Advanced Fortran Topics - Hands On Sessions

Inhalt

Introductory Comments

- Before you start the first exercise, please read **the appropriate chapter from the organizational document** to perform setup of the course environment
- There is **no need to do *all*** the exercises.
- You are bound to make mistakes. Therefore, writing test programs (comparing expected with actual results) is a vitally important step in the development cycle. And one learns more from the mistakes than from the successes.
- Source code including **examples** from the talk and **solutions for the exercises** will be provided at the end of each day via the download URL:
  https://doku.lrz.de/display/PUBLIC/PRACE+Course%3A+Advanced+Fortran+Topics

  (You can use the `wget` command to perform downloads via UNIX command line by doing copy and paste of an archive name URL).

Exercises for Day 1

Session 1 – Reverse communication interface

Some knowledge of OpenMP is useful for doing this exercise.

The directory `hands_on/skel/reverse_communication` contains example code for the integration routine that uses reverse communication, as discussed in the slides.

a. Why is that code (still) not thread-safe? You can check the fact by activating the OpenMP clauses in the calling program and running with OMP_NUM_THREADS set to a value larger than 1.
b. Re-design the code to be thread-safe without adding any OpenMP directives in the module `mod_integration`. Hint: Introduce a derived type.

Session 1 – Operations on sparse matrices

The derived type `sparse` mentioned in the slides can be used to implement sparse matrices, and thus a sparse matrix-vector multiplication. A sparse matrix is represented by a rank-1 array

```
TYPE( sparse ), ALLOCATABLE :: sa(:)
```

which is allocated and subsequently has its (few) matrix elements set via repeated invocation of a module procedure `set_element()` on each array element of `sa` (the latter is intended to correspond to a matrix row in form of a linked list ordered by ascending index values). The folder `hands_on/skel/sparse` contains files `mod_sparse.f90` and `test_sparse.f90` copies of which you will need to suitably modify:

a. Add a module function that implements the multiplication of a sparse matrix by a `real(dk)` vector and overload the multiplication operator.
b. Run the resulting program and check the results. Then, check for memory leaks by building with debugging options and running the command through Valgrind:

   ```
   valgrind --tool=memcheck --leak-check=yes --track-origins=yes \
            --show-reachable=yes ./test_my_sparse.exe
   ```

   Where does the observed problem come from?
c. Consider further changes to your program to avoid the problem.

Two slightly different solutions are provided in the `hands_on/solutions/sparse` folder as

- `mod_sparse_simple.f90` and `test_sparse_simple.f90`
- `mod_sparse.f90` and `test_sparse.f90`

Note: the sparse implementation demonstrated here is horribly inefficient, because pointer chasing causes loss of spatial locality. Much better-performing storage schemes for sparse matrices exist, which you should use in productive environments.

Session 1 – Sparse matrices continued

In the slides, an overloaded structure constructor was shown for an object of type `sparse`. This suggests a much more efficient way of storing the row of a sparse matrix. Create a new folder, copy the module to it and improve the type definition. Apart from adding a constructor overload, you will also need to modify the implementations of the module procedures, of course. Then check that the main program still executes with the same results without any changes to it. Finally, create a new main program that uses the overloaded constructor instead of setting elements individually. The solution to this exercise will be contained in the folder `hands_on/solutions/sparse_crs.`

Session 2 – Physical bodies

Starting from the type definition

```
TYPE :: body
  REAL( dk ) :: mass
  REAL( dk ) :: pos(3), vel(3)
END TYPE
```

please define two extensions of that type (in the same module that defines the above type) which describe

- a body that has an electrical charge,
- a body that rotates.

Then write a procedure `print_body` that prints out all components of an object whose dynamic type may be any of the three types described above.

In a main program, please declare an allocatable polymorphic variable of declared type `body`, allocate it to be in turn any of the three types above, define reasonable values, and invoke `print_body` on that object.

Suppose a physical body starts out being of `TYPE( body )` but later acquires a charge (for example, by being struck by lightning). How can the previous values of the non-charge type components be retained when the object changes its dynamic type?

Finally, declare an unlimited polymorphic entity in the main program and allocate it to be of one of the above types. How can you invoke `print_body` on that object?

You can start out from the skeleton code given in `hands_on/skel/polymorphic_body`. The solution will be available in `hands_on/solutions/polymorphic_body`.

