MACI User Guide

Dorian Krause[†] and Rolf Krause[◇]

[†]Institute of Computational Science
Università della Svizzera Italiana, Via Guiseppe Buffi 6, CH-6900 Lugano
e-mail: dorian.krause@usi.ch

[◇]Institute of Computational Science
Università della Svizzera Italiana, Via Guiseppe Buffi 6, CH-6900 Lugano
e-mail: rolf.krause@usi.ch

Last Update: May 17, 2013

1 Introduction

The Multiscale Atomistic Continuum Interface (MACI) is a generic C++ interface for the coupling of Molecular Dynamics and Finite Element codes. It's main purpose is to connect commodity MD and FE codes and realize the transfer of displacements, velocities and other physical quantities between the codes.

MACI is a fully parallel code based on the Message Passing Interface (MPI) standard. In fact, to run a MACI driven simulation, at least two MPI processes are required for the MD and the FE code.

2 Overview

The core MACI library is a C++ library. The functionality of the library is exported to Python using SWIG. On the Python level, an object-oriented interface is provided which allows to perform many actions on this level rather than modifying the core library.

As a third-level, the python module has been enhanced with XML parsing capabilities allowing users to specify the majority of simulation parameters in a portable human-readable format. The ability to parse XML files has been added late to the code. It is the authors experience that while scripting languages are well suited to describe the simulation flow, the inclusion of simulation parameters typically leads to bad maintainable scripts due to repeated changes (typically by commenting out parts), e.g. when a parameter space is to be explored. In our experience the separation of the simulation flow (in the form of Python modules and scripts) and simulation parameters (in the form of XML files) is a reasonable approach to reduce complexity in the scientific workflow. This is especially important in multiscale simulations where three different domain codes (MD, FE and coupling code) must be handled in a consistent way.

MACI implements the coupling method first introduced by Fackeldey et al.[1]. It implements Least-Squares and L^2 projection-based coupling. On the Python level a RATTLE time integration scheme and a multirate time integration scheme are implemented.

MACI uses a generic interface to communicate with the FE and MD code. This interface has been designed with some assumptions on the commodity codes but is generic enough to be implemented by many different codes. Currently, MACI can interface to Sandia's LAMMPS MD code, the Tremolo MD code developed at the University of Bonn and the UG Finite Element

code (with some restrictions). Additionally, MACI uses various third-party packages such as Trilinos, PETSc or UMFPACK to solve arising linear systems.

3 Installation

3.1 System Requirements

MACI is written in portable C++ and should be usable on any Unix flavor. MACI has been successfully compiled and run on Linux and Mac Os X.

3.2 Dependencies

MACI has the following dependencies on external software:

Name	Req	Description	Comment
cmake	Yes	Build System	cmake is used for the core library and many subprojects and external packages used in MACI.
SWIG	Yes	Creates Wrapper	Creates Python interface automatic from C++ code.
Python	Yes	Interpreter	Required to drive simulation.
wmpi	Yes	Wrapped MPI	A tiny Python module providing a limited set of MPI functionality on the scripting level. This module is part of the MACI repository.
spblas	Yes	Sparse BLAS	The current MACI version depends on Sparse BLAS. Unfortunately, there are few (the authors are not aware of any) standard conforming high-performance implementations available. In future versions, a Sparse BLAS implementation will be optional and other packages, such as Intel's MKL and Berkeley Oski will be usable.
Trilinos	No	Linear Algebra Package	Provides parallel solver routines for the computation of Lagrange multipliers.
PETSc	No	Linear Algebra Package	Provides parallel solver routines for the computation of Lagrange multipliers.
UMFPACK	No	Direct Solver	Provides solver routines for the computation of Lagrange multipliers. This solver can only be used if each connected component of the handshake region is fully contained in the domain of a single processor.

UG	Yes	FE code	Currently only the UG (or Obslib, resp.) code implements the interface to MACI. In fact, the FE interface of MACI is a copy of the UG interface implemented in the mscoupling code.
LAMMPS	No	MD code	Sandia's Open Source Molecular Dynamics Code. Very few changes to LAMMPS are necessary to allow for linking it with the MACI interface (see below).
Tremolo	No	MD code	Tremolo is developed at the University of Bonn by the Group of Prof. Griebel. It has been the only MD code usable with MACI for a long time. The interface has been developed during an internship of the first author at the JSC Jülich and has lower code quality than the majority of the MACI code. Many changes to the Tremolo code were necessary, rendering our Tremolo version incompatible with the official version. LAMMPS is now the recommended MD code to use with MACI but there are still features of the MD interface (such as stress calculations) which are only available with Tremolo.

