h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Darmstadt University of Applied Sciences
– Faculty of Computer Science –

Implementing a distributed lattice gas cellular automaton using Rust, Rayon, and MPI

by
Zebreus

CONCEPT

In the fourth lab, we built a lattice gas cellular automaton (LGCA) using Rust, Rayon, and MPI. Our LGCA is based on the FHP-I model. It uses a hexagonal grid and does not support resting particles. We will also try using a lookup table to process collisions quickly. We used rayon to divide the processing of the core rows over multiple threads. In addition, we used MPI via the rsmpi Rust bindings for distributed memory parallelism. Our implementation uses AVX512 SIMD instructions to process 64 cells at once. Our solution can process 492 Gigabytes of cell data per second when running on four intel nodes on the Virgo cluster.

REQUIREMENTS

The following requirements are given by the lab instructions:

- Essentially, the lattice gas consists of a hexagonal array (the lattice), and a set of barriers.
- Each cell in the lattice is connected to 6 neighbors (see the illustrations in the report referenced above).
- Time is discrete (i.e., t = 0, 1, 2, 3...).
- Each connection is boolean: Thus, at time t, there is either 0 or 1 particle (but never two or more particles) traveling from any given cell a to any cell b if a and b are neighbors (there might also be a particle traveling from b to a; the two links are independent of each other).
- The state of the lattice at time = t can be calculated in two phases:
- Transport: First, the particles travel linearly. For example, if cells a, b, and c are linear neighbors (e.g., c is to the right of b, and b is to the right of a), then a particle traveling from a to b at time = 1 will continue to travel to c at time = 2 (in the absence of collisions, see below).
- Scattering: Second, collisions with other particles and with the barriers are taken into account. Collisions between particles are non-deterministic: More than one outcome is (sometimes) possible, and the simulation chooses between these outcomes randomly. See page 14 (188) in [Has87] for a table of the possible collisions.
- Particles reflect off barriers.

IMPLEMENTATION CONCEPTS

The FHP-I model uses hexagonal cells. Each cell can contain up to six particles, one in each direction. The directions are called west, north-west, north-east, east, south-east, and south-west. We will store the cells row-wise in an even-r horizontal layout as shown in Figure 1

