Carlo.jl: A general framework for Monte Carlo simulations in Julia

Lukas Weber

Center for Computational Quantum Physics, The Flatiron Institute,
162 Fifth Avenue, New York, New York 10010, USA
Max Planck Institute for the Structure and Dynamics of Matter,
Luruper Chaussee 149, 22761 Hamburg, Germany

lweber@flatironinstitute.com

Abstract

Carlo.jl is a Monte Carlo simulation framework written in Julia. It provides MPI-parallel scheduling (including parallel tempering), organized storage of input, checkpoint, and output files, as well as statistical postprocessing. With a minimalist design, it aims to aid the development of high-quality Monte Carlo codes, especially for demanding applications in condensed matter and statistical physics. This hands-on user guide shows how to implement a simple code with Carlo.jl, how to run an existing stochastic series expansion code based on Carlo.jl, and provides benchmarks to show its efficacy.

Contents

1   Introduction

Monte Carlo methods have been a powerful computational tool since the very beginnings of the fields of statistical and condensed matter physics [1]. Initially built to study the classical Boltzmann distribution, they perform well on a wide variety of high-dimensional integration problems due to their ability to break the curse of dimensionality. In quantum systems, where a probabilistic interpretation of the partition function is in general not possible, quantum Monte Carlo algorithms can exploit certain quantum-to-classical mappings to make powerful and accurate predictions for models that are out of reach of other state-of-the art methods [2].

In exchange for simulating high-dimensional problems, quantum Monte Carlo simulations in particular rely on efficient implementations making use of massive parallelism. In the course of a simulation, many random samples are produced that have to be stored and statistically postprocessed to arrive at results with reliable error bars. These requirements typically necessitate some amount of "bookkeeping" code that is unrelated to the Monte Carlo algorithm itself, but still crucial to its performance. This inspired the development of earlier frameworks such as the ones included in ALPS [3] and the subsequent ALPSCore [4] in C++.

The Julia programming language [5] is becoming increasingly popular as an alternative to traditional scientific computing languages such as C++ and FORTRAN. Its heavy reliance on type inference, dynamic dispatch, and just-in-time compilation allow it to provide both high performance and the flexibility of a high-level scripting language. Furthermore, its modern approach to dependency management greatly facilitates installing and reusing existing software.

The purpose of Carlo.jl is to help implementers of Monte Carlo codes in Julia to write high-quality, high-performance packages without worrying about the peripheral bookkeeping tasks that are common to all Monte Carlo simulations. Its features include (i) Monte Carlo aware parallel scheduling (including parallel tempering) (ii) organized and reproducible storage of simulation inputs, checkpoints, and results, as well as (iii) statistical postprocessing of autocorrelated Monte Carlo samples.

In Section 2, the features of Carlo.jl and their inner workings are explained in detail. Section 3 contains three examples: an Ising code that provides a minimal skeleton for implementing Monte Carlo algorithms in Carlo.jl, a state-of-the-art stochastic series expansion Monte Carlo code [6], and an example of the parallel tempering feature. Afterwards, in Section 4, we show correctness and performance benchmarks of Carlo.jl, including a direct performance comparison of a Julia and a C++ implementation of the stochastic series expansion code.
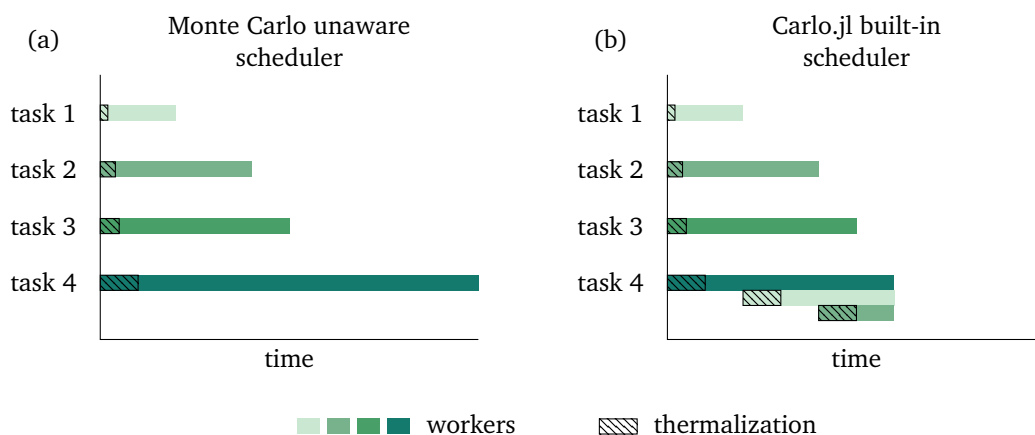
Figure 1: **Parallelism in Monte Carlo simulations**. (a) Monte Carlo unaware parallelization over different parameters (e.g. system sizes, temperatures) can lead to suboptimal resource usage if tasks are not well balanced. (b) Carlo.jl parallelizes over different parameter sets but also takes advantage of the possibility to share work between different Monte Carlo processes for the same task. Each new process needs to thermalize first, which is a serial cost that can only be reduced by more fine-grained parallelization.

Finally, in Section 5, a short conclusion and outlook are given. Apart from this user guide, users may refer to the Carlo.jl documentation[1] for more detailed descriptions of each component.

2 Features

The purpose of this section is to give a high-level overview of the way Carlo.jl works without descending too far into technical details and concrete usage instructions. For a more hands-on introduction to the usage of Carlo.jl in practice, see Section 3.

2.1 Monte Carlo aware parallel scheduling

One useful property of Monte Carlo simulations is that they are embarrassingly parallel. An arbitrary number of random processes can be allocated to many workers simultaneously, which, once thermalized, can all produce samples that are finally averaged together. Apart from this intrinsic parallelism, simulations are often run for a variety of different parameters in parallel. Both of these kinds of parallelism can be exploited using a standard parallel scheduler. However, if the load between different parameter sets is not balanced, the resource usage of this approach is not optimal: Most workers can run out of work while few are still stuck simulating the last few expensive parameter sets (Fig. 1(a)). In Monte Carlo simulations, such a situation can always be avoided by simply allowing the idle workers to join other tasks and share the work (Fig. 1(b)).

Carlo.jl uses MPI.jl [7] to orchestrate this kind of Monte Carlo aware, work-sharing scheduling and automatically averages the results. Each parameter set in the simulation corresponds to a *task* and each task may have multiple *runs*, which correspond to independent random processes that are averaged. If workers are idle, the scheduler will assign them a new task from the set of unfinished tasks in a round-robin manner and start a new run for that task, provided the remaining work is not less that the number of thermalization sweeps.

---

[1]The documentation can be viewed online on GitHub, built locally by running cd docs; julia --project make.jl, or accessed directly within the Julia REPL.

On top of this model of trivial parallelism, Carlo.jl also supports nontrivial MPI parallelism: The work of each single run may optionally be shared by a team of MPI ranks. This advanced feature, called *parallel run mode* allows the natural implementation of certain Monte Carlo meta-techniques such as parallel tempering, which will be illustrated in Section 3.3. Another possible use case is the implementation of branching-random walk codes that propagate an population of walkers in parallel.

2.2 Organized storage of inputs, checkpoints and results

As mentioned in the previous section, large-scale Monte Carlo simulations can involve scanning large parameter spaces and produce large numbers of random samples. In addition, long-running simulations should also implement checkpointing so that valuable computation time is not wasted when an error occurs or the job runs out of its allocated run time on a computing cluster. These boundary conditions induce data management challenges at the beginning, during, and at the end of a simulation. This section explains how Carlo.jl handles these challenges for the user.

2.2.1 Input

Rather than specifying simulation parameters in traditional configuration files, Carlo.jl leverages the dynamic nature of Julia to set parameters programmatically. The recommended workflow for this is to write a job script as shown in Listing 1. The job script can be split into three parts. First, the `TaskMaker` from `Carlo.JobTools` is used to procedurally generate a list of parameter sets. Then, this list is retrieved using `make_tasks(tm)` and fed into the `JobInfo` alongside other options such as the maximum run time limit and the checkpoint interval. Finally, `start(job, ARGS)` initiates the Carlo.jl command line interface (CLI) when the job script is executed. The CLI allows *running* the simulation, and for unfinished simulations allows checking their *status* and *merging* the already available data to obtain preliminary results.

