

EDF R&D



FLUID DYNAMICS, POWER GENERATION AND ENVIRONMENT DEPARTMENT  
SINGLE PHASE THERMAL-HYDRAULICS GROUP

6, QUAI WATIER  
F-78401 CHATOU CEDEX

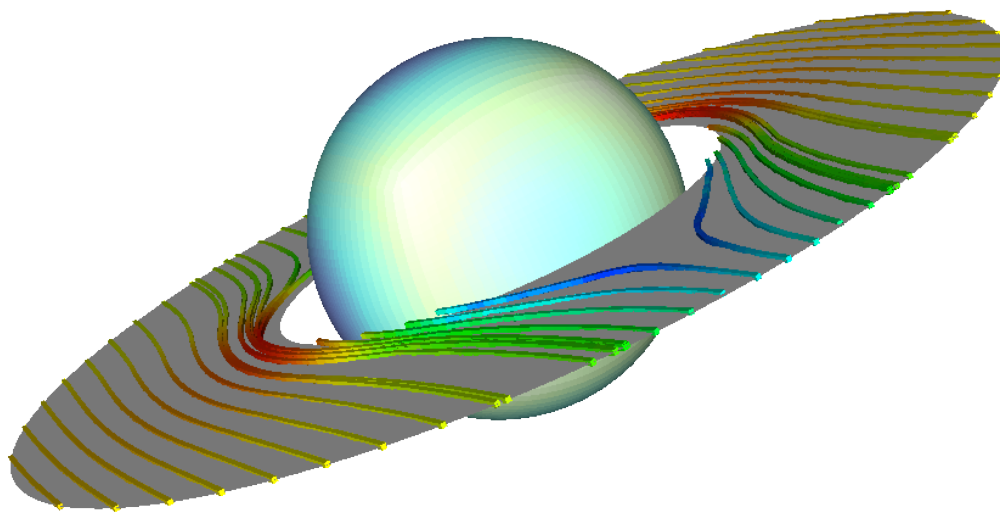
TEL: 33 1 30 87 75 40  
FAX: 33 1 30 87 79 16

JULY 2012

*Code\_Saturne* documentation

***Code\_Saturne* version 2.3.1 installation guide**

contact: [saturne-support@edf.fr](mailto:saturne-support@edf.fr)



EDF R&D	<i>Code_Saturne</i> version 2.3.1 installation guide	<i>Code_Saturne</i> documentation Page 1/ <a href="#">16</a>
---------	--	--

## TABLE OF CONTENTS

<b>1</b>	<b>Installation basics</b>	2
<b>2</b>	<b>Third-Party libraries</b>	3
2.1	INSTALLING THIRD-PARTY LIBRARIES FOR <i>Code_Saturne</i>	3
2.2	LIST OF THIRD-PARTY LIBRARIES USABLE BY <i>Code_Saturne</i>	4
2.3	NOTES ON SOME THIRD-PARTY TOOLS AND LIBRARIES	5
2.3.1	<i>Python and PyQt4</i>	5
2.3.2	<i>SCOTCH and PT-SCOTCH</i>	6
2.3.3	<i>MED</i>	6
<b>3</b>	<b>Preparing for build</b>	6
3.1	SOURCE TREES OBTAINED THROUGH A SOURCE CODE REPOSITORY	7
<b>4</b>	<b>Configuration</b>	7
4.1	DEBUG BUILDS	8
4.2	SHARED OR STATIC BUILDS	8
4.3	COMPILER FLAGS AND ENVIRONMENT VARIABLES	8
4.4	MPI COMPILER WRAPPERS	9
4.5	ENVIRONMENT MODULES	9
4.6	REMARKS FOR VERY LARGE MESHES	10
4.7	EXAMPLE CONFIGURATION COMMANDS	10
4.8	CROSS-COMPILING	11
4.8.1	<i>Cross-compiling configuration for Blue Gene/P</i>	12
4.8.2	<i>Cross-compiling configuration for Blue Gene/Q</i>	13
4.9	TROUBLESHOOTING	14
<b>5</b>	<b>Compile and install</b>	14
<b>6</b>	<b>Post-install</b>	15
<b>7</b>	<b>Installing for SYRTHES coupling</b>	15
<b>8</b>	<b>Shell completion</b>	15
<b>9</b>	<b>Caveats</b>	16
9.0.1	<i>Moving an existing installation</i>	16
9.0.2	<i>Known issues with some packages</i>	16

## 1 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 be 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-2.3.1/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.