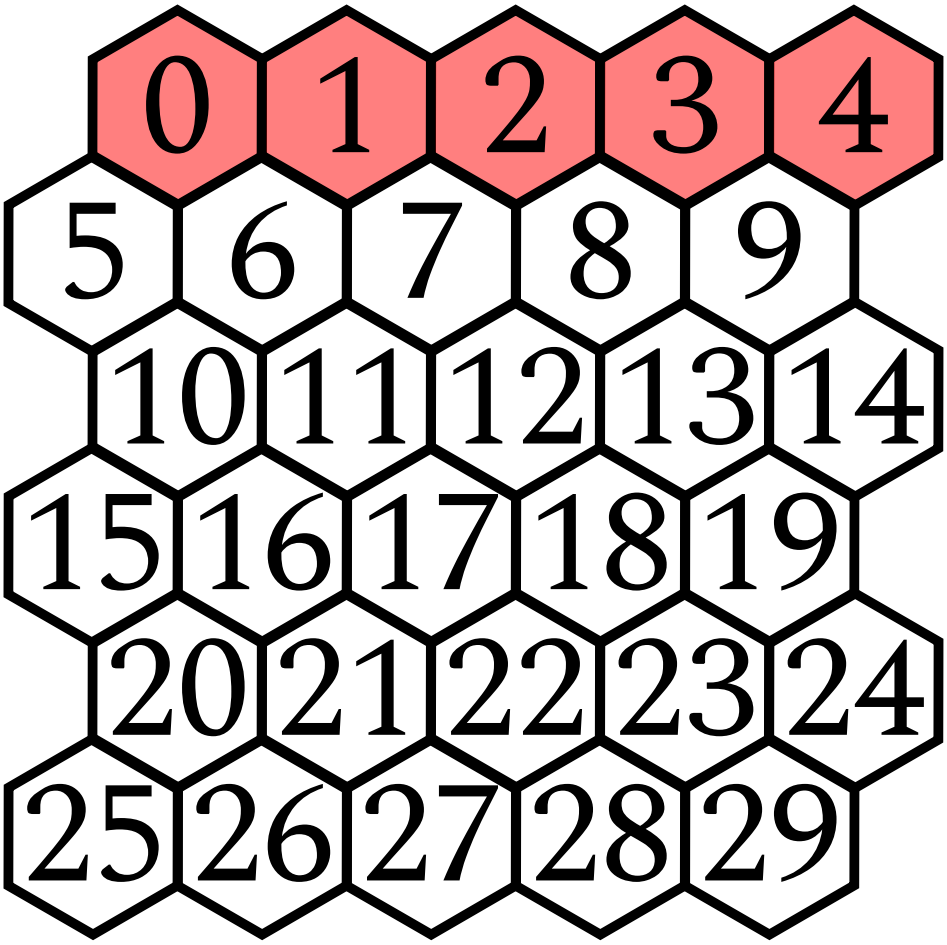

Figure 1. "even-r" horizontal layout

Cells are represented by bytes with one bit for each direction, as shown in Figure 2.

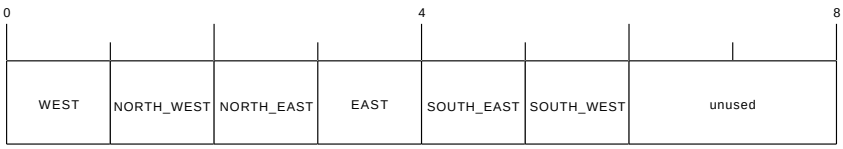

Figure 2. A single cell represented as a byte

We will represent each row as an array of cells. The whole grid is represented as an array of rows. A single row will always be processed in one go from left to right.

First, all neighboring cells are inspected to gather the particles that will move into the current cell. At the grid's edges, the movement step reflects particles that would leave the grid. If the new direction also points outside of the grid, the particle is reflected in the direction it came from.

After the movement step, collisions inside the new cell are processed. The rules for the collisions are quite simple: * If a cell only has two particles traveling in opposite directions, they will rotate by either 60° or -60°. * If a cell has three particles traveling in opposite directions, they will rotate by 60° (or -60°, but that is the same). * If a cell has exactly four particles with two pairs of opposite directions, they will rotate by either 60° or -60°.

We identified four types of rows that need to be processed differently:

- Even row: The first, third, fifth, … rows are even. The core of the row will be processed normally, but the first and last cells need to be processed differently, as they are missing some neighbors.
- Odd row: The second, fourth, … rows are odd. The same as even rows, but the first and last cells are processed differently because other neighbors are missing.
- Top row: The first row of the grid is similar to an even row, but it has no neighbors to the north, so all particles that would move north are reflected.
- Bottom row: The last row of the grid is similar to an odd row, but it has no neighbors to the south, so all particles that would move south are reflected. Because the bottom row is an odd row, our implementation will only support grids with an even number of rows.

IMPLEMENTATION

We define four functions for the four different cases. Each function has a special case for the first and last cell and delegates the core cells of the row to the `process_row_core` function. The top and bottom rows also have a special case for the core of them, as they need to reflect particles that would move out of the grid.

Nearly all particles will be processed in the grid's core for big grids, so we want to optimize that case as much as possible. We will use AVX512 SIMD instructions to process 64 cells at once. We will also try using a lookup table to process collisions quickly.

PROCESSING ROWS

We define four functions for the four different cases. Each function has a special case for the first and last cell and delegates the core cells of the row to the `movement_core` function. The top and bottom rows also have a special case for the core of them, as they need to reflect particles that would move out of the grid.

To calculate the next state of a cell, we need to look at all of its neighboring cells and move the particles into the current cell. The movement logic for core cells is shown in Listing 1. We need special logic to handle the missing neighbors for cells that are not in the core. Instead of the missing neighbor, we use a particle from the current cell as the neighbor. For example, the logic for the top-right cell is shown in Listing 2.