While recommended, the use of `TaskMaker` and the CLI is optional. For more information on how to manually start simulations from within Julia, interested readers are referred to the documentations of `JobInfo` and the `start` function.

2.2.2 Checkpointing

In a high-performance computing environment, calculations often run on limited time allocations that may be shorter than the length of the calculation one wishes to perform. If a calculation runs out of time or uses too much memory, it may be terminated by the environment, which leads to the loss the entire progress.

Checkpointing, i.e. saving the complete Monte Carlo configuration to disk at regular intervals, allows to gracefully recover from such situations and simply continue the calculation from the point it left off. Carlo provides a simple interface to save the configuration state to an HDF5 [8,9] group. Internally, it makes sure that these checkpoints are never corrupted, even if the process is terminated in the middle of writing.[2] Running a simulation as described in the previous subsection will automatically resume from checkpoints if they exist.

While packages like JLD2.jl [10] exist that allow automatically saving arbitrary Julia objects, the Carlo.jl checkpointing interface instead requires developers to manually choose which data to save in what way. This can be helpful in reducing the checkpoint file size and encourages maintaining stable serialization interfaces.

---

[2]This works by writing to a temporary file first and then moving that file to the final location, overwriting the old file. Moving (as opposed to writing) is an atomic operation that cannot be interrupted.

```julia
# example_job.jl

using Carlo
using Carlo.JobTools
using Ising

tm = TaskMaker()

tm.sweeps = 20000
tm.thermalization = 2000
tm.binsize = 100

Ts = range(1, 3, 10)
Ls = [8, 12, 16]
for L in Ls
    for T in Ts
        tm.T = T
        tm.Lx = L
        tm.Ly = L
        task(tm)
    end
end

job = JobInfo("example_job", Ising.MC;
    run_time = "24:00:00",
    checkpoint_time = "30:00",
    tasks = make_tasks(tm),
)
start(job, ARGS)
```

Listing 1: **Carlo job script**. This snippet generates a job for the Ising code implemented in Section 3.1. Arbitrary parameters can be assigned to the TaskMaker tm. The function task(tm) creates a snapshot of all currently set values and turns it into a task that will be simulated. The list of tasks is returned by make_tasks and passed to the JobInfo structure. The detailed meanings of the options are explained in Section 3.1.

2.2.3 Results

Like checkpoints, Carlo.jl stores simulation results in HDF5 files. Inspecting these files allows retrieving the entire Monte Carlo time series. In the postprocessing step that is described in Section 2.3, these raw files are merged and averaged to produce means, statistically correct error bars and approximate autocorrelation times. The merged results are saved in a human-readable JSON file. For subsequent postprocessing, the Carlo.ResultTools module provides a simple way to import these JSON files into a Julia Dataframe (e.g. from DataFrames.jl [11]).

To facilitate reproducibility, and conform with the best practices of research data management, the result files produced by Carlo.jl contain all parameters used to run the simulation as well as the version of Carlo.jl and of the package containing the Monte Carlo algorithm implementation.

2.3 Postprocessing

Monte Carlo time-series are in general autocorrelated. Carlo.jl performs postprocessing on measured observables to provide the user with statistically sound means and standard errors using a binning analysis. The binning analysis works in two steps.

The first, *internal binning* step is performed during the simulation before Monte Carlo measurements are written to disk. The bin size for this step is set by `binsize` in the task parameters. The purpose of internal binning is simply to reduce memory and disk-I/O overhead. As long as `binsize` is not big compared to the total number of samples, it does not influence the magnitude of the final errors computed by Carlo.jl.

The second binning step happens once the simulation is complete. For a given task, there will be multiple files containing measurement results from different independent runs that were scheduled by Carlo.jl (Fig. 1(b)). These internally binned Monte Carlo time series are concatenated and binned further. The purpose of this "rebinning" step is to produce sufficiently large bins to recover uncorrelated statistics that allow extracting the statistical errors using the standard estimator

$$\sigma_A = \sqrt{\frac{1}{N_{\text{rebin}}(N_{\text{rebin}} - 1)} \sum_n (\overline{A}_n - \overline{A})^2}, \tag{1}$$

where $\overline{A}_n$ are the means of each rebinned sample and $\overline{A} = \sum_n \overline{A}_n / N_{\text{rebin}}$. By default, the number of "rebins" N_{rebin} is chosen using the heuristic

$$N_{\text{rebin}}(N) = N_{\text{rebin}}^{\text{min}} + \max\left\{0, \sqrt[3]{N - N_{\text{rebin}}^{\text{min}}}\right\}, \tag{2}$$

where N is the total number of measured bins and $N_{\text{rebin}}^{\text{min}} = 10$ is a minimum in case there are only few bins in the simulation. Choosing a cube-root scaling is a simple way to ensure that both N_{rebin} and the length of each rebin, N/N_{rebin}, approach infinity for increasing N, while devoting more resources to the latter. The default value of N_{rebin} can be overridden using the task parameter `rebin_length` in the job script.

A rough estimate for the autocorrelation time τ_{auto} is calculated based on the ratio of the rebinned error σ_A and naive error computed without any rebinning, $\tilde{\sigma}_A$, [12]

$$\tau_{\text{auto}} = \frac{\sigma_A^2}{2\tilde{\sigma}_A^2}. \tag{3}$$

Note that τ_{auto} is measured in units of the internal bin size (set by the `binsize` parameter). To find the autocorrelation time in terms of Monte Carlo sweeps, `binsize` should be set to 1.

Apart from the observables themselves, users can also define *evaluables* that depend on the observables via some function. The bias-corrected means and errors for evaluables are then propagated using the jackknife [13] method on the rebinned samples.

3 Examples

The specifics of implementing and running codes with Carlo.jl are best explained using examples. In the following, we will show three examples, in raising complexity.

The first example, Section 3.1, will be the paradigmatic Metropolis Monte Carlo algorithm for the Ising model. While this algorithm itself is very simple and of limited use in practice, it provides a minimal skeleton for implementing new Monte Carlo algorithms in Carlo.jl without the distractions of method-specific details.

The second example, Section 3.2, will be an implementation of the stochastic series expansion Monte Carlo [6] approach on top of Carlo.jl. While this implementation is a separate package and we will not go into details about its inner workings, it here serves to show how Carlo.jl is useful in state-of-the-art Monte Carlo calculations. This will be supplemented in Section 4 by a benchmark directly comparing Julia and C++ performance for that code.

The last example, Section 3.3, will have a split purpose. First, it will introduce the parallel-tempering feature that is included in Carlo.jl. Second, it will show how parallel-tempering was implemented as a simple wrapper on top of the parallel run mode feature.

3.1 Ising model

In this section, we will give a hands-on usage example for Carlo.jl by implementing and running the Metropolis Monte Carlo algorithm [1] for the Ising model on a square lattice,

$$H = -\sum_{x,y} \sigma_{x,y}^z (\sigma_{x+1,y}^z + \sigma_{x,y+1}^z), \tag{4}$$

with periodic boundary conditions. All code for this example is available in a separate repository [14].

3.1.1 Implementation

First, we need to implement the algorithm. It is a good idea to write such an implementation as its own Julia package. In the following, we will assume that we have generated a package called `Ising`, with a file `Ising.jl` containing all the code.

In this file, we declare a new subtype of `AbstractMC`.

```
# Ising.jl
module Ising

using Carlo
using HDF5

struct MC <: AbstractMC
    T::Float64

    spins::Matrix{Int8}
end
```

The type `MC` holds two fields: the temperature `T` and a matrix of integers representing the spins on the square lattice.

The constructor of `MC` should accept a dictionary of parameters that are defined in the job script (see Listing 1).

```
function MC(params::AbstractDict)
    Lx = params[:Lx]
    Ly = get(params, :Ly, Lx)
    T = params[:T]
    return MC(T, zeros(Lx, Ly))
end
```

We read the temperature and lattice dimensions from the parameters and construct the `MC` structure accordingly.

