A practical guide to replica-exchange Wang–Landau simulations

Thomas Vogel¹, Ying Wai Li² and David P Landau³

¹ Department of Physics, University of North Georgia, Dahlonega, GA 30597, USA
² National Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
³ Center for Simulational Physics, The University of Georgia, Athens, GA 30602, USA

E-mail: tavogel@ung.edu, yingwaili@ornl.gov, dlandau@physast.uga.edu

Abstract. This paper is based on a series of tutorial lectures about the replica-exchange Wang–Landau (REWL) method given at the IX Brazilian Meeting on Simulational Physics (BMSP 2017). It provides a practical guide for the implementation of the method. A complete example code for a model system is available online. In this paper, we discuss the main parallel features of this code after a brief introduction to the REWL algorithm. The tutorial section is mainly directed at users who have written a single-walker Wang–Landau program already but might have just taken their first steps in parallel programming using the Message Passing Interface (MPI). In the last section, we answer “frequently asked questions” from users about the implementation of REWL for different scientific problems.

1. Introduction

Since its introduction in 2013 [1, 2], the replica-exchange Wang–Landau (REWL) method has been used to study a variety of problems, amongst them the self-assembly of lipid bilayer membranes [3, 4], polymer adsorption in the hydrophobic-polar (HP) model [5], magnetic phase transition in coupled spin-lattice systems [6], higher-dimensional parameter space sampling [7], fully-connected magnetic models of dipoles [8], network reliability [9, 10], stochastic analytic continuation of quantum Monte Carlo data [11], or ferroelectric phase transitions [12].

REWL combines the Wang–Landau (WL) method [13, 14] with elements from canonical replica-exchange methods, often termed ”parallel tempering” (PT) [15, 16]. The idea was to develop a framework that is able to run on large-scale supercomputers while still leveraging the advantages of the WL method on the individual computing cores. However, we understand that there might be considerable technical barriers for the users to overcome before they can actually start to experiment with REWL. One major advantage of the original, single-walker WL method that made it so popular and robust is that it is straightforward and easy to implement. Since

*This manuscript has been co-authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).
this is less so for REWL, we decided to write this paper to provide practical guidance and help. In Section 2, we first review the main ideas of REWL without going into too much detail conceptually. In Section 3 we introduce the basic functionality of the Message Passing Interface (MPI) before discussing, in great detail, essential code segments to convert a single-walker WL code into a parallel REWL version. A full REWL example code can be found online [17]. Finally, in Section 4 we answer questions concerning implementation details that were frequently asked via e-mail and during the IX Brazilian Meeting on Simulational Physics (BMSP 2017) and other meetings.

2. REWL: A brief description of the algorithm

Here we briefly outline the Replica–Exchange Wang–Landau (REWL) algorithm. For a detailed description of the algorithm, readers are encouraged to refer to the original publications [1, 2] and the corresponding conference proceedings [3, 5]. To begin with, the first step is to determine the relevant energy range over which the physical system can range, as in Wang–Landau sampling. This total energy range is then divided into multiple, smaller, overlapping windows (Figure 1). This approach is similar to an early attempt to parallize Wang–Landau sampling [18], with two major differences: in the previous scheme, the simulations for different energy windows were independent and could be run at different times. In REWL, a single simulation takes place with all energy windows running individual Wang–Landau sampling processes at the same time. This is necessary because at fixed intervals, i.e., every fixed number of Monte Carlo steps, replica-exchanges between the windows are performed. The probability of accepting a replica exchange, \( P(X \leftrightarrow Y) \), of configurations \( X \) and \( Y \) between walkers \( i \) and \( j \) is given by:

\[
P(X \leftrightarrow Y) = \min \left[ 1, \frac{g_i(E_X)}{g_i(E_Y)} \frac{g_j(E_Y)}{g_j(E_X)} \right],
\]

where \( g_i(E_X) \) is the current estimator for the density of states of walker \( i \) at the energy of its configuration \( X \).

Figure 1. A schematic diagram for replica–exchange Wang–Landau sampling. Each of the multiple, overlapping energy windows covers a part of the entire energy range in the original Wang–Landau sampling.

Every energy window performs one or more ordinary Wang–Landau sampling processes with their own sets of histogram and density of states, except for the occasional replica-exchanges. Since each of them proceeds with its own “pace” and could be at a different stage towards convergence, it should be noted that the simulation terminates only when the “slowest” window
has attained convergence, i.e., the pre-defined value of the modification factor is reached. Before that, every energy window has to keep performing Wang-Landau sampling even if its own density of states has converged, so as to facilitate the replica-exchanges with the windows that have not finished.

