Large Eddy Simulation in a Tube Bundle using *Code_Saturne* and User Subroutines 03_TB_LES_SRC

1 Introduction

Flows through tubes bundles are encountered in many heat exchanger applications. One example is in nuclear power plants where the nuclear fuel rods are cooled by passing cold water over them, but similar arrangements are also employed in more conventional exchangers. The objective of this type of heat exchanger is, of course, to extract as much heat as possible from the tubes. In a computational study, in order to calculate the heat transfer reliably, it is necessary to resolve the flow around the tubes correctly.

A typical staggered tube bundle configuration, which is to be studied in this exercise, can be seen in figure 1. It is assumed to consist of a large number of tubes, so the flow around any one is considered to be similar to that around the others, when the flow is fully developed. A 2D cross-section of the computational domain employed is highlighted (dashed lines). Periodic boundary conditions are applied in the 3 directions, X, Y, Z.

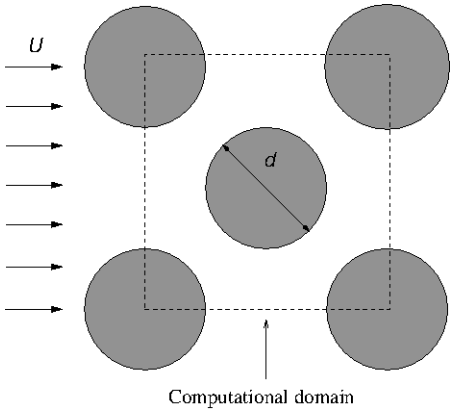

Figure 1: Flow description

The flow is to be computed using Large Eddy Simulation (LES). This technique allows for solving turbulence in time by filtering the flow field and separating large and small scales of turbulence. Here, the Smagorinsky [1] model is used. This is a basic model in

which the filter width is proportional to the grid size, therefore turbulence inside a cell (or control volume, for finite volume-based codes) is modelled, while the rest is resolved.

The mesh of the 2D cross-section representing the domain (see figure 1) contains about 6.5 million control volumes. The diameter of the tubes is $D = 0.0217$ and the pitch divided by the diameter is $P/D = 2.07$. the Reynolds number to be simulated is $Re = 18000$, based on the bulk velocity and the diameter.

The mesh is plotted in figure 2. The 2D plane represented in the figure is extruded along the Z-direction for a length of 2D and over 128 cells.

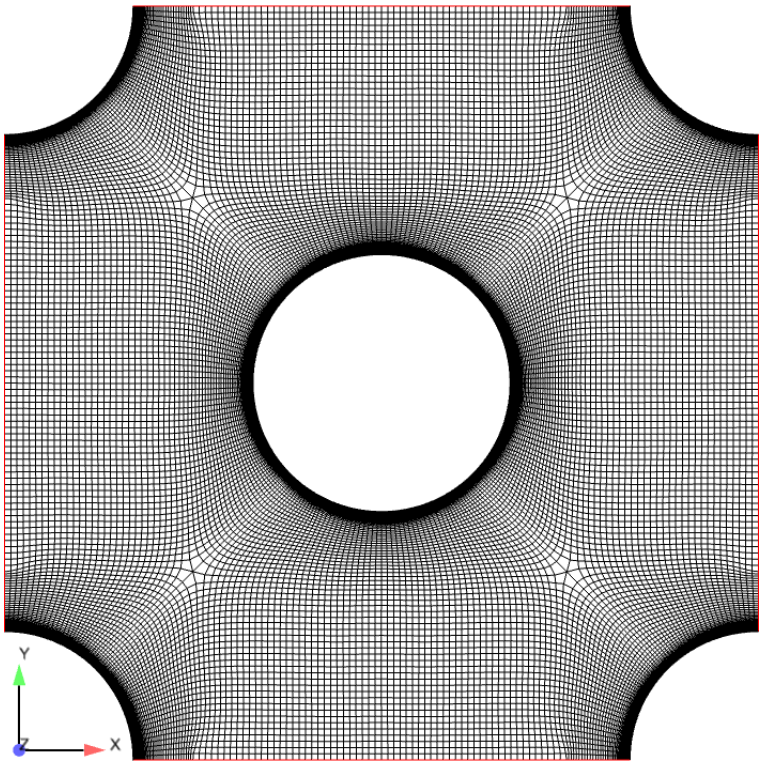

Figure 2: Mesh of 2D cross-section (X - Y)