The remainder of the implementation consists of methods for Carlo.jl's `AbstractMC` interface.

```
function Carlo.init!(mc::MC, ctx::MCContext, params::AbstractDict)
    mc.spins .= rand(ctx.rng, Bool, size(mc.spins)) .* 2 .- 1
    return nothing
end
```

The `Carlo.init!` function is called to initialize the Monte Carlo configuration at the beginning of a new simulation (but not when restarting from a checkpoint). In addition to the `MC` object, it receives a `MCContext`, which is our handle to interact with Carlo.jl's functionality. Here, we use the integrated random number generator `ctx.rng` to generate a spin configuration. It is important to always use `ctx.rng` for random numbers to maintain reproducibility from a given random seed.[3]

Next we implement the Monte Carlo updates or sweeps.

```julia
function periodic_elem(spins::AbstractArray, x::Integer, y::Integer)
    return spins[mod1.((x, y), size(spins))...]
end

function Carlo.sweep!(mc::MC, ctx::MCContext)
    Lx = size(mc.spins, 1)

    for _ = 1:length(mc.spins)
        i = rand(ctx.rng, eachindex(mc.spins))
        x, y = fldmod1(i, size(mc.spins,1))

        neighbor(dx, dy) = periodic_elem(mc.spins, x + dx, y + dy)
        ratio = exp(
            -2.0 / mc.T *
            mc.spins[x, y] *
            (neighbor(1, 0) + neighbor(-1, 0)
            + neighbor(0, 1) + neighbor(0, -1)),
        )

        if ratio >= 1 || ratio > rand(ctx.rng)
            mc.spins[x, y] *= -1
        end
    end
    return nothing
end
```

According to the Metropolis algorithm, we propose a spin flip and calculate the energy difference between the new and old configurations, which can be reduced to a calculation of the neighboring spins. Note the use of functions like `mod1` and `fldmod1`, which are helpful when applying modular arithmetic to the 1-based indices of Julia arrays.

Next are the measurements.

```julia
function Carlo.measure!(mc::MC, ctx::MCContext)
    mag = sum(mc.spins) / length(mc.spins)

    energy = 0.0
    correlation = zeros(size(mc.spins, 1))

    for y = 1:size(mc.spins, 2)
        for x = 1:size(mc.spins, 1)
            neighbor(dx, dy) = periodic_elem(mc.spins, x + dx, y + dy)
            energy += -mc.spins[x, y]*(neighbor(1,0)+neighbor(0,1))

            # in practice, one should use more lattice symmetries!
            correlation[x] += mc.spins[1, y] * mc.spins[x, y]
        end
    end
```

---

[3]The seed can be set in the job script via the parameter seed.

```
    measure!(ctx, :Energy, energy / length(mc.spins))

    measure!(ctx, :Magnetization, mag)
    measure!(ctx, :AbsMagnetization, abs(mag))
    measure!(ctx, :Magnetization2, mag^2)
    measure!(ctx, :Magnetization4, mag^4)

    measure!(ctx, :SpinCorrelation, correlation ./ size(mc.spins, 2))
    return nothing
end
```

We calculate the magnetization and its moments, the energy per spin, and the spin correlation function. The function measure!(ctx, name, value) passes each result to Carlo.jl. As we see for the correlation function, observables need not be scalars. Vectors (or higher-rank) arrays are also supported.

The next method is optional, but can be used to register *evaluables*, that is, quantities that depend on Monte Carlo expectation values. Carlo.jl will perform error-propagation and bias correction on these (see Section 2.3).

```
function Carlo.register_evaluables(
        ::Type{MC}, eval::AbstractEvaluator, params::AbstractDict
)
    T = params[:T]
    Lx = params[:Lx]
    Ly = get(params, :Ly, Lx)

    evaluate!(eval, :BinderRatio,
        (:Magnetization2, :Magnetization4)
    ) do mag2, mag4
        return mag2 * mag2 / mag4
    end

    evaluate!(eval, :Susceptibility, (:Magnetization2,)) do mag2
        return Lx * Ly * mag2 / T
    end

    evaluate!(eval, :SpinCorrelationK, (:SpinCorrelation,)) do corr
        corrk = zero(corr)
        for i = 1:length(corr), j = 1:length(corr)
            corrk[i] += corr[j] * cos(2pi/length(corr)*(i-1)*(j-1))
        end
        return corrk
    end

    return nothing
end
```

Using evaluate!, we provide formulas for useful quantities such as the Binder ratio, the susceptibility and the Fourier transform of the correlation function. Note that Carlo.register_evaluables is called during the postprocessing step, after the simulation is complete. It has no access to any simulation state.

Finally, we implement the checkpointing interface.

```
function Carlo.write_checkpoint(mc::MC, out::HDF5.Group)
    out["spins"] = mc.spins
    return nothing
```

```
end

function Carlo.read_checkpoint!(mc::MC, in::HDF5.Group)
    mc.spins .= read(in, "spins")
    return nothing
end

end # module Ising
```

When starting a simulation from a checkpoint, `Carlo.read_checkpoint!` is called instead of `Carlo.init!`, so that we have to recover everything from disk that is not already calculated in the constructor `MC(params)`. The temperature is set in the constructor so we do not have to save it to disk. The spin configuration, however, has to be saved. For more information on how to read and write to an `HDF5.Group`, see the documentation of HDF5.jl [8].

This concludes the `Ising.jl` file and our implementation. Next, we will use it to run some simulations.

3.1.2 Running simulations

To run a simulation, we first need to install Carlo.jl and the Ising module. This works via the Julia REPL.

```
julia>]
pkg> add Carlo
pkg> add https://github.com/lukas-weber/Ising.jl
```

Readers who have followed along in the last section and wrote their own code in a different directory, can instead import it in development mode using `dev path/to/Ising`.

Next, we write the job script from Listing 1. The meanings of the parameters are as follows.

Lx, Ly, T used by our implementation to set the size and temperature of the Ising model.

thermalization the number of sweeps that are performed in the thermalization phase of the simulation, before measurements are taken.

sweeps the number of measurement sweeps.

binsize the internal bin size, that is the number of samples that are averaged before writing to disk. This should be much smaller than the total number of sweeps but large enough to not waste too much disk space in large simulations.

There are more optional parameters that are explained in greater detail in the documentation for `TaskInfo` and `JobInfo`.

Using the given system sizes and temperatures, we generate a collection of tasks. Then, all information about the job is fed to the `JobInfo` constructor, which additionally takes the job name (here `"example_job"`) the Monte Carlo type we implemented (`Ising.MC`), as well as the total run time and the interval between checkpoints.

Finally, `start(job, ARGS)` invokes the Carlo.jl CLI, which we can run from the shell as follows.

```
$ julia example_job.jl --help
$ julia example_job.jl run
```

The first line gives an overview over the CLI, the second line starts the simulation. For long-running simulations, the commands `status` and `merge` are helpful to check the progress and postprocess those results that are already available, respectively.
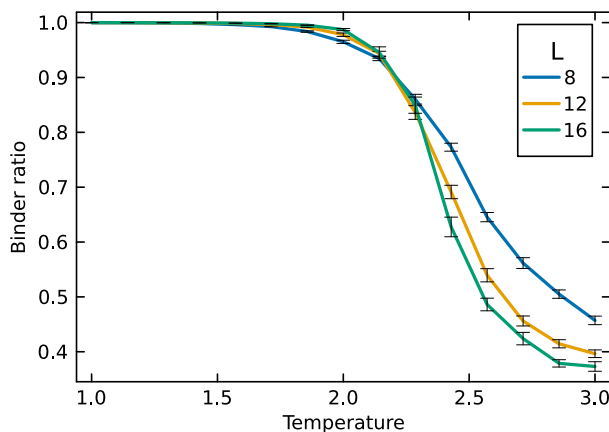
Figure 2: **Binder ratio**, $\left\langle m^2 \middle| m^2 \right\rangle^2 / \left\langle m^4 \middle| m^4 \right\rangle$, of the magnetization m, in the $L \times L$ square-lattice Ising model as a function of temperature. As one would expect, there is a crossing close to the critical temperature $T_c \approx 2.269$. The error bars are calculated automatically by Carlo.jl using the jackknife method.

