CSC Summer School in HIGH-PERFORMANCE COMPUTING

June 26 – July 4, 2018 • Espoo, Finland

Exercise Assignments

General instructions for the exercises

Computing servers

We will use CSC's Cray XC40 supercomputer Sisu for the exercises. Log onto Sisu using the provided trainingXXX username and password, for example

```
ssh -X training075@sisu.csc.fi
```

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

```
module load emacs
```

```
emacs -nw prog.F90
emacs prog.F90
```

Also other popular editors (such as vim and nano) are available.

In case you have working parallel program development environment in your laptop (Fortran or C compiler, MPI development library, etc.) you may also use that.

Simple compilation and execution

Compilation on Sisu is carried out with the **ftn** (Fortran) and **cc** (C) wrapper commands:

```
ftn -o my_mpi_exe test.f90
cc -o my_mpi_exe test.c
```

These wrapper commands include automatically everything needed for building MPI programs.

Typically 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.

Batch jobs

You can run small (short parallel or serial) jobs interactively using the **aprun** application placement utility:

```
aprun –n 4 ./my_mpi_exe
```

Here –n specifies the number of MPI tasks. Larger/longer runs should be submitted via batch system. An example batch job script (let's call it `sisu_job.sh`) for a MPI job

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

The batch script is then submitted with the **sbatch** command as

```
sbatch sisu_job.sh
```

We have a special reservation **summer_school** for the summer school which can be utilized with the **--reservation** flag:

```
sbatch --reservation=summer_school sisu_job.sh
```

See the Sisu User's Guide at **http://research.csc.fi/sisu-user-guide** for more details.

Skeleton codes and example solutions

For most of the exercises, skeleton codes are provided in Fortran and C in the subdirectory of the corresponding section at the summer school git repository. Each exercise has their own subdirectories. Some exercise skeletons have sections marked with "TODO" for completing the exercises.

In addition, most of the exercises have exemplary full codes (that can be compiled and run) in the `solutions` folder. Note that these examples are seldom the only (or even the best) way to solve the given problem.

Working with UNIX and version control exercises

1. Version control with git

a) Let's configure your local git. Set up your username and email with:

```
git config --global user.name "nickname"
git config --global user.email "user@email.com"
```

b) Fork the course repository at https://github.com/csc-training/summerschool
Make sure you are logged into GitHub with the account that you have created.

c) Clone **your own forked** repository to the computer you are using

```
git clone https://github.com/user/summerschool
```

Pay attention that it is indeed your repository by checking the suggested url. Think also where in filesystem you currently are because the git repository will be created there. Usually you want to clone to your home directory.

d) Open the `README.md` with a text editor and add some short note to yourself.
Save the file and then add it to staging, commit it, and finally push to the repository. After every step, check the current status with `git status`, and use:

```
git add README.md
git commit -m "added new note"
git push
```

2. Makefiles, modules, and batch job system

a) Clone your own repository also to Sisu. In Sisu, go to the folder `unix/` and try to compile the `prog.c` using the make command (**`make prog`**). Compilation should fail with an error message telling that you should use gnu compiler.

b) Issue the correct module command to switch the development environment from **`PrgEnv-cray`** to **`PrgEnv-gnu`** and try again to compile the program.

c) See the batch job script `job_script.sh` and add the missing aprun command to execute the program that you just compiled. The program should be run using 24 cores! Send your job to the queue and check the results.

Fortran 95/2003 exercises

Session: Getting started with Fortran

1. Program compilation and execution

Compile the supplied "Hello world"-program (`programming/fortran/hello/hello.F90`) and run it. Modify the program such that you define some variables and assign some values for them. Make some (likely less meaningful) computations using those variables and print out the values of those.

2. Control structures

a) Define a two-dimensional (nx-by-ny), real-valued array and initialize it to the values of the function $f(x,y) = x^2 + y^2$ in the square $x \in [0,1], y \in [0,1]$ using nx = ny = 10. Exercise skeleton is available in `programming/fortran/control-structures/do-loop.F90`.

b) Write a control structure which checks whether an integer is negative, zero, or larger than 100 and performs corresponding **`write`**. Investigate the behavior with different integer values.

c) Fibonacci numbers are a sequence of integers defined by the recurrence relation

$F_n = F_{n-1} + F_{n-2}$

with the initial values $F_0=0$, $F_1=1$. Print out Fibonacci numbers $F_n < 100$ using a **`do while`** loop.