2 Objectives

In this tutorial, the flow across the tube bundle will be computed using LES. Due to the large size of the mesh, it will be necessary to run the calculation in parallel. The main objective of this session involves:

- Adapting the necessary subroutines (remote supercomputer), including restarting from a previous simulation
- Running the simulation on a remote supercomputer
- Visualisation of the results on the local training machine

Note that this tutorial is tailored for this particular supercomputer and the setup will be different for another type of machine. The remote machine is an IBM Blue Gene/Q called

Blue Joule located at the Hartree Centre, Daresbury Laboratory. It was ranked 13^{th} in the June 2012 SuperComputer Top 500 list. Remote connection will be compulsory and files will have to be copied across.

3 *Code_Saturne*

Code_Saturne is a multipurpose CFD software designed to solve the Navier-Stokes equations for 2D, 2D axisymmetric or 3D flows. Its main module is designed for the simulation of flows which may be steady or unsteady, laminar or turbulent, incompressible or potentially dilatable, isothermal or not. The code employs a finite volume discretisation and allows the use of various mesh types, including hybrid arrangements (where the mesh contains several kinds of elements) and structural non-conformities (hanging nodes).

The code includes specific modules, referred to as 'specific physics', for the treatment of Lagrangian particle tracking, semi-transparent radiative transfer, gas, pulverised coal and heavy fuel oil combustion, electricity effects (Joule effect and electric arcs), compressible flows, but these are not required for the present exercise.

Code_Saturne is open-source; it can be downloaded, modified and/or redistributed under the terms of the GNU General Public License as published by the Free Software Foundation.

The following instructions provide a guide through the steps required to prepare and run the present cases in *Code_Saturne* version 4.0, and subsequently postprocess the results using the open-source postprocessing software *ParaView* and *xmgrace*.

All necessary files are stored in a shared folder available for all the participants. A pdf copy of this tutorial is to be found in:

```
/gpfs/home/training/dxp58/shared/Tutorials/01_TB_LAM_SRC
```

It is recommended to open the pdf file, and to copy and paste the necessary commands to set up the case.

The command to open a pdf file is `evince` and should be used to open the file as follows: `evince 03_TB_LES_SRC.pdf`

4 Guide to run *Code_Saturne*

All the setup is performed on the remote supercomputer as no GUI is used for this tutorial. It is then necessary to connect there by typing:

```
[bash: $] ssh -X username@login.joule.hartree.stfc.ac.uk
```

Software usage and environment variables are loaded by modules. Modules are designed to load variables and parameters to the Linux environment for specific programs, in this case *Code_Saturne* . The following instruction is to be typed to get *Code_Saturne* working:

```
[bash:$]  module load ibmmpi parmetis/4.0.3 scotch/6.0.0 libxml2/2.9.0
[bash:$]  module load code_saturne/4.0.2
```

Typing `code_saturne` allows to check that loading the module has been effective:

```
[bash:$] code_saturne
Usage: /gpfs/packages/ibm/code_saturne/4.0.2/bin/code_saturne <topic>

Topics:
  help
  autovnv
  compile
  config
  create
  gui
  info
  run
  salome

Options:
  -h, --help  show this help message and exit
```

5 Preparation of the simulation

The steps followed in this section are:

- The creation of the study and case
- The modification of the user subroutines
- The compilation and the creation of the executable
- The copy of the mesh, the partition and the restart files to the right location
- The modification of scripts used to run the simulation

5.1 Test case creation

If the **TUBE_BUNDLE_LES** directory does not exist yet, the study needs to be created by

- Going to the desired location for the study
- Typing

```
[bash:$] code_saturne create -s TUBE_BUNDLE_LES -c LES_SRC
```

which returns:

```
code_saturne 4.0 study/case generation
  o Creating study 'TUBE_BUNDLE_LES'...
  o Creating case  'LES_SRC'...
```

Alternatively, if the study **TUBE_BUNDLE_LES** already exists, only the case **LES_SRC** is created in the **TUBE_BUNDLE_LES** directory, as:

```
[bash:$] code_saturne create -c  LES_SRC
code_saturne 4.0 study/case generation
  o Creating case  'LES_SRC'...
```

The file structure can be seen in figure 3. The details of the structure are:

