Installing and compiling Albany/LCM and Trilinos on Fedora or SEMS/CEE

The Albany/LCM team*

May 24, 2017

1 Introduction

This document describes the necessary steps to install Albany/LCM and Trilinos on a machine with Fedora Linux or the Linux CEE/SEMS environment. If you want a shortcut, obtain the script `install_albany.sh` (stored in `Albany/doc/LCM/install`) and then try

```
./install_albany
```

This will install and build Trilinos and Albany in the current directory. If the script does not complete it will tell you why and with help from this document you will be able to complete the install.

2 Required Packages

The following packages should be installed using the `dnf` command

```
blas
blas-devel
boost
boost-devel
boost-openmpi
boost-openmpi-devel
boost-static
cmake
hdf5
hdf5-devel
hdf5-openmpi
hdf5-openmpi-devel
hdf5-static
lapack
lapack-devel
matio
matio-devel
netcdf
netcdf-devel
netcdf-openmpi
netcdf-openmpi-devel
netcdf-static
openmpi
openmpi-devel
```

*Original version by Julián Rímoli

```
yaml-cpp
yaml-cpp-devel
environment-modules
gcc-c++
git
```

For example, to install the first package you should type

```
sudo dnf install blas
```

Make sure that all these packages are installed, specially if you create a script to do so. If a package is not installed because of a typo then the compilation will fail.

It may be necessary to logout and login for the module alias from the `environment-modules` package to become active.

Optional but strongly recommended packages:

```
clang
clang-devel
gitk
```

3 Repository Setup with GitHub

In a web browser go to `www.github.com`, create an account and set up ssh public keys. If you require push privilges for Albany, email Glen Hansen at `gahanse@sandia.gov` and let him know that. On the other hand, if you require push privileges for Trilinos, it is best if you contact the Trilinos developers directly. Go to `www.trilinos.org` for more information.

It is strongly recommended that you join the `AlbanyLCM` Google group to receive commit notices. Go to `groups.google.com/forum/#!forum/albanylcm` and join the group. You can also browse the source code at `github.com/gahansen/Albany`.

4 Directory Structure

In your home directory, create a directory with the name `LCM`:

```
mkdir LCM
```

Change directory to the newly created one:

```
cd LCM
```

Check out the latest version of Trilinos, which is hosted now on GitHub:

```
git clone git@github.com:trilinos/Trilinos.git Trilinos
```

Finally, check out the latest version of Albany:

```
git clone git@github.com:gahansen/Albany.git Albany
```

At this point, the directory structure should look like this:

```
LCM
  |- Albany
  |- Trilinos
```

5 Environment Variables

In `~/.bashrc`, the following variables are needed:

```
export LCM_DIR=~/LCM
export MODULEPATH=$LCM_DIR/Albany/doc/LCM/modulefiles
```

The `LCM_DIR` variable should contain the location of the top-level `LCM` directory.

6 Build Script

Create a symbolic link to the build script `LCM/Albany/doc/LCM/build/build.sh` to the top-level `LCM` directory. Make sure that the build script is executable and read only:

```
cd ~/LCM
chmod 0555 build.sh
```

The `build.sh` script performs different actions according to the name with which it is invoked. This is accomplshed by creating symlinks to `build.sh` and using them to run it. For example:

- `clean.sh` will delete all traces of the corresponding build and will create a new configuration script based on the corresponding template.
- `config.sh` will attempt to configure the build.
- `build.sh` (original name) will build using cmake.
- `test.sh` will run the cmake tests.
- `update.sh` will execute git pull in the package repository, and if combined with dash below, it will send a report of changed files to CDash.
- `dash.sh` will post the results of ctest to configured CDash site.
- symlinks with combinations of the above (e.g. `clean-config-build.sh`) will perform the specified actions in sequence. See `build.sh` for valid sequences.

For example, the following symbolic links will create separate commands for clean up, configuring and testing:

```
ln -s build.sh clean.sh
ln -s build.sh config.sh
ln -s build.sh test.sh
```

They could also be combined for convenience:

```
ln -s build.sh clean-config.sh
ln -s build.sh clean-config-build.sh
ln -s build.sh clean-config-build-test.sh
ln -s build.sh config-build.sh
ln -s build.sh config-build-test.sh
```

There is also a script `LCM/Albany/doc/LCM/install/albany-lcm-symlinks.sh` that will create the appropriate symbolic links.

The build system is based on CMake. Thus the ouput verbosity level can be controlled by passing `-V` or `-VV` as a final option to `build.sh` or its aliases.

7 Parallel Schwarz and DTK

This section applies only if using the Schwarz alternating method in parallel by means of the Data Transfer Kit (`DTK`). Otherwise it can be safely ignored.

The current parallel implementation of the Schwarz method requires `DTK`, which is tightly integrated to Trilinos, specifically `STK`. Go to the the top-level `LCM` directory and create a symbolic link to the `DTK` CMake fragment that resides in `LCM/Albany/doc/LCM/build`, then download the `DTK` package from:

`https://github.com/ORNL-CEES/DataTransferKit/archive/2.0.0.tar.gz`.

Expand the package inside the Trilinos directory and rename it:

```
cd ~/LCM
ln -s Albany/doc/LCM/build/dtk-frag.sh .
cd Trilinos
curl -L https://github.com/ORNL-CEES/DataTransferKit/archive/2.0.0.tar.gz -O
tar zxf 2.0.0.tar.gz
mv DataTransferKit-2.0.0 DataTransferKit
```

The configuration scripts will detect the presence of `DTK` and configure it appropriately. Also, parallel Schwarz will be enabled when compiling Albany.