Upon completion, REWL yields a number of fragments of overlapping density of states. As the density of states is determined up to a constant, only the relative values, or the curvatures, are meaningful at this point. To combine two fragments of density of states, say the $i^{th}$ and the $j^{th}$ fragments $g_i(E)$ and $g_j(E)$, a joining point needs to be chosen as a reference point to rescale or shift one of the density of states fragments. The inverse microcanonical temperature $\beta = d\log[g(E)]/dE$ is first calculated for the overlapping regions, i.e., the derivatives of the logarithm of the density of states. Different numerical methods taking multiple bins into account can be used for this calculation. The joining point is chosen as the energy where the $\beta$-values from the two overlapping pieces best agree, denoted by $E_{join}$. $g_j(E)$ is then rescaled by the difference between itself and $g_i(E)$ at $E_{join}$, i.e., $g_{j,\text{rescaled}}(E) = g_j(E) \cdot g_i(E_{join})/g_j(E_{join})$. Once the density of states for the entire energy range is recovered, it can be normalized in the same way as if it had been obtained by the ordinary Wang–Landau scheme. Note that this scheme has been generalized and applied to reconstruct two-dimensional or higher-dimensional densities of states [5, 7].

3. REWL: A tutorial

3.1. MPI: the Message Passing Interface

Here, we further assume that each energy window performs its own Wang–Landau sampling on a single computing unit. The replica-exchange operations then require communications between a pair of computing units. Also, each computing unit needs to communicate with all other computing units from time to time to inform them of its progress towards convergence. In practice, such communications among computing units can be implemented using a parallel programming interface, the Message Passing Interface (MPI).

MPI is a standard, or specification, for a message passing library, which defines the application programming interface (API) of a message-passing system for communications between or among computing units. These can be CPU cores or compute nodes on a workstation, for instance. MPI was originally designed for distributed memory computing architectures (such as workstations), but it can also be used on shared memory architectures (e.g. multicore CPUs on a laptop). In both cases, MPI employs a distributed memory programming model (Figure 2), which assumes that each computing unit possesses independent memory and access to it is not shared with other computing units. There are different MPI implementations, such as MPICH [19], MVAPICH [20], and OpenMPI [21], to name a few. All of them provide C/C++ and Fortran interfaces. For interested readers there are a number of useful references [22–24] that provide more detail about MPI programming.

![Figure 2. Message Passing Interface (MPI) assumes distributed memory programming model.](image)

3.2. Hello World: Introduction to the necessary MPI functions

In the following sample code we demonstrate the very essential MPI functions necessary to initialize the MPI environment and to send and receive data between different computing cores in a C/C++ program.
### Example: basic point-to-point communication

```c
#include <stdio.h>
#include "mpi.h"   // Include MPI header file containing the library's API

int main (int argc, char* argv[]) {
    int my_rank, num_procs;

    // Start up MPI
    MPI_Init (&argc, &argv);

    // Get current process' "ID" number (rank)
    MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

    // Get total number of processes
    MPI_Comm_size (MPI_COMM_WORLD, &num_procs);

    // Store the status of whether a communication is successful or not
    MPI_Status status;

    printf("Hello World! This is process %d of %d
", my_rank, num_procs);

    int message = 42;       // Content of the message
    int msg_tag = 23;       // A unique tag for each message

    if (my_rank == 0)       // Process 0 receives a message from Process 1
    {                      
        MPI_Recv(&message, 1, MPI_INT, 1, msg_tag, MPI_COMM_WORLD, &status);
        printf("rank 0: recvd message %d from %d
", message, status.MPI_SOURCE);
    }
    else if (my_rank == 1)  // Process 1 sends a message to Process 0
    {                      
        MPI_Send(&message, 1, MPI_INT, 0, msg_tag, MPI_COMM_WORLD);
        printf("I am Rank 1. Sent a message to Rank 0.
");
    }
    else                   // All other processes sit and wait
        printf("I am Process %d. I have nothing to do.
", my_rank);

    MPI_Finalize();        // Finish up MPI

    return 0;
}
```

- **MPI_Init** initializes MPI with the command line arguments supplied to the program's executable.
- **MPI_Comm_rank** gets the rank number of the current MPI process inside the specific MPI communicator.
- **MPI_Comm_size** gets the total number of processes in the specific MPI communicator.
- **MPI_Finalize** finishes up and exits MPI.
More specifically, we demonstrate in this example a simple point-to-point communication where MPI rank 0 receives a message (data) containing an integer number (42) from MPI rank 1. If there are more than two MPI ranks initiated, these MPI ranks are not involved in the communication. MPI rank 0 calls MPI\_Recv, while MPI rank 1 calls MPI\_Send. In this example we use the global communicator MPI\_COMM\_WORLD, a predefined communication channel that includes all processes. Note that we will use dedicated communicators (channels) and more specialized communication functions later.

We suggest running this example code and familiarizing yourself with it. It includes all necessary concepts to understand the fundamental functionality of any MPI program. Note that the header file mpi.h which contains all the MPI application program interfaces (APIs) has to be included at the beginning of the program.