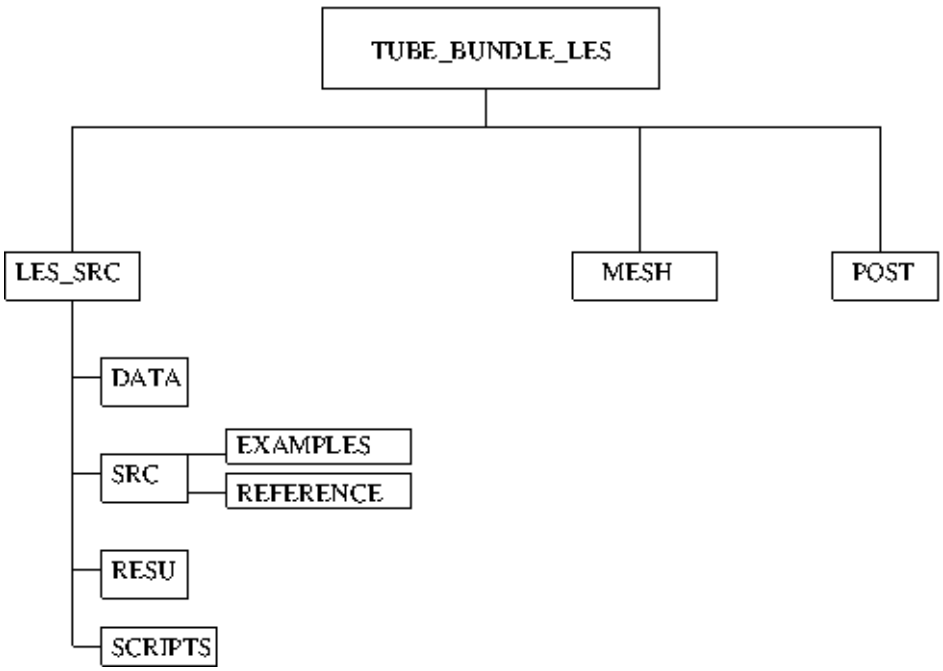

Figure 3: File structure created by *Code_Saturne*

- **LES_SRC**: This is the case file. A study can have several cases, for example each using different turbulence models, different boundary conditions, etc.
  - **DATA**: This directory contains the script to launch the GUI, `SaturneGUI`. This is also where the restarting and other input files should be copied.
  - **SRC**: This is where the user subroutines should be. During the compilation stage the code will re-compile all the subroutines present in this directory. All the user subroutines are available in the **REFERENCE** directory. Examples how to program them can be found in **EXAMPLES**. Depending on what is needed for the simulation only a few of these might be required.
  - **RESU**: This directory contains the files required for a simulation to run as well as the result files.
  - **SCRIPTS**: This is where the script to launch simulations is available.
- **MESH**: The program will read the mesh from this directory. The mesh formats that can be read are I-DEAS, CGNS, Gambit Neutral files (neu), EnSight, pro-star/STAR4 (ngeom), Gmsh, NUMECA Hex, MED, Simail and Meta-mesh files.
- **POST**: This is an empty directory designed to contain postprocessing macros that may be relevant in some cases.

5.2 Case set up

5.2.1 Restart, mesh and partition files

In the following section a restart file generated from a previous calculation will be used. The previous simulation ran over a long period of time and performed so that time averaged quantities could be obtained. The necessary input files will have to be copied from the shared directory. The required files are

- **checkpoint** directory
- mesh_output
- **partition_input**

Copy these into the **DATA** directory by typing:

```
cd LES_SRC
cp -r \
~/../shared/Tutorials/03_TB_LES_SRC/CASEFILES/MESHES/checkpoint DATA/.
cp -r \
~/../shared/Tutorials/03_TB_LES_SRC/CASEFILES/MESHES/mesh_output DATA/.
cp -r \
~/../shared/Tutorials/03_TB_LES_SRC/CASEFILES/MESHES/partition_input \
DATA/.
```

5.2.2 Users subroutines

The subroutines have to be copied in the **SRC** directory. They are obtained by typing the following from the **REFERENCE** directory:

```
[bash:$] cd SRC
[bash:$] cp REFERENCE/cs_user_parameters.f90 .
[bash:$] cp REFERENCE/cs_user_parameters.c .
[bash:$] cp REFERENCE/cs_user_boundary_conditions.f90 .
[bash:$] cp REFERENCE/cs_user_source_terms.f90 .
[bash:$] cp REFERENCE/cs_user_extra_operations.f90 .
```

5.2.3 Description of the subroutines

This tutorial is entirely based on user subroutines modifications. The subroutines employed are:

- cs_user_parameters.f90: Definition of all the calculation parameters.
- cs_user_parameters.c: Definition of time averaged quantities.
- cs_user_boundary_conditions.f90: Definition of the boundary conditions.
- cs_user_source_terms.f90: Used to add source terms to all transport equations.
- cs_user_extra_operations.f90: Used to perform any kind of operation at the end of each time step.

