Introduction

NEMO for Nucleus for European Modelling of the Ocean is a state-of-the-art modelling framework for research activities and forecasting services in ocean and climate sciences, that has been developed in a sustainable way by a European consortium since 2008. The NEMO ocean model has 3 major components:

- NEMO-OPA models the ocean {thermo}dynamics and solves the primitive equations (./src/OCE)
- NEMO-SI3 simulates sea ice {thermo}dynamics, brine inclusions and subgrid-scale thickness variations (./src/ICE)
- NEMO-TOP/PISCES models the {on,off}line oceanic tracers transport and biogeochemical processes (./src/TOP)

Not only does the NEMO framework model the ocean circulation, it offers various features to enable

- The seamless creation of embedded zooms thanks to 2-way nesting package AGRIF
- The opportunity to integrate an external biogeochemistry model
- Versatile data assimilation
- The generation of diagnostics through effective XIOS_ system
- The roll-out Earth system modeling with coupling interface based on OASIS

Several built-in configurations are provided to evaluate the skills and performances of the model which can be used as templates for setting up new configurations (./cfgs). The user can also check out available idealized test cases that address specific physical processes(./tests).

The NEMO source code is written in Fortran 95 and some of its prerequisite tools and libraries are already included in the ./ext subdirectory. It contains the AGRIF preprocessing program conv; the FCM build system and the IOIPSL library for parts of the output.

System dependencies

The following package are required:

- Perl interpreter
- Fortran compiler (ifort, gfortran, pgfortran, ...),
- Message Passing Interface (MPI) implementation (e.g. OpenMPI or MPICH).
- Network Common Data Form (NetCDF) library with its underlying Hierarchical Data Form (HDF)
- XIOS subsystem for handling I/O

NEMO, by default, takes advantage of some MPI features introduced into the MPI-3 standard. The XIOS library requires the parallel IO support from NetCDF/HDF5. It is also requested to compile those libraries with the same version of the MPI implementation that both NEMO and XIOS are compiled and linked with.

Compiling NEMO executable

The main script to compile and create executable is called `makenemo` and located in the CONFIG directory; it is used to identify the routines you need from the source code, to build the makefile and run it. As an example, compile GYRE with my_arch to create a MY_GYRE configuration:

```
./makenemo -m 'my_arch' -r GYRE -n 'MY_GYRE'
```

In our Docker implementation, the 'arch_file' must necessarily be 'gfortran_docker' in order to respect the current installation.

```
./makenemo -m gfortran_docker -r GYRE -n 'MY_GYRE'
```

Docker image

The NEMO package includes several submodels (i.e. ICE, PISCES, etc.) which can be activated through the use of pre-compiler keys in order to get specific configuration. In order to support the most commonly used configuration we opted to create an image with the GYRE_PISCES configuration and an image, which is layered on top of the GYRE_PISCES, referring to the ORCA_ICE_PISCES configuration. The first step to create an image of an application / service with docker is to choose the proper starting image for the application / service to be installed; if the starting image is not present in the local disk, it will be downloaded from Docker Hub. In our case, the choice was the UBUNTU bionic operating system 18.04, the latest version with long-term support (LTS) from Canonical. Clearly, the starting image is a basic image of the operating system, in which there are only a few basic commands and libraries necessary for its operation. In order to create an image, it was necessary to create a Dockerfile containing the starting image. In our case:

Dockerfile

Some useful commands and compilers software stack is than installed, such as wget, gcc, gfortran and g++. Moreover, the additional instructions of the Dockerfile include the installation of the MPI libraries (mpich) and the NEMO dependencies such as the HDF5 libraries, the NetCDF and PNetCDF libraries and the XIOS libraries useful for the parallel management of I/O. Some configuration files have been inserted into the utility folder, necessary for the installation of the XIOS libraries and the NEMO model. At the appropriate time, these files will be copied and inserted into the docker image using the COPY command:

```
COPY ./utility/arch-X64_Docker.* /home/docker-nemo/library/XIOS/branchs/xios-2.5/arch/
```

