# ALPSCore Tutorial

Simons Foundation
Many Electron Collaboration
Summer School Coding Workshop

June 16 – June 18
2017

Alexander Gaenko
7/14/2017

# ALPSCore tutorial: links

Login information:

https://git.io/simons-2017-login

These slides:

https://git.io/alpscore-tut-2017-02-24

Accompanying code:

https://github.com/galexv/ALPSCore_Tutorial2

ALPSCore website:

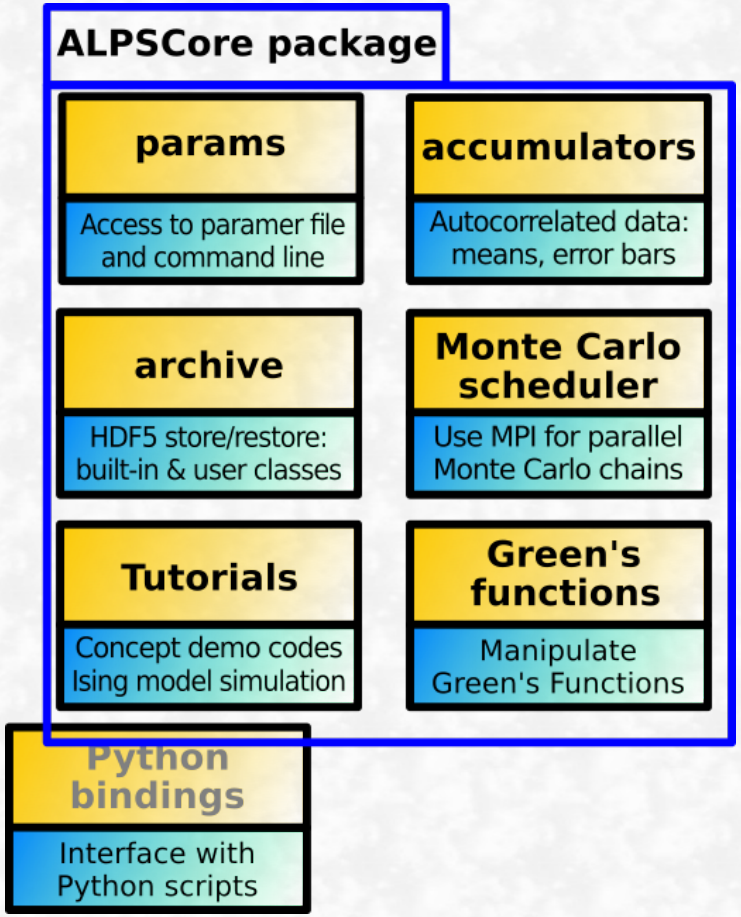
http://www.alpscore.org/

# What is ALPSCore?

- Software library for writing physics simulation codes
- Primarily for Monte  Carlo methods (as of now).

*However...*

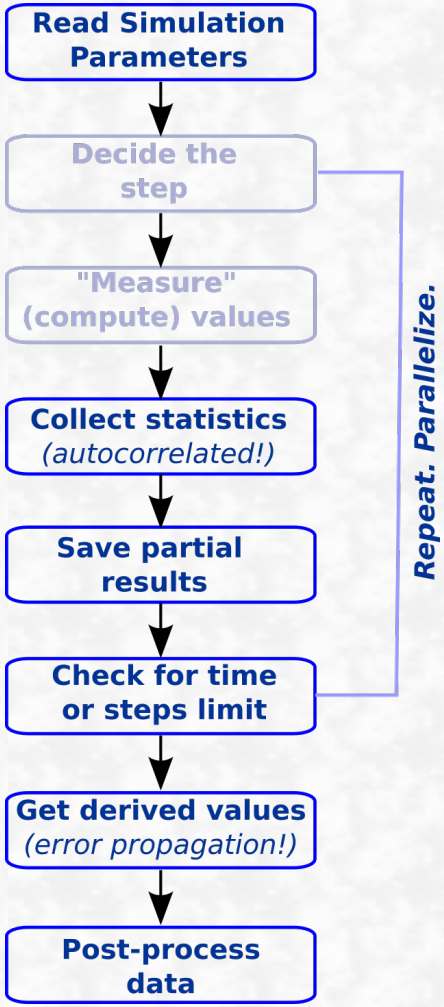
- Has support for Green's Functions;
- Other functionality is in works.



