# Benchmarking Strategies for Parallel 2D Ising Model Simulation
**Advanced Methods for Scientific Computing - A.Y. 2023/2024**

Davide Patricelli
*Engineering Physics*
*Politecnico di Milano - Milan, Italy*
*Email: Davide.patricelli@mail.polimil.it*
*Student ID: 10865590*

## 1. Introduction and project goal

The simulation of the two-dimensional (2D) Ising Model, a fundamental model in statistical mechanics, presents some computational challenges, particularly near critical temperatures where critical slowing down occurs and when dealing to paralellization due to the need of spins interaction. This project focuses on benchmarking parallelized algorithms for the 2D Ising Model simulation, aiming to enhance computational efficiency and accuracy in different high-performance computing environments. The project is a group work where team members explore individual solutions within the context of cooperation and task division. The general objective is to investigate various parallelization strategies for Ising simuation, an area that has seen limited exploration in existing literature.

In this report, I will show and analyze the main aspects with which I dealt throughout the course of the project, in Particular:

i. **Metropolis-Hastings Algorithm:** Implementation of the serial algorithm as well as the code structure and functionality that are common to most of the serial and parallel implementation.
ii. **OpenMP parallelization:** The core of my work is to propose and bechmarking three different implementation of multi core parallel programming. With the aim of studying their performance in particular with reference of lattice dimension. The focal point remains the Metropolis-Hastings Algorithm, where significant performance improvements are achievable through the parallelization of the huge number of required random number extractions.
iii. **Equilibrium detection** Implementing methods for equilibrium detection is crucial for enabling automatic simulation termination. This becomes particularly challenging in parallel implementations, where issues of memory consistency and performance may arise. Addressing these complexities is essential to ensure efficient and reliable simulations.
iv. **Simulation organization:** from the beginning of the work the project had the interest of proferossor Ezio Puppin and his fellowmen at physic department. He also give us the oportunity of running our program on multi core and multi node cluster (INFN RECAS and POLITECNICO DI MILANO CINECA). For that reason a part of my work was also the one of organizing simulation and results storing in a thoughtfull way.

The goal is to evaluate the performance, scalability, and efficiency of these algorithms in simulating the 2D Ising Model.

## 2. Physics Backgound

### 2.1. Ferromagnetic Phase Transition

Magnetic materials exhibit fascinating behavior as they transition between different phases, governed by the temperature-induced changes in their magnetic properties. One such phenomenon is the ferromagnetic phase transition, a critical process that has both macroscopic and microscopic implications.

### Macroscopic Behavior

At elevated temperatures, the magnetic material exhibits a disordered phase, behaving as a paramagnetic or nonmagnetic substance. However, as the temperature decreases, a critical point known as the Curie temperature is reached.

Below the Curie temperature, the material undergoes a phase transition, manifesting a macroscopic magnetization,as T decrease maximum magnetization condition is reached, referred to as saturation magnetization.

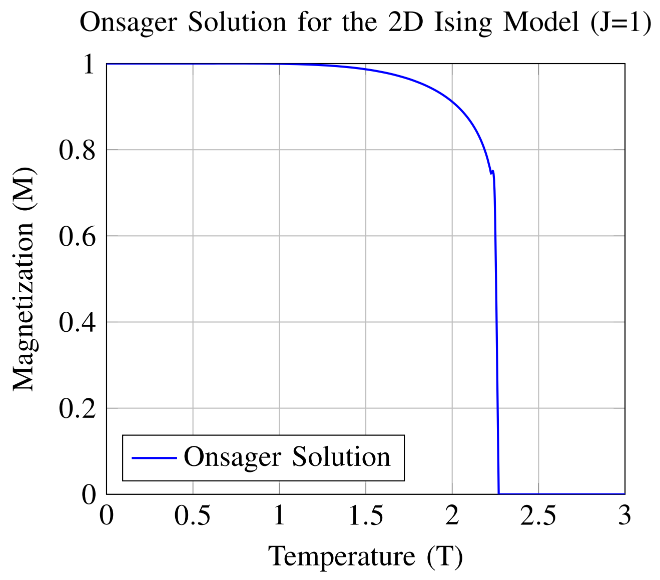

Figure 1. Theoretical magnetization curve of the 2D Ising Model as predicted by the Onsager solution, representing phase transition.

## Microscopic Symmetry Breaking

At a microscopic level magnetism can be related to the intrinsic magnetic moments linked to atoms spins in the cystal structure

At high temperatures, thermal energy dominates, allowing for random orientations of magnetic moments. The system exhibits high-temperature symmetry, with no preferred direction of magnetization. However, as the temperature drops below the critical temperature, a spontaneous breaking of rotational symmetry occurs.

This symmetry breaking leads to the emergence of a preferred direction for magnetization. The material undergoes a transition from a state with no preferred orientation to a state where there is a consistent alignment of magnetic moments. Additionally, the material may exhibit the formation of magnetic domains, where each domain has a distinct and consistent orientation of magnetization.

In summary, the ferromagnetic phase transition involves a transition from a disordered, high-temperature state to an ordered, low-temperature state as in figure 2.1 . This transition is marked by the spontaneous alignment of magnetic moments, the breaking of rotational symmetry, and the emergence of macroscopic magnetization in the material.

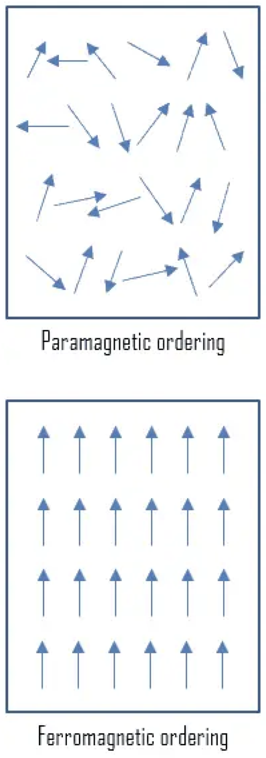

Figure 2. Microscopic simmetry breaking related to paramagnetic ferromagnetic transition that occours at Curie Temperature

## 2.2. The ising model

The Ising model, a classical model in statistical mechanics, provides valuable insights into the behavior of magnetic materials during the ferromagnetic phase transition. Introduced by Ernst Ising in 1925, this model simplifies the complex interactions between magnetic moments in a material.