Session: Arrays in Fortran

3. Dynamic arrays

Define a two-dimensional real-valued array, which should be dynamically allocatable, allocate the array with sizes read in from the user input, and initialize it similarly as in Exercise 2. A skeleton code is provided in `programming/fortran/arrays/arrays.F90`.

4. Laplacian

Write a double **do** loop for evaluating the Laplacian using the finite-difference approximation

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

Start from the skeleton `programming/fortran/laplacian/laplacian.F90`. Test your implementation with the function implemented in Exercise 2. The analytic solution for that function is $\nabla^2 f(x,y) = 4$, check the correctness of your implementation.

5. Array intrinsic functions

Start from the dynamically allocatable array presented in exercise 3 or from the skeleton code provided at `programming/fortran/intrinsics/intrinsics.F90`. Use suitable array intrinsic functions to print out the following details:

a) What is the sum of elements across the 2nd dimension of A?

b) Find the location coordinates of the maximum elements of A.

c) What is the absolute minimum value of the array A?

d) Are all elements of A greater than or equal to zero?

e) How many elements of A are greater than or equal to 0.5?

Session: Procedures and modules

6. Subroutines and modules

Write three subroutines for i) array initialization, ii) computing a Laplacian, iii) printing out a given array, and put them in a separate module. Then, write a main program that defines two (dynamically allocatable) arrays, **`previous`** and **`current`**; initialize the **`previous`** array; apply the Laplacian on the initialized array; and print out both arrays (after and before the Laplacian) by calling the three module procedures described above. A skeleton code provided in `programming/fortran/subroutines`.

Session: Input and output

7. File I/O

Implement a function that reads the values from a file `bottle.dat`. The file contains a header line: "# nx ny", followed by lines representing the rows of the field. A skeleton code is provided in `programming/fortran/io/io.F90`.

Session: Derived types

8. Derived types

a) Define a derived type for a temperature field. Do the definition within a module (let's call it heat). The type contains the following elements:

- Number of grid points nx (=number of rows) and ny (=number of columns) (integers)
- The grid spacings dx and dy in the x- and in the y-directions (real numbers)
- An allocatable two-dimensional, real-valued array containing the data points of the field.

Define the real-valued variables into double precision, using the **`ISO_FORTRAN_ENV`** intrinsic module.

b) Append this module with a subroutine that takes as input the number of grid points in both dimensions and returns a field type where the metadata (grid points and grid spacings; let us set the latter to 0.01) has been initialized.

Session: Further useful features

9. Vector module

In `programming/fortran/vecmod` the file `vec.F90` contains a simple user interface for a program that performs basic operations for vectors given by the user (here 3D vectors for simplicity). Your task is to implement operations for calculating the vector length (norm), as well as summation, subtraction, dot product and vector (cross) product between two vectors. Overload them to match with the syntax in `vec.F90`. Placeholders for these in `vec_mod.F90`.

10. Heat equation

Finalize the implementation of our two-dimensional heat equation solver (see the Appendix) by filling in the missing pieces of code (marked with "TODO" in files `core.F90` and `io.F90`) in `heat/serial/fortran`. You can compile the code with the provided makefile.

a) The main task is to write the procedure that evaluates the temperature field based on a previous iteration (the routine is called **`evolve`** here)
$$u^{m+1}(i,j) = u^m(i,j) + \Delta t\ \alpha\ \nabla^2 u^m(i,j)$$
utilizing our earlier implementation of the Laplacian in the last term. The skeleton codes readily contain suitable values for time step Δt and for the diffusion constant α. Run the code with the default initialization.

b) Another task is to implement a reading in of the initial field from a file, routine **`read_field`** located in the file `io.F90`. Test your implementation with the provided `bottle.dat`.

The complete solver is provided in `heat/serial/fortran/solution`.

Introduction to C

1. Basics and compiling

Write from a scratch a short program that prints out a sentence (e.g. “hello world!”). Compile and execute the program.