Session 2 – Implement the date class

1. Fill in all necessary details needed for implementation of the date and datetime types discussed in the lecture. Some subroutines which do the necessary computational stuff are provided in `hands_on/skel/datetime`. Please start out by implementing the missing subroutines `inc_date` and `inc_datetime` as well as overloading the structure constructor for the above types.
2. Bind the procedures written in part 1 of this exercise as well as `write_date` to their appropriate types and uncomment the statements in the test program that execute them, to check that your code works correctly.

3. Suppose you have a type definition

   ```
   TYPE :: person
     PRIVATE
     CHARACTER( LEN=nmx ) :: name
     CLASS( date ), POINTER :: birthday => null()
   END TYPE
   ```
   and functions that initialize objects of that type or return the name or birthday of a person. How can you print out the birthday of a person without needing to copy an object? The datetime skeleton also contains a file `person.f90` that you can start out from.

The solution for this exercise is in the folder `hands_on/solutions/datetime`. **Note:** Because many compilers mishandle the code intended for part 3., please check against the result produced by the NAG compiler.

Session 2 – Geometric objects

A rectangle might be characterized by the type definition

```
TYPE :: rectangle
  PRIVATE
  REAL( dk ) :: length = 0.0_dk
  REAL( dk ) :: breadth = 0.0_dk
END TYPE
```
Implement a type-bound procedure which calculates the area of a rectangle, as well as an initialization procedure and a procedure to perform a stretching of a rectangle along a chosen direction. Since a square is a special kind of rectangle, one might consider the following type definition:

```
TYPE, EXTENDS( rectangle ) :: square
… ! fill in missing bits
END TYPE
```
How do you handle the inheritance behavior, especially that of the type-bound procedures? Assume that a function `adjust` exists that takes an object of `TYPE( rectangle )` as an argument and increases the size of the object if its area is too small. What happens if an entity of `TYPE( square )` is used as an actual argument in an invocation of this function? Try to improve on the type design.

Please consult the two Fortran codes in `hands_on/solutions/shapes` for the "bad" and "good" versions of the design.

Session 2 – improving an interface class

The folder `hands_on/skel/interface_class` contains a runnable example for the interface class discussed in the lecture. Unfortunately, the main program still contains a dependency on the module that declares the type extension. What is the cause of the problem? Try to find a solution such that the statement "`use mod_file_handle`" can be replaced by "`use mod_handle`".

The solution will be available in `hands_on/solutions/interface_class`.

Exercises for Day 2

Session 3 – Collision of physical bodies

Write a type-bound procedure that calculates the front-on collision of physical bodies, starting from the code written for the above exercise. The following assumptions may be made: For the base type, the conservation of energy and momentum defines the result. In the center-of-mass system that moves with velocity

$$\vec{V} = (m_1\vec{v_1} + m_2\vec{v_2})/(m_1 + m_2)$$

the velocities $\vec{w_j}' = \vec{v_j}' - \vec{V}$after the collision can be obtained from the corresponding $\vec{w_i}$ before the collision by

$$m_1\vec{w}'_1 + m_2\vec{w}'_2 = 0 = m_1\vec{w_1} + m_2\vec{w_2} \quad m_1w_1^2 + m_2w_2^2 = m_1{w'}_1^2 + m_2{w'}_2^2$$

For two spherical charged bodies that collide the charges after the collision are defined by

$$\frac{q'_1}{R_1} = \frac{q'_2}{R_2}$$

and you can assume that the electrostatic interaction is negligible. The radii R_1, R_2 of the objects shall be supplied through the interface.

The solution will be available in `hands_on/solutions/polymorphic_body`.

Session 3 – Using asynchronous I/O

The directory `hands_on/skel/aio` contains source code for a ray tracer. This code performs I/O of the complete picture array at the end of the calculation. The resulting file can be viewed with the `display` command. Convert this program to use asynchronous I/O by putting the data transfer statements inside the outer loop that processes the tiles in the picture. In particular, you can reduce the needed amount of storage from `size**2` to `size*nbuf`, where `nbuf` is the number of I/O buffers available for asynchronous I/O. For which picture size do you observe a performance advantage?

The solution will be available in `hands_on/solutions/aio.`

Note: For good I/O performance on the LRZ HPC systems it is recommended to change to the $SCRATCH directory and execute the program there.

Session 3 – User-defined derived type I/O for sparse matrices

Starting from the solutions for the sparse matrix multiplication from session 2, add a generic binding and module procedures that support formatted as well as list-directed output of objects of type sparse. Modify the test program to exercise these features.

The solution for this exercise is contained in the folder `hands_on/solutions/sparse` (files `mod_sparse_io.f90` and `test_sparse_io.f90`).

