ALPSCore Tutorial

DRAFT Version 2

Alexander Gaenko
2/24/2017

ALPSCore tutorial: links

These slides:

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

Accompanying code:

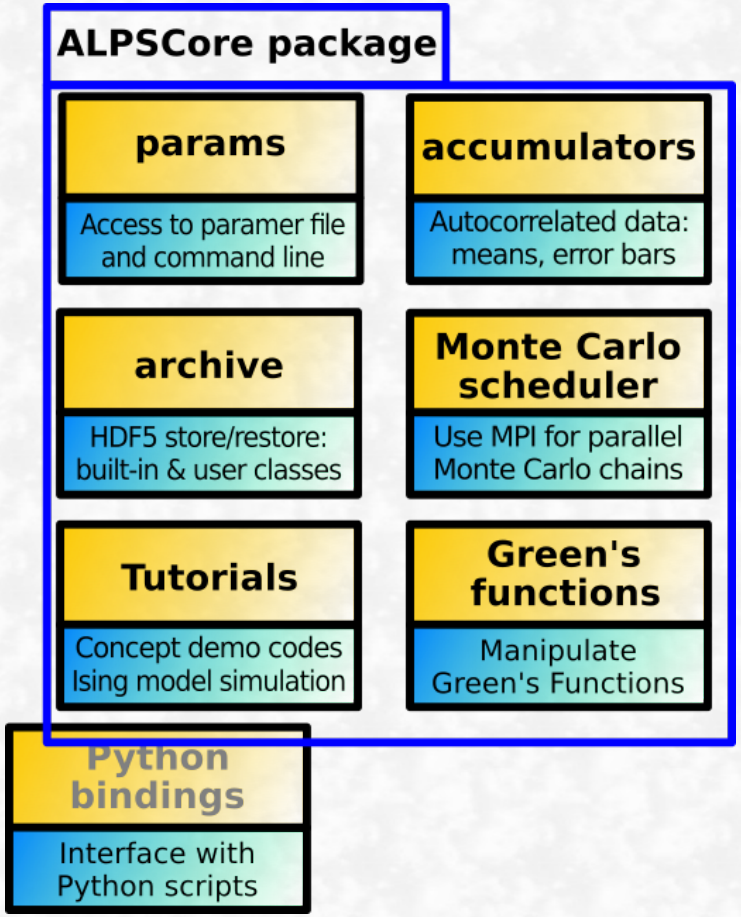
https://github.com/galexv/ALPSCore_Tutorial2

What is ALPSCore?

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

However...

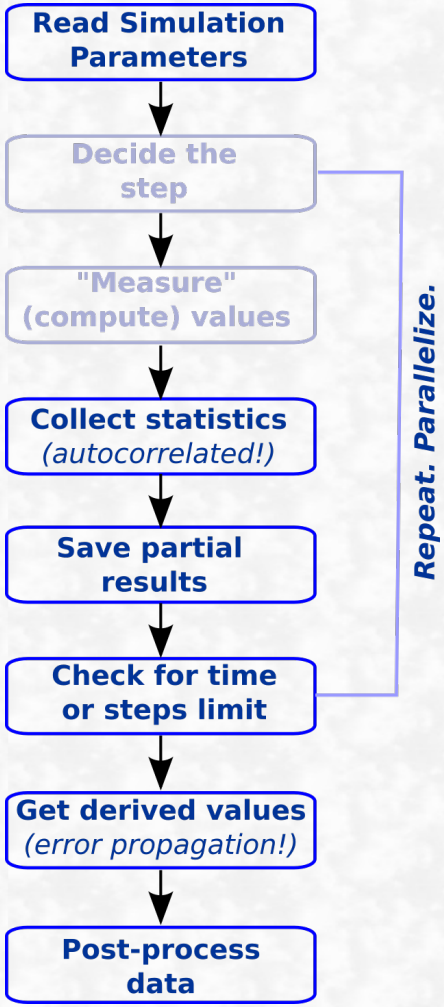
- Has support for Green's Functions;
- Other functionality is in works.



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?



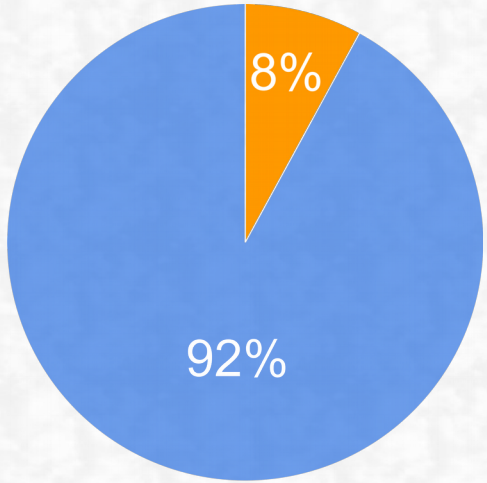
Look at a typical simulation:

- 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!

