Report on the containerization of ICON

Remo Dietlicher, Sergey Kosukhin, Rafael Sarmiento,
William Sawyer, Andre Walser

December 3-5 2019

1 Application

ICON is a modeling framework unifying global numerical weather prediction and climate projection. While it contains modules for the entire earth system, this project focuses on the land and atmosphere components. The climate mode of ICON is almost entirely GPU-capable using OpenACC statements.

2 Containerization approach

2.1 Starting point

Before the hackathon, we had two setups: one that builds ICON for CPU, and one for GPU using OpenACC. At this point, the PGI compiler is the only viable option for OpenACC support. To keep the container simple both the CPU and GPU versions have been compiled with this compiler. The latest release of the PGI community edition has been used. At the time of the hackathon, this was PGI 19.10.

2.2 Verification Framework

The ICON testing infrastructure has a tool that allows comparing model output beyond bit-wise identity. This is necessary when validating a GPU-capable binary against a CPU references. For a set of key model variables (including temperature, pressure, humidity, ...) the growth of small perturbations is computed using an ensemble of model runs. The deviation of these perturbed runs from the unperturbed reference are representative of errors that may occur due to different implementations of transcendental functions functions in GPU code and serve as a tolerance range when comparing output from GPU and CPU binaries.

Using this method, the containerized model both in CPU and GPU mode has been validated against a CPU reference on the host system. Performance is measured using internal ICON timers that can easily be compared across the different versions: containerized and host build for CPU and GPU.

2.3 Containerizing ICON

The containerization of the model has been split into two main steps. First, we built a container with the numerous dependencies of ICON using the PGI 19.10 compiler, in particular with the CUDA support for GPU execution. In the second step we built ICON inside the container by running it in interactive mode. Using these steps we were able to verify that the containerized built on Daint as well as our laptop for both the CPU and GPU versions produced correct results.

2.3.1 Building the image

The Dockerfile for the build container is:

```
FROM nvidia/cuda:10.1-devel-ubuntu16.04

RUN useradd -ms /bin/bash cscs

RUN apt-get update && \
    apt-get install \
    wget \
    ssh \
    vim \
    ksh \
    less \
    environment-modules \
    git \
```

```
    rsync \
    libcurl4-openssl-dev \
    libxml2-dev \
    python2.7 \
    python-pip --yes --no-install-recommends && \
    apt-get clean

RUN pip install --upgrade pip setuptools

WORKDIR /home/cscs

USER cscs

RUN pip install --user easybuild

RUN git clone https://github.com/lucamar/production.git && \
    cd production && \
    git checkout tsa76

ENV PATH=$PATH:/home/cscs/.local/bin \
    EASYBUILD_MODULES_TOOL=EnvironmentModulesC \
    #EASYBUILD_MODULES_TOOL=EnvironmentModules \
    EASYBUILD_MODULE_SYNTAX=Tcl \
    EASYBUILD_ROBOT_PATHS=/home/cscs/production/easybuild/easyconfigs:/home/cscs/.local/easybuild/easyconfigs

#RUN eb binutils-2.30-GCCcore-7.3.0.eb -r

RUN mkdir -p /home/cscs/.local/easybuild/sources/PGI
COPY pgilinux-2019-1910-x86-64.tar.gz /home/cscs/.local/easybuild/sources/PGI/
COPY PGI-19.10-GCC-7.3.0-2.30.eb /home/cscs/production/easybuild/easyconfigs/p/PGI/
RUN eb PGI-19.10-GCC-7.3.0-2.30.eb -r

COPY Szip-2.1.1-PGI-19.10-GCC-7.3.0-2.30.eb /home/cscs/production/easybuild/easyconfigs/s/Szip/
COPY OpenMPI-4.0.1-PGI-19.10-GCC-7.3.0-2.30.eb /home/cscs/production/easybuild/easyconfigs/o/OpenMPI/

COPY HDF5-1.10.5-PGI-19.10-GCC-7.3.0-2.30.eb /home/cscs/production/easybuild/easyconfigs/H/HDF5/
RUN eb HDF5-1.10.5-PGI-19.10-GCC-7.3.0-2.30.eb -r

COPY netCDF-4.6.1-PGI-19.10-GCC-7.3.0-2.30.eb /home/cscs/production/easybuild/easyconfigs/n/netCDF/

RUN eb CMake-3.14.1.eb -r

COPY netCDF-Fortran-4.4.4-PGI-19.10-GCC-7.3.0-2.30.eb /home/cscs/production/easybuild/easyconfigs/n/netCDF-Fortran/
RUN eb netCDF-Fortran-4.4.4-PGI-19.10-GCC-7.3.0-2.30.eb -r

COPY MPICH-3.1.4-PGI-19.10-GCC-7.3.0-2.30.eb /home/cscs/production/easybuild/easyconfigs/m/MPICH/

RUN eb MPICH-3.1.4-PGI-19.10-GCC-7.3.0-2.30.eb -r

RUN ln -s /usr/lib/x86_64-linux-gnu/libxml2.a ~/.local/lib/ && \
    mkdir ~/.local/include && \
    ln -s /usr/include/libxml2 ~/.local/include/

COPY JasPer-2.0.14-PGI-19.10-GCC-7.3.0-2.30.eb /home/cscs/production/easybuild/easyconfigs/j/JasPer
RUN eb JasPer-2.0.14-PGI-19.10-GCC-7.3.0-2.30.eb -r

COPY ecCodes-2.13.0-PGI-19.10-GCC-7.3.0-2.30.eb /home/cscs/production/easybuild/easyconfigs/c/ecCodes/
RUN eb ecCodes-2.13.0-PGI-19.10-GCC-7.3.0-2.30.eb

ENV ICON_DOCKER=1

RUN echo  '. /usr/share/modules/init/bash' >> /home/cscs/.bashrc && \
    echo  'module use /home/cscs/.local/easybuild/modules/all' >> /home/cscs/.bashrc

USER root

RUN apt-get update && \
    apt-get install \
    rsync \
    bison \
    flex \
    default-jdk \
    gfortran \
    ant --yes --no-install-recommends && \
    apt-get clean
```

```
USER cscs

RUN git clone https://github.com/claw-project/claw-compiler.git && \
    cd claw-compiler && \
    git submodule init && \
    git submodule update --remote && \
    mkdir build && \
    cd build && \
    . /usr/share/modules/init/sh && \
    module use /home/cscs/.local/easybuild/modules/all && \
    module load PGI MPICH CMake && \
    FC=mpif90 CC=mpicc CXX=mpicxx cmake -DCMAKE_INSTALL_PREFIX=/home/cscs/.local/claw \
                                        -DOMNI_MPI_FC="MPI_FC=mpif90" \
                                        -DOMNI_MPI_CC="MPI_CC=mpicc" .. && \
    make && \
    make install
```