Note that either Trilinos, PETSc or UMFPACK are required. It is possible to link against all of these libraries at the same time and chose the solver at runtime. Similarly, either LAMMPS, Tremolo or mdgpgpu are required for a successful build. However, MACI only supports compilation with exactly one of these codes. The same limitation applies to the FE code(s).

3.3 Component Locations

The different MACI components and dependencies can be retrieved as follows:

Name	Location	Description and Comments
cmake	http://www.cmake.org	It is recommended to use cmake version 2.8 or later.
SWIG	http://www.swig.org	Tested with version 1.3.40.
Python	http://www.python.org	Python is installed on most Linux systems. MACI has been used mainly with version 2.4.3.
wmpi	https://github.com/kraused/maci-code.git	
spblas	http://math.nist.gov/spblas	NIST provides an un-optimized reference implementation of Sparse-BLAS.
Trilinos	http://trilinos.sandia.gov	Trilinos is installed on many clusters and supercomputers.

PETSc	http://www.mcs.anl.gov/petsc	PETSc is installed on many clusters and supercomputers.
UG and mscoupling	Upon request	The mscoupling package is a wrapper around the UG datastructure which also implements linear elasticity and Cauchy-Born based elasticity. mscoupling is not compatible with the current trunk version of UG.
LAMMPS	http://lammps.sandia.gov	Small modifications are necessary to use LAMMPS with MACI (see below).
Tremolo	Upon request	The directory includes the modified Tremolo sources and the interface layer.
MACI	https://github.com/kraused/maci-code.git	

3.4 Shared Libraries

As explained earlier, MACI is driven by a collection of Python modules and Python scripts. A Python module is a dynamic shared object (DSO), i.e. a shared library on Linux systems. It is therefor important to build Python modules as shared libraries and all other (external) as position-independent code. For project components which use cmake as their build system, the flag

```
-DBUILD_SHARED_LIBS:BOOL=ON
```

to the cmake arguments. For other software (such as LAMMPS) it should be ensured that -fPIC is part of the compilation and linking flags. This ensures that position-independent code is created.

It should be pointed out that there are a couple of problems associated with the extensive usage of DSOs in scientific applications like MACI:

- There is limited or no support for DSOs on High-End Computing platforms (e.g., the Cray XT architecture). The authors decision to use the Python language was majorly influenced by the support of this scripting language on the IBM BlueGene architecture. However, MACI has not been tested on BlueGene.
- On heavily loaded systems, the execution time can grow substantially due to the need to load various DSOs.
- There is an increase in complexity associated with the management of many shared libraries.

3.5 Building LAMMPS

Here, we describe how to set-up LAMMPS such that it can be used with MACI.

1. Download a fresh LAMMPS version from the Sandia page.

2. The next step is to adapt one of the MAKE/Makefile.XYZ files. Here, we use Makefile.linux. For this purpose we apply the following patch (shorted for editorial reasons):

```
--- MAKE/Makefile.linux 2010-07-13 16:00:15.000000000 +0200
+++ MAKE/Makefile.linux 2010-07-29 22:13:39.000000000 +0200
@@ -6,10 +6,10 @@
-CC =            icc
-CCFLAGS =       -O
+CC =            mpicxx
+CCFLAGS =       -O -fPIC
 DEPFLAGS =      -M
-LINK =          icc
-LINKFLAGS =     -O
+LINK =          mpicxx
+LINKFLAGS =     -O -fPIC

@@ -29,18 +29,18 @@
-MPI_INC =       -DMPICH_SKIP_MPICXX
+MPI_INC =
 MPI_PATH =
-MPI_LIB =       -lmpich -lpthread
+MPI_LIB =

-FFT_INC =       -DFFT_FFTW
-FFT_PATH =
-FFT_LIB =       -lfftw
+FFT_INC = $(FFTW_POST_COMPILE_OPTS) -DFFT_FFTW
+FFT_PATH =
+FFT_LIB = $(FFTW_POST_LINK_OPTS)
```

   The necessary changes depend on the system configuration. Here, we choose the MPI compiler wrapper and use the environment variables FFTW_POST_XYZ_OPTS which are set by the fftw/2.1.5 module on this system. It is now possible to compile the LAMMPS libraries using

```
% make makelib
% make -f Makefile.lib linux
```

   However, compiling MACI with this LAMMPS version will not yet succeed.

