eCSE project eCSE08-11

Coupling across the Continuum-Particle Divide with code_saturne and GROMACS

Charles Moulinec[1], Yvan Fournier[2], Marta Garcia[3], Victor Lopez[3], Wendi Liu[1] and James Gebbie-Rayet[1]

[1] UKRI-STFC, Sci-Tech Daresbury, Warrington, WA4 4AD, UK

[2] EDF R&D, 78400, Chatou, FR

[3] Barcelona Supercomputing Center, 08034, Barcelona, SP

Abstract

Simulating interaction between biological structures requires knowledge about whether they would repel or attract each other when they are close to each other. This work explains how to build a workflow to inform on this interaction. The macro-structure is modelled using code_saturne, the micro-structure using GROMACS and they are coupled together through the PLE library. The coupling being staggered, the same computational resources can be used by each software, when the other one is idle. This optimisation is orchestrated by the DLB library. Tests carried out on ARCHER2 show good performance up to 64 GROMACS instances running concurrently on 64 nodes of the machine.

Keywords Multi-scale workflow - Coupling - Resource optimisation

1 Introduction

Simulation is becoming ever more important in the life sciences as the complexity of both research problems and information from lab-based experiments increase. This is reinforced by the award of the Nobel Prize in Chemistry in 2013 for the development of multi-scale simulation methodologies for complex chemical systems [1]. In the biomolecular simulation community to date, multi-scale modelling has largely focused on loosely coupling together two particle-based methodologies. However, these couplings are limited in the range of scales and applications that they can deal with. There are a number of methodologies that are capable of running simulations at larger length scales, each of which has slightly different approaches to fluidics and neighbour interactions. Each of these methodologies requires either detail of the atomic positions which can then be coarsened into larger particle components (beads) or requires the partition of the system using a strategy such that the system is composed of particulate repeat units. This parameterisation is often painful to configure for users and lends itself to parameterisation from the traditional routes of crystal structures for problems in the life sciences. A novel mesoscale methodology has been developed at Leeds University [7, 8] which uses a continuum approach to modelling the thermal dynamics of such biological systems using Fluctuating Finite Element Analysis (FFEA), which relies on meshes to describe the biological system. This approach allows orders of magnitude time- and length-scale increase over traditional particle-based methodologies. It is thus possible to be able to very rapidly sample states for mesoscale systems on time scales that are simply unattainable using particle-based methods without unfeasibly long access to very large high performance computing. This work is part of a long-term plan that aims to better understand processes occurring in crowded cellular environments. The number of proteins per cell is of the order of tens of millions, and no open-source software is currently able to jointly simulate their time-based evolution and potential interaction (splitting, sticking, and binding) at this scale.

This particular project focuses on the implementation of a multi-scale workflow, and not the physics itself, to help understand how large systems containing many proteins at the macro-scale either bind together, or repel each other, when they are situated within a crowded cellular environment as parameterised by micro-scale simulations. To achieve this, information from the micro-scale (solved by molecular dynamics (MD)) is required by the macro-scale (solved by a mesh-based method). On the computational side, it means that the protein evolution is simulated at the macro-scale, using a computational structural mechanics (CSM) approach (code_saturne [5, 3]), and potential binding/repulsion is handled at the micro-scale, using a molecular dynamics approach (GROMACS [2]). A coupling between CSM and MD is required to exchange information between the scales to decide which action follows (binding/sticking or repulsion). Generalising this investigation to many proteins (at least several hundreds of thousands within a single cell) can only be possible using high performance computing (HPC).

This report is organised as follows. Section 2 gives a description of the coupling, Section 3 a short introduction to the software and libraries used by the workflow, Section 4 shows the steps to prepare GROMACS for MPMD simulations, Section 5 how both code_saturne's and GROMACS' installations have to be adapted, Section 6 presents the coupling between code_saturne and GROMACS. Section 7 gives a short description of the test case, Section 8 explains how the coupling is run, Section 9 presents some results, before Section 10 draws some conclusions, and gives some directions for future work.

2 Description of the coupling

The coupling is multi-scale, the micro-scale simulations carried out using GROMACS informing the macro-scale simulation performed using code_saturne. Both software run in a staggered way, code_saturne being paused, when GROMACS runs before it feeds back on information whether the proteins would repel or attract each other. Several alternatives are considered to tackle the problem. The first one is to rely on MPI for this, using `MPI_Spawn` to run the GROMACS processes. Unfortunately, this instruction is not supported by MPICH on ARCHER2. The second approach would be based on bash scripting together with

`MPI_Comm_split` in both software. However, both these approaches would not make the coupled workflow robust enough. It is then decided to couple code_saturne and GROMACS through the PLE library [4], using its integrated coupling strategy. That also means that the integration in code_saturne is lightweight, compared to GROMACS'.
To take full advantage of the fact that the coupled workflow is staggered, it is also decided to get best use of the computational resources, with GROMACS using code_saturne resources while this one is idle. The DLB library [6] offers this option and is implemented in this work.