Listing 1. Movement calculation for core cells

```rust
            let mut new_cell = (west.raw & TO_EAST)
                | (north_west.raw & TO_SOUTH_EAST)
                | (north_east.raw & TO_SOUTH_WEST)
                | (east.raw & TO_WEST)
                | (south_east.raw & TO_NORTH_WEST)
                | (south_west.raw & TO_NORTH_EAST);
```

Listing 2. Movement calculation for the top-right corner cell

```rust
    // Handle border of first cell
    result[0].raw = (below[0].raw & TO_NORTH_EAST)
        | (below[1].raw & TO_NORTH_WEST)
        | (current[1].raw & TO_WEST)
        | ((current[0].raw & TO_NORTH_EAST) << 2)
        | ((current[0].raw & TO_NORTH_WEST) << 4)
        | ((current[0].raw & TO_WEST) << 3);
```

Our default collision handling is implemented in the `process_collision` function, as shown in Listing 3. It uses a match statement to map the collision cases to the results. The default cases leave the value unchanged. In cases where there are two options, a real random number is used to select one.

Listing 3. Default collision handling using real randomness

```rust
pub fn process_collision(&mut self) {
    self.raw = match self.raw {
        // Two opposing particles
        const { TO_WEST | TO_EAST } => {
            if RNG.with(|f| f.borrow_mut().gen::<bool>()) {
                TO_NORTH_EAST | TO_SOUTH_WEST
            } else {
                TO_SOUTH_EAST | TO_NORTH_WEST
            }
        }
        const { TO_SOUTH_EAST | TO_NORTH_WEST } => {
            if RNG.with(|f| f.borrow_mut().gen::<bool>()) {
                TO_NORTH_EAST | TO_SOUTH_WEST
            } else {
                TO_EAST | TO_WEST
            }
        }
        const { TO_SOUTH_WEST | TO_NORTH_EAST } => {
            if RNG.with(|f| f.borrow_mut().gen::<bool>()) {
                TO_SOUTH_EAST | TO_NORTH_WEST
            } else {
                TO_EAST | TO_WEST
            }
        }

        // Three particles
        const { TO_SOUTH_WEST | TO_NORTH_WEST | TO_EAST } => {
            TO_SOUTH_EAST | TO_NORTH_EAST | TO_WEST
        }
        const { TO_SOUTH_EAST | TO_NORTH_EAST | TO_WEST } => {
            TO_SOUTH_WEST | TO_NORTH_WEST | TO_EAST
        }

        // Four particles with opposing holes
        0b00110110 => {
            if RNG.with(|f| f.borrow_mut().gen::<bool>()) {
                0b00011011
            } else {
                0b00101101
            }
        }
        0b00011011 => {
            if RNG.with(|f| f.borrow_mut().gen::<bool>()) {
                0b00101101
            } else {
                0b00110110
```

```
                }
            }
            0b00101101 => {
                if RNG.with(|f| f.borrow_mut().gen::<bool>()) {
                    0b00011011
                } else {
                    0b00110110
                }
            }

            // Everything else
            _ => self.raw,
        }
    }
```

While this collision implementation is correct, its performance is quite bad because we generate a random number for every collision. It also has lots of branching, and the compiler cannot auto-vectorize it.

OPTIMIZING

For big grids, nearly all particles will be processed in the core of the grid, so we want to optimize that case as much as possible. As the GSI has machines that support AVX512, we want to write code that utilizes these instructions to process 64 cells at once. While we could have tried to write the movement and collision logic in x86 assembly, we chose to try to write logic that can be auto-vectorized by the compiler when processing large arrays of cells. We used https://godbolt.org to view the generated machine code and make sure that our implementation supports AVX512. To enable auto-vectorization for AVX512 instructions, we used the rust flags shown in Listing 4.

Listing 4. Rust flags for enableing AVX512

```
-C opt-level=3
-C target-
feature=+avx2,+avx,+sse2,+avx512vl,+avx512f,+avx512bw,+avx512cd,+avx512dq,+avx512vnni
```

CONSOLE

The basic function we use for processing the core of a row is shown in Listing 5. It takes read-only references to the current row and the row above and below it. It then iterates through them using sliding windows so all six neighbors of the current cell are available at once. The new state of each cell is calculated and placed into the cell of the result array.