# Brief history of ALPSCore

- ALPSCore is the "core ALPS library"
- ALPS is "Algorithms and Libraries for Physics Simulations" (see http://alps.comp-phys.org)
- ALPSCore goals:
  - Be more compact
  - Faster development cycle
  - More extensive documentation.
- Focuses on the most often used components.

# Why use ALPSCore?

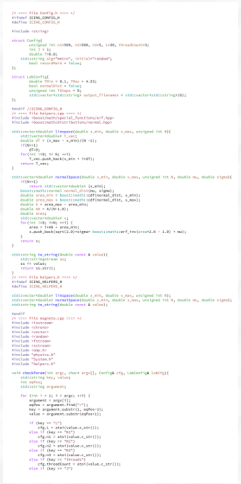


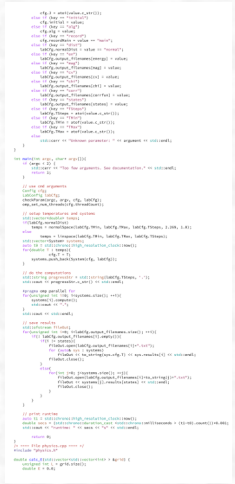

## Look at a typical simulation:

- A lot of „boilerplate“ code;
- Often non-trivial;
- Performance is important!
- Python (or other scripting): hard to combine with C++

**ALPSCore helps to focus on science!**

# 2D Ising simulation code: "from scratch"





- ~430 lines of C++ code
- ~50 lines do "science"
- No error estimation.
- Series of simulations
- OpenMP parallel

Caveat: it's "apple to oranges" comparison, but gives us some idea.

(Adapted and simplified from: https://github.com/s9w/magneto/)