3 Short introduction to software and libraries

code_saturne and GROMACS are coupled using PLE, and runtime optimisation is achieved by DLB. code_saturne's license is GNU GPL v2 or above, and GROMACS's, PLE's and DLB's GNU LGPL v2.1 or above.

3.1 code_saturne

code_saturne is an open-source multi-purpose multi-physics industrial software primarily developed by EDF. Each release contains a legacy part for computational fluid dynamics that relies on the finite-volume method to discretise the Navier-Stokes equations [5], and a newly fully integrated discretisation based on the compatible discrete operator (CDO) method mainly to handle other physics [3]; the latter is used in this work. The software is written in C/C++ (80%), Fortran (5%) and Python (15%), and uses the hybrid MPI+OpenMP paradigm to handle parallelism on CPU distributed memory machines. The code is based on a "ghost cell" method for both parallelism and periodicity. MPI-IO is used to handle input/output for large meshes, and serial IO for much smaller ones. code_saturne version 8.3.1 is used in this project.

3.2 GROMACS

GROMACS is an open-source versatile package used to perform molecular dynamics [2]. It is primarily designed for biochemical molecules like proteins that have a lot of complicated bonded interactions. It is written in C++ and C, and the hybrid MPI+OpenMP paradigm is used to handle parallelisation on CPU distributed memory machines. GROMACS is also very efficient on GPUs, but this version of the software is not used in this project. GROMACS version 2022.4 is used here, but later ones would be easily adaptable.

3.3 PLE

The PLE (Parallel Location and Exchange) library [4], developed by EDF, aims at providing high performance utility features, including coupling to code_saturne, in a manner also usable by other software. It is written in C for maximum portability, and is very efficient on CPU distributed memory machines. Version 2.0.4 of PLE is used here.

3.4 DLB

The DLB (Dynamic Load Balancing) library [6], developed by BSC, is a collection of tools targeting HPC applications. Its main goal is to help with the dynamic load balancing of hybrid codes, which use a nested parallelism model e.g MPI+OpenMP. DLB provides several components to modify resources at runtime, as well as a profiler to gather performance metrics. These are: LeWI, DROM, TALP and DLB_Barrier. The DLB_Barrier functionality is the only one used in this work, to orchestrate the distribution of the resources in the workflow. GROMACS containing its own internal DLB library that has a different purpose than this DLB developed by BSC, it is decided to change the BSC library name from `dlb` to `dlb_dlb` and the header `dlb.h` to `dlb_dlb.h`. Version 3.5 of DLB is used here.

4 Preparing GROMACS for MPMD simulations

4.1 Adding a new local world communicator to GROMACS

To be able to run MPMD simulations using code_saturne and GROMACS, both software need an internal world communicator to run the independent parts of their share, when they are both coupled. If this exists in code_saturne, it is not fully implemented in GROMACS. Some operations, mainly to broadcast the command line options and to detect hardware are carried out using `MPI_COMM_WORLD`. A new local communicator is required, through a new function to be implemented, to prevent GROMACS from hanging when coupled with code_saturne. The new files are called:

- `localworldcommunicator.h`
- `localworldcommunicator.cpp`

The function `LocalWorldCommunicator::LocalWorldCommunicator(MPI_Comm world)` contains the communicator, its size and the rank number the process is run on.
An argument is added to `gmx_detect_hardware` to point to the local world communicator, and therefore not use of `MPI_COMM_WORLD` in two `MPI_Allreduce` operations, when detecting the hardware.

4.2 Avoiding broadcasting the options

When using MPMD with code_saturne, GROMACS hangs at the very start of the simulation, because the options are broadcast to all the processes of the coupled simulation that relies on `MPI_COMM_WORLD`. However, GROMACS is written in a way that each process reads the command line (options), so, no broadcast is actually required for MPI-only jobs. The call for `broadcastArguments` is therefore commented out.

4.3 Ignoring thread affinity

GROMACS checks for thread affinity, through the function `gmx_check_thread_affinity_set`. This is to be found in `threadaffinity.cpp`. To achieve this, an `MPI_Allreduce` operation based on `MPI_COMM_WORLD` is implemented. As the thread affinity feature is not used in this work, the two calls to the function `gmx_check_thread_affinity_set` have therefore been commented in the file `runner.cpp`. The following functions should be worked on in the future, to account for the local world communicator.

1. `detectDefaultAffinityMask`
2. `gmx_check_thread_affinity_set`
3. `Mdrunner::mdrunner()`

5 Adapting code_saturne and GROMACS installations

code_saturne and GROMACS installations have to be adapted for each software to communicate with each other through PLE, but also for both of them to support DLB.