## 2 Third-Party libraries

For a minimal build of *Code\_Saturne*, a Posix system with a C and a Fortran compiler, a Python interpreter and a `make` tool should be sufficient. For parallel runs, an MPI library is also necessary. To build an use the GUI, Libxml2 and PyQt4 (which in turn requires Qt4 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 §2.2.

### 2.1 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 §2.2 are available through the distribution’s package manager.<sup>1</sup> This requires administrator privileges, but is by far the easiest way to install third-party libraries for *Code\_Saturne*.

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. For example, on a Debian or Ubuntu system, `libxml2` is usually installed by default, but `libxml2-dev` must also be installed for the *Code\_Saturne* build to be able to use the former.

- On many large compute clusters, Environment Modules allow the administrators to provide multiple versions of many scientific libraries, as well as compilers or MPI libraries, using the `module` command. More details on Environment Modules may be found at <http://modules.sourceforge.net>. 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

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

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 and 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.

## 2.2 List of third-party libraries usable by *Code\_Saturne*

The list of third-party software usable with *Code\_Saturne* is provided here:

- 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. Starting with version 2.3, *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 anymore, but only for unit testing. 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.
- PyQt4 is required by the *Code\_Saturne* GUI. PyQt4 in turn requires Qt4, 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 Libxml2 is available.
- Libxml2 is required to read XML files edited with the GUI. If this library is not available, only user subroutines may be used to setup data.
- 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,
- MED is necessary to read or write mesh and visualization files using the MED format, mainly used by the SALOME platform.
- libCCMIO is necessary to import mesh files generated 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.

- 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 usually slightly lower than that obtained with SCOTCH or PT-SCOTCH, but these libraries are faster.

Though broadly available, the license is quite restrictive, so SCOTCH or PT-SCOTCH may be preferred (*Code\_Saturne* may be built with both METIS and SCOTCH libraries).

For developers, the GNU Autotools (Autoconf, Automake, Libtool) as well as gettext will be necessary. To build the documentation, pdfL<sup>A</sup>T<sub>E</sub>X and fig2dev (part of TransFig) will be necessary.

## 2.3 Notes on some third-party tools and libraries

### 2.3.1 Python and PyQt4

*Code\_Saturne* requires a Python interpreter, with Python version 2.4 or above. The base scripts should work both with Python 2 or Python 3 versions, but have not been tested recently with the latter. The GUI is Python 2 only, so using Python 3 is not currently recommended.

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 may for example use a 32-bit version of Python with 64-bit *Code\_Saturne* libraries and executables.

The GUI is written in PyQt4 (Python bindings for Qt4), so but Qt4 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. When running on a system which does not provide these libraries, there are several alternatives:

- build *Code\_Saturne* without the GUI. If built with Libxml2, 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 PyQt4 installed, and a compute cluster with an older system, this may avoid requiring a build of Qt4 and PyQt4 on the cluster if users find this too daunting.
- Install a local Python interpreter, and add Qt4 bindings to this interpreter.

Python (<http://www.python.org>) and Qt4 (<http://qt.nokia.com/products>) 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 Qt4) may be a lengthy process in terms of compilation time, but is well automated and usually devoid of nasty surprises.<sup>2</sup>

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 PyQt4 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 PyQt4 bindings, and may be used by *Code\_Saturne*'s `configure` script by passing `PYTHON=<path_to_python_executable>`.

<sup>2</sup>The only case in which the *Code\_Saturne* developers have has issues with Qt4 was when trying to force an install into 64-bit mode with the GNU compilers (version 4.1.2) on a PowerPC 64 architecture running SLES 10 Linux, on which compilers default to building 32 bit code, although 64 bit is available. Using default options on the same machine led to a perfectly functional 32-bit Qt installation

- add Python Qt4 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.<sup>3</sup>

The process is similar to the previous one, but SIP and PyQt4 installation requires a few additional configuration options in this case. See the SIP and PyQt4 reference guides for detailed instructions, especially the *Building a Private Copy of the SIP Module* section of the SIP guide.