3.3. Parallel functions/extensions for REWL in detail

At this point we assume that the reader has a working Wang–Landau code and we discuss necessary extensions to convert that single-processor code into a parallel REWL code. While there are different ways of implementing each functionality, the basic design assumptions are the following:

- Every walker, performing ordinary Wang–Landau sampling, gets its own MPI process or rank.
- \(nm\) MPI processes are launched (\(m\) - number of walkers in each energy window; \(n\) - number of energy windows), each with its individual energy range, histogram, and density-of-states estimator. There is no “master” process.
- All data and replica-exchanges are facilitated exclusively via MPI communications.

All code snippets shown in the following are taken from the code used to produce the data presented in [1, 2]. Please feel free to download the full code online [17].

3.3.1. Preparations: setting up MPI and communicators

We will demonstrate in this section how individual communicators are set up and used instead of relying on the global communicator MPI\_COMM\_WORLD for communications. This could be advantageous for various reasons. First and foremost, this keeps the design clean and communications compartmentalized. Another benefit is that resilience measures can be added in a way that the failure of a single process or data transfer can be repaired and will not shut down the communications between uninvolved processes and eventually the whole simulation [25, 26].

Every process will be part of two separate communicators, one to communicate with all other walkers in the neighboring energy window on the left, the other to exchange data with any walker in the energy window on the right. For example, in Figure 3 walkers 4–6 would be in two communicators: one includes walkers 1–6, the other includes walkers 4–9. The communicators can be set up as follows.

```c
// Set up local MPI communicators for replica-exchange (RE)
// - each process belongs to two local groups/communicators
// - each process has different local IDs in different communicators generally

int numprocs; // total number of walkers
int myid; // my rank(ID) in the global communicator, MPI\_COMM\_WORLD
int multiple; // number of walkers having the same energy window
int comm_id; // ID for a communicator
int mylocalid[2]; // my ID in local communicators
```
// Get the group of processes in MPI_COMM_WORLD (i.e., all processors)
MPI_Group world;
MPI_Comm_group(MPI_COMM_WORLD, &world);

// The followings are for defining a list of MPI local communicators
int *ranks; // an array to store global IDs
ranks = (int*) malloc(2*multiple*sizeof(int)); // for walkers in a local comm.

int numLocalComm = (numprocs/multiple) - 1; // number of local communicators
MPI_Group *mpi_local_group; // an array to store local groups
MPI_Comm *mpi_local_comm; // an array to store local comm.

for (int i=0; i<numLocalComm; i++) // i: counter for local communicators
{
    // For each walker in local communicator (group) i, calculate the global ID
    // (its rank in MPI_COMM_WORLD) and put them in the 'ranks' array
    for (int j=0; j<2*multiple; j++) // j: counter for walkers in a local comm.
    { ranks[j] = i*multiple+j;

    // Create local group
    MPI_Group_incl(world, 2*multiple, ranks, &mpi_local_group[i]);

    // Create communicator for that group
    MPI_Comm_create(MPI_COMM_WORLD, mpi_local_group[i], &mpi_local_comm[i]);

    free(ranks);

    // Now get my local ID in the local communicators
    if (myid < (numprocs-multiple)) // Every processor except those in the
    { // last window
        comm_id = 2*(myid/(2*multiple));
        MPI_Comm_rank(mpi_local_comm[comm_id], &mylocalid[0]);
        // I am now part of communicator comm_id with local_id[0]=mylocalid[0]
    }
    else
    { mylocalid[0] = INT_MAX; // just give it an invalid value

    if (myid >= multiple) // Every processor except those in the
    { // first window
        comm_id = 2*((myid-multiple)/(2*multiple)) + 1;
        MPI_Comm_rank(mpi_local_comm[comm_id], &mylocalid[1]);
        // I am now part of communicator comm_id with local_id[1]=mylocalid[1]
    }
    else
    { mylocalid[1] = INT_MAX; // just give it an invalid value

}
3.3.2. Exchanging configurations  This is the heart of REWL or any replica-exchange simulation in general. There are three main steps:

(i) Find an exchange partner for each walker from the neighboring energy window. These are randomly assigned by one walker in the energy window who then communicates the exchange partner IDs via the MPI Scatter function.

(ii) Each of the assigned pairs then determine the exchange probability following Equation (1). This itself is a three-step process if we just count the MPI communications involved.
   (a) First, both walkers must communicate their current energies to their partner. The function MPI_Sendrecv_replace can conveniently be used for this purpose: it exchanges the content of the same variable in both partners. Both partner can then individually calculate their part of contribution \( g_i(E_X)/g_i(E_Y) \); \( g_j(E_Y)/g_j(E_X) \) for the other partner to the total exchange probability. This cannot be done by only one partner because the information about the other’s \( g(E) \) is not available.
   (b) One of the partners then sends its result via the basic MPI_Send function to the other, who uses MPI_Receive to receive it, calculates the total exchange probability \( P(X \leftrightarrow Y) \), and decides whether an exchange will take place or not by comparing it to a random number.
   (c) This decision is communicated to the partner via the basic MPI_Send and MPI_Recv functions.

