EXERCISE ASSIGNMENTS

General exercise instructions

Computing servers

We will use CSC’s Cray XC40 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:  
module load emacs

emacs –nw prog.F90  
emacs prog.F90

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

Simple compilation and execution

Compilation and execution are done via the ftn (and cc for C programs as needed) wrapper commands and the aprun scheduler:

% ftn –o my\_exe test.F90

% cc –o my\_C\_exe test.c  
% aprun –n 1 ./my\_exe

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

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

When requiring multiple cores (in case of coarrays exercises), just increment the –n switch for aprun, e.g. –n 4 for running with 4 images.

Skeleton codes

For most of the exercises, skeleton codes are provided in the exercises folder, under a corresponding subdirectory. Generally, you should look for sections marked with “TODO” for completing the exercises.

Useful new features

1. Operating system utilities
2. Get all command line arguments into a string and print it.
3. Get number of arguments, get them one-by-one (also command itself) and print them.
4. Find out the value of environment variable $HOME.
5. Execute a Unix-command 'echo $HOME' and check the exit status.
6. New allocatable features

Take two integer arrays and try

* Automatic allocation
* Automatic resizing
* Allocating an array in a function
* Moving an allocation from an allocated an initialized array to an unallocated array.

Give a try also for the “allocatable scalar” feature by defining an allocatable character string and allocating and initializing it in one go. See answers/ex2/alloc.F90.

1. New pointer features and contiguous attribute

Run the program ex3/pointer.F90 to see how the new pointer features work. Then fix ex3/contiguous.F90 such that you get the diagonal as intended.

1. Experimenting with asynchronous I/O

Compile the program ex4/async.F90 with Cray compiler and run it to see, whether asynchronous I/O is in fact asynchronous on Sisu. Do the same with Intel and GNU compilers.

Language interoperability

1. Call C function from Fortran

Supply the correct C binding interface block for the C function DotProduct.

1. Accessing global C data from Fortran
2. Modify Fortran module file (ex6/globalmod.F90) to map C variables correctly to Fortran representation.
3. Repeat run with Intel compiler, too and realize that the C struct may not be optimal from data alignment point of view. Fix the alignment in C-code make sure Fortran gets corrected, too.
4. Fix the Fortran main program (ex6/global.F90) to get a correct C-to-F pointer mapping.
5. Implement a function “mygetenv”

This mygetenv-function should take a null-terminated Fortran string as an input and return a Fortran string. It should call the getenv function from C library directly. This can be accomplished by use of assignment operator that maps type(c\_ptr) i.e., char \* in C-language, to Fortran character strings. You will need to provide the interface definition to enable getenv calls from Fortran.

Fortran coarrays

1. Hello World revisited

Write the “hello world” program demonstrated in the lectures, but revising it such that the images print their messages in consecutive order, i.e.,  
Hello! From Image 1  
Hello! From Image 2  
...

1. Reading input data

Write a test program where a parameter is being read in by the image #1 from the keyboard and then distributed to other images.

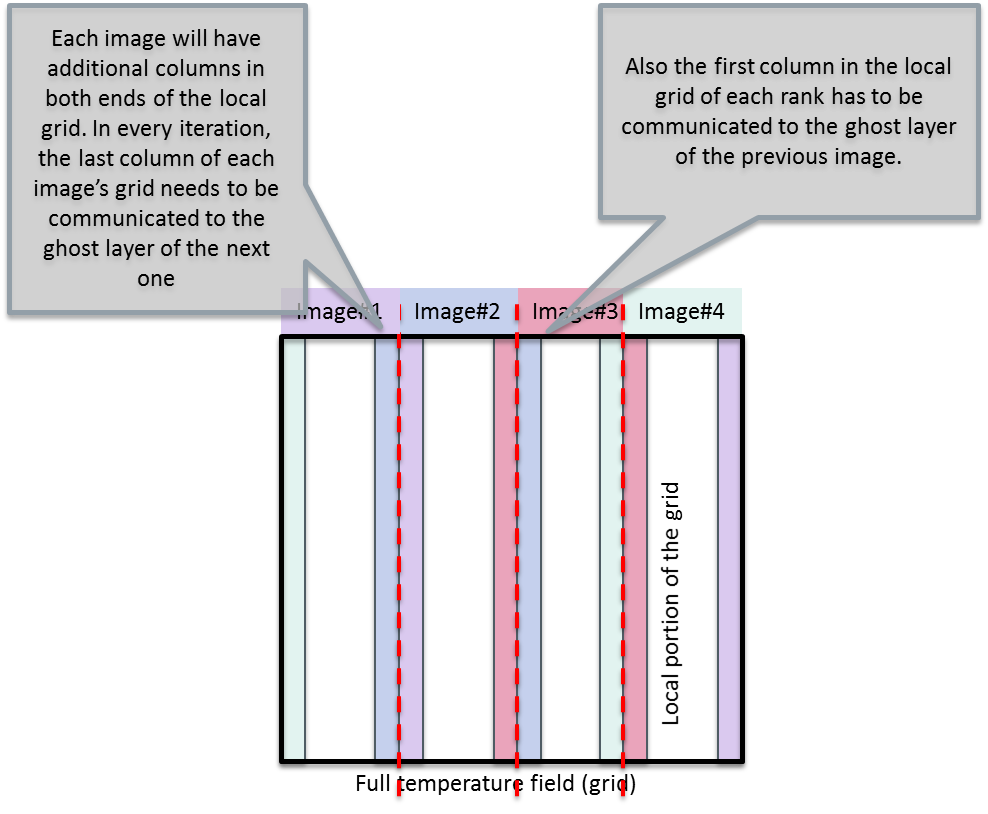
1. Global maximum

Write a program where all images generate a set of (dfferent) random numbers, and find the maximum value of them. Then the rank #1 finds the global maximum out of the local maxima and prints it out.