5.1 Steps for code_saturne to see GROMACS

5.1.1 Addition to the configuration step

code_saturne's installation relies on `GNU Autotools`. To add GROMACS as an external library, the following steps are taken:

- A new file is created under the `m4` folder, called `cs_gromacs.m4`
- A new line, `m4_include([m4/cs_gromacs.m4])` is added to the original file `aclocal.m4`
- Two new lines are added to the file called `configure.ac`, namely `CS_AC_TEST_GROMACS` and `echo " GROMACS support: "$cs_have_gromacs""`
- The following lines are added to the file `code_saturne_build.cfg.in.in`:

```
[gromacs]
have: @cs_have_gromacs@
cppflags: @GROMACS_CPPFLAGS@
ldflags: @GROMACS_LDFLAGS@
libs: @GROMACS_LIBS@
```

To make sure that these changes are accounted for, `./sbin/bootstrap` is typed in the `code_saturne` folder. `$GROMACSPATH` being the environment variable pointing to GROMACS's installation, the options to be passed to the `configure` instruction read:

```
./configure    \
....... \
--with-gromacs=yes   \
--with-gromacs-lib=$GROMACSPATH/lib \
--with-gromacs-include=$GROMACSPATH/include \
.......
```

In case GROMACS is found, the following line should update on its support, showing this extra line in the list of all the libraries/tools present in the installation:

```
.......
GROMACS support: yes
.......
```

5.1.2 Changes in some code_saturne's Python files

Six Python files are amended, to be able to carry out coupled code_saturne and GROMACS simulations.

- `cs_case_coupling.py`
- `cs_case.py`
- `cs_case_domain.py`
- `cs_exec_environment.py`
- `cs_config.py`
- `cs_create.py`

Each coupled simulation (study and case) is prepared as:

```
code_saturne create --study CS_STUDY --case CS_CASE --gromacs GROMACS_CASE
```

where:

- `CS_STUDY` is the main folder for the simulation
- `CS_CASE` is the folder that manages code_saturne's simulation
- `GROMACS_CASE` is the folder that manages GROMACS's simulation(s)

The number of GROMACS's simulations, and their respective folder is prepared by a bash script (see Appendix 1).

5.2 Steps for code_saturne to see DLB

Following similar steps as for adding GROMACS to code_saturne (see Section 5.1.1), DLB is added to the list of libraries supported by code_saturne. A new file `cs_dlb.m4` is created, and the following lines are added to the file `code_saturne_build.cfg.in.in`:

```
[dlb]
have: @cs_have_dlb@
cppflags: @DLB_CPPFLAGS@
ldflags: @DLB_LDFLAGS@
libs: @DLB_LIBS@
```

On ARCHER2, code_saturne requires that the following options are added to the `configure` instruction:

```
--with-dlb=yes \
--with-dlb-lib=$DLBPATH/lib \
--with-dlb-include=$DLBPATH/include \
CPPFLAGS="-I$DLBPATH/include" \
LIBS="$DLBPATH/lib/libdlb_instr.a -L/usr/lib64 -lpthread -lrt -lhwloc"
```

It is also required to add the path to DLB library to the `${LD_LIBRARY_PATH}$` environment variable.

5.3 Steps for GROMACS to see both external libraries, PLE from code_saturne

GROMACS' installation relies on CMake. A file called `FindEXTDLB.cmake` is added under the `cmake` folder of the distribution and the following lines are added to the main `CMakeLists.txt` file:

```
option(GMX_EXTPLE "Add the PLE library" OFF)
mark_as_advanced(EXTDLB)

if (GMX_EXTPLE)
  find_package(EXTPLE)
  if(EXTPLE_FOUND)
    include_directories(SYSTEM ${EXTPLE_INCLUDE_DIR})
    list(APPEND GMX_COMMON_LIBRARIES ${EXTPLE_LIBRARY}/libple.a)
    set(HAVE_EXT_PLE 1)
  else()
    message(FATAL_ERROR "External PLE library was not found.
    Please add the correct path to CMAKE_PREFIX_PATH")
  endif()
endif()
```

5.4 Steps for GROMACS to see the external DLB

DLB is plugged into GROMACS in the same way as PLE, using a new file called `FindEXTDLB.cmake` and the following lines are added to the main `CMakeLists.txt` file:

```
option(GMX_EXTDLB "Add the DLB library" OFF)
mark_as_advanced(EXTDLB)

if (GMX_EXTDLB)
  find_package(EXTDLB)
  if(EXTDLB_FOUND)
    include_directories(SYSTEM ${EXTDLB_INCLUDE_DIR})
    list(APPEND GMX_COMMON_LIBRARIES ${EXTDLB_LIBRARY}/libdlb_instr.so)
    set(HAVE_EXT_DLB 1)
  else()
    message(FATAL_ERROR "External DLB library was not found.
    Please add the correct path to CMAKE_PREFIX_PATH")
  endif()
endif()
```

5.5 Remark on the installation of the workflow

As the installation of the PLE library happens during the installation of code_saturne, it is required to install code_saturne twice, the first time for PLE to be available for GROMACS's installation. The following steps should be followed:

1. Install DLB
2. Install code_saturne that sees DLB (without GROMACS)
3. Install GROMACS that sees DLB and PLE (available from code_saturne)
4. Re-install code_saturne (that already sees DLB) for it to see GROMACS

Note that it might be possible to install PLE directly, meaning that Step 2, where code_saturne is installed in full, would be different, but this has not been tested in this work.

6 Coupling macro- (code_saturne) and micro-scale (GROMACS)

The coupling between code_saturne and GROMACS is carried out using the PLE library, as code_saturne is already PLE-compliant. All the developments related to the PLE library are inspired by an existing coupling, between code_saturne and another software called SYRTHES. However, this coupling involves exchanges of data at mesh boundaries or between mesh volumes, which is not required for the present work. This part is therefore not implemented in the coupling between code_saturne and GROMACS.