# 2D Ising simulation code: Using ALPSCore

```cpp
/* ==== File ising.cpp ==== */
#include "ising.hpp"

// Defines the parameters for the ising simulation
alps::params& ising_sim::define_parameters(alps::params& parameters) {
    alps::mcbase::define_parameters(parameters)
        .description("2D ising simulation")
        .define<int>("length", "size of the periodic box")
        .define<int>("sweeps", 1000000, "maximun number of sweeps")
        .define<int>("thermalization", 10000, "number of sweeps for thermalization")
        .define<double>("temp", "temperature of the system");
    return parameters;
}

// Creates a new simulation.
ising_sim::ising_sim(parameters_type const & prm, std::size_t seed_offset)
    : alps::mcbase(prm, seed_offset)
    , length_(parameters["length"])
    , sweeps_(0)
    , thermalization_sweeps_(int(prm["thermalization"]))
    , total_sweeps_(prm["sweeps"])
    , beta_(1. / prm["temp"].as<double>())
    , spins_(length_, std::vector<int>(length_,0))
    , current_energy_(0)
    , current_magnetization_(0)
{
        measurements
            << alps::accumulators::FullBinningAccumulator<double>("Energy")
            << alps::accumulators::FullBinningAccumulator<double>("Magnetization")
            << alps::accumulators::FullBinningAccumulator<double>("AbsMagnetization")
            << alps::accumulators::FullBinningAccumulator<double>("Magnetization^2")
            << alps::accumulators::FullBinningAccumulator<double>("Magnetization^4")
            ;

    // Initializes the spins
    for(int i=0; i<length_; ++i) {
        for (int j=0; j<length_; ++j) {
            spins_[i][j] = (random() < 0.5 ? 1 : -1);
        }
    }

    // Calculates initial magnetization and energy
    for (int i=0; i<length_; ++i) {
        for (int  j=0; j<length_; ++j) {
            current_magnetization_ += spins_[i][j];
            int i_next=(i+1)%length_; // wrap around (PBC)
            int j_next=(j+1)%length_; // wrap around (PBC)
            current_energy_ += -(spins_[i][j]*spins_[i][j_next]+
                                 spins_[i][j]*spins_[i_next][j]);
        }
    }
}

// Performs the calculation at each MC step; decides if the step is accepted.
void ising_sim::update() {
    using std::exp;

    int i = int(length_ * random());
    int j = int(length_ * random());

    int i_nxt = (i+1) % length_;
    int i_prv = (i-1+length_) % length_;
    int j_nxt = (j+1) % length_;
    int j_prv = (j-1+length_) % length_;

    double delta=2.*spins_[i][j]*
                  (spins_[i_nxt][j]+
                   spins_[i_prv][j]+
                   spins_[i][j_nxt]+
                   spins_[i][j_prv]);

    if (delta<=0. || random() < exp(-beta_*delta)) {
        current_energy_ += delta;
        current_magnetization_ -= 2*spins_[i][j];
        spins_[i][j] = -spins_[i][j];
    }
}

// Collects the measurements at each MC step.
void ising_sim::measure() {
    ++sweeps_;

    if (sweeps_<thermalization_sweeps_) return;

    const double n=length_*length_; // number of sites
    double tmag = current_magnetization_ / n; // magnetization

    measurements["Energy"] << (current_energy_ / n);
    measurements["Magnetization"] << tmag;
    measurements["AbsMagnetization"] << fabs(tmag);
    measurements["Magnetization^2"] << tmag*tmag;
    measurements["Magnetization^4"] << tmag*tmag*tmag*tmag;
}

// Returns a number between 0.0 and 1.0 with the completion percentage
double ising_sim::fraction_completed() const {
    double f=0;
    if (sweeps_ >= thermalization_sweeps_) {
        f=(sweeps_-thermalization_sweeps_)/double(total_sweeps_);
    }
    return f;
}
```

```cpp
/* ==== File ising.hpp ==== */
#pragma once

#include <alps/mc/mcbase.hpp>

typedef std::vector< vector<int> > storage_type;

// Simulation class for 2D Ising model (square lattice).
class ising_sim : public alps::mcbase {
  private:
    int length_; // the same in both dimensions
    int sweeps_;
    int thermalization_sweeps_;
    int total_sweeps_;
    double beta_;
    storage_type spins_;
    double current_energy_;
    double current_magnetization_;

  public:
    /// Constructor
    ising_sim(parameters_type const & parms, std::size_t seed_offset = 0);

    /// Defines model-specific parameters
    static alps::params& define_parameters(alps::params& parameters);

    /// MC step
    void update();
    /// Measurements of quantities
    void measure();

    /// How far we are proceeded
    double fraction_completed() const;
};

/* ==== File main.cpp ==== */
#include "ising.hpp"
#include <iostream>
#include <alps/accumulators.hpp>
#include <alps/mc/api.hpp>
#include <alps/mc/mcbase.hpp>
#include <alps/mc/stop_callback.hpp>
#include <alps/mc/mpiadapter.hpp>

int main(int argc, char* argv[])
{
    namespace aa=alps::accumulators;

    typedef alps::mcmpiadapter<ising_sim> my_sim_type;

    alps::mpi::environment env(argc, argv);
    alps::mpi::communicator comm;
    const int rank=comm.rank();
    const bool is_master=(rank==0);

        alps::params parameters(argc, argv, comm);

        my_sim_type::define_parameters(parameters)
            .define<std::size_t>("timelimit", 5, "Time limit for the computation");

        if (parameters.help_requested(std::cout) ||
            parameters.has_missing(std::cout)) {
            return 1;
        }

        my_sim_type sim(parameters, comm);
       sim.run(alps::stop_callback(size_t(parameters["timelimit"])));

        aa::result_set results = sim.collect_results();

        if (is_master) {
        std::cout << results << std::endl;

        aa::result_wrapper mag4=results["Magnetization^4"];
        aa::result_wrapper mag2=results["Magnetization^2"];

        aa::result_wrapper binder_cumulant=1-mag4/(3*mag2*mag2);
        std::cout << "Binder cumulant: " << binder_cumulant
                  << std::endl;

        }
        return 0;
}
```