### 2.3.2 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 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 is 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 BlueGene/P type machines, it may be necessary to start the build process, let it fail, run this executable manually using `mpirun`, then pursue the build process.

### 2.3.3 MED

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

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.

## 3 Preparing for build

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

<sup>3</sup>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.

```
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.

### 3.1 Source trees obtained through a source code repository

For developers obtaining the code was obtained through a version control system such as Subversion, an additional step is required:

```
$ cd saturne
$ autoreconf -vi
$ 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

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 developpers building the code from a repository can be expected to need a more complete developpement 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 pdf
$ make install-pdf
```

## 4 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 options with `--disable-`. Similarly, for every `--with-`, `--without-` is also possible.

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



```
$ /home/user/Code_Saturne/2.3/src/code_saturne-2.3.1/configure \
--prefix=/home/user/Code_Saturne/2.3/arch/prod \
--with-med=/home/user/opt/med-3.0 \
CC=/home/user/opt/mpich2-1.4/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 §3. 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.

## 4.1 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-2.3.1/configure \
--prefix=/home/user/Code_Saturne/2.3/arch/dbg \
--with-med=/home/user/opt/med-3.0 \
--enable-debug \
CC=/home/user/opt/mpich2-1.4/bin/mpicc FC=gfortran
```

## 4.2 Shared or static builds

By default, on most architectures, both shared and static libraries for *Code\_Saturne* will be built, and the executables will be linked with shared libraries. To disable either shared or static libraries, add either `--disable-shared` or `--disable-static` to the options passed to `configure`. This will speed-up the build, process as each file will only be built once, and not twice.

In some cases, a shared build may fail due to some dependencies on static-only MPI libraries. In this case, `--disable-shared` will be necessary. Disabling shared libraries has also been seen to avoid issues with linking 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.

## 4.3 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 for the C parts of FVM and the Kernel).

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.

## 4.4 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`.

*Never* 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, but they can lead to build failures, especially in cross-compilation configurations.<sup>4</sup>

## 4.5 Environment Modules

As noted in §2.1, 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 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 environnement 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.

<sup>4</sup>`configure` determines which libraries are necessary to link the Fortran runtime using a C or C++ compiler as a linker. This avoids conflicts between linking with a C++ compiler and linking with a Fortran compiler when both runtimes are necessary, for example when using the MED library. When using an MPI Fortran wrapper, extra libraries that are not normally necessary will be added to those we link with, and the Libtool script that is part of the build system will often try to add further dependencies, mixing-up front-end and compute node compiler options and libraries (Libtool may be very practical when it works, but in complex situations where it guesses incorrectly at the commands it should run, it always acts as if it knows best, and is very difficult to work around).

## 4.6 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  $2^{31}/12 \approx 178.10^6$ . 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, with the `--enable-long-gnum` option. Local indexes will still use the default int size, so memory consumption will only be slightly increased.

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.

Note also that METIS 4 is known to crash for meshes in the range of 35 million cells and above, so METIS 5 or SCOTCH are necessary. Partitioning a 158 million hexahedral mesh using METIS 5 on a front-end node with 128 Gb memory is possible, but partitioning the same mesh on cluster nodes with 24 Gb each may not, so using parallel partitioning PT-SCOTCH or PARMETIS should be preferred.

## 4.7 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-2.3.1/configure
```

should be sufficient.