6.1 Coupling code_saturne to GROMACS through PLE

Following the example of the file called `cs_syr_coupling.cpp`, and its corresponding header, a new file called `cs_gro_coupling.cpp`, and its corresponding header are created. They contain the following public functions, to be called within code_saturne to call for the coupling:

- `cs_gro_coupling_define` which defines new GROMACS couplings
- `cs_gro_coupling_all_init` which initialises new GROMACS couplings
- `cs_gro_coupling_all_finalize` which finalises new GROMACS couplings
- `cs_gro_coupling_n_couplings` which returns the number of GROMACS couplings
- `cs_gro_coupling_log_setup` which informs on the coupling with GROMACS

The structure used by code_saturne for the coupling with GROMACS reads:

```
typedef struct {
  char               *gro_name;        /* Application name */
  char                time_step_mode;  /* Time stepping mode */
  int                 verbosity;       /* Verbosity level */
  int                 visualization;   /* Visualization output flag */

  /* Communication-related members */
#if defined(HAVE_MPI)
  MPI_Comm            comm;            /* Associated MPI communicator */
  int                 n_gro_ranks;     /* Number of associated GROMACS ranks */
  int                 gro_root_rank;   /* First associated GROMACS rank */
#endif
} cs_gro_coupling_t;
```

6.2 Coupling GROMACS to code_saturne through PLE

GROMACS requires `gmx_mpi`, its MPI-ied executable to know that it is coupled with code_saturne but GROMACS also needs some functions to activate its coupling through PLE.

6.2.1 Addition of -app-name as a command line option

This is done in the file `legacymdrunoptions.h`, where the size of `pa` is changed from 48 to 49 to add the `-app-name` option, as:

```
        { "-app-name",
          FALSE,
          etSTR,
          { &mdrunOptions.ExternalCoupling },
          "HIDDENExternal coupling, most certainly with code_saturne "}
        } // CM: Option for coupling with code_saturne
```

The structure `MdrunOptions` is updated in `mdrunoptions.h`, as:

```
//! \internal \brief Collection of all options of mdrun that are not processed separately
struct MdrunOptions
{
    ..................
    //! Potential external coupling with code_saturne
    const char* ExternalCoupling = nullptr;
};
```

6.2.2 Instrumentation of GROMACS with PLE

Two new structures are added to `mdrun.cpp` as:

```
struct cs_gro_coupling_t
{
  char            *app_name;        /* application name, if given */

  MPI_Comm         comm;            /* Associated MPI communicator */

  int              n_dist_ranks;    /* Number of associated distant ranks */
  int              root_dist_rank;  /* First associated distant rank */

  cs_gro_coupling_type_t   type;
};

struct cs_gro_coupling
{
  int    app_num;      /* App. num for the GROMACS executable */
  char  *name;         /* Name of the current GROMACS executable
                          in the coupling process */

  char  *wdir;         /* application working directory, if given */

  int    do_coupling;  /* 0 = do not code coupling, else yes */

  int    n_couplings;  /* Number of cfd codes coupled with GROMACS */
  char **c_names;      /* Related name instances with each cfd code */

  cs_gro_coupling_t    **couplings; /* Array of pointers to cs_gro_coupling_t
                                       structures */
};
```

Now that a local world communicator is available (see Subsection 4.1), it is possible to use `MPI_Comm_split` to create it for GROMACS in `mdrun.cpp` and to instrument it using PLE functions.

6.3 Resource optimisation using DLB

The coupling between code_saturne and GROMACS is staggered, meaning that when code_saturne runs, GROMACS is idle, and vice versa. It is therefore possible for the 2 software to use the same computational resources, when the other one is idle. To make sure that GROMACS' instances are called and released when needed, DLB is used through its internal barriers. Two barriers are defined in each software, and their activation is presented in the following.

6.3.1 DLB in code_saturne

The additions are carried out in the file called `cs_cdo_main.cpp`, that manages all CDO simulations, and specifically in the function `cs_cdo_main`. DLB is initialised and two barriers are created as:

```
  DLB_Init(0, NULL, NULL);

  dlb_barrier_t* barrier1 = DLB_BarrierNamedRegister("Barrier_1",
  DLB_BARRIER_LEWI_OFF);
  dlb_barrier_t* barrier2 = DLB_BarrierNamedRegister("Barrier_2",
  DLB_BARRIER_LEWI_OFF);

  /* DLB requires every process to finalize DLB_Init before anyone
  calls DLB_Barrier. */
  MPI_Barrier(MPI_COMM_WORLD);
```

The call to the barriers are performed at the end of the time loop, at a specific time-step chosen by the user. In the example taken below, the 5^{th} one is selected. To be sure that all the processes of the coupling are synchronised an `MPI_Barrier` is set for code_saturne local world communicator `cs_glob_mpi_comm`.

```
    if (domain->time_step->nt_cur == 5) {

      MPI_Barrier(cs_glob_mpi_comm);
```

```
    DLB_BarrierNamed(barrier1);
    DLB_BarrierNamed(barrier2);
  }
```