(iii) Finally, the actual configurations are exchanged, if the decision was made to do so. The function MPI_Sendrecv_replace is used again.

The following example code demonstrates how this could be implemented. Note how all data exchanges only use local communicators.

```c
// Arguments passing into the function:
// - swap_direction: indicates exchange with left or right neighbor window
// - index_akt: the own, current histogram index
```

**Figure 3.** A figure to demonstrate how local communicators are set up.
int replica_exchange(int swap_direction, int index_akt)
{
    int i_new;          // histogram index of my configuration
    double myfrac;      // my local exchange probability
    double otherfrac;   // local exchange probability of my swap partner
    double wk;          // the combined exchange probability
    double randx;       // a random number between 0 and 1

    int change = 0;      // boolean: 0 for not exchanging, 1 for exchanging
    int swap_partner = -1; // ID of swap partner (for receive_buffer)

    // +++ Step (i): everyone has to find its swap-partner
    int *pairs;          // array to store exchange
    pairs = (int*) malloc(2*multiple*sizeof(int));  // partner for each process

    if (mylocalid[swap_direction] == 0)  // I am the temporary ‘master’ for
        // this task
    {
        ...
        // Here, the master randomly assigns pairs of walkers for the exchange,
        // and store this information in the array ‘pairs’
    }

    // At this point swap partner assignments get communicated from the
    // master’s ‘pairs’ array to everyone’s ‘swap_partner’ variable
    if ((swap_direction==0) && (myid<(numprocs-multiple)))
        // the walkers from the last window should not swap
        {
            // determine which local communicator to use
            comm_id = 2*(myid/(2*multiple));  // all integer, the ‘/’ is a division
            MPI_Scatter(pairs, 1, MPI_INT, &swap_partner, 1, MPI_INT, 0, 
                         mpi_local_comm[comm_id]);
        }
    
    if ((swap_direction==1) && (myid>=multiple))  // the walkers from the first
        { // window should not swap
            // determine which local communicator to use
            comm_id = ((myid-multiple)/(2*multiple))*2 + 1;
            MPI_Scatter(pairs, 1, MPI_INT, &swap_partner, 1, MPI_INT, 0, 
                        mpi_local_comm[comm_id]);
        }

    free(pairs);

    if (swap_partner != -1)  // if there is a swap-partner for me...
        // (if I am at a global boundary, I might not have
        // a swap partner this time)
    {
        // +++ Step (ii)(a): get histogram index from my swap partner
        // and calculate my contribution to the acceptance probability
    }

```
i_new = index_akt;
MPI_Sendrecv_replace(&i_new, 1, MPI_INT, swap_partner, 1, swap_partner, \
1, mpi_local_comm[comm_id], &status);

if ((i_new>Emaxindex) || (i_new<Eminindex))
    // received index is out of my energy range
    myfrac = -1.0;
else
    // calculate my part of the exchange probability (g(myE)/g(otherE))
    myfrac = exp(lngE[index_akt] - lngE[i_new]);

if (mylocalid[swap_direction] < multiple)
    // I receive the other portion of probability,
    // calculate the total prob., then send the decision to my partner
{
    // +++ Step (ii)(b): get my partner’s part of the exchange probability
    MPI_Recv(&otherfrac, 1, MPI_DOUBLE, swap_partner, 2, \
    mpi_local_comm[comm_id], &status);

    // calculate combined exchange probability and roll the dice
    if ((myfrac>0.0) && (otherfrac>0.0))
    {
        randx = (1.0*rand())/(RAND_MAX+1.0));
        wk = myfrac*otherfrac;
        if (randx < wk) change = 1; // decision to exchange
    }

    // +++ Step (ii)(c): tell my exchange partner the exchange decision
    MPI_Send(&change,1,MPI_INT,swap_partner,3,mpi_local_comm[comm_id]);
}
else // I send my part of exchange probability and await decision
{
    // +++ Step (ii)(b): send my exchange probability part to my partner
    MPI_Send(&myfrac, 1, MPI_DOUBLE, swap_partner, 2, \
    mpi_local_comm[comm_id]);

    // +++ Step (ii)(c): receive exchange decision
    MPI_Recv(&change, 1, MPI_INT, swap_partner, 3, \
    mpi_local_comm[comm_id], &status);
}

// +++ Step (iii): if decision was positive, exchange replicas
if (change == 1)
    MPI_Sendrecv_replace(&latticepoint[0], numberspins+2+1, MPI_INT, \
    swap_partner, 1, swap_partner, 1, mpi_local_comm[comm_id], &status);
}

// Returns whether or not configs were actually exchanged
return(change);
```
3.3.3. Checking flatness The existing function to check the flatness criterion for a single-walker Wang–Landau program can be extended by the following code to check the flatness of all other walkers in the same energy window. At this moment, the integer variable myflat has the value 1 if the histogram for the individual walker was determined to be flat, or 0 otherwise.

```c
int otherflat; // a Boolean to store whether other walkers have a flat histogram or not (0: not flat ; 1: flat)

if (myid%multiple == 0) // this process serves as the 'master' in the energy window and receives individual results from all other walkers
{
    for (int i=1; i<multiple; i++)
    {
        MPI_Recv(&otherflat, 1, MPI_INT, myid+i, 66, MPI_COMM_WORLD, &status);  
        myflat *= otherflat; // if one walker has a '0' (histogram not flat), overall result will be '0'
    }

    for (int i=1; i<multiple; i++) // now let everybody else know
        MPI_Send(&myflat, 1, MPI_INT, myid+i, 88, MPI_COMM_WORLD);
}
else // others send my flatness to 'master' and receive combined result
{
    MPI_Send(&myflat, 1, MPI_INT, myid-(myid%multiple), 66, MPI_COMM_WORLD);
    MPI_Recv(&otherflat, 1, MPI_INT, myid-(myid%multiple), 88, MPI_COMM_WORLD, &status);

    // replace individual flatness result by combined result
    myflat = otherflat;
}
return(myflat);
```

