Containerisation of LFRic with Docker

Iva Kavčič, Met Office, UK

1 Introduction

This report presents work on containerisation of LFRic with Docker during the ESiWACE2 project "Container Hackathon for Modellers" in December 2019. Section 2 gives an overview of LFRic and its main build dependency, PSyclone. Instructions on building and running LFRic in Docker CE can be found in section 3. Instructions on how to run the LFRic Gungho container with Sarus on Piz Daint and results of the runs are in section 4.

The associated scripts and inputs are hosted in the "LFRIC" subdirectory of the Container Hackathon GithHub repository. For more information on building the LFRic Docker container please contact Iva Kavčič, Met Office and Luca Marsella, CSCS (mentor).

2 LFRic and PSyclone: A quick overview

LFRic[1] is the new weather and climate modelling system being developed by the UK Met Office to replace the existing Unified Model (UM) in preparation for exascale computing in the 2020s. LFRic uses the GungHo[2] dynamical core and runs on a semi-structured cubed-sphere mesh.

The design of the supporting infrastructure follows object-oriented principles to facilitate modularity and the use of external libraries.[3,4] One of the guiding design principles, imposed to promote performance portability, is "separation of concerns" between the science code and parallel code. An application called PSyclone, developed at the STFC Hartree Centre, can generate the parallel code enabling deployment of a single source science code onto different machine architectures.

PSyclone is a domain-specific compiler and source-to-source translator developed for use in finite element, finite volume and finite difference codes. Using the information from a supported API, PSyclone generates code exploiting different parallel programming models.

2.1 LFRic repository and wiki

The LFRic repository and the associated wiki are hosted at the Met Office Science Repository Service (MOSRS). The code is BSD-licensed, however browsing the LFRic wiki and code repository requires login access to MOSRS. Please contact the LFRic team manager, Steve Mullerworth, to be granted access to the repository.

Once the access has been granted the LFRic trunk can be checked out using Subversion or FCM version control systems. SVN is recommended for checking out the code for runs only as it is easier to install.

2.1.1 LFRic code structure

The LFRic trunk (at revision 21509) is structured as follows:

- `bin` - Rose executables;
- `extra` - Utilities (e.g. job submission scripts);
- `GPL` - Rose source used in LFRic;
- `gungho` - Gungho dynamical core (one of the main science applications);
- `infrastructure` - LFRic infrastructure supporting science applications;
- `jules` - Interface to the MO JULES land surface model;
- `lfric_atm` - LFRic atmospheric model (Gungho dynamical core, UM Physics, JULES and SOCRATES);
- `mesh_tools` - Mesh generation tools;
- `miniapps` - "Standalone" science and infrastructure applications (e.g. Gravity Wave application);
- `socrates` - Interface to the MO radiative transfer ("Suite Of Community RAdiative Transfer codes") model;
- `um_physics` - Interface to the MO UM Physics parameterisation schemes.

2.2 PSyclone repository and wiki

Both PSyclone and the Fortran parser it uses are open source and hosted on GitHub. Their wikis are also hosted on GitHub:

- PSyclone wiki;
- fparser wiki.

The documentation is hosted on Read the Docs:

- PSyclone documentation;
- fparser documentation;

or PSyclone and fparser repositories for functionality merged to master but not yet part of an official release.

2.2.1 PSyclone in LFRic

LFRic wiki hosts pages on the use of PSyclone in LFRic, starting with the PSyclone in LFRic wiki. As mentioned in section 3, not every PSyclone release works with every LFRic trunk revision. The LFRic–PSyclone compatibility table is given in this LFRic wiki (requires login).

3 Building LFRic Docker container

3.1 Gungho benchmark

As outlined in section 2.1, the LFRic trunk is divided into several applications. This section outlines how to build a container benchmark of the Gungho application.