After the simulation has completed, we should see a directory `example_job.data`, which contains raw data and check points, and a file `example_job.results.json`, which contains the postprocessed results of our simulation. The submodule `Carlo.ResultTools` provides tools for reading these results back into Julia as follows:

```julia
using Plots
using DataFrames
using Carlo.ResultTools

df = DataFrame(ResultTools.dataframe("example_job.results.json"))
plot(df.T, df.BinderRatio;
    xlabel = "Temperature",
    ylabel = "Binder ratio",
    group = df.Lx,
    legendtitle = "L",
)
```

The output of this snippet can be seen in Fig. 2.

To inspect a raw Monte Carlo time series, we can read it directly from the respective HDF5 file. For example, for the first temperature and system size,

```julia
using HDF5

samples = h5read(
    "example_job.data/task0001/run0001.meas.h5",
    "observables/Energy/samples",
)
```

retrieves a vector containing each recorded sample of the energy.

In the example above, we ran Carlo.jl with a single processor. In order to use the parallel scheduler, we can simply launch the Carlo.jl CLI with MPI.

```
$ mpirun -n $ncores julia example_job.jl run
```

If the simulation has already been completed before, this will stop without doing further work. To restart the simulation, we should use the flag `run -r` or call the `delete` command before running.

In production runs, it is advisable to configure MPI.jl and HDF5.jl to use the system-provided binaries.

```
using MPIPreferences
MPIPreferences.use_system_binary()
using HDF5
HDF5.API.set_libraries!(
    "path/to/libhdf5.so",
    "path/to/libhdf5_hl.so",
)
```

Further, by default, Julia will use multithreading for linear algebra operations. Without care, together with MPI parallelization, this can cause overusage of the available CPUs, so it should be controlled, e.g. by setting the environment variable `OPENBLAS_NUM_THREADS=1`.

3.2 Stochastic series expansion

In the last section, we covered the Metropolis algorithm for the Ising magnet. More complicated models of magnetism – or other quantum phenomena – generally contain noncommuting operators, which require more sophisticated quantum Monte Carlo algorithms to solve. One quantum Monte Carlo method that is particularly effective for lattice models of bosons is the stochastic series expansion [6]. In basic terms, it works by expanding the trace over Boltzmann operator $Z = \text{Tr}\, e^{-\beta H}$ with the Hamiltonian H and the inverse temperature β into a power series and performing the resulting sums over over operator products using Monte Carlo sampling. More details can be found, e.g. in Ref. [15].

StochasticSeriesExpansion.jl [16] is an implementation of the abstract-loop flavor [17] of the algorithm that can flexibly handle any model consisting of bond operators acting on a graph of bonds (as long as the sign problem [18] is not severe). It is built in Julia on top of Carlo.jl, based on the C++ code that was used in earlier publications to simulate frustrated magnets in cluster bases [19–21] and spin-photon coupled systems [22].

While this arbitrary model capability requires some code to specify the Hamiltonians and associated observables, some models are supported out of the box. One example, that we will explore in this section are arbitrary-spin anisotropic Heisenberg models of the form

$$H = \sum_{\langle i,j|i,j\rangle} J_{ij}\left(\mathbf{S}_i \cdot \mathbf{S}_j + d_{ij}S_i^z S_j^z\right) + \sum_i h_i^z S_i^z + D_i^x (S_i^x)^2 + D_i^z (S_i^z)^2, \tag{5}$$

where $\langle i,j|i,j\rangle$ sums over the bonds of an arbitrary lattice. As long as the signs of J_{ij} and D_i^x in combination with the bonds of the lattice do not imply magnetic frustration, the model is sign-problem free. This is the case, e.g., for $J_{ij} > 0$ on a bipartite lattice if D_i^x does not change sign throughout the lattice.

As a registered package, we can install StochasticSeriesExpansion.jl (and Carlo.jl if we have not already) from the Julia REPL.

```
julia>]
pkg> add StochasticSeriesExpansion
```

Then we can write a job script. In this example, we calculate the susceptibility of an effective model for the anisotropic antiferromagnet $BaNi_2V_2O_8$ from Ref. [23], which is also available as an example in the StochasticSeriesExpansion.jl documentation.

```
# bani2v2o8.jl

using Carlo
using Carlo.JobTools
```

```
using StochasticSeriesExpansion

tm = TaskMaker()
tm.sweeps = 80000
tm.thermalization = 10000
tm.binsize = 100

temperatures = range(0.05, 4, 20)
system_sizes = [10, 20]

tm.model = MagnetModel
tm.S = 1
tm.J = 1
tm.Dz = 0.005645 # D^{QMC}_{EP(XY)}/J^{QMC}_n from PRB 104, 065502

tm.measure = [:magnetization]
for L in system_sizes
    tm.lattice = (unitcell = UnitCells.honeycomb, size = (L, L))

    for T in temperatures
        tm.T = T
        task(tm)
    end
end

job = JobInfo(
    splitext(@__FILE__)[1],
    StochasticSeriesExpansion.MC;
    run_time = "24:00:00",
    checkpoint_time = "30:00",
    tasks = make_tasks(tm),
)

start(job, ARGS)
```

The meanings of the StochasticSeriesExpansion.jl-specific parameters are the following:

model the model to simulate, here `MagnetModel` which corresponds to Eq. (5).

S the spin magnitude.

T the temperature.

J,d,h,Dx,Dz correspond to the parameters in Eq. (5). Can also be made site-dependent, see StochasticSeriesExpansion.jl documentation.

lattice the unit cell and the dimensions of the supercell. Honeycomb and other popular ones are predefined. Otherwise, an instance of `UnitCell` can be passed.

measure specifies the observables to be measured (on top of the energy and specific heat). Here, just measure uniform magnetization-related quantities. `:staggered_magnetization` would additionally compute the staggered magnetization. There is an operator string estimator interface for implementing custom observable estimators.

For details, the reader is referred to the StochasticSeriesExpansion.jl documentation.

Even though the run time scaling of stochastic series expansion is linear in system size and inverse temperature, this example takes more computational effort than the previous one for the Ising model. It is therefore recommended to run it with MPI.
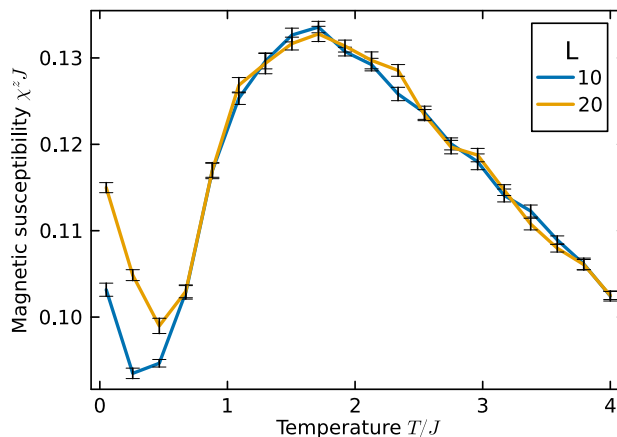
Figure 3: **Example calculation with StochasticSeriesExpansion.jl**. The magnetic susceptibility perpendicular to the easy plane, χ^z, of an effective model for $BaNi_2V_2O_8$, as a function of the temperature T.

```
$ mpirun -n $ncores julia example_job.jl run
```

Nevertheless, it is still tractable within a few minutes on a workstation or strong laptop and should yield the result file `bani2v2o8.result.json`. Like in the Ising example, we can use `Carlo.ResultTools` to plot the observable `MagChi` which corresponds to the static uniform magnetic susceptibility Fig. 3. We observe the dip feature that is caused by the easy-plane anisotropy D^z and is also observed in experiment [23]. However, this feature is not quite converged with system size at $L = 20$ and larger systems should be simulated in practice.

This concludes the present example, which showcases the effectiveness of a state-of-the-art Monte Carlo code built on Carlo.jl. Since this code was ported from an earlier C++ version, it also gives us an opportunity to directly compare the performance between the two languages, which we will do in Section 4.

3.3 Parallel tempering and the parallel run mode

Parallel tempering [24] is a technique that allows speeding up autocorrelation times in an ensemble of simulations by introducing additional Monte Carlo moves that exchange configurations between different simulation parameters. A typical application of this are glassy models, which can be simulated ergodically at high temperatures but freeze at low temperatures. If we simulate a chain of increasing temperatures in parallel, the probability distributions of adjacent temperatures may have some overlap so that higher temperature configurations may be accepted as updates in lower temperature simulations and vice versa. This leads to a steady exchange of configurations up and down the chain of temperatures, which can decrease the overall autocorrelation time significantly.

