EDF R&D

Fluid Dynamics, Power Generation and Environment Department
Single Phase Thermal-Hydraulics Group

6, quai Watier
F-78401 Chatou Cedex

Tel: 33 1 30 87 75 40
Fax: 33 1 30 87 79 16

SEPTEMBER 2019

Code_Saturne documentation

Code_Saturne version 6.0.0 installation guide

contact: saturne-support@edf.fr

TABLE OF CONTENTS

1 *Code_Saturne* Automated or manual installation

Code_Saturne may be installed either directly through its GNU Autotools based scripts (the traditional `configure`, `make`, `make install`) sequence), or using an automated installer (`install_saturne.py`), which generates an initial `setup` file when run a first time, and builds and installs *Code_Saturne* and some optional libraries based on the edited `setup` when run a second time. The use of this automated script is briefly explained in the top-level `README` file of the *Code_Saturne* sources, as well as in the comments of `setup` file. It is not detailed further in this documentation, which details the manual installation, allowing a finer control over installation options.

Note that when the automatic installer is run, it generates a build directory, in which the build may be modified (re-running `configure`, possibly adapting the command logged at the beginning of the `config.status` file) and resumed.

2 Installation basics

The installation scripts of *Code_Saturne* are based on the GNU Autotools, (Autoconf, Automake, and Libtool), so it should be familiar for many administrators. A few remarks are given here:

- As with most software with modern build systems, it is recommended to build the code in a separate directory from the sources. This allows multiple builds (for example production and debug), and is considered good practice. Building directly in the source tree is not regularly tested, and is not guaranteed to work, in addition to "polluting" the source directory with build files.
- By default, optional libraries which may be used by *Code_Saturne* are enabled automatically if detected in default search paths (i.e. `/usr/` and `/usr/local`. To find libraries associated with a package installed in an alternate path, a `--with-<package>=...` option to the `configure` script must given. To disable the use of a library which would be detected automatically, a matching `--without-<package>` option must be passed to `configure` instead.
- Most third-party libraries usable by *Code_Saturne* are considered optional, and are simply not used if not detected, but the libraries needed by the GUI are considered mandatory, unless the `--disable-gui` or `--disable-frontend` option is explicitly used.

When the prerequisites are available, and a build directory created, building and installing *Code_Saturne* may be as simple as running:

```
$ ../../code_saturne-6.0.0/configure
$ make
$ make install
```

The following chapters give more details on *Code_Saturne*'s recommended third-party libraries, configuration recommendations, troubleshooting, and post-installation options.

3 Compilers and interpreters

For a minimal build of *Code_Saturne* on a Linux or Posix system, the requirements are:

- A C compiler, conforming at least to the C99 standard.
- A Fortran compiler, conforming at least to the Fortran 95 standard and supporting the ISO_C_BINDING Fortran 2003 module.
- A Python interpreter, with Python version 2.6 or above.

For parallel runs, an MPI library is also necessary (MPI-2 or MPI-3 conforming). To build and use the GUI, PyQt 4 or 5 (which in turn requires Qt 4 or 5 and SIP) are required. Other libraries may be used for additional mesh format options, as well as to improve performance. A list of those libraries and their role is given in §5.2.

For some external libraries, such as Catalyst (see 5.2), a C++ compiler is also required.

The SALOME platform V9 and above requires Python 3, older versions Python 2, and a matching version should be used when building with SALOME support.

In practice, the code is known to build and function properly at least with the GNU compilers 4.4 and above (up to 8.x at this date), Intel compilers 11 and above (up to 18 at this date), IBM XL Fortran 14 and C 12 compilers, and Clang (tested with 3.7 or above).

Note also that while *Code_Saturne* makes heavy use of Python, this is for scripts and for the GUI only; The solver only uses compiled code, so we could for example use a 32-bit version of Python with 64-bit *Code_Saturne* libraries and executables. Also, the version of Python used by ParaView/Catalyst may be independent from the one used for building *Code_Saturne*.

4 Loading an environment

If installing and running *Code_Saturne* requires sourcing a given environment or loading environement modules (see §5.1), the `--with-shell-env` option allows defining the path for a file to source, or if no path is given, loading default modules.

By default, the main `code_saturne` command is a Python script. When sourcing an environment, a launcher shell script is run first, loads the required environment, then calls Python with the `code_saturne.py` script.

5 Third-Party libraries

For a minimal build of *Code_Saturne*, a Linux or Posix system with C and Fortran compilers (C99 and Fortran 95 with Fortran 2003 ISO C bindings conforming respectively), a Python (2.6 or later) interpreter and a `make` tool should be sufficient. For parallel runs, an MPI library is also necessary (MPI-2 or MPI-3 conforming). To build and use the GUI, Qt 4 or 5 with PyQt 4 or 5 Python bindings (which in turn requires SIP) are required. Other libraries may be used for additional mesh format options, as well as to improve performance. A list of those libraries and their role is given in §5.2.

5.1 Installing third-party libraries for *Code_Saturne*

Third-Party libraries usable with *Code_Saturne* may be installed in several ways:

- On many Linux systems, most of libraries listed in §5.2 are available through the distribution's package manager.[1] This requires administrator privileges, but is by far the easiest way to install third-party libraries for *Code_Saturne*.

[1] On Mac OS X systems, package managers such as Fink or MacPorts also provide package management, even though the base system does not.

Note that distributions usually split libraries or tools into runtime and development packages, and that although some packages are installed by default on many systems, this is generally not the case for the associated development headers. Development packages usually have the same name as the matching runtime package, with a `-dev` postfix added. Names might also differ slightly. For example, on a Debian system, the main package for Open MPI is `openmpi-bin`, but `libopenmpi-dev` must also be installed for the *Code_Saturne* build to be able to use the former.

- On some systems, especially compute clusters, Environment Modules allow the administrators to provide multiple versions of many scientific libraries, as well us compilers or MPI libraries, using the `module` command. More details on Environment Modules may be found at `http://modules.sourceforge.net` or `https://github.com/TACC/Lmod`. When being configured and installed *Code_Saturne* checks for modules loaded with the `module` command, and records the list of loaded modules. Whenever running that build of *Code_Saturne*, the modules detected at installation time will be used, rather than those defined by default in the user's environment. This allows using versions of *Code_Saturne* built with different modules safely and easily, even if the user may be experimenting with other modules for various purposes.

- If not otherwise available, third-party software may be compiled an installed by an administrator or a user. An administrator will choose where software may be installed, but for a user without administrator privileges or write access to `usr/local`, installation to a user account is often the only option. None of the third-party libraries usable by *Code_Saturne* require administrator privileges, so they may all be installed normally in a user account, provided the user has sufficient expertise to install them. This is usually not complicated (provided one reads the installation instructions, and is prepared to read error messages if something goes wrong), but even for an experienced user or administrator, compiling and installing 5 or 6 libraries as a prerequisite significantly increases the effort required to install *Code_Saturne*.