In the Ising model, magnetic moments are represented as discrete spins, which can take two values: up or down. The interactions between neighboring spins are typically modeled with a simple coupling term. This seemingly straightforward model captures essential features of ferromagnetic materials and serves as a foundation for understanding the emergence of macroscopic magnetization during the phase transition.

In particular in the 2D Ising model, we consider a square lattice where each site has a spin that can take a value of either +1 or -1. The key assumption is that each spin interacts only with its nearest four neighbors (left, right, above, and below). The Hamiltonian of the system is given by:

$$H = -J \sum_{\langle i,j \rangle} s_i s_j \tag{1}$$

where $s_i$ and $s_j$ are the spin values at sites $i$ and $j$, $J$ is the interaction strength, and the sum is over nearest neighbor pairs.

In the Ising model, the change in energy ($\Delta E$) and magnetization ($\Delta M$) resulting from the flipping of a single spin $s_i$ can be evaluated as follows:

$$\Delta E = -2 \cdot s_i \sum_{\text{neighbors}} s_j \tag{2}$$

This equation captures the interaction of the flipped spin with its neighboring spins, and the factor of 2 accounts for the fact that each neighboring spin is counted twice.

The change in magnetization is given by:

$$\Delta M = 2 \cdot s_i \tag{3}$$

Here, the factor of 2 accounts for the fact that the magnetization changes by twice the value of the flipped spin.

These expressions are fundamental to understanding the dynamics of the Ising model and then for the following development.

### 2.3. Periodic Boundary Conditions

To mimic an infinite lattice and reduce edge effects, periodic boundary conditions are applied. This means that the grid is conceptualized as a torus where the top and bottom edges are connected, as are the left and right edges. As a result, each spin has four neighbors, even at the edges of the lattice.

## 3. Expected Correct Result for $J = 1$ at Temperatures Between 0 and 3 Kelvin

For the 2D Ising Model the expected behavior at temperatures between 0 and 3 Kelvin can be summarized as follows:

- **At Low Temperatures (0 - Tc)**: In this range, the system is expected to exhibit spontaneous magnetization. At absolute zero ($T = 0$), all spins should align, resulting in maximal magnetization. As temperature increases, thermal fluctuations induce random spin flips, but the overall magnetization remains significant due to the system's tendency towards ordered states.
- **Near the Critical Temperature (Tc)**: At the critical temperature, $Tc \approx 2.269$, the system undergoes a phase transition from an ordered to a disordered state. Here, the magnetization abruptly decreases, marking the loss of spontaneous magnetization. This point is critical for observing the behavior of the Ising Model in simulations.
- **At High Temperatures (Tc - 3)**: Beyond the critical temperature, thermal agitation dominates, leading to a disordered state where spins are randomly oriented. In this regime, the magnetization should approach zero, indicating the lack of long-range order in the system.

This temperature-dependent behavior of magnetization is crucial for verifying the correctness of the implemented simulation algorithms. The algorithms implemented reproduce these expected results, providing confidence in their accuracy and effectiveness for simulating the 2D Ising Model.

## 4. Metropolis-Hastings Algorithm

### 4.1. Introduction

In the context of Monte Carlo simulations, the main task is to generate states in accordance with the Boltzmann probability distribution. The conventional approach of random state selection, coupled with acceptance or rejection based on the Boltzmann factor, proves inefficient due to exponentially small acceptance probabilities. Instead, Monte Carlo methodologies found on Markov processes as the main tool for state set generation.

A Markov process, denoted as $\{u, v\}$, dynamically generates a new system state $v$ given an initial state $u$ in a stochastic manner. The transition probability, $P(u \to v)$, signifies the likelihood of transitioning from state $u$ to $v$. For a valid Markov process, these transition probabilities must remain invariant over time and depend solely on the current states $u$ and $v$.

In the Monte Carlo simulation paradigm, the Markov process is iteratively employed to construct a Markov chain of states. The selection of the Markov process is pivotal, ensuring that the resultant Markov chain converges to the desired Boltzmann distribution. The conditions of ”ergodicity” and ”detailed balance” are imposed on the Markov process to guarantee its effectiveness and validity.

The condition of *ergodicity* ensures that the Markov process can traverse from any state to any other state given a sufficiently extended runtime. This is imperative for generating states with accurate Boltzmann probabilities. On the other hand, the condition of *detailed balance* guarantees that the equilibrium distribution obtained aligns with the Boltzmann distribution, ensuring equal transition rates into and out of any state when the system reaches equilibrium.

In essence, the utilization of Markov processes in Monte Carlo simulations involves crafting a sequence of states converging to the Boltzmann distribution. The fulfillment of the conditions of ergodicity and detailed balance is fundamental for the integrity and efficiency of the simulation. Whitouth entering in the details of the theory, just present Metropolis algorithm, a powerful tool in Monte Carlo simulations that operates in accordance with the theoretical principles discussed here,

## 4.2. The Algorithm

The Metropolis-Hastings algorithm for the 2D Ising model can be summarized in the following steps:

1) Initialize the lattice with spins either randomly or in an ordered state.
2) Randomly select a spin from the lattice (ensuring ergodicity).
3) Calculate the energy change ($\Delta E$) that would result if this spin were flipped. This can be computed based on the interaction with its nearest neighbors expressed in 2.
4) Decide whether to flip the spin in accordance with detailed balance one has:
   - If $\Delta E \leq 0$, flip the spin
   - If $\Delta E > 0$, flip the spin with a probability of $e^{-\Delta E/kT}$, where $k$ is the Boltzmann constant and $T$ represents the temperature.

   .
5) Evaluate change in magnetization with 3.
6) Repeat steps 2 to 4 for a large number of iterations to ensure the system reaches equilibrium.

These steps allow the system to explore different configurations and approach the Boltzmann distribution at a given temperature, facilitating the study of various thermodynamic properties in the Ising model.