Carlo supports parallel tempering through the `ParallelTemperingMC` wrapper, which is a meta-implementation of the `AbstractMC` interface that allows to run other implementations with parallel tempering. We will first illustrate its usage with an example that is of interest to readers planning to use parallel tempering. Afterwards, we sketch its implementation as an example for implementing new algorithms with the parallel run mode feature in Carlo.jl.

3.3.1 Usage

Parallel tempering can be used with any implementation of the `AbstractMC` interface as long as the following methods are implemented:

```
Carlo.parallel_tempering_log_weight_ratio(
    mc::MC, parameter::Symbol, new_value
)

Carlo.parallel_tempering_change_parameter!(
    mc::MC, parameter::Symbol, new_value
)
```

Both of them receive the name of the `parameter` along which parallel tempering is performed. In a tempering update, this parameter may be changed to a `new_value`. The first function returns the logarithmic weight change $\log[W(p')/W(p)]$ that would result from changing the `parameter` from p to p'. The second function puts that same change into action. Both the Ising reference implementation and `StochasticSeriesExpansion.jl` implement this interface and can be used with `ParallelTemperingMC`.

    Parallel tempering is useful in situations where the untempered autocorrelation time of a simulation is longer than the time it takes to exchange replicas in the tempering process. Such a situation is not easy to achieve for the SSE algorithm in standard models due to its efficient global loop updates, but can be useful in more involved situations [25]. To keep the presentation of this usage example transparent, we will not introduce an involved model here and instead return to the Ising reference implementation that only contains local updates and is therefore known to have long autocorrelation times around the critical point.

    Using `ParallelTemperingMC` requires just a few changes to the job script of Listing 1,

```julia
# parallel_tempering.jl

using Carlo
using Carlo.JobTools
using Ising

tm = TaskMaker()

tm.sweeps = 20000
tm.thermalization = 2000
tm.binsize = 100

Ts = range(1.5, 4, 30)

# contract temperatures around Tc for better distribution overlap
Tc = 2.269
Ts += 0.5 .* (Tc .- Ts) ./ (0.6 .+ (Tc .- Ts) .^ 2)


tm.parallel_tempering = (
    mc = Ising.MC,
    parameter = :T,
    values = Ts,
    interval = 1
)

Ls = [32, 64]
for L in Ls
    tm.Lx = L
    tm.Ly = L
```

```
    # tm.T is set implicitly by ParallelTemperingMC

    task(tm)
end

job = JobInfo(
    splitext(@__FILE__)[1],
    ParallelTemperingMC; # underlying MC set in tm.parallel_tempering
    run_time = "24:00:00",
    checkpoint_time = "30:00",
    tasks = make_tasks(tm),
    ranks_per_run = length(tm.parallel_tempering.values),#needs to match!
)

start(job, ARGS)
```

Most notably, we replace the MC algorithm in `JobInfo` by `ParallelTemperingMC` and use the parameter `tm.parallel_tempering` to specify the details of how parallel tempering should be performed as follows

mc the underlying model MC implementation.

parameter name of the parameter along which parallel tempering is performed, e.g. `:T` for the temperature.

values a chain of values of the parameter, which is simulated in parallel with nearest neighbor replica exchange.

interval the number of sweeps between proposing replica exchanges.

Lastly, `ParallelTemperingMC` has to be run in parallel run mode, which as of Carlo v0.2.3 has to be enabled manually by setting the `ranks_per_run` option in `JobInfo`. In the case of `ParallelTemperingMC`, the number of ranks per run has to be set to the number of parameters in the parallel tempering chain.

Running the job script works just as in the previous examples,

```
$ mpirun -n $N julia parallel_tempering.jl run
```

except running with MPI is required and the number of MPI ranks N has to be equal to `m * ranks_per_run + 1` with some integer m. For the example above, $N = 61$ will simulate the two length-30 parallel tempering chains for $L = 32$ and $L = 64$ in parallel.

As shown in Fig. 4, in this case, parallel tempering greatly decreases the statistical uncertainties close to the critical point and cures thermalization issues that arise at low temperatures. It should be mentioned that in the case of the Ising model, the critical slowing down can be more efficiently cured by introducing cluster updates [26]. However, designing such global updates is not always possible for more complicated models.

3.3.2 Implementation

In the last subsection, we explained the usage of the parallel tempering feature. We now sketch its underlying implementation as an example of writing an algorithm with the parallel run mode feature of Carlo.jl.

In general, introducing parallel tempering into a simulation greatly increases the complexity of the parallel scheduling, since multiple simulations need to be orchestrated to run in lock step. However, Carlo.jl already supports this kind of parallelism through its parallel run mode. Parallel tempering (and similar meta-algorithms) can therefore be implemented as a thin layer on top of parallel run mode.
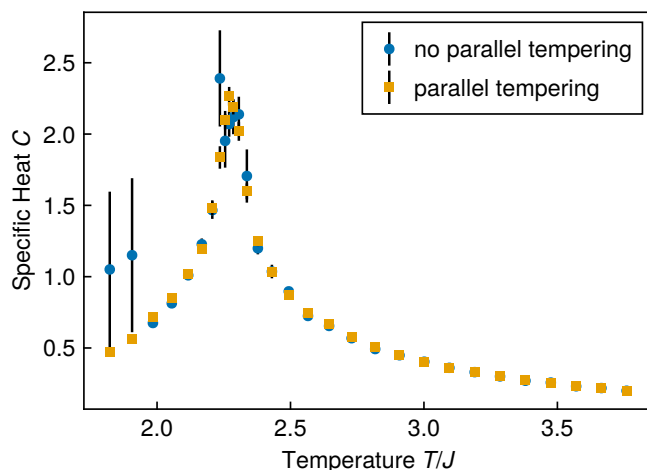
Figure 4: **Parallel tempering in the Ising model.** The specific heat of the 64×64 square lattice Ising model as a function of the temperature, simulated with and without parallel tempering. Both simulations are performed with 20000 measurement sweeps and 2000 thermalization sweeps.

To use parallel run mode, algorithms have to implement an alternative version of the `AbstractMC` interface, which essentially receives an additional MPI communicator (Table 1) spanning a team of MPI ranks that are collaborating on a single Carlo.jl run (that is, a single bar in Fig. 1). In the parallel tempering scenario, we thus assign one chain of the tempered parameter to one Carlo run, and write a meta-implementation of the parallel run `AbstractMC` interface that runs the simulations for each value of the parameter.

```
mutable struct ParallelTemperingMC{T} <: AbstractMC
    parameter_name::Symbol
    parameter_values::Vector{T}
    tempering_interval::Int

    parallel_measure::ParallelMeasurements
    chain_idx::Int
    child_mc::AbstractMC
end
```

The name of the tempered parameter is stored in `parameter_name` with the values stored in `parameter_values`. Tempering updates are performed every `tempering_interval` sweeps, and `ParallelMeasurements` is a helper object for performing distributed measurements. Each instance of `ParallelTemperingMC` corresponds to one link in the parallel tempering chain and a single parameter value. `chain_idx` stores the current position of the link and `child_mc` stores a child Monte Carlo implementation that does the simulation between the tempering updates.

Most of the implementation work is then to implement the interface functions and calling the respective function of the child implementation, making it believe that it is running normally. In the constructor and the initialization function, we retrieve the configuration and inject the current value of the tempering parameter into the normal parameter set before passing it on to the child.

Table 1: Comparison between the standard single run interface for a type `MC <: AbstractMC`, where one Carlo.jl *run* (i.e. one simulation of a single parameter set) is handled by one MPI rank, to the parallel run interface, where the work of one run is shared among a team of MPI ranks. The communicator comm passed as an additional argument spans this team of ranks. *The constructor has the same signature, but additionally, `params[:_comm]` contains the MPI communicator. †In these methods, only the 0th rank of comm is allowed to call `measure!(ctx,...)`, read, or write to HDF5. The second argument of the checkpointing functions is `nothing` on the other ranks.

