A practical guide to the usage of scientific software at CEN

Software group

Hamburg, August 18, 2025

Contents

1. Introductory notes

This document is intended for users recently arrived to a CEN (Centrum für Erdsystemforschung und Nachhaltigkeit) institute of the University of Hamburg, who are not familiar with the steps needed to deploy their computing applications (a numerical model, for instance) in the available computing systems. The document covers topics like source code compilation and linking against libraries, job submission in the local workstation or in a cluster and output post-processing possibilities. The brief descriptions of the installed software packages given here are taken from the respective sources, which can be found by following the hyperlinks given throughout the text.

In this practical guide it is assumed that the user 1) is using either a local Desktop computer running a Debian Linux distribution or using a local Thin Client terminal connected to a central server and 2) wants to deploy a computer application either locally in his/her Linux machine or in the central Linux compute cluster "marin".

Before moving on to the next chapters, we suggest the new user first reads the material included in the CEN-IT web page `http://www.cen.uni-hamburg.de/facilities/cen-it.html` containing all necessary basic information for a quick start in the computer environment. Whereas detailed information about the Unix operating system (GNU/Linux) should be searched elsewhere.

CEN-IT (CEN IT Services) provides IT support to all institutes clustered in CEN. The support includes hardware procurement and repair, software installation and maintenance, mail and websites management and server administration. In case of special problems and questions, which are not solved after reading this document and the material found in the CEN-IT web pages, please send an email request to the CEN-IT Help-Desk using *helpdesk.cen@uni-hamburg.de*.

Just like the software tree described ahead, this document is also being constantly updated. The user is therefore asked to search for the newest version of this document in the CEN-IT web page or under `/data/share/CIS/cen_sw_guide`. This document comes with supplementary material, which can be found under `/data/share/CIS/cen_sw_guide/suppl`.

2. The scientific software tree

This first section introduces the user to the ways of accessing the scientific software packages installed in the central servers. This software tree is automatically mounted on every workstation or server residing inside the network.

In order to guide the user's shell to executables located within the software tree, their locations have to be listed within the user's PATH environment variable. This is accomplished in two ways. In the first case, some paths (`/bin`, `/usr/bin`, `/usr/local/bin`, `/client/bin`, etc) are set automatically when the user logs in. In those folders, executables or symbolic links to executables exist, so that the user can use these packages at all times without further action. In the second case the paths are to be set by the "module" system, what requires special user action in the form of commands that make the package available. This module system allows the user to access without much effort specific software packages installed on the central servers and to set the correct paths to executables and libraries.

2.1. The /client/bin location

In the local folders `/bin` and `/usr/bin` the user will find system tools, like editors and shells and some scientific tools, like the system perl, the system python or the system C/Fortran compilers. The software packages installed locally are provided by the operating system distribution and can only be upgraded as part of a general operating system upgrade (for instance when upgrading from Debian Linux 10 a.k.a "buster" to Debian Linux 12 a.k.a "bookworm").

In the case of office-related packages with no need to choose between different versions and which do not require special environment variable setup, symbolic links to those package executables, manual pages and info document files are simply created in the folder `/client`, within the appropriate directory (`/bin`, `/sbin`, `/man` or `/share`). These directories are appended automatically to the environment variables in the global shell startup scripts allowing the user to use these packages without any action. Please note that headers, libraries or data belonging to those packages are not linked and therefore the user should search for them in the system if needed.

At login time `/client/bin` is therefore added to the user's PATH environment variable so that the software in `/client/bin` can be started from the command line by just typing the program's command name. In `/client/bin` the user will find links to software administrated by CEN-IT for which only one version is provided and which do not require special setup. This is the case of email clients, web browsers and non-scientific office tools. The most relevant packages found under `/client/bin` are:

- Libreoffice - Open source personal productivity office suite: Writer, Calc, Impress, Draw, Math and Base (`http://www.libreoffice.org`)
- Rdesktop - Open source client for Windows remote desktop services (`http://www.rdesktop.org`)

In `/client/bin` only the symbolic links to the main commands of each package listed above exist. The packages themselves are installed in the central software tree, under `/sw/<os-name>`, where `<os-name>` is one of the Linux distributions available and maintained by CEN-IT (see below).

- `bookworm-x64` (Debian GNU/Linux 12 "bookworm", 64 bits)
- `centos7-x64` (CentOS Linux 7, 64 bits)

2.2. The module system

All the scientific software to be used in the Linux-based computers is made available through modules (`http://modules.sourceforge.net/`). Each module contains the information needed to configure the shell for an application by setting the appropriate environment variables. To be able to use a certain software package (and with a specific version), all that is needed to do is to load the corresponding module and then type the application's executable name on the command line.

The module system is therefore a command line tool to manage environments of software installed in the central software tree. Some key advantages of using the module system are: (1) each user can immediately view the current software configuration; (2) it is easy to switch between different software versions; (3) paths and other environment variables are automatically set.

The most important commands of the module system are:

- `module avail` – Displays all modules that are available (but not necessarily loaded) on the system.
- `module list` – Displays all modules that are currently loaded.
- `module load <module-name>` – Loads the module `<module-name>`. This command needs to be executed before the software package can be used.
- `module unload <module-name>` – Unloads the module `<module-name>`. This command removes all the changes to the shell environments executed when loading the module.
- `module switch <old-module-name> <new-module-name>` – Unloads `<old-module-name>` and then loads `<new-module-name>`. This command can be used to switch software versions.

- `module show <module-name>` – Shows information about the modification/creation of the PATH, MANPATH or other environment variables.
- `module purge` – Unloads all default and previously loaded modules.

By convention, module names (as listed by the commands `module avail` or `module list`) have the format `<package-name>/<package-version>`. If the user wants to list all versions of a single package that are installed in the software tree, the command `module avail <package-name>` can be issued. For only a few packages a module with name `<package-name>/default` exists. This corresponds to the standard version of the package as explicitly defined by CEN-IT. The standard version can be in some cases the most recent version or in other cases the most stable release of the software. To load the standard version it is sufficient to do `module load <package-name>`, i.e. without the version specified.

A practical example of usage is given next. If the user wants to use the default Intel Fortran 90 compiler (i.e., the nost recent version), the usage is:

```
module load intel
```

To switch from the default version to some specific version (e.g., 24.0.2), one needs to do:

```
module switch intel intel/24.0.2
```

To completely erase the access to the loaded Intel compiler, the syntax is:

```
module unload intel/24.0.2
```

Note that no modules are loaded by default, so, for frequently used software packages, the user can add the `module load <module-name>` instructions to the login shell script file (`.cshrc` or `.bashrc`). If some software version is available only on a specific platform, the user can load the module only on that platform by nesting the `module load ...` command in one platform-specific `if` block in the login shell script. An example is given below for the cases

if using tcsh:

```
if ( `lsb_release -cs` == "bookworm" ) then
module load intel
module load cdo/2.3.0-gccsys
module load matlab/2024b
endif
```

or if using bash:

```
if [ "$(lsb_release -cs)" == "bookworm" ]; then
module load intel
module load cdo/2.3.0-gccsys
module load matlab/2024b
fi
```