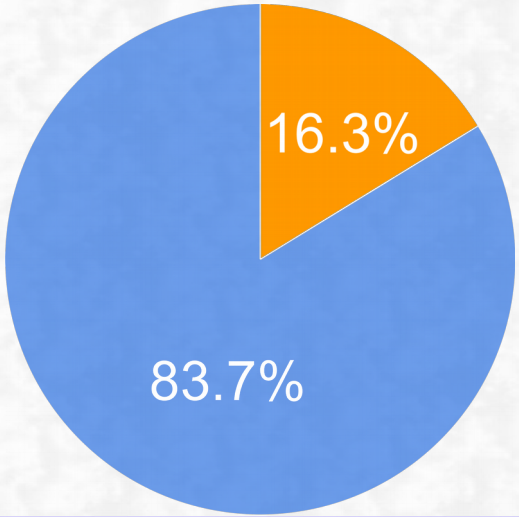
2D Ising simulation code: "from scratch"

- 490 lines of C++ code
- 39 lines do "science"
- No error estimation!
- OpenMP parallel

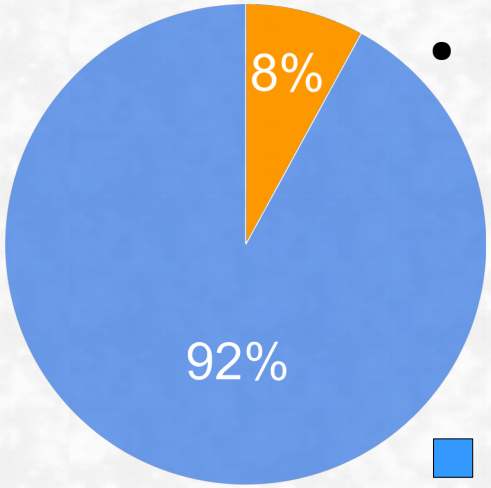


2D Ising simulation code: Using ALPSCore

- 246 lines of C++ code
- 40 lines do "science"
- Error bars for autocorrelated data!
- MPI parallel (thousands of cores!)
- Graceful exit on signal or timeout!



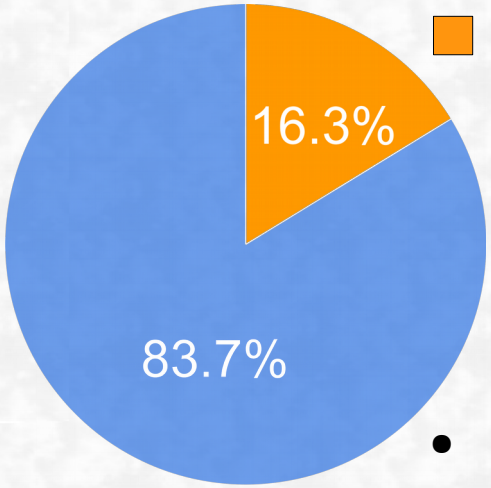
2D Ising code: "from scratch" vs ALPSCore



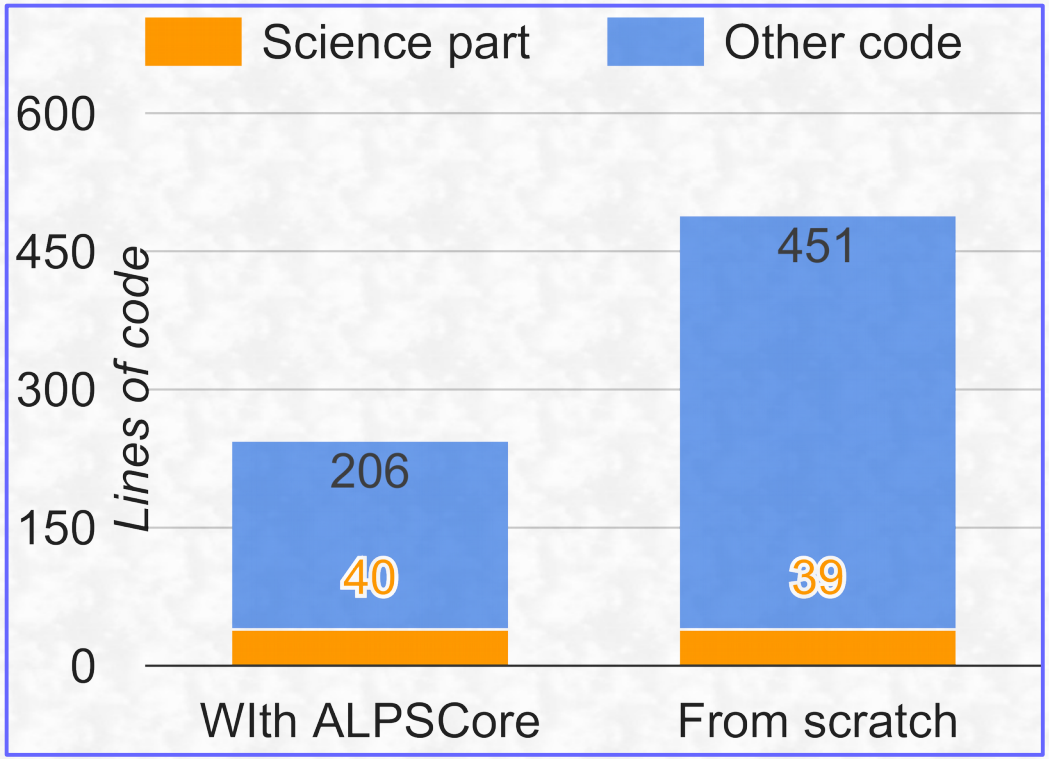
- from scratch

Other code

Science code



- with ALPSCore



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.