These files are the XIOS and NEMO arch files preset with the current docker installation: compilers, dependencies, paths, etc. An additional script to use when the model is launched is `run_job.sh`, that clearly has default settings, but it can be customized by the user. We included the official NEMO release 4.0.1 available through the NEMO svn repository:

```
RUN svn co http://forge.ipsl.jussieu.fr/nemo/svn/NEMO/releases/release-4.0.1
```

The last command consists in the compilation of the NEMO code:

```
RUN ./makenemo -m gfortran\_docker -r GYRE\_PISCES -n GYRE\_PISCES\_CSCS
```

The ORCA_ICE_PISCES image has been created on top of the GYRE_PISCES image:

```
FROM asccmcc/nemo:gyre
```

And the ORCA_ICE_PISCES configuration is built:

```
RUN ./makenemo -m gfortran\_docker -r ORCA2\_ICE\_PISCES -n ORCA\_ICE\_PIS
```

Both images are available on Dockerhub: asccmcc/nemo:gyre and asccmcc/nemo:orca

Test and validation

The docker images have been tested on a mac-os node and on the parallel architecture available at CSCS named PizDaint: Piz Daint (6th place in November 2019 Top500 supercomputer list) is composed of more than 5000 Cray XC50 compute nodes, each equipped with a 12-core Intel Xeon E5-2690 v3 Haswell CPU with 64 GiB RAM (499.2 double-precision peak Gflop/s), and one NVIDIA Tesla P100 GPU (4.7 peak Tflop/s). The nodes communicate using Cray Aries interconnect. The Linux operating system is Cray CLE/7.0.UP01 (SLES/15). The programming environment used for the native build was:

```
    PrgEnv-gnu/6.0.5
    gcc/8.3.0
    cray-mpich/7.7.10
    craype-haswell
    craype-network-aries
    craype/2.6.1
    slurm/19.05.3-2
    cray-libsci/19.06.1
    cray-netcdf-hdf5parallel/4.6.3.1
    blitz/1.0.2
```

To build and run the container images, we used docker/17.06.0-ce and sarus/1.0.1. The programming environment used for the container build was:

```
ubuntu/18.04.3
gcc/7.4.0
mpich/3.1.4
hdf5/1.8
netcdf-c/4.7.2
netcdf-f/4.5.2
blitz/1.0.2
```

We were given access to 160 nodes (1920 cores). Two validation tests have been performed.

Reproducibility test

Two executions have been launched changing the domain decomposition; in the first run we used 4x2 cores and in the second one we used 2x4 decomposition, than we compared the netcdf output files.

Restartability test

One execution is launched for 1080 time steps, during which the restart files are stored at step 540, we refers this run as LONG run. The second execution is launched starting from the restart files created in the previous one, we refers to as SHORT run. We, thus, compare the outputs files got from the LONG run and the ones got from the SHORT run.

We executed both tests on Daint and on mac-os and all the validation tests have been passed.

Scalability test

The aim of this test is to verify the performance of the container when executing parallel application on parallel architectures. With these tests we compared the execution time obtained from a run using the nemo container against the execution time obtained with NEMO natively compiled on the target machine. We conducted a strong and weak scaling study using he gyre configuration as it allows to easily change the space resolution. We fixed the workload for each core by using a subdomain made of 152 x 69 points. The results are depicted in figure 1:

The execution time here reported does not include the time spent for the first ten time steps during which typically some I/O operations and initialization are performed and we also omitted the time spent in the last ten time steps where the output and restart files are written. In other world, the considered execution time excludes the I/O and initialization phases at all. The jump in the execution time, that we have between 6 cores and 24 cores, is because we pass from one node to two nodes but also because with 6 cores we partially saturate the node, while with 24 cores two nodes are entirely occupied. The weak scaling curve reveals that the containerization of the NEMO model does not introduce any significant penalties in the computational performance.

We executed also the strong scalability analysis by using a global domain made of 2400 x 1600 points which represents the workload of a 1/8 degree model resolution. Because of memory constraints, we were able to execute this configuration by using 8 nodes (96 cores) at least. Namely, we used 96, 384 and 1536 cores.

The speedup curve is satisfying with a parallel efficiency greater than 70%. Also in this case, the I/O and the initialization phases are omitted. The tests again proved that the use of the container does not introduce any performance lost.

