Parallel Monte Carlo Simulation

Parallel Lattice Monte Carlo Simulation[1]

In this section, we implement the MC simulation for a 2-dimensional Ising model on parallel computers, based on a spatial decomposition scheme.

SPATIAL DECOMPOSITION—STRIPS

In parallel computing, a crucial step is to decompose the computing task into subtasks, each of which is assigned to a processor in a parallel computer. A simple parallelization scheme is called **spatial decomposition**, in which the physical space is decomposed into subsystems of equal size. For a $L \times L$ 2-dimensional Ising-spin lattice, the simplest decomposition is based on strips. Suppose that there are P processors (labeled 0, 1,..., P-1), then we can partition the lattice into P strips, each consisting of L/P rows of spins. Processor p is assigned spin rows from $(P/L)p$ to $(P/L)(p + 1)$.

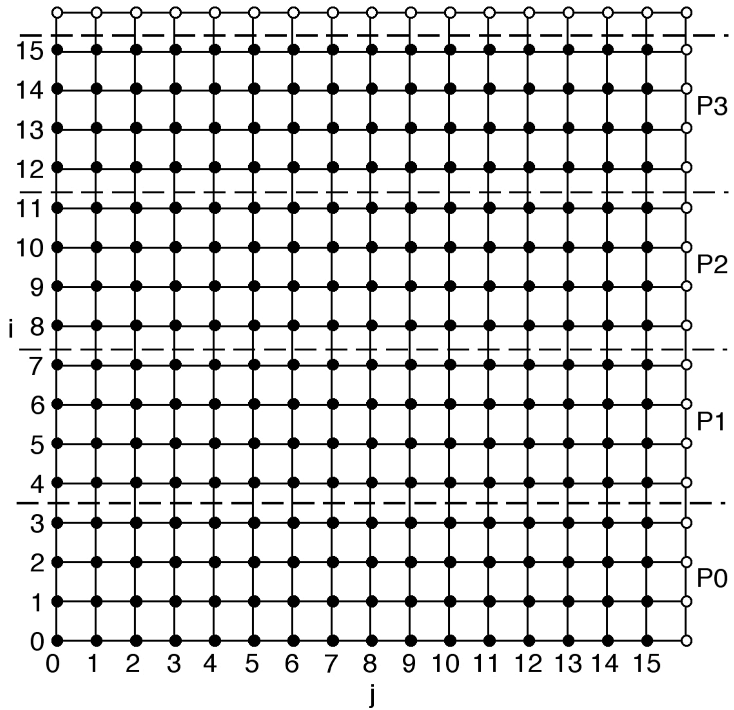


Figure: 16×16 grid points are partitioned using strips into 4 subsystems, each of which is assigned to a processor in a parallel computer.

Interprocessor caching: Note that, for example, the top-row spins in processor 2 interact with the bottom-row spins in processor 3. Similarly the bottom-row spins in processor 2 interact with the top-row spins in processor 1. In order for processor 2 to compute energy changes associated with updating its resident spins, it must "cache" the bottom row of processor 3 and the top row of processor 1 to itself prior to computing these energy changes. Each strip is thus augmented with the top and bottom cached spin rows.

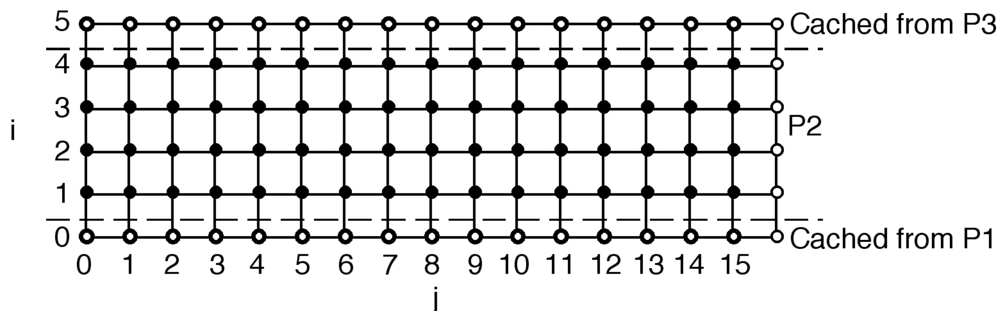


Figure: Augmented strip with cached top and bottom spin rows.