Note a couple of things here. At the end, the variable myflat contains the combined result for all energy windows. That is, the main program for each walker will only call the function to go to the next Wang–Landau iteration if all individual walkers have a flat histogram. In this example we used only the basic MPI_Send and MPI_Recv functions for demonstration purposes. (See how the two separate messages that every process exchanges with the temporary master process have different tags, 66 and 88). If we would also have introduced local communicators for the walkers within every energy window, as shown in Section 3.3.1, the function MPI_Allreduce could have accomplished exactly the same within each local communicator, where the product of the same variable from all processors would be calculated and then broadcast within that local communicator – the value of that variable would be replaced by the product, and every processor would have the same value at the end.

3.3.4. Merging densities of states from different walkers and terminating the simulation If it is determined above that all histograms with the same energy window are flat, all walkers in that energy window would then proceed to the next WL iteration and merge (average) their
density-of-states estimators. Without showing the explicit code snippet here, we recommend using the function `MPI_Allreduce` on the local communicators again. The function collects a data array (containing the density of states) from each process in the communicator, reduces the data by applying a predefined reduction function, and then redistributes the result to all processes. The reduction function used here is `MPI_SUM`. The summed density of states can then be divided locally by the number of processes in the communicator to calculate the average. Finally, to decide whether to terminate the simulation, we invoke another `MPI_Allreduce` on `MPI_COMM_WORLD` using the reduction function `MPI_MAX`:

```c
MPI_Allreduce(&lnf, &lnf_slowest, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
```

to check whether any of the modification factors $f_i$ from the individual energy window is still above the predefined termination value, in which case the simulation simply keeps running.

4. REWL: FAQ

Over the past few years we received many e-mails and questions during workshops asking for practical advice on certain implementation details of REWL. Most of these questions are recurring and of general relevance, so we would like to answer them in the following.

**List of topics:**

4.1) Parallel programming model: OpenMP vs. MPI
4.2) Alternatives for histograms with fixed bin size
4.3) Convergence of REWL with equal-size energy windows
4.4) Why is replica-exchange necessary at all?
4.5) Creation of initial configurations
4.6) What if new low-energy states are found?
4.7) What to do with rejections?
4.8) End of simulation
4.9) Miscellaneous questions

**4.1. Parallel programming model: OpenMP vs. MPI**

**Q:** Between OpenMP [27] and MPI, which one is more efficient / effective to write the parallel replica-exchange Wang–Landau code?

**A:** We recommend using MPI because it will work on machines with both shared and distributed memory. If the code is written using MPI it will be much more portable if you want to run it for larger systems, with more walkers, or on different supercomputers. If you use OpenMP [27] you will be restricted to shared-memory systems, typically a single node on a supercomputer. At that point the efficiency is of secondary importance.

**4.2. Alternatives for histograms with fixed bin size**

**Q:** For various reasons it is difficult to attain a flat histogram for systems with a continuous energy function. Choosing a good bin size is a constant puzzle/inconvenience/annoyance. Are there ways to solve this problem?
A1: Discretization (or binning) of the energy range for a system with a continuous energy function often introduces systematic errors; extra care has to be taken to choose a suitable bin size. One possible solution is to use the so-called kernel or global updates, where multiple histogram bins are simultaneously updated by different amounts according to a kernel function. This allows, in principle, for the energy bins to be arbitrarily narrow. Different kernel functions have been proposed and tested [28]. Gauss kernels are very popular to update binned thermodynamic functions in different WL-type methods in the molecular-dynamics community [29]. It was recently shown that simulations can be significantly sped up when using global updates even when using histograms with bin sizes about ten times smaller compared to simulations with single-bin updates [29].