## 4.3. Parallelization Strategy

The primary aim was to parallelize the flipping procedure effectively. To optimize parallelism, we divided the square lattice into square blocks, each corresponding to a thread. This approach allows individual threads to handle serial flips within their designated block, enabling simultaneous execution.By doing so, parallelization for both random extraction and flip instructions is exploited, enhancing overall efficiency and performance.The challenge lies in managing the spin at the boundaries of the blocks, where the proximity of a nearest neighbor from another block introduces complexities. This issue necessitates careful consideration, giving rise to various parallel strategies to effectively address the interactions between neighboring blocks. These strategies will be presented in the next paragraph. While aspects related to consistently managing lattice state and magnetization value will be presented in the implementation chapter.

## 4.4. Sliding Window

In this parallel Ising algorithm, spin updates selectively target spins away from block boundaries. To tackle the challenge posed by boundary spins, a periodic translation of the lattice matrix is incorporated. This translation ensures thorough visits and updates of spins at the boundaries, bolstering the algorithm’s parallel efficiency. Notably, the period of translation becomes a critical parameter. A low period impacts performance, introducing synchronization and vector operation overhead. Conversely, a high period affects the system’s evolution towards equilibrium, as certain spins remain unreachable for extended periods, influencing the equilibrium-reaching process. Balancing this parameter is crucial for achieving optimal algorithm performance and equilibrium behavior.

## 4.5. Domain Decomposition

In this approach the utilization of atomic updates specifically for boundary sites is adopted while all thread run cuncurrently. This strategic choice ensures thread safety and consistency throughout parallelized simulations.

The critical factor in this implementation is the size of the block that could influences performance. The reason lies in the inverse relationship between the percentage of boundary spins over total and the total number of spins within a block. When the block size increases, the percentage of boundary spins decreases, leading to a proportional reduction in the number of atomic operations relative to the total. This optimization can be achieved by maintaining a fixed number of threads while increasing the lattice size, with the possibility of enhancing overall performance during parallelized simulations.

### 4.6. CheckBoard

The lattice is effectively divided into two alternating sublattices, creating a checkerboard-like pattern. The simulation advances by updating spins on each sublattice in alternating steps, contributing to the overall dynamics of the Ising model.

A crucial consideration in optimizing performance is the selection of the number of operations between each switch of the sublattices. Striking the right balance is essential to mitigate potential slowdowns arising from synchronization and associated overhead. Too low a number of operations may lead to frequent sublattice switches, incurring additional computational cost, while too high a number may impact the algorithm's ability to reach equilibrium. This delicate balance is critical in achieving an optimal trade-off between computational efficiency and the equilibrium state of the system.

## 5. Implementation

### 5.1. general organization

In the organization of the Ising model code, a modular structure is adopted, primarily consisting of two classes type. The first class ragards the lattice, it serves the purpose of managing the lattice and its properties such as state , energy and magnetization. The second type of classes are dedicated to implementing the Metropolis algorithm for Monte Carlo simulations. Its functionalities include the application of the Metropolis algorithm to update lattice configurations, managing acceptance/rejection criteria, and executing Monte Carlo steps to evolve the system. This modular approach is designed to facilitate flexibility, allowing for the introduction of additional classes, especially for diverse Monte Carlo simulations, and easing the process of benchmarking. Within the remainder of this chapter, we explore the characteristics of diverse classes. In both scenarios, the utilization of abstract classes becomes pivotal to emphasize the common functionalities crucial for different algorithms, which often vary in their implementation details. The definition of *virtual methods* plays a significant role in facilitating this process. Moreover the most important structure in terms of memory are menaged using *smart pointer* in order to have an efficient memory managemnt during simulations.

### 5.2. Class `AbstractLattice`

This class serves as an abstract base for lattice structure simulations. It outlines the fundamental methods required for any lattice model, ensuring a standardized approach across different implementations.

**Constructor**

The constructor is implicitly defined, as this is an abstract class. Derived classes must provide their own constructors to initialize specific lattice configurations.

**Public Methods**

- `initialize`: A pure virtual function that must be overridden to set up the lattice according to specific rules or data.
- `evaluate_energy const`: Computes and returns the current energy state of the lattice. It provides a measure of the system's stability.
- `print_lattice const`: Offers a visualization or textual representation of the lattice's current state, aiding in debugging and analysis.
- `get_interaction_energy const`: returns the energy $J$ related to interactions between lattice spins. Essential for understanding lattice dynamics.

**Destructor**

- `AbstractLattice`: A virtual destructor ensuring proper cleanup of derived class objects. It prevents memory leaks and other resource mismanagement issues in polymorphic use cases.

**Usage and Extension**

This class is designed to be extended by specific lattice models. By inheriting from `AbstractLattice`, derived classes must implement the pure virtual methods, thus adhering to the prescribed interface. This approach promotes consistency and reusability in lattice simulations.

**Design Rationale**

The use of pure virtual functions enforces a contract for derived classes, ensuring that they provide specific implementations for initialization, energy evaluation, and visualization. This design choice supports polymorphism and extensibility, key principles in object-oriented programming, especially relevant in simulation frameworks where different models might need to be tested and compared.

## 5.3. Class `SquareLattice`

The `SquareLattice` class extends the '`AbstractLattice` and implements a square lattice structure, namely a bravais lattice in which each spin has four nearest neighbours, primarily used in simulations. This class provides specific functionalities for initializing, visualizing, and calculating the properties of a square lattice.

**Constructor**

`SquareLattice`: Constructs a `SquareLattice` object with specified interaction strength and lattice size. The constructor initializes the lattice and random lattice vectors calling `initialize` method.