For the users working on a workstation running the Debian "bookworm" distribution, the following packages are available through the module system. They include compilers, libraries, general-purpose and specific-use geophysical processing packages. The most relevant packages used for post-processing of model and observational geophysical data will be presented in more detail in section 5.

```
atom/1.63                          intel_inspector/2024               ↩
    netcdf_fortran/4.6.1
axisem3d/04062025                  intel_vtune/2023                   norsar/3d↩
    -2022.0
cdo/2.3.0-gccsys                   intel_vtune/2024                   norsar/3d↩
    -2023.1
cdo/2.5.0-gccsys                   intel_vtune/2025                   octave↩
    /8.3.0
cmake/3.27.7                       jdk/1.8.0_144                      openfoam/12
comsol/5.2a                        jdk/21.0.7                         openfoam↩
    /12-precompiled
default                            julia/1.11.2                       openmpi↩
    /4.1.6-gccsys
ferret/7.6.0                       julia/1.9.3                        openmpi↩
    /5.0.5-gccsys
filezilla/3.65.0                   lfortran/0.41.0                    openmpi↩
    /5.0.5-static-gcc132
flow3d/2022.1                      maple/2023                         openmpi↩
    /5.0.5-static-intel24
flow3d/2023.2                      maple/2024                         openmpi↩
    /5.0.5-static-intel25
gcc/13.2.0                         mathematica/13.3.1                 openmpi↩
    /5.0.7-gccsys
gcc/15.1.0                         mathematica/14.1                   openmpi↩
    /5.0.7-static-gcc132
gdal/3.7.2                         matlab/2023b                       openmpi↩
    /5.0.7-static-gcc151
geos/3.12.0                        matlab/2024a                       openmpi↩
    /5.0.7-static-intel25
git/2.42.0                         matlab/2024b                       panoply↩
    /5.2.9
git/2.49.0                         matlab/2025a                       paraview↩
    /5.11.2
gmt/6.4.0                          mendeley/2.101.0                   paraview↩
    /5.13.0
google_earth/7.3.6                 miniforge3/24.7.1                  paraview↩
    /5.13.3
grads/2.2.1                        mpich/4.1.2-gccsys                 pycharm↩
    /2024.2.3
hdf/4.2.16                         mpich/4.2.2-gccsys                 r/4.3.1
hdf5/1.14.2                        mpich/4.2.2-static-gcc132          reveal/5.0
idl/8.8.3                          mpich/4.2.2-static-intel24         reveal/6.0
idl/9.1                            mpich/4.2.2-static-intel25         rstudio↩
    /2023.09
intel/23.2.1                       mpich/4.3.0-gccsys                 rstudio↩
    /2024.09
intel/24.0.1                       mpich/4.3.0-static-gcc132          ruby/3.2.2
intel/24.2.0                       mpich/4.3.0-static-gcc151          scilab↩
    /2023.1.0
```

```
intel/25.0.1                          mpich/4.3.0-static-intel25          seismicunix↩
    /4.4.r11-gccsys
intel/25.0.4                          ncl/6.3.0-nodap-precompiled         seismicunix↩
    /4.4.r11-noxdr-gccsys
intel/25.1.0                          nco/5.1.8                           seismicunix↩
    /4.4.r28-gccsys
intel/25.1.1                          ncview/2.1.9                        seismicunix↩
    /4.4.r28-noxdr-gccsys
intel_inspector/2023                  netcdf/c-4.9.2-fortran-4.6.1-cxx4-4.3.1 seismicunix↩
    /default
```

2.3. The /sw structure

As previously stated, scientific software packages not provided by the vendor of the operating system are served by a central server and (always) available via the directory `/sw`, which is a link to `/opt/cen/sw`. The command `df -h /sw` shows the filesystem ("nfs4.isi.cen.uni-hamburg.de":`/ifs/cen/sw`) serving the `/opt/cen/sw` mount.

The software installation system enforces a consistent naming scheme for each package installation directory. The general form is `/sw/<os-name>/<package-name>-<package-version>-<special-feature>-<compiler>`

where

- `<os-name>` is the operating system name;
- `<package-name>` is the software package name;
- `<package-version>` is the software package version;
- `<special-feature>`, if present, might be:
  - `precompiled`, if the installation was not performed from source code;
  - `static`, if the installation does not contain shared libraries;
  - `openmpi`, to indicate the message passing interface (MPI) implementation the package has been linked against;
- `<compiler>`, if present, indicates that the package can only be used with a certain compiler.

Static versions of central software packages are installed in order for the user to avoid problems arising from the usage of libraries that are not on the linker's path. For some packages, shared versions are also available. More on static and shared libraries will be given in the next chapter. The user is here just warned to the existence of both types of installed libraries in the software tree.

The `<compiler>` tag can be, for instance, `gcc132` or `intel25` and reflects the compiler used during installation. More on the available compilers is also given in the next chapter.

3. Building an application

In this section a practical example is used to demonstrate the resources available to build a user application.

Creating an executable of a computer application is a process that can be divided into two components: first the ”compilation” and second the ”linking”. The term ”building” can be used to refer to the whole process of going from the source code to the executable, thus encompassing both steps.

During the compilation phase, the source code files written either in C, Fortran or other language (the .c, .f, .f90, ... files) are interpreted by the compiler to produce machine language instructions corresponding to the source code lines. The output of compilation alone are ”object” files (.o files) which can not be executed. The linking phase, on the other hand, uses the object files and creates one executable file.

3.1. Compilers available

A brief description of the four C/Fortran compilers available for the Linux platforms is given next together with the way to access them with the ”module” system. Please note that the compiler versions given below correspond to the most recent version installed at the date of writing of this document. The software tree has, however, earlier versions which are kept for a certain period of time to insure usage consistency for a specific task (a research project, a Ph.D. or M.Sc. work, etc).

3.1.1. Gnu Compiler Collection

The GNU Compiler Collection (`http://gcc.gnu.org`) includes front-ends for C, C++, Objective-C, Fortran, Java, Ada, and Go, as well as libraries for these languages (libstdc++, libgcj,...).

```
access: module load gcc/13.2.0
commands: gcc, c++, cpp, g++, gcov, gfortran, ...
```

3.1.2. Intel Compilers

The Intel OneAPI product (`https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html`) offers C, C++ and Fortran compilers combined with the performance-oriented Intel Math Kernel Library, Intel Integrated Performance Primitives and Intel Threading Building Blocks.

```
access: module load intel/25.1.0
commands: icc, ifort, ...
```

3.2. Compiling the source code

In order to illustrate the use of the above tools to develop a user application, we are going to make use of the Fortran source code described in Appendix A. The Fortran program numerically solves the shallow water equations, a set of coupled partial differential equations widely used in Geophysical appplications. The reader interested in more details about the model, namely the approximations it encloses and the way to discretize and solve the set of equations, should at this point consult the appendix. The source code in its simplest form can be found there. The respective file is in the folder /data/share/CIS/cen_sw_guide/suppl/case1. Several modifications of that original source code will be developed throughout this section; the respective files can be found in /data/share/CIS/cen_sw_guide/suppl/case[2-5]. In order to try out the code and follow the steps described in this guide, the user is asked to first copy the contents of folder /data/share/CIS/cen_sw_guide into a directory of his/her choice, either in the home directory or under /scratch.

The Fortran source code in folder "case1" is composed of a single file, containing a main program and a subroutine. Throughout the whole guide the GNU compiler "gfortran" will be used. In principle, building this single source code would simply be:

```
gfortran -o sw model.f
```

The flag -o allows the user to specify the name of the output executable. For other compiler flags, the user should consult the respective manual page by issuing (in this particular case of the GNU compiler) "man gfortran".

The above compilation command can in fact be split into two steps:

```
gfortran -c model.f
gfortran -o sw model.o
```

which are the two phases of first compiling the source code into an object and then linking the object and called libraries into an executable. The compiler also serves as linker but in reality it is calling the command "ld" with some options.

The subroutine present in the source code is called to perform the model output in NetCDF (Network Common Data Format - `http://www.unidata.ucar.edu/software/netcdf`) and, to that end, it uses functions from the NetCDF Fortran library and includes a special header file (the "netcdf.inc"). So, a dependency is built

in the subroutine to functions external to it. So, in reality, the compilation and linking steps above will fail since the NetCDF-related functions are unknown to the compiler. The correct way is 1) to include the NetCDF headers at compilation time using flag "-I" and 2) to link against the library containing the netCDF functions using flag "-L".

```
gfortran -c model.f -I<netcdf-include-dir>
gfortran -o sw model.o -L<netcdf-library-dir> -lnetcdff
```

The NetCDF package is installed in the software tree and is compiled separately for all the available compilers. The proper way to compile and link against libraries is explored next.

3.3. Linking against libraries

There are two types of libraries: the static and the shared libraries. Linkers give preference to shared over static libraries if both are found in the location given by the "-L" flag. This can lead to problems when executing the application since at runtime it might be that the shared libraries are not found. In this case, users have to set the variable LD_LIBRARY_PATH to point to the shared library location or encode the runtime library search path into their executables.