A2: Another workaround is to avoid the use of a histogram and binning, in general. Recently, a histogram-free version of multicanonical / Wang–Landau sampling has been proposed to address this problem [30]. The energies generated during the simulation are stored directly in a data set instead of in a histogram, which allows for the global update of the estimation of the density of states. Although the sampling weight is still adjusted based on the (inverse of the) estimator of the density of states, this algorithm employs a statistical test as the criterion to proceed to the next iteration instead of a flat histogram. The use of a histogram is thus not necessary.

4.3. Convergence of REWL with equal-size energy windows

Q: Every walker proposes a replica-exchange every $N$ steps of ordinary Wang–Landau random walk. The parameter $N$ is universal among walkers, so the implementation effectively synchronizes every $N$ steps. Essentially, all walkers will perform exactly (or almost exactly) the same number of random walks. Therefore, the underlying area of the histogram will be the same. However, as the distribution $g(E)$ is intrinsically different in every energy window (therefore the areas under the $g(E)$ curve are different), if we ensure the same number of walks in each window, the histogram [and $g(E)$] in some windows will be consistently above other windows. I think this is a problem for the Wang–Landau algorithm to converge. How do you work around it?

A: Undoubtedly, $N$ is universal and a way to intrinsically synchronize all walkers with the assumption that all walkers run equally fast, which is reasonable since they all do the same amount of operations. So it should not slow the whole procedure down significantly. Note that when reducing the modification factor $f$, we only wait until all walkers with the same window fulfill the flatness criterion. Walkers in different windows can be in different iterations of the WL scheme (that is, they have different $f$) but all walkers with the same window always have the same $f$. This is not a problem as the replica-exchange is still valid. But if the walkers in the same window had different $f$, that would be a problem.

While there is no convergence problem in principle, it is true that this is far from optimal. In the optimal case the number and widths of your energy windows should be chosen in a way that all walkers converge at about the same time. That means, each energy window has different numbers of energy bins. See Figure 1b in [1], for example. Of course, in principle, this is impossible since one does not know the density of states and the amount of work needed to estimate it beforehand. We did the run for the mentioned Figure 1b after we already knew the solution to the problem just to demonstrate the effect of optimized energy windows on the efficiency. In practice, one could use the information gathered in pre-runs (see below) to estimate a good bin number for different windows. However, investigations have also shown that the convergence behaviors for different energy windows could change and desynchronize towards the end of the simulation [31]. This is an unsolved question and is a research problem itself.
4.4. Why is replica-exchange necessary at all?

**Q:** If we perform independent WL runs in each window without the replica-exchange step, will the algorithm still produce reasonably good results? I recall that the error mostly came from the boundary of the window, and since we have overlapping windows, it may not be a problem.

**A:** This was actually done in an early attempt to parallelize WL [18]. While this can be beneficial and lead to the correct result for some systems, it prevents the algorithm, in general, from reaching all regions of the phase space, in particular those associated with low temperatures. In fact, there have been ergodicity problems with certain “difficult” systems such as those with first-order transitions. An example is the study of surface adsorption of the hydrophobic-polar lattice model proteins [32]. Some low-energy states, which are known to exist, were never generated because the random walker in the lowest energy window would never “heat up” enough to overcome the entropic barrier around that state. In other words, the walker could be trapped in a local minimum in the energy landscape and would only be able to escape this by exchanging configurations with a higher-energy (“hotter”) window.

4.5. Creation of initial configurations

**Q:** At the beginning of the simulation, how to find the starting configuration / energy for each energy window? Do I just randomly create configurations until, eventually, I created at least one for every energy window? Can I just copy over configurations from one window into another if they are in the overlap regions between two windows?

**A:** As you say, one way is for each energy window to start from a randomly chosen configuration, which probably has an energy that is outside of the energy window. Then you could apply Monte Carlo moves to generate new configurations, until an energy that falls within the energy window is discovered. However, this can take an arbitrarily long time for large systems and is therefore probably not the best way of doing it.

For systems with known ground states, for example the Ising model, one can start with that ground state configuration and perform a Metropolis simulation at high temperatures (or even infinite temperature, for that matter, just accept every trial move). Whenever a configuration with an energy $E_{i,c} \pm \epsilon$ is created (where $E_{i,c}$ is the energy at the center of energy window $i$ and $\epsilon$ a reasonable energy range), it can be taken as the initial configuration for energy window $i$. This method is very fast and was used to create initial configurations for the spin-model simulations shown in [1, 2]. It took only a few minutes to create initial configurations even for the largest system sizes that have been studied.

Another way is to perform a WL pre-run on the global energy range and store the configurations that can be used as the starting configurations for each energy windows in a similar way. This way might be slower, but could be the only way if low-energy states are unknown and a pre-run is necessary for determining the relevant energy range anyway (see question 4.6). In a similar manner, initial states can be created “along the way” if a pre-run is performed to determine suitable energy window ranges for runtime balancing, see question 4.3.