Session 4 – Parameterized derived types

Take a look at the skeleton code you find inside `hands_on/skel/pdt/.`

1. As indicated, define an abstract parameterized derived type `matT` with kind parameter k and len parameters m, n. Then, define a type extension with a rank-2 real array component `mat`. In the main program, declare allocatable objects of the kind indicated in the skeleton main program. Further, define an assignment that can handle different `kind` parameters for `rmatT`, e.g. real32 and real64 and any `len` parameter. Also define a matrix multiplication that can handle different `kind` parameters for `rmatT`, e.g. real32 and real64. Try to get the indicated code fragments in the main program running.
2. Now add the `private` attribute to the parameterized type `rmatT`. What is required to get the main program to compile and run again?

Session 4 – Trajectories of Physical Bodies

Define a type that – reusing code from the Session 4 exercise "Physical Bodies" - can describe the trajectory of a particle of `CLASS(body)`. This means that the physical state of the particle at some number of points in time must be stored inside an entity of such a type.

1. What type components must be defined to fully describe such a trajectory? Write an overloaded (generic) structure constructor for an entity of that type that defines an initial condition for the trajectory.
2. Write a procedure that updates the trajectory by adding an additional time step, assuming that mass or charge of the particle do not change. Deal with the storage limitation of an entity of `TYPE(trajectory)` by writing data to disk if necessary, without changing existing interfaces.
3. What do you need to do if an entity of `TYPE(trajectory)` goes out of scope or is deallocated?

Please start out with the skeleton code in `hands_on/skel/trajectory`. The entries marked FIXME indicate where missing bits need to be added; a test program that runs the code is also supplied. Reference output is provided in a subdirectory.

Session 4 – Integration example

Fill in the details on implementing the integration example from today's lecture. Some code which provides a simple default integrator is available in `hands_on/skel/integration`, as well as code that indicates how Fortran GSL interface functions (which provide more refined capabilities) should be used**.** Please note that the main program needs to change (read the FIXME entries there), and the `qdr.f90` module needs appropriate updates.

Exercises for Day 3

Session 5 (easy) – Calling GSL from Fortran

Write a program that calculates the value of the error function

$$\mathrm{erf}(x) \coloneqq \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\,dt$$

for values of the argument between 0.0 and 2.0. Use the implementation supplied by the GNU scientific library and use the variant of the function that also calculates the error estimate. Documentation is available at https://www.gnu.org/software/gsl/manual/ and the Makefile provides the necessary variables (`$GSL_LIB`) to link against the library. A C implementation is available in `hands_on/skel/interop_gsl`, and the solution for this exercise will be contained in `hands_on/solutions/interop_gsl`

Note that a nearly complete Fortran interface to the library is available via
https://github.com/reinh-bader/fgsl/

Session 5 (more difficult) – Using a POSIX call from Fortran

The API call `getpwnam()` allows to extract information about a specific user name from the authentication databases. Write a Fortran module and program that contains the necessary type definition, BIND(C) interfaces and module procedures to produce and print out a user's HOME directory. What must be taken care of in case a user name does not exist? Study the man page for `getpwnam()` for information on its interface.

Skeleton code for the exercise is in `hands_on/skel/interop_struct`. A fully working main program is available, but the module `mod_passwd` must be implemented.

The solution for this exercise will be found in `hands_on/solutions/interop_struct`.

Session 5 – Dynamic dummy arguments

In the lecture, we have seen how C-defined types with pointer components can be handled in Fortran 2003. Now consider the reverse: How can the Fortran interface

```
SUBROUTINE generate_data(arr)
  REAL(c_float), ALLOCATABLE, INTENT(OUT) :: arr(:)
END SUBROUTINE
```

- which is non-interoperable (why?) - be used from C? Assume that the data for `arr` is generated within the subroutine and that these data should be accessible within C. See `hands_on/skel/interop_deferred_shape` for skeleton code.

With the Fortran 2018 extensions to C interoperability, it is permitted to directly use interfaces such as the above, i.e. it is possible to add BIND(C) to the above interface. This removes the need to introduce a “handle” derived type. Implement a C main program to do such a call after correctly setting up a descriptor for the allocatable object.

The solution for this exercise will be found in
`hands_on/solutions/interop_deferred_shape.`

Session 5 – C library interface with function argument

The folder `hands_on/skel/interop_interface` contains source files `c_libcall.[c,h]` implementing a library call with prototype

```
float Sum_fun( float (*fun)(float x, void *params), void *params );
```

which takes a function argument to be provided by the client. Furthermore, the following files are supplied:

- an example main program in C (`cmain.exe`) which you can build using the `Makefile` – you will need to inspect this to see how the argument function is implemented.
- skeletons for a Fortran library module `f_lib.f90` and a Fortran main program `fmain.f90`, which will not successfully compile ...