A template Dockerfile to build the LFRic Gungho container is available below.

```
##############################################################################
# LFRic environment: Builds and runs LFRic Gungho benchmark with PSyclone OMP.
# Prerequisites: LFRic build environment built with the system  GCC compiler
                 (version 7.4.0) using Docker template 'lfric_deps.docker' and
                 the relevant installation scripts.
##############################################################################
#
FROM lfric-deps:gnu
#
# Define home and working directories
ENV HOME /usr/local/src
WORKDIR /usr/local/src
#
# Set the compiler environment
ENV COMP_PACKAGE_DIR $HOME/gnu_env
# Set the following environment variables
ENV INSTALL_DIR $COMP_PACKAGE_DIR/usr
ENV BUILD_DIR $COMP_PACKAGE_DIR/build
ENV PFUNIT $INSTALL_DIR
ENV PATH $INSTALL_DIR/bin:$PATH
ENV FC gfortran
ENV FPP "cpp -traditional-cpp"
ENV LDMPI mpif90
ENV FFLAGS "-I$BUILD_DIR/XIOS/inc -I$INSTALL_DIR/include -I$INSTALL_DIR/mod"
ENV LDFLAGS "-L$BUILD_DIR/XIOS/lib -L$INSTALL_DIR/lib -L/usr/lib -lstdc++"
ENV CPPFLAGS "-I$INSTALL_DIR/include -I/usr/include"
ENV LD_LIBRARY_PATH $INSTALL_DIR/lib:$INSTALL_DIR/lib64:$LD_LIBRARY_PATH
# Path to PSyclone configuration file
ENV PSYCLONE_CONFIG /usr/local/share/psyclone/psyclone.cfg
#
# Adds config file for MPICH for Sarus on Piz Daint
RUN echo "/usr/local/src/gnu_env/usr/lib" > /etc/ld.so.conf.d/mpich.conf \
 && ldconfig
#
# For most applications one OMP thread is enough (can be set in batch submit script)
ENV OMP_NUM_THREADS 1
#
# Set option to apply OpenMP optimisations with PSyclone
ENV LFRIC_TARGET_PLATFORM meto-spice
#
# Copy LFRic trunk inside the container
COPY LFRic_trunk.tar .
# Unpack LFRic trunk
```

```
RUN tar -xf LFRic_trunk.tar \
# Navigate to "gungho" directory and build application
 && cd LFRic_trunk/gungho \
 && make build -j \
# Navigate to example directory to run the application from
 && cd example
#
# Paths to executables and example directory
ENV PATH $HOME/LFRic_trunk/gungho/bin:$PATH
WORKDIR $HOME/LFRic_trunk/gungho/example
```

We have saved the template Dockerfile above as `lfric_gungho.docker` and we built it with the command below:

```
docker build --network=host --add-host $HOSTNAME:127.0.0.1 -f \
  lfric_gungho.docker -t lfric-gungho:gnu .
```

The template starts the build from the `lfric-deps:gnu` Docker container which contains the libraries needed by the code: MPICH, YAXT, HDF5, NetCDF, NetCDF-Fortran, NetCDF-C++, XIOS and pFUnit. This dependency container was built from the script `lfric_deps.docker` by running

```
docker build --network=host --add-host $HOSTNAME:127.0.0.1 -f \
  lfric_deps.docker -t lfric-deps:gnu .
```

The scripts that build the libraries are also provided in the Hackathon LFRic Docker repository:

- `install_lfric_env.sh` sets up the environment and builds the dependencies of LFRic without the XIOS;
- `install_xios_env.sh` creates architecture files needed to build XIOS and builds XIOS.

3.2 Gravity Wave benchmark

This section outlines how to build a container benchmark of the Gravity Wave application, one of the LFRic miniapps (see section 2.1 for more details).