For the following examples, Let us define environment variables repectively reflecting the *Code\_Saturne* source path, installation path, and a path where optional libraries are installed:

```
$ SRC_PATH=/home/projects/Code_Saturne/2.3
$ INSTALL_PATH=/home/projects/Code_Saturne/2.3
$ 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 `ivano`, using the Intel compilers (made available through environment modules) may look like this:

```
$ module purge
$ module load intel_compilers/12.1.1.256
$ module load open_mpi/gcc/1.4.5
$ $SRC_PATH/code_saturne-2.3.1/configure \
--prefix=$INSTALL_PATH/arch/ivanoe_ompi \
--with-blas=/opt/intel/composerxe_xe.2011.sp1.7.256/mkl \
--with-libxml2=$CS_OPT/libxml2-2.8/arch/ivanoe \
--with-hdf5=$CS_OPT/hdf5-1.8.9/arch/ivanoe \
--with-med=$CS_OPT/med-3.0/arch/ivanoe \
--with-cgns=$CS_OPT/cgns-3.1/arch/ivanoe \
--with-ccm=$CS_OPT/libccmio-2.6.23/arch/ivanoe \
--with-scotch=$CS_OPT/scotch-5.1.12/arch/ivanoe_ompi \
--with-metis=$CS_OPT/parmetis-4.0/arch/ivanoe_ompi \
CC=mpicc FC=ifort CXX=icpc
```

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 MPICH2). 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.

## 4.8 Cross-compiling

On machines with different front-end and compute node architectures, such as IBM Blue Gene/P, 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 §6) 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 PyQt4 will run on the front-end node only.
- Libxml2 must be available for the compute nodes if the GUI is used.
- HDF5, MED and CGNSlib 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.

- libCCMIO is used by the Preprocessor exclusively, so it may be needed on the front-end node only.
- SCOTCH or METIS may be used by a front-end node build of the partitioner, as serial partitioning of large meshes requires a lot of memory.
- PT-SCOTCH or PARMETIS may be used by a compute node build of the partitioner.

#### 4.8.1 Cross-compiling configuration for Blue Gene/P

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

```
$ $SRC_PATH/code_saturne-2.3.1/configure \
--prefix=$INSTALL_PATH/arch/frontend \
--with-hdf5=$CS_OPT/hdf5-1.8.6/arch/frontend \
--with-med=$CS_OPT/med-3.0/arch/frontend \
--with-cgns=$CS_OPT/cgns-3.1/arch/frontend \
--with-scotch=$CS_OPT/scotch-5.1.11/arch/frontend \
PYTHON=$CS_OPT/python/arch/frontend/bin/python \
CFLAGS="-m64" FCFLAGS="-m64" CXXFLAGS="-m64"
```

In this example, the front-end node is based on an IBM Power architecture, on which the GCC compiler is available, but produces 32-bit code by default. Adding the "-m64" flags force the compiler into 64-bit mode, allowing the Preprocessor to import meshes up into the 100-million cell range.

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:

```
$ $SRC_PATH/code_saturne-2.3.1/configure \
--prefix=$INSTALL_PATH/arch/bgp \
--with-blas=/opt/ibmmath/essl/4.4 \
--with-libxml2=$CS_OPT/libxml2-2.3.32/arch/bgp \
--with-hdf5=$CS_OPT/hdf5-1.8.6/arch/bgp \
--with-med=$CS_OPT/med-3.0/arch/bgp \
--with-cgns=$CS_OPT/cgns-3.1/arch/bgp \
--with-scotch=$CS_OPT/scotch-5.1.11/arch/bgp \
--disable-sockets --disable-dlloader --disable-nls \
--disable-frontend --enable-long-gnum \
--build=ppc64 --host=bluegenep \
CC=mpixlc_r FC=bgxlf90_r CXX=mpixlcxx_r \
PYTHON=$CS_OPT/python/arch/bgp/bin/python
```

Here, the `--build=ppc64 --host=bluegenep` options ensure the `configure` script into cross-compilation mode. With a front-end base on an Intel or AMD, architecture, `--build=x86_64` or `--build=amd64` should be used instead of `--build=ppc64`. For the host (target) architecture, `--host=bluegenep` is recognized by current GNU Autoconf versions, so it is preferred, but `--host=ppc` also works well. Actually, any choice of build and host architectures recognized by Autoconf would probably work, as long as build and host are different.

The C++ compiler is also 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 §2.3.3).

The thread-safe compiler wrappers used here should not be necessary for *Code\_Saturne*, but in our experience, the ESSL BLAS are correctly detected only with those wrappers, not with the single-threaded versions.<sup>5</sup>

<sup>5</sup>This might be due to a bug in the ESSL BLAS detection of *Code\_Saturne*, although the code has been checked.

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. This implies a serial build of SCOTCH on the front-end node, and a parallel build (PT-SCOTCH) on the compute nodes. Both are optional. Similarly, METIS could be used on the front-end node, and PARMETIS on the compute nodes.

## 4.8.2 Cross-compiling configuration for Blue Gene/Q

In our example, the front-end node is based on an IBM Power architecture, running under Red Hat Enterprise Linux 6, on which the Python/Qt4 environment should be available as an RPM package, and installed by the administrators. If this is not possible, the Python/Qt4 aspects of the Blue Gene/P example may be adapted here.

On the compute nodes, the IBM XL compilers produce static object files by default, so 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.

As the front-end nodes of Blue Gene/Q machines may be expected to run Red Hat EL 6.x linux variants, instead of 5.x for blue Gene/P, more up-to date compilers and libraries (such as Python/Qt4) should be available as packages, easily installable by the system administrators.

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

- LibXml: to reduce the size and simplify the installation of this library, the `--with-ftp=no`, `--with-http=no`, and `--without-modules` options may be used with `configure`.
- HDF5: building this library with its `configure` script is a pain<sup>6</sup>, but installing HDF5 1.8.9 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 3.1 is based on CMake, and no specific problems have been observed.
- MED: building MED 3.0.5 for *Code\_Saturne* is easier than previous versions, as a new `--disable-fortran` 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.

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