5.3 Detailed changes of the user subroutines

5.3.1 cs_user_parameters.f90

This file contains all the numerical and physical parameters of the calculation that can be changed by the user. Most of them are enclosed inside a test (if statement). This is meant to avoid problems when running both from the files generated by the GUI (`.xml` files) and user subroutines. In case of a parameter being defined by the `.xml` file and by the user subroutine, the value set in the user subroutine will have precedence. Activating a test and setting a value is performed by changing `if(.false.)then` into `if(.true.)then`

Here is the list of all the parameters that have to be changed within the subroutine:

- Open the file with an editor (`gedit` for example [1]).
- Activate the Smagorinsky turbulence model by changing the code at **line 549** so it looks like:

```
if (ixmlpu.eq.0) then
  iturb = 40
endif
```

- Activate the unsteady algorithm with constant time step (**line 706**)

```
if (.true.) then
  idtvar = 0
endif
```

- Set the number of time steps to 16500 (**line 715**)

```
if (.true.) then
  ntmabs = 13000
endif
```

- Set the time step to 7.5×10^{-6} (**line 722**)

```
if (.true.) then
  dtref  = 7.5d-6
endif
```

- The fluid properties and reference values (**line 1078**) are set to

```
if (.true.) then
  ro0    = 1.2d0
  viscl0 = 1.82d-5
  cp0    = 1017.24d0
endif
```

- The reference velocity needs to be prescribed as (**line 1265**)

[1] In `gedit` you can type Crtl-I to go to a specific line numbers. Line numbers are displayed by going to *Edit-Preferences*

```
if (.true.) then
  uref = 12.5d0
endif
```

- The last modification concerns the setup of monitoring points. Modify the text (**line 1536**) as follows:

```
! --- probes output step

if (.true.) then

  nthist = 1
  frhist = -1.d0

endif

! --- Number of monitoring points (probes) and their positions
!     (limited to ncaptm=100)
if (.true.) then
  ncapt  = 6
  tplfmt = 1 ! time plot format (1: .dat, 2: .csv, 3: both)
  xyzcap(1,1) = -0.03d0
  xyzcap(2,1) = 0.0005d0
  xyzcap(3,1) = 0.01d0

  xyzcap(1,2) = -0.015d0
  xyzcap(2,2) = 0.015d0
  xyzcap(3,2) = 0.01d0

  xyzcap(1,3) = 0.015d0
  xyzcap(2,3) = 0.015d0
  xyzcap(3,3) = 0.01d0

  xyzcap(1,4) = 0.0158d0
  xyzcap(2,4) = 0.008d0
  xyzcap(3,4) = 0.01d0

  xyzcap(1,5) = 0.0158d0
  xyzcap(2,5) = 0.0d0
  xyzcap(3,5) = 0.01d0

  xyzcap(1,6) = 0.238d0
  xyzcap(2,6) = 0.0d0
  xyzcap(3,6) = 0.01d0

endif
ineedf = 1 ! activate calculation of the forces on Boundary faces
```

The variable `ineedf = 1` is required to compute forces computed at boundary faces. This is used to calculate lift and drag (see section 5.3.5)

5.3.2 cs_user_parameters.c

In order to compute the time averaged variables, it is necessary to use the cs_user_parameters.c subroutine.