8 Modules

Modules are used to create different environments for the configuration and compilation of both Albany and Trilinos. To see the available modules that correspond to different thread models, compilers and build types:

```
module avail
```

This results in something like:

```
------------------ /home/amota/LCM/Albany/doc/LCM/modulefiles ------------------
cuda-gcc-debug       lcm/sems             pthreads-gcc-small
cuda-gcc-mixed       lcm/serial           release
cuda-gcc-profile     lcm/small            serial-clang-debug
cuda-gcc-release     lcm/tpls             serial-clang-mixed
cuda-gcc-small       mixed                serial-clang-profile
debug                openmp-gcc-debug     serial-clang-release
lcm/cee              openmp-gcc-mixed     serial-clang-small
lcm/clang            openmp-gcc-profile   serial-gcc-debug
lcm/common           openmp-gcc-release   serial-gcc-mixed
lcm/cuda             openmp-gcc-small     serial-gcc-profile
lcm/debug            openmp-intel-debug   serial-gcc-release
lcm/fedora           openmp-intel-mixed   serial-gcc-small
lcm/finalize         openmp-intel-profile serial-intel-debug
lcm/gcc              openmp-intel-release serial-intel-mixed
lcm/intel            openmp-intel-small   serial-intel-profile
lcm/mixed            profile              serial-intel-release
lcm/openmp           pthreads-gcc-debug   serial-intel-small
lcm/profile          pthreads-gcc-mixed   small
lcm/pthreads         pthreads-gcc-profile
lcm/release          pthreads-gcc-release
```

The naming convention for the `*-*-*` modules follows the pattern

```
[thread model]-[toolchain]-[build type]
```

The `[thread model]` option refers to the thread parallelism model that the code will use by means of the Kokkos package in Trilinos. Currently the sopported models are: `serial` that works for all supported compilers, `openmp` that works with the `GCC` and `Intel` toolchains, `pthreads` that works for all supported compilers, and `cuda` that is only supported for the `GCC` toolchain. The installation and configuration of the Cuda framework is complex. Much more detailed information can be found at `http://developer.nvidia.com/cuda`.

Currently the options for `[toolchain]` are `gcc`, `clang` and `intel` if the Intel compilers are installed, and for `[build type]` are `debug` (includes symbolic information), `release` (optimization enabled), `profile` (symbolic in-

formation and optimization enabled for profiling), `small` (minimizes size of executables) and `mixed` (TRILINOS compiled in release mode and ALBANY compiled in debug mode). The `clang` toolchain requires installation of the `clang` and `clang-devel` packages. The `debug`, `release`, `profile`, `small` and `mixed` modules are convenience aliases for `serial-gcc-debug`, `serial-gcc-release`, `serial-gcc-profile`, `serial-gcc-small` and `serial-gcc-mixed` modules, respectively.

Build directories are created within the LCM top-level directory and named according to the loaded module and package specified to the build.sh script, e.g.:

```
albany-build-gcc-release
```

In addition, for TRILINOS an install directory similarly named is created at the LCM top-level directory.

9 Configuring and compiling

Assuming that we want to compile with a `serial` thread model using the `gcc` tool chain in `debug` mode, load the appropriate module:

```
module load lcm/fedora
module load serial-gcc-debug
```

For the SEMS/CEE environment, this would be:

```
module load lcm/sems
module load serial-gcc-debug
```

Now first configure and compile TRILINOS. Within the top-level `LCM` directory type:

```
./config-build.sh trilinos [# processors]
```

For example, if you want to build using 16 processors, type:

```
./config-build.sh trilinos 16
```

Finally, repeat the procedure for ALBANY:

```
./config-build.sh albany [# processors]
```

For example, if you want to build a version of the code using 16 processors, type:

```
./config-build.sh albany 16
```

Note that to compile a version of ALBANY with a specific thread model, toolchain and build type, the corresponding version of TRILINOS must exist.

10 After Initial Setup

The procedure described above configures and compiles the code. From now on, configuration is no longer required so you can rebuild the code after any modification by simply using the `build.sh` script. For example:

```
./build.sh albany 16
```

There are times when it is necessary to reconfigure, for example when adding or deleting files under the `LCM/Albany/src/LCM` directory. This is generally anounced in the commit notices.

Also, note that both TRILINOS and ALBANY are heavily templetized C++ codes. Building the debug version of ALBANY requires large amounts of memory because of the huge size of the symbolic information required for debugging. Thus, if the compiling procedure stalls, try reducing the number of processors.

11 Running and Debugging LCM

After building Albany, you might want to run and/or debug the code. Tools were built in Trilinos (decomp, epu, etc.) that are necessary for parallel execution. The environment created by loading the appropriate module sets the proper paths so that the executables that correspond to the type of build are accessible.

12 Commiting Changes and Code Style

Albany is a simulation code for researchers by researchers. As such, vibrant development of new and exciting capabilities is strongly encouraged. For these reasons, don't be afraid to commit changes to the master git repository. We only ask that you don't break compilation or testing. So please make sure that the tests pass before you commit changes. Also, follow the development discussion here:

```
https://github.com/gahansen/Albany/issues
```

In addition, within LCM we strongly encourage you to follow the C++ Google style guide that can be found at `http://google-styleguide.googlecode.com/svn/trunk/cppguide.html`. This style is somewhat different to what is currently used in the rest of Albany, but we believe that the Google style is better in that it advocates more style differentiation between the different syntactic elements of C++. This in turn makes reading code easier and helps to avoid coding errors.

The `clang-format` tool can be used for this. There is a `.clang-format` file in the `Albany/src/LCM` directory that conforms to the C++ Google conding standard. Thus, all that is needed to reformat a source file in place is the command:

```
clang-format -i [source file name]
```