  Even though it is more time-consuming, compiling and installing third-party software may be necessary when no matching packages or Environment Modules are available, or when a more recent version or a build with different options is desired.

- When *Code_Saturne* is configured to use the SALOME platform, some libraries inclued in that platform may be used directly; this is described in §7.8.

5.2 List of third-party libraries usable by *Code_Saturne*

The list of third-party software usable with *Code_Saturne* is provided here:

- PyQt version 4 or 5 is required by the *Code_Saturne* GUI. PyQt in turn requires Qt (4 or 5), Python, and SIP. Without this library, the GUI may not be built, although XML files generated with another install of *Code_Saturne* may be used.

  If desired, Using PySide instead of PyQt4/SIP should require a relatively small porting effort, as most of the preparatory work has been done. The development team should be contacted in this case.

- HDF5 is necessary for MED, and may also be used by CGNS.

- CGNSlib is necessary to read or write mesh and visualization files using the CGNS format, available as an export format with many third-party meshing tools. CGNS version 3.1 or above is required.

- MED is necessary to read or write mesh and visualization files using the MED format, mainly used by the SALOME platform (`www.salome-platform.org`).

- libCCMIO is necessary to read or write mesh and visualization files generated or readable by **STAR-CCM+** using its native format.

- SCOTCH or PT-SCOTCH may be used to optimize mesh partitioning. Depending on the mesh, parallel computations with meshes partitioned with these libraries may be from 10% to 50% faster than using the built-in space-filling curve based partitioning.

  As SCOTCH and PT-SCOTCH use symbols with the same names, only one of the 2 may be used. If both are detected, PT-SCOTCH is used. Versions 6.0 and above are supported.

- METIS or PARMETIS are alternative mesh partitioning libraries. These libraries have a separate source tree, but some of their functions have identical names, so only one of the 2 may be used. If both are available, PARMETIS will be used. Partitioning quality is similar to that obtained with SCOTCH or PT-SCOTCH.

  Though broadly available, the PARMETIS license is quite restrictive, so PT-SCOTCH may be preferred (*Code_Saturne* may be built with both METIS and SCOTCH libraries). Also, the METIS license was changed in March 2013 to the Apache 2 license, so it would not be surprising for future PARMETIS versions to follow. METIS 5.0 or above and PARMETIS 4.0 or above are supported.

- Catalyst (`http://www.paraview.org/in-situ/`) or full ParaView may be used for co-visualization or in-situ visualization. This requires ParaView 4.2 or above.

- `eos-1.2` may be used for thermodynamic properties of fluids. it is not currently free, so usually available only to users at EDF, CEA, or organisms participating in projects with those entities.

- **freesteam** (`http://freesteam.sourceforge.net`) is a free software thermodynamic properties library, implementing the IAPWS-IF97 steam tables, from the International Association for the Properties of Water and Steam (IAPWS). Version 2.0 or above may be used.

- CoolProp (`http://www.coolprop.org`) is a quite recent library open source library, which provides pure and pseudo-pure fluid equations of state and transport properties for 114 components (as of version 5.1), mixture properties using high-accuracy Helmholtz energy formulations (or cubic EOS), and correlations of properties of incompressible fluids and brines. Its validation is based at least in part on comparisons with REFPROP.

- BLAS (Basic Linear Algebra Subroutines) may be used by the `cs_blas_test` unit test to compare the cost of operations such as vector sums and dot products with those provided by the code and compiler. If no third-party BLAS is provided, *Code_Saturne* reverts to its own implementation of BLAS routines, so no functionality is lost here. Optimized BLAS libraries such as Atlas, MKL, ESSL, or ACML may be very fast for BLAS3 (dense matrix/matrix operations), but the advantage is usually much less significant for BLAS 1 (vector/vector) operations, which are almost the only ones *Code_Saturne* has the opportunity of using. *Code_Saturne* uses its own dot product implementation (using a superblock algorithm, for better precision), and $y \leftarrow ax + y$ operations, so external BLAS1 are not used for computation, but only for unit testing (so as to be able to compare performance of built-in BLAS with external BLAS). The Intel MKL BLAS may also be used for matrix-vector products, so it is linked with the solver when available, but this is also currently only used in unit benchmark mode. Note that in some cases, threaded BLAS routines might oversubscribe processor cores in some MPI calculations, depending on the way both *Code_Saturne* and the BLAS were configured and interact, and this can actually lead to lower performance. Use of BLAS libraries is thus useful as a unit benchmarking feature, but has no influence on full calculations.

- PETSc (Portable, Extensible Toolkit for Scientific Computation, `http://www.mcs.anl.gov/petsc/`) consists of a variety of libraries, which may be used by *Code_Saturne* for the resolution of linear equation systems. In addition to providing many solver options, it may be used as a bridge to other major solver libraries.

For developers, the GNU Autotools (Autoconf, Automake, Libtool) as well as gettext will be necessary. To build the documentation, pdfLaTeX and `fig2dev` (part of TransFig) will be necessary.

5.3 Notes on some third-party tools and libraries

5.3.1 Python and PyQt

The GUI is written in PyQt (Python bindings for Qt), so Qt (version 4 or 5) and the matching Python bindings must be available. On most modern Linux distributions, this is available through the package manager, which is by far the preferred solution.

On systems on which both PyQt4 and Pyqt5 are available, PyQt4 will be selected by default, but the selection may be forced by defining `QT_SELECT=4` or `QT_SELECT=5`.

When running on a system which does not provide these libraries, there are several alternatives:

- build *Code_Saturne* without the GUI. XML files produced with the GUI are still usable, so if an install of *Code_Saturne* with the GUI is available on an other machine, the XML files may be copied on the current machine. This is certainly not an optimal solution, but in the case where users have a mix of desktop or virtual machines with modern Linux distributions and PyQt installed, and a compute cluster with an older system, this may avoid requiring a build of Qt and PyQt on the cluster if users find this too daunting.

- Install a local Python interpreter, and add Qt5 bindings to this interpreter.

  Python (`http://www.python.org`) and Qt (`https://www.qt.io`) must be downloaded and installed first, in any order. The installation instructions of both of these tools are quite clear, and though the installation of these large packages (especially Qt) may be a lengthy process in terms of compilation time, but is well automated and usually devoid of nasty surprises.

  Once Python is installed, the SIP bindings generator (`http://riverbankcomputing.co.uk/software/sip/intro`) must also be installed. This is a small package, and configuring it simply requires running `python configure.py` in its source tree, using the Python interpreter just installed.

  Finally, the PyQt bindings (`http://riverbankcomputing.co.uk/software/pyqt/intro`) may be installed, in a manner similar to SIP.