3. The next step is to apply some modifications to the LAMMPS source code. First we must add a pointer for the piggyback data to the Atom class:

```
--- atom.h      2010-01-14 20:44:09.000000000 +0100
+++ atom.h      2010-07-29 22:37:11.000000000 +0200
@@ -105,6 +105,10 @@

+  /* ---------- MACI ---------- */
+  void* pb;
+  /* ---------- MACI ---------- */
+
   // functions

   Atom(class LAMMPS *);
```

   Moreover, we need to modify atom.cpp to add some callbacks.

```
--- atom.cpp    2010-06-18 21:49:12.000000000 +0200
+++ atom.cpp    2010-07-29 23:00:33.000000000 +0200
@@ -260,6 +260,11 @@
    generate an AtomVec class
 ------------------------------------------------------------------------- */

+/* ---------- MACI ---------- */
```

```
+AtomVec *(*NewAtomVecMultiscaleAtomicCB)(LAMMPS *, int, char **) = 0;
+AtomVec *(*NewAtomVecMultiscaleFullCB)(LAMMPS *, int, char **) = 0;
+/* ---------- MACI ---------- */
+
 AtomVec *Atom::new_avec(const char *style, int narg, char **arg)
 {
   if (0) return NULL;
@@ -270,6 +275,13 @@
 #include "style_atom.h"
 #undef ATOM_CLASS

+  /* ---------- MACI ---------- */
+  else if (0 == strcmp(style,"multiscale_atomic"))
+        return NewAtomVecMultiscaleAtomicCB(lmp, narg, arg);
+  else if (0 == strcmp(style,"multiscale_full"))
+        return NewAtomVecMultiscaleFullCB(lmp, narg, arg);
+  /* ---------- MACI ---------- */
+
   else error->all("Invalid␣atom␣style");
   return NULL;
 }
```

Similarly, `modify.cpp` must be modified:

```
--- modify.cpp  2010-07-16 01:23:37.000000000 +0200
+++ modify.cpp  2010-07-29 23:02:47.000000000 +0200
@@ -542,6 +542,11 @@
    add a new fix or replace one with same ID
 ------------------------------------------------------------------------- */

+/* ---------- MACI ---------- */
+Fix *(*NewRattleIntegratorCB)(LAMMPS *, int, char **) = 0;
+Fix *(*NewVerletIntegratorCB)(LAMMPS *, int, char **) = 0;
+/* ---------- MACI ---------- */
+
 void Modify::add_fix(int narg, char **arg)
 {
   if (domain->box_exist == 0)
@@ -598,6 +603,13 @@
 #include "style_fix.h"
 #undef FIX_CLASS

+/* ---------- MACI ---------- */
+  else if (0 == strcmp(arg[2],"multiscale_RattleIntegrator"))
+        fix[ifix] = NewRattleIntegratorCB(lmp, narg, arg);
+  else if (0 == strcmp(arg[2],"multiscale_VerletIntegrator"))
+        fix[ifix] = NewVerletIntegratorCB(lmp, narg, arg);
+/* ---------- MACI ---------- */
+
   else error->all("Invalid␣fix␣style");

   // if fix is new, set it's mask values and increment nfix
```

4. With these modifications it is possible to build `LAMMPS` and link with `MACI`. A patch for `LAMMPS` can also be found in the `MACI` code at `md/lammps/src/lammps-XYZ.patch`. However, typically such a patch will not be valid for a different `LAMMPS` version.

3.6 Manual installation with `cmake`

If all components have been build (and eventually installed) it is possible to configure and build `MACI`. The correct command line is

```
% cmake -DBUILD_SHARED_LIBS:BOOL=ON -DCMAKE_BUILD_TYPE:STRING=Debug \
        -DCMAKE_INSTALL_PREFIX:STRING=<INSTALL DIRECTORY> \
```

```
-DTrilinos_INCLUDE_DIR:STRING=$TRILINOS_PACKAGEROOT/include \
-DTrilinos_LIB_DIR:STRING=$TRILINOS_PACKAGEROOT/lib \
-DPETSc_INCLUDE_DIR:STRING=$PETSC_PACKAGEROOT/include \
-DPETSc_LIB_DIR:STRING=$PETSC_PACKAGEROOT/lib \
(add UG, Lammps, Sparse BLAS, UMFPACK etc.) \
<PATH TO MACI>
```

With this command a `Debug` version of the code is build (i.e. without optimization). For production runs, the `Release` build type should be used.