- ~150 lines of C++ code
- ~50 lines do "science"
- Error bars for autocorrelated data!
- Proper error propagation!
- MPI parallel (thousands of cores!)
- Graceful exit on signal or timeout!
- Built-in usage help message

# Using ALPSCore: *Pro & Contra*

- Ready-to-use framework;
- A lot of features "for free";
- Correct statistics;
- Concentrate on the "science part";
- Easy to link with your application.

- You have to install it;
- Trivial problems require more code;
- Some learning curve;
- Insists on preferred build system: CMake;
- Introduces library dependence.

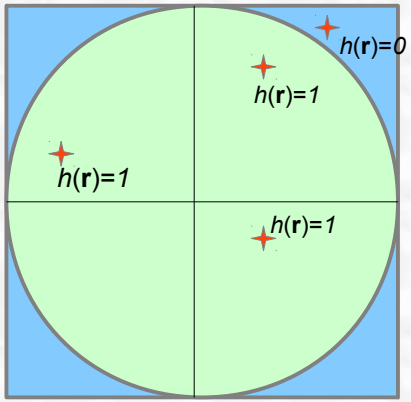
*Not unlike Make vs manual compilation, or CMake vs Make...*

# What is the plan for this tutorial?

1. Assume that ALPSCore is installed!
2. Make a very simple Monte Carlo simulation demo.
3. Parallelize the demo.
4. Make a 2D Ising model simulaton code.

# Exercise 1: Preface

**Task**: Compute π by Monte Carlo method.




$$\frac{S_{cir}}{S_{sqr}} = \frac{\pi}{4}$$

"hit function" $h(\boldsymbol{r}) = \begin{cases} 1 \text{ for } \boldsymbol{r} \in \text{Cir} \\ 0 \text{ for } \boldsymbol{r} \notin \text{Cir} \end{cases}$

$$\frac{S_{cir}}{S_{sqr}} = \frac{N_{cir}}{N_{sqr}} = \frac{1}{N}\sum_{i=1}^{N} h(\boldsymbol{r}_i) = \langle h \rangle$$

It is not hard to write the program to compute the mean $\langle h \rangle$.
*But...*

What about error bars?
Do you really remember those formulas? 😊

# Exercise 1: Hands-on

**Task**: Computing π by Monte Carlo method.

1. Build the program:

```
$ mkdir 000build
$ cd 000build
$ cmake ..
$ make
```

2. Run :

```
$ ./alpsdemo --trials=1000
```

3. Request help:

```
$ ./alpsdemo --help
```

4. Timed runs, varying trial count:

```
$ time -p ./alpsdemo --trials=100
```

5. Timed runs, varying time limits:

```
$ time -p ./alpsdemo \
   --trials=999999 \
   --timelimit=10
```

6. Start a long run; then press **Ctrl-C**; observe graceful termination!

# Exercise 1: Implementation specifics

Directory: `step1_pi/`

`simulation.hpp` describes simulation class (the model)

`simulation.cpp` implements the model

- *constructor:* constructs the model
- *update():* performs MC trial
- *measure():* measures the quantities
- *fraction_completed():* are we done yet?

This is where the "science" is.

`main.cpp` sets up and runs the simulation

`CMakeLists.txt` specifies application structure

Let's look at the code!

