ALPSCore tutorial

Alexander Gaenko

June 15, 2016

INTRODUCTION

What it is about

ALPSCore originated from Algorithms and Libraries for Physics Simulations (ALPS) `http://alps.comp-phys.org`

The grand idea: Make the library code from ALPS available with shorter development cycle and decent documentation.

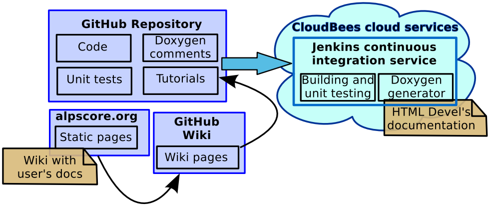
- What is the library code?
  - That will most probably useful for many applications.
  - However, ALPSCore is oriented towards Monte Carlo.
- What is the development cycle?
  - Introducing features (by request?)
  - Testing the features
  - Fixing bugs
  - Documenting
- What is decent documentation?
  - User's documentations
  - Tutorials (like this one!)
  - Developer's Doxygen-generated reference.

Web sites

Contributors:

- Emanuel Gull's group, University of Michigan (USA);
- Lukas Gamper, ETH Zurich (Switzerland);
- ... and many other ALPS contributors.

- Source code: `https://github.com/ALPSCore/ALPSCore`
- Documentation & tutorials: `http://alpscore.org`
- CloudBees for Continuous Delivery.



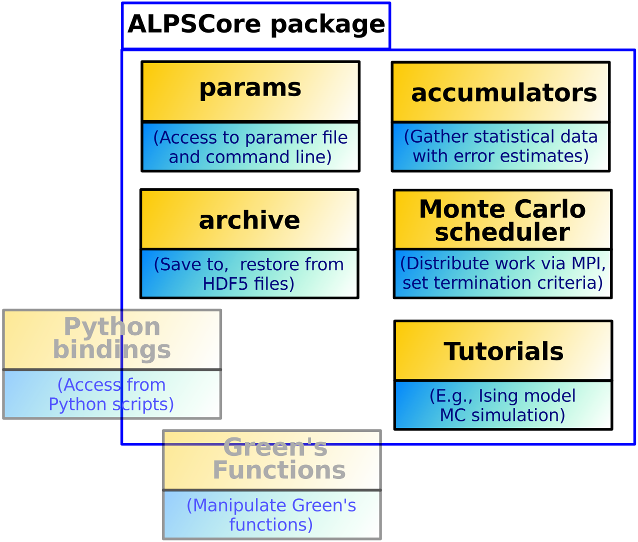
Why to use ALPSCore?

Think of a typical MC simulation to-do:

1. Read the simulation parameters.
2. Decide the step in the phase space.
3. Calculate ("measure") the values of interest.
4. Collect statistics properly, taking into account autocorrelation.
5. Compute derived values with proper error propagation.
6. Save intermediate results regularly.
7. Set the step or time limit on the simulation.
8. Parallelize the whole thing.

Except the highlighted items, everything else is "boilerplate". Sometimes non-trivial one!
Use ALPSCore to minimize boilerplate programming and concentrate on relevant science.

Overview


ALPSCore package
params
(Access to paramer file and command line)
accumulators
(Gather statistical data with error estimates)
archive
(Save to, restore from HDF5 files)
Monte Carlo scheduler
(Distribute work via MPI, set termination criteria)
Python bindings
(Access from Python scripts)
Tutorials
(E.g., Ising model MC simulation)
Green's Functions
(Manipulate Green's functions)

INSTALLING ALPSCore

Tutorial prerequisites

What do you need to *know?*

1. Basic Unix command-line operations
2. Basic C++
3. Optional: git
4. Optional: installing packages for your OS
5. Optional: CMake

What do you need to *have?*

1. Git
2. CMake
3. C++ compiler
4. HDF5 package
5. Boost package
6. MPI package

Obtainig ALPSCore

How to get ALPSCore?

- Downloading release:

`https://github.com/ALPSCore/ALPSCore/releases`

- Getting from GitHub:

`git clone https://github.com/ALPSCore/ALPSCore.git`

- Installing an ALPSCore package:
  - Via `macports` or `homebrew` on MacOS X.
  - Via `portage` on Gentoo Linux.
  - Debian or RedHat binary packages may be created if demand arises.

Building and compilation

There are (of course!) prerequisites.

- Tools:
  - C++03 (not `C++1x` — by policy).
    Tested with GCC 4.2+, Intel 10.0+, Clang 3.2+
  - CMake 2.8.12+
- Libraries:
  - HDF5 1.8+
  - Boost 1.54.0+
  - MPI (technically, optional)
  - GoogleTest (included!)

