Abelian Sandpile code explanation

This version uses dynamic board dimensions (user input) and starts with all cells initialized to **4 grains**.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer code

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A white background with black text

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer code

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

MPI logic

**🧠 The Sandpile Simulation Recap (Sequential)**

In the **sequential** (normal, single-process) version:

1. You have a 2D board of integers.
2. A cell is **unstable** if it has value >= 4.
3. If it is unstable, it "topples":
   * It loses 4 grains.
   * Each of its 4 neighbors gets 1 grain.
4. You repeat this until no cells are unstable anymore.

This is a **synchronous method** — meaning:  
→ You first **compute** what the next state will be using a separate array,  
→ then **copy** the result into the real board.

**🧩 Goal of MPI Parallelization**

Make the program **run faster** by letting **multiple processes (or cores)** each handle **a portion** of the board **at the same time**.

This is called **data parallelism** — splitting the board into parts.

**👷 How MPI Parallelization Works (Step-by-Step)**

**Step 1: Split the Board Between Processes**

Assume your board is 10 rows × 10 columns and you have 2 processes:

* **Process 0**: handles rows 0–4
* **Process 1**: handles rows 5–9

Each process works only on **its own part** of the board.

**Step 2: Add "Halo Rows"**

To update the edges of your sub-board correctly, you need **information from neighbor processes**.

So you add **extra rows**:

* One at the top,
* One at the bottom.

Each process now has:

* Its real rows
* Plus 1 extra row from the neighbor above
* Plus 1 extra row from the neighbor below

These extra rows are called **halo rows** (or ghost cells).

**Step 3: Halo Exchange**

Each process **sends** its top/bottom border rows to its neighbor, and **receives** the neighbor's row.

* Process 0 sends its last row to Process 1 (bottom halo),
* Process 1 sends its first row to Process 0 (top halo),
* This allows them to compute topples correctly at the shared boundary.

This is done with MPI\_Sendrecv or MPI\_Isend/MPI\_Irecv.

**Step 4: Compute the Next State (Like Before)**

Each process:

* Creates a **local buffer** for new values (like in your code),
* Loops over **its part of the board** (excluding halos),
* Topples unstable cells,
* Writes the result into the buffer.

Then the buffer is copied back to the main sub-board.

**Step 5: Check if We Are Done (Converged)**

Each process checks:  
**Did any cell topple?** → changed = 1 if yes, else 0.

Then all processes share this information using:

MPI\_Allreduce(&changed, &global\_changed, 1, MPI\_INT, MPI\_LOR, MPI\_COMM\_WORLD);

* This tells everyone if **anyone** had a topple.
* If no one did, we are done.

**🧼 Summary in Simple Terms**

| **Sequential Code** | **MPI Parallel Code** |
| --- | --- |
| One big board | Each process has a piece of the board |
| Single stabilizeBoard() function | Each process runs its own stabilization loop |
| Single buffer | Each process has its own buffer |
| Accesses all neighbors directly | Use **halo exchange** to get neighbors' edges |
| Loop until no more changes | All processes check changed, reduce to global\_changed |

HALOS

We have 2 processes

Original Board:

[ 0 ]

[ 1 ] Process 1 owns rows 0-2

[ 2 ] ← Process 0

[ 3 ] ← Process 1

[ 4 ] Process 2 owns rows 3-5

[ 5 ]

Process 1's view with halo:

[ H ] ← halo row from Process 2 (fake row 3)

[ 0 ]

[ 1 ]

[ 2 ]

[ H ] ← halo row from Process 2 (not needed here, but shown for symmetry)

Process 2's view with halo:

[ H ] ← halo row from Process 1 (fake row 2)

[ 3 ]

[ 4 ]

[ 5 ]

**Why Halos Are Necessary**

Consider this rule from your sandpile simulation:

Each cell gives 1 grain to each of its 4 neighbors if it has ≥ 4.

So, to update cell (x, y), you need values of:

* (x-1, y) → above
* (x+1, y) → below
* (x, y-1) → left
* (x, y+1) → right

If (x+1, y) is **in another process**, you still need its value!  
⟶ That’s why you store it in a **halo cell**.

**Do MPI processes have a *waiting period*?**

**✅ Yes — during halo exchange, processes:**

1. **Send** their updated edge rows/columns to neighbors
2. **Wait to receive** the edge data (halo) from neighbors

**🔄 This ensures synchronization at every simulation step.**

**🧠 Blocking vs Non-Blocking Matters**

**1. 🔴 Blocking (e.g., MPI\_Send, MPI\_Recv, MPI\_Sendrecv)**

* The process **waits** during the communication.
* You **can't compute anything** while waiting.
* **Easy to write**, but can create **idle time**.

**2. 🟢 Non-blocking (e.g., MPI\_Isend, MPI\_Irecv + MPI\_Wait)**

* Start communication and then **do computation** while the messages are in flight.
* When you're ready to use the halo data, you **wait only if it's not done yet**.
* This enables **communication-computation overlap**, reducing idle time.
* More complex, but **much faster** for large grids and many steps.

**🔬 Example: Overlap Communication and Computation**

If your halo update looks like this:

// Start non-blocking send/recv

MPI\_Irecv(...);

MPI\_Isend(...);

// Compute interior cells (not touching the halo)

compute\_inner\_cells();

// Wait for halos to arrive

MPI\_Wait(...);

// Compute edge cells (that need the halo data)

compute\_edge\_cells();

This lets you **hide communication time** behind computation — a classic optimization for stencil codes.

**Suggested Learning Path**

1. **Start with your serial code.**
2. **Parallelize with MPI using blocking MPI\_Sendrecv for halo exchange.**
3. **Test correctness thoroughly** (make sure the final stable sandpile matches serial output).
4. **Once comfortable, implement non-blocking MPI with overlap.**
5. **Profile and compare performance.**

DON’T DECLARE VARIABLES INSIDE LOOPS

OPTIMIZE THE SERIAL SOLUTION

BUILD THE SYNCHRONOUS AND ASYNCHRONOUS VERSION AND USE THE FASTER ONE

MAY WANT TO LOOK AT PERSISTENT COMMUNCATION

How to run and compile

Type the following command in terminal:

run\_mpi.bat

MPI\_Send(&data, count, MPI\_INT, dest\_rank, tag, MPI\_COMM\_WORLD);

&data: pointer to data to send

count: number of elements

MPI\_INT: type of data

dest\_rank: which process to send to

tag: an ID for the message (can be any int)

MPI\_COMM\_WORLD: communicator (default group of all processes)

Example

if (rank == 0) {

int number = 42;

MPI\_Send(&number, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD);

printf("Process 0 sent %d to Process 1\n", number);

MPI\_Recv(&buffer, count, MPI\_INT, src\_rank, tag, MPI\_COMM\_WORLD, &status);

&buffer: where to store received data

count: max number of elements

MPI\_INT: type of data

src\_rank: which process to receive from (or MPI\_ANY\_SOURCE)

tag: message tag to match (or MPI\_ANY\_TAG)

status: receives info about the message (optional)

Example

else if (rank == 1) {

int number;

MPI\_Recv(&number, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

printf("Process 1 received %d from Process 0\n", number);