At the end of the simulation, still in `cs_cdo_main`, DLB is terminated using the following instruction:

```
DLB_Finalize();
```

Note: To get the coupling with GROMACS working in the current version of code_saturne, it is required to add a condition on SYRTHES coupling into the `cs_domain.cpp` file, to avoid any unexpected synchronisation at this stage. This reads:

```
bool
cs_domain_needs_iteration(cs_domain_t  *domain)
{
  bool  one_more_iter = true;
  cs_time_step_t  *ts = domain->time_step;

  if (cs_syr_coupling_n_couplings() > 0) /* CM: New condition */
    cs_coupling_sync_apps(0,      /* flags */
                          ts->nt_cur,
                          &(ts->nt_max),
                          &(ts->dt_ref));
```

6.3.2 DLB in GROMACS

DLB is initialised in the function `Mdrunner::mdrunner()` of `runner.cpp` as:

```
    DLB_Init(0, NULL, NULL);

    dlb_barrier_t* barrier1 = DLB_BarrierNamedRegister("Barrier_1",
    DLB_BARRIER_LEWI_OFF);
    dlb_barrier_t* barrier2 = DLB_BarrierNamedRegister("Barrier_2",
    DLB_BARRIER_LEWI_OFF);

    /* DLB requires every process to finalize DLB_Init before anyone
    calls DLB_Barrier. */
    MPI_Barrier(MPI_COMM_WORLD);
```

The first barrier is called immediately after these intructions to pause GROMACS as:

```
    DLB_BarrierNamed(barrier1);
```

The second barrier is called at the end of the function, after making sure that all the GROMACS processes are synchronised and just before finalising the call to DLB:

```
    MPI_Barrier(cr->mpi_comm_mysim);
    DLB_BarrierNamed(barrier2);

    DLB_Finalize();
```

6.3.3 Description of the algorithm involving the DLB barriers

The several steps of the algorithm are enumerated as:

1. code_saturne runs while GROMACS is made idle because of its own `barrier`.
2. When code_saturne's `barrier1` is released, GROMACS starts running its share until it hits its own `barrier2`.
3. code_saturne then recognises its own `barrier2` and starts running again, while GROMACS is made idle again.

7 Test case

The main goal of this project being to build a framework for multiscale modelling, without focusing on the physical modelling involved in the protein interaction, a unique simplified test case is used in this work, which is scaled depending on the resources used.

7.1 code_saturne's contribution

The configuration is made of arrays of individual ellipsoids, all of the same size $S = 52,264$ cells. The arrays are created by loading independent ellipsoids, that are then all gathered in the same `mesh_input.csm` file, code_saturne seeing this as a single problem. The vertex-based CDO algorithm is used to solve a Poisson equation with constant right-hand side.

7.2 GROMACS' contribution

For sake of simplicity, the lyzozyme in water tutorial (`http://www.mdtutorials.com/gmx/lysozyme/index.html`) is retained as the test case for any instance of GROMACS to be run. If there are several of them, they would run concurrently. It would be very easy to make use of different inputs for each of these instances. The preparation of the simulation is done prior to the run of the coupling, and input files are copied across when needed. These preprocessing stages read:

```
pdb2gmx -f 1AKI_clean.pdb -o 1AKI_processed.gro -water spce
gmx_mpi editconf -f 1AKI_processed.gro -o 1AKI_newbox.gro -c -d 1.0 -bt cubic
gmx_mpi solvate -cp 1AKI_newbox.gro -cs spc216.gro -o 1AKI_solv.gro -p topol.top
gmx_mpi grompp -f ions.mdp -c 1AKI_solv.gro -p topol.top -o ions.tpr
gmx_mpi genion -s ions.tpr -o 1AKI_solv_ions.gro -p topol.top -pname NA -nname CL -neutral
gmx_mpi grompp-f minim.mdp -c 1AKI_solv_ions.gro -p topol.top -o 1AKI.tpr
```

8 Running the coupling using MPMD

When the coupling is used, code_saturne and GROMACS simulations are performed in different folders, and some care has to be taken to make this possible.

8.1 On a local machine using OpenMPI

In this case, `mpirun` supports the option `-wdir` to assign the correct working directory to the correct code. An example of such an MPMD simulation reads:

```
mpirun --oversubscribe -np 8 -wdir ./CS_CASE/RESU/CS_CASE \
                  ./cs_solver --app-name gromacs --logp : \
       --oversubscribe -np 8 -wdir ./GRO/GRO_0
                    ${GROMACS_PATH}/bin gmx_mpi mdrun -v   \
                    -deffnm 1AKI -app-name code_saturne
```

8.2 On ARCHER2 using the queuing system

Getting the coupling running fine on ARCHER2, between code_saturne and GROMACS through PLE and using DLB requires some care:

1. `srun` being used in conjunction with the `--multi-prog` option assigning the correct process indices, working directories and options to both software is required.

2. `-wdir` needs to be directly implemented as an option in GROMACS itself, and added to the `pa` array (see Subsubsection 6.2.1 that explains how `-app-name` is added).