A template Dockerfile to build the LFRic Gravity Wave container is available below.

```
################################################################################
# LFRic environment: Builds and runs LFRic Gravity Wave benchmark.
# Prerequisites: LFRic build environment built with the system  GCC compiler
                 (version 7.4.0) using Docker template 'lfric_deps.docker' and
                 the relevant installation scripts.
################################################################################
#
FROM lfric-deps:gnu
#
```

```
# Define home and working directories
ENV HOME /usr/local/src
WORKDIR /usr/local/src
#
# Set the compiler environment
ENV COMP_PACKAGE_DIR $HOME/gnu_env
# Set the following environment variables
ENV INSTALL_DIR $COMP_PACKAGE_DIR/usr
ENV BUILD_DIR $COMP_PACKAGE_DIR/build
ENV PFUNIT $INSTALL_DIR
ENV PATH $INSTALL_DIR/bin:$PATH
ENV FC gfortran
ENV FPP "cpp -traditional-cpp"
ENV LDMPI mpif90
ENV FFLAGS "-I$BUILD_DIR/XIOS/inc -I$INSTALL_DIR/include -I$INSTALL_DIR/mod"
ENV LDFLAGS "-L$BUILD_DIR/XIOS/lib -L$INSTALL_DIR/lib -L/usr/lib -lstdc++"
ENV CPPFLAGS "-I$INSTALL_DIR/include -I/usr/include"
ENV LD_LIBRARY_PATH $INSTALL_DIR/lib:$INSTALL_DIR/lib64:$LD_LIBRARY_PATH
# Path to PSyclone configuration file
ENV PSYCLONE_CONFIG /usr/local/share/psyclone/psyclone.cfg
#
# Adds config file for MPICH for Sarus on Piz Daint
RUN echo "/usr/local/src/gnu_env/usr/lib" > /etc/ld.so.conf.d/mpich.conf \
 && ldconfig
#
# For most applications one OMP thread is enough (can be set in batch submit script)
ENV OMP_NUM_THREADS 1
#
# Copy LFRic trunk inside the container
COPY LFRic_trunk.tar .
# Unpack LFRic trunk
RUN tar -xf LFRic_trunk.tar \
# Navigate to "gravity_wave" directory and build application
 && cd LFRic_trunk/miniapps/gravity_wave \
 && make build -j \
# Navigate to example directory to run the application from
 && cd example
#
# Paths to executables and example directory
ENV PATH $HOME/LFRic_trunk/miniapps/gravity_wave/bin:$PATH
WORKDIR $HOME/LFRic_trunk/miniapps/gravity_wave/example
```

The above template was saved as `lfric_gwave.docker` and built with the command below:

```
docker build --network=host --add-host $HOSTNAME:127.0.0.1 -f \
  lfric_gwave.docker -t lfric-gwave:gnu .
```

As for the Gungho benchmark, the template starts the build from the `lfric-deps:gnu` Docker container.

3.3 Library versions and settings

- All libraries were dynamically linked to make sure that the LFRic container will use the optimised libraries of the host system: the `install_lfric_env.sh` script contains commented out instructions for a static build if required.

- The MPICH version used in the current Met Office (MO) LFRic build system is 3.3. Here we used version 3.1.4 to ensure ABI compatibility with the Cray MPI library available on Piz Daint: please have a look at the MPICH Wiki for more information on the ABI Compatibility Initiative.

- HDF5, NetCDF, NetCDF-Fortran, NetCDF-C++ and pFUnit are also slightly older than in the current MO LFRic build system, whereas YAXT is the same version. To install the same versions of the above packages that are currently used by LFRic, please use the alternative install script (dynamic linking example) `install_lfric_env_current.sh`.

- XIOS is a tricky beast to build as not every revision/release will work with every compiler and/or every compiler release. For instance, the MO GCC 6.1.0 environment uses revision 1537 which is too old for GCC 7.4.0 used here. Unfortunately, the current XIOS trunk produces a build which segfaults at runtime for the GCC release and other libraries used in this container. There is, however, a relatively recent revision before the segfault bug that was appropriate for this container.

- Here we used the same PSyclone release (1.7.0) that is used by the current LFRic trunk (as of 15 December 2019), built with Python 2 environment (the move to Python 3 and the newest PSyclone 1.8.1 is under way). Not every PSyclone release will work with every LFRic trunk revision. The LFRic–PSyclone compatibility table is give in this LFRic wiki (requires login).