Here's what you are required to do:

1. Please add the missing functionality in the Fortran code using only Fortran 2003-style interoperability features, so it performs the same function as the C code. What limitations apply?
2. Using the additional features from Fortran 2018, extend the semantics of the interface so that also non-interoperable arguments can be used from Fortran. Check that this works by calling the function with a polymorphic actual argument.

The solutions will be provided in `hands_on/solutions/interop_interface`.

General comments on coarray programming

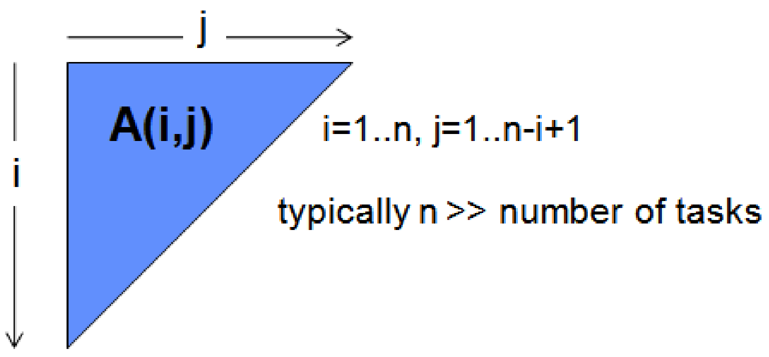
To compile and execute coarray programs, you need to have one of the following compilers installed on your system:

- The Intel Fortran compiler, version 19.0 or higher (the most recent version is available via https://software.intel.com/content/www/us/en/develop/tools/oneapi.html)
- The gfortran compiler, version 9 or higher, together with the Opencoarrays package (see http://www.opencoarrays.org/ for more information)
- The NAG Fortran compiler, version 7 or higher

If you work on the LRZ HPC systems, please consult the web page at https://doku.lrz.de/display/PUBLIC/Coarray+Fortran+on+LRZ%27s+HPC+systems to inform yourself about how parallel Fortran programs are compiled and started.

Session 6 – Data distribution

Consider a triangular matrix:



a. Make a copy of the serial program `hands_on/skel/triangle/triangular.f90` into your working directory. The program reads in matrix size and a row index from the command line. It then sets up

$$A(i, j) = i + j$$

and prints out the specified row. Parallelize this program in a manner that distributes data as evenly as possible across images.

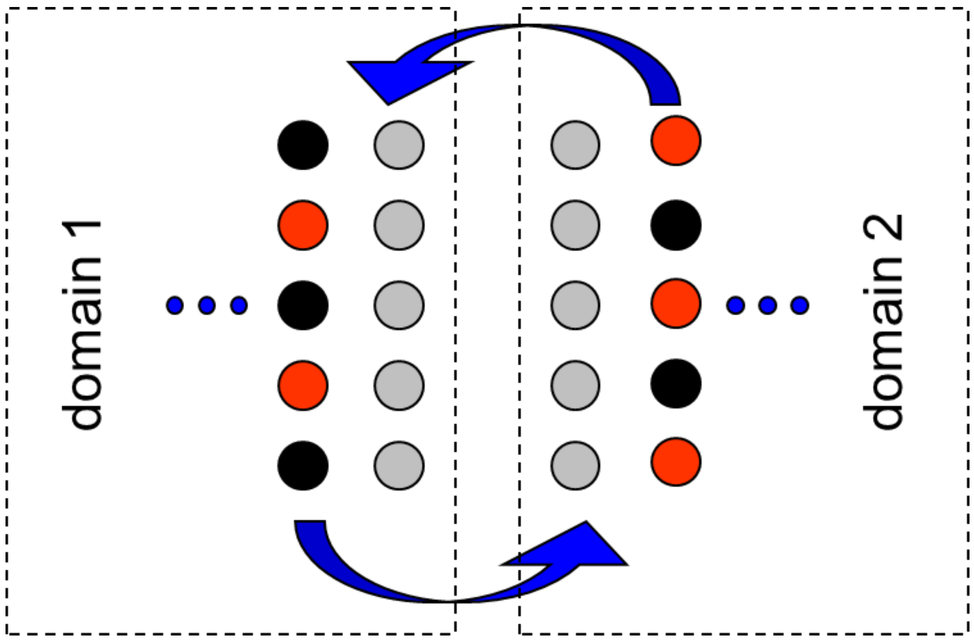
b. Implement the full matrix-vector multiplication for triangular matrices. Use the most efficient access pattern for calculation of the scalar products. The coarray used to store one of the vectors should be allocatable. Do not forget to implement the necessary communication to assure the complete result is available on all images. Is it possible to use a collective function for this purpose?