Building and compilation

- ALPSCore will try to find Boost, HDF5 and MPI.
- Usually, if you can run `mpicc`, ALPSCore will also find MPI.
- Building using CMake command line:

```
$ cmake -DBOOST_ROOT=/path/to/boost \
        -DHDF5_ROOT=/path/to/hdf5 \
        -DCMAKE_INSTALL_PREFIX=/usr/local/ALPSCore \
        /path/to/alpscore/sources
$ make
$ make install
```

- There are some other CMake variables, less frequently used

Exercise 1

Exercise 1: Download and install this tutorial

Set the correct URL

```
$ cd ~
$ mkdir alpstut
$ cd alpstut
$ git clone https://github.com/ALPSCore/tutorial2
$ cd tutorial2
$ tutorial=$PWD
$ ls -l
```

You should see a list of files and no error messages.

Exercise 2

Exercise 2: Download/install prerequisites

Ubuntu Linux

```
$ sudo apt-get install cmake
$ sudo apt-get install libhdf5-dev
$ sudo apt-get install libboost-all-dev
$ sudo apt-get install mpi-default-dev
```

Mac OS X, port system FIXME

```
$ sudo port install gcc???
$ sudo port install cmake
$ sudo port install libhdf5-dev???
$ sudo port install libboost-all-dev???
$ sudo port install mpi-default-dev???
```

Exercise 2

Exercise 2 (cont): Download/install prerequisites

Test that you indeed have them:

```
$ cmake --version
$ g++ --version
$ h5cc --version
$ mpicxx --version
```

Exercise 3

Exercise 3: Download and install ALPSCore.

```
1 $ git https://github.com/ALPSCore/ALPSCore
2 $ cd ALPSCore
3 $ mkdir build
4 $ cd build
5 $ export ALPSCore_DIR=$PWD/install
6 $ cmake -DCMAKE_INSTALL_PREFIX=$ALPSCore_DIR ..
7 $ make
8 $ make test
9 $ make install
```

1: get ALPSCore from GitHub repository.

2–4: create a directory for the build

5: denote where it will be installed

6: generate the build

7–9: do the build, run the tests, and install

USING ALPSCore

Using ALPSCore in your program

Note: `ALPSCore_DIR` is pointing to the installation directory.
FIXME: change, shorten URLs

- How `CMakeLists.txt` should look to use `ALPSCore`: see `https://github.com/galexv/ALPSCore_tutorial/blob/master/step1.trivial/CMakeLists.txt`
- Catch: compilers!
- In-source builds are messy

```
$ export CXX=$(which needed_cpp)
$ export CC=$(which needed_cc)
$ mkdir 000build
$ cd 000build
$ cmake ..
```

Using ALPSCore in your program

A few possible catches:

- Catch: if compilers change:
  `$ rm -rf CMake*`
- Catch: if CMake files change:
  `$ cmake .`
  in the build directory
- Catch: after accidental in-source build, remove the generated files.

Exercise 4

Exercise 4: Build and run a dummy program that uses ALPSCore and does nothing.

FIXME: shorten URLs

The code is at `$tutorial/step1_trivial`.

CMake file online: `https://https://github.com/ALPSCore/tutorial2/blob/master/step1_trivial/CMakeLists.txt`

Source file online: `https://https://github.com/ALPSCore/tutorial2/blob/master/step1_trivial/main.cpp`

Exercise 4

Exercise 4: Build and run a dummy program that uses ALPSCore and does nothing.

FIXME: shorten URLs

The code is at `$tutorial/step1_trivial`.

CMake file online: `https://https://github.com/ALPSCore/tutorial2/blob/master/step1_trivial/CMakeLists.txt`

Source file online: `https://https://github.com/ALPSCore/tutorial2/blob/master/step1_trivial/main.cpp`

```
$ cd $tutorial/step1_trivial
$ mkdir 000build
$ cd 000build
$ cmake ..
$ make
$ ./alpsdemo
```

Parameters

- `alps::params` class is responsible for parameter parsing.
- `boost::program_options` is the engine.
- See Doxygen documentation (link from `http://alpscore.org/`) for detailed info.

Features:

- One can use input file, override with command line.
- Input file may contain sections.
- Parameters must be defined to make it known.
- Unknown parameters in silently ignored.
- Auto-generated help message.
- Accessing an undefined parameter throws an exception.
- You can assign to parameters, which makes them defined.
- Potential information loss $\Rightarrow$ exception.

Exercise 5

Exercise 5: Build and run a program that uses parameters.

FIXME: shorten URLs

The code is at `$tutorial/step2_params`.

CMake file online: `https://https://github.com/ALPSCore/tutorial2/blob/master/step2_params/CMakeLists.txt`

Source file online: `https://https://github.com/ALPSCore/tutorial2/blob/master/step2_params/main.cpp`

- Play with the different values of parameters.
- Try to override them form the command line.
- Change the program to make `--loud` parameter an integer, with 0 meaning "be quiet".

Exercise 5

Exercise 5: Build and run a program that uses parameters.

FIXME: shorten URLs

The code is at `$tutorial/step2_params`.