3. It is assumed that the number of instances of GROMACS that are run is the exact same number as the number of compute nodes used by code_saturne. To achieve this, when more than 2 compute nodes are required, the GROMACS option `-multidir` is activated, with, as an argument, the number of oversubscribed GROMACS instances to be run. This option informs GROMACS to trigger its internal `MPI_Comm_split` procedure, to have as many GROMACS instances as compute nodes, each of these instances being ran independently in their own folder.

4. The same strategy as on the laptop is carried out, by oversubscribing each node with half of it being dedicated to part of the code_saturne simulation and the other half by an individual GROMACS instance. In consequence, for each node, the first 128 cores are for a subdomain of code_saturne and the last 128 ones are for GROMACS.

An example of the bottom of the SLURM submission script (to be run on 2 nodes of ARCHER2) reads:

```
.............
cat <<EOF > multiprog.conf
0-127,256-383 cs_solver -wdir ./CS_CASE/RESU/CS_STUDY --app-name gromacs
128-255,384-511 gmx_mpi  mdrun -v -deffnm 1AKI -app-name code_saturne -wdir GCASE -multidir G0 G1
EOF

srun --multi-prog multiprog.conf
```

The file `multiprog.conf`, to be used in conjunction of the `--multi-prog` option for `srun` contains 2 lines. The first one has got the list of the core/MPI processes code_saturne is run on, as well as the name of the executable and its options, and the second one, the same syntax, but for GROMACS. Distributing the load this way e.g. half of each compute node for code_saturne and the other half for GROMACS allows the DLB library to optimise the local-to-each-node resources, that share the same memory, when running these MPMD simulations.

9 Results

The first subsection shows the potential of code_saturne on its own, to load many independent meshes, knowing that there exist tens of millions of proteins in each biological cell. The second subsection shows the performance of the coupling between code_saturne and GROMACS on ARCHER2.

9.1 Many meshes as input for code_saturne

code_saturne is used because of its ability to handle extremely large meshes. Single meshes are usually dealt with by the software. However this work is about assessing how many N independent meshes of size S it can work with, and for the software to see them as a single mesh of size $N \times S$ made of non-contiguous parts. A set of N ellipsoids is looked at, each of them made of $S = 52,264$ cells.

Number of compute nodes	Number of ellipsoids	IO strategy	Total number of cells	Time to load all the files one by one
4	8	MPI-IO	418,112	0.4 s
4	64	MPI-IO	3,344,896	2.8 s
32	512	MPI-IO	26,759,168	56.9 s
32	4,096	MPI-IO	214,073,344	437.3 s
32	32,768	MPI-IO	1,712,586,752	2833.1 s
256	262,144	Serial	13,700,694,016	3495.1 s
Number of compute nodes	Number of ellipsoids	IO strategy	Total number of cells	Time to load the arrays of files one by one
512	1,048,576	MPI-IO	54,802,776,064	64 files in 24214.75 s
1,024	2,097,152	MPI-IO	109,605,552,128	2 files in 2591.61 s

Table 1: Size of the final meshes and time to load them by parts of ellipsoid meshes.

The first 6 rows of Table 1 show the total number of loaded ellipsoid meshes, as well as the time and IO strategy to load them. When increasing the number of fully populated compute nodes, serial IO gets much faster than MPI-IO. For instance, $262,144$ independent ellipsoids are loaded in ARCHER2 memory in less than 1 hour. The step to load them all can be seen as preprocessing, as, after this step, a file containing all these individual meshes is dumped onto the disk, and reloaded if needed, using MPI-IO, and also striping, in the case of the $13,700,694,016$ cell mesh, when using Lustre, for instance.
The last two rows of Table 1 show the results for loading over 1 million (resp. 2 millions) of ellipsoids from pre-computed arrays of ellipsoids.
A test is carried out on the NVME partition of ARCHER2, to write the large mesh of 13,700,694,016 cells on the disk. It takes about 775 seconds to dump the 1.5 TB file on the disk using the `-c -1` striping option.

Note that solving a Poisson equation of the 109 billion element mesh using 1,024 nodes of ARCHER2 in conjonction with the CDO formulation only takes only 300.50 seconds (BiCGstab2 is used as a solver and Jacobi as a preconditioner).

9.2 Performance of the coupling up to 64 compute nodes

Table 2 shows the performance of the coupling on 4, 8, 16, 32, and 64 nodes of ARCHER2, respectively. The case on 4 nodes is the baseline for code_saturne. Its mesh size, as well as the number of individual ellipsoids are doubled with the number of compute nodes, mimicking a weak scaling test for code_saturne when it runs on its own. GROMACS is called at code_saturne 5^{th} iteration.

Number of compute nodes (code_saturne+GROMACS)	Number of ellipsoids	Compute time of zeroth time-step	Averaged compute time per time-step without GROMACS	Compute time with GROMACS
4 (2 + 2)	64	29.93 s	0.15 s	2.57 s
8 (4 + 4)	128	31.02 s	0.16 s	3.78 s
16 (8 + 8)	256	61.02 s	0.16 s	3.20 s
32 (16 + 16)	512	164.56 s	0.19 s	3.41 s
64 (32 + 32)	1,024	647.76 s	0.27 s	4.01 s

Table 2: Compute time in code_saturne in case of coupling with GROMACS.