(**orange** and **blue** are clickable – *try!*)

# Exercise 1: Hands-on, continued

**Task**: Use parameter file; play with the code.

7. Use parameter file:

```
$ ./alpsdemo ../demo.ini
```

8. Override params from file:

```
$ ./alpsdemo ../demo.ini \
    --trials=9999
```

9. Change the code:

`--trials=0` should mean "till timeout"

(*solution*)

*File "demo.ini"* :

```
# how long to run?
timelimit = 100

# how many trials?
trials = 1000
```

(remember, orange *is clickable!*)

# Exercise 1: Implementation specifics

`simulation.cpp` `main.cpp`

Code uses *parameters* : to allow user to pass input data

Each parameter has:

- ✔ Name (*e.g.* `"trials"`)
- ✔ Type (*e.g.* `int`)
- ✔ Description (*e.g.* `"Number of trials"`)
- ✔ Optionally: a *default value*

`define_parameters()` is responsible for defining simulation-specific parameters

Let's look at the code!

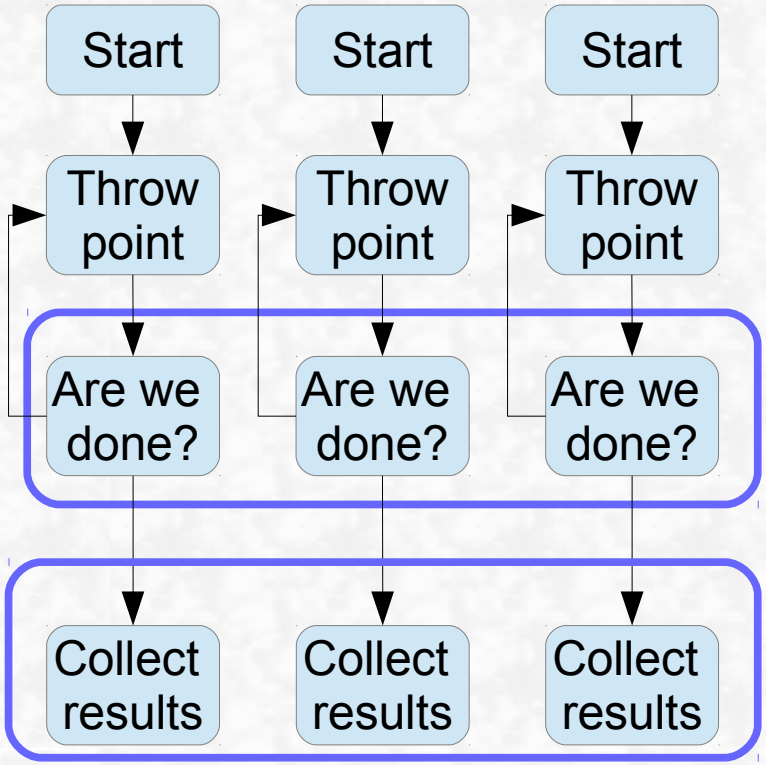
(remember, **orange** and **blue** *are clickable!*)

# Exercise 1: Take-home message

1. Linking `ALPSCore` to your application is very simple.
2. Most of the framework is ready-made for you; you only code "science";
3. You get parameters, scheduling, graceful termination, and error bars — all provided by the library.

# Exercise 2: Preface

Point throwing can be done independently!




- Completion check requires occasional communications ;
- Results are collected from all processes;
- The rest is completely parallel!

# Exercise 2: Hands-on

**Task**: Compute π in parallel.

Let's look at the code!

1. Examine the differences (main.diff):
   1. MPI is initialized.
   2. A few operations are parallel-aware:
      params ctor  simulation ctor
   3. An adapter class is used.
   4. Only master process should print.