  When this is finished, the local Python interpreter contains the PyQt bindings, and may be used by *Code_Saturne*'s `configure` script by passing `PYTHON=<path_to_python_executable`.

- add Python Qt bindings as a Python extension module for an existing Python installation. This is a more elegant solution than the previous one, and avoids requiring rebuilding Python, but if the user does not have administrator privileges, the extensions will be placed in a directory that is not on the default Python extension search path, and that must be added to the `PYTHONPATH` environment variable. This works fine, but for all users using this build of *Code_Saturne*, the `PYTHONPATH` environment variable will need to be set.[2]

  The process is similar to the previous one, but SIP and PyQt installation requires a few additional configuration options in this case. See the SIP and PyQt reference guides for detailed instructions, especially the *Building a Private Copy of the SIP Module* section of the SIP guide.

5.3.2 Scotch and PT-Scotch

Note that both Scotch and PT-Scotch may be built from the same source tree, and installed together with no name conflicts.

For better performance, PT-Scotch may be built to use threads with concurrent MPI calls. This requires initializing MPI with `MPI_Init_thread` with `MPI_THREAD_MULTIPLE` (instead of the more restrictive `MPI_THREAD_SERIALIZED`, `MPI_THREAD_FUNNELED`, or `MPI_THREAD_SINGLE`, or simply using `MPI_Init`). As *Code_Saturne* does not support thread models in which different threads may call MPI

[2] In the future, the *Code_Saturne* installation scripts could check the `PYTHONPATH` variable and save its state in the build so as to ensure all the requisite directories are searched for.

functions simultaneously, and the use of `MPI_THREAD_MULTIPLE` may carry a performance penalty, we prefer to sacrifice some of PT-Scotch's performance by requiring that it be compiled without the `-DSCOTCH_PTHREAD` flag. This is not detected at compilation time, but with recent MPI libraries, PT-Scotch will complain at run time if it notices that the MPI thread safety level in insufficient.

Detailed build instructions, including troubleshooting instructions, are given in the source tree's `INSTALL.txt` file. In case of trouble, note especially the explanation relative to the `dummysizes` executable, which is run to determine the sizes of structures. On machines with different front-end and compute node architectures, it may be necessary to start the build process, let it fail, run this executable manually using `mpirun`, then pursue the build process.

5.3.3 MED

The Autotools installation of MED is simple on most machines, but a few remarks may be useful for specific cases.

Note that up to MED 3.3.1, HDF5 1.8 is required, while MED 4.x uses HDF5 1.10.

MED has a C API, is written in a mix of C and C++ code, and provides both a C (`libmedC`) and an Fortran API (`libmed`). Both libraries are always built, so a Fortran compiler is required, but *Code_Saturne* only links using the C API, so using a different Fortran compiler to build MED and *Code_Saturne* is possible.

MED does require a C++ runtime library, which is usually transparent when shared libraries are used. When built with static libraries only, this is not sufficient, so when testing for a MED library, the *Code_Saturne* `configure` script also tries linking with a C++ compiler if linking with a C compiler fails. This must be the same compiler that was used for MED, to ensure the runtime matches. The choice of this C++ compiler may be defined passing the standard `CXX` variable to `configure`.

Also, when building MED in a cross-compiling situation, `--med-int=int` or `--med-int=int64_t` (depending on whether 32 or 64 bit ids should be used) should be passed to its `configure` script to avoid a run-time test.

5.3.4 libCCMIO

Different versions of this library may use different build systems, and use different names for library directories, so using both the `--with-ccm=` or `--with-ccm-include=` and `--with-ccm-lib=` options to `configure` is usually necessary. Also, the include directory should be the toplevel library, as header files are searched under a `libccmio` subdirectory[3]

A libCCMIO distribution usually contains precompiled binaries, but recompiling the library is recommended. Note that at least for version 2.06.023, the build will fail building dump utilities, due to the `-l adf` option being placed too early in the link command. To avoid this, add `LDLIBS=-ladf` to the makefile command, for example:

```
make -f Makefile.linux SHARED=1 LDLIBS=-ladf
```

(`SHARED=1` and `DEBUG=1` may be used to force shared library or debug builds respectively).

Finally, if using libCCMIO 2.6.1, remove the `libcgns*` files from the libCCMIO libraries directory if also building *Code_Saturne* with CGNS support, as those libraries are not required for CCMIO, and are are an obsolete version of CGNS, which may interfere with the version used by *Code_Saturne*.

Note that libCCMIO uses a modified version of CGNS's ADF library, which may not be compatible with that of CGNS. When building with shared libraries, the reader for libCCMIO uses a plugin architecture

[3] this is made necessary by libCCMIO version 2.6.1, in which this is hard-coded in headers including other headers. In more recent versions such as 2.06.023, this is not the case anymore, and an `include` subdirectory is present, but it does not contain the `libccmioversion.h` file, which is found only under the `libccmio` subdirectory, and is required by *Code_Saturne* to handle differences between versions, so that source directory is preferred to the installation `include`.

to load the library dynamically. For a static build with both libCCMIO and CGNS support, reading ADF-based CGNS files may fail. To work around this issue, CGNS files may be converted to HDF5 using the `adf2hdf` utility (from the CGNS tools). By default, CGNS post-processing output files use HDF5, so this issue is rarer on output.

5.3.5 freesteam

This library's build instructions mention bindings with ascend, but those are not necessary in the context of *Code_Saturne*, so building without them is simplest. Its build system is based on scons, and builds on relatively recent systems with Python 2.7 should be straightforward.

With Python versions lower than 2.6, the command-line arguments allowing to choose the installation prefix (so as to place it in a user directory) are ignored, and its `SConstruct` file is not complete enough to allow setting flags for linking with an alternative, user-installed Python library outside the default linker search path. In this case, editing the `SConstruct` file to change the default paths is an ugly, but simple solution.

5.3.6 CoolProp

This library's build system is based on CMake, and building it is straightforward, though some versions seem to have build issues (the 5.1.0 release is missing a file, while 5.1.1 release builds fine). CoolProp uses submodules which are downloaded using `git clone https://github.com/CoolProp/CoolProp.git --recursive`, but may be missing when downloading a zip file.

