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 2011

Code_Saturne documentation

Code_Saturne version 2.0.2 installation guide

contact: saturne-support@edf.fr

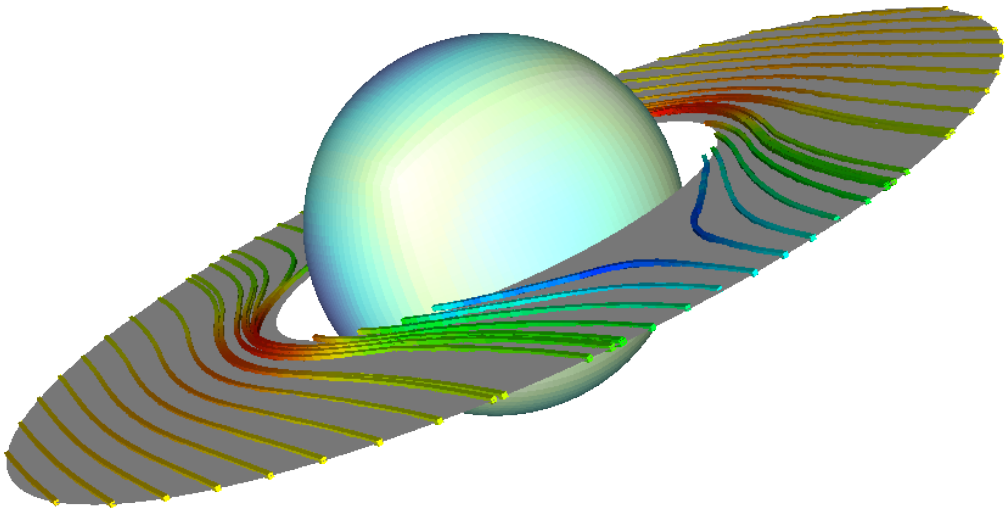

TABLE OF CONTENTS

1 *Code_Saturne* subsystems

The *Code_Saturne* CFD tool is built of several libraries, some of which are optional, and which may themselves use third-party tools. Base libraries on which others are built must be installed first, The libraries comprising *Code_Saturne* are the following (in recommended order of installation).

- BFT (Base Functions and Types) is mainly a portability layer for the rest of the code. It is written in ANSI C, and is used by all the other parts of the code, so it should be installed first.
- the Preprocessor (originally named "Enveloppe *Code_Saturne*", or ECS) also comprises the optional partitioner, and a preprocessed file dump tool. Building *Code_Saturne* without this package is possible, if it is available on another machine, but it is a part of any standard build. In addition to the required BFT library, this package may use third-party libraries such as MED, CGNS, libCCMIO to allow import of the corresponding mesh formats, and METIS, and SCOTCH for domain partitioning.
- FVM (Finite Volume Mesh) is used to provide finite volume mesh I/O and some other services such as interpolation, possibly in parallel using MPI. This package requires the BFT library, and may use third-party libraries such as MED and CGNS to allow export to the corresponding mesh formats. To build a *Code_Saturne* with parallel execution support, FVM must be built with MPI.
- MEI (Mathematical Expression Interpreter) is used to allow definition of complex properties and boundary conditions using mathematical expressions defined through the GUI, avoiding the need for user subroutines in many cases. This package is optional, as it is only useful when the GUI is used. It requires the BFT library.
- The *Code_Saturne* solver, or kernel (NCS stands for "Noyau *Code_Saturne*", or *Code_Saturne* Kernel in French) is the major and (final) part of the installation. It requires BFT, FVM, and optionally MEI. For parallel execution, it must be built with the same MPI library as FVM. This module also contains the optional GUI and most of the code's documentation.

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.

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

3 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 §3.2.

3.1 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 §3.2 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*.

  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 us 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 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.

3.2 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 Kernel (NCS) for certain operations such as vector sums, dot products, etc. If no third-party BLAS is provided, *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.

reverts to its own implementation of BLAS routines, similar to the legacy Netlib BLAS, so no functionality is lost here. With optimized BLAS such as Atlas, MKL, ESSL, and others, simple operations such as vector dot product, $Ax + y$, and such are often an order of magnitude faster than with the usual compiler optimizations, leading to an overall performance increase of about 10%.

- PyQt4 is required by the *Code_Saturne* GUI (part of NCS). 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.