3.3.1. Shared libraries

Shared libraries have the extension *.so. The main difference between these and the static is that the libraries are not included at linking time, but only at run time. At linking time only information is retained in the generated program about which libraries are to be used at run time. The runtime linker then looks at the start of a program for the required libraries, loads all in memory and only fills the placeholders for the function addresses.

There are some advantages to the use of shared libraries:

- Less storage, since there is no copy of the library in the executable.
- Libraries can be easily replaced without re-compilation.
- Programs can also load shared libraries with certain operating system functions (e.g., plugins).
- Shared libraries will find further dependencies. The user must only indicate the direct dependencies.

The correct compilation and linking of the "case1" code against the shared version of the netCDF library is:

```
gfortran -c model.f -I/sw/bookworm-x64/netcdf_fortran-4.6.1-gccsys/include

gfortran -o sw model.o -L/sw/bookworm-x64/netcdf_fortran-4.6.1-gccsys/lib -lnetcdff

or (equivalent):

gfortran -o sw model.o /sw/bookworm-x64/netcdf_fortran-4.6.1-gccsys/lib/libnetcdff.so
```

The runtime linker will try to find the shared library "libnetcdff.so" in the following places:

- If set, in the list of paths in $LD_LIBRARY_PATH, the analog of $PATH for libraries;
- In the system standard places like /lib, /usr/lib and /usr/local/lib.
- In the list encoded in the program, passed with the flag "-L";

Encoding a list in the executable is done with flag "-L" as above. However, it will not suffice in this case since a shared version of the library is also found in /usr/lib, what is given preference. So, additionally, one must add the flag "-rpath <path to library>" in the linking step. Because this option must be passed from the compiler to the linker, the syntax is not obvious:

```
gfortran -o sw model.o -L/sw/bookworm-x64/netcdf_fortran-4.6.1-gccsys/lib -Wl,-rpath -Wl,/↩
    sw/bookworm-x64/netcdf_fortran-4.6.1-gccsys/lib -lnetcdff
```

As the RPATH is now explicitly encoded in the executable, the program will take exactly the shared library residing in /sw/bookworm-x64/netcdf_fortran-4.6.1-gccsys/lib and not the one in /usr/lib. There are some commands the user can use to check which paths for shared libraries are encoded:

- "ldd <executable>" Lists which shared libraries are loaded and where.
- "readelf -a <executable> | grep RPATH" : Lists the list of rpaths encoded in the binary.
- "nm -D <executable>" : Lists which functions are used/provided by a shared library.

3.3.2. Static libraries

To avoid problems at run time, the software tree also offers static versions of certain packages (those with the tag "static" under /sw/<os-name>)

The static libraries are simply archives (similar to tar archives) with extension *.a that contain object files (*.o). Instead of a collection of object files, the *.a static library file

can be directly given to the linker. In principle, building the application with static libraries could simply be:

```
gfortran -c model.f -I/sw/bookworm-x64/netcdf_fortran-4.6.1-gccsys/include
gfortran -o sw model.o -L/sw/bookworm-x64/netcdf_fortran-4.6.1-gccsys/lib -lnetcdff

or (equivalent):

gfortran -o sw model.o /sw/bookworm-x64/netcdf_fortran-4.6.1-gccsys/lib/libnetcdff.a
```

However the above leads to "undefined reference" errors, i.e., additional function dependencies that arise since the static version of the netCDF library was linked with other static libraries (like the HDF5, SZIP, CURL, etc). All direct and indirect dependencies must be specified, otherwise there will be an error. The order of the paths and libraries given by the "-L" flag is also important since the linker searches only in the subsequent libraries for suitable functions.

So, linking against a static library is not so straightforward but has the advantage that all needed libraries will be integrated into the executable (the latter becomes, of course, larger) and therefore there are no problems during run time. To ease the process of static linking there is in every netCDF installation (under /sw/<os-name>/netcdf_fortran-<version>-static-<compiler>/bin) one script named "nf-config" which tells the user the exact syntax:

```
module load intel

/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/bin/nf-config --flibs

which gives

-L/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/lib -lnetcdff -L/sw/bookworm-x64/↩
    libaec-1.1.2-gccsys/lib -L/sw/bookworm-x64/hdf5-1.14.2-static-gccsys/lib -L/sw/bookworm↩
    -x64/netcdf_c-4.9.2-static-gccsys/lib -lnetcdf -lnetcdf -lm -lhdf5_hl -lhdf5 -lsz -lz -↩
    lzip -lcurl -lm -ldl
```

The user only has to copy the output of "nf-config –flibs" to his/her linking command. The final correct static building sequence of the source code in "case1" is:

```
ifx -O3 -c model.f -I/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/include
ifx -O3 -o sw model.o -L/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/lib -lnetcdff ↩
    -L/sw/bookworm-x64/libaec-1.1.2-gccsys/lib -L/sw/bookworm-x64/hdf5-1.14.2-static-gccsys↩
    /lib -L/sw/bookworm-x64/netcdf_c-4.9.2-static-gccsys/lib -lnetcdf -lnetcdf -lm -↩
    lhdf5_hl -lhdf5 -lsz -lz -lzip -lcurl -lm -ldl
```

The choice of which netCDF library to use from those in /sw should obey the following:

1. First of all, of course, use one that resides in the folder of the target operating system, e.g., /sw/bookworm-x64;
2. Use one of those with "static" in their name to avoid problems with "rpath";

3. Use the one corresponding to the compiler that has been used to compile the source code, e.g., netcdf_fortran-4.6.1-static-intel25, if the compiler of choice is the Intel compiler version 25.*;

Several other numeric libraries are installed in the software tree. They can be found under /sw/spack<os-name>/<package-name>. A selected list is given below:

```
arpack-ng/3.9.0
fftw/3.3.10
igraph/0.10.6
netlib-lapack/3.11.0
netlib-scalapack/2.2.0
parmetis/4.0.3
petsc/3.20.0
qhull/2020.2
qrupdate/1.1.2
scalasca/2.6.1
suite-sparse-5.13.0
```

3.4. Developing the application

3.4.1. Multiple source code files

When the source code becomes very large it is of good practice to divide it into separate more easily-manageable files. It is natural to do a separation into subroutines that deal with a certain aspect of the application. This is exemplified in /data/share/-CIS/cen_sw_guide/suppl/case2, where the initial source code is split into a main program file and subroutine files:

```
main.f - calls the subroutines and contains the main loop
initial.f - generates the initial conditions
loop_u.f - steps forward the x-momentum equation
loop_v.f - steps forward the y-momentum equation
loop_eta.f - steps forward the continuity equation
bound.f - applies the boundary conditions
out_cdf.f - outputs the model results in NetCDF format
```

Note as well that a header file (”param.h”) is now included in the main program and in all subroutines with the line:

```
include 'param.h'
```

It contains the variable declarations and some definitions of constants. This way, these declarations have to be done only at one place and the user can very easily change their values.

The two steps taken above for compiling and linking are similar; only that now the compiler will have to generate object files for each of the source code files. The *.o

files are then statically linked together (and with all the dependent libraries: netcdf, hdf, etc) at the linking stage to create one executable program.

```
ifx -c main.f
ifx -c initial.f
ifx -c loop_u.f
ifx -c loop_v.f
ifx -c loop_eta.f
ifx -c bound.f
ifx -c out_cdf.f -I/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/include
ifx -o sw main.o initial.o loop_u.o loop_v.o loop_eta.o bound.o out_cdf.o -L/sw/bookworm-↩
    x64/netcdf_fortran-4.6.1-static-intel25/lib -lnetcdff -L/sw/bookworm-x64/libaec-1.1.2-↩
    gccsys/lib -L/sw/bookworm-x64/hdf5-1.14.2-static-gccsys/lib -L/sw/bookworm-x64/netcdf_c↩
    -4.9.2-static-gccsys/lib -lnetcdf -lnetcdf -lm -lhdf5_hl -lhdf5 -lsz -lz -lzip -lcurl -↩
    lm -ldl
```

3.4.2. C preprocessing

The C preprocessor implements the macro language used to transform C, Fortran, etc programs before they are compiled. It is useful, for instance, to activate/deactivate parts of the source code, thus allowing to include some optional features in the application. More detailed information about C preprocessing can be found under `http://gcc.gnu.org/onlinedocs/cpp`.