Moreover, we performed a strong scalability test with a smaller resolution which defines a global domain with 1200 x 800 points (this represents the workload of a 1/4 degree resolution).This resolution allowed us to start the analysis with one node up to 128 nodes (1536 cores). The resulta reported in the following figure

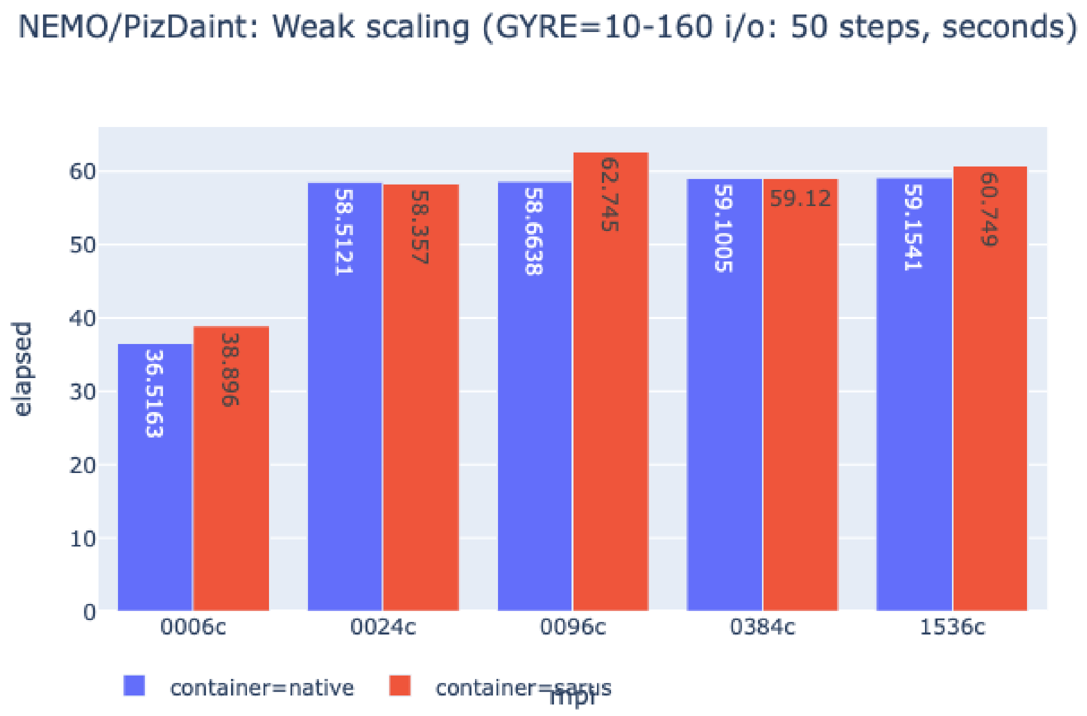

Figure 1: weak_gyre

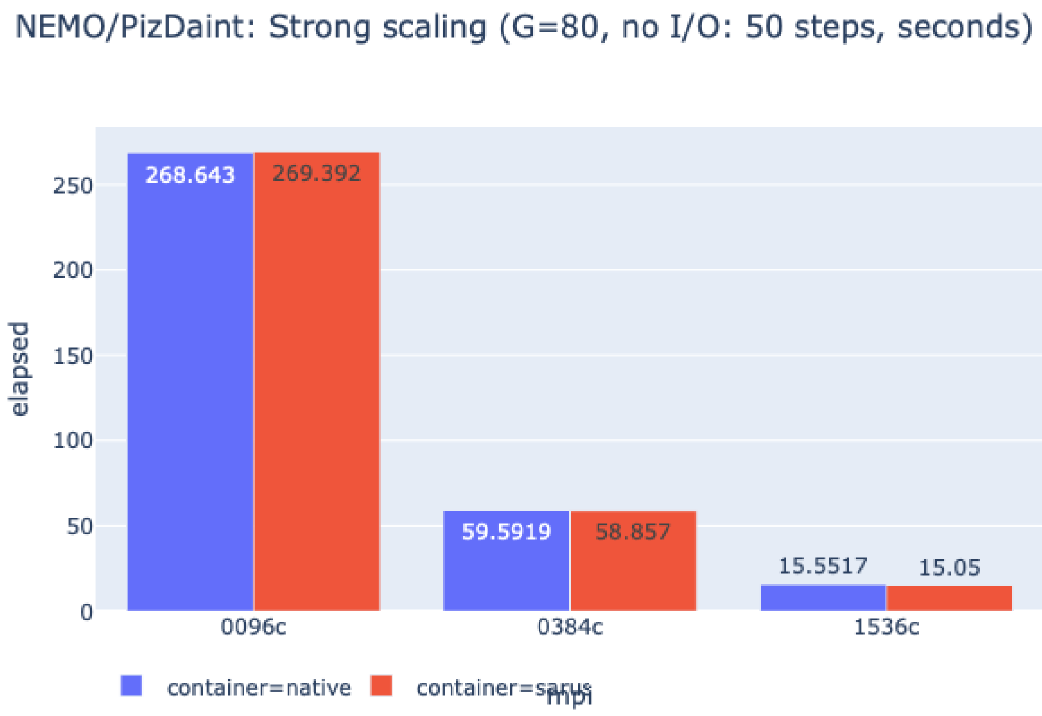

Figure 2: strong_gyre80