- SWIG is required to build MEI Python bindings for the GUI (Python development headers are also necessary). Without this tool, the GUI will not allow the user to define MEI expressions, although the Solver will still be able to interpret such expressions when built with MEI and using an XML file produced by an install using MEI and its Python bindings. A minimum of SWIG version 1.3.30 is required, due to bugs in older versions.

- Libxml2 is required by the Kernel (NCS) 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. It may be used by the Preprocessor for mesh import, FVM for post-processing output.

- 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. It may be used by the Preprocessor for mesh import, FVM for post-processing output (the former is usually more useful than the latter, as the built-in EnSight output is readable by most post-processing tools).

- MED is necessary to read or write mesh and visualization files using the MED format, mainly used by the SALOME platform. It may be used by the Preprocessor for mesh import, FVM for post-processing output (the former is usually more useful than the latter, as the built-in EnSight output is readable by most post-processing tools).

- libCCMIO may be used by the Preprocessor to import mesh files generated by **STAR-CCM+** using its native format.

- METIS may be used by the partitioner, part of the ECS (Preprocessor) package, to optimize mesh partitioning. Depending on the mesh, parallel computations with meshes partitioned with this library may be from 10% to 50% faster than using the built-in space-filling curve based partitioning.

  Though broadly available, the license is quite restrictive, so SCOTCH may be preferred.

- SCOTCH is an alternative mesh partitioning library, and may also be used by the ECS (Preprocessor) package. quality is usually slightly higher than that obtained with METIS, though this library is not as fast.

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.

3.3 Notes on some third-party tools and libraries

3.3.1 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.[2].

  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`.

- 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.[3]

  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.

3.3.2 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

[2] 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

[3] 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.

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.

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.

4 Preparing for build

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

```
$ tar xvzf bft-1.1.5.tar.gz
$ tar xvzf ecs-2.0.2.tar.gz
$ tar xvzf fvm-0.15.3.tar.gz
$ tar xvzf mei-1.0.1.tar.gz
$ tar xvzf ncs-2.0.2.tar.gz
```

If the packages are provided as zip files, `unzip` may be substituted for `tar xvzf`. If for example you unpacked the directory in a directory named `/home/user/Code_Saturne/2.0/src`, you will now have sub-directories named `bft-1.1.5`, `ecs-2.0.2`, `fvm-0.15.3`, `mei-1.0.1`, and `ncs-2.0.2`.

For developers obtaining the code was obtained through a version control system such as Subversion, running a `./sbin/bootstrap` script is also required for each package; for example, for BFT:

```
$ cd bft-1.0.5
$ ./sbin/bootstrap
$ cd ..
```

This script requires the GNU Autotools (Autoconf, Automake, and Libtool) to be installed.

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, for BFT:

```
$ mkdir bft-1.0.5_build
$ cd bft-1.0.5_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.

5 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 for a given package, 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, for the FVM package, on an account

named users, with a specific installation of the MED IO library, based on a system-wide installation of HDF5 (which will be auto-detected), and using MPI compiler wrappers, a configure command may look like this:

```
$ /home/user/Code_Saturne/2.0/fvm-0.15.3/configure \
--prefix=/home/user/Code_Saturne/2.0/arch/prod \
--with-bft=/home/user/Code_Saturne/2.0/arch/prod \
--with-med=/home/user/opt/med-3.0 \
CC=/home/user/opt/mpich2-1.4/bin/mpicc
```

We may note that as FVM requires BFT, the installation path prefix of BFT must be provided using `--with-bft=` if BFT is installed in a non-default system path. If BFT is installed under `/usr` or `/usr/local` (which is the default id no `--prefix=` option was provided to the BFT configuration, it will be detected automatically, and the `--with-bft=` does not need to be provided. The same holds true for the FVM and MEI libraries when configuring the *Code_Saturne* kernel (NCS).

In the rest of this section, we will assume that we are in a build directory separate from sources, as described in §4. 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.

5.1 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:

```
$ /home/user/Code_Saturne/2.0/fvm-0.15.3/configure \
--prefix=/home/user/Code_Saturne/2.0/arch/prod \
--with-bft=/home/user/Code_Saturne/2.0/arch/dbg \
--with-med=/home/user/opt/med-3.0 \
--enable-debug \
CC=/home/user/opt/mpich2-1.4/bin/mpicc
```

Debug and non debug packages may be mixed, so in this example, we could use a non-debug build of BFT, bit for consistency and better debugging capabilities, it is recommended to have a full debug version alongside a production version.

5.2 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 by default. 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.

5.3 MPI compiler wrappers

MPI environments generally provide compiler wrappers, usually with names similar to `mpicc` 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 only works in some cases, as a fixed list of libraries is tested for, so using MPI compiler wrapper is the simplest and safest solution. Simply use a `CC=mpicc` or similar option instead of `--with-mpi`.[4].

MPI compiler wrappers only need to be used for the C compiler, and only for FVM and for the kernel. *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.[5]

5.4 Environment Modules

As noted in §3.1, 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 allow 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 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 environement 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.

5.5 Batch Systems

Code_Saturne's `configure` script tries to detect if a resource management or job scheduling system is available based on the presence of common commands. If such a system for which *Code_Saturne* has a template is detected, that template will be automatically inserted in run scripts. If multiple systems are detected (which may be the case when one system provides wrappers for another), the script will complain unless the `--with-batch=` option is specified. This option may either be used with a predefined template (`CCC`, `LOADL`, `LOADL_BG`, `LSF`, `PBS`, `SGE`, or `SLURM`), but an absolute path to

[4] On Linux distributions such as Debian or Ubuntu, `--with-mpi` works with Open MPI, but `CC=mpicc` is necessary to build with MPICH2

[5] `configure` determines which libraries are necessary to link the Fortran runtime using a C compiler as a linker. 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 is guesses incorrectly at the commands it should run, it always acts as if it knows best, and is very difficult to work around).

a local template file may also be chosen.

5.6 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/<package_name>_auto_flags.sh` script in all of CS's packages (for example, `config/cs__auto_flags.sh` for the Kernel. This file is 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.

To disable automatic setting of flags, the `--disable-auto-flags` option may be used, and the classical `CPPFLAGS`, `CFLAGS`, `FCCFLAGS`, `LDFLAGS`, and `LIBS` environment variables may be used. A mix of automatic and prescribed flags may also be experimented. for packages other than the Kernel, `CPP_ADD`, `CC_ADD`, `LD_ADD`, and `LIBS_ADD` may be used to add options without overriding automatic flags. For the kernel, the standard flags provided by the user are appended to the automatic flags (which may be disabled if only user flags are desired).

6 Compile and install

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

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

The solver (NCS) also includes the documentation, which may be built with:

```
$ 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).

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 FVM library 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. Use

of 64-bit ids should not in theory be necessary for meshes under 2 billion cells, whether for partitioning or post-processing output, but practical limits might 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 front-end nodes with a large amount of memory may be necessary to optimize partitioning of large meshes. The backup solution is to simply use the built-in parallel Morton Z curve partitioning, which may produce partitions of lower quality, but runs transparently at *Code_Saturne* calculation initialization, and is highly scalable.

8 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.

9 Installing for SYRTHES coupling

When coupling with SYRTHES using MPI, both *Code_Saturne* and SYRTHES must use the same MPI library. Coupling with SYRTHES is also possible using TCP/IP sockets, but this recommended only when running on a single node. On compute clusters, the MPI environment will distribute SYRTHES and *Code_Saturne* instances to different processors when they are coupled using MPI, while using sockets, the SYRTHES process will usually run on a node on which *Code_Saturne* is also running, which may degrade performance. On a single node, this is not an issue, and allows coupling *Code_Saturne* and SYRTHES even with serial builds. In any case, the `--with-syrthes=` option of the Kernel's `configure` script must be used so that *Code_Saturne* may find SYRTHES.

10 Example configuration and build commands

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

```
$ ../../saturne/configure
```

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/2.0/src
$ INSTALL_PATH=/home/projects/Code_Saturne/2.0
$ 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 `ivanoe`, using the Intel compilers (made available through environment modules) is described in the following examples.

First, environment modules are loaded if necessary:

```
$ module purge
$ module load intel_compilers/12.0.3.174
$ module load open_mpi/gcc/1.4.3
```

10.1 BFT build

To configure and install BFT, we use the following commands:

```
$ mkdir -p bft-1.0.5_build/ivanoe && cd bft-1.0.5_build/ivanoe
$ $SRC_PATH/bft-1.0.5/configure \
--prefix=$INSTALL_PATH/arch/ivanoe \
CC=icc
$ make -j 4 && make install && make clean
$ cd ..
```

