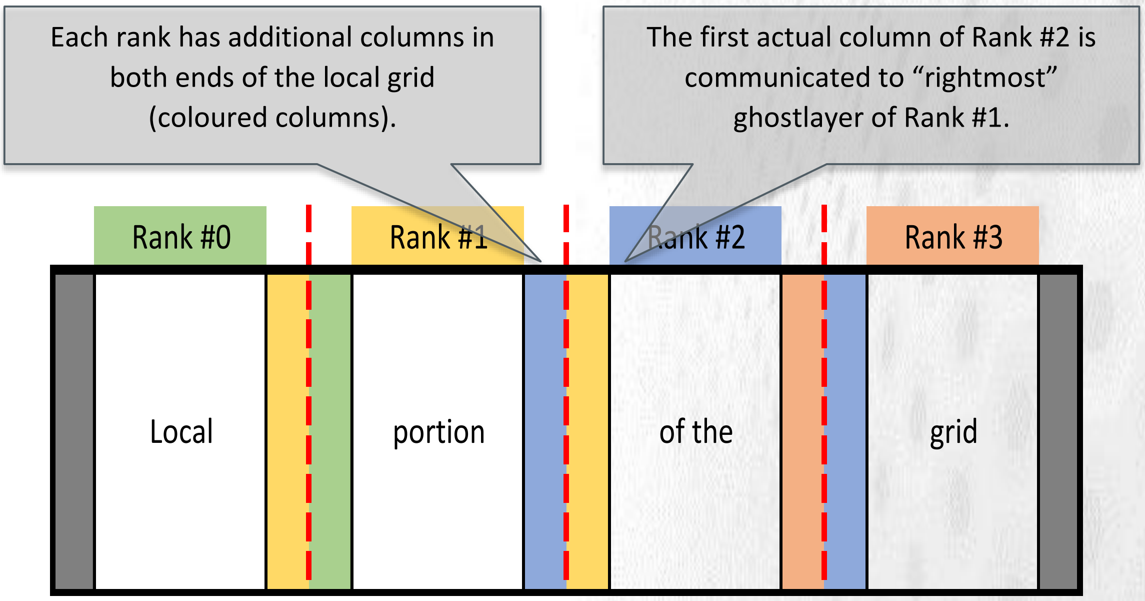
4. Parallel heat equation

Parallelize the whole heat equation program with MPI, by dividing the grid in columns (for Fortran – for C substitute row in place of each mention of a column) and assigning one column block to one task. A domain decomposition, that is.

The tasks are able to update the grid independently everywhere else than on the column boundaries – there the communication of a single column with the nearest neighbor is needed. This is realized by having additional ghost layers that contain the boundary data of the neighboring tasks. As the system is aperiodic, the outermost ranks communicate with single neighbor, and the inner ranks with the two neighbors.

Insert the proper MPI routines into skeleton codes available at `ex4_heat_mpi(.c|.f90)` and `ex4_main(.c|.f90)` (search for"TODO"s). You may use the provided `Makefile_ex4` for building the code.

A schematic representation of column-wise decomposition looks like:


Each rank has additional columns in both ends of the local grid (coloured columns).
The first actual column of Rank #2 is communicated to "rightmost" ghostlayer of Rank #1.
Rank #0
Rank #1
Rank #2
Rank #3
Local
portion
of the
grid

Remember to update all ghost layers at each iteration.

MPI II: Collective operations & communicators

5. Collective operations

In this exercise we test different routines for collective communication. First, write a program where rank 0 sends and array containing numbers from 0 to 7 to all the other ranks using collective communication.

Next, let us continue with four processes with following data vectors:

Task 0:	0	1	2	3	4	5	6	7
Task 1:	8	9	10	11	12	13	14	15
Task 2:	16	17	18	19	20	21	22	23
Task 3:	24	25	26	27	28	29	30	31

In addition, each task has a receive buffer for eight elements and the values in the buffer are initialized to -1. Implement a program that sends and receives values from the data vectors to receive buffers using a single collective communication routine for each case, so that the receive buffers will have the following values:

a)

Task 0:	0	1	-1	-1	-1	-1	-1	-1
Task 1:	2	3	-1	-1	-1	-1	-1	-1
Task 2:	4	5	-1	-1	-1	-1	-1	-1
Task 3:	6	7	-1	-1	-1	-1	-1	-1

b)

Task 0:	-1	-1	-1	-1	-1	-1	-1	-1
Task 1:	0	8	16	17	24	25	26	27
Task 2:	-1	-1	-1	-1	-1	-1	-1	-1
Task 3:	-1	-1	-1	-1	-1	-1	-1	-1

c)

Task 0:	8	10	12	14	16	18	20	22
Task 1:	-1	-1	-1	-1	-1	-1	-1	-1
Task 2:	40	42	44	46	48	50	52	54
Task 3:	-1	-1	-1	-1	-1	-1	-1	-1

Tip: you might want to create two communicators

You can start from scratch or use a skeleton file `exercises/ex5_collectives.(c|f90)`.

MPI III: Non-blocking communication & user-defined datatypes

6. Non-blocking communication

a) Go back to the exercise 3 “Message chain” and implement it using non-blocking communication.
b) Revisit the exercise 5 “Collective operations” where you replace the operations with their non-blocking counterparts.

7. Non-blocking communication in heat equation

Implement the halo exchange in heat equation using non-blocking communication.

8. Vector datatype

Write a program that sends a non-contiguous data structure, such as a row (in Fortran) or a column (in C) of an array from one process to another by using your own datatype. A skeleton code is provided in `exercises/ex8_vector_type(.c|.f90)`.

9. Subarray datatype

Write a program that sends a sub-block of a matrix from one process to another by using the subarray datatype. A skeleton code is provided in `exercises/ex9_subarray_type(.c|.f90)`.

MPI IV: Communication topologies & one-sided communication

10. Testing Cartesian process topologies

Create a one-dimensional Cartesian process topology for the message chain.

11. Two dimensional heat equation

Create a two dimensional Cartesian process topology for the halo exchange in the heat equation. Parallelize the heat equation in two dimensions and utilize user defined datatypes in the halo exchange during the time evolution. MPI contains also a contiguous datatype MPI_Type_contiguous which can be used in order to use user defined datatypes both in x- and y-directions in the halo exchange. A skeleton code is provided in `exercises/ex11_heat_2d(.c|.f90).` Utilize user-defined datatypes also in the I/O related communication.

12. Single-sided message chain

Starting from the Exercise 3 ”message chain”, implement a similar communication pattern but use now one-sided communication.

13. One-sided communication in the heat equation solver

Rewrite the halo exchange of the heat equation program using one-sided communication.