Under /data/share/CIS/cen_sw_guide/suppl/case3 the reader can find a modification of the "case2" source code that exemplifies the use of C preprocessing to generate optional features. The main program ("main.F") has now one option and the "initial.F" subroutine has now three options. At the very beginning of these two source code files there is now a file (cppdefs.h) being included. The syntax is (e.g., for the main program):

```
      program main

#include "cppdefs.h"

      Fortran code here

#ifdef OPTION1
      Fortran code here
#endif

      Fortran code here

#ifdef OPTION2
      Fortran code here
#endif

      Fortran code here

     end
```

In file cppdefs.h there is simply a list of options that are either defined or undefined, as follows:

```
#define OPTION1
#undef OPTION2
```

Building the application needs now a step before compilation. The source code has to be run first through the C preprocessor ("cpp"). The source code files, that now have the extension *.F (to denote that it has C directives inside), will be interpreted at the light of the options defined in cppdefs.h and will then be written to files with extension *.f, which can be further compiled and linked as before.

```
cpp -P -traditional main.F > main.f
cpp -P -traditional initial.F > initial.f
cpp -P -traditional loop_u.F > loop_u.f
cpp -P -traditional loop_v.F > loop_v.f
cpp -P -traditional loop_eta.F > loop_eta.f
cpp -P -traditional bound.F > bound.f
cpp -P -traditional out_cdf.F > out_cdf.f
ifx -O3 -c main.f
ifx -O3 -c initial.f
ifx -O3 -c loop_u.f
ifx -O3 -c loop_v.f
ifx -O3 -c loop_eta.f
ifx -O3 -c bound.f
ifx -O3 -c out_cdf.f -I/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/include
ifx -o sw main.o initial.o loop_u.o loop_v.o loop_eta.o bound.o out_cdf.o -L/sw/bookworm-↩
    x64/netcdf_fortran-4.6.1-static-intel25/lib -lnetcdff -L/sw/bookworm-x64/libaec-1.1.2-↩
    gccsys/lib -L/sw/bookworm-x64/hdf5-1.14.2-static-gccsys/lib -L/sw/bookworm-x64/netcdf_c↩
    -4.9.2-static-gccsys/lib -lnetcdf -lnetcdf -lm -lhdf5_hl -lhdf5 -lsz -lz -lzip -lcurl -↩
    lm -ldl
```

3.5. Parallelizing the application

When the application becomes too large a problem to be solved by one single processor, maybe because of lack of memory or the computation takes too long to complete, the only way is to distribute the task among several processors which then run the job in parallel. There are two main ways of parallelizing a source code, depending if the target is a shared-memory system or a distributed-memory system. In the first case the system is composed of several processors (called threads) sharing the same memory (e.g., a dual- or quad-core machine) and in the second case is a system of several processors (called nodes) each having its own memory (e.g., a cluster of single-processor machines connected by a fast interconnect). In practice, the nowadays systems are a mixture of both (i.e., clusters of multiple-core machines).

There are, accordingly, two parallel programing modes, one designed for share-memory systems, the OpenMP (Open Multi-Processing - `http://openmp.org`), and another initially designed for distributed memory machines but that nowadays also support shared-memory, the MPI (Message Passing Interface). MPI is a widely accepted standard for communication among nodes that run a parallel program on a distributed-memory system. The interface is a library of routines that can be called from C

or Fortran programs to pass messages between processes on the same computer or on different computers. Therefore, MPI can be used to program shared-memory or distributed-memory applications. There are two important open-source implementations of MPI: OpenMPI and MVAPICH2, both are installed in the software tree.

3.5.1. OpenMPI

The Open MPI Project (`http://www.open-mpi.org`) is an open source implementation of the Message Passing Interface standard that is developed and maintained by a consortium of academic, research, and industry partners.

```
access: module load openmpi/5.0.7-static-<compiler>

where <compiler> can be one of the above: gcc132, intel25

commands: mpicc, mpic++, mpiCC, mpif77, mpif90, mpirun, ...
```

3.5.2. MPICH

MPICH (`http://www.mpich.org/`) is a high-performance and widely portable implementation of the Message Passing Interface standard.

```
access: module load  mpich/4.3.0-static-<compiler>

where <compiler> can be one of the above: gcc132, intel25

commands: mpicc, mpic++, mpiCC, mpif77, mpif90, mpirun, ...
```

The complexity of programming with MPI depends on the type of parallelization envisaged (fine or coarse grained parallelization) and might or not require a different restructuring of the source code. In any case it involves the inclusion of calls to MPI subroutines. To keep the example simple in this guide, the parallelization here conducted is a domain decomposition, i.e., the model grid is horizontally split into tiles and each tile is given to a single processor, which advances the calculation in its domain and sends/receives data to/from the adjacent tiles. Here, modifications to the code involve not just calls to MPI subroutines but also splitting the ”do-loops” into tiles.

In folder /data/share/CIS/cen_sw_guide/suppl/case4 the user can find the parallelized version of ”case3”. The important changes are in ”main.F” and in the ”loop_*.F” subroutines. The main program has, as a first thing to do, to initialize the MPI environment and, as a last step, to finalize the MPI environment (see below). In-between these two calls, all instructions will be executed by every process except otherwise stated. One way of making only a dedicated process execute part of the calculations is to explicitly associate part of the code with a certain process by stating its rank

(variable "myrank" below), which is given by the "call MPI_COMM_RANK". This is done in the loop_*.F subroutines, where chunks of the "do-loops" are given to specific ranks with an "if-statement".

```
      program main

      include 'mpif.h'

! initialize distributed-memory MPI
      CALL MPI_INIT(ierr)

! get number of process in the group associated with the communicator
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
! get rank of the local process in the group associated with the communicator
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)

       Fortran code here

! finalize distributed-memory MPI
      CALL MPI_FINALIZE(ierr)

      end
```

But now the program is being effectively split into a number of processes (variable nprocs above), so that the information each one has at each instant is different. So, a communication between all processes is needed to exchange the information. There are a few ways of doing that, either by calls to MPI_SEND and MPI_RECV or, as chosen here, by calls to MPI_GATHER and MPI_SCATTER. "case4" is simple since it has only 2 processes (ranks 0 and 1) and the only task needed is to gather all information in each of them. In subroutines "loop_*.F" there are now calls to MPI_GATHER:

```
! gather from all processes the u variable from the local ul variable
      CALL MPI_GATHER(ul,size_ul,MPI_REAL,u,size_ul,MPI_REAL,0,MPI_COMM_WORLD,ierr)
      CALL MPI_GATHER(ul,size_ul,MPI_REAL,u,size_ul,MPI_REAL,1,MPI_COMM_WORLD,ierr)
```

The program then proceeds to the next subroutine and is executed by the 2 processes. Please note that the MPI usage in this example is very simplified. The user should consult a manual on MPI programming since it is not the scope of the present guide. The objective is to have a source code that can be executed in parallel so that the building process is mentioned as well as, further ahead in section 4, the procedure to run a parallel job.

Building the application is exactly as before, except with "gfortran" simply replaced by the MPI wrapper to the compiler, the "mpif90".

```
cpp -P -traditional main.F > main.f
cpp -P -traditional initial.F > initial.f
cpp -P -traditional loop_u.F > loop_u.f
cpp -P -traditional loop_v.F > loop_v.f
cpp -P -traditional loop_eta.F > loop_eta.f
cpp -P -traditional bound.F > bound.f
cpp -P -traditional out_cdf.F > out_cdf.f
mpif90 -O3 -c main.f
mpif90 -O3 -c initial.f
mpif90 -O3 -c loop_u.f
```

```
mpif90 -O3 -c loop_v.f
mpif90 -O3 -c loop_eta.f
mpif90 -O3 -c bound.f
mpif90 -O3 -c out_cdf.f -I/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/include
mpif90 -o sw main.o initial.o loop_u.o loop_v.o loop_eta.o bound.o out_cdf.o -L/sw/bookworm↩
    -x64/netcdf_fortran-4.6.1-static-intel25/lib -lnetcdff -L/sw/bookworm-x64/libaec-1.1.2-↩
    gccsys/lib -L/sw/bookworm-x64/hdf5-1.14.2-static-gccsys/lib -L/sw/bookworm-x64/netcdf_c↩
    -4.9.2-static-gccsys/lib -lnetcdf -lnetcdf -lm -lhdf5_hl -lhdf5 -lsz -lz -lzip -lcurl -↩
    lm -ldl
```

3.6. Building the application with a Makefile

This section describes the usage of the ”make” program and the contents of the file it uses to automatically build a user application, the so-called ”Makefile”. That file contains the relationships between the source, object and executable files and allows the user to easily manage the source code from large applications. Instead of typing the compiling and linking commands manually for every source code file and every time something is changed in the code, the ”make” utility can be issued and perform this task with absolutely no effort. Furthermore, the make program keeps track of which part of the source code has changed since last compilation, so that only those parts are re-compiled.