Its user documentation is good, but its installation documentation is poor, so recommendations are provided here

To download and prepare CoolProp for build, using an out-of-tree build (so as to avoid polluting the source tree with cache files), the following commands are recommended:

```
$ git clone https://github.com/CoolProp/CoolProp.git --recursive
$ cd CoolProp
$ git checkout release
$ cd ..
$ mkdir CoolProp_build
$ cd CoolProp_build
```

Then configure, build, and install, run:

```
$ cmake \
-DCOOLPROP_INSTALL_PREFIX=$INSTALL_PATH/arch/$machine_name \
-DCOOLPROP_SHARED_LIBRARY=ON \
$COOLPROP_SRC_PATH
```

Followed by:

```
$ make
$ make install
$ make clean
```

CoolProp's installer only installs one C wrapper header, not the C++ headers required for lower-level access, so the following commands must also be run:

```
$ cp -rp $COOLPROP_SRC_PATH/include $INSTALL_PATH/arch/$machine_name
$ rm -f $INSTALL_PATH/arch/$machine_name/CoolPropLib.h
```

Alternatively, to copy less files and avoid changing the structure provided by CoolProp:

```
$ cp -r $COOLPROP_SRC_PATH/include $INSTALL_PATH/arch/$machine_name
\
$ cp -r $COOLPROP_SRC_PATH/externals/fmtlib/fmt \
$INSTALL_PATH/arch/$machine_name/include/
```

To install CoolProp's Python bindings (used by the GUI when available), the straigthforward method is to go into the CoolProp source directory, into the wrappers/Python subdirectory, then run:

```
$ export PYTHONPATH=COOLPROP_INSTALL_PREFIX/lib/$python_version/site
packages :PYTHONPATH $ python setup.py install --prefix=$COOLPROP_INSTALL_PREFIX
```

Although this is not really an out-of-tree build, the Python setup also cleans the directory.

5.3.7 Paraview or Catalyst

By default, this library is built with a GUI, but it may also be be built using OSMesa for offscreen rendering. The build documentation on the ParaView website and Wiki details this. For use with *Code_Saturne*, the recommended solution is to build or use a standard ParaView build for interactive visualization, and to use its CatalystScriptGeneratorPlugin to generate Python co-processing scripts. A second build, using OSMesa, may be used for in-situ visualization. This is the Version *Code_Saturne* will be linked to. A recommended cmake command for this build contains:

```
$ cmake \
-DCMAKE_INSTALL_PREFIX=$INSTALL_PATH/arch/$machine_name_osmesa \
-DPARAVIEW_BUILD_QT_GUI=OFF \
-DPARAVIEW_USE_MPI=ON \
-DPARAVIEW_ENABLE_PYTHON=ON \
-DPARAVIEW_INSTALL_DEVELOPMENT_FILES=ON \
-DVTK_USE_X=OFF \
-DOPENGL_INCLUDE_DIR=IGNORE \
-DOPENGL_xmesa_INCLUDE_DIR=IGNORE \
-DOPENGL_gl_LIBRARY=IGNORE \
-DOSMESA_INCLUDE_DIR=$MESA_INSTALL_PREFIX/include \
-DOSMESA_LIBRARY=$MESA_INSTALL_PREFIX/lib/libOSMesa.so \
-DVTK_OPENGL_HAS_OSMESA=ON \
-DVTK_USE_OFFSCREEN=OFF \
$PARAVIEW_SRC_PATH
```

More info may also be found on the ParaView Wiki: (http://www.paraview.org/Wiki/ParaView/ParaView_And_Mesa_3D).

Catalyst editions (http://www.paraview.org/Wiki/ParaView/Catalyst/BuildCatalyst) may be used instead of a full ParaView build, but some coprocessing scripts may not work depending on what is included in the editions, so this is recommended only for advanced users.

On some systems, loading the Catalyst module as a plug-in (which is the default) seems to interfere with the detection of required OpenGL2 features or extensions required by ParaView 5.2 an above. In this case, Catalyst support may be linked in the standard manner by using the `--disable-catalyst-as-plugin` configuration option. A less extreme option is to use the `--enable-dlopen-rtld-global` option, which changes the system options with which libraries are loaded (possibly impacting all plugins). This seems to be sufficient with OSMesa 17.x versions. Using the `DL_PRELOAD` environment variable at runtime to preload the OSMesa library also avoids the issue.

6 Preparing for build

If the code was obtained as an archive, it must be unpacked:

`tar xvzf saturne.tar.gz`

If for example you unpacked the directory in a directory named `/home/user/Code_Saturne`, you will now have a directory named `/home/user/Code_Saturne/saturne`.

It is recommended to build the code in a separate directory from the source. This also allows multiple builds, for example, building both an optimized and a debugging version. In this case, choose a consistent naming scheme, using an additional level of sub-directories, for example:

```
$ mkdir saturne_build
$ cd saturne_build
$ mkdir prod
$ cd prod
```

Some older system's `make` command may not support compilation in a directory different from the source directory (`VPATH` support). In this case, installing and using the GNU `gmake` tool instead of the native `make` is recommended.

6.1 Source trees obtained through a source code repository

For developers obtaining the code was obtained through a Git repository, an additional step is required:

```
$ cd saturne
$ ./sbin/bootstrap
$ cd ..
```

In this case, additional tools need to be available:

- GNU Autotools: Autoconf, Automake, Libtool (2.2 or 2.4), and Gettext.
- Bison (or Yacc) and Flex (or Lex)
- PdfLaTeX and TransFig
- Doxygen (1.8.7 or more recent). The path to Doxygen can be specified during the configure phase with `configure DOXYGEN=PATH_TO_DOXYGEN`.

These tools are not necessary for builds from tarballs; they are called when building the tarball (using `make dist`), so as to reduce the number of prerequisites for regular users, while developers building the code from a repository can be expected to need a more complete development environment.

Also, to build and install the documentation when building the code from a repository instead of a tarball, the following stages are required:

```
$ make doc
$ make install-doc
```

7 Configuration

Code_Saturne uses a build system based on the GNU Autotools, which includes its own documentation.

To obtain the full list of available configuration options, run: `configure --help`.

Note that for all options starting with `--enable-`, there is a matching option with `--disable-`. Similarly, for every `--with-`, `--without-` is also possible.

Select configuration options, then run `configure`, for example:

```
$ /home/user/Code_Saturne/6.0.0/src/code_saturne-6.0.0/configure \
--prefix=/home/user/Code_Saturne/6.0.0/arch/prod \
--with-med=/home/user/opt/med-4.0 \
CC=/home/user/opt/mpich-3.2/bin/mpicc FC=gfortran
```

In the rest of this section, we will assume that we are in a build directory separate from sources, as described in §6. In different examples, we assume that third-party libraries used by *Code_Saturne* are either available as part of the base system (i.e. as packages in a Linux distribution), as Environment Modules, or are installed under a separate path.

7.1 Debug builds

It may be useful to install debug builds alongside production builds of *Code_Saturne*, especially when user subroutines are used and the risk of crashes due to user programming error is high. Running the code using a debug build is significantly slower, but more information may be available in the case of a crash, helping understand and fix the problem faster.