Packages will be added to the build if they are found by `cmake`. This requires the definition of `XYZ_INCLUDE_DIR` and `XYZ_LIB_DIR` as arguments to `cmake`.

To use `MACI` it **must** be installed to a second directory. The installation will create the folders `$CMAKE_INSTALL_PREFIX/bin`, `$CMAKE_INSTALL_PREFIX/lib/maci` and `$CMAKE_INSTALL_PREFIX/share/maci`. In the `bin` folder, the executable `maci.exe` and the wrapper `maci` are placed.

Using a build system like `cmake` to maintain the various components of a `MACI` installation can be tedious, especially if different code versions are to be maintained at the same time. Therefore we have developed `build.py`, a small application, which allows for maintaining and building `MACI` and its components.

3.7 Simplified Installation with `build.py`

The `Python` script `build.py` should simplify the task of maintaining a modular code basis. It is very generic and can be also used to maintain other software than `MACI` as long as this software is `cmake` based. `build.py` takes a high-level `XML` description of the code as input. The `XML` file describes the packages and dependencies. It also allows to include external packages.

The following `XML` file describes a project containing three packages `A`, `B` and `C`, where `A` and `B` are `cmake` project and `C` is an external project with a module file. In this example `A` depends on `B` and `C`. We see that `build.py` supports the module evironment and can read environment variables.

```
<config>
        <!-- Name of the configuration to distinguish build directories. If the
             name contains slashes a directory structure will be created -->
        <name>
                A_with_B/also_with_C
        </name>
        <!-- Release or Debug configurations. This global value can be
             overwritten by the individual projects -->
        <type>
                Debug
        </type>
        <!-- On systems using the modules environment system these modules
             (and only these) are guaranteed to be loaded -->
        <modules>
                C/2.0.1
        </modules>
        <!-- Additional options can be defined which are passed to the cmake
             program -->
        <option>
                <key>
                        BUILD_SHARED_LIBS:BOOL
                </key>
                <val>
                        ON
                </val>
        </option>
```

```
<!-- Here comes the list of packages to be managed. Note that the order
     of the packages matters. They are processed from top to bottom.
     Cyclic dependencies are not possible. -->
<packages>
        <package>
                <name>
                        B
                </name>
                <!-- The source folder of the package. All paths must be
                     absolut. -->
                <dir>
                        ${path_to_B}
                </dir>
        </package>
        <package>
                <name>
                        A
                </name>
                <!-- The source folder of the package. All paths must be
                     absolut. -->
                <dir>
                        ${path_to_A}
                </dir>
                <dependency>
                        <name>
                                B
                        </name>
                </dependency>
                <dependency>
                        <name>
                                C
                        </name>
                        <include>
                                ${C_PACKAGEROOT}/include
                        </include>
                        <lib>
                                ${C_PACKAGEROOT}/lib
                        </lib>
                </dependency>
        </package>
```

`build.py` assumes that each package provides `INSTALL()` commands which copy all include files to the folder `$CMAKE_INSTALL_PREFIX/include` and all libraries to `$CMAKE_INSTALL_PREFIX/lib` (this is not necessary for projects on which no other project depend). It installs the package in the local folder, i.e. `c` would be installed in `A_with_B/also_with_C/C`.

It should be noted that `build.py` is a rather small program and only implements those features required for a successful build of `MACI`. For example, it does not support individual option lines for the packages. However, it is easily possible to enhance `build.py` for this purpose.

We collect configuration files for different hosts in the `config` folder in the `MACI` repository. These files can serve as guidelines for creating configuration for new hosts. The most recent versions for each host should be collected in this directory.

4 Running `MACI`

Once `MACI` has been compiled and installed it can be run like most other `MPI`-based applications:

```
mpiexec -np 32 -hostfile $PBS_NODEFILE <PATH TO MACI>/bin/maci [args] \
                              <PYTHON INPUT> <XML SPEC FILE>
```

The details of the MPI start up mechanism will depend on the system. Note that the XML spec file is only necessary if the the Python script relies on the high-level Python library. This is the recommended usage of MACI.