Listing 5. Function for processing the core of a row without collisions

```rust
pub fn process_core<const WIDTH: usize>(
    above: &[u8; WIDTH - 1],
    current: &[u8; WIDTH],
    below: &[u8; WIDTH - 1],
    result: &mut [u8; WIDTH - 2]
) {
    let context_iterator = above
        .array_windows::<2>()
        .zip(current.array_windows::<3>())
        .zip(below.array_windows::<2>())
        .zip(result.iter_mut());

    context_iterator.for_each(
        |(
            (([north_west, north_east], [west, _current, east]), [south_west, south_east]),
            result,
        )| {
            let new_cell = (west & 0b00100000)
                | (north_west & 0b00010000)
                | (north_east & 0b00001000)
                | (east & 0b00000100)
                | (south_east & 0b00000010)
                | (south_west & 0b00000001);
            *result = new_cell;
            // Collision handling omitted for now
        },
    )
}
```

Listing 6 shows the generated assembly for the function. We can see that the compiler autovectorized the function and used the AVX512 instructions to process 64 cells at once. At first, `vpbroadcastb` is used to populate the 512-bit vector registers `zmm0` - `zmm5` with the mask for each direction. The result will be placed into the `zmm6` register. The main loop first uses `vpandq` to set each cell in the result register to the value of the north-western cell masked by a mask that removes all particles except the one moving from northwest to southeast. This also resets the value of all other particles in `zmm6` to zero. Then `vpternlogq` is used with each of the five remaining directions to add the particles that move into the current cell from that direction. After that, `zmm6` is stored back into memory. This way, 64 cells are processed at once. If there are more than 64 cells left, the above is repeated. Otherwise, the program uses non-vectorized logic for the remaining cells.

Listing 6. Assembly for `process_core` *without collisions*

```x86asm
.LCPI0_12:
        .byte   16
.LCPI0_13:
        .byte   32
.LCPI0_14:
        .byte   8
.LCPI0_15:
        .byte   4
.LCPI0_16:
        .byte   2
.LCPI0_17:
        .byte   1

process_core:
        vpbroadcastb    zmm0, byte ptr [rip + .LCPI0_12]
        vpbroadcastb    zmm1, byte ptr [rip + .LCPI0_13]
        vpbroadcastb    zmm2, byte ptr [rip + .LCPI0_14]
        vpbroadcastb    zmm3, byte ptr [rip + .LCPI0_15]
        vpbroadcastb    zmm4, byte ptr [rip + .LCPI0_16]
        vpbroadcastb    zmm5, byte ptr [rip + .LCPI0_17]
.LBB0_1:
        vpandq  zmm6, zmm0, zmmword ptr [rdi + rax]
        vpternlogq      zmm6, zmm1, zmmword ptr [rsi + rax], 248
        vpternlogq      zmm6, zmm2, zmmword ptr [rdi + rax + 1], 248
        vpternlogq      zmm6, zmm3, zmmword ptr [rsi + rax + 2], 248
        vpternlogq      zmm6, zmm4, zmmword ptr [rdx + rax + 1], 248
        vpternlogq      zmm6, zmm5, zmmword ptr [rdx + rax], 248
        vmovdqu64       zmmword ptr [rcx + rax], zmm6
        add     rax, 64
        cmp     rax, 9984
        jne     .LBB0_1
        ... ; Non-vectorized logic for the remaining cells (lt 64)
        ret
```

This implementation is correct, but it does not handle collisions yet. The trivial way of adding the collision handling we defined above breaks auto-vectorization. We will need to find a different way to implement collisions. It is probably impossible to auto-vectorize the collision handling using true randomness; it has much conditional code that is difficult to vectorize. We tried a few different implementations, like using a simple match statement without randomness (Listing 13), a static lookup table (Listing 14) and if-else statements (Listing 15), but they all broke auto-vectorization. However, we found that if we were using multiple if statements, the compiler could transform them into vectorized assembly. Listing 7 shows the implementation we ended up with. It needs to reassign `new_cell` sometimes because autovectorization breaks otherwise. We are not entirely sure why this is the case.

NOTE The lookup table solution for collisions might be possible when the CPU supports AVX512-VBMI, but we did not have access to such a CPU.