2. Datatypes and expressions

a) Simple arithmetic expressions. Perform simple expressions (additions, subtractions, multiplications, divisions) with different type of variables (integers, floats, doubles) and print out results with **`printf`** (use correct format). Investigate also what happened in type conversions when combining different types. File `datatypes-expressions/expressions.c` contains a skeleton code to start with.

b) Using pointers. Add some pointer variables to the previous exercise assign them to addresses of existing variables. Perform simple expressions using both the original and pointer variables, and investigate their values after expressions. A skeleton code is provided in `datatypes-expressions/pointers.c`

3. Control structures

a) Write a control structure which checks whether an integer is negative, zero, or larger than 100 and performs corresponding **`printf`**. Investigate the behavior with different integer values.

b) In the file `control-structures/my_pow_prog.c` you find a skeleton code for a program that calculates powers. Finish the code by adding a missing update loop.

c) Fibonacci numbers are a sequence of integers defined by the recurrence relation

$$F_n = F_{n-1} + F_{n-2}$$

with the initial values $F_0=0$, $F_1=1$. Print out Fibonacci numbers $F_n < 100$ using a **`while`** loop.

4. Data structures and functions

a) Create an array for storing the first 20 Fibonacci numbers, and write a **for** loop for performing the evaluation. A skeleton code is provided in file `datastructures-functions/fibonacci.c`.

b) Write a function that takes two pointer arguments and adds 1 to the values pointed to by both pointers. The behaviour should be as follows:

```
int a, b;
a = 10;
b = 20;
add_one_to_both(&a, &b);
// at this point a == 11 and b == 21
```

There is no skeleton code provided.

5. Data structures and functions for heat equation

Skeleton codes for these exercises are provided under `datastructures-functions-heat`

a) Create a two-dimensional 258x258 array of double precision numbers. Initialize the array such that values are:

20.0 on the left boundary
85.0 on the upper boundary
70.0 on the right boundary
5.0 on the lower boundary

and otherwise zeros.

A skeleton code is provided in the file `2d_array.c`

b) Write a double for loop for evaluating the Laplacian using the finite-difference approximation:

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

As an input use the 258x258 array of Exercise a (or start from the skeleton `laplacian.c`). Evaluate the Laplacian only at the inner 256x256 points, the outer points are used as boundary condition. As a grid spacing, use dx=dy=0.01.

c) Create a struct for a temperature field. The struct has the following elements:
- number of grid points nx and ny in the x- and in the y-direction
- the grid spacings dx and dy in the x- and in the y-directions
- the squares of grid spacings dx2 and dy2

- two dimensional array containing the data points of the field. The array contains also the boundary values, so its dimensions are nx+2 x ny+2

d) Implement the initialization of the two dimensional array (exercise a) and finite-difference Laplacian (exercise b) in their own functions, which take as input the struct representing the temperature field (exercise 5c).

6. Dynamic arrays

a) Implement the two-dimensional array within the struct of Exercise 5c as dynamic array. Allocate space for the array based on the command line arguments providing the x- and y-dimensions of array. A skeleton code and header is provided in `dynamic-arrays/dynamic_array.c(/.h)`.

7. Multiple files

a) Implement the functions in Exercise 5d in a separate source file, and write also a header file providing the function declarations. Include the header file and call the routines in the main program (You may start also from the model solution of exercises 5d).

b) The files `multiple-files/pngwriter.*` provide a utility function **`save_png`** for writing out a two dimensional array in the .png file format. Investigate the header file `multiple-files` /`pngwriter.h` and use the utility function for writing out the array of the exercise 4 b. You can also start from the skeleton code in `multiple-files /libpng.c`. You need to include the header file, compile also the file `multiple-files` /`pngwriter.c` and link your program agains the libpng library, i.e. use –lpng linker option.
Try to build executable first directly in the command line and then with the provided makefile `Makefile` (i.e. issue command "**`make`**"). The .png file can be investigated with **eog** program ("eog file.png").

8. File I/O

Implement a function that reads the initial temperature field from a file `bottle.dat`. The file contains a header line: "# ncols nrows", followed by lines representing the rows of the temperature field. A skeleton code is provided in `fileio/read.c`

9. Heat equation

Finalize the implementation of the two-dimensional heat equation. You can utilize code written in the previous exercises or use the provided skeleton codes under `exercises/heat/C/serial`. The source files and parts where you need to write code and indicated with **TODO**s (try e.g. "grep TODO *.c *.h"). There is a ready to use `Makefile`, so you can build the whole code just with **`make`**.