Figure 1 shows the time spent during the zeroth time step of code_saturne when the meshes are loaded (left), the time spent in code_saturne alone (centre), when each MPI task is loaded in the same way, and the time spent in GROMACS instances (right), when each of the nodes carries out the same simulation.

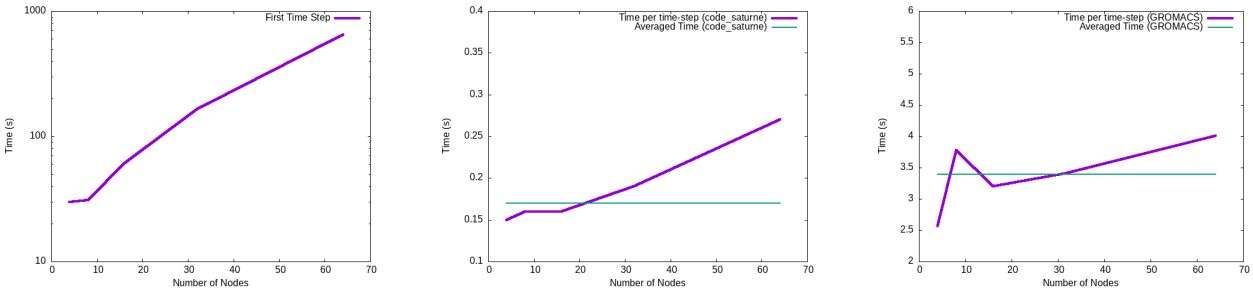

Figure 1: Left: Time spent in the zeroth time step. Centre: Time spent in code_saturne alone. Right: Time spent in GROMACS alone.

10 Conclusions - Future Work

A workflow to inform how biological entities would interact with each other is implemented, using code_saturne for the macro-scale, GROMACS for the micro-scale, with the PLE library to couple them both and the DLB library to optimise computational resource. It is tested up to 64 nodes of ARCHER2, meaning that, when required, 64 concurrent GROMACS instances are run, using the computational resrouces of code_saturne that is then made idle.
In the future, the workflow will be profiled, and a larger number of nodes/GROMACS instances will be tested. Finally, the physics involved in the coupling will be added.

Acknowledgements

This work is funded under the embedded CSE programme of the ARCHER2 UK National Supercomputing Service (http://www.archer2.ac.uk). The simulations are carried out on ARCHER2 thanks to the Service Level Agreement between STFC and EPSRC. The authors also would like to thank ARCHER2 support, and especially David Henty for his excellent suggestions.

References

[1] https://www.nobelprize.org/prizes/chemistry/2013/summary/.

[2] M.J. Abraham, T. Murtola, R. Schulz, S. Páll, J.C. Smith, B. Hess, and E. Lindahl. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1-2:19–25, 2015.

[3] J. Bonelle, Y. Fournier, and C. Moulinec. New polyhedral discretisation methods applied to the richards equation: CDO schemes in Code_Saturne. *Computers & Fluids*, 173:93–102, 2018.

[4] Y. Fournier. Massively parallel location and exchange tools for unstructured meshes. *International Journal of Computational Fluid Dynamics*, 34(7-8):549–568, 2020.

[5] Y. Fournier, J. Bonelle, C. Moulinec, Z. Shang, A.G. Sunderland, and J.C. Uribe. Optimizing Code_Saturne computations on petascale systems. *Computers & Fluids*, 45(1):103–108, 2011.

[6] M. Garcia, J. Labarta, and J. Corbalan. Hints to improve automatic load balancing with LeWi for hybrid applications. *Journal of Parallel and Distributed Computing*, 74(9):2781–2794, 2014.

[7] Robin C. Oliver, Daniel J. Read, Oliver G. Harlen, and Sarah A. Harris. A stochastic finite element model for the dynamics of globular macromolecules. *Journal of Computational Physics*, 239:147–165, 2013.

[8] Albert Solernou, Benjamin S. Hanson, Robin A. Richardson, Robert Welch, Daniel J. Read, Oliver G. Harlen, and Sarah A. Harris. Fluctuating finite element analysis (ffea): A continuum mechanics software tool for mesoscale simulation of biomolecules. *PLOS Computational Biology*, 14:1–29, 03 2018.