Here, having a consistent and practical naming scheme is useful. For a side-by-side debug build for the example above, we simply replace `prod` by `dbg` in the `--prefix` option, and add `--enable-debug` to the configure command:

```
$ cd ..
$ mkdir dbg
$ cd dbg
$ ../../code_saturne-6.0.0/configure \
--prefix=/home/user/Code_Saturne/6.0.0/arch/dbg \
--with-med=/home/user/opt/med-4.0 \
--enable-debug \
CC=/home/user/opt/mpich-3.2/bin/mpicc FC=gfortran
```

7.2 Shared or static builds

By default, on most architectures, *Code_Saturne* will be built with shared libraries. Shared libraries may be disabled (in which case static libraries are automatically enabled) by adding `--disable-shared` to the options passed to `configure`. On some systems, the build may default to static libraries instead.

It is possible to build both shared and static libraries by not adding `--disable-static` to the `configure` options, but the executables will be linked with the shared version of the libraries, so this is rarely useful (the build process is also slower in this case, as each file is compiled twice).

In some cases, a shared build may fail due to some dependencies on static-only libraries. In this case, `--disable-shared` will be necessary. Disabling shared libraries is also necessary to avoid issues with linking user functions on Mac OSX systems.

In any case, be careful if you switch from one option to the other: as linking will be done with shared libraries by default, a build with static libraries only will not completely overwrite a build using shared libraries, so uninstalling the previous build first is recommended.

7.3 Relocatable builds

By default, a build of *Code_Saturne* is not movable, as not only are library paths hard-coded using *rpath* type info, but the code's scripts also contain absolute paths.

To ensure a build is movable, pass the `--enable-relocatable` option to `configure`.

Movable builds assume a standard directory hierarchy, so when running `configure`, the `--prefix` option may be used, but fine tuning of installation directories using options such as `--bindir`, `--libdir`, or `--docdir` must not be used (these options are useful to install to strict directory hierarchies, such as when packaging the code for a Linux distribution, in which case making the build relocatable would be nonsense anyways, so this is not an issue. [4]

7.4 Compiler flags and environment variables

As usual when using an Autoconf-based `configure` script, some environment variables may be used. `configure --help` will provide the list of recognized variables. `CC` and `FC` allow selecting the C and Fortran compiler respectively (possibly using an MPI compiler wrapper).

Compiler options are usually defined automatically, based on detection of the compiler (and depending on whether `--enable-debug` was used). This is handled in a `config/cs_auto_flags.sh` and `libple/config/ple_auto_flags.sh` scripts. These files are sourced when running `configure`, so any modification to it will be effective as soon as `configure` is run. When installing on an exotic machine, or with a new compiler, adapting this file is useful (and providing feedback to the *Code_Saturne* development team will enable support of a broader range of compilers and systems in the future.

The usual `CPPFLAGS`, `CFLAGS`, `FCCFLAGS`, `LDFLAGS`, and `LIBS` environment variables may also be used, an flags provided by the user are appended to the automatic flags. To completely disable automatic setting of flags, the `--disable-auto-flags` option may be used.

7.5 MPI compiler wrappers

MPI environments generally provide compiler wrappers, usually with names similar to `mpicc` for C, `mpicxx` for C++, and `mpif90` for Fortran 90. Wrappers conforming to the MPI standard recommendations should provide a `-show` option, to show which flags are added to the compiler so as to enable MPI. Using wrappers is fine as long as several third-party tools do not provide their own wrappers, in which case either a priority must be established. For example, using HDF5's `h5pcc` compiler wrapper includes the options used by `mpicc` when building HDF5 with parallel IO, in addition to HDF5's own flags, so it could be used instead of `mpicc`. On the contrary, when using a serial build of HDF5 for a parallel build of *Code_Saturne*, the `h5cc` and `mpicc` wrappers contain different flags, so they are in conflict.

Also, some MPI compiler wrappers may include optimization options used to build MPI, which may be different from those we wish to use that were passed.

To avoid issues with MPI wrappers, it is possible to select an MPI library using the `--with-mpi` option to `configure`. For finer control, `--with-mpi-include` and `--with-mpi-lib` may be defined separately.

Still, this may not work in all cases, as a fixed list of libraries is tested for, so using MPI compiler wrappers is the simplest and safest solution. Simply use a `CC=mpicc` or similar option instead of `--with-mpi`.

There is no need to use an `FC=mpif90` or equivalent option: in *Code_Saturne*, MPI is never called directly from Fortran code, so Fortran MPI bindings are not necessary.

7.6 Environment Modules

As noted in §5.1, on systems providing Environment Modules with the `module` command, *Code_Saturne*'s `configure` script detects which modules are loaded and saves this list so that future

[4] In the special case of packaging the code, which may require both fine-grained control of the installation directories and the possibility to support options such as `dpgg`'s `--instdir`, it is assumed the packager has sufficient knowledge to update both *rpath* information and paths in scripts in the executables and python package directories of a non-relocatable build, and that the packaging mechanism includes the necessary tools and scripts to enable this.

runs of the code use that same environment, rather than the user's environment, so as to allows using versions of *Code_Saturne* built with different modules safely and easily.

Given this, it is recommended that when configuring and installing *Code_Saturne*, only the modules necessary for that build of for profiling or debugging be loaded. Note that as *Code_Saturne* uses the module environment detected and runtime instead of the user's current module settings, debuggers requiring a specific module may not work under a standard run script if they were not loaded when installing the code.

The detection of environment modules may be disabled using the `--without-modules` option, or the use of a specified (colon-separated) list of modules may be forced using the `--with-modules=` option.

7.7 Remarks for very large meshes

If *Code_Saturne* is to be run on large meshes, several precautions regarding its configuration and that of third-party software must be taken.

in addition to local connectivity arrays, *Code_Saturne* uses global element ids for some operations, such as reading and writing meshes and restart files, parallel interface element matching, and post-processing output. For a hexahedral mesh with N cells, the number of faces is about $3N$ (6 faces per cell, shared by 2 cells each). With 4 cells per face, the $face \rightarrow vertices$ array is of size of the order of $4 \times 3N$, so global ids used in that array's index will reach 2^{31} for a mesh in the range of $2^{31}/12 \approxeq 178.10^6$. In practice, we have encountered a limit with slightly smaller meshes, around 150 million cells.

Above 150 million hexahedral cells or so, it is thus imperative to configure the build to use 64-bit global element ids. This is the default. Local indexes use the default int size. To slightly decrease memory consumption if meshes of this size are never expected (for example on a workstation or a small cluster), the `--disable-long-gnum` option may be used.

Recent versions of some third-party libraries may also optionally use 64-bit ids, independently of each other or of *Code_Saturne*. This is the case for the Scotch and Metis, MED and CGNS libraries. In the case of graph-based partitioning, only global cell ids are used, so 64-bit ids should not in theory be necessary for meshes under 2 billion cells. In a similar vein, for post-processing output using nodal connectivity, 64-bit global ids should only be an imperative when the number of cells or vertices approaches 2 billion. Practical limits may be lower, if some intermediate internal counts reach these limits earlier.