Single run interface

```
MC(parameters::AbstractDict)
init!(::MC, ::MCContext, params::AbstractDict)
sweep!(::MC, ::MCContext)
measure!(::MC, ::MCContext)
register_evaluables(::TypeMC, ::AbstractEvaluator, params::AbstractDict)
write_checkpoint(::MC, ::HDF5.Group)
read_checkpoint!(::MC, ::HDF5.Group)
```

Parallel run interface

```
MC(parameters::AbstractDict)*
init!(::MC, ::MCContext, params::AbstractDict, comm::MPI.Comm)
sweep!(::MC, ::MCContext, comm::MPI.Comm)
measure!(::MC, ::MCContext, comm::MPI.Comm)†
register_evaluables(::Type{MC}, ::AbstractEvaluator, params::AbstractDict)
write_checkpoint(::MC, ::Union{HDF5.Group,Nothing}, comm::MPI.Comm)†
read_checkpoint!(::MC, ::Union{HDF5.Group,Nothing}, comm::MPI.Comm)†
```

```julia
function ParallelTemperingMC(params::AbstractDict)
    config = params[:parallel_tempering]
    MC = config.mc
    tempering_interval = config.interval

    comm = params[:_comm]
    chain_idx = MPI.Comm_rank(comm) + 1

    modified_params = deepcopy(params)
    modified_params[config.parameter] = config.values[chain_idx]

    return ParallelTemperingMC(
        config.parameter,
        collect(config.values),
        config.interval,
        ParallelMeasurements(),
        chain_idx,
        MC(modified_params),
    )
end

function Carlo.init!(
    mc::ParallelTemperingMC,
    ctx::MCContext,
```

```
    params :: AbstractDict ,
    comm :: MPI.Comm,
)
    modified_params = deepcopy(params)
    modified_params[mc.parameter_name] = mc.parameter_values[mc.chain_idx]

    Carlo.init!(mc.child_mc, ctx, modified_params)
end
```

In the sweep! function we let the child do sweeps, performing a tempering update periodically.

```
function Carlo.sweep!(
    mc::ParallelTemperingMC, ctx::MCContext, comm::MPI.Comm
)
    Carlo.sweep!(mc.child_mc,
        make_parallel_context(ctx, mc.parallel_measure))

    if ctx.sweeps % mc.tempering_interval == 0
        synchronize_measurements!(
            ctx, mc.parallel_measure, mc.chain_idx, comm
        )
        tempering_update!(mc, ctx, comm)
    end
    return nothing
end
```

The function `make_parallel_context` creates a modified `MCContext` that reroutes possible measurements using `mc.parallel_measure`.

Then, periodically, `synchronize_measurements!` performs the necessary communications that collect these measurements from different ranks and combine them into vectors in the correct order. This is necessary because in parallel run mode, only the zeroth rank of the comm is allowed to do measurements. Afterwards, `tempering_update!` implements a standard neighbor swapping update [24] calling the functions `parallel_tempering_log_weight_ratio` to compare weight ratios and, if accepted, `parallel_tempering_change_parameter!` to switch the tempering parameter values. For further details, the interested reader is referred to the source code (`src/parallel_tempering.jl`).

For the measurements, analogously, we call the `Carlo.measure!` function on the child.

```
function Carlo.measure!(
    mc::ParallelTemperingMC, ctx::MCContext, comm::MPI.Comm
)
    Carlo.measure!(mc.child_mc,
        make_parallel_context(ctx, mc.parallel_measure)
    )
end
```

The `register_evaluables` function works the same in single run and parallel run mode. There is no MPI communication as this function is called only once per run in the postprocessing phase, after the actual simulation is already done. For the parallel tempering implementation, this still takes some bookkeeping, as we need to string together the evaluables from the child implementations in a similar way as the measurements. We will skip these details here for brevity.

Finally, the checkpointing functions require some more MPI communication. The second argument in both of them is `nothing` on any rank except the 0th rank of comm. We therefore communicate the state, given by the chain permutation, the parallel measurement object, and the internal state of the child implementations, before reading from or writing to the HDF5 file.

```julia
function Carlo.write_checkpoint(
    mc::ParallelTemperingMC,
    out::Union{HDF5.Group,Nothing},
    comm::MPI.Comm,
)
    chain_permutation = MPI.Gather(mc.chain_idx, comm)
    child_mcs = MPI.gather(mc.child_mc, comm)
    parallel_measures = MPI.gather(mc.parallel_measure, comm)

    if MPI.Comm_rank(comm) == 0
        out["chain_permutation"] = chain_permutation

        for (i, (child_mc, parallel_measure)) in
            enumerate(zip(child_mcs, parallel_measures))
            Carlo.write_checkpoint(child_mc,
                create_group(out, "child_mcs/$i"))
            Carlo.write_checkpoint(parallel_measure,
                create_group(out, "parallel_measures/$i"))
        end
    end
    return nothing
end

function Carlo.read_checkpoint!(
    mc::ParallelTemperingMC,
    in::Union{HDF5.Group,Nothing},
    comm::MPI.Comm,
)
    child_mcs = MPI.gather(mc.child_mc, comm)

    if MPI.Comm_rank(comm) == 0
        chain_permutation = read(in, "chain_permutation")
        parallel_measures = [
            Carlo.read_checkpoint(ParallelMeasurements,
            in["parallel_measures/$i"])
            for i in eachindex(child_mcs)
        ]

        for (i, child_mc) in enumerate(child_mcs)
            Carlo.read_checkpoint!(child_mc, in["child_mcs/$i"])
        end
    else
        chain_permutation = nothing
    end

    mc.chain_idx = MPI.scatter(chain_permutation, comm)
    mc.child_mc = MPI.scatter(child_mcs, comm)

    mc.parallel_measure = MPI.scatter(parallel_measures, comm)

    parallel_tempering_change_parameter!(
        mc.child_mc,
        mc.parameter_name,
        mc.parameter_values[mc.chain_idx],
    )
```

```
    return nothing
end
```

This concludes the implementation of `ParallelTemperingMC` as an example of leveraging the parallel run mode feature. While some MPI communication and bookkeeping is required, the heavy lifting in terms of scheduling is being done behind the scenes by Carlo.jl. The main source of complexity in the parallel tempering implementation is the design requirement that it should work transparently with normal `AbstractMC` implementations. Utilities for this kind of interfacing, such as the `ParallelMeasurements` used above may also be useful for other meta-algorithms. While they are not currently part of the public Carlo.jl API, this may change in a future release.

4 Benchmarks

In this section, we provide benchmarks to show (i) the run time scaling of the parallel scheduler and (ii) the validity of the statistical postprocessing in the face of strongly autocorrelated data. For both, we use the Ising example code implemented in Section 3.1. Finally, we (iii) directly compare the performance of the Julia and C++ versions of the stochastic series expansion code.

4.1 Scheduler run time scaling

For the run time scaling, we consider a workload inspired by Listing 1, where we make the replacements `Ls = [100, 200, 300, 500]` and `sweeps = 200000` to increase the computational effort. Since the Ising code scales quadratically with the linear system size `Lx`, the tasks of this workload are quite imbalanced, comparable to the situation in Fig. 1. Nevertheless, we observe nearly linear scaling of the run time with number CPUs used (Fig. 5). We expect that the deviation from perfect scaling in this case is dominated by the serial cost of thermalization, which, based on the number of sweeps, is about 1% of the total serial run time.
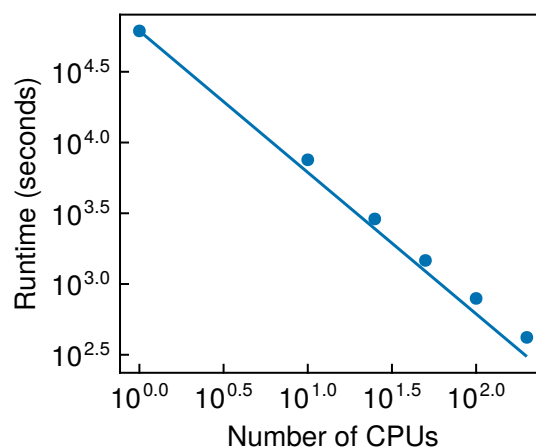

Figure 5: **Run time scaling with number of CPUs** of the example job script from Listing 1, with the replacements `Ls = [100, 200, 300, 500]` and `sweeps = 200000`. The solid line shows the ideal linear speedup.