3.4 Tips & tricks

3.4.1 Libraries

- The container tool Sarus supported on Piz Daint can add the proper hook to the host MPI library if the command `ldconfig` has been run to configure dynamic linker runtime bindings. Since we have installed the MPICH library in a non-default location, the command `ldconfig` would not be able to find the library in the custom path within the container. Therefore, before running `ldconfig` we added the path of MPICH to `/etc/ld.so.conf.d/mpich.conf` as in the example below (please look here for more information):

```
# Adds config file for MPICH for Sarus on Piz Daint
RUN echo "/usr/local/src/gnu_env/usr/lib" > /etc/ld.so.conf.d/mpich.conf \
 && ldconfig
```

- Some libraries (e.g. YAXT) perform a test run of a minimal MPI code during the configure step of the installation procedure: the test might fail within the Docker container if the local hostname cannot be resolved correctly. In order to do that, the local hostname should be available in the file `/etc/hosts`, which can be achieved

adding the option `--add-hostname $HOSTNAME:127.0.0.1` to Docker build command. If this solution does not work, one needs to edit the `/etc/hosts` file of the Docker container directly using `docker run` and commit the change with `docker commit`, since editing the `/etc/hosts` file is not possible within the Dockerfile. For more details, please check this link.

- PSyclone configuration file `psyclone.cfg` can end up in different locations during PSyclone installation. Please see PSyclone configuration documentation for the most common locations. If none of those work, try the good old `find / -name "psyclone.cfg"` search.

3.4.2 LFRic code

- Once the LFRic trunk was checked out, the `Makefile`s of the tested Gungho and Gravity Wave applications needed to be modified from

```
export EXTERNAL_DYNAMIC_LIBRARIES = yaxt yaxt_c netcdff netcdf hdf5 \
                                    $(CXX_RUNTIME_LIBRARY)
export EXTERNAL_STATIC_LIBRARIES = xios
```

  to

```
export EXTERNAL_DYNAMIC_LIBRARIES =
export EXTERNAL_STATIC_LIBRARIES = yaxt yaxt_c xios netcdff netcdf \
                                   hdf5_hl hdf5 z :libstdc++.a
```

  for the build to complete. For this reason we made changes to `Makefile`s outside the container and then copied the tarballs into the container.

- By default, running command-line `make build` builds LFRic with MPI but not OpenMP. To enable PSyclone OpenMP optimisations, the environment variable `LFRIC_TARGET_PLATFORM` was set to `meto-spice` value for the Gungho benchmark.

4 Running LFRic Docker container on Piz Daint

4.1 Gungho benchmark

The Docker image of the LFRic Gungho benchmark was loaded with `sarus` and run on Piz Daint. The Slurm batch script below was used as a template for running the benchmark.

```
#!/bin/bash -l
#SBATCH --job-name=job_name
#SBATCH --time=01:00:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=6
#SBATCH --cpus-per-task=1
#SBATCH --constraint=gpu
#SBATCH --output=lfric-gungho.%j.out
#SBATCH --error=lfric-gungho.%j.err
```

```
module load daint-gpu
module load sarus
module unload xalt
srun sarus run --mount=type=bind,source=$PWD/input/gungho,\
     destination=/usr/local/src/LFRic_trunk/gungho/example \
     --mpi load/library/lfric-gungho:gnu \
     bash -c "export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK && \
     gungho ./gungho_configuration.nml
```

The local folder `input/gungho` is a copy of the folder `LFRIC_trunk/gungho/example` and contains:

- Namelist `gungho_configuration.nml`;
- Four mesh files required for the multigrid preconditioner (`l_multigrid` flag set to `.true.` in the `&multigrid` namelist) used in Gungho (`mesh24.nc`, `mesh12.nc`, `mesh6.nc` and `mesh3.nc`);
- `iodef.xml` file required for parallel IO.