surprisingly highlighted a superlinear speedup which could be explained with a better exploitation of the memory hierarchy available in the computing nodes when the dimension of the sub domain becomes smaller.

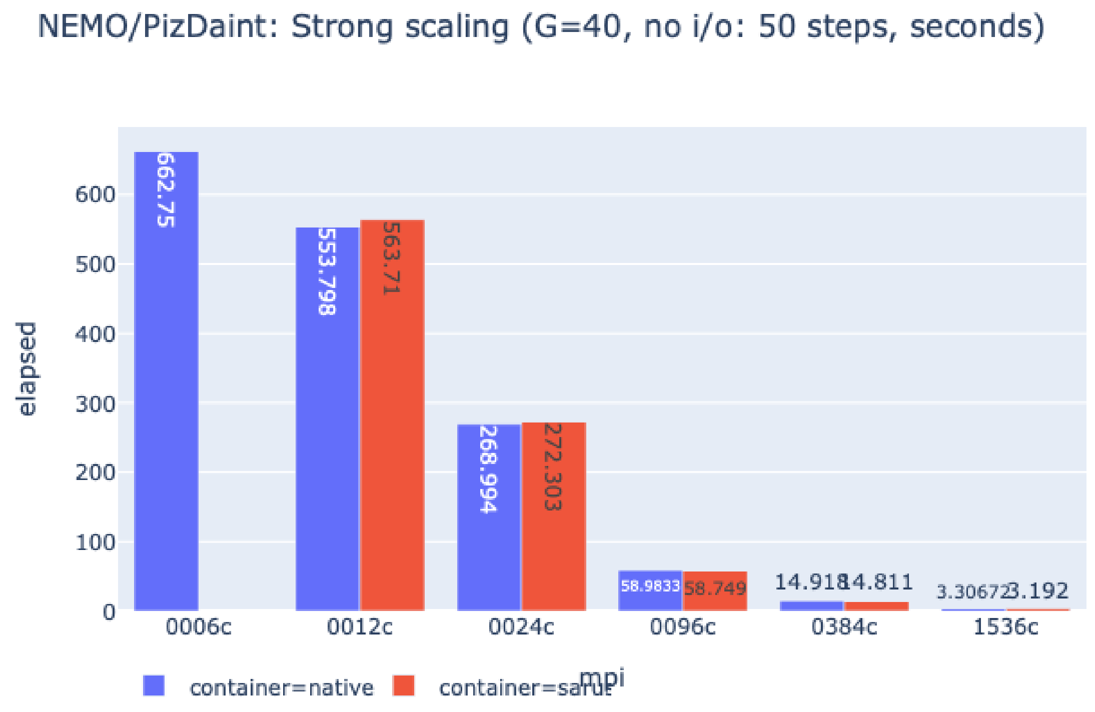

Figure 3: strong_gyre40

Finally, we used the ORCA_ICE_PISCES configuration for testing the performance behavior of the NEMO container when dealing with a realistic configuration taking also into account the time for reading forcing data and to periodically write the prognostic variables.