10.2 Preprocessor build

Once BFT is installed, to configure and install the Preprocessor, we may use the following commands:

```
$ mkdir -p ecs-2.0.2_build/ivanoe && cd ecs-2.0.2_build/ivanoe
$ $SRC_PATH/ecs-2.0.2/configure \
--prefix=$INSTALL_PATH/arch/ivanoe \
--with-bft=$INSTALL_PATH/arch/ivanoe
--with-hdf5=$CS_OPT/hdf5-1.8.7/arch/ivanoe \
--with-med=$CS_OPT/med-3.0/arch/ivanoe \
--with-cgns=$CS_OPT/cgns-3.1/arch/ivanoe \
--with-ccm=$CS_OPT/libccmio-2.6.19/arch/ivanoe \
--with-scotch=$CS_OPT/scotch-5.1.11/arch/ivanoe \
--with-metis=$CS_OPT/parmetis-3.2/arch/ivanoe \
CC=icc
$ make -j 4 && make install && make clean
$ cd ..
```

10.3 FVM build

Once BFT is installed, to configure and install FVM, we may use the following commands:

```
$ mkdir -p fvm-0.15.3_build/ivanoe && cd fvm-0.15.3_build/ivanoe
$ $SRC_PATH/fvm-0.15.3/configure \
--prefix=$INSTALL_PATH/arch/ivanoe_ompi \
--with-bft=$INSTALL_PATH/arch/ivanoe \
--with-hdf5=$CS_OPT/hdf5-1.8.6/arch/ivanoe_ompi \
--with-med=$CS_OPT/med-3.0/arch/ivanoe_ompi \
--with-cgns=$CS_OPT/cgns-3.1/arch/ivanoe_ompi \
CC=mpicc
$ make -j 4 && make install && make clean
$ cd ..
```

In the example above, we have appended the `_ompi` postfix to the architecture name, 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 HDF5, CGNS, and MED

if they are built with MPI-IO support (which is not currently used by *Code_Saturne* for these libraries). 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 FVM (and thus *Code_Saturne*) does not yet exloit MPI-IO for these libraries.

10.4 MEI build

Once BFT is installed, to configure and install MEI, we may use the following commands:

```
$ mkdir -p mei-1.0.2_build/ivanoe && cd mei-1.0.2_build/ivanoe
$ $SRC_PATH/mei-1.0.2/configure \
--prefix=$INSTALL_PATH/arch/ivanoe \
--with-bft=$INSTALL_PATH/arch/ivanoe \
--with-python-exec=$CS_OPT/python-2.7.1/arch/ompi/bin \
--with-swig-exec=$CS_OPT/swig-1.3.39/arch/ompi/bin \
CC=icc $ make -j 4 && make install && make clean
$ cd ..
```

10.4.1 Using a specific Python interpreter

Note that the MEI's `--with-python-exec=` option allows specifying the path where the `python` executable will be found, but not the name of the executable. On Linux distributions where several Python interpreters are installed under `/usr/bin` using different names, such as `python`, `python2.6`, and `python2.7`, only the default may be used. If necessary, a workaround is to create a wrapper script named `python` in a directory named `bin`, which calls the desired Python interpreter and passes arguments. Using the installation's target directory seems a natural solution, for example:

```
$ cd $INSTALL_PATH/arch/ivanoe/bin
$ echo "#/bin/sh" > python
$ echo "/usr/bin/python2.7 $@" >> python
$ chmod +x python
$ cd -
```

The `bin` directory containing the `python` wrapper may now be passed to the `--with-python-exec=` `configure` option.

10.5 Solver build

```
$ mkdir -p ncs-2.0.2_build/ivanoe && cd ncs-2.0.2_build/ivanoe
$ $SRC_PATH/ncs-2.0.2/configure \
--prefix=$INSTALL_PATH/arch/ivanoe_ompi \
--with-bft=$INSTALL_PATH/arch/ivanoe \
--with-fvm=$INSTALL_PATH/arch/ivanoe_ompi \
--with-mei=$INSTALL_PATH/arch/ivanoe \
--with-prepro=$INSTALL_PATH/arch/ivanoe \
--with-blas=/opt/intel/composerxe-2011.3.174/mkl \
--with-libxml2=$CS_OPT/libxml2-2.3.32/arch/ivanoe \
--with-syrthes=/home/projects/syrthes3.4.3 \
--with-python-exec=$CS_OPT/python-2.7.1/arch/ompi/bin \
CC=mpicc FC=ifort $ make -j 4 && make install && make clean
$ cd ..
```

The remarks regarding the selection of a Python interpreter as in §10.4.1 hold here also.