Command ”make” works according to file dependencies; for instance, it knows that in order to create an object file at least one *.f or *.c file is needed. All these dependencies can be specified in the Makefile, which should reside in the same directory as the source files. The ”make” utility also checks modification times of the files and if one of the source code files is modified it will run again its compilation.

The most basic form of a ”Makefile” is as follows:

```
target : source file(s)
    command (must be preceded by a tab)
```

A target given in the ”Makefile” is a file which will be created or updated when any of its source files are modified. There is a default target for makefiles called ”all”. The ”make” utility will first execute this target or, in its absence, the first target, if no specific one is specified. The commands given in the subsequent lines are executed in order to create the target file.

In our ”case1” the process is divided into two steps, corresponding again to the compilation and linking phases:

```
sw: model.o
    ifx -O3 -o sw model.o -L/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/lib -↩
        lnetcdff -L/sw/bookworm-x64/libaec-1.1.2-gccsys/lib -L/sw/bookworm-x64/hdf5-1.14.2-↩
        static-gccsys/lib -L/sw/bookworm-x64/netcdf_c-4.9.2-static-gccsys/lib -lnetcdf -↩
        lnetcdf -lm -lhdf5_hl -lhdf5 -lsz -lz -lzip -lcurl -lm -ldl
model.o:
    ifx -O3 -c model.f -I/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/include
```

In the above ”Makefile”, the first target is the name of the application executable ”sw”. In order to ”make” it the Makefile is saying that an object ”model.o” is needed (i.e., a dependency exists), and therefore a compilation of the source code ”model.f” is required. This automatically will execute the second target (and that will occur before the first target). The second target performs the usual compilation. After that, the command of the first target, the linking phase, is executed.

Once the ”Makefile” is created, the ”make” command can be run by simply typing the following in the command line:

```
make -f <make-filename>
```

If the <make-filename> is simply called ”Makefile” or ”makefile” than typing ”make” suffices. Specific targets listed in the ”Makefile” can also be specified; this way only that target (and its corresponding source files) will be executed.

The ”make” program allows the use of macros to store names of files, similarly to environment variables. The syntax can be as follows:

```
OBJECTS = main.o bound.o initial.o
```

Command ”make” will then use this definition in other parts of the ”Makefile”; the user simply needs to type $(OBJECTS).

Here is a ”Makefile” for ”case2” using a macro.

```
OBJECTS = main.o bound.o initial.o loop_eta.o loop_u.o loop_v.o out_cdf.o

sw: $(OBJECTS)
    ifx -O3 -o sw $(OBJECTS) -L/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/lib -↩
        lnetcdff -L/sw/bookworm-x64/libaec-1.1.2-gccsys/lib -L/sw/bookworm-x64/hdf5-1.14.2-↩
        static-gccsys/lib -L/sw/bookworm-x64/netcdf_c-4.9.2-static-gccsys/lib -lnetcdf -↩
        lnetcdf -lm -lhdf5_hl -lhdf5 -lsz -lz -lzip -lcurl -lm -ldl
main.o:
    ifx -c main.f -I/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/include
bound.o:
    ifx -c bound.f -I/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/include
initial.o:
    ifx -c initial.f -I/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/include
loop_u.o:
    ifx -c loop_u.f -I/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/include
loop_v.o:
    ifx -c loop_v.f -I/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/include
loop_eta.o:
    ifx -c loop_eta.f -I/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/include
out_cdf.o:
    ifx -c out_cdf.f -I/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/include
```

A macro value can also be specified when running make. This is useful when the user wants to specify, for instance, different compiler flags. If the ”Makefile” is:

```
FFLAGS=-O2

OBJECTS = main.o bound.o initial.o loop_eta.o loop_u.o loop_v.o out_cdf.o
```

```
sw: $(OBJECTS)
    ifx $(FFLAGS) -o sw $(OBJECTS) -L/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/↩
        lib -lnetcdff -L/sw/bookworm-x64/libaec-1.1.2-gccsys/lib -L/sw/bookworm-x64/hdf5↩
        -1.14.2-static-gccsys/lib -L/sw/bookworm-x64/netcdf_c-4.9.2-static-gccsys/lib -↩
        lnetcdf -lnetcdf -lm -lhdf5_hl -lhdf5 -lsz -lz -lzip -lcurl -lm -ldl
main.o:
    ifx $(FFLAGS) -c main.f -I/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/include
bound.o:
    ifx $(FFLAGS) -c bound.f -I/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/include
initial.o:
    ifx $(FFLAGS) -c initial.f -I/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/↩
        include
loop_u.o:
    ifx $(FFLAGS) -c loop_u.f -I/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/↩
        include
loop_v.o:
    ifx $(FFLAGS) -c loop_v.f -I/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/↩
        include
loop_eta.o:
    ifx $(FFLAGS) -c loop_eta.f -I/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/↩
        include
out_cdf.o:
    ifx $(FFLAGS) -c out_cdf.f -I/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/↩
        include
```

then the following will override the value of FFLAGS in the ”Makefile” and build the code with the newly defined compiler flags:

```
make 'FFLAGS=-O3'
```

There are some important macros used internally by the ”make” program.

```
$@      Full name of the current target.
$<      The source file of the current dependency.
\$?      A list of files for current dependency which are out-of-date.
```

The way macros are evaluated can also be manipulated. Assuming that a macro is defined as ”SOURCES = main.F initial.F” than using ”$(SOURCES:.F=.o)” within the ”Makefile” substitutes ”.F” at the end with ”.o”, allowing to have a new list but this time of objects.

By itself, ”make” knows already that in order to create a ”.o” file, it must use ”gfortran -c” on the corresponding ”.f” file. These rules are built into make, but they can be customized and other can be specified. Below is the ”Makefile” to be used with the source code under /data/share/CIS/cen_sw_guide/suppl/case5. It starts by defining how to treat each file type (according to its extension). For instance the rule ”.F.o” tells ”make” how to treat the ”.F” files and generate an object file out of them. In particular it says the ”*.F” files have to be run through the C preprocessor defined by macro $(CPP) and only afterwards they can be compiled by $(FC) to produce the object file. In the second part, some macros are defined, like the compiler and respective

flags, paths for the includes and libraries are given and the list of source files is written. At the end, there are two targets; the first builds the executable and the second cleans the intermediate and not further needed files.

```
#  Makefile:

.SUFFIXES: .o .f .F

.F.o:
    $(CPP) $(CPPFLAGS) $*.F > $*.f
    $(FC) -c $(FFLAGS) $(LDFLAGS) $*.f

.f.o:
    $(FC) -c $(FFLAGS) $(LDFLAGS) $<

.F:
    $(FC) -o $@ $(FFLAGS) $(LDFLAGS) $<

.f:
    $(FC) -o $@ $(FFLAGS) $(LDFLAGS) $<

CPP = cpp
CPPFLAGS = -P -traditional
FC = mpif90
FFLAGS = -O3
LIBS = -L/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/lib -lnetcdff -L/sw/bookworm-↩
    x64/libaec-1.1.2-gccsys/lib -L/sw/bookworm-x64/hdf5-1.14.2-static-gccsys/lib -L/sw/↩
    bookworm-x64/netcdf_c-4.9.2-static-gccsys/lib -lnetcdf -lnetcdf -lm -lhdf5_hl -lhdf5 -↩
    lsz -lz -lzip -lcurl -lm -ldl
INCLUDES = -I/sw/bookworm-x64/netcdf_fortran-4.6.1-static-intel25/include
LDR = mpif90
LDFLAGS = $(INCLUDES)
CC = mpicc
SHELL = /bin/sh

BIN = sw

SOURCES = main.F initial.F loop_u.F loop_v.F loop_eta.F bound.F out_cdf.F

OBJECTS = $(SOURCES:.F=.o)

$(BIN): $(OBJECTS)
    $(LDR) $(FFLAGS) $(LDFLAGS) -o $(BIN) $(OBJECTS) $(LIBS)

clean:
    rm -rf *.o *.f
```

4. Running the application

In this section the users will find information on how to run their applications, either locally in their own workstation or remotely on a central compute server.