The reference file is empty so it is necessary to fill in the section for the "time moments". An example can be found in the same files under the **EXAMPLES** directory. To define a moment or time averaged value, it is necessary to call the function cs_time_moment_define_by_field_ids. The parameters are:

- name: name of associated moment
- n_fields: number of associated fields
- field_id: ids of associated fields
- component_id: ids of matching field components (-1 for all)
- type: moment type (CS_TIME_MOMENT_MEAN or CS_TIME_MOMENT_VARIANCE)
- nt_start: starting time step (or -1 to use t_start)
- t_start: starting time
- restart_mode: behavior in case or restart: CS_TIME_MOMENT_RESTART_RESET, CS_TIME_MOMENT_RESTART_AUTO, or CS_TIME_MOMENT_RESTART_EXACT
- restart_name: name in previous run, NULL for default

For example, to calculate the time average of pressure, the following should be written (inside de cs_user_time_moments routine, line **212**):

```
{   int moment_f_id[] = {CS_F_(p)->id};
    int moment_c_id[] = {-1};
    int n_fields = 1;
    cs_time_moment_define_by_field_ids("P_mean",
                                       n_fields,
                                       moment_f_id,
                                       moment_c_id,
                                       CS_TIME_MOMENT_MEAN,
                                       10000, /* nt_start */
                                       -1,   /* t_start */
                                       CS_TIME_MOMENT_RESTART_AUTO,
                                       NULL);
}
```

Here the array moment_f_id has the id of the pressure obtained from the CS_F field pointer array. In this case, because the pressure is a scalar, the components array moment_c_id has been set to -1, i.e. all components. In the case of velocity, the function should be called like this:

```
{   int moment_f_id[] = {CS_F_(u)->id};
    int moment_c_id[] = {-1};
    int n_fields = 1;
    cs_time_moment_define_by_field_ids("Vel_mean",
```

```
                                        n_fields,
                                        moment_f_id,
                                        moment_c_id,
                                        CS_TIME_MOMENT_MEAN,
                                        10000, /* nt_start */
                                        -1,    /* t_start */
                                        CS_TIME_MOMENT_RESTART_AUTO,
                                        NULL);
}
```

Finally, to compute the average of UU and UV, the following can be done:

```
  {
    /* Moment <u v>. */
    int moment_f_id[] = {CS_F_(u)->id, CS_F_(u)->id};
    int moment_c_id[] = {0, 0};
    int n_fields = 2;
    cs_time_moment_define_by_field_ids("UU_mean",
                                        n_fields,
                                        moment_f_id,
                                        moment_c_id,
                                        CS_TIME_MOMENT_MEAN,
                                        10000,/* nt_start */
                                        -1,    /* t_start */
                                        CS_TIME_MOMENT_RESTART_AUTO,
                                        NULL);
  }
  {
    /* Moment <u v>. */
    int moment_f_id[] = {CS_F_(u)->id, CS_F_(u)->id};
    int moment_c_id[] = {0, 1};
    int n_fields = 2;
    cs_time_moment_define_by_field_ids("UV_mean",
                                        n_fields,
                                        moment_f_id,
                                        moment_c_id,
                                        CS_TIME_MOMENT_MEAN,
                                        10000,/* nt_start */
                                        -1,    /* t_start */
                                        CS_TIME_MOMENT_RESTART_AUTO,
                                        NULL);
  }
```

5.3.3 cs_user_boundary_conditions.f90

In this subroutine the boundary conditions are defined. The user variables `ifac, ielt, nlelt` have to be declared, after `INSERT_VARIABLE_DEFINITIONS_HERE`

```
! INSERT_VARIABLE_DEFINITIONS_HERE
integer ifac, ielt, nlelt
```

Since periodicity is used in all three directions only boundary conditions for the tube walls have to be defined. Wall faces are labelled by two keywords, `CYLIN` and

CYLIN_OUT. In order to get a group of boundary faces, *Code_Saturne* uses the function getfbr called as follow:

```
 call getfbr("CYLIN␣or␣CYLIN_OUT", nlelt, lstelt)
```

The function takes as argument a string with the definitions of the faces to be grouped and returns an array lstelt containing the face numbers. The total number of faces found with the prescribed characteristics is nlelt.

Once the array with the faces corresponding to the walls is filled, it is necessary to do a loop and assign the type of boundary conditions. This is done by first obtaining the face number ifac, which is stored in the lstelt array, i.e. ifac = lstelt(ilelt). Then, set the array is set to itypfb(ifac) = iparoi [2]. This will automatically set standard wall boundaries to all the variables. The piece of code to be added to set up the boundary conditions should be written after this line: INSERT_MAIN_CODE_HERE. It should look like this

```
! INSERT_MAIN_CODE_HERE
call getfbr("CYLIN␣or␣CYLIN_OUT", nlelt, lstelt)
!==========
do ilelt = 1, nlelt
  ifac = lstelt(ilelt)
  itypfb(ifac)   = iparoi
enddo
```

There are many examples on how to set other types of boundary conditions in the **EXAMPLES** directory.