Directory:
`step2_parallel_pi/`

2. Build and run the program:

```
$ mkdir 000build
$ cd 000build
$ cmake ..
$ make
$ ./alpsdemo --help
```

3. Do timed runs:

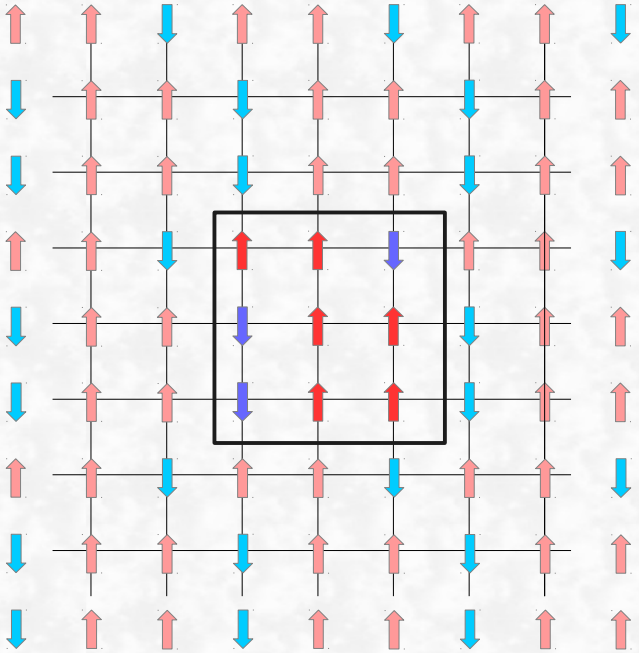
```
$ time -p mpiexec -n 1 \
    ./alpsdemo --trials=999999

$ time -p mpiexec -n 2 \
    ./alpsdemo --trials=999999
```

# Exercise 2: Take-home message

1. Once you implemented serial MC,
   it's easy to make it parallel.
2. The requested MC steps are distributed
   among the parallel processes.
3. The processes occasionally communicate,
   to check if it's time to stop.

# Exercise 3: Preface

**Task**: "Physics" problem: 2D Ising model.



Energy: $E = -\frac{1}{N}\sum_{(i,j)} s_i s_j$

Magnetization: $M = \frac{1}{N}\sum_{i} s_i$

Temperature *T*; $\beta = 1/T$

Boltzmann distribution: $p_i \sim e^{-\beta E}$

Mean values: $\langle E \rangle, \langle M \rangle, \langle |M| \rangle, \langle M^2 \rangle, \langle M^4 \rangle$

Binder Cumulant: $U = 1 - \frac{\langle M^4 \rangle}{3\langle M^2 \rangle^2}$

# Exercise 3: Approach

**Task**: "Physics" problem: 2D Ising model.

Directory: `step3_ising/`

Monte Carlo step:

1. Flip a random spin;
2. Update energy;
3. Update magnetization;
4. Accept or reject update.

Let's look at the code!

**Problem**: Measurements are *correlated*.

1. Naive approach: too narrow error bars;
2. Use `LogBinningAccumulator` for the correlated data;
3. Use `FullBinningAccumulator` if you need arithmetic.

*More on this later!*

# Exercise 3: Hands-on

**Task**: Try different simulation parameters and accumulators.

1. Build and run the code ($ `cmake .. && make`)

2. Use parameter file with different accumulators:

```
$ ./ising2_mc ../ising.ini --acc=nobin
$ ./ising2_mc ../ising.ini --acc=logbin
$ ./ising2_mc ../ising.ini --acc=fullbin
```

Look at energy, magnetization, and Binder Cumulant.
Try decrease the temperature.
Note different error bars!

3. Extra points: parallelize!

1) Use `mcmpiadapter`

2) Initialize MPI

3) Pass communicator to:
   - params
   - simulation constructors

4) Make sure only master prints.

# Exercise 3: Accumulators

| Accumulator name | Mean? | Error bar? | Auto correlation? | Nonlinear error propagation? | Memory cost |
|---|---|---|---|---|---|
| Mean | ✔ | ✗ | ✗ | ✗ | $O(D)$ |
| NoBinning | ✔ | ✔ | ✗ | ✗ | $O(D)$ |
| LogBinning | ✔ | ✔ | ✔ | ✗ | $O(D \log N)$ |
| FullBinning | ✔ | ✔ | ✔ | ✔ | $O(D \min(N,128))$ |

$D$: size of data $N$: number of data points

# Exercise 3: Take-home message

1. ALPSCore provides means to collect statistics properly;
2. Neglected autocorrelation results in too optimistic (polite speak for "*wrong*") error bars;
3. Use binning accumulators (preferably FullBinning) for your Markov chain simulations;
4. Accumulators work automagically for MPI parallelized simulations.