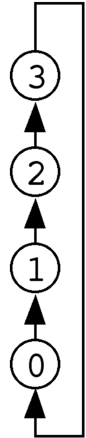
The "caching" operations are implemented by message passing—sending and receiving data through the communication links between processors. In order to perform message passings, each processor needs to know its identity and the destination to whom it is sending a particular message. In a message

[1] D. W. Heermann and A. N. Burkitt, *Parallel Algorithms in Computational Science* (Springer-Verlag, Berlin, 1991).

passing software system such as the MPI (message passing interface), a process rather than a processor is the proper unit to perform a subtask since multiple processes can be created per processor. The system keeps track of the sequential IDs of currently running processes, 0, 1,..., P-1 (not the Unix process ID). The system also provides a mechanism for a running process to inquire its sequential ID. Each process should have the following variables:

`int sid`	The sequential process ID of this process
`int up_id = (sid + 1)%P`	The sequential process ID of the process handling the upper strip
`int lw_id = (sid + P - 1)%P`	The sequential process ID of the process handling the lower strip

Note that the processes logically form a ring (or torus). Message passings often take a circular pattern as shown in the Figure.



Global summation: To compute physical quantities like the magnetization, each process performs summation over its "resident" spins. These "local" contributions are then summed globally over processes. Most message passing systems provide subroutines to perform such global summations.

Single program multiple data (SPMD) program: Our parallel MC program will be based on the SPMD model. Namely, multiple copies of a single executable are created as separate processes. Each process starts executing the program from the first line of code. After calling a system call inquiring its sequential process ID, each process knows its identity. So each process picks up a proper strip and starts executing MC simulations locally. Except for occasional message exchanges and global operations, the simulation proceeds asynchronously, i.e., there is no global clock synchronizing the processes.

Parallel random-number generation: Because of the asynchronicity among processes, initialization of a random-number sequence using timing command will probably result in different sequences for different processes, if the timing routine uses many clock ticks (~ 10^3) per second. Otherwise, use the sequential process ID to initialize a sequence, e.g., sid + 1 (to avoid 0 seed).

Data Structures

`int L`: Number of grid points in each direction
`int P`: Number of processors; `LP = L/P` is the number of spin rows per processor; `LP2 = LP + 2` is the number of augmented rows including the top and bottom cached rows.
`int s[LP2][L]`: Spin variables. `s[0][]` and `s[LP+1][]` are the cached bottom and top spin rows, and `s[i][]` are the resident spin rows for i = 1, ..., LP.

DETAILED-BALANCE CONFLICT

In parallel Ising-spin MC simulations, different processes update different spin sites. However, we need to assure that concurrent spin updates are mutually independent. Otherwise, the detailed-balance condition—the foundation of the Metropolis algorithm—will be violated.

Let's consider a simple case of 1-dimensional Ising-spin lattice.

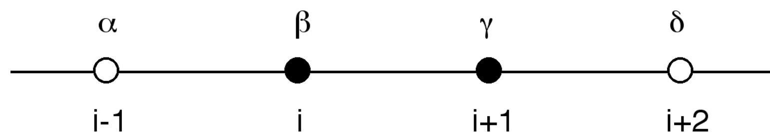


Figure: One-dimensional Ising spin model.

Suppose that processes 0 and 1 are concurrently flipping spin i and i+1, respectively. We denote the current spin variables at lattice points, i−1, i, i+1, and i+2 as α, β, γ, and δ, respectively. Since each

process is doing their updating independently, the transition probability for both flips to be accepted is given by

$$\Pi_{-\beta,-\gamma\leftarrow\beta,\gamma} = \min\left[1,\exp\left(-2J(\alpha\beta+\beta\gamma)/k_BT\right)\right]\min\left[1,\exp\left(-2J(\beta\gamma+\gamma\delta)/k_BT\right)\right].$$

In fact, the correct transition probability for flipping two consecutive spins to satisfy the detailed-balance condition is

$$\Pi_{-\beta,-\gamma\leftarrow\beta,\gamma} = \min\left[1,\exp\left(-2J(\alpha\beta+\gamma\delta)/k_BT\right)\right],$$

the value of which can be different from that of the previous expression. For example, suppose that the current spin configuration is given by (α,β,γ,δ) = (1,1,1,−1). Then the former individual updates give the transition probability,

$$\Pi_{-\beta,-\gamma\leftarrow\beta,\gamma} = \exp\left(-4J/k_BT\right),$$

while the latter correct transition probability is

$$\Pi_{-\beta,-\gamma\leftarrow\beta,\gamma} = 1.$$

By updating two consecutive spins independently, therefore, we are violating the detailed-balance condition and thus it is no longer a valid Markov chain to achieve the equilibrium probability density.

We must avoid updating consecutive spins simultaneously. In our strip-decomposition implementations, we will only consider cases where each strip consists of at least two spin rows. If each process updates its spins row-by-row starting from the top row toward the bottom, the conflict can be avoided.