4.1. Running on a local machine

Recently acquired workstations have more than one CPU cores and thus have multi-processing capabilities. The user can therefore use the MPI and OMP paradigms to parallelize their codes and link their applications against the openMPI and MPICH libraries (like it was performed in section 3). After loading the respective MPI module (openmpi in the example below), running the application can be done by using the command "mpirun":

```
module load openmpi/5.0.7-static-<compiler>

mpirun -np <number of processes> /<path-name>/<executable-name>

which in the "case5/parallel" of the shallow water model is:

module load openmpi/5.0.7-static-intel25

mpirun -np 2 ./sw
```

Note that when deploying the job across different workstations the flag "-machinefile <hostsfile>" is needed together with the flag "-prefix < /sw/<os-name>/mpi/openmpi-<version>-static-<compiler> >", the latter telling the remote machines where to find the "mpirun" command.

The application produces, of course, results. In terms of data storage, the Unix/Linux users have available:

- home directory disk space. Data can be stored permanently there (/home/zmaw/<user-id>) up to the amount of 20GB. The disk space is backed up, so, in case of an emergency, old versions of files can be restored.
- workstation local disk space. If the user works on a dedicated Linux workstation there will generally be disk space under /scratch/local1. There is no backup of data under /scratch, so data is lost in case of hard disk failures or accidental deletion of files.
- central server disk space. From a local workstation, the user is also connected to central server disk space, which resides under /data (either /data/cen or /data/share). When in cluster "marin" (see below), the user has also central server disk space available under /scratch. Those disk spaces belong, however, to dedicated projects or groups, so the user can only store data in the project/-group he/she belongs to.

4.2. Running on a computing cluster

The German Climate Compute Center - DKRZ (`http://www.dkrz.de`) provides the CEN users with High Performance Computing (HPC) capabilities, namely the Bull/Atos HLRE-4 ”levante”. Details about that system can be found in the DKRZ web page. CEN-IT maintains also one compute cluster, which CEN users can use to execute smaller dimension applications.

4.2.1. Cluster overview

The currently available system is the computer cluster ”marin”. It is accessible to all CEN users, although parts of it are (or will be) reserved to specific working groups from the University of Hamburg. Cluster ”marin” consists of several login nodes for interactive login and work connected to a large filesystem:

- Login node ”marin02”:
  - Processors: 2x Intel Xeon Gold 6248R, 3.00GHz = 44 Cores, 88 Threads
  - Memory: 730 GB RAM
- Login node ”marin03”:
  - Processors: 2x Intel Xeon E5-2697A v4, 2.60GHz = 30 Cores, 60 Threads
  - Memory: 970 GB RAM
- Login node ”marin04”:
  - Processors: 2x Intel Xeon E5-2697A v4, 2.60GHz = 30 Cores, 60 Threads
  - Memory: 970 GB RAM
- Login node ”marin05”:
  - Processors: AMD EPYC 7713P 64-Core = 60 Cores, 120 Threads
  - Memory: 970 GB RAM
- Login node ”marin06”:
  - Processors: 2x Intel Xeon Gold 6142, 2.60GHz = 30 Cores, 60 Threads
  - Memory: 730 GB RAM
- Login node ”marin07”:
  - Processors: 2x Intel Xeon Gold 6142, 2.60GHz = 30 Cores, 60 Threads

  - Memory: 730 GB RAM

- Login node "marin08":
  - Processors: 4x Intel Xeon Gold 6142, 2.60GHz = 52 Cores, 104 Threads
  - Memory: 600 GB RAM

The "marin" storage provides central disk space in a CEPH Filesystem (`https://www.ceph.com`) running on 8 Softiron HD11120 nodes. These nodes are invisible to users, who only see the file systems the storage nodes provide for the head nodes. Depending on the group/project they belong to, "marin" users can store data in /scratch/<project>. There are directory quotas on the project folders, so that all data below a project folder counts to this quota, independent of the owner or the group the file belongs to. Once the quota is reached, no data can be further stored and the members of the project will have to remove some data. No automated backup is made of the data stored under /scratch.

Project subfolders have to be created by CEN-IT. Users should look in /scratch if there is already a project they belong to. If the user does not find his/her project, the respective project/group leader and/or the master user should be contacted in order to tell CEN-IT to create the project folder.

5. Post processing utilities

In this last section, the users are introduced to the most relevant scientific software packages suitable for the advanced processing and visualization of geophysical observational or model data.

5.1. Geophysical data processing and visualization software

There is a large number of programs and libraries available in the /sw tree, as described in section 2, and the user can always issue the command "module avail" from the command line to check all previously and newly installed software. An overview of the most relevant software packages is here given.

CDO - Climate Data Operators

CDO (https://code.mpimet.mpg.de/projects/cdo) is a collection of command line operators to manipulate and analyze Climate data files. Supported file formats are GRIB, netCDF, SERVICE and EXTRA. There are more than 250 operators available.

```
access: module load cdo/2.5.0-gccsys
command: cdo
```

COMSOL Multiphysics

The COMSOL Multiphysics (https://www.comsol.com/comsol-multiphysics) is a commercial engineering simulation software environment that helps in all steps of the modeling process, like defining geometry, meshing, specifying physics, solving and then visualizing results.

```
access: module load comsol/5.2a
command: comsol
```

Ferret - Data visualization and Analysis

Ferret (https://ferret.pmel.noaa.gov/Ferret) is an interactive computer visualization and analysis environment designed to meet the needs of oceanographers and meteorologists analyzing large and complex gridded data sets.

```
access: module load ferret/7.6.0
command: ferret
```

FLOW-3D

FLOW-3D (http://www.flow3d.com) is a commercial software providing flow simulation solutions for engineer investigations on the dynamic behavior of liquids and gases, in particular, on the solution of time-dependent (transient), free-surface problems in one, two and three dimensions, and models confined flows and steady-state problems.

```
access: module load flow3d/2023.2
command: flow3d
```

GMT - Generic Mapping Tools

GMT (https://www.generic-mapping-tools.org) is an open source collection of UNIX tools allowing to manipulate data sets (including filtering, trend fitting, gridding, projecting, etc.) and produce data illustrations. GMT supports common map projections and comes with support data such as coastlines, rivers, and political boundaries.

```
access: module load gmt/6.4.0
Commands: GMT, plus all executables under /sw/<os-name>/gmt-<version>-gccsys/bin.
```

GrADS - Grid Analysis and Display System

The Grid Analysis and Display System (GrADS - http://opengrads.org) is an interactive desktop tool used for easy access, manipulation, and visualization of earth science data. The data format can be binary, GRIB, NetCDF or HDF-SDS.

```
access: module load grads/2.2.1
command: bufrscan, grads, grib2scan, gribmap, gribscan, gxeps, gxps, gxtran, stnmap, wgrib
```

IDL - Interactive Data Language

The Interactive Data Language (IDL - https://www.nv5geospatialsoftware.com/Products/IDL) is a commercial software for data analysis, visualization, and cross-platform application development.

```
access: module load idl/9.1
command: idl, idlde
```

Mathematica

Mathematica (http://www.wolfram.com/mathematica) is a commercial computational software program used in scientific, engineering, and mathematical fields and other areas of technical computing.

```
access: module load mathematica/14.1
command: mathematica
```

