Parallel programming with MPI

Practicalities

1. Computing servers

We will use CSC's Cray supercomputer Sisu for the exercises. Log onto Sisu using the provided tnrgXX username and password, e.g.

% **ssh -X trng10@sisu.csc.fi**

For editing program source files you can use e.g. Emacs editor with or without (the option –nw) X-Windows:
emacs -nw prog.f90
emacs prog.f90

Also other popular editors (vim, nano) are available.

2. Simple compilation and execution

Compilation and execution are done via the **ftn** and **cc** wrapper commands and the **aprun** scheduler:

% **ftn -o my_mpi_exe test.f90**

% **cc -o my_mpi_exe test.c**
% **aprun -n 4 ./my_mpi_exe**

The wrapper commands include automatically all the flags needed for MPI programs

We will use the default Cray compiling environment. There are also other compilers (GNU and Intel) available on Sisu, which can be changed via (for example)
% **module swap PrgEnv-cray PrgEnv-gnu**

Use the commands **module list** and **module avail** to see the currently loaded and available modules, respectively.

4. Batch jobs

Larger/longer runs should be submitted via batch system. Example batch job script for MPI calculation:

```
!/bin/bash
#SBATCH -t 00:10:00
#SBATCH -J MPI_job
#SBATCH -o out.%j
#SBATCH -e err.%j
#SBATCH -p test
#SBATCH -N 2
aprun -n 48 ./my_mpi_exe
```

The batch script is submitted with sbatch command:

```
% sbatch sisu_job.sh
```

See Sisu guide http://research.csc.fi/sisu-user-guide for more details.

5. Skeleton codes

Exercise material can be found under the directory `/appl/courses/MPI_course` in Sisu. Skeleton codes both for Fortran 90 and C are provided in the `exercises` subdirectory. Generally, you do not need to read through all of the code, but look for sections marked with TODO for completing the exercises.

Exercise assignments

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**.
g) Use non-blocking communication.

4. 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:

h)

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

i)

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

j)

Task 0:	48	52	56	60	64	68	72	76
Task 1:	-1	-1	-1	-1	-1	-1	-1	-1
Task 2:	-1	-1	-1	-1	-1	-1	-1	-1
Task 3:	-1	-1	-1	-1	-1	-1	-1	-1

Tip: these are sums of columns

You can start from scratch or use a skeleton file `exercises/ex4_collectives.(c|f90)`.

5. Testing Cartesian process topologies

Create a one-dimensional Cartesian process topology for the message chain of Exercise 3.

6. Parallel heat equation

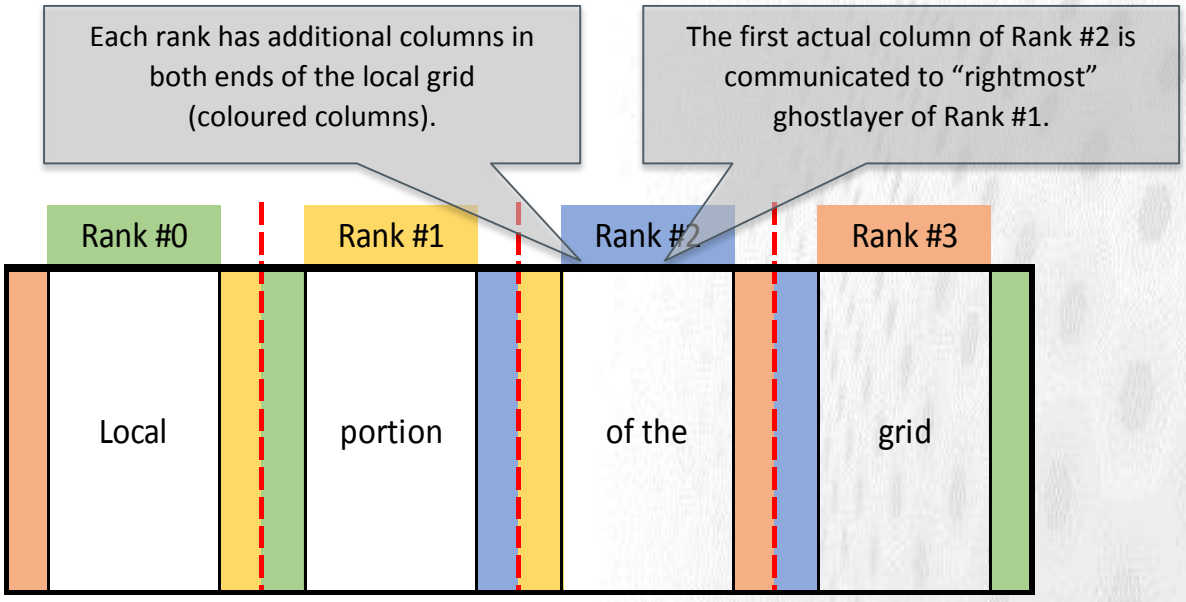
See the appendix for an introduction to the heat equation solver.

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. The periodicity in the other direction is accounted as earlier. Make all the MPI tasks print their own parts of the grid on different files, e.g. with tt as the timestep and mm the rank: `heat_tt_mm.png`.

Insert the proper MPI routines into skeleton codes available at `ex6_heat_mpi(.c|.f90)` and `ex6_main(.c|.f90)` (search for"TODO"s). You may use the provided `Makefiles` (i.e. **`make -f Makefile_C`** or **`make -f Makefile_Fortran`**) for building the code.

A schematic representation of column-wise decomposition looks like:



Remember to update all ghost layers at each iteration.

Appendix: Heat equation solver

The heat equation is a partial differential equation that describes the variation of temperature in a given region over time

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u$$

where $u(x, y, z, t)$ represents temperature variation over space at a given time, and α is a thermal diffusivity constant.

We limit ourselves to two dimensions (plane) and discretize the equation onto a grid. Then the Laplacian can be expressed as finite differences as

$$\nabla^2 u(i,j) = \frac{u(i-1,j) - 2u(i,j) + u(i+1,j)}{(\Delta x)^2} + \frac{u(i,j-1) - 2u(i,j) + u(i,j+1)}{(\Delta y)^2}$$

Where Δx and Δy are the grid spacing of the temperature grid u(i,j). We can study the development of the temperature grid with explicit time evolution over time steps Δt:

$$u^{m+1}(i,j) = u^m(i,j) + \Delta t\ \alpha\ \nabla^2 u^m(i,j)$$

There are solvers for the 2D equation implemented with C and Fortran in `exercises`. You can compile the program by adjusting the Makefile as needed and typing “make”.

The solver carries out the time development of the 2D heat equation over the number of time steps provided by the user. The default geometry is a flat rectangle (with grid size provided by the user), but other shapes may be used via input files - a bottle is give as an example. Examples on how to run the binary:

- ./heat (no arguments - the program will run with the default arguments: 256x256 grid and 500 time steps)
- ./heat bottle.dat (one argument - start from a temperature grid provided in the file bottle.dat for the default number of time steps)
- ./heat bottle.dat 1000 (two arguments - will run the program starting from a temperature grid provided in the file bottle.dat for 1000 time steps)
- ./heat 1024 2048 1000 (three arguments - will run the program in a 1024x2048 grid for 1000 time steps)

The program will produce a .png image of the temperature field after every 10 iterations. You can change that from the parameter image_interval. Images can be viewed e.g. with **eog *png**