Listing 7. Non-random collision calculation using if statements

```rust
    // Three particle collisions
    if new_cell == 0b00101010 {
        *result = 0b00010101;
    }
    if new_cell == 0b00010101 {
        *result = 0b00101010;
    }
    new_cell = *result;

    // Two particle collisions
    if new_cell == 0b00100100 {
        *result = 0b00010010;
    }
    if new_cell == 0b00010010 {
        *result = 0b00001001;
    }
    if new_cell == 0b00001001  {
        *result = 0b00100100;
    }
    new_cell = *result;

    // Four particle collisions
    if new_cell == 0b00110110  {
        *result = 0b00011011;
    }
    if new_cell == 0b00101101 {
        *result = 0b00110110;
    }
    if new_cell == 0b00011011 {
        *result = 0b00101101;
    }
```

If we add that logic to the process_core function from above, the generated

The assembly for the `process_core` function with the collision implementation from Listing 7 is shown in Listing 8

Listing 8. Assembly for `process_core` with non-random collisions

```x86asm
.LCPI0_28:
        .byte   16
.LCPI0_29:
        .byte   32
.LCPI0_30:
        .byte   8
```

```
.LCPI0_31:
        .byte   4
.LCPI0_32:
        .byte   2
.LCPI0_33:
        .byte   1
.LCPI0_34:
        .byte   42
.LCPI0_35:
        .byte   21
.LCPI0_36:
        .byte   36
.LCPI0_37:
        .byte   18
.LCPI0_38:
        .byte   9
.LCPI0_39:
        .byte   54
.LCPI0_40:
        .byte   27
.LCPI0_41:
        .byte   45

process_core:
        mov     r11, rcx
        xor     eax, eax
        ; Masks for each direction
        vpbroadcastb    zmm0, byte ptr [rip + .LCPI0_28]
        vpbroadcastb    zmm1, byte ptr [rip + .LCPI0_29]
        vpbroadcastb    zmm2, byte ptr [rip + .LCPI0_30]
        vpbroadcastb    zmm3, byte ptr [rip + .LCPI0_31]
        vpbroadcastb    zmm4, byte ptr [rip + .LCPI0_32]
        vpbroadcastb    zmm5, byte ptr [rip + .LCPI0_33]
        ; Masks for each collision case
        vpbroadcastb    zmm6, byte ptr [rip + .LCPI0_34]
        vpbroadcastb    zmm7, byte ptr [rip + .LCPI0_35]
        vpbroadcastb    zmm8, byte ptr [rip + .LCPI0_36]
        vpbroadcastb    zmm9, byte ptr [rip + .LCPI0_37]
        vpbroadcastb    zmm10, byte ptr [rip + .LCPI0_38]
        vpbroadcastb    zmm11, byte ptr [rip + .LCPI0_39]
        vpbroadcastb    zmm12, byte ptr [rip + .LCPI0_40]
        vpbroadcastb    zmm13, byte ptr [rip + .LCPI0_41]
.LBB0_1:
        vpandq  zmm14, zmm0, zmmword ptr [rdi + rax]
        vpternlogq      zmm14, zmm1, zmmword ptr [rsi + rax], 248
        vpternlogq      zmm14, zmm2, zmmword ptr [rdi + rax + 1], 248
        vpternlogq      zmm14, zmm3, zmmword ptr [rsi + rax + 2], 248
        vpternlogq      zmm14, zmm4, zmmword ptr [rdx + rax + 1], 248
```

```
        vpternlogq      zmm14, zmm5, zmmword ptr [rdx + rax], 248
        ; Process collisions
        vpcmpeqb        k1, zmm14, zmm6
        vpcmpeqb        k2, zmm14, zmm7
        vmovdqu8        zmm14 {k1}, zmm7
        vmovdqu8        zmm14 {k2}, zmm6
        vpcmpeqb        k1, zmm14, zmm8
        vpblendmb       zmm15 {k1}, zmm14, zmm9
        vpcmpeqb        k1, zmm14, zmm9
        vmovdqu8        zmm15 {k1}, zmm10
        vpcmpeqb        k1, zmm14, zmm10
        vmovdqu8        zmm15 {k1}, zmm8
        vpcmpeqb        k1, zmm15, zmm11
        vpblendmb       zmm14 {k1}, zmm15, zmm12
        vpcmpeqb        k1, zmm15, zmm13
        vmovdqu8        zmm14 {k1}, zmm11
        vpcmpeqb        k1, zmm15, zmm12
        vmovdqu8        zmm14 {k1}, zmm13
        vmovdqu64       zmmword ptr [r11 + rax], zmm14
        add     rax, 64
        cmp     rax, 9984
        jne     .LBB0_1
        ... ; Non-vectorized logic for the remaining cells (lt 64)
        ret
```

This implementation always turns to the right in collisions. Implementing true randomness without sacrificing performance is not trivial. As a workaround, we adjusted our algorithm to always turn in the same direction, except when there is a particle, then we turn left. This way, two colliding particle streams will scatter, alternating between turning left and right. This behavior is not perfect, but it should be good enough for our purposes. We will revisit this when we have implemented it better and may be able to improve it then. The final implementation of `process_core` is shown in Listing 9.