### Class Attributes

- `J` interaction strenght between spins
- `L` Number of spins per side
- `N` Total number of spins
- `M` Total magnetizazion of the lattice
- `E` Total energy of the lattice
- `Mrand` Magnetization of the random initialized lattice
- `E_rand` Energy of the random initialized lattice
- `lattice` lattice vector organized in a linear structure
- `randomLattice` initial lattice vector organized in a linear structure

**Public Methods**

- `print_lattice const`: Visualizes the current state of the lattice on standard output. Spins are represented as 'o' for -1 and 'x' for 1, aiding in a quick and intuitive understanding of the lattice state.
- `evaluate_energy const`: Computes the total energy of the lattice based on its current state and the interaction strength. This function is crucial for understanding the stability and dynamics of the lattice.
- `initialize`: Initializes the lattice with random spin states and calculates its initial total energy and magnetization. This method sets the starting point for simulations.
- `get_lattice const`: Provides access to the lattice vector. This method allows external functions to interact with the lattice's current state.
- `get_interaction_strength const`: Gets the interaction strength for the lattice spins.
- `get_magnetization const`: Gets the magnetization of the lattice.
- `get_energy const`: Gets the energy of the lattice.
- `increment_magnetization` Increments the magnetization of the lattice by the value passed as argument.
- `increment_energy` Increments the energy of the lattice by the value passed as argument.
- `restore_random_lattice` Restores the lattice to the randomly initialized state.

**Implementation Details**

The `SquareLattice` class represents a specific type of lattice model. It uses a vector to store the lattice state, with each element representing a spin. The energy calculation takes into account the interaction between neighboring spins, following the principles of statistical mechanics.

The constructor and the `initialize` method play a crucial role in setting up the lattice for simulations. They ensure that the lattice starts from a well-defined state, either random or specified, for accurate and reproducible simulation results.

**Design Rationale**

This class follows the principles of object-oriented design, extending the `AbstractLattice` to provide specific functionality for square lattices. By encapsulating the lattice-related operations within this class, the design promotes modularity and reusability, allowing the `SquareLattice` to be easily integrated into larger simulation frameworks. The choice of using linear vectors for lattice representation is driven by the need for efficient access and manipulation of lattice states, which is critical for performance in simulations. Moreover the choiche of not using array is related to the necessity of having in the same code the possibility of changing the lattice size, reaching high value of nu,ber of spins.

**Usage in Simulations**

The `SquareLattice` class is used to create and manipulate square lattice models in simulations. Its methods provide essential capabilities like initializing the lattice, calculating its energy, and visualizing its state. These functionalities are crucial for studying phenomena like phase transitions, magnetization properties, and other physical characteristics in statistical mechanics simulations.

## 5.4. Class AbstractMonteCarloSimulation

The `AbstractMonteCarloSimulation` class serves as a foundation for implementing various Monte Carlo simulation models. It defines a standardized interface that must be adhered to by derived classes. This design ensures consistency and ease of comparison between different Monte Carlo simulations.

### Constructor

The class has no explicit constructor, being an abstract class. Derived classes are required to provide their own constructors for initializing specific simulation configurations.

### Public Methods

- `simulate_phase_transition`: A pure virtual function that must be overridden to perform the simulation of a phase transition over a range of temperatures.

### Protected Methods

- `simulate_step`: A pure virtual function simulating a step in the Monte Carlo simulation, where a step corresponds to a submultiple of Monte Carlo steps. It involves updating the lattice, magnetization, and energy based on the provided parameters.
- `store_results_to_file`: A pure virtual function for storing the results of the simulation, including energy and magnetization, to a file.
- `create_rand_vector`: A pure virtual function for creating a random vector, indicating the site to flip during the simulation, with the idea of reuising it in different temperature simulation. The size and content of the vector can vary based on the specific simulation implementation. **NOTE:** This method is deprecated due to high memory usage in larger lattice.

### Destructor

- `~AbstractMonteCarloSimulation`: A virtual destructor ensuring proper cleanup of derived class objects. This helps prevent memory leaks and resource mismanagement in polymorphic use cases.

### Usage and Extension

Derived classes must inherit from `AbstractMonteCarloSimulation` and provide concrete implementations for the pure virtual methods. This adherence to the prescribed interface promotes consistency and reusability in Monte Carlo simulations.

### Design Rationale

The use of pure virtual functions enforces a contract for derived classes, ensuring they provide specific implementations for simulation phases, random vector creation, simulation steps, and result storage. This design choice supports polymorphism and extensibility, crucial principles in object-oriented programming. It facilitates the comparison and testing of different Monte Carlo simulation models within a standardized framework.

### Derived class

In this case from the abstract class several classes are obtained 3, they inherit its methods structure but provides different implementations and additionl method.

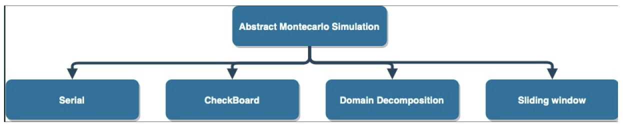

Figure 3. Class Hierarchy of the Metropolis Algorithm implementations

## 5.5. Other Class Members

Furthermore, there exist additional shared methods and data members across the algorithms. While these components serve a common purpose, their distinct implementations and interfaces preclude them from being defined as virtual methods or having a uniform description of data members. However, for the sake of an efficient presentation, it is beneficial to initially introduce their overarching goals before delving into the specific details of implementation for each algorithm. In particular each class will have a :

- `flip` The method implementing the flipping logic in a Monte Carlo simulation while respecting detailed balance is typically responsible for determining whether to accept or reject a random change in the system configuration.

While the common data members are:

- `lattice`: Private instance of Square lattice, that is used in the simulation.
- `RandVect`: `std::unique_ptr<std::vector<int>>` – Random vector of size proportional to the number of iteration.Depending on implementation it could have different sizes, in particular in parallel codes each thread generates then its own private random vector. **NOTE:** as its relative method `create_rand_vector` its use is deprecated.
- `EnergyResults`: `std::unique_ptr<std::vector<float>>` – Vector to store energy results.
- `MagnetizationResults`: `std::unique_ptr<std::vector<float>>` – Vector to store magnetization results.
- `Temperatures`: `std::unique_ptr<std::vector<float>>` – Vector to store temperatures visited.
- `T_MIN`: Minimum temperature for the simulation.
- `T_MAX`: Maximum temperature for the simulation.
- `T_STEP`: Temperature step size for the simulation.
- `L`: Size of the lattice.
- `N`: Total number of lattice sites.
- `IT`: Total number of iterations/flip in the simulation.
- `rng`: Mersenne Twister 19937 generator.It is a popular pseudorandom number generator known for its long period and strong statistical properties. Its efficient parallelization makes it well-suited for scenarios requiring multiple independent streams of random numbers concurrently, enhancing its value in parallel computing for simulations and algorithm. Indeed when coming to parallel execution each thread will have its private and different seed generator.
- `dist`: Uniform distribution in [0, 1).