```
$ $SRC_PATH/code_saturne-2.3.1/configure \
--prefix=$INSTALL_PATH/arch/frontend \
--with-hdf5=$CS_OPT/hdf5-1.8.9/arch/frontend \
--with-med=$CS_OPT/med-3.0/arch/frontend \
--with-cgns=$CS_OPT/cgns-3.1/arch/frontend \
--with-scotch=$CS_OPT/scotch-5.1.12/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:

<sup>6</sup>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).



```
$ $SRC_PATH/code_saturne-2.3.1/configure \
--prefix=$INSTALL_PATH/arch/bgq \
--with-blas=/opt/ibmmath/essl/5.1 \
--with-blas-type=ESSL \
--with-libxml2=$CS_OPT/libxml2-2.8/arch/bgq \
--with-hdf5=$CS_OPT/hdf5-1.8.9/arch/bgq \
--with-med=$CS_OPT/med-3.0/arch/bgq \
--with-cgns=$CS_OPT/cgns-3.1/arch/bgq \
--with-scotch=$CS_OPT/scotch-5.1.12/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 §2.3.3).

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. This 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 supersede 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.

## 4.9 Troubleshooting

If `configure` fails and reports an error, the message should be sufficiently clear in most cases 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.

## 5 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).

## 6 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.

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<sup>7</sup>ome 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.

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.

## 7 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.

## 8 Shell completion

If using the bash shell, you may 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.

---

<sup>7</sup>S



## 9 Caveats

### 9.0.1 Moving an existing installation

*Never move an installed build* of *Code\_Saturne*. As the build system is based on the GNU Autotools, not only are library paths hard-coded using *rpath* type info, but the code's scripts also contain absolute paths. 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. If you are not able to update those paths without further explanation, you should not even think about moving the build.

The mistake of moving an installed build is most often done not by beginners, but by more experienced users or administrators who believe they know enough to force a behavior against the logic of the build system. Except for those experienced not only in installing codes but also in maintaining advanced Autotool scripts for at least one software package, this is usually presumptuous.

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.

### 9.0.2 Known issues with some packages

On quite a few clusters, some issues have been encountered when using versions of Open MPI built with the Intel compiler suite. Typically, crashes in the MPI-IO layers are almost guaranteed, but issues may also arise later, apparently with some collectives, so disabling MPI-IO is not enough.

Using Intel compilers for *Code\_Saturne* itself and a build of Open MPI using the GNU compilers usually works fine, and the performance difference should be minimal (it can be higher using the GNU compilers also for *Code\_Saturne* itself), so the issue is not in *Code\_Saturne* itself, but it requires further investigation. In any case, avoiding this buggy combination may avoid you many wasted hours.