10.6 Cross-compiling

On machines with different front-end and compute node architectures, such as IBM Blue Gene/P, cross-compiling is necessary; some parts of the code need to be built for the front-end, other parts need to be built for the compute nodes, and some parts need to be built for both:

- BFT will be needed both by the Preprocessor and the Kernel, so it must be built for both architectures.
- The Preprocessor only needs to be built for the front-end.
- If used, MEI needs to be built for the compute nodes, but a build on the front-end with Python bindings will be necessary for the GUI to allow defining MEI expressions.
- FVM and NCS (the Kernel) only need to be built for the compute nodes, though a build for the front-end may be useful for mesh verification. In any case, the Python interpreter used by the GUI and scripts will only need to run on the front-end or service nodes: execution on the compute nodes is limited to the `cs_solver` executable when launched using `mpirun`. using

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

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, 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.

In the following subsections, only the `configure` commands are detailed, as the creation of build directories and the running of `make` and `make install` commands has already been described for other cases, and bring nothing new here.

In our 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.

An additional complexity stems from this 32-bit default, as we have had difficulty forcing the build of Qt4 in 64-bit mode on such a front-end node running SUSE Linux Enterprise Server 10, on which it is not pre-installed. Forcing 64-bit mode leads to link errors, so we use a 32-bit version. This implies that we must also use a 32-bit build of Python for PyQt4, which in turn means that the Python bindings for MEI used by the GUI must be 32-bit code, also requiring an additional build of BFT.

10.6.1 BFT configurations

For the compute nodes:

```
$ $SRC_PATH/bft-1.0.5/configure \
--prefix=$INSTALL_PATH/arch/bgp \
--without-zlib \
--build=ppc64 \
--host=bluegenep \
CC=bgxlc
```

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

For the front-end nodes:

```
$ $SRC_PATH/bft-1.0.5/configure \
--prefix=$INSTALL_PATH/arch/frontend \
CC_ADD="-m64" \
LD_ADD="-m64"
```

For the Python bindings on the front-end nodes:

```
$ $SRC_PATH/bft-1.0.5/configure \
--prefix=$INSTALL_PATH/arch/frontend_python
```

10.6.2 Preprocessor configuration

The Preprocessor only needs to be built for the front-end nodes (forcing the compiler in 64-bit mode):

```
$ $SRC_PATH/ecs-2.0.2/configure \
--prefix=$INSTALL_PATH/arch/frontend \
--with-bft=$INSTALL_PATH/arch/frontend \
--with-hdf5=$CS_OPT/hdf5-1.8.7/arch/frontend \
--with-med=$CS_OPT/med-3.0/arch/frontend \
--with-cgns=$CS_OPT/cgns-3.1/arch/frontend \
--with-ccm=$CS_OPT/libccmio-2.6.1/arch/frontend \
--with-scotch=$CS_OPT/scotch-5.1.11/arch/frontend \
--with-metis=$CS_OPT/metis-5.0rc1/arch/frontend \
CC_ADD="-m64" \
LD_ADD="-m64"
```

10.6.3 FVM configuration

For compute nodes, the following configuration may be used if no MED or CGNS output is necessary:

```
$ $SRC_PATH/fvm-0.15.3/configure \
--prefix=$INSTALL_PATH/arch/bgp \
--with-bft=$INSTALL_PATH/arch/bgp \
--build=ppc64 \
--host=bluegenep \
CC=mpixlc
```

CGNS support may be configured normally, but FVM output is a bit more involved: MED uses C++ runtime libraries, which need to be defined explicitly for the code to link correctly. The `--with-med-dep-dirs` and `--with-med-dep-libs` options, which may define comma-separated values (no whitespace) allow defining this and passing this info to the Kernel build, but an additional complication is that `libstdc++` should not be defined through a `-lstdc++` option, but through a absolute path: otherwise, Libtool tends to confuse front-end and compute node libraries, and add front-end library options to the compute-node link stage, leading to an error. There is no clean way around this, so we pass the absolute path to that library using the `LIBS` variable:

```
$ $SRC_PATH/fvm-0.15.3/configure \
--prefix=$INSTALL_PATH/arch/bgp \
--with-bft=$INSTALL_PATH/arch/bgp \
--with-hdf5=$CS_OPT/hdf5-1.8.6/arch/bgp \
--with-med=$CS_OPT/med-3.0/arch/bgp \
--with-med-dep-dirs=/opt/ibmcmp/vacpp/bg/9.0/bglib \
--with-med-dep-libs=ibmc++ \
--with-cgns=$CS_OPT/cgns-3.1/arch/bgp \
--build=ppc64 \
--host=bluegenep \
LIBS=/bgsys/drivers/ppcfloor/ppc/gnu-linux/powerpc-bgp-linux/lib/libstdc++.a \
CC=mpixlc
```