First, you need two temperature fields for the current and for the previous time steps. Allocate the 2D arrays either based on default dimensions or based on command line arguments. Initialize the boundary values of temperature fields as in Exercise 5a, or read the initial data from `bottle.dat`. Write a function that evaluates the new temperature based on previous one, utilizing the Laplacian:

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

The skeleton code `main.c` contains proper values for time step Δt and for the diffusion constant α. Within the main routine evolve the temperature field e.g. for 500 steps and write out every 10th field into a .png file.

The program can be run with different command line arguments:

- ./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)

Visualize the results using eog or animate:
`eog heat_*png` or **`animate heat_*png`**

(Note: model solution contains also a disc in the center of grid as default initial pattern for the temperature field)

Temperature field of bottle.dat after 500 time steps

Introduction to C++

1. Basics and compiling

Write from a scratch a short program that prints out a sentence (e.g. “hello world!”). Compile and execute the program.

2. Objects

Take a look at the skeleton provided in `cpp-objects/class-demo.cpp`. Skeleton provides a simple class declaration. Extend the example with a member function that prints the class member values. Add a call to this member function to the main program.

3. STL and std::vector containers

Start from the skeleton code provided in `cpp-stl/vector.cpp` that will ask the user to input 10 numbers.

a) Introduce a `std::vector` container to store these numbers. Then modify the code such that the numbers provided by the user are appended to this container.

b) Use `std::sort` function provided by `<algorithm>` to sort these numbers after they are given.

c) Use `std::min_element` and `std::max_element` from `<algorithm>` to find what is the smallest and largest element in the container.

4. Multidimensional array class for heat equation

Skeleton codes for these exercises are provided under
`cpp-array2d/`

a) Create a two-dimensional 258x258 array of double precision numbers using the `Matrix<T>` class template provided to you in `matrix.h`. Introduce an initialization function that will initialize the input array such that values are:

   20.0 on the left boundary
   85.0 on the upper boundary
   70.0 on the right boundary

5.0 on the lower boundary
and otherwise zeros.
A skeleton code is provided in the file `array2d.cpp`

b) Write a function that implements a double for loop for evaluating the Laplacian using the finite-difference approximation:

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

As an input use the 258x258 array of Exercise a. Evaluate the Laplacian only at the inner 256x256 points, the outer points are used as boundary condition. As a grid spacing, use dx=dy=0.01.

c) **BONUS**: Generalize the given `Matrix<T>` template class into a temperature field class template `Field<T>`. The class has the following attributes:
- number of grid points nx and ny in the x- and in the y-direction
- the grid spacings dx and dy in the x- and in the y-directions
- two dimensional data array containing the data points of the field. The array should also contain the boundary values, so its dimensions are nx+2 x ny+2.
Finally, implement the initialization of the two dimensional array (exercise a) and finite-difference Laplacian (exercise b) in their own functions, which take as input the class representing the temperature field (exercise c).

OpenMP exercises

1. Hello world

In this exercise you will test that you are able to compile and run an OpenMP application. Take a look at a simple example program in `hello-world/` that has been parallelised using OpenMP. It will first print out a hello message (in serial) after which each thread will print out an "X" character (in parallel).

Compile this so that you enable OpenMP, and run it with 1, 2 and 4 threads. Do you get the expected amount of “X”s?

2. Parallel region and data sharing

Take a look at the exercise skeleton in `data-sharing/`. 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 you get different results.

3. Work sharing for a simple loop

In `work-sharing/` is a skeleton implementation for a 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.

4. Vector sum and race condition

In `race-condition/` you will find a serial code that sums up all the elements of vector A, initialized as $A = [1,2, \dots, N]$. For reference, from the arithmetic sum formula we know that the result should be $S = N(N+1)/2$.

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?

5. Reduction and critical

Continue with the previous 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. An example solution can be found in `race-condition/`.