Currently, the only argument MACI takes is `--input_path "<PATH␣A>:<PATH␣B>"`, which defines a list of files where MACI searches for input (XML files, Python files, etc.). This allows to distribute the input data over different directories, which can be helpful sometimes. The option parser is based on the optparse module (which is now deprecated it will need to be replaced in the future).

The maci program will take care of the initialization of environment variables (such as LD_LIBRARY_PATH and the PYTHONPATH) for all components of the installation, except for external packages (like Trilinos or PETSc). It is therefore **not** possible to relocate the MACI tree after installation. Since a clean build of MACI requires only a few minutes we believe that the increased comfort of this approach (since there is no need to set LD_LIBRARY_PATH or PYTHONPATH manually) justifies this loss of flexibility.

5 Description of the Python interface

As mentioned earlier MACI is run by a combination of Python scripts and XML specification files. In this section, we describe the MACI core library implemented in Python.

The MACI core library is available to user scripts by the name maci. The module **must not** be imported explicitly. maci as well as the mpi module (which is implemented in wmpi) are preloaded by the MACI executable.

5.1 Running MD and FE simulations

The following example shows how we can easily run a molecular dynamics simulation using the MACI interface:

```
here = maci.ProcElement(mpi.world, maci.SPECFILE)
tk   = maci.MDToolkit  (mpi.world, maci.SPECFILE)
here.init(tk)

tk.init()
tk.createInitialWave()

verlet = maci.VerletIntegrator(here, tk, maci.SPECFILE)
verlet.run()
```

The script starts by creating an instance here of type maci.ProcElement. The constructor of maci.ProcElement takes two arguments: First, it requires a communicator which normally will be mpi.world (the Python-level equivalent to MPI_COMM_WORLD). The second argument is the specification file maci.SPECFILE which is passed to the executable. The maci.ProcElement collects information about the processing element such as it's task (whether it runs MD or FE).

The next step is to set up a "toolkit". Here, we run a pure Molecular Dynamics simulation and hence create an instance of maci.MDToolkit on all processing elements.

Once we have created the instances here and tk we need to initialize both. Also, we create an initial wave as starting conditions.

Lastly, we create an instance of `maci.VerletIntegrator` to perform the time integration and call its `run()` member function which performs the time integration.

This small examples shows that only a couple of lines are required to run a (parallel) simulation with `MACI`. Of course, complexity cannot be completely hidden in the core library. For this example, the `XML` input file (which we discuss in the next section) must specify the missing parameters for the simulation.

In order to run the same wave propagation example with Finite Elements, it is only necessary to replace `maci.MDToolkit` by `maci.FEToolkit`.

5.2 Running Coupled Simulations

The following example (taken from the `CRACK` regression test in `MACI`) shows an example of a coupled simulation:

```
# Boundary forces. This is a bit tedious
# since they cannot be defined "generic"
# (without knowing the size of the grid)
def g(x,y,z):
        if x > 278.5:
                return [ +0.5, 0.0, 0.0 ]
        if x < -0.5:
                return [ -0.5, 0.0, 0.0 ]
        return [ 0.0, 0.0, 0.0 ]


here = maci.ProcElement(mpi.world, maci.SPECFILE)
if maci.ProcMD == here.mytype():
        tk = maci.MDToolkit(here.comm, maci.SPECFILE)
        here.init(tk)
else:
        tk = maci.FEToolkit(here.comm, maci.SPECFILE)
        here.init(tk)

hs = maci.HandshakeGeoDescr(here, maci.SPECFILE)

if maci.ProcMD == here.mytype():
        tk.init()
if maci.ProcFE == here.mytype():
        tk.init(hs.weight)
        tk.addSurfaceForce(g)


rattle = maci.RattleIntegrator(here, hs, tk, maci.SPECFILE)
rattle.run()
```

As in the previous example, we start by instantiating `maci.ProcElement`. Depending on the type of the processing element (which can be queried with the `mytype()` member function), we create an instance of `maci.MDToolkit` or `maci.FEToolkit`. The decision whether a processing element is of type `maci.ProcMD` or `maci.ProcFE` is guided by the `XML` input file.

In the next step, before initializing `tk` we create an instance of `maci.HandshakeGeoDescr` which (as the name suggests) is a description of the handshake geometry. More precisely, `maci.HandshakeGeoDescr` stores all information about the geometric primitives which constitute the handshake region (currently we only support cuboidal primitives) as well as the

value of the weighting function at the boundary (so that it can be interpolated for interior points). It is important to notice that `hs.weight` differs on MD and FE processing elements, i.e. `maci.HandshakeGeoDescr` computes the correct (consistent) weighting depending on the task of the processing element.