The IO file, however, was not used as the `use_xios_io` flag in the `&io` namelist was set to `.false.`. If no diagnostic output (e.g. plots) are required, it is recommended that the `write_diag` flag in the `&io` namelist is also set to `.false.`. All input files are available as the Gungho input archive in the Hackathon LFRic Docker repository. The mesh files and related namelists required for the Gungho higher resolution tests below (`C48`, `C96` and `C192` mesh configurations), were produced as described in Appendix A.1.

The Gungho application produces a single text output for a serial run (single MPI task) or `PET**.gungho.Log` outputs for multiple MPI tasks, as well as checksums stored in `gungho-checksum.txt` output. Depending on the `&io` namelist's settings, it also produces diagnostic outputs such as

- Overall application run times stored in `timer.txt` (set by the `subroutine_timers` flag to `.true.`);
- Overall application counter of halo calls stored in `halo_calls_counter.txt` (set by the `subroutine_counters` flag to `.true.`);
- Plots of the results (set by the `write_diag` and `diagnostic_frequency` options);
- XIOS client output and error logs if `use_xios_io` flag is set to `.true.` (`xios_client.out` and `xios_client.err` for a serial run/single MPI task; `xios_client_**.out` and `xios_client_**.err` for multiple MPI tasks).

4.1.1 Results

Tables 1-3 show times for completing the Gungho benchmark on Piz Daint Cray XC50 with different mesh resolutions, number of nodes, MPI tasks and OpenMP threads. Table 4 shows results of the additional tests for the `C24` mesh configuration, run with 1 MPI task and on 1, 2 and 4 OpenMP threads, respectively.

The times are from the LFRic `timer.txt` outputs and Slurm job outputs. Note that the Slurm completion times are longer, as they include overheads of job submission (prologue and epilogue).

OMP threads	C24, 1 node			
	LFRic timer		Slurm time	
	1 MPI p.n.	6 MPI p.n.	1 MPI p.n.	6 MPI p.n.
1	328.91 s	72.22 s	00:08:06 (486 s)	00:01:56 (116 s)
2	185.56 s	44.33 s	00:03:24 (204 s)	00:00:57 (57 s)

Table 1: Gungho benchmark runtimes on Piz Daint for `C24` mesh configuration, run on 1 compute node with 1 and 6 MPI tasks per node, respectively.

OMP threads	C48, 1 node			
	LFRic timer		Slurm time	
	1 MPI p.n.	6 MPI p.n.	1 MPI p.n.	6 MPI p.n.
1	1354.05 s	280.76 s	00:22:51 (1371 s)	00:05:00 (300 s)
2	776.13 s	177.2 s	00:13:18 (798 s)	00:03:26 (206 s)

Table 2: Gungho benchmark runtimes on Piz Daint for `C48` mesh configuration, run on 1 compute node with 1 and 6 MPI tasks per node, respectively.

OMP threads	C96, 6 nodes, 6 MPI p.n.		C192, 6 nodes, 6 MPI p.n.	
	LFRic timer	Slurm time	LFRic timer	Slurm time
1	194.59 s	00:03:33 (213 s)	801.52 s	00:13:40 (820 s)
2	128.43 s	00:02:57 (177 s)	517.95 s	00:08:57 (537 s)

Table 3: Gungho benchmark runtimes on Piz Daint for `C96` and `C192` mesh configurations, respectively, run on 6 compute nodes with 6 MPI tasks per node.

OMP threads	C24, 1 node, 1 MPI p.n.	
	LFRic timer	Slurm time
1	394.76 s	00:06:58 (418 s)
2	231.69 s	00:04:32 (272 s)
4	153.93 s	00:03:14 (194 s)

Table 4: Gungho benchmark runtimes on Piz Daint for `C24` mesh configuration, run on 1 compute node with 1 MPI task per node and 1, 2 and 4 OpenMP threads, respectively.