Algorithm: Parallel MC simulation of a 2-dimensional Ising model

---

```
/* Main program */
Initialize the spins, s[i][j] (1 ≤ i ≤ L/P; 0 ≤ j ≤ L-1)
LocalSum_A = 0
for sweep =1 to maximum_sweep
  Send the bottom-row spins cyclically downwards, s[1][]
  Receive the top-cache-row spins, s[L/P+1][], cyclically from the strip above
  for i = L/P downto 2
    Update the i-th row spins, s[i][]
  endfor
  Send the top-row spins cyclically upwards, s[L/P][]
  Receive bottom-cache-row spins, s[0][], cyclically from the strip below
  Update the bottom row spins, s[1][]
  LocalSum_A = LocalSum_A + A(s^N)
endfor
Calculate GlobalSum_A as a global sum of LocalSum_A over all processes
Average_A = Sum_A/maximum_sweep

/* Update the i-th row spins */
for j = 0 to L-1
  Compute the change in potential energy, δV, with a single spin flip, s_i,j → -s_i,j
  if dV ≤ 0
    accept the flip, s_i,j ← -s_i,j
  else if rand() ≤ exp(-δV/k_BT) then
    accept the flip, s_i,j ← -s_i,j
  endif
endfor
```

MPI Implementation of Parallel Lattice MC

STRIP ID

Each process uses the following code to determine its own sequential process ID (rank) and its upper and lower neighbors' ranks.

```
MPI_Comm_rank(MPI_COMM_WORLD, &sid);
up_id = (sid + 1)%P;
lw_id = (sid + P - 1)%P;
```

SPIN-ROW CACHING

The following code can be used to cache a spin row. We use integer arrays, `bufs` and `bufr`, to compose and receive a message, respectively. Integers `i_send` and `i_recv` are the rows to be sent and received. For example, if a process is caching the top extra row, `i_recv = L+1`, from the upper neighbor, then it needs to send the bottom resident row, `i_send = 1`, to the lower neighbor. In this example, the destination of its message, `dest`, is `lw_id`.

```
int bufs[L],bufr[L];
/* Message buffering */
for (j=0; j<L; j++)
  bufs[j] = s[i_send][j];
/* Message exchange */
if (myparity == 0) { /* Even node: send & recv */
  MPI_Send(bufs,L,MPI_INT,dest,10,MPI_COMM_WORLD);
  MPI_Recv(bufr,L,MPI_INT,MPI_ANY_SOURCE,10,MPI_COMM_WORLD,&status);
}
else if (myparity == 1) { /* Odd node: recv & send */
  MPI_Recv(bufr,L,MPI_INT,MPI_ANY_SOURCE,10,MPI_COMM_WORLD,&status);
  MPI_Send(bufs,L,MPI_INT,dest,10,MPI_COMM_WORLD);
}
else /* Single layer: Exchange information with myself */
  for (j=0; j<L; j++) bufr[j] = bufs[j];
/* Message storing */
for (j=0; j<L; j++)
  s[i_recv][j] = bufr[j];
}
```

DEADLOCK AVOIDANCE

A circular send and receive relation could cause a deadlock. In the following code, suppose that the destination node, `inode`, is defined as in the figure below.

```
MPI_Send(bufs,L,MPI_INT,inode,10,MPI_COMM_WORLD);
MPI_Recv(bufr,L,MPI_INT,MPI_ANY_SOURCE,10,MPI_COMM_WORLD,&status);
```

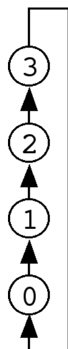


Figure: Circular send-receive relation.

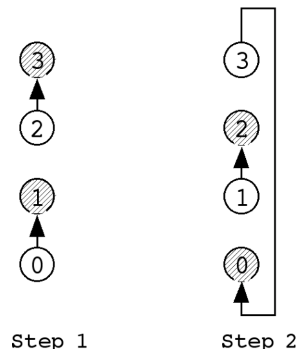


Figure: Avoiding circular message passing.

With a finite buffer size in the receiver's communication system, a sender blocks until the receiver's buffer is cleared. However in the example above, each processor cannot start receiving until its send operation is completed.

To avoid this deadlock, we classify the processors into even and odd processors. We use `int myparity` to represent the parity of the sequential processor ID. `myparity` is 0|1 if the processor ID, `sid`, is even|odd. If there is only one processor, `myparity` = 2. In this case, no message passing is performed but rather the row `s[1][]` is duplicated at `s[L+1][]` and `s[L][]` is duplicated at `s[0][]`. The wraparound condition is then implemented without performing the modulo operation. In this way, the resulting MPI code works even on a single processor.

MAGNETIZATION

After each processor computes the local sum, `double localM`, of the spins in its strip, the following call computes the global sum, `double globalM`, of the spins over all the strips.

```
MPI_Allreduce(&localM, &globalM, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```