4.2 Validity of the postprocessing

To confirm the accuracy of the statistical postprocessing, we consider the Ising model with $L = 20$ and $T = 2.3$, close to the critical point. In this region, the local updates of the Metropolis algorithm that we implemented are quite inefficient, leading to long autocorrelation times that require a careful analysis to retrieve the correct statistics.

To obtain the statistics of this model, there are two ways. The first one is to run many identical, independent simulations and create a histogram of the results. The second, more practical approach is to run a single simulation and perform a binning analysis on the correlated time series. By comparing the histogram of many independent simulations with the average results from the binning analysis implemented in Carlo.jl (Fig. 6(a)), we confirm that the binning procedure outlined in Section 2.3 yields the correct errors for observables. As an example for the jackknifing feature, the Binder ratio (Fig. 6(b)) also shows correct statistics. Finally, in Fig. 6(c) we show the autocorrelation function for the energy per spin. We find that the autocorrelation time estimated from the binning analysis using Eq. (3) is a good estimate for the true decay of the autocorrelations.

4.3 Performance comparison between Julia and C++

The StochasticSeriesExpansion.jl package mentioned in Section 3.2 is a Julia translation of the earlier C++ code [27] that was used in Refs. [17, 19–22]. Since many of their inner workings are identical and both codes have seen a similar amount of optimization efforts, they provide an ideal opportunity for benchmarking the performance of Julia for quantum Monte Carlo applications, which we do in the following.
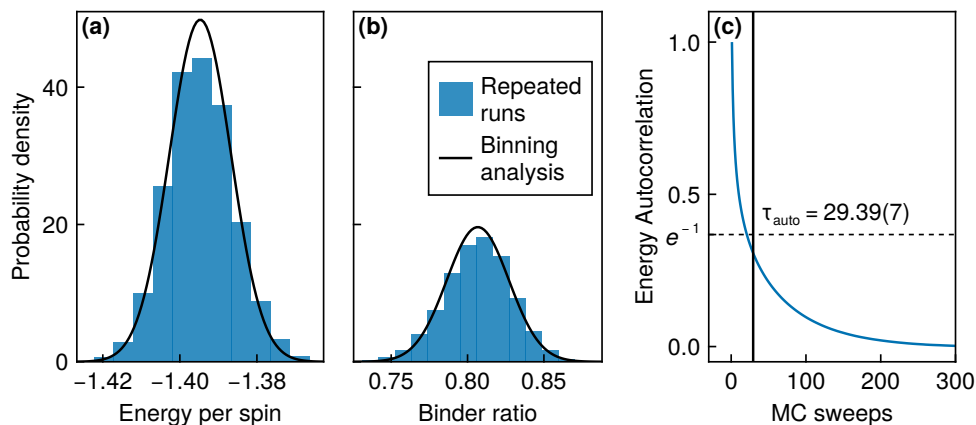

Figure 6: **Binning analysis benchmark** using the 20×20 square-lattice Ising model at temperature $T = 2.3$, close to the critical point. The same simulation with 20000 sweeps and 4000 thermalization sweeps is repeated 9000 times. We then compare the distribution of these results to the distribution predicted by the binning analysis. (a) Histogram of the energy per spin compared to a Gaussian distribution with the average mean and standard deviation predicted from the binning analysis. (b) Histogram of the Binder ratio compared to a Gaussian distribution with the average mean and standard deviation predicted from jackknifing. (c) The energy autocorrelation function, averaged over all runs, compared to the autocorrelation time τ_{auto} predicted by Carlo.jl.
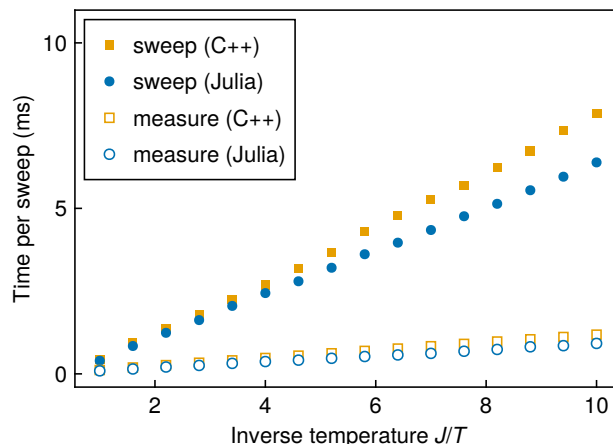
Figure 7: **Language performance comparison** between the Julia and C++ implementations of the same SSE QMC algorithm. Shown are the average times spent on a sweep and a measurement as a function of inverse temperature $\beta = 1/T$ for the fully frustrated bilayer with interlayer to intralayer coupling ratio $J/J_\perp = 2$ and 30×30 dimer unit cells.

Although the codes behave identically at the core, a number of details need to be taken into account for a quantitative performance comparison. Firstly, the models implemented for use with them are not the same. For example, the anisotropic magnet from Section 3.2 is not implemented in the C++ code. A model that is implemented for both codes is the spin-1/2 fully-frustrated bilayer [28, 29] in the interlayer dimer basis, which we therefore use here. It is accessible through the `ClusterModel` in StochasticSeriesExpansion.jl. Another, more technical intricacy are the length of the SSE operator string and the number of worm updates. Both are parameters that are automatically tuned by the algorithm during its thermalization phase. Both codes use exactly the same controller logic for this, however the result is subject to noise. To eliminate this noise, we set both of them to the same values, predetermined by an earlier simulation. Lastly, the C++ version used a custom implementation of the mt19937 Mersenne Twister [30]. Carlo.jl uses Julia's `Random.Xoshiro` by default, which for this comparison, we replace by `Random.MersenneTwister`. All of these settings are implemented in the job script from Listing 2. The C++ code was compiled with GCC 12.2.0 with `-march=znver4 -ffast-math` and link-time optimization enabled. The Julia version was 1.11.1. The two simulations were carried out on AMD EPYC 9474F CPUs with 16 MPI ranks at a time.

In Fig. 7, both codes show the characteristic linear scaling of the SSE algorithm with inverse temperature. The Julia version shows slightly better performance than the C++ version for both sweeps and measurements. This should not be seen as proof that one language is inherently faster or slower, as further optimizations on either side are likely possible.

Some plausible explanations for the different performance could be the following. First, the sweep time could be partly affected by the different performance of the Mersenne Twister implementations. Second, due to just-in-time compilation, the Julia compiler has more compile-time knowledge. For example, both codes allow the SSE bond operators to connect an arbitrary number of sites. If the number of sites per operator is known at compile-time, which has been ensured in the Julia code, several expensive integer divisions can be replaced by cheaper bit shift operations. Achieving something similar in C++ is possible using sophisticated templating techniques, however, in the present code, this is only done in part. Lastly, the reason could be a random difference in the LLVM optimizer, used by Julia, and the GCC optimizer, used for C++ here, that happens to be relevant for the performance of the given code on the given architecture.

```julia
using Carlo
using Carlo.JobTools
using StochasticSeriesExpansion
using Random

tm = TaskMaker()
tm.sweeps = 100000
tm.thermalization = 10000
tm.binsize = 100

tm.model = ClusterModel
tm.inner_model = MagnetModel
tm.cluster_bases = (StochasticSeriesExpansion.ClusterBases.dimer,)

tm.measure_quantum_numbers = [(;name=Symbol(), quantum_number=2)]
tm.parameter_map = (;
    J = vcat([:Jperp], repeat([:Jpar],8))
)

tm.Jpar = 1
tm.Jperp = 0.5

betas = range(1,10,16)
L = 30

tm.lattice = (
    unitcell = UnitCells.fully_frust_square_bilayer,
    size=(L,L)
)
for beta in betas
    tm.init_opstring_cutoff = 160000*beta/maximum(betas)
    tm.init_num_worms = 8
    tm.num_worms_attenuation_factor = 0.0
    tm.T = 1/beta
    tm.Lx = L
    tm.Ly = L
    task(tm)
end

job = JobInfo(
    "path/to/job",
    StochasticSeriesExpansion.MC;
    run_time = "24:00:00",
    checkpoint_time = "30:00",
    rng = Random.MersenneTwister,
    tasks = make_tasks(tm),
)

start(job, ARGS)
```

Listing 2: The StochasticSeriesExpansion.jl job file used for the performance comparison in Fig. 7.