5.3.4 cs_user_source_terms.f90

This subroutine is used to drive the flow, adding a constant pressure gradient in the momentum equation. The test at **line 202** should be changed from if (.true.) to if (.false.), before changing the values of ckp=0.d0 and qdm=800.d0. The final code should look like:

```
 if (.false.) return
 ! ---------------------
ckp  = 0.d0
qdm  = 800.d0

do iel = 1, ncel
   crvimp(1,1,iel) = - volume(iel)*cpro_rom(iel)*ckp
enddo

do iel = 1, ncel
   crvexp(1,iel) =   volume(iel)*qdm
enddo
```

This will add a source term to the explicit part of the momentum equation.
Note Several subroutines are available in cs_user_source_terms.f90 to add source terms in the transport equations. In the present case, modifications must be done in the subroutine ustsnv, since it controls the source terms for the momentum equations solved with the coupled velocity solver.

[2] "paroi" means wall in french

5.3.5 cs_user_extra_operations.f90

In this subroutine lift and drag coefficients are computed at each time step.

- The local variables are declared first

```
! INSERT_VARIABLE_DEFINITIONS_HERE
integer          iel, ii, ifac
integer          ilelt  , nlelt
integer, allocatable, dimension(:) :: lstelt
double precision cyl_area, radius, dia, depth
double precision xcof(3), torque, tcof
double precision xfor(3)
double precision, dimension(:,:), pointer :: bfprp_for
```

- Then, memory is allocated for the array used for the selection of the faces and the values of the stresses are retrieved using the `field_get_val_v` function:

```
! INSERT_ADDITIONAL_INITIALIZATION_CODE_HERE
allocate(lstelt(max(ncel,nfac,nfabor)))
if (ineedf.eq.1) call field_get_val_v(iforbr, bfprp_for)
```

- The next step is to create a test on `ineedf` to make sure the subroutine is only used if the surface stresses have been calculated (see section 5.3.1).

- Local arrays and variables are initialised. The first part of the MAIN CODE should look like:

```
  ! INSERT_MAIN_CODE_HERE
if (ineedf.eq.1) then
  do ii = 1, ndim
    xfor(ii) = 0.d0
    torque   = 0.d0
    xcof(ii) = 0.d0
    tcof     = 0.d0
  enddo
  dia    = 2.17d-2
  radius = dia/2.d0
  depth  = dia
```

- The `getfbr` function is used to obtain the list of faces that belong to the cylinder located in the centre of the domain. The faces have been labelled as `CYLIN`, as

```
  call getfbr("CYLIN", nlelt, lstelt)
  !==========
```

- The next step is to loop on all the boundary faces labelled by `CYLIN`, which are stored in the array `lstelt` of size `nlelt`. The forces calculated by the code are stored in an 2D array called `bfprp_for`. The first dimension of the array is the direction (1 stands for X, 2 for Y and 3 for Z). The second dimension is the face index. The sum over the faces is stored in a local array called `xfor` of size 3 (one for each direction).

- The torque is also computed by multiplying the Y component of the force by the X component of the centre of face, and subtracting the product of the X component of the force and the Y component of the centre of the face. This is because the centre of the cylinder is at (0,0) leading to $T = F_x\Delta y - F_y\Delta_x$

```
do ilelt = 1, nlelt
  ifac = lstelt(ilelt)
  do ii = 1, ndim
    xfor(ii) = xfor(ii) +  bfprp_for(ii, ifac)
  enddo
  torque = xfor(2)*cdgfbo(1,ifac) - xfor(1)*cdgfbo(2,ifac)
enddo
```

- So far, only sums per processors (local sums) have been computed. Since the simulation is run in parallel, to obtain global sums, it is necessary to call the corresponding subroutines for communication between processors.

```
if (irangp.ge.0) then
  call parrsm(ndim,xfor)
  call parsom(torque)
endif
```

  The integer `irangp` is -1 if the calculation is serial and the rank number, if it is parallel.