4.2 Gravity Wave benchmark on Piz Daint

The Docker image of the LFRic Gravity Wave benchmark was loaded with `sarus` and run on Piz Daint. The Slurm batch script below was used as a template for running the benchmark.

```
#!/bin/bash -l
#SBATCH --job-name=lfric-gwave
#SBATCH --time=00:30:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --partition=normal
#SBATCH --constraint=gpu
```

```
module load daint-gpu
module load sarus
module unload xalt
srun sarus run \
     --mount=type=bind,source=$PWD/input/gwave,\
     destination=/usr/local/src/LFRic_trunk/gwave/example \
     --mpi load/library/lfric-gwave:gnu \
     bash -c "export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK && \
     gravity_wave ./gravity_wave_configuration.nml"
```

The local folder `input/gwave` contains the namelist `gravity_wave_configuration.nml` and the mesh file `mesh24.nc`: both files are available in the Gravity Wave input archive of the Hackathon LFRic Docker repository. This folder is a trimmed-down copy of the folder `LFRIC_trunk/miniapps/gravity_wave/example`. There are also other input files present in the LFRic repository: `mesh12.nc`, `mesh6.nc` and `mesh3.nc` (for the multigrid preconditioner) and `iodef.xml` (for parallel IO). They are not present in the `input/gwave` archive as neither `l_multigrid` nor `use_xios_io` flags were set to `.true.` for the Piz Daint run.

The Gravity Wave application produces outputs similar to the Gungho application depending on the namelists' settings. On a single Piz Daint Cray XC50 node the benchmark took around 5 minutes to complete on a single MPI task (truncated output below).

```
Batch Job Summary Report for Job "lfric-gwave" (18507334) on daint
--------------------------------------------------------------
Start                End                  Elapsed     Timelimit
-------------------  -------------------  ----------  ---------
2019-12-04T12:06:38  2019-12-04T12:11:50  00:05:12    01:00:00
--------------------------------------------------------------
Username    Account     Partition   NNodes   Energy
----------  ----------  ----------  ------  --------------
hck01       hck         normal           1   29.18K joules
```

5 Summary

The LFRic Docker containers for Gungho and Gravity Wave benchmark were built inside the Ubuntu 18.04 virtual environment on a laptop and then deployed and run on Piz Daint using the local libraries. Building containers was somewhat challenging due to the LFRic build dependencies and quirks of the Ubuntu virtual machine. A Docker container for only the libraries and build dependencies needed by the code seems as a very useful tool, for both building benchmark containers and providing LFRic software stack to collaborators who may not have access to the requirements or expertise to build the environment.

Running the Docker containers with Sarus on Piz Daint showed to be relatively straightforward in comparison with the process of building them. The Gungho benchmark was successfully run with different mesh resolutions, number of nodes, MPI tasks and OpenMP threads. The results show good scaling, with the actual runtimes shorter than the Slurm completion times as expected.

The Met Office has been exploring various approaches for containerisation of its models and the experiences from this hackathon will be very useful in coming up with the optimal strategies for different applications.

References

[1] Adams SV, Ford RW, Hambley M, Hobson JM, Kavcic I, Maynard CM, Melvin T, Mueller EH, Mullerworth S, Porter AR, Rezny M, Shipway BJ and Wong R (2019): *LFRic: Meeting the challenges of scalability and performance portability in Weather and Climate models.* Journal of Parallel and Distributed Computing.

[2] *GungHo - a next generation atmospheric dynamical core for weather and climate modelling.* Accessed: 15 December 2019.

[3] *LFRic - a modelling system fit for future computers.* Accessed: 15 December 2019.

[4] *Next generation atmospheric model development.* Accessed: 15 December 2019.

A LFRic mesh generation

This section describes how to generate higher resolution cubed-sphere meshes used in the LFRic Gungho runs on Piz Daint (see section 4.1 for more details on the runs). To generate the required meshes, we used the `lfric-gungho:gnu` Docker container (see section 3 for the container build instructions).

A.1 LFRic mesh tools

LFRic application `mesh_tools` (see section 2.1 for more details on LFRic code structure) contains all prerequisites for the generation of cubed-sphere and biperiodic Cartesian meshes (we only used the former in the Piz Daint tests).