CMake file online: `https://https://github.com/ALPSCore/tutorial2/blob/master/step2_params/CMakeLists.txt`

Source file online: `https://https://github.com/ALPSCore/tutorial2/blob/master/step2_params/main.cpp`

- Play with the different values of parameters.
- Try to override them form the command line.
- Change the program to make `--loud` parameter an integer, with 0 meaning "be quiet".

```
$ cd $tutorial/step2_params
$ mkdir 000build
$ cd 000build
$ cmake ..
$ make
```

Simple simulation class (doing nothing)

- Simulation that just says "I am running."
- At this point, it is a good idea to indicate a debug or a release build:

STOPPED HERE

```
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
 # or:
$ cmake -DCMAKE_BUILD_TYPE=Release ..
```

- Make command options:

```
$ make VERBOSE=1 # runs make printing all commands
$ make -j2 # run 2 Make processes, to compile faster
$ nice make -j4 # 4 low-priority processes
```

Compute π by Markov chain MC

The problem:

- Integral of objective function over an area.
- Trivial Metropolis step to stay inside the area.
- Area: unit square; any step outside is rejected.
- Objective function:
  1 if inside an inscribed circle, 0 otherwise.
- Expected result: π (if multiplied by 4).

FIXME: picture

We need to use:

- **Accumulators**: named observable to gather statistics
- **Results**: named as accumulators, allow arithmetic operations (with error propagation!)
- Accumulators & Results can hold a vector
  (e.g., for vector-valued or parametrized objective function)

Types of accumulators

Types of accumulators:

1. Mean only (cheapest, least useful): `MeanAccumulator<double>`
2. No binning (cheap, no autocorrelation info): `NoBinningAccumulator<double>`
3. Full binning (most expensive, autocorrelation, error propagation): `FullBinningAccumulator<double>`
4. Log binning (less memory demanding, no error propagation): `LogBinningAccumulator<double>`

If a method is not available for the given accumulator type, it throws!

Exercise ??

Exercise ??: Compute π by Markov chain MC.
Note: the simulation code is split into 2 files.

- Compile and run the program.
- Run with various time limits.
- Run with different step sizes, compare autocorrelation length.
- Replace `FullBinningAccumulator` to `NoBinningAccumulator` and see the error-bars.

Checkpointing the simulation

- Checkpoint: save intermediate results, load to resume
- ALPSCore utilizes HDF5 format
  - Cross-platform
  - Hierarchical structure: Groups (∼directories), Data (∼files)
- ALPSCore can save/load:
  - Basic types (`int`, `double` *etc.*)
  - Vectors of basic types and of vectors of basic types *etc.*
  - Accumulators and parameters
  - Any user-defined class
    - Via `save()`/`load()` class members
    - Via traits (harder to do — more complex code)
- Parameters can be constructed from HDF5 file too
  - Should not try to define them again in this case!
    Use `my_params.is_restored()`.

Running and resuming

The code is in `$tutorial/step5_pi_checkpoint` .

1. Build as usual.
2. Run as usual (note more options available!)

```
$ ./alpsdemo --help
$ # Run for 5 sec
$ ./alpsdemo --step 1 --timelimit 5
```

3. Note new files appear:
   - "`*.out`" file contains simulation results
   - "`*.clone.h5`" file contains checkpoint
4. Restore the checkpoint:

```
$ # Run for 10 more sec:
$ ./alpsdemo alpsdemo.clone.h5 --timelimit 10
```

- Note:
  - compulsory `--step` is read from the checkpoint
  - parameters can be overridden (like `--timelimit`)

How to use MPI?

- Not many changes compared to the sequential version.
  1. Use `alps::mcmpiadapter`<*SequentialSimulationClass*> as your simulation class.
  2. Initialize MPI environment
  3. Make sure that the parallel processes do not conflict for input/output
  4. Use special constructor for parameters
- Note that the completion is checked only at certain intervals (1 sec minimum)

See the next slides for code changes!

MPI: Header files

```
#include <alps/mc/stop_callback.hpp>



class MySimulation : public alps::mcbase {
// ... rest of your class ...
};


// ... rest of your program ...
```

MPI: Header files

```
#include <alps/mc/stop_callback.hpp>

#include <alps/mc/mpiadapter.hpp>

class MySimulation : public alps::mcbase {
// ... rest of your class ...
};

typedef alps::mcmpiadapter<MySimulation> MyMpiSimulation;
// ... rest of your program ...
```

MPI: main()

```
    alps::params p(argc, (const char**)argv);
    MySimulation::define_parameters(p)
            .description("Dummy Monte Carlo program")
// etc...










//....
```

MPI: main()

```
    alps::params p(argc, (const char**)argv);
    MyMpiSimulation::define_parameters(p)
            .description("Dummy Monte Carlo program")
// etc...
    // init MPI, obtain communicator
    alps::mpi::environment mpi_env(argc, argv);
    alps::mpi::communicator comm;
    // Parse the parameters on master...
    alps::params p;
    if (comm.rank()==0) { // on master process...
        alps::params p_master(argc, (const char**)argv);
        p=p_master;
    }
    broadcast(comm, p, 0); // ...broadcast parameters to all.
//....
```

Exercise: MPI

- Build the MPI-parallelized program.
- Run with different number of processes.
- Observe checkpoint names.
- Try to restore from checkpoints.

Exercise: 2D Ising simulation

- The same principle as any other MC simulation.
- The program uses user-defined datatype, therefore needs loading/saving for it.
- Try to change the code to parallelize it.