4.6. What if new low-energy states are found?

**Q:** For a model of which the ground state energy and the energy landscape are not known a priori, what is the best way to start: performing parallel-tempering pre-runs to estimate the global energy range, or starting right away with REWL and adapting energy boundaries on the fly?
The first would cost much more computing time, the latter might be problematic with regard to convergence.

A: When simulating a new model, most of the time we are in that situation where the ground state information is missing. Both suggested ways of dealing with this problem are possible and are used in practice. Typically, we prefer performing pre-runs to estimate the ground state energy. One advantage is that during these pre-runs we can also generate initial configurations for different energy windows (see also question 4.5). These pre-runs have been very fast for most of our studies, in particular because it is often sufficient to know a few very-low energy states, rather than the lowest. For example, if we are interested in studying a freezing transition, it might be sufficient to know a couple of states from the crystalline phase to determine the energy ranges. An alternative to parallel-tempering pre-runs could be conventional WL runs with an excessively large energy range for which the histogram never becomes flat.

But sometimes, the sampling of the actual ground states is in fact the interest. In such cases it is inevitable to adapt energy ranges and energy windows every time a new lowest energy is found. It is important, however, to restart the whole WL scheme after making this change. That is, all histograms need to be deleted and reset to the initial value. There have been attempts to avoid a complete restart [33]. While this is an effective approach to aid the discovery of the ground states, it is harder to control the behavior of the simulation and could potentially result in artificial signals in thermodynamic functions.

4.7. What to do with rejections?

Q: While performing the ordinary WL random walk there could be configurations generated that fall outside of the energy window. Do you still increase the histogram entry for the configuration after such a trial move?

A: Yes, that should always be done in any WL simulation. You certainly reject the newly proposed state which is outside the energy range, but very importantly, count the original energy again. That is, update the corresponding density of states and histogram entries. See [34].

Q: When proposing a replica-exchange, it could be that $E_X$ is not in its target energy window or $E_Y$ is not in the home energy window. While the replica-exchange is rejected, do you update $H(E)$?

A: Yes, and also $g(E)$, for the original energy. This situation is, in fact, the same as the one described directly above. Alternatively, this could be interpreted as a Monte Carlo move with zero acceptance probability (see question directly below).

Q: If the replica-exchange is rejected by chance according to the acceptance probability, do you update $H(E)$?

A: Yes, and also $g(E)$, for the original energy. This is always done in any Monte Carlo scheme.

4.8. End of simulation

Q: If the walkers in every energy window can proceed to the next WL iteration on their own pace, some walkers will reach their final modification factor and terminate the WL sampling before others. How to assure that a replica-exchange is only proposed with “living” walkers, that have not terminated the simulation yet?
A: We suggest doing it differently: keep all walkers alive until all others reach the termination criterion as well. This is for exactly the reason of assuring that there will always be a replica-exchange partner for every walker. In our implementation, walkers who have finished their WL iterations just keep walking, without updating \( f \) anymore, to be able to provide exchange configurations. All walkers terminate simultaneously when all have reached the termination criterion individually. See section 3.3.4 for an implementation example. It would be optimal if all walkers would reach the termination criterion naturally at the same time. For this to happen one would have to create energy windows of different sizes. We have experimented with this approach and shown an example in [2], Figure 1c. The total runtime for each window is, again, estimated based on measurements from pre-runs.

4.9. Miscellaneous questions

Q: When there are multiple walkers with the same energy window, a walker may receive more than one proposal for replica-exchange from its neighboring walkers. Do you only accept the first and ignore all others? How do you handle this situation?

A: For the replica-exchange, your scheme seems valid but might not be optimal. We randomly draw pairs of walkers from neighboring windows. So, by construction, no walker can receive more than one proposal at a time. This also ensures that every walker gets a proposal in the first place, assuming there are equally many walkers in each of the neighboring energy windows.

Q: How do you choose \( N \), the number of ordinary Monte Carlo sweeps before a replica-exchange step is proposed? Any typical choice? How does it affect the results?

A: Unfortunately, that mainly depends on the physical system at hand. Optimally, you want to give the walkers enough time to “diffuse” through the energy window between replica-exchanges. That is, give them enough time to be able to “randomly walk to the other side” of the energy window. Ideally, measurements are made in pre-runs and \( N \) is adapted accordingly before the final run. A bad choice of \( N \) would typically affect the performance much more than the actual results in the sense that it can take much longer to converge to the correct results.

Q: What data are actually exchanged during the replica-exchange step?

A: The walkers from neighboring energy windows only exchange their current configurations (including the corresponding energies). Any other data, in particular \( g(E) \) and \( H(E) \), remain with the walkers. From the point of view of each of the two walkers individually, this can be seen as “randomly” proposing a new configuration for the next Monte Carlo step.

Q: It is mentioned that “\( g(E) \) from all \( m \) walkers in the same energy window are averaged and redistributed”. What do you mean by “redistributed”? Do you try to “stitch” the \( g(E) \) among energy windows? (bring them into the same height?)