Running `make build` or `make build -j` in the `mesh_tools` directory produces two executables for mesh generation in the `bin/` directory, `cubedsphere_mesh_generator` and `planar_mesh_generator`, and the executable `summarise_ugrid` that summarises mesh files information (LFRic stores meshes in UGRID format).

Cubed-sphere or planar meshes can then be produced by running

```
../bin cubedsphere_mesh_generator cubedsphere.nml
```

or

```
../bin planar_mesh_generator planar_mesh.nml
```

from the `mesh_tools/example` directory. The `C48` mesh can be generated by copying the `cubedsphere.nml` to `cubedsphere_C48.nml` namelist, modifying the new namelist as shown below

```
&cubedsphere_mesh_generator
  mesh_filename = 'mesh_C48.nc'
  nmeshes       = 1
  mesh_names    = 'dynamics'
  edge_cells    = 48
  smooth_passes = 0
/
```

and then running

```
../bin cubedsphere_mesh_generator cubedsphere_C48.nml
```

from the `mesh_tools/example` directory. The `C96` and `C192` meshes are also produced in a similar way.

A.2 Using LFRic container for mesh generation

To build the mesh generation executables, we navigated to the `LFRic_trunk/mesh_tools` directory in the `lfric-gungho:gnu` container and modified the `Makefile` from

```
export EXTERNAL_DYNAMIC_LIBRARIES = netcdff netcdf hdf5 yaxt yaxt_c
```

to

```
export EXTERNAL_DYNAMIC_LIBRARIES =
export EXTERNAL_STATIC_LIBRARIES = netcdff netcdf hdf5 yaxt yaxt_c
```

before running `make build -j` (similar `Makefile` modifications were also required for the Gungho and Gravity Wave benchmarks; see section 3.4.2 for details). The required mesh files, `C48`, `C96` and `C192`, were then generated as described above.

A.3 Higher resolution Gungho benchmark on Piz Daint

The generated mesh files were copied from the LFRic Gungho container to the host Ubuntu 18.04 virtual environment platform in order to be transferred to Piz Daint for the runs described in section 4.

Files can be copied from a Docker container to the host platform by using command

```
docker cp <containerId>:/file/path/within/container /host/path/target
```

where `docker ps` gives the container ID.

Running Gungho with multigrid preconditioner requires four-component mesh chain: base mesh and three lower-resolution meshes, each one twice as coarse as the mesh before. Hence, running `C48` jobs required mesh files for the `C48` base mesh and `C24`, `C12` and `C6` coarser resolution meshes. It also required modifying the `gungho_namelist.nml` input file as shown below.

- `filename` option in the `&base_mesh` namelist was modified from

  ```
  filename        = 'mesh_C24.nc'
  ```

  to

  ```
  filename        = 'mesh_C48.nc'
  ```

- `ugrid` option in the `&multigrid` namelist was modified from

  ```
  ugrid      = 'prime', 'mesh_C12.nc', 'mesh_C6.nc', 'mesh_C3.nc'
  ```

to

```
ugrid      = 'prime', 'mesh_C24.nc', 'mesh_C12.nc', 'mesh_C6.nc'
```

For easier work the modified namelist input file was saved as `gungho_configuration_C48.nml`. The higher resolution (`C96` and `C192`) mesh chains and namelist input files were generated in a similar way and then transferred to Piz Daint.

A.4 Higher resolution Gungho benchmark inside the container

The generated mesh files can be copied into the `LFRic_trunk/gungho/example` directory inside the LFRic Gungho container. After modifying the namelist input files as shown above, simply run e.g. `C48` configuration as

```
../bin/gungho gungho_configuration_C48.nml
```

in serial or

```
mpirun -np $MPI_TASKS ../bin/gungho gungho_configuration_C48.nml
```

for parallel runs (it is recommended to have multiples of 6 `MPI_TASKS` for the cubed-sphere runs). The container changes need to be committed so that the generated meshes and namelist files are saved for later, either in the same or in a new container.