- Lift and drags coefficients are obtained by normalising the force as $C = F/(\frac{1}{2}\rho U^2 A)$:

```
cyl_area = depth*dia
do ii=1,ndim
   xcof(ii) = xfor(ii) /(0.5d0*ro0*uref**2*cyl_area)
enddo
tcof = torque/(0.5d0*ro0*uref**2*cyl_area*radius)
```

- Coefficients are written into a file. User files can be written with pointers specifically left unused in the code for this type of tasks. The pointers available for the user files are stored in `impusr()`.
  - A file called 'coefficients.dat' is opened
  - Its headers are written at the first iteration only. This can be achieved by a test on `ntcabs` (current time step) and `ntpabs` (previous time step). The rest of the subroutine should look like:

```
if (irangp.eq.0) then
   if (ntcabs.eq.ntpabs+1) then
     open(unit=impusr(1), file="coefficients.dat")
     write(impusr(1),*) "#File␣with␣Cl␣and␣Cd"
     write(impusr(1),*) "#R", radius
     write(impusr(1),*) "#Uinf␣", uref
     write(impusr(1),*) "#rho␣", ro0
     write(impusr(1),*) "#Front␣area␣", cyl_area
     write(impusr(1),*) "#Structure␣of␣the␣results:"
     write(impusr(1),*) "#Col␣1:␣Time"
     write(impusr(1),*) "#Col␣2-3:␣Force␣in␣x␣and␣y"
```

```
      write(impusr(1),*) "#Col␣4:␣Torque"
      write(impusr(1),*) "#Col␣5:␣Cd"
      write(impusr(1),*) "#Col␣6:␣Cl"
      write(impusr(1),*) "#Col␣7:␣Ct"
    endif
    write(impusr(1),1001)ttcabs,xfor(1),xfor(2), &
                         torque,xcof(1),xcof(2),tcof
    if (ntcabs.eq.ntmabs) close(impusr(1))
endif
endif
1001 format(7(e18.9,1x))
```

Once all subroutines have been modified the code should be compiled as:

```
[bash:$] code_saturne compile
```

in order to check that no compilation errors occur.

5.4 Finalisation of the case setup - Extra files

In order to be able to run the jobs on the Blue Gene, extra files need to be dealt with.

The partitioning stage has not been presented here, but partitions generated by METIS-5.0.2 are available for $1,204$ and $2,048$ subdomains in the directory **partition_input**, which is available in the shared directory and has to be copied to **DATA**:

```
[bash:$] cp -r ~/../shared/Tutorials/03_TB_LES_SRC/CASEFILES/MESHES/ \
partition_input TUBE_BUNDLE_LES/LES_SRC/DATA/.
```

5.4.1 cs_users_scripts.py

Some information is missing for the code to run, namely the location of the `mesh_output` file, the **partition_input** and the **checkpoint** directories. To take them into account,

- Go to the **DATA** directory
- Copy here the file `cs_user_scripts.py` available in the **REFERENCE** directory. The **REFERENCE** directory is also in the **DATA** directory.
- Open the file and go to **line 138** to change the *domain.mesh_input* value from `None` to [3]:

```
    if domain.param == None:
      domain.mesh_input = "DATA/mesh_output"
      domain.partition_input = None
      domain.restart_input = None
```

- Go to **line 139** to change the *domain.partition_input* value from `None` to

[3] Note that in python, indentation is mandatory.

```
      if domain.param == None:
        domain.mesh_input = "DATA/mesh_output"
        domain.partition_input = "DATA/partition_input"
        domain.restart_input = None
```

- Go to **line 140** to change the *domain.restart_input* value from `None` to

```
      if domain.param == None:
        domain.mesh_input = "DATA/mesh_output"
        domain.partition_input = "DATA/partition_input"
        domain.restart_input = "DATA/checkpoint"
```

5.4.2 Queuing submission script - `runcase`

Blue Joule is composed of 1 login node (also called frontend) and 6,144 compute nodes. Each of the compute node has 16 processor cores.