Partitioning a 158 million hexahedral mesh using serial Metis 5 or Scotch on a front-end node with 128 Gb memory is possible, but partitioning the same mesh on cluster nodes with "only" 24 Gb each may not, so using parallel partitioning PT-Scotch or ParMetis should be preferred.

7.8 Installation with the SALOME platform

To enable SALOME platform (http://www.salome-platform.org) integration, the `--with-salome` configuration option should be used, so as to specify the directory of the SALOME installation (note that this should be the main installation directory, not the default application directory, also generated by SALOME's installers).

With SALOME support enabled, both the CFDSTUDY salome module (available by running `code_saturne salome)` after install) and the *code_aster* coupling adapter should be available.

Note that the CFDSTUDY module will only be usable with a PyQt version similar to that used in SALOME. PyQt5 is used by SALOME versions 8 and above, while PyQt4 is used for older versions. Other aspects of SALOME integration should not be impacted by an incompatible PyQt version.

Also, SALOME expects a specific directory tree when loading modules, so the CFDSTUDY and *code_aster* coupling adapter my be ignored when installing with a specified (i.e. non-default) `--datarootdir` path in the *Code_Saturne* `configure` options.

Note that specifying a SALOME directory does not automatically force the *Code_Saturne* `configure` script to find some libraries which may be available in the SALOME distribution, such as HDF5, MED, or CGNS. To indicate that the versions from SALOME should be used, without needing to provide the full paths, the following configuration options may be used for HDF5, CGNS, and MED respectively, as well as for Catalyst when available in a given Salome platform variant.

```
--with-hdf5=salome
--with-cgns=salome
--with-med=salome
--with-catalyst=salome
```

As CGNS and MED file formats are portable, MED or CGNS files produced by either *Code_Saturne* or SALOME remain interoperable.[5]

Unless a specific`--with-medcoupling` option is given, a compatible MEDCoupling library is also searched for in the SALOME distribution.

Also note that for SALOME builds containing their own Python interpreter and library, using that same interpreter for *Code_Saturne* may avoid some issues, but may then require sourcing the SALOME environment or at least its Python-related `LD_LIBRARY_PATH` for the main *Code_Saturne* script to be usable.

7.9 Example configuration commands

Most available prerequisites are auto-detected, so to install the code to the default `/usr/local` sub-directory, a command such as:

```
$ ../../code_saturne-6.0.0/configure
```

should be sufficient.

For the following examples, Let us define environment variables respectively reflecting the *Code_Saturne* source path, installation path, and a path where optional libraries are installed:

```
$ SRC_PATH=/home/projects/Code_Saturne/6.0.0
$ INSTALL_PATH=/home/projects/Code_Saturne/6.0.0
$ CS_OPT=/home/projects/opt
```

For an install on which multiple versions and architectures of the code should be available, configure commands with all bells and whistles (except SALOME support) for a build on a cluster named `athos`, using the Intel compilers (made available through environment modules) may look like this:

```
$ module purge
$ module load intel_compilers/2016.0.047
$ module load open_mpi/gcc/2.0.1
$ $SRC_PATH/code_saturne-6.0.0/configure \
--prefix=$INSTALL_PATH/arch/athos_ompi \
--with-blas=/opt/mkl-2016.0.047/mkl \
--with-hdf5=$CS_OPT/hdf5-1.10/arch/athos \
--with-med=$CS_OPT/med-4.0/arch/athos \
--with-cgns=$CS_OPT/cgns-3.4/arch/athos \
--with-ccm=$CS_OPT/libccmio-2.06.23/arch/athos \
--with-scotch=$CS_OPT/scotch-6.0/arch/athos_ompi \
--with-metis=$CS_OPT/parmetis-4.0/arch/athos_ompi \
--with-eos/$CS_OPT/eos-1.2.0/arch/athos_ompi \
CC=mpicc FC=ifort CXX=icpc
```

[5] At the least, files produced with a given version of CGNS or MED should be readable with the same or a newer version of that library.

In the example above, we have appended the `_ompi` postfix to the architecture name for libraries using MPI, in case we intend to install 2 builds, with different MPI libraries (such as Open MPI and MPICH-based Intel MPI). Note that optional libraries using MPI must also use the same MPI library. This is the case for PT-Scotch or ParMetis, but also HDF5, CGNS, and MED if they are built with MPI-IO support. Similarly, C++ and Fortran libraries, and even C libraries built with recent optimizing C compilers, may require runtime libraries associated to that compiler, so if versions using different compilers are to be installed, it is recommended to use a naming scheme which reflects this. In this example, HDF5, CGNS and MED were built without MPI-IO support, as *Code_Saturne* does not yet exploit MPI-IO for these libraries.

To avoid copying platform-independent data (such as the documentation) from different builds multiple times, we may use the same `--datarootdir` option for each build so as to install that data to the same location for each build.

7.10 Cross-compiling

On machines with different front-end and compute node architectures, such as IBM Blue Gene/Q, cross-compiling is necessary. To install and run *Code_Saturne*, 2 builds are required:

- a "front-end" build, based on front-end node's architecture. This is the build whose `code_saturne` command, GUI, and documentation will be used, and with which meshes may be imported (i.e. whose Preprocessor will be used). This build is not intended for calculations, though it could be used for mesh quality criteria checks. This build will thus usually not need MPI.
- a "compute" build, cross-compiled to run on the compute nodes. This build does not need to include the GUI, documentation, or the Preprocessor.

A debug variant of the compute build is also recommended, as always. Providing a debug variant of the front-end build is not generally useful.

A post-install step (see §9) will allow the scripts of the front-end build to access the compute build in a transparent manner, so it will appear to the users that they are simply working with that build.

Depending on their role, optional third-party libraries should be installed either for the front-end, for the compute nodes, or both:

- BLAS will be useful only for the compute nodes, and are generally always available on large compute facilities.
- Python and PyQt will run on the front-end node only.
- HDF5, MED, CGNSlib, and libCCMIO may be used by the Preprocessor on the front-end node to import meshes, and by the main solver on the compute nodes to output visualization meshes and fields.
- Scotch or Metis may be used by a front-end node build of the solver, as serial partitioning of large meshes requires a lot of memory.
- PT-Scotch or ParMetis may be used by the main solver on the compute nodes.

7.10.1 Cross-compiling configuration for Blue Gene/Q

In our example, the front-end node is based on an IBM Power architecture, runnning under Red Hat Enterprise Linux 6, on which the Python/Qt4 environment should be available as an RPM package, and installed by the administrators.

On the compute nodes, the IBM XL compilers produce static object files by default, so specifying the `--disable-shared` option is not necessary for libraries using Autotools-based installs when using those

compilers, though using `--build=ppc64 --host=bluegeneq` in this case ensures the cross-compiling environment is detected. This environment is the suggested default, and prevents shared library builds. To allow building with shared libraries while still ensuring the cross-compiling environment is detected, use `--host=powerpc64-bgq-linux` instead.