6. Using the OpenMP library functions

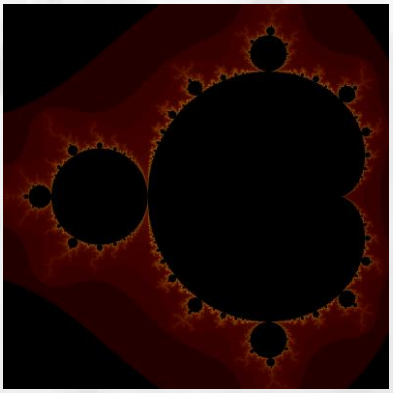
Modify the Hello World program in `hello-world/`, so that it uses **`omp_get_num_threads()`** and **`omp_get_thread_num()`** library functions and prints out the total number of active threads as well as the ID of each thread. An example solution can be found in `lib-funcs/`.

7. 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.

Example solutions can be found in `openmp-loops/` and `openmp/`, respectively.

8. (Bonus) Using OpenMP tasks for dynamic parallelization



Mandelbrot set of complex numbers can be presented as two dimensional fractal image. The computation of the pixel values is relatively time consuming, and the computational cost varies for each pixel. Thus, simply dividing the two dimensional grid evenly to threads leads into load imbalance and suboptimal performance.

The file `tasks/mandelbrot.c|F90` contains a recursive implementation for calculating the Mandelbrot fractal, which can be parallelized dynamically with OpenMP tasks. Insert missing pragmas for parallelizing the code (look for TODOs in the source code), and investigate the scalability with varying number of threads.

MPI I: Introduction to MPI

1. Parallel Hello World

a) Write a simple parallel program that prints out something (e.g. “Hello world!”) 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 prints out also its rank and so that the process with rank 0 prints out the total number of MPI processes as well. A solution can be found in `mpi/hello-world/`.

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 the skeleton code found in `mpi/message-exchange/`.
b) Increase the message size to 100 000, recompile and run.

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 the skeleton code found in `mpi/message-chain/`.

b) Rewrite the program using **`MPI_Sendrecv`** instead of **`MPI_Send`** and **`MPI_Recv`**.
c) (Bonus) Employ **`MPI_PROC_NULL`** in treating the special cases of the first and last task in the chain.
d) (Bonus) Use the status parameter to find out how much data was received, and print out this piece of information for all receivers.

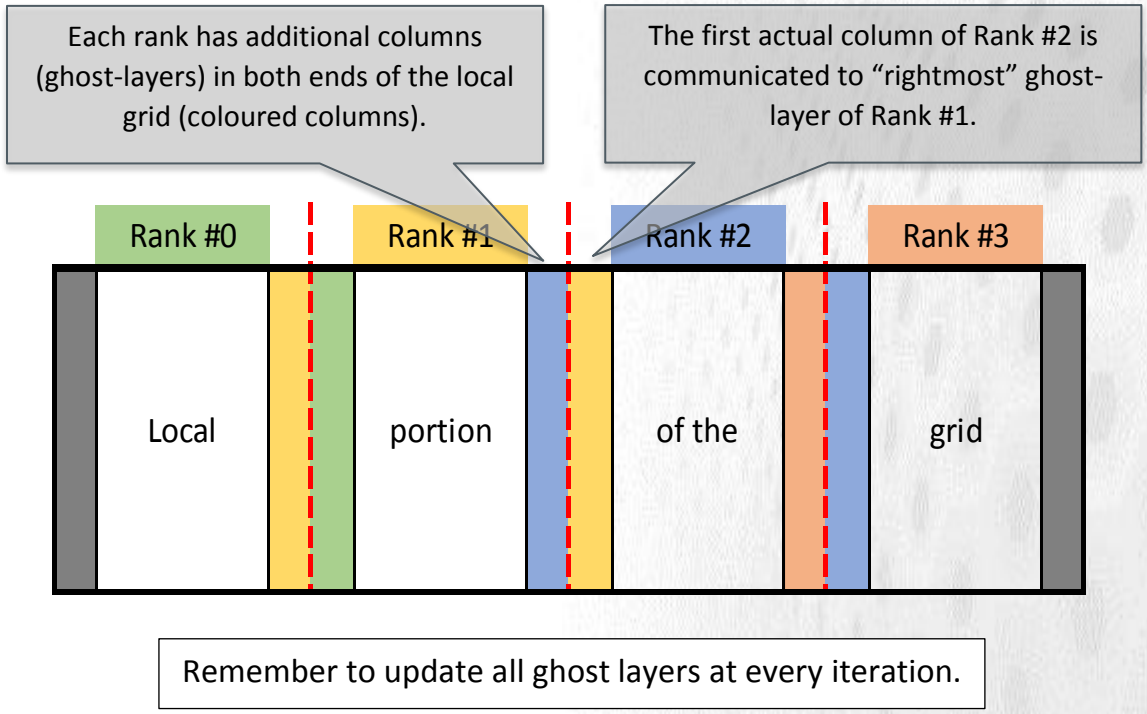
4. Parallel heat equation

Parallelize the whole heat equation program with MPI, by dividing the grid in blocks of 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 MPI 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 non-periodic; the outermost ranks communicate with single neighbor, and the inner ranks with the two neighbors.