Conclusion and final considerations

The learning curve for Docker container has been quite rapid; in less than one week we had the first working NEMO image. The tuning of the image parameters and its porting to a parallel architecture has been immediate thanks to the expertise of the mentor who has followed the activity. The most positive unexpected results we had, was related to the computational performance achieved with the containerized version of NEMO. That version behaves exactly as the native one. As last comment we cite a practical use of the container we experienced. Considering that typically a climate modeler launches a bash runscript that prepares the working directory for the execution and at the end launches the model in parallel, we chose to include in the container only the execution of the executable file, this, in turn, implies that we still have the runscript for preparing the working directory but instead of launching the executable in parallel, an instance of the container is created and executed in parallel.

Reproducibility experiments

GYRE_namelist/repro_2x4_namelist_cfg

```
nn_it000=1
```

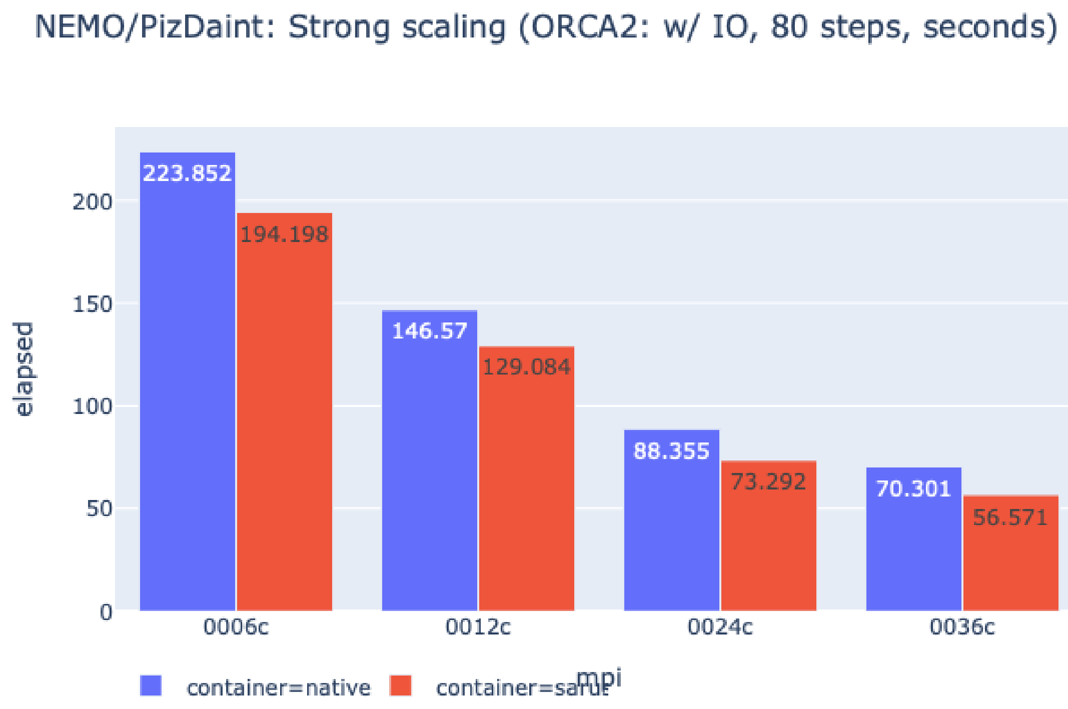

Figure 4: strong_orca

```
nn_itend=1080
nn_GYRE=1
ln_bench=.false.
jpnj=2 <---
jpni=4
ln_timing=.true.
```

GYRE_namelist/repro_4x2_namelist_cfg

```
nn_it000=1
nn_itend=1080
nn_GYRE=1
ln_bench=.false.
jpnj=4 <---
jpni=2
ln_timing=.true.
```

```
for f in `\ls GYRE*.nc | grep -v restart`; do echo $f; cdo diff $f ../REPROD_JG_4x2_G/$f; done
```

Restart experiments

```
for f in `\ls *SHORT*restart*.nc`; do
    echo $f; f1=${f/SHORT/LONG};
    cdo diff $f ../RESTART_LONG_G/$f1;
done
```