1. Parallelizing a heat equation solver with coarrays

Here we will parallelize a 2D heat equation solver (see the Appendix) using coarrays. We will start from a complete serial Fortran implementation of the solver (exercises/ex11, heat\_serial.F90 and heat\_serial\_main.F90). Take your time to get acquainted with the program and how to build (“make serial”) and run it.

The approach is to partition the grid into domains column-wise and have the whole grid declared as a coarray. The images are able to make the update in their domains independently (i.e. simultaneously) besides the outermost (domain-bordering) grid points, for which data has to be read from the neighboring images.



We will then parallelize it with coarrays as follows:

* Declare the temperature grids to be coarrays throughout the code. We can restrict ourselves to have the grid width being evenly dividable by the number of images (if you want some challenge, you can lift this restriction for a general case). Major modifications to the initialize subroutine will be needed to have a matching initial conditions with the serial implementation.
* Implement a new routine for “halo swap” i.e. performing the update of the ghost layers columns (see above). Let us call it exchange.
* The output routine needs to be adapted for either doing the single-writer I/O (i.e. gathering the full grid to one image), or alternatively, all images can print their local portions of the grid.
* Time and energy permitting, also reading in an input file (read\_input) can be converted to support coarrays parallelization.

A reference coarray implementation is being provided in the answers/ex11 folder, heat\_coarray.F90 and heat\_coarray\_main.F90. You can build it by “make parallel”.

**Types and procedure pointers**

1. Component visibility and generic interfaces

Modify the skeleton exercises/ex12/compvis.F90 to hide as much of the person\_t components as possible and write set\_name, set\_age and init\_person\_t. Also make the generic interface “person\_t” refer to init\_person\_t function.

1. Abstract interfaces and procedure pointers

The skeleton for this exercise is provided in exercises/ex13/absif.F90 file. Do the following modifications:

1. Create two abstract interfaces. One for a function that takes two vectors as an input arguments and returns a vector and another for a function that takes same arguments, but returns a scalar.
2. Create actual implementations for functions computing the sum, elementwise product and dot product of two vectors.
3. Initialize two vectors v1=(1,2,3,4,5) and v2=(5,4,3,2,1) and compute v3=v1+v2, v3=v1.\*v2 (elementwise product) and v3=v1\*v2 (dot product). Do the type constructors work as expected?
4. Create functions vecfun(fun,v1,v2) and scalfun(fun,v1,v2) which take as their first argument scalar or vector functions with the interface created in the part a) of this exercise. Is it possible to compute the results of part c) of this exercise by using the functions vecfun and scalfun?

Object-oriented features

1. Type-bound procedures

Skeleton files together with a Makefile can be found from exercise folder exercises/ex14.

1. Add functions created in the Exercise 13 as type-bound procedures of the given vector type.
2. Implement operators for the type-bound procedures that you implemented in part a). Note that you can also use operator names like .prod. in addition to \*.
3. Procedure pointers

Write map function that takes a pointer to function *f* mapping integers to integers, and integer array **A=[***x*1,*x*2,…]as arguments and returns allocated array containing [*f(x1),f(x2),…].* Skeleton files for this exercise are provided in the exercises/ex15 folder.

1. Object-bound procedures

Extend the idea of Exercise 15 by creating array of parametrized functions: *fi(x)=x+i, x=1,..,3,* and *gi(xi)=x-i, i=1,..,3* Hint: you need to encapsulate functions *f(x,i)* and *g(x,i)* inside at type that contains procedure pointer and the parameter *i*. Skeleton files for this exercise are provided in the exercises/ex16 folder.

Complex data structures in Fortran

1. Memoization

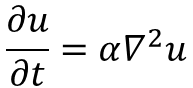
Fill in the type declaration data\_t and functions initdata and calcdata so that the data structure acts as a memoized sum of two integers used as shown in the main program. Use the initdata as an initializer for the data\_t structure. Skeleton files for this exercise are provided in the exercises/ex17 folder.

1. Linked list

Complete the linked list data structure and associated type-bound procedures for append, get and finalizer. Also, implement a procedure that checks whether or not the keys in a given linked list are in ascending order. Hint: implement the < operator for a node. The finalizer should print the key-data pair of each node as it deallocates them. Skeleton files for this exercise are provided in the exercises/ex18 folder.

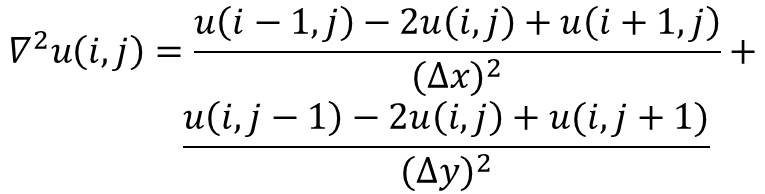
Appendix: Heat equation solver

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



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



Where ∆x and ∆y 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 ∆t:



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:

1. ./heat (no arguments - the program will run with the default arguments: 256x256 grid and 500 time steps)
2. ./heat bottle.dat (one argument - start from a temperature grid provided in the file bottle.dat for the default number of time steps)
3. ./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)
4. ./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 image\_interval iterations.