```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, "maximum 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;
}

/* ==== 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;
}
```