Lastly, we initialize the toolkits (note that the weight must be passed to the `maci.FEToolkit` instance, since the weighting must be known for assembling mass matrices, etc.), create an instance of `maci.RattleIntegrator` and call the `run()` routine. As the name suggests, implements `maci.RattleIntegrator` the RATTLE time integrator for Differential Algebraic Equations (DAEs).

In this example, we also define a surface force term for the FE equation of motions. In the current version of `MACI`, this is done by defining a function `g` which takes three coordinate values as arguments and returns the applied force. The function can be registered with the `addSurfaceForce()` function.

Currently, surface points cannot be identified by means other than their spatial coordinates. This means that it is not possible to transfer the definition of `g` to another geometry without modifying the values (`278.5` and `-0.5`) which define the boundary, to which the external force should be applied.

The classes `maci.ProcElement`, `maci.HandshakeGeoDescr` and `maci.RattleIntegrator` are implemented in the core library based on the `MACI C++` library. They export only a small number of functions that should be used by users (basically, the two examples depicted above cover all the functionality that is needed in the first place) and advanced users will likely need to implement new functionality in the `MACI` core library. In **??**, we will discuss the structure of the code and how to add new functionality.

6 Description of the `XML` Format

The core `Python` library of consists of a collection of `Python` classes, the behavior of which is completely determined by the `XML` input file. As we have seen in the last section it is possible with `MACI` to implement a solver for the multiscale DAE in only a few lines. A big portion of the complexity of the problem is hidden in the `C++` and `Python` core library, but another part can be found in the `XML` specification files.

The rational forsplitting of the `MACI` input into `Python` and `XML` code is that, while `Python` is a good way to specify "what to do', `XML` is much better in specifying parameters for the execution. Unfortunately, it is not possible to draw a clear line between functionality provided on the `Python` level and in the `XML` files. For example, while one might argue that the choice of the projection operator used to evaluate constraints should be done on the `Python` level, we consider this choice as a parameter for the e.g. the `maci.RattleIntegrator`.

Each `MACI` specfile begins with the `<simulation>` node:

```
<simulation dim = "X">
        ...
</simulation>
```

The dim attribute is mandatory. It is used by MACI to determine which libraries to load at startup. In the next subsections we will go through the different child nodes of <simulation> that must be defined by the user. Each node corresponds to a class in the Python core library of MACI.

One important feature of the XML parser in MACI (which is based on the xml.dom.minidom Python module) is that XML files are preprocessed. Since the XML standard does not include the possibility to include other XML files, this must be done prior to parsing the XML content. The syntax for including other XML files is taken from the C preprocessor:

```
<simulation dim = "X">
        ...

        #include "handshake.xml"
</simulation>
```

7 Regression Testing Framework

The first version of MACI has been developed together with a suite of unit tests. However, as the complexity of the code and the relations between classes increased (in retrospective one can argue that this was mainly caused by design flaws) unit testing was abandoned. Since then, MACI was developed without automated testing.

Recently, we have added a suite of regression tests to MACI. The execution of the tests is driven by a small Python library which is controlled by a set of input files in JSON format. Each test is a JSON object and information about the host and the environment are read in from additional JSON files allowing us to test different build configurations. This testing library was designed particular for parallel code like MACI and it assumes that the execution host runs a batch system. Testing results are gathered in a report and can be send via e-mail to a responsible person.

Currently the test system is limited to detecting whether a run passes or fails during execution. The ability to check for correctness of output results will be added in future extensions of the system.

7.1 Running the test suite

The regression tests are collected in the tests folder in the MACI trunk. The script testing.py is responsible for execution of the tests:

```
./testing.py example_env.json
```

It searches through the input directory (specified in the input file) for files named tests.json. These files are read and for each combination of tests and number of processing elements it submits a job to the batch system. For a user specified amount of time it continuously checks the status of the submitted jobs. If either all jobs have finished or if the maximum time for testing (specified in the input file) is exceeded it aborts all unfinished jobs, creates a report and submits this report to the responsible person via e-mail.