## 5.6. Class SerialMetropolis

### Class `SerialMetropolis`

This class implements a serial version of the Metropolis algorithm for simulating a phase transition in a lattice system. It inherits from the `AbstractMonteCarloSimulation` class and provides a concrete implementation of the required methods for a serial simulation.To avoid redundancy, the focus will be exclusively on the implementation of the most crucial methods.

### Constructor

The constructor initializes the simulation parameters and allocates memory for vectors to store results. It takes the interaction strength, lattice size, temperature range, temperature step, and the total number of iterations as parameters.

### Public Methods

**`simulate_phase_transition`**
This method performs the simulation of a phase transition over a range of temperatures. It iterates through temperatures, performing for each one a

Monte Carlo simulation to equilibrium of the lattice calling simulate_step method,and store the results and reinizialize the lattice to the random one 4.

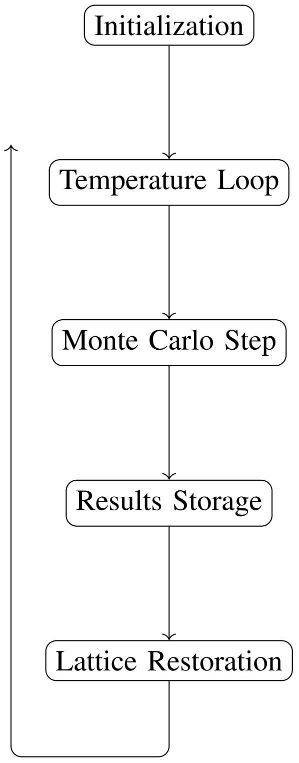

Figure 4. Flowchart for `simulate_phase_transition` method

**`simulate_step`**

In this context the method simulates a Monte Carlo simulation by flipping spins based on the Metropolis algorithm ,using the relative method, for a given number of iterations.

### Design Rationale

The class is designed to provide a serial implementation of the Metropolis algorithm, adhering to the abstract interface specified in the `AbstractMonteCarloSimulation` class. The use of the Metropolis algorithm ensures detailed balance and the generation of statistically valid results.

### Usage and Extension

Derived from the abstract class, `SerialMetropolis` provides a concrete implementation suitable for serial simulations and it is the refernce serial benchmark for speed up. It should be extended for parallel implementations, promoting consistency and reusability in lattice simulations.

### Derived from Abstract Class

`SerialMetropolis` inherits from the `AbstractMonteCarloSimulation` class, ensuring that it provides specific implementations for initialization, energy evaluation, and visualization. This design choice supports polymorphism and extensibility, key principles in object-oriented programming.

## 5.7. Parallel Members

Also considering parallel algorithms, common features emerge. The primary concept involves dividing the lattice into blocks corresponding to the number of threads 6. This choice imposes two requirements on the number of threads:

- It must be a perfect square.
- It must fit the lattice size; in other words, the total number of spins must be a multiple of the number of threads.

The first requirement can be relaxed for flexibility, however doing so introduces some serialization in the execution of block operations. In project's classes both costraint are fulfilled, and all one have a method:

- `set_block_size` that given a certain number of threads verify that the two codition are met, and define the spins per side of a blocks.

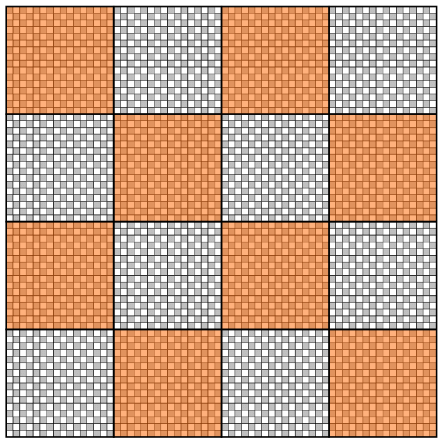

Figure 5. Division of the square lattice in blocks, here color identify the checkboard pattern. Notably in this example, as in general, the number of blocks is 16, a perfect squere

Furthermore parallel class will have some common data members:

- `NUMTHREAD`: Number of threads used in parallel execution.
- `A`: Number of spins per block side.
- `ThreadStart`: `std::unique_ptr<std::vector<int>>` A vector of length `NUMTHREAD`, denoting the starting indices for each block, usefull in the context of a linear lattice implementation. This approach facilitates the extraction of random spins within the initial block and subsequently translates the index using an offset obtained from the vector element, accessible through the block index

In exploring parallelization strategies within *OpenMP*, various possibilities were considered. However, the chosen approach revolves around the utilization of the task construct. The decision to employ `openMP task` constructs stems from their ability to facilitate more dynamic parallelism. Additionally, task constructs prove to be highly beneficial when implementing auto-stop algorithms, given their flexibility in terms of cancellation. This choice allows for a more adaptable and efficient parallelization scheme within the context of the simulation. In general, the parallelization scheme is presented within a `single` region using a `for` loop that spawns tasks corresponding to the number of blocks.
The correct updating of lattine Energy and Magnetization is achieved working with private variables `E_loc` and `M_loc` and only periodically update the lattice state using the proper methods and `atomic` instructions.

## 5.8. Class: `SlidingWindow`

This class represents a simulation framework for the Ising model with a sliding window parallelization strategy. It inherits from the abstract class `AbstractMonteCarloSimulation` and provides specific implementations for key methods in order to implement the idea oh having freezed boundary spins and periodic tranlsation of the lattice matrix. To porsuit this goal two additional members are present:

- `NumSlide`: numerical attribute rapresenting how many time the lattice must be translated. It needs fine tuning to achieve good performance and correct results.
- `translate_matrix`: Translate the matrix by a vector $\mathbf{v} = (1,1)$. This operation is meaningful and permissible, taking into account the periodic boundary conditions.