MATLAB - Matrix Laboratory

MATLAB (http://www.mathworks.com/products/matlab/) is a commercial software that integrates mathematical methods, visualization and a flexible environment for technical computing and to explore data, create algorithms, and create custom tools.

```
access: module load matlab/2025a
command: matlab
```

NCL - NCAR Command Language

The NCAR Command Language (NCL - http://www.ncl.ucar.edu), a product developed at the National Center for Atmospheric Research (NCAR), is an interpreted language designed specifically for scientific data processing and visualization.

```
access: module load ncl/6.3.0-nodap-precompiled
command: ncl, ncargcc, ncargf77, ncargf90 and many others under /sw/<os-name>/ncl-<version↩
    >-precompiled/bin
```

Ncview

Ncview (https://cirrus.ucsd.edu/ncview) displays the content of netCDF files using the X Window System graphical user interface. All variables in the file can be examined in 1D or 2D and animated in time.

```
access: module load ncview/2.1.9
command: ncview
```

NCO - NetCDF Operators

The netCDF Operators (NCO - `http://nco.sourceforge.net/`) are a suite of programs that facilitate manipulation (e.g., averaging, hyper-slabbing, attribute editing) and analysis of data stored in the netCDF or HDF formats.

```
access: module load nco/5.1.8
command: ncap2, ncatted, ncbo, ncea, ncecat, ncflint, ncks, ncpdq, ncra, ncrcat, ncrename, ↩
    ncwa
```

Paraview

ParaView (`http://www.paraview.org`) is an open-source data analysis and visualization application that can analyze data qualitative and quantitatively interactively in 3D or using batch processing capabilities.

```
access: module load paraview/5.13.3
command: paraview
```

Python

Python (`http://www.python.org`) is an object-oriented, interpreted, and interactive programming language. It serves also as basis for other software packages. The Anaconda python distribution is currently supported:

```
access: module load miniforge3/24.7.1
command: python, ipython, spyder, pip, mamba
```

Many Python modules are installed in the software tree (see below) and more can be installed upon request.

```
basemap
cython
ffnet
foolscap
genshi
httplib2
igraph
ipython
matplotlib
mpi4py
```

```
netcdf4
networkx
nose
numexpr
numpy
pandas
parallel_python
paramiko
paste
pastedeploy
pastescript
pexpect
pil
pip
progressbar
pupynere
pycairo
pycrypto
pydap
pygobject
pygtk
pyhdf
pyngl
pynio
pyopenssl
pyqt
pysparse
python_dateutil
rpy2
scikits.image
scikits.statsmodels
scipy
seawater
setuptools
sip
sympy
tables
twisted
virtualenv
wxpython
xlrd
zope_interface
...
```

Octave

GNU Octave (`http://www.gnu.org/software/octave`) is a high-level interpreted language, similar (and compatible) to MATLAB, intended for numerical computations. It provides capabilities for the numerical solution of linear and nonlinear problems, and for performing other numerical experiments. It also provides extensive graphics capabilities for data visualization and manipulation.

```
access: module load octave/8.3.0
command: octave
```

R

The R Project for Statistical Computing (`http://www.r-project.org`) is a free software environment for statistical computing and graphics.

```
access: module load r/4.3.1
command: R
```

Almost all R packages are installed in the software tree (see below for examples) and more can be installed upon request.

```
extremes
fields
foreign
gdata
gstat
gtools
ismev
lattice
lmoments
mapdata
mapproj
maps
maptools
narray
ncdf
ncdf4
proj4
rcolorbrewer
rgdal
rmatlab
rnetcdf
ruby
ruby-netcdf
rworldmap
sp
spacetime
spam
teachingdemos
xts
zoo
...
```

Scilab

Scilab (`http://www.scilab.org`) is free open source software for numerical computation providing a powerful computing environment for engineering and scientific applications.

```
access: module load scilab/2023.1.0
command: scilab
```

5.2. Accessing the software from outside the Campus

Within the private network, software licenses are hosted on the server "license.cen.uni-hamburg.de" and tools like "matlab", "idl" or several compilers need them. The user can use the licenses from home if the corresponding environment variable is set correctly:

```
LM_LICENSE_FILE=1704@matlab.rrz.uni-hamburg.de:1700@lizenzsrv8.rrz.uni-hamburg.de:1955↩
    @license.cen.uni-hamburg.de
```

The license server "license.cen.uni-hamburg.de" is not available outside the private network and external access is blocked from a firewall. To be able to use software requesting a license on that server, the user has to first establish a tunnel through a secure shell ("ssh") connection and only then issue the software command.

```
for matlab access:
ssh -L 1704:matlab.rrz.uni-hamburg.de:1702 <username>@login.cen.uni-hamburg.de

for idl or mathematica access:
ssh -L 1700:lizenzsrv8.rrz.uni-hamburg.de:1701 <username>@login.cen.uni-hamburg.de

for intel compiler access:
ssh -L 1955:license.cen.uni-hamburg.de:1955 -L 1962:license.cen.uni-hamburg.de:1962 <↩
    username>@login.cen.uni-hamburg.de
```

A. Example: the ”shallow water” model

This section describes the numerical model used throughout this document to exemplify the procedures of compiling and linking, running the application and post-processing the model output.

A.1. Model equations

The shallow water equations model the propagation of disturbances in incompressible fluids (like water) resulting from gravitational or rotational accelerations. In this model's framework it is assumed that the depth of the fluid is small compared to the wave length of the disturbance. The shallow water approximation therefore provides a reasonable model for the propagation of tsunamis in the ocean, since tsunamis are very long waves (with hundreds of km wavelength) that propagate over depths of a few km.

The partial differential equations, enclosing the principles of conservation of mass and momentum, can be seen below. Time, t, and the two horizontal spatial dimensions, x, y are the independent variables and the perturbed fluid elevation, η, and the two-dimensional fluid velocity, u, v are the dependent variables. The force acting on the fluid is gravity, g, and earth rotation effects are contained in the terms including the planetary vorticity, f, also termed the Coriolis parameter. The total water depth at rest is H.

$$\frac{\partial u}{\partial t} - fv = -g\frac{\partial \eta}{\partial x} \tag{1}$$

$$\frac{\partial v}{\partial t} + fu = -g\frac{\partial \eta}{\partial y} \tag{2}$$

$$\frac{\partial \eta}{\partial t} + \frac{\partial (Hu)}{\partial x} + \frac{\partial (Hv)}{\partial y} = 0 \tag{3}$$

The expressions above are a simplification of the more complete Navier-Stokes equations under the assumptions: the fluid being of uniform density and in hydrostatic balance; the fluid flowing over a horizontal flat surface (so that the elevation of the surface is also the layer thickness); the slope of the surface elevation being small; the horizontal scale of the flow being large compared to the fluid depth; the flow velocity being independent of depth and the friction with the bottom surface being negligible.

A.2. Discretization of the equations

To keep the resulting source code simple the equations are discretized and integrated with an explicit FTCS ("forward in time centered in space") scheme, which is conditionally stable depending on the CFL condition $\frac{\Delta t}{2\Delta x}$, where Δt is the time step of integration and Δx is the horizontal spacing of the discretization. So, if all the partial derivatives are approximated by finite differences:

$$\frac{\partial u}{\partial t} = \frac{u(t+\Delta t) - u(t)}{\Delta t} \tag{4}$$

$$\frac{\partial \eta}{\partial x} = \frac{\eta(x+\Delta x) - \eta(x-\Delta x)}{2\Delta x} \tag{5}$$

the shallow water equations (1-3) can be rewritten (with the spatial discretization $x = i\Delta x$ and $y = j\Delta y$):

$$u(i,j) = ub(i,j) + f\Delta t vb(i,j) - g\frac{\Delta t}{2\Delta x}[\eta b(i+1,j) - \eta b(i-1,j)] \tag{6}$$

$$v(i,j) = vb(i,j) + f\Delta t u(i,j) - g\frac{\Delta t}{2\Delta y}[\eta b(i,j+1) - \eta b(i,j+1)] \tag{7}$$

$$\eta(i,j) = \eta b(i,j) - H(i,j)\frac{\Delta t}{2\Delta y}[u(i+1,j) - u(i-1,j)] - \\ - H(i,j)\frac{\Delta t}{2\Delta y}[v(i,j+1) - v(i,j-1)] \tag{8}$$

Note that in the above discretized equations the variables at time step t are called ub, vb and ηb and at time step $t + \Delta t$ are called u, v and η.