The remote machine is set up to use a queuing system. These types of systems are usually found on multi-users machines and allow for a better resource allocation. In order to run a job it is required to submit a set of instructions to the queueing system. From these instructions, the system tries to match the requirements to the available resources. The file `runcase` under the **`SCRIPTS`** directory contains instructions for the queuing system (LoadLeveler) and it should look like this:

```
#!/bin/bash
#-----------------------------------------------------------------
#
# Batch options for IBM LoadLeveler (example: BlueGene/Q)
# ================================
#
# To obtain all available options, run the LoadLeveler GUI,
# "xloadl", choose the "build job" option; select options,
# then select "save". Options and their matching syntax
# may then be inferred from the saved file.
#
# @ bg_size           = 128
# @ class             = qres01
# @ job_name          = tube_bundle_les
# @ job_type          = bluegene
# @ step_name         = runcase
# @ environment       = COPY_ALL
# @ output            = $(job_name).$(jobid).out
# @ error             = $(job_name).$(jobid).err
# @ wall_clock_limit  = 12:00:00
# @ notification      = complete
# @ executable        = runcase
# @ queue
#-------------------------------------------------------------------------------
export LOADL_RANKS_PER_NODE=16
# Change to submission directory
if test -n "$LOADL_STEP_INITDIR" ; then cd $LOADL_STEP_INITDIR ; fi
```

```
module purge
module load ibmmpi

# Ensure the correct command is found:
export PATH=/gpfs/packages/ibm/code_saturne/4.0.2/bin:$PATH

# Run command:
\code_saturne run
```

Make sure that the following variables are set:

- bg_size is set to 128 (number of requested nodes)
- the variable LOADL_RANKS_PER_NODE is set to 16 (number of ranks on each node)

Once the file is modified, the job can be submitted to the queuing system by typing:

```
[bash:$] llsubmit runcase
```

If everything goes fine, information on the current job is available from the `llq` command.

6 Results

Once the calculation is finished, the results have to be copied back to the local machine in order to visualise them. `scp` is used for this purpose but it only works from the **local machine** to the remote machine.

- Open a new terminal on the local machine
- Create a new directory called **RESULTS_FROM_BGQ**
- Go to this new directory as

```
[bash:$] cd RESULTS_FROM_BGQ
```

- Create a new directory called **LES_SRC**
- Go to this new directory as

```
[bash:$] cd LES_SRC
```

- In the case of copying the monitoring and the postprocessing directories (not forgetting to add the `-r` option), type:

```
[bash:$] scp -r bglogin2:PATH_FROM_RESU/monitoring .
[bash:$] scp -r bglogin2:PATH_FROM_RESU/postprocessing .
```

Vectors, contours and streamlines might be visualised by *ParaView* (see figure 4 for an example).

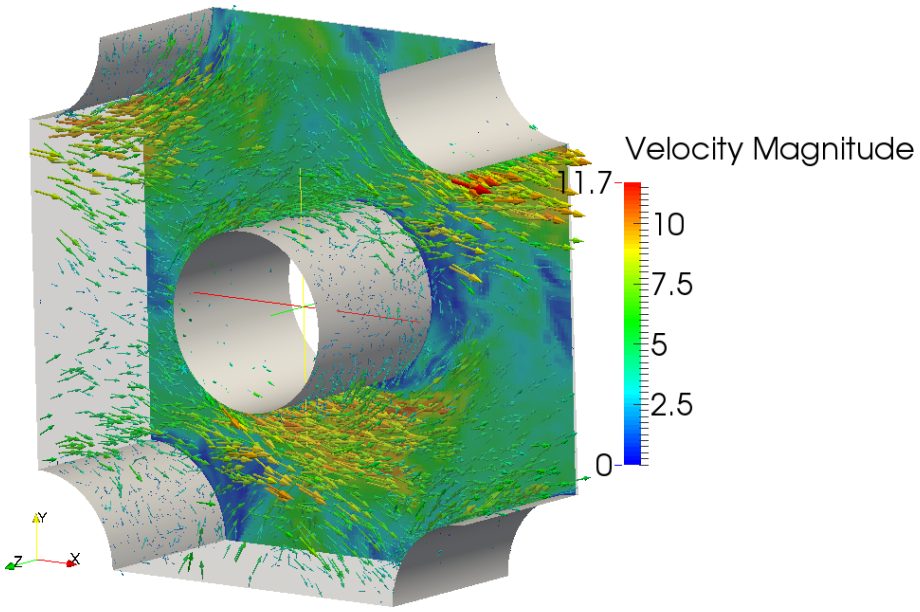

Figure 4: Instantaneous velocity field

References

[1] SMAGORINSKY, J. 1963 General circulation experiments with the primitive equations: I the basic equations. *Monthly Weather Review* **91**, 99–164.