### `simulate_phase_transition`

The method in the parallel implementation performs the same work as in the serial case. However, periodically, after a certain number of iteration (determined by `numSlide`), synchronization occurs. After this synchronization, the lattice undergoes a translation operation trough method `translate_matrix` to update boundary states.

### `simulate_step`

Similarly, this method in the parallel implementation closely resembles its serial counterpart. However, random numbers are extracted in parallel by each thread, and if these numbers correspond to sites on the lattice's border, no action is taken. This adjustment ensures that the simulation follows the desired behavior while efficiently leveraging parallel processing capabilities.

### Design Rationale

The adoption of the sliding window parallelization strategy is motivated by its efficiency in preserving dynamic parallelism. This strategy allows for periodic lattice translation during the phase transition and accounts for frozen boundary conditions in the simulation step, thereby enhancing the realism of the Ising model simulation.

It is essential to note that while this strategy offers advantages in parallel processing, there may be potential overhead due to synchronization and the periodic execution of translation tasks.

### Usage and Extension

The class is designed to be extended for specific lattice models. Inheriting from `AbstractMonteCarloSimulation`, derived classes must implement pure virtual methods, ensuring adherence to the prescribed interface. This design promotes scalability and performance in parallel simulations.

## 5.9. Class: `DomainDecomposition`

This class serves as a simulation framework for the Ising model, employing a domain decomposition parallelization strategy, where boundary flip are performed atomically. It inherits from the abstract class `AbstractMonteCarloSimulation` and provides specific implementations for key methods. The primary focus is on managing diverse computational tasks concurrently. An additional member is introduced to realize the concept of domain decomposition:

- `atomic_flip`: An atomic version of the spin-flipping function, specifically designed for boundary sites to ensure thread-safe operations.

### `simulate_phase_transition`

In the parallel implementation, this method mirrors the serial case's functionality except for the needing of the mechanusm for creating tasks.

### `simulate_step`

Similarly, this method in the parallel implementation closely resembles its serial counterpart. However, random numbers are extracted in parallel by each thread. If these numbers correspond to sites on the lattice's border, the `atomic_flip` method is utilized to ensure thread safety. This modification ensures that the simulation adheres to the desired behavior while efficiently utilizing parallel processing capabilities.

### Design Rationale

The adoption of the domain decomposition parallelization strategy is motivated by its effectiveness in handling diverse computational tasks concurrently. Unlike sliding windows, tasks in this strategy do not require synchronization once spawned; they proceed independently until completion. The only potential overhead is associated with atomic operations, which ensures thread-safe boundary updates during the simulation step.

### Usage and Extension

This class is designed to be extended for specific lattice models. Inheriting from `AbstractMonteCarloSimulation`, derived classes must implement pure virtual methods, ensuring adherence to the prescribed interface. This design promotes scalability and performance in parallel simulations.

## 5.10. Class: `CheckBoard`

This class represents a simulation framework for the Ising model with a checkerboard parallelization strategy, where two sublattice alternate the flipping operations avoiding conflicts on boundary spins, to visualize the pattern there is 6. Similar to the previous, it inherits from the abstract class `AbstractMonteCarloSimulation` and provides specific implementations for key methods to incorporate checkboard pattern beside common feature specified in previous sections. An additional members contribute to this strategy:

- `NumFlipPerBlock`: A numerical attribute representing the number of spins flipped per block during each sublattice switch. Fine-tuning this parameter is crucial for achieving optimal performance and accurate results.

**NOTE:** here to avoid waste of available threads, the number of blocks is two time the mumber of threads, in that way all the thread wark durig alternating phase.

### `simulate_phase_transition`

This method in the parallel implementation performs the same work as in the serial case, but with task management. However, it operates on a checkerboard pattern: two phases are alterneted, one regarding odd index and one even one. After a certain number of iterations per block fixed by `NumFlipPerBlock` synchronization occours and the simulation switches to other sublattice.

### `simulate_step`

This method in the parallel implementation closely resembles its serial counterpart, since now boundary problem is menaged in calling method.The difference is only in random numbers that are extracted in parallel by each thread.

## Design Rationale

The selection of the checkerboard parallelization strategy is driven by its efficiency in concurrently managing diverse computational tasks. Similar to the sliding window approach, this strategy involves tasks operating independently, necessitating synchronization only upon their initiation. However, it's noteworthy that in the checkerboard strategy, synchronization events may occur more frequently, especially when switching between sub-lattices. This increased frequency of synchronization could potentially introduce significant overhead, particularly if the parameter `NumFlipPerBlock` is not chosen judiciously.

## Usage and Extension

The class is designed to be extended for specific lattice models. Inheriting from `AbstractMonteCarloSimulation`, derived classes must implement pure virtual methods, ensuring adherence to the prescribed interface. This design promotes scalability and performance in parallel simulations.

## 6. Benchmark

### 6.1. Strategy

The benchmarking strategy is designed to systematically evaluate the performance of various Ising Monte Carlo simulation methods across different lattice sizes ($L$). The objective is to automate the simulation process by taking user-defined parameters and then executing the simulations for different methods in an organized manner, with results being stored systematically.

**User-defined Parameters**

- `L_MIN` and `L_MAX`: These parameters determine the range of lattice sizes for the benchmarking process. The step is to double it at each simulation.
- `T_MIN`, `T_MAX`, `T_STEP`: The minimum, maximum, and step values for the temperature, allowing exploration of the phase space.
- `interactionStrength`: This parameter represents the strength of interaction in the Ising model, influencing the behavior of the simulation.
- `NUMTHREAD`: The user specifies the number of threads for parallel simulations to harness multicore processing capabilities.
- `filename`: A user-defined filename to store the performance results of the benchmarking process in an organized manner.

**Automated Simulation Process**

The benchmarking process iterates over varying lattice sizes ($L$) following an empirical law to decide the number of iteration to reach equilibrium `IT` , specifically $L^{4.25}$. This approach helps maintain a consistent basis for evaluating different methods.