The image is build as follows:

- Download PGI Community edition Version 19.10 from `https://www.pgroup.com/products/community.htm` and put it to the `./image` subdirectory. This particular version of PGI might not be available anymore. If this is the case, you either have to find it somewhere else or adjust the files in the `./image` subdirectory (i.e. the Dockerfile and easyblocks) to the version of PGI compiler that you can get. During the hackathon we did not have a recipe for claw, we add the installing commands directly on the Dockerfile.
- Switch to the `./image` subdirectory: `cd ./image`. This folder contains all the build recipes for the dependencies of ICON. They are on the ContainerHackathon repository.
- Build the image: `docker build -t icon-base:pgi-mpich -f Dockerfile .`

Once the building is finished, the command `docker images` should show that the image has been added to the local Docker registry.

2.3.2 Running the image

Once the image is built, the container can be run. At this point we have an image that contains everything necessary to build ICON. Note that we have defined the environment variable `ICON_DOCKER=1`, which is used to tell the ICON's build system that the compilation will be performed within a container.

The next step is then to built ICON within the container. We did this on the same laptop where we did the building to avoid multiple file transfers to Piz Daint in case it was needed to modify the image. For that we run the container with two directories mounted. One contains the ICON source and the other contains data (e.g. ICON grid files) for testing the resulting ICON binaries:

```
docker run -it -v /absoulute/path/to/icon/source/directory:/icon \
               -v /absolute/path/to/poolFolder_prefix:/icon-data-pool \
               --rm icon-base:pgi-mpich-dev
```

At this point we can run the configure wrappers of ICON, which can be found inside `/icon/config/examples/docker`.

2.4 Running ICON within the container

We transferred both the image and the ICON binaries to Piz Daint in order to run ICON within the container using Sarus. We ran test with using both ICON with CPU-only and GPU support on a single node.

2.5 Challenges

The major challenge was running the container on multiple MPI processes. The main issue here was that ICON is usually run by a scripting mechanism that prepares the environment, input data, namelists and finally runs the model. This tool chain works well for a single process where we launch the model from within the container, but does not work in an MPI environment where we need to launch multiple instances of the container. We were not able to resolve this problem by the end of the hackathon, but have taken it home for the continuation of the ICON containerization effort.

3 Conclusions

The expectation of this hackathon was to get a container running on CPU. Exceeding this goal by running on GPU has been very satisfying, especially considering that we started out with an entirely new build system.

The process of containerizing revealed the shortcomings of the scripting systems that is used to launch ICON experiments. A result of this was that the container prepared during the hackathon was still Daint-specific and could not be easily ported to other machines. Abstracting the scripting system is planned for the continuation of this effort.

Since none of us came with any containerization experience, the help we received from our mentor, Rafael Sarmiento, was highly useful to 'kick-start' our efforts with ICON. We would like also to thank the organizers for offering this hackathon as a service to ESiWACE-2 participants.