For the compute nodes, the following remarks may be mode for prerequisites:

- HDF5: building this library with its `configure` script is a pain[6], but installing HDF5 1.8.9 or above using CMake is as simple as on a workstation, and simply requires choosing the correct compilers and possibly a few other options (in the EDF *Code_Saturne* build, the GCC compilers were chosen to reduce risks, and the Fortran wrappers were not needed, so not built).
- CGNS: building CGNS is preferentially based on CMake, and no specific problems have been observed.
- MED: building MED 3.0.5 or above for *Code_Saturne* is easier than previous versions, as a new `--disable-fortan` option is available for the `configure` script. Both the C and C++ compiler wrappers must be specified, and the link may fail with the GNU compilers, due to some shared library issue (trying to force `--disable-shared`). With the IBM XL compilers, the same build works fine, as long as the `CXXFLAGS=-qlanglvl=redefmac`) option is passed. Adding the HDF5 tools path to the `$PATH` environment variable for the configuration stage may also be required.
- CoolProp: the GNU (not XL) C++ compiler must be used, and the additional `-DCMAKE_CXX_FLAGS=-U__powerpc__` option must be passed to the `cmake` command.

For an example, let us start with the front-end build:

```
$ $SRC_PATH/code_saturne-6.0.0/configure \
--prefix=$INSTALL_PATH/arch/frontend \
--with-hdf5=$CS_OPT/hdf5-1.10.3/arch/frontend \
--with-med=$CS_OPT/med-4.0/arch/frontend \
--with-cgns=$CS_OPT/cgns-3.4/arch/frontend \
--with-scotch=$CS_OPT/scotch-6.0/arch/frontend
```

For the compute node, we use the same version of Python (which is used only for the GUI and scripts, which only run on the front-end or service nodes), but the compilers are cross-compilers for the compute nodes:

```
$ $SRC_PATH/code_saturne-6.0.0/configure \
--prefix=$INSTALL_PATH/arch/bgq \
--without-blas
--with-hdf5=$CS_OPT/hdf5-1.10/arch/bgq \
--with-med=$CS_OPT/med-4.0/arch/bgq \
--with-cgns=$CS_OPT/cgns-3.4/arch/bgq \
--with-scotch=$CS_OPT/scotch-6.0/arch/bgq \
--disable-sockets --disable-dlloader -disable-nls \
--disable-frontend --enable-long-gnum \
--build=ppc64 --host=bluegeneq \
CC=/bgsys/drivers/ppcfloor/comm/xl/bin/mpixlc_r \
CXX=/bgsys/drivers/ppcfloor/comm/xl/bin/mpixlcxx_r \
FC=bgxlf95_r
```

The C++ compiler is specified, as it will be needed for the link stage due to C++ dependencies in the MED library, which is a static library in this example (see §5.3.3).

[6] It requires running a `yodconfigure` script and adapting other scripts (see documentation), then running this as a submitted job (or under a SLURM allocation if you are lucky enough to use this resource manager).

Note that in the above examples, we specified an install of the SCOTCH partitioning library both for the front-end and for the compute nodes. The implies a serial build of SCOTCH on the front-end node, and a parallel build (PT-SCOTCH) on the compute nodes. Both are optional, and the serial partitioning on the front-end nodes should only be used as a backup or as a reference for parallel partitioning. Unless robustness or quality issues are encountered with parallel partitioning, it should supercede serial partitioning, as it allows for a simpler toolchain even for large meshes. Similarly, METIS could be used on the front-end node, and PARMETIS on the compute nodes.

7.10.2 Compiling for Cray X series

For Cray X series, when using the GNU compilers, installation should be similar to that on standard clusters. Using The Cray compilers, options such as in the following example are recommended:

```
$ $SRC_PATH/code_saturne-6.0.0/configure \
--prefix=$INSTALL_PATH/arch/xc30 \
--with-hdf5=$CS_OPT/hdf5-1.10/arch/xc30 \
--with-med=$CS_OPT/med-4.0/arch/xc30 \
--with-cgns=$CS_OPT/cgns-3.4/arch/xc30 \
--with-scotch=$CS_OPT/scotch-6.0/arch/xc30 \
--disable-sockets --disable-nls \
--disable-shared \
--host=x86_64-unknown-linux-gnu \
CC=cc \
CXX=CC \
FC=ftn
```

In case the automated environment modules handling causes issues, adding the `--without-modules` option may be necessary. In that case, caution must be exercised so that the user will load the same modules as those used for installation. This is not an issue if modules for *Code_Saturne* is also built, and the right dependencies handled at that level.

Note that to build without OpenMP with the Cray compilers, `CFLAGS="-h noomp"` and `FCFLAGS="-h noomp"` need to be added.

7.11 Troubleshooting

If `configure` fails and reports an error, the message should be sufficiently clear in most case to understand the cause of the error and fix it. Do not forget that for libraries installed using packages, the development versions of those packages are also necessary, so if configure fails to detect a package which you believe is installed, check the matching development package.

Also, whether it succeeds or fails, `configure` generates a file named `config.log`, which contains details on tests run by the script, and is very useful to troubleshoot configuration problems. When `configure` fails due to a given third-party library, details on tests relative to that library are found in the `config.log` file. The interesting information is usually in the middle of the file, so you will need to search for strings related to the library to find the test that failed and detailed reasons for its failure.

8 Compile and install

Once the code is configured, it may be compiled and installed; for example, to compile the code (using 4 parallel threads), then install it:

```
$ make -j 4 && make install
```

To compile the documentation, add:

```
$ make pdf && make install-pdf
```

To clean the build directory, keeping the configuration, use `make clean`; To uninstall an installed build, use `make uninstall`. To clear all configuration info, use `make distclean` (`make uninstall` will not work after this).

8.1 Installing to a system directory

When installing to a system directory, such as `/usr` or `/usr/local`, some Linux systems may require running `ldconfig` as root or sudoer for the code to work correctly.

9 Post-install

Once the code is installed, a post-install step may be necessary for computing environments using a batch system, for separate front-end and compute systems (such as Blue Gene systems), or for coupling with SYRTHES 4 or code_aster. The global default MPI execution commands and options may also be overridden.

Copy or rename the `<install-prefix>/etc/code_saturne.cfg.template` to `<install-prefix>/etc/code_saturne.cfg`, and uncomment and define the applicable sections.

If used, the name of the batch system should match one of the templates in `<install-prefix>/share/code_saturne/batch`, and those may also be edited if necessary to match the local batch configuration[7]

Also, the `compute_versions` section allows the administrator to define one or several alternate builds which will be used for compute stages. This is especially useful for installation on BlueGene type machines, where 2 separate builds are required (one for the front-end nodes and one for the compute nodes). The compute-node build may be configured using the `--disable-frontend` option so as only to build and install the components required to run on compute-nodes, while the front-end build may be configured without MPI support. The front-end build's post-install step allows definition of the associated compute build.