Despite these technical details that make performance comparisons of this kind hard to interpret, we can confidently conclude that it is possible to write state-of-the-art QMC codes in Julia without paying a price in performance.

5 Conclusion

Carlo.jl is a framework for writing Monte Carlo simulation codes in Julia. It features a Monte Carlo aware parallel scheduler with parallel tempering support, organized storage of input, checkpoint, and result files, as well as statistical postprocessing of the Monte Carlo results.

We have illustrated these features and the general usage of the framework by providing examples for the Ising model, the SSE QMC algorithm, and parallel tempering. Our benchmarks show the efficient run time scaling of the scheduler and the correctness of the statistical post-processing. Additionally, since the SSE QMC code is based on an earlier C++ implementation, we performed a language performance comparison, in which the Julia version was faster, demonstrating the applicability of Julia for Monte Carlo workloads.

Based on these features and properties, the availability of a framework like Carlo.jl should enable the quick development of new user-friendly high-performance Monte Carlo codes in Julia.

Acknowledgments

References

[1] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller and E. Teller, *Equation of state calculations by fast computing machines*, J. Chem. Phys. **21**, 1087 (1953), doi:10.1063/1.1699114.

[2] J. Gubernatis, N. Kawashima and P. Werner, *Quantum Monte Carlo methods*, Cambridge University Press, Cambridge, UK, ISBN 9781107006423 (2016), doi:10.1017/CBO9780511902581.

[3] B. Bauer et al., *The ALPS project release 2.0: Open source software for strongly correlated systems*, J. Stat. Mech.: Theory Exp. P05001 (2011), doi:10.1088/1742-5468/2011/05/P05001.

[4] A. Gaenko et al., *Updated core libraries of the ALPS project*, Comput. Phys. Commun. **213**, 235 (2017), doi:10.1016/j.cpc.2016.12.009.

[5] J. Bezanson, S. Karpinski, V. B. Shah and A. Edelman, *Julia: A fast dynamic language for technical computing*, (arXiv preprint) doi:10.48550/arXiv.1209.5145.

[6] A. W. Sandvik, *Stochastic series expansion method with operator-loop update*, Phys. Rev. B **59**, R14157 (1999), doi:10.1103/PhysRevB.59.R14157.

[7] S. Byrne, L. C. Wilcox and V. Churavy, *MPI.jl: Julia bindings for the message passing interface*, JuliaCon Proc. **1**, 68 (2021), doi:10.21105/jcon.00068.

[8] D. Robinson et al., *Hierarchical data format, version 5*, https://github.com/HDFGroup/hdf5.

[9] L. Van Munoz et al., *HDF5.jl*, https://github.com/JuliaIO/HDF5.jl.

[10] J. Isensee et al., *JLD2.jl*, https://github.com/JuliaIO/JLD2.jl.

[11] M. Bouchet-Valat and B. Kamiński, *DataFrames.jl: Flexible and fast tabular data in Julia*, J. Stat. Softw. **107**, 1 (2023), doi:10.18637/jss.v107.i04.

[12] V. Ambegaokar and M. Troyer, *Estimating errors reliably in Monte Carlo simulations of the Ehrenfest model*, Am. J. Phys. **78**, 150 (2010), doi:10.1119/1.3247985.

[13] R. G. Miller, *The Jackknife – A review*, Biometrika **61**, 1 (1974), doi:10.2307/2334280.

[14] L. Weber, *Ising reference implementation from section 3*, Zenodo (2024), doi:10.5281/zenodo.12981007.

[15] A. W. Sandvik, A. Avella and F. Mancini, *Computational studies of quantum spin systems*, AIP Conf. Proc. **1297**, 135 (2010), doi:10.1063/1.3518900.

[16] L. Weber, *StochasticSeriesExpansion.jl*, Zenodo (2024), doi:10.5281/zenodo.13993851.

[17] L. Weber, A. Honecker, B. Normand, P. Corboz, F. Mila and S. Wessel, *Quantum Monte Carlo simulations in the trimer basis: First-order transitions and thermal critical points in frustrated trilayer magnets*, SciPost Phys. **12**, 054 (2022), doi:10.21468/SciPostPhys.12.2.054.

[18] G. Pan and Z. Y. Meng, *The sign problem in quantum Monte Carlo simulations*, in *Encyclopedia of condensed matter physics*, Elsevier, Amsterdam, Netherlands, ISBN 9780323914086 (2024), doi:10.1016/B978-0-323-90800-9.00095-0.

[19] J. L. Jiménez et al., *A quantum magnetic analogue to the critical point of water*, Nature **592**, 370 (2021), doi:10.1038/s41586-021-03411-8.

[20] L. Weber, N. Caci and S. Wessel, *Cluster quantum Monte Carlo study of two-dimensional weakly coupled frustrated trimer antiferromagnets*, Phys. Rev. B **106**, 035141 (2022), doi:10.1103/PhysRevB.106.035141.

[21] L. Weber, A. Y. D. Fache, F. Mila and S. Wessel, *Thermal critical points from competing singlet formations in fully frustrated bilayer antiferromagnets*, Phys. Rev. B **106**, 235128 (2022), doi:10.1103/PhysRevB.106.235128.

[22] L. Weber, E. Viñas Boström, M. Claassen, A. Rubio and D. M. Kennes, *Cavity-renormalized quantum criticality in a honeycomb bilayer antiferromagnet*, Commun. Phys. **6**, 247 (2023), doi:10.1038/s42005-023-01359-x.

[23] E. S. Klyushina, J. Reuther, L. Weber, A. T. M. N. Islam, J. S. Lord, B. Klemke, M. Månsson, S. Wessel and B. Lake, *Signatures for Berezinskii-Kosterlitz-Thouless critical behavior in the planar antiferromagnet* $BaNi_2V_2O_8$, Phys. Rev. B **104**, 064402 (2021), doi:10.1103/PhysRevB.104.064402.

[24] D. J. Earl and M. W. Deem, *Parallel tempering: Theory, applications, and new perspectives*, Phys. Chem. Chem. Phys. **7**, 3910 (2005), doi:10.1039/B509983H.

[25] E. Torres, L. Weber, L. Janssen, S. Wessel and M. M. Scherer, *Emergent symmetries and coexisting orders in Dirac fermion systems*, Phys. Rev. Res. **2**, 022005 (2020), doi:10.1103/PhysRevResearch.2.022005.

[26] U. Wolff, *Collective Monte Carlo updating for spin systems*, Phys. Rev. Lett. **62**, 361 (1989), doi:10.1103/PhysRevLett.62.361.

[27] L. Weber, *frust*, Zenodo (2024), doi:10.5281/zenodo.14205557.

[28] E. Müller-Hartmann, R. R. P. Singh, C. Knetter and G. S. Uhrig, *Exact demonstration of magnetization plateaus and first-order Dimer-Néel phase transitions in a modified Shastry-Sutherland model for* $SrCu_2(Bo_3)_2$, Phys. Rev. Lett. **84**, 1808 (2000), doi:10.1103/PhysRevLett.84.1808.

[29] F. Alet, K. Damle and S. Pujari, *Sign-problem-free Monte Carlo simulation of certain frustrated quantum magnets*, Phys. Rev. Lett. **117**, 197203 (2016), doi:10.1103/PhysRevLett.117.197203.

[30] M. Matsumoto and T. Nishimura, *Mersenne twister*, ACM Trans. Model. Comput. Simul. **8**, 3 (1998), doi:10.1145/272991.272995.

[31] L. Weber, *loadleveller: C++ Monte Carlo framework*, Zenodo (2022), doi:10.5281/zenodo.12982245.

[32] S. Danisch and J. Krumbiegel, *Makie.jl: Flexible high-performance data visualization for Julia*, J. Open Source Softw. **6**, 3349 (2021), doi:10.21105/joss.03349.