This set of discretized equations, subject to initial and boundary conditions, are the base for the source code used in section 3. All source code files can be found under `/data/share/CIS/cen_sw_guide/suppl`. The simplest version of the program, corresponding to an initial Gaussian perturbation propagating over a flat bottom in the f-plane subject to bi-periodic open boundary conditions, is presented next.

A.3. Fortran source code

```
      program main

! Solves the shallow water equations
!
! nuno.serra@uni-hamburg.de
```

```
      implicit none

! variable definitions
      integer, parameter :: nx=100,ny=100
      real, dimension(nx,ny) :: u,ub,v,vb,eta,etab,f,h
      integer :: i,j,n
      real :: dx,dy,dt,nt,dump,g,pi
      real :: lx,ly,kx,ky,rx,ry

! constants
      nt=86400.
      dump=60.
      dx=1000.
      dy=1000.
      dt=1.
      g=9.8
      pi=acos(-1.)

! parameters
      lx=nx*dx
      ly=ny*dy
      kx=(3/lx)*2*pi
      ky=(3/ly)*2*pi
      rx=dt/(2*dx)
      ry=dt/(2*dy)

! 2D initial conditions
      do j=1,ny
      do i=1,nx

! flat bottom topography
      h(i,j)=1000.

! constant coriolis parameter
      f(i,j)=0.8e-4

! initial gaussian perturbation
      u(i,j)=0.
      ub(i,j)=u(i,j)
      v(i,j)=0.
      vb(i,j)=v(i,j)
      eta(i,j)=0.1*exp(-((kx*(i-50)*dx)**2+(ky*(j-50)*dy)**2))
      etab(i,j)=eta(i,j)

      enddo
      enddo

! start main loop
      do n=1,int(nt/dt)

      do j=2,ny-1
      do i=2,nx-1
! x-momentum equation
      u(i,j)=ub(i,j)+f(i,j)*dt*vb(i,j)
     & -g*rx*(etab(i+1,j)-etab(i-1,j))
      enddo
      enddo

      do j=2,ny-1
      do i=2,nx-1
! y-momentum equation
      v(i,j)=vb(i,j)-f(i,j)*dt*u(i,j)
     & -g*ry*(etab(i,j+1)-etab(i,j-1))
      enddo
      enddo
```

```
      do j=2,ny-1
      do i=2,nx-1
! continuity equation
      eta(i,j)=etab(i,j)
     & -h(i,j)*rx*(u(i+1,j)-u(i-1,j))
     & -h(i,j)*ry*(v(i,j+1)-v(i,j-1))
      enddo
      enddo

! bi-periodic boundary conditions

! east and west
      do j=1,ny
      u(nx,j)=u(2,j)
      u(1,j)=u(nx-1,j)
      v(nx,j)=v(2,j)
      v(1,j)=v(nx-1,j)
      eta(nx,j)=eta(2,j)
      eta(1,j)=eta(nx-1,j)
      enddo
! north and south
      do i=1,nx
      u(i,ny)=u(i,2)
      u(i,1)=u(i,ny-1)
      v(i,ny)=v(i,2)
      v(i,1)=v(i,ny-1)
      eta(i,ny)=eta(i,2)
      eta(i,1)=eta(i,ny-1)
      enddo

! pass information to previous time step
      do j=1,ny
      do i=1,nx
      ub(i,j)=(u(i,j)+ub(i,j))/2
      vb(i,j)=(v(i,j)+vb(i,j))/2
      etab(i,j)=(eta(i,j)+etab(i,j))/2
      enddo
      enddo

! output in netcdf format
      if (mod((n-1)*dt,dump).eq.0) then
      call out_cdf(etab,ub,vb,h)
      endif

! end main loop
      enddo

      end

      subroutine out_cdf(etab,ub,vb,h)

      implicit none
      include 'netcdf.inc'

      integer, parameter :: nx=100,ny=100
      real, parameter :: dump=60.
      real, dimension(nx,ny) :: etab,ub,vb,h
      integer :: icdf,iret,cdfid,ifill
      integer :: xposdim,yposdim,timedim
      integer :: tid,eta_id,u_id,v_id,h_id
      integer, dimension(3) :: base_date
      integer, dimension(3) :: dims
      integer, dimension(3) :: corner,edges
      data icdf /0/
      save
```

```
      if (icdf.eq.0) then

! enter define mode
      cdfid = nccre('sw.cdf',ncclob,iret)
      ifill = ncsfil(cdfid,ncnofill,iret)

! define dimensions
      xposdim = ncddef(cdfid,'xpos',nx,iret)
      yposdim = ncddef(cdfid,'ypos',ny,iret)
      timedim = ncddef(cdfid,'time',ncunlim,iret)

! define variables and attributes

! 1d vars
      dims(1) = timedim

! time
      tid = ncvdef(cdfid,'time',ncfloat,1,dims,iret)
      call ncaptc(cdfid,tid,'long_name',ncchar,4,'time',iret)
      call ncaptc(cdfid,tid,'units',ncchar,33,
     &     'seconds since 2000-01-01 00:00:00',iret)

! 2d vars
      dims(2) = yposdim
      dims(1) = xposdim

! h
      h_id = ncvdef(cdfid,'h',ncfloat,2,dims,iret)
      call ncaptc(cdfid,h_id,'long_name',ncchar,1,'h',iret)
      call ncaptc(cdfid,h_id,'units',ncchar,1,'m',iret)

! 3d vars
      dims(3) = timedim
      dims(2) = yposdim
      dims(1) = xposdim

! u
      u_id = ncvdef(cdfid,'u',ncfloat,3,dims,iret)
      call ncaptc(cdfid,u_id,'long_name',ncchar,1,'u',iret)
      call ncaptc(cdfid,u_id,'units',ncchar,3,'m/s',iret)

! v
      v_id = ncvdef(cdfid,'v',ncfloat,3,dims,iret)
      call ncaptc(cdfid,v_id,'long_name',ncchar,1,'v',iret)
      call ncaptc(cdfid,v_id,'units',ncchar,3,'m/s',iret)

! eta
      eta_id = ncvdef(cdfid,'eta',ncfloat,3,dims,iret)
      call ncaptc(cdfid,eta_id,'long_name',ncchar,3,'eta',iret)
      call ncaptc(cdfid,eta_id,'units',ncchar,1,'m',iret)

! global attributes
      call ncaptc(cdfid,ncglobal,'experiment',ncchar,80,
     &  'sw model',iret)

      base_date(1) = 1900
      base_date(2) = 1
      base_date(3) = 1

      call ncapt(cdfid,ncglobal,'base_date',nclong,3,base_date,iret)
      call ncendf(cdfid,iret)

      else

      cdfid = ncopn('sw.cdf',ncwrite,iret)
      ifill = ncsfil(cdfid,ncnofill,iret)
```

```
      endif

!  end of define mode and start of store mode

      icdf = icdf + 1

! store 1d variables
      corner(1) = icdf

! time
      tid = ncvid(cdfid,'time',iret)
      call ncvpt1(cdfid,tid,corner,float(icdf-1)*dump,iret)

! store 2d variables
      corner(1) = 1
      corner(2) = 1
      edges(1) = nx
      edges(2) = ny

! h
      h_id = ncvid(cdfid,'h',iret)
      call ncvpt(cdfid,h_id,corner,edges,h,iret)

! store 3d variables
      corner(1) = 1
      corner(2) = 1
      corner(3) = icdf
      edges(1) = nx
      edges(2) = ny
      edges(3) = 1

! u
      u_id = ncvid(cdfid,'u',iret)
      call ncvpt(cdfid,u_id,corner,edges,ub,iret)

! v
      v_id = ncvid(cdfid,'v',iret)
      call ncvpt(cdfid,v_id,corner,edges,vb,iret)

! eta
      eta_id = ncvid(cdfid,'eta',iret)
      call ncvpt(cdfid,eta_id,corner,edges,etab,iret)

      call ncclos(cdfid,iret)

      end
```