Appendix 1

The following script is used to prepare the coupling and to run it on ARCHER2:

```
#!/bin/bash
#
############################################################
####### DEFINITION OF SOME VARIABLES ###################
############################################################
#
export cs_gro_prefix="/work/c01/c01/vcz18385/\
AMD_ROME/GNU/11.2.0/SOFTWARE/ECSE_GROMACS/120525"
#
export gromacs_exe="${cs_gro_prefix}/GROMACS/gromacs-2022.4/install/bin/gmx_mpi "
export codesaturne="${cs_gro_prefix}/SATURNE/8.3.1_GROMACS_eCSE/\
code_saturne-8.3.1/arch/Linux/bin/code_saturne "
export input_files="${cs_gro_prefix}/INPUTS"
#
############################################################
####### INPUT VALUES #######################################
############################################################
#
if [ $# = 10 ]
then
 jobname=$1
 total_nodes=$2
 total_procs=$3
 hours=$4
 minutes=$5
 cs_study=$6
 cs_case=$7
 gro_case=$8
 nsteps=$9
 budget=${10}
else
```

```
  echo "cs_gromacs_submission_script.sh jobname total_nodes \
  total_nprocs hours minutes cs_study cs_case gro_case nsteps budget"
  exit
fi
#
#########################################################################
####### CREATING THE CASE AND STUDY FOR CODE_SATURNE AND GROMACS ####
#########################################################################
#
echo ""
echo "Creating the study for code_saturne and GROMACS"
echo ""
#
${codesaturne} create --study ${cs_study} --case ${cs_case} --gromacs ${gro_case}
#
gro_multidir=""
#
#####################################################
####### COPYING THE REQUIRED FILES ################
#####################################################
#
mesh_input="mesh_input_ELLIPSOID_0_0_0_248_155_155.csm"
#
cp ${input_files}/CODE_SATURNE/${mesh_input} ${cs_study}/MESH/.
cp ${input_files}/CODE_SATURNE/cs_user*.cpp ${cs_study}/${cs_case}/SRC/.
cp ${input_files}/CODE_SATURNE/cs_user_scripts.py ${cs_study}/${cs_case}/DATA/.
#
#####################################################################
####### CREATE AND PREPARE THE FOLDER TO RUN THE SIMULATION ######
#####################################################################
#
folder_id=${cs_study}
cd ${cs_study}/${cs_case}/RESU
${codesaturne} run --initialize --id ${folder_id}
cd -
#
#############################################################################
####### STRING OF CHARACTERS CONTAINING GRO_i * (total_nodes) ############
#############################################################################
#
list_gro_folders=""
gro_0="GRO_0"
gro_wdir=‘pwd‘/${cs_study}/${gro_case}
#
for (( i=0; i<${total_nodes}; i++ ))
do
  mkdir ${cs_study}/${gro_case}/GRO_$i
  list_gro_folders=${list_gro_folders}" GRO_$i"
  if [ ${total_nodes} -gt 1 ]; then
    gro_multidir="-multidir "
    gro_0=""
  fi
  cp ${input_files}/GROMACS/1AKI/* ${cs_study}/${gro_case}/GRO_$i/.
done
#
if [ ${total_nodes} -gt 1 ]; then
  gro_multidir=${gro_multidir}${list_gro_folders}
else
  gro_multidir=""
  gro_wdir=${gro_wdir}/${gro_0}
fi
```

```
#
############################################################
####### SUBMISSION SCRIPT (SLURM + SRUN) - MPMD ##########
############################################################
#
array_size=$((2*${total_nodes}+1))
#
list_nodes_even=""
list_nodes_odd=""
#
declare -a sequence_proc[array_size]
#
for ((i = 0; i <= 2*${total_nodes}; i++)); do
  sequence_proc[i]=$((128*$i));
done
#
add_comma=""
#
for ((i = 1; i <= ${total_nodes}; i++)); do
  if [ $i -gt 1 ]; then
    add_comma=","
  fi
  list_nodes_even=${list_nodes_even}${add_comma}${sequence_proc[2*(i-1)]}-\
  $((${sequence_proc[2*i-1]}-1));
  list_nodes_odd=${list_nodes_odd}${add_comma}${sequence_proc[2*i-1]}-\
  $((${sequence_proc[2*i]}-1));
done
#
cd ${cs_study}
#
cat > $jobname << TAG
#!/bin/bash

# Slurm job options (job-name, compute nodes, job time)
#SBATCH --job-name=$jobname
#SBATCH --output=$jobname.%j
#SBATCH --error=$jobname.%j
#SBATCH --time=$hours:$minutes:00
#SBATCH --nodes=${total_nodes}

#SBATCH --account=$budget
#SBATCH --partition=standard
#SBATCH --qos=standard

module load PrgEnv-gnu

export OMP_NUM_THREADS=1

export DLB_ARGS="--silent"

SARGS="--cpus-per-task=1 --cpu_bind=rank"

cat <<EOF > multiprog.conf
${list_nodes_even} `pwd`/${cs_case}/RESU/${cs_study}/cs_solver \
-wdir `pwd`/${cs_case}/RESU/${cs_study} --app-name gromacs
${list_nodes_odd} ${gromacs_exe} mdrun -v -deffnm 1AKI -app-name code_saturne \
-nsteps 10 -wdir ${gro_wdir} ${gro_multidir}
EOF

srun ${SARGS} --multi-prog multiprog.conf
TAG
```

```
chmod 755 $jobname
sbatch $jobname
```

It is run as:

```
cs_gromacs_submission_script.sh \
    jobname                     \
    total_nodes                 \
    total_nprocs                \
    hours                       \
    minutes                     \
    cs_study                    \
    cs_case                     \
    gro_case                    \
    nsteps                      \
    budget
```