For each lattice size, the following steps are executed:

1) **Serial Metropolis Simulation:** Utilizes the `SerialMetropolis` class to simulate the Ising model serially. The execution time is recorded.
2) **Domain Decomposition Simulation:** Uses the `DomainDecomposition` class for parallel simulation employing the domain decomposition strategy. The execution time is measured.
3) **Sliding Window Simulation:** Employs the `SlidingWindow` class for parallel simulation with the sliding window parallelization strategy. The execution time is captured.
4) **Performance Results Storage:** The execution times for each simulation method at the current lattice size are stored. A performance file is generated for each lattice size ($L$), containing the execution times for serial, domain decomposition, and sliding window simulations.

This automated and systematic approach allows for a comprehensive evaluation of the efficiency and scalability of each Ising Monte Carlo simulation method, facilitating analysis and comparison of results across different lattice sizes.

**Run Instructions**

To execute the benchmark simulation, follow the steps below:

1) **Makefile Compilation:** The provided code includes a Makefile for easy compilation. Open a terminal and navigate to the directory containing the source code and the Makefile. Run the following command:

```
make
```

This will compile the source code and generate an executable file named `simulation`.

2) **Run the Simulation:** After successful compilation, run the simulation executable with the following command:

```
./simulation
```

The simulation will execute automatically, iterating over the specified lattice sizes and simulation methods based on the user-defined parameters. The results will be stored in an organized manner in performance files.

These instructions provide a simple and straightforward way to compile and run the benchmark simulation on your system. Adjustments to the compilation and execution processes can be made according to specific system configurations if necessary.

## 6.2. Physical results

The simulation results demonstrate that all methods, , are capable of replicating the phase transition in the 2D Ising model. However, it is crucial to note that the effectiveness of some method depends on proper tuning of specific parameters to ensure ergodicity and accurate results.

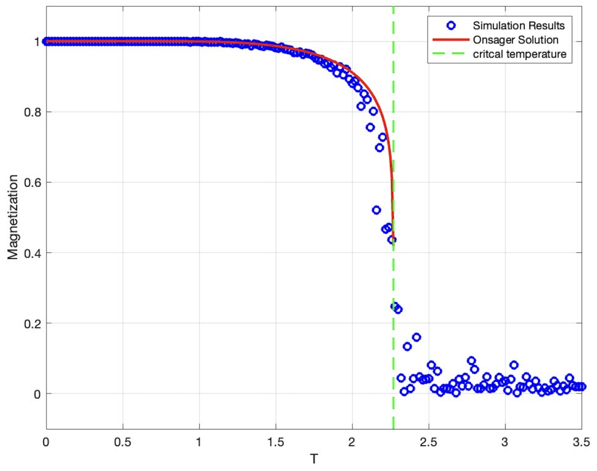

Figure 6. Magnetization over temperature plot, blue dots are results from sliding window alghoritm, while red line represent exact solution. Critical temperature is finded at T = 2.27K as predicted by theory.

Indeed in the case of the Sliding Window and Checkerboard methods, the translation period and switching period, respectively, play a significant role in achieving reliable simulations. Fine-tuning these parameters is essential to maintaining ergodicity throughout the simulation. If not properly adjusted, these methods may require more iterations to converge and yield accurate results.

On the other hand, the Domain Decomposition parallel strategy proves to be advantageous, especially when studying the dynamics of the phase transition. This method demonstrates better accuracy in reproducing the equilibrium dynamics for a single temperature, in accordance with Serial Metropolis, making it a preferable choice for investigations that require a detailed understanding of the transition process and equilibrium fluctuation.

In summary, while all parallelization strategies can successfully capture the phase transition, the Domain Decomposition method stands out for its ability to provide more accurate and reliable results, comparable with serial one, without the need for extensive parameter tuning. If the aim is to explore the dynamic aspects of the transition may find Domain Decomposition to be a more efficient and robust choice for the simulations.

## 6.3. Performance

The performance of the parallel implementations: sliding window, domain decomposition, and checkerboard was evaluated in comparison to the serial implementation.

### Sliding Window

For domain decomposition, the number of matrix translations was chosen to be half the block size, ensuring that all spins become border spins at least once during the simulation. However, the performance of domain decomposition was observed to be considerably worse than the serial implementation. This degradation in performance is likely attributed to the synchronization tasks required at each matrix translation, also considering that the flip method accepting or rejecting spins flip can bring to unbalanced workload expecially near the equilibrium.

### Checkerboard

In the case of checkerboard parallelization, the number of flips per block was set to 10% of the spins in a block. Similar to sliding window, the performance was notably worse than the serial implementation, indicating potential overhead due to synchronization tasks during sublattice switching operations, where previous consideration on worload hold also in this case.

**Speedup in Domain Decomposition**

In domain decomposition, some speedup was observed, and the magnitude of this improvement was found to be dependent on the number of threads. The speedup $SU_p$, where $p$ is the number of thread, was calculated as the ratio of the serial execution time to the parallel execution time. The results were as follows:

- $SU_{p=4}$ no speed up
- $SU_{p=16} = 1.1$
- $SU_{p=36} = 1.36$
- $SU_{p=64} = 2.81$
- $SU_{p=121} = 2.36$

These speedup values indicate that domain decomposition achieves improved performance with an increasing number of threads.

**Lattice Size Influence**

It was anticipated that an increase in lattice size might lead to performance improvements since the percentage of border spins and, consequently, atomic operations would decrease. However, the observed speedup exhibited little fluctuation, and there was no substantial improvement as the lattice size increased. This could be attributed to limitations in exploring higher-dimensional lattices, as the serial implementation becomes increasingly time-consuming.

## 6.4. Checkboard in CUDA vs OpenMP

In the context of our work the Checkerboard algorithm exhibits exceptional performance on CUDA. When implemented on GPU architectures. The sheer abundance of Arithmetic Logic Units (ALUs) in them allows for the creation of an exceptionally fine-grained checkerboard, facilitating an extensive parallel execution of computational tasks. On the other and the limited CPU capabilities in terms of multi core make the algorithm's performance constrained by the need for synchronization and efficient management of parallel workloads. The delicate balance between maximizing parallelism and minimizing synchronization overhead remains a critical consideration for achieving optimal results with the Checkerboard algorithm on CPU architectures.