The solutions for this problem will be available in the folder `hands_on/solutions/triangle`

Session 6 – Heat conduction parallelized

Starting out from the skeletons `mod_heat.f90` and `heat.f90` introduced on day 1, parallelize this code using coarrays.

a. Introduce a one-dimensional domain decomposition along the y direction. One method to deal with the boundary cell problem consists in assigning each image an additional column (“halo” or “ghost” cells, colored grey in the figure below) at its boundaries to another domain which receives data from the task that hosts that domain:



Only the halo cells need to be involved in communication; in this example, these form contiguous arrays. For now, please only run a fixed (sufficiently large) number of iterations, omitting the termination criterion.

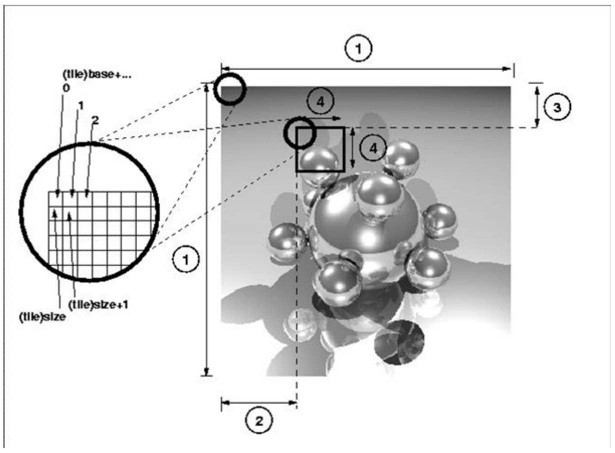
b. Start with a small problem size to check whether correct results (the printout from multiple tasks should be collected on image 1) are obtained.
c. When running with a problem size of 200 x 200, up to how many images does your code scale?
d. Add the necessary reduction call that enables consistent evaluation of the termination condition. Note: Not all compilers support collectives yet.

The solutions for this problem will be available in the folder `hands_on/solutions/heat_caf` (files `mod_heat_static.f90` and `heat_static.f90`).

Exercises for Day 4

Session 7 – parallelizing a ray tracer

The subdirectory `hands_on/skel/aio` contains a serial ray-tracer code (the same we've seen before in a different context), which computes a pretty picture. The central function is `calc_tile()`, which computes one tile of the picture. The size of one tile and of the whole picture is hardcoded at the start of the main program. Note that the code assumes that the picture size is a multiple of the tile size. In the version given, the picture size is 4000 x 4000 and the tile size is 200 x 200.



Parallelize the code. Several possible strategies exist, but the variant we suggest uses **events** to coordinate the I/O between images. No coarrays beyond the event variable itself will be needed. Make sure that your parallel code computes the correct result (this is easy since you can always display the picture). What speedup does your code obtain going from 1 to the maximum reasonable number of images? Also, compare with the baseline performance from the serial code.

The folder `hands_on/solutions/ray` will contain the solution for this exercise and further variants of it.

Session 7 – parallel library call

For the matrix-vector multiply, rewrite the main program to use a library call with a coarray dummy argument.

Session 8 – Separate execution of even and odd images

In a program unit, declare a coarray

```
INTEGER :: index[*]
index = this_image()
```

and then set up two teams constituted of the even and odd images, respectively. Print out all team-local values of `index` from the first image of each team. Can you assure that the printout from the “odd” team is always preceded by the printout from the “even” team?

The solution for this exercise is contained in `hands_on/solutions/teams`.

Session 8 – Coindexing in teams and team-local allocation

In a program unit, declare allocatable coarrays

```
INTEGER, ALLOCATABLE :: i_init(:)[:], i_team(:)[:]
```

Allocate `i_init` to an array of size 1, and initialize it with the value `this_image()`. Then, decompose the initial team into subteams with three images each (except, possibly, the last one), and change execution to the subteam context. On the first image of each team, print out the value

- on the last image of the “right neighbour” team
- on the last image of the initial team

Then, still inside the subteam context, allocate `i_team` to size 1 on subteam 1, size 2 on subteam 2 etc. Print out the result of `size(i_team)` on image 1 of each subteam in order of the subteam identifiers. Hint: use events to perform the necessary serialization. After leaving the subteam context, check the allocation status of `i_init` and `i_team`.