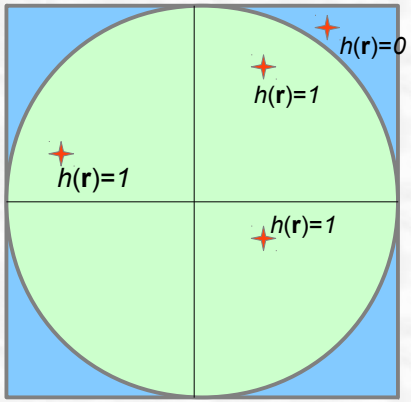
Not unlike Make vs manual compilation, or CMake vs Make...

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: 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: 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: Hands-on

Task: Computing π by Monte Carlo method.

1. Build the program:

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

2. Run without arguments:

```
$ ./alpsdemo
```

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: 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
```

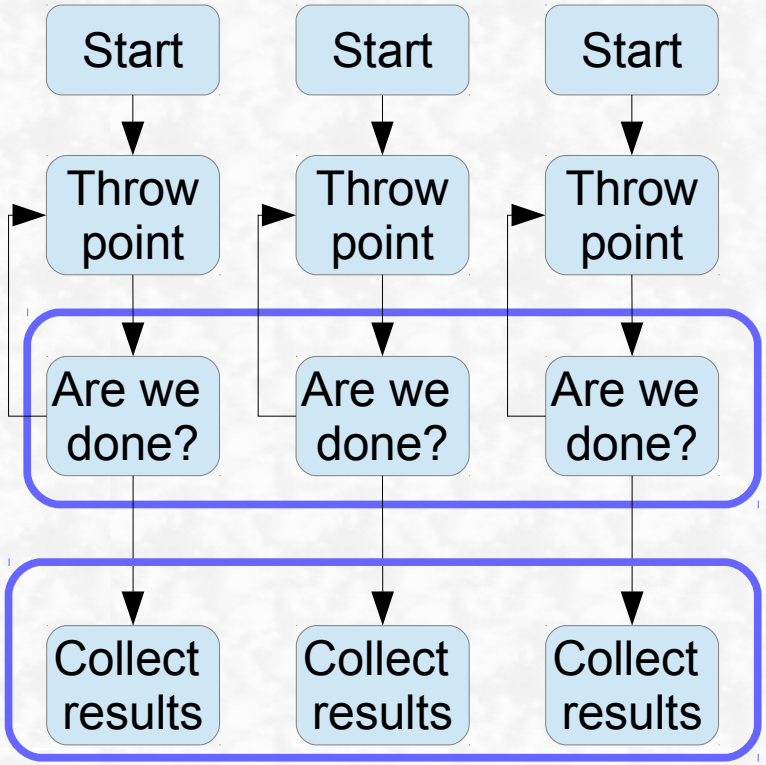
(remember, orange *is clickable!*)

Exercise 1: 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:

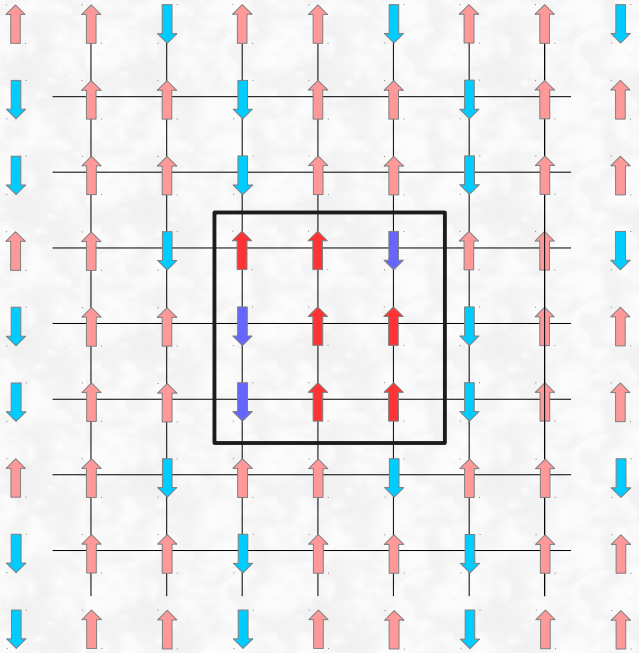
```
$ time -p mpiexec -n 1 \
    ./alpsdemo --trials=999999

$ 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.

Exercise 3: Hands-on

Task: Try different simulation parameters and accumulators.

1. Build and run the code

2. Use parameter file with different accumulators:

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

Look at energy and Binder Cumulant.
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: Take-home message

1. ALPSCore provides means to collect statistics properly;
2. Autocorrelation results in too optimistic error bars;
3. Use binning accumulators (preferably FullBinning) for your Markov chain simulations.