Listing 9. Function for processing the core of a row with fake collisions

RUST

```rust
pub fn process_core<const WIDTH: usize>(
    above: &[u8; WIDTH - 1],
    current: &[u8; WIDTH],
    below: &[u8; WIDTH - 1],
    result: &mut [u8; WIDTH - 2]
) {
    let context_iterator = above
        .array_windows::<2>()
        .zip(current.array_windows::<3>())
        .zip(below.array_windows::<2>())
        .zip(result.iter_mut());

    context_iterator.for_each(
        |(
            (([north_west, north_east], [west, _current, east]), [south_west, south_e
ast]),
```

```
        result,
    )| {
        let new_cell = (west & 0b00100000)
            | (north_west & 0b00010000)
            | (north_east & 0b00001000)
            | (east & 0b00000100)
            | (south_east & 0b00000010)
            | (south_west & 0b00000001);
        *result = new_cell;

        if (new_cell == 0b00100100) && (south_east & 0b00010000 == 0) {
            *result = 0b00010010;
        }
        if (new_cell == 0b00100100) && (south_east & 0b00010000 != 0) {
            *result = 0b00001001;
        }
        if (new_cell == 0b00011011) && (south_east & 0b00010000 == 0) {
            *result = 0b00101101;
        }
        if (new_cell == 0b00011011) && (south_east & 0b00010000 != 0) {
            *result = 0b00110110;
        }

        if new_cell == 0b00010010 && (east & 0b00001000 == 0) {
            *result = 0b00001001;
        }
        if new_cell == 0b00010010 && (east & 0b00001000 != 0) {
            *result = 0b00100100;
        }
        if new_cell == 0b00101101 && (east & 0b00001000 == 0) {
            *result = 0b00110110;
        }
        if new_cell == 0b00101101 && (east & 0b00001000 != 0) {
            *result = 0b00011011;
        }

        if new_cell == 0b00001001 && (north_east & 0b00000100 == 0) {
            *result = 0b00100100;
        }
        if new_cell == 0b00001001 && (north_east & 0b00000100 != 0) {
            *result = 0b00010010;
        }
        if new_cell == 0b00110110 && (north_east & 0b00000100 == 0) {
            *result = 0b00011011;
        }
        if new_cell == 0b00110110 && (north_east & 0b00000100 != 0) {
            *result = 0b00101101;
        }
```

```
64
65              if new_cell == 0b00101010 {
66                  *result = 0b00010101;
67              }
68              if new_cell == 0b00010101 {
69                  *result = 0b00101010;
70              }
71              new_cell = *result; // This line does nothing, but auto-
  vectorization breaks without it
72          },
73      )
74  }
```

For collisions on the borders of the grid, we use a less optimized implementation which is just calling the `proccess_collision` function (Listing 3) for each cell. Only a small portion of the cells will be on the border of the grid, so this should not have a big impact on performance.

We measured the average duration for one cell for a 2000x2000 grid with 5000 rounds and got the following results:

- With real random collisions: 3.76415ns per cell
- With AVX fake random collisions: 0.22619ns per cell
- With AVX fake random collisions but with vectorization disabled: 2.50546ns per cell
- Without collisions: 0.06601ns per cell

The results show that our vectorized implementation has a 10x speedup over the non-vectorized implementation. Our test machine only supports 256-bit vector operations, so the results above only reflect that. We expect the performance to be better on machines that support AVX512.

MAIN LOOP

The core of the simulation is based around two grids, `grid_a` contains the current state, and `grid_b` contains the next state. We fill the top row, the core rows, and the bottom row of grid_b in separate functions. After `grid_b` is done, it is swapped with `grid_a`, and the process is repeated. The main loop is shown in Listing 10.