All default MPI execution commands and options may be overriden using the `mpi` section. Note that only the options defined in this section are overridden; defaults continue to apply for all others.

For relocatable builds using ParaView/Catalyst, a `CATALYST_ROOT_DIR` environment variable may be used to specify the Catalyst location in case that was moved also.

10 Installing for SYRTHES coupling

Coupling with SYRTHES 4 requires defining the path to SYRTHES 4 at the post-install stage.

When coupling with SYRTHES 4, both *Code_Saturne* and SYRTHES must use the same MPI library, and must use the same version of the PLE (Parallel Location and Exchange) library from *Code_Saturne*. By default, PLE is built as a sub-library of *Code_Saturne*, but a standalone version may be configured and built, using the `libple/configure` script from the *Code_Saturne* source tree, instead of the top-level `configure` script. *Code_Saturne* may then be configured to use the existing install of PLE using the `--with-ple` option. Similarly, SYRTHES must also be configured to use PLE.

Alternatively, SYRTHES 4 may simply be configured to use the PLE library from an existing *Code_Saturne* install.

[7] Some batch systems allow a wide range of alternate and sometimes incompatible options or keywords, and it is for all practical purposes impossible to determine which options are allowed for a given setup, so editing the batch template to match the local setup may be necessary.

11 Shell configuration

If *Code_Saturne* is installed in a non-default system directory (i.e. outside `/usr` or `/usr/local`, it is recommended to define an alias (in the user's `.alias` or `.profile` file, so as to avoid needing to type the full path when using the code:

```
alias code_saturne="$prefix/code_saturne-$version/bin/code_saturne"
```

Note that when multiple versions of the code are installed side by side, using a different alias for each will allow using them simultaneously, with no risk of confusion.

If using the bash shell, you may also source a bash completion file, so as to benefit from shell completion for *Code_Saturne* commands and options, either using

```
.  <install-prefix>/etc/bash_completion.d/code_saturne
```

or

```
source <install-prefix>/etc/bash_completion.d/code_saturne
```

On some systems, only the latter syntax is effective. For greater comfort, you should save this setting in your `.bashrc` or `.bash_profile` file.

12 Caveats

12.0.1 Moving an existing installation

Never move a non-relocatable installation of *Code_Saturne*. Using `LD_LIBRARY_PATH` or `LD_PRELOAD` may allow the executable to run despite *rpath* info not being up-to-date, but in environments where different library, versions are available, there is a strong risk of not using the correct library. In addition, the scripts will not work unless paths in the installed scripts are updated.

To build a relocatable installation, see section 7.3.

If you are packaging the code and need both fine-grained control of the installation directories, and the possibility to support options such as `dpgg`'s `--instdir`, it is assumed you have sufficient knowledge to update both *rpath* information and paths in scripts in the executables and python package directories, and that the packaging mechanism includes the necessary tools and scripts to enable this.

In any other case, you should not even think about moving a non-relocatable build.

If you need to test an installation in a test directory before installing it in a production directory, use the `make install DESTDIR=<test_prefix>` provided by the Autotools mechanism rather than configuring an install for a test directory and then moving it to a production directory. Another less elegant but safe solution is to configure the build for installation to a test directory, and once it is tested, re-configure the build for installation to the final production directory, and rebuild and install.

12.0.2 Dynamic linking and path issues on some systems

On Linux systems and Unix-like, there are several ways for a library or executable to find dynamic libraries, listed here in decreasing priority:

- the `LD_PRELOAD` environment variable explicitly lists libraries to be loaded with maximum priority, before the libraries otherwise specified (useful mainly for instrumentation and debugging, and should be avoided otherwise);
- the `RPATH` binary header of the dependent library or executable; (if both are present, the library has priority);
- the `LD_LIBRARY_PATH` environment variable;

- the `RUNPATH` binary header of executable;
- `/etc/ld.so.cache`;
- base library directories (`/lib` and `//usr/lib`);

Note that changing the last two items usually require administrative privileges, and we have encountered cases where installing to `//usr/lib` was not sufficient without updating `/etc/ld.so.cache`. We do not consider `LD_PRELOAD` here, as it has other specific uses.

So basically, when using libraries in non-default paths, the remaining options are between `RPATH` or `RUNPATH` binary headers, or the `LD_LIBRARY_PATH` environment variable.

The major advantage of using binary headers is that the executable can be run without needing to source a specific environment, which is very useful, especially when running under MPI (where the propagation of environment variables may depend on the MPI library and batch system's configuration), or running under debuggers (where library paths would have to be sourced first).

In addition, the `RPATH` binary header has priority over `LD_LIBRARY_PATH`, allowing the installation to be "protected" from settings in the user environment required by other tools. this is why the *Code_Saturne* installation chooses this mode by default, unless the `--enable-relocatable` option is passed to `configure`.

Unfortunately, the ELF library spec indicates that the use of the `DT_RPATH` entry (for `RPATH`) has been superseded by the `DT_RUNPATH` (for `RUNPATH`). Most systems still use `RPATH`, but some (such as SUSE and Gentoo) have defaulted to `RUNPATH`, which provides no way of "protecting" an executable or library from external settings.

Also, the `--enable-new-dtags` linker option allows replacing `RPATH` with `RUNPATH`, so adding `-Wl,--enable-new-dtags` to the `configure` options will do this.

The addition of `RUNPATH` to the ELF specifications may have corrected the oversight of not being able to supersede an executable's settings when needed (though `LD_PRELOAD` is usually sufficient for debugging, but the bright minds who decided that it should replace `RPATH` and not simply supplement it did not provide a solution for the following scenario:

1. *Code_Saturne* in installed, along with the MED and HDF libraries, on a system where `--enable-new-dtags` is enabled by default.
2. Another code is installed, with its own (older) versions of MED and HDF libraries; this second code requires sourcing environment variables including `LD_LIBRARY_PATH` to work at all, so the user adds those libraries to his environment (or the admins add it to environment modules).
3. *Code_Saturne* now uses the older libraries, and is not capable of reading files generated with more recent versions

The aforementioned scenario occurs with *Code_Saturne* and Syrthes, on some machines, and could occur with *Code_Saturne* and some SALOME libraries, and there is no way around it short of changing the installation logic of these other tools, or using a cumbersome wrapper to launch *Code_Saturne*, which could still fail when *Code_Saturne* needs to load a Syrthes or SALOME environment for coupled cases. A wrapper would lead to its own problems, as for example Qt is needed by the GUI but not the executable, so to avoid *causing* issues with a debugger using its own version of Qt, separate sections would need to be defined. None of those issues exist with `RPATH`.

To avoid most issues, the *Code_Saturne* scripts also update `LD_LIBRARY_PATH` before calling executable modules, but you could be affected if running them directly from the command line.