7.2 Description of the environment in JSON

In the input file to testing.py, we define the "environment", i.e. the executable to run, the input and output folders, which modules should be loaded before running the code and how to run parallel jobs with MPI. Below, an example for such a configuration file is given:

```
{
  "type" : "env",

  "name" : "Name␣of␣the␣environment/what␣we␣are␣doing",

  # The host configuration containing details about the
  # execution environment
  "host" : "/home/user/maci-code/tests/hosts/cub.json",

  # The mail address of the responsible for these tests.
  # Test results will be mailed to this address.
  "responsible" : "dorian.krause@usi.ch",

  # The input  test directory
  "input" : "/home/user/maci-code/tests",

  # The output test directory. This directory name
  # depends on the time of execution.
  "output": "/home/user/out.%a-%b-%y-%H-%M",

  # The MACI executable
  "exe" : "/home/user/build/maci/lammps/debug/maci/bin/maci",

  # The configuration used for building the executable. The individual
  # items in the list can be used as "prereq" for tests so that tests
  # which do not run with this configuration can be excluded.
  "config" : [ "gnu", "open␣mpi", "lammps", "ug", "trilinos", "petsc", "spblas" ],

  # The modules to load before the executable can run
  "modules" : [ "trilinos", "fftw/2.1.5" ],

  # The maximal time (in minutes) to wait for completion of the test jobs before
  # cancelling
  "max_walltime" : 120,

  # MPI infos. The MPI version should be in synch
  # with the executable
  "mpi" :
  {
    "type" : "open␣mpi",

    # How to execute jobs
    "cmd" : "mpiexec␣-np␣%npes␣-hostfile␣$PBS_NODEFILE␣-x␣OMP_NUM_THREADS=1␣%exe␣
        %args"
  }
}
```

The output and cmd strings are processed by testing.py. To allow for time-depenent output folders, the variables %a, %b, %y, %H and %M are replaced by the name of the current day, current month, the year, the hour and the minute, resp. Similar, in the MPI cmd string, %npes is replaced by the number of processing elements requested (note that currently there is no support for hybrid execution), %exe is replaced by the name of the exectutable and %args by the input arguments to the executable.

It is important to note that the value max_walltime is not the maximum walltime for the individual jobs but for the whole testing process. It should be chosen larger than the execution plus the anticipated queueing time for the jobs.

7.3 Description of a host in JSON

In the host file, all information about the execution host are collected. An example is shown below:

```
{
  "type"  : "host",

  # The identifier for the host
  "host"  : "cub.inf.usi.ch",

  # The batch system on the host
  "batch" :
  {
    "type" : "pbs",

    # Define the executable interface
    "sub"  : "qsub␣
        -lnodes=$(max(1,int(%npe/8))):ppn=$(min(%npe,8)),vmem=$(2*min(%npe,8))gb,walltime=%wtime␣
        -joe␣-N␣%name␣%script",
    "del"  : "qdel␣␣%jobid",
    "stat" : "qstat␣%jobid"
  },

  # The command used to send mails
  "sendmail" : "mail␣-s␣\"%subject\"␣%recipient␣<␣%file"
}
```

Note that the definition of the `sub` member of `batch` is allowed to contain simple formulas which are evaluated by the testing library. Each expression between `$(` and the matching `)` will be evaluated by the python parser. This ability is currently only enabled for the `sub` string but can be easily applied to other strings, too.

7.4 Description of a test in JSON

Tests are defined in `JSON` files with the name `tests.json`. In each file, multiple tests can be defined. Each test is it's own `JSON` object:

```
{
  "name" : "Unique_name_for_test",
  "descr" : "Short␣human␣readable␣description",
  "npes" : [ 1, 8 ],
  "arguments" : "--input_path␣\"%pwd:%pwd/XYZ\"␣in.py␣in2.xml",
  "output" : [ "out.dat", "other_out.dat" ],
  "prereq" : [ "lammps", "ug", "trilinos" ]
}
```

Each `JSON` test object must have a unique name. For each number in the `npes` array, one job with the specified number of processing elements is submitted. While the executable is specified in the testing environment, each test can define its own arguments. The string is processed and the variable `%pwd` is replaced by the folder containing the `tests.json` file. The `out` member is currently unused but is intended to be used for `diff`'ing with reference output. The members of the `prereq` array are compared to the `config` array in the environment definition and only if all prerequisites are fulfilled the test will run.

References

1. Konstantin Fackeldey, Dorian Krause, Rolf Krause, and Christoph Lenzen. Coupling molecular dynamics and continua with weak constraints. *Multiscale Modeling & Simulation*, 9(4): 1459 – 1494, 2011.