Note that MED is generally more useful for the Preprocessor than for post-processing: meshes built using the SALOME platform will be produced in MED format, so importing it may be useful. For post-processing, other formats such as EnSight may be used, so MED is only useful if required by a downstream calculation. In addition, MED requires arrays to be output in one piece (the parallel IO enabled by recent MED 3 releases is not yet handled by FVM). Global arrays must be gathered to the first compute node, which may require more memory than available even for mid-sized meshes (CGNS and EnSight output may use partial writes, so they do not have this limitation). MED output may thus be useful for a boundary mesh with pressure data usable by a solid mechanics code such as *Code_Aster*, but it will not be usable for the large volume meshes we expect on Blue Gene type machines. Building FVM with support for MED may thus be worthwhile only if truly required by a downstream application.

A front-end configuration is more standard:

```
$ $SRC_PATH/fvm-0.15.3/configure \
--prefix=$INSTALL_PATH/arch/frontend \
--with-bft=$INSTALL_PATH/arch/frontend \
--with-hdf5=$CS_OPT/hdf5-1.8.6/arch/frontend \
--with-med=$CS_OPT/med-3.0/arch/frontend \
CC_ADD="-m64" \
LD_ADD="-m64"
```

10.6.4 MEI configuration

For the compute nodes, Python bindings are not useful:

```
$ $SRC_PATH/mei-1.0.2/configure \
--prefix=$INSTALL_PATH/arch/bgp \
--with-bft=$INSTALL_PATH/arch/bgp \
--disable-python-bindings \
--build=ppc64 \
--host=bluegenep \
CC=bgxlc
```

For the front-end nodes, only the Python bindings are generally useful. The kernel build should only be used for mesh verification (if built at all), so it will not need evaluation of mathematical expressions:

```
$ $SRC_PATH/mei-1.0.2/configure \
--prefix=$INSTALL_PATH/arch/bgp \
--with-bft=$INSTALL_PATH/arch/bgp \
--with-python-exec=$CS_OPT/python-2.7.1/arch/frontend/bin \
--with-swig-exec=$CS_OPT/swig-1.3.39/arch/frontend/bin \
```

10.6.5 Kernel configuration

Finally, when all other packages are installed, the Kernel may be configured. For the compute nodes, the following command may be used:

```
$ $SRC_PATH/ncs-2.0.2/configure \
--prefix=$INSTALL_PATH/arch/bgp_ompi \
--with-bft=$INSTALL_PATH/arch/bgp \
--with-fvm=$INSTALL_PATH/arch/bgp \
--with-mei=$INSTALL_PATH/arch/bgp \
--with-prepro=$INSTALL_PATH/arch/frontend \
--with-blas=/opt/ibmmath/essl/4.4 \
--with-libxml2=$CS_OPT/libxml2-2.3.32/arch/frontend \
--with-syrthes=/home/projects/syrthes3.4.3 \
--with-batch=LOADL_BG \
--disable-sockets \
--disable-dlloader \
--disable-nls \
--build=ppc64 \
--host=bluegenep \
--with-python-exec=$CS_OPT/python-2.7.1/arch/frontend/bin \
CC=mpixlc_r FC=bgxlf90_r
```

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.[6]

Note that the front-end version of the Preprocessor is used even for the compute node build, as only the Kernel itself will run on the latter. For the front-end nodes, the following command may be used:

```
$ $SRC_PATH/ncs-2.0.2/configure \
--prefix=$INSTALL_PATH/arch/bgp_ompi \
--with-bft=$INSTALL_PATH/arch/bgp \
--with-fvm=$INSTALL_PATH/arch/bgp \
--without-mei \
--with-prepro=$INSTALL_PATH/arch/frontend \
--with-python-exec=$CS_OPT/python-2.7.1/arch/frontend/bin \
CFLAGS="-m64" \
FCFLAGS="-m64" \
LDFLAGS="-m64"
```

[6] This might be due to a bug in the ESSL BLAS detection of *Code_Saturne*, although the code has been checked.