Implement this "halo exchange" operation into the routine `exchange` by inserting the suitable MPI routines into skeleton codes available in `heat/mpi-p2p/`. You may use the provided makefiles for building the code.

A schematic representation of column-wise decomposition looks like:



Remember to update all ghost layers at every iteration.

MPI II: Collective operations

5. Collective operations

In this exercise we test different routines for collective communication. Write a program for four MPI processes, such that each process has a data vector with the following data:

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 communication that sends and receives values from these data vectors to the receive buffers using a **single collective** routine in each case, so that the receive buffers will have the following values:

a)

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

b)

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

c)

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

d)

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

You can start from scratch or use the skeleton code found in `mpi/collectives/`.

6. Communicators and collectives

Continue the previous exercise, now implementing a pattern with the receive buffer contains the following values:

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

An example solution can be found in `mpi/communicator/`.

MPI III: Non-blocking communication

7. Non-blocking communication

a) Go back to the exercise “Message chain” and implement it using non-blocking communication.
b) Revisit the one or more of the exercise “Collective operations” where you replace the operation(s) with their non-blocking counterparts.
c) Rewrite the Exercise a) with persistent communication operations.

Example solutions can be found in `non-blocking/`.

8. Non-blocking communication in heat equation

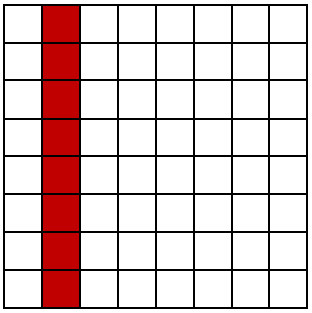
Implement the halo exchange in the heat equation solver using non-blocking communication. For some additional challenge, you can overlap the update (**`evolve`**) of the interior part of the domain (those independent on the halo areas) with the **`exchange`** routine. This approach implies that you will need to divide these into two stages both. The more straightforward approach is to replace just the communication operations in **`exchange`** with non-blocking operations. An example solution (following the more challenging task) can be found in `mpi_ip2p/`.

MPI IV: User-defined datatypes

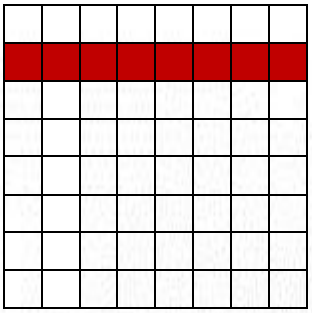
9. Using custom datatypes

Write a program that sends the highlighted elements of a 2D array using user defined datatypes from one MPI task to another. Note the different assignments for C and Fortran, and remember that C stores arrays in a row-major order and Fortran in a column-major order. You can start from skeleton codes in `datatypes`

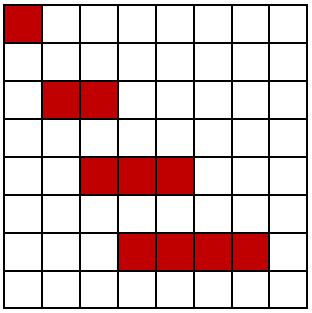
a) **C**



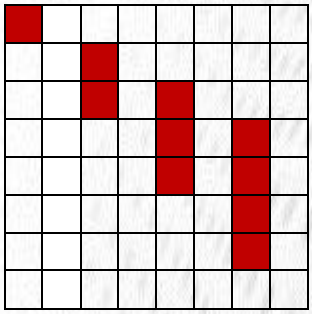
Fortran



b) **C**



Fortran



c) Write a program that sends an array of structures (derived types in Fortran) between two tasks. Implement a user-defined datatype that can be used for sending the structured data and verify that the communication is performed successfully. Check the size and true extent of your type. A skeleton code is provided in `datatypes/struct_type(.c|.F90)`.

d) Implement the send of structured data also by sending just a stream of bytes (type **`MPI_BYTE`**). Verify correctness and compare the performance of these two approaches.