A: At the end of an iteration, the \( g(E) \)’s from all walkers in that window are averaged, and then every walker in that window proceeds with that average. That is, after the averaging, all walkers in the energy window have the same density of states to start the new iteration. There is no need to join the density-of-states estimators at this point.
Acknowledgments
We thank N. Casper, Y. Ren, Y. Shevchenko, T. K. Bose, and D. Moreira for lively discussions and acknowledge their contributions with regard to contents of the FAQ section of this paper. Y.W.L. was sponsored by the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References
[1] Vogel T, Li Y W, Wüst T and Landau D P 2013 Phys. Rev. Lett. 110 210603
[2] Vogel T, Li Y W, Wüst T and Landau D P 2014 Phys. Rev. E 90 023302
[3] Vogel T, Li Y W, Wüst T and Landau D P 2014 J. Phys.: Conf. Ser. 487 012001
[4] Gai L, Vogel T, Maerzke K A, Iacovella C R, Landau D P, Cummings P T and McCabe C 2013 J. Chem. Phys. 139 054505
[5] Li Y W, Vogel T, Wüst T and Landau D P 2014 J. Phys.: Conf. Ser. 510 012012
[6] Perera D, Vogel T and Landau D P 2016 Phys. Rev. E 94 043308
[7] Valentim A, Rocha J C S, Tsai S H, Li Y W, Eisenbach M, Fiore C E and Landau D P 2015 J. Phys.: Conf. Ser. 640 012006
[8] Shevchenko Y A, Makarov A G, Andryushchenko P D and Nefedev K V 2017 J. Exp. Theor. Phys. 124 982
[9] Ren Y 2016 private communication. Work lead to the development of another parallelization scheme for Wang–Landay simulations [10]
[10] Ren Y, Eubank S and Nath M 2016 Phys. Rev. E 94 042125
[11] Casper N 2016 Stochastic analytic continuation of Quantum Monte Carlo data Master’s thesis Technical University Braunschweig, Germany
[12] Yuk S F, Li Y W, Cooper V R and Eisenbach M 2017 private communication
[13] Wang F and Landau D P 2001 Phys. Rev. Lett. 86 2050
[14] Wang F and Landau D P 2001 Phys. Rev. E 64 056101
[15] Geyer C J 1991 Markov chain Monte Carlo maximum likelihood Computing Science and Statistics, Proceedings of the 23rd Symposium on the Interface (Interface Foundation of North America) p 156 retrieved from the University of Minnesota Digital Conservancy URL http://hdl.handle.net/11299/58440
[16] Hukushima K and Nemoto K 1996 J. Phys. Soc. Japan 65 1604
[17] Demonstration codes complementary to the Replica-Exchange Wang–Landau sampling tutorial lectures given at the IX Brazilian Meeting on Simulational Physics, August 21–25, 2017, Natal, Brazil. https://github.com/yingwaili/bmsp2017
[18] Cunha-Netto A G, Caparica A A, Tsai S H, Dickman R and Landau D P 2008 Phys. Rev. E 78 055701
[19] MPICH – High-Performance Portable MPI. http://www.mpich.org/
[20] MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE. http://mvapich.cse.ohio-state.edu/
[21] Open MPI: Open Source High Performance Computing. https://www.open-mpi.org/
[22] Lawrence Livermore National Laboratory’s MPI tutorial. https://computing.llnl.gov/tutorials/mpi/
[23] Message Passing Interface (MPI) Forum - contains the official documents for the MPI Standards. http://www.mpi-forum.org/
[24] Pacheco P S 1996 Parallel Programming with MPI (San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.) ISBN 1-55860-339-5
[25] Bland W, Bouteiller A, Herault T, Bosilca G and Dongarra J 2013 Int. J. High Perform. Comput. Appl. 27 244
[26] Bland W, Du P, Bouteiller A, Herault T, Bosilca G and Dongarra J 2013 Concurr. Comput. Pract. Exper. 25 2381
[27] OpenMP - The OpenMP API specification for parallel programming. http://www.openmp.org/
[28] Zhou C, Schulthess T C, Torbrügге S and Landau D P 2006 Phys. Rev. Lett. 96 120201
[29] Junghans C, Perez D and Vogel T 2014 J. Chem. Theory Comput. 10 1843
[30] Li Y W and Eisenbach M 2017 A Histogram-Free Multicanonical Monte Carlo Algorithm for the Basis Expansion of Density of States Proceedings of the Platform for Advanced Scientific Computing Conference PASC ‘17 (New York, NY, USA: ACM) p 10:1
[31] Zhao Y, Cheung S W, Li Y W and Eisenbach M 2014 Physics Procedia 57 43
[32] Li Y W, Wüst T and Landau D P 2013 Phys. Rev. E 87 012706
[33] Wüst T and Landau D P 2008 Computer Physics Communications 179 124
[34] Schulz B J, Binder K, Müller M and Landau D P 2003 Phys. Rev. E 67 067102