## 7. Equilibrium detection

### 7.1. Comparison with Exact Solution

### Introduction

The investigation of phase transitions in statistical physics, particularly within the framework of the Ising model, often necessitates the delicate task of detecting equilibrium states. Detecting equilibrium is crucial for unraveling the behaviors of physical systems at different temperatures and gaining insights into the emergence of macroscopic order. In this pursuit, a key methodology revolves around comparing simulation outcomes with the exact solution provided by Onsager's formula for the Ising model, assuming a constant interaction strength ($J$).

Onsager's formula for the magnetization ($m_{\text{exact}}$) in a two-dimensional Ising model without an external magnetic field is given by:

$$m_{\text{exact}} = (1 - \sinh^{-4}(2J))^{\frac{1}{8}} \tag{4}$$

The equilibrium state is identified when the absolute difference between the simulated magnetization per site ($m$) and the exact magnetization ($m_{\text{exact}}$) falls below a predefined tolerance. Mathematically, this condition is expressed as:

$$|m - m_{\text{exact}}| < \text{tolerance} \tag{5}$$

The tolerance is often defined in terms of the percentage of spins that are not aligned and can be adjusted by the user.

While this method may seem deceptively simple, its utility is evident in its effectiveness. It is usefull for exploring convergence steps, analyzing fluctuations at equilibrium, and studying the dynamic behavior of the system to equilibrium. The precision in capturing equilibrium dynamics enhances our understanding of phase transitions and critical phenomena in the Ising model, contributing significantly to the broader field of statistical physics.

### Serial implementation

Consist of just adjusting the class in order to accept a `tolerance` data member. Then the only major change is in `simulate_phase_transition` method where every *Montecarlo step*, namely every *total number of spins* flips, the error is evaluated, and if the condition 5 is reached the simulation terminates.

## 7.2. Parallel Equilibrium Detection

In parallelizing the equilibrium detection method within the Ising model simulation, the primary focus was on the domain decomposition algorithm, given its superior performance characteristics. The challenge at hand was to devise an efficient strategy for evaluating the error while minimizing the impact on performance, particularly in the context of synchronization tasks.

The core of the parallel equilibrium detection revolves around a shared control mechanism: a common boolean variable, denoted as `continue_flag`. Each thread loops over flip while `continue_flag = true`, in the meanwhile upon completing a fixed number of Monte Carlo steps ($N$/NUMTHREAD, where $N$ is the total number of spins) it performs the following operation:

1) atomically updates the magnetization
2) checks for the error condition,
3) `if` the condition is satisfied, atomically sets the `continue_flag` to `false`.

This design ensures that when a thread raises the `continue_flag`, it triggers a collective halt in the simulation. While some threads may continue with a few remaining flips, their impact on the final result is minimal. This approach strikes a balance between efficient parallel execution and the timely termination of the simulation when the equilibrium condition is met. The strategic use of atomic operations facilitates concurrent updates without necessitating explicit synchronization at every step, allowing for a seamless integration of parallelism into the equilibrium detection process.

## 8. Alternative Convergence Criterion

In contrast to the fixed number of simulation steps approach, an alternative convergence criterion is employed, dynamically assessing the simulation's progress based on the fluctuation of magnetization. This method focuses on the standard deviation (SD) of the last few magnetization values, adapting the termination decision to the behavior of the system rather than a predefined number of steps.

The simulation continues until either the fluctuation falls below a specified threshold or a maximum number of steps is reached. The fluctuation is determined by calculating the SD of the magnetization values recorded over a rolling window. If the SD becomes sufficiently small, indicating that the system has stabilized, the simulation is terminated.

This adaptive approach ensures that the simulation does not unnecessarily extend beyond the point of convergence, allowing for efficient exploration of the temperature space. The threshold for fluctuation can be fine-tuned based on the desired level of accuracy and computational efficiency. Additionally, this criterion is particularly useful for studying fluctuations near equilibrium, providing insights into the dynamic behavior of the Ising model as it transitions between different phases. The dynamic stop criterion thus offers a flexible and responsive mechanism to capture the equilibrium state of the Ising model while also facilitating a detailed analysis of its fluctuation dynamics.

## 9. Conclusion And Further Development

In conclusion, the parallelization of the Ising model presented a significant challenge, and while we achieved notable speed-ups with domain decomposition, sliding window, and checkerboard methods, the journey involved valuable insights. Implementing parallel methods also without speedup was crucial for comprehensively understanding the problem and refining the implementation.

The achieved speed-ups with domain decomposition, underscore the potential of parallel computing for simulating phase transitions. However, it's essential to acknowledge that the performance of the sliding window and checkerboard methods could benefit from finer parameter tuning or exploration on alternative architectures, so never say never.

Beyond the current achievements, there are exciting prospects for enhancing the parallel Ising model simulations by combining our methods with other parallelism techniques. One intriguing avenue is exploring coarser-grained parallelism strategies, such as those used in temperature tempering algorithms developed by our project group.

Collaboration with parallel computing methodologies like MPI (Message Passing Interface) could open new possibilities. Combining the sliding window approach with MPI with Domain Decomposition OpenMP, for instance, might lead to an even more scalable and efficient simulation framework, leveraging the strengths of both methodologies.

Looking forward, an important next step is to fully implement and integrate convergent methods into the simulation framework,likely also before the exam since there is a good base. This will make the program not only an effective tool for exploring phase transitions but also an initial resource for researcher in the physics Department at Politecnico di Milano from which we had the hint for the project.

## References

[1] M. E. J. Newman and G. T. Barkema. *Monte Carlo methods in statistical physics*. Oxford: Clarendon Press, 1999.

[2] Joshua Romero et al. “High performance implementations of the 2D Ising model on GPUs”. In: *Computer Physics Communications* 256 (Nov. 2020), p. 107473. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2020.107473. URL: http://dx.doi.org/10.1016/j.cpc.2020.107473.