10. (Bonus) Two-dimensional heat equation

Parallelize the heat equation in two dimensions and utilize user-defined datatypes in the halo exchange during the time evolution. Hint: MPI contains also a contiguous datatype **`MPI_Type_contiguous`** that can be used (together with **`MPI_Type_Vector`**) to construct user-defined datatypes for rows and columns to do the halo exchange in both x- and y-directions.

A skeleton code is provided in heat/`mpi-2d/`. Utilize user-defined datatypes also in the I/O related communication.

Note the Cartesian process topologies (the **`comm2d`** communicator) that have been employed to simplify the communication. These were not discussed in the lectures, but you can find some slides about them for further reading in the lecture notes.

Hybrid MPI+OpenMP programming

1. Hybrid Hello World

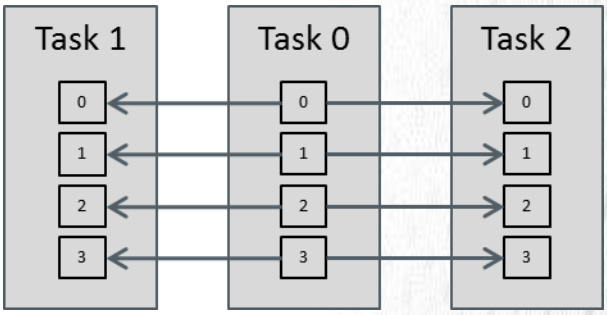
Write a simple hybrid MPI+OpenMP where threads are launched in MPI tasks. Have each thread to print out both the MPI rank and thread id. Investigate also the thread support level of underlying MPI implementation.

2. Heat equation solver hybridized with MPI+OpenMP

Refer back to the two OpenMP-parallelized implementations of the heat equation solver, and combine the loop-level parallelization with some previous MPI implementation of the solver, e.g. that of the module MPI I. This is the **`MPI_THREAD_FUNNELED`** mode.

3. Multiple thread communication

Write a simple hybrid program where each OpenMP thread communicates using MPI. Implement a test case where the threads of task 0 send their thread id to corresponding threads in other tasks (see the picture). Remember to employ the **`MPI_THREAD_MULTIPLE`** thread support mode (on Sisu it needs to be enabled by setting the environment variable **`MPICH_MAX_THREAD_SAFETY=multiple`**).



4. Hybrid heat equation solver revisited

Now combine the OpenMP implementation, where the threads are kept alive throughout the program execution, with the same MPI version. Now the MPI communication in the halo exchange is carried out in the **`MPI_THREAD_SERIALIZED`** or **`MPI_THREAD_MULTIPLE`** mode.

Parallel I/O

1. Parallel I/O with Posix

a) Write data from all MPI tasks to a single file using the spokesman strategy. Gather data to a single MPI task and write it to a file. The data should be kept in the order of the MPI ranks.
b) Verify the above write by reading the file using the spokesman strategy. Use different number of MPI tasks than in writing.
c) Implement the above write so that all the MPI tasks write in to separate files.

Skeleton codes and example solutions are found in `mpi/posix-io/`.

2. Simple MPI-I/O

Modify the previous "Parallel I/O with Posix" exercise so that all MPI tasks participate in the writing/reading in to a single file using MPI-IO. An example solution can be found in `mpi/mpi-io/`.

3. Reading & writing 2D arrays with MPI-I/O

In this exercise we will modify the heat equation solver program so that it can write out its state, and restart from it. A starting point and an example solution are provided in heat/`mpi-io/`

a) Add code to `write_restart` so that the application writes out a file called HEAT_RESTART.dat, describing the state using MPI-I/O routines. See the TODO's for the detailed specification of what to write. Use a collective write routine for the large data array.
b) Add code to `read_restart` so that the application can read the file. If the file exists, then it will try to restart from it. Check that the results look sane.

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 a solver for the 2D equation implemented with Fortran (including some C for printing out the images). 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 100 iterations. You can change that from the parameter image_interval. You can visualise the images using the command animate: `animate heat_*.png`.