Listing 10. One step of the main loop

```rust
movement_top_row(&grid_a[0], &grid_a[1], &mut grid_b[0]);

grid_a
    .windows(3)
    .zip(grid_b.iter_mut().skip(1))
    .enumerate()
    .for_each(|(row_index, (context, result))| {
        let above = &context[0];
```

```
        let current = &context[1];
        let below = &context[2];
        if ((row_index + 1) % 2) == 0 {
            movement_even_row(above, current, below, result);
        } else {
            movement_odd_row(above, current, below, result);
        }
    });

movement_bottom_row(&grid_a[WIDTH-2], &grid_a[WIDTH-1], &mut grid_b[WIDTH-1]);

std::mem::swap(&mut grid_a, &mut grid_b);
```

SHARED MEMORY PARALLELISM WITH RAYON

We will use rayon to divide the processing of the core rows over multiple threads. `par_windows` can be used instead of `windows`, and `par_iter_mut` can be used instead of `iter_mut` to process the rows in parallel. The main loop with rayon is shown in Listing 11.

Listing 11. One round using rayon

RUST

```
movement_top_row(&grid_a[0], &grid_a[1], &mut grid_b[0]);

grid_a
    .par_windows(3)
    .zip(grid_b.par_iter_mut().skip(1))
    .enumerate()
    .for_each(|(row_index, (context, result))| {
        let above = &context[0];
        let current = &context[1];
        let below = &context[2];
        if ((row_index + 1) % 2) == 0 {
            movement_even_row(above, current, below, result);
        } else {
            movement_odd_row(above, current, below, result);
        }
    });

movement_bottom_row(&grid_a[WIDTH-2], &grid_a[WIDTH-1], &mut grid_b[WIDTH-1]);

std::mem::swap(&mut grid_a, &mut grid_b);
```

Multithreading measurements

- Without rayon: 0.21101ns per cell
- With 1 thread: 0.21776ns per cell
- With 2 threads: 0.11265ns per cell
- With 4 threads: 0.060394ns per cell

- With 6 threads: 0.046102ns per cell
- With 7 threads: 0.042037ns per cell
- With 8 threads: 0.040239ns per cell
- With 9 threads: 0.040608ns per cell
- With 10 threads: 0.040653ns per cell
- With 10 threads: 0.038776ns per cell
- With 16 threads: 0.038706ns per cell
- With 20 threads: 0.050061ns per cell

We made measurements for a 2000x2000 grid with 5000 rounds. An intel i9-11950H was used for testing. Based on the Willow Cove microarchitecture, it has eight cpus and 16 cores. The measurements show that the performance is best with eight cores. I am not sure why this is the case. I would like to think that a single core keeps the shared vector units on its CPU busy all the time because there are nearly no branches, and most instructions in the main loop are vector instructions. However, it may as well be that there is a memory bottleneck.

Afterward, we discovered that our processor actually supports the required AVX512 instructions. We redid the eight threads measurement and got a performance of 0.034543ns per cell.

DISTRIBUTED MEMORY PARALLELISM WITH MPI

Our project uses MPI via the rsmpi Rust bindings for distributed memory parallelism. Every MPI node will process a consecutive chunk of rows. For example, if we have two nodes and eight rows, node 0 will process rows 0-3, and node 1 will process rows 4-7. Our implementation is shown in Listing 12. Before each round, every node sends its top row to the previous node and its bottom row to the next node. This way, all nodes that are not at the top or bottom have the context to treat their top/bottom rows like core rows. The first and last nodes still process their rows like core rows.

Listing 12. One round using MPI and rayon

```rust
mpi::request::scope(|scope| {
    let mut guards = Vec::new();

    if let Some(previous_rank) = previous_rank {
        guards.push(WaitGuard::from(
            world
                .process_at_rank(previous_rank)
                .immediate_send(scope, unsafe {
                    std::mem::transmute::<&mut [Cell; WIDTH], &mut [u8; WIDTH]>(
                        &mut grid_a[0],
                    )
                }),
```

```
        ));
        guards.push(WaitGuard::from(
            world.process_at_rank(previous_rank).immediate_receive_into(
                scope,
                unsafe {
                    std::mem::transmute::<&mut [Cell; WIDTH], &mut [u8; WIDTH]>(
                        receive_top,
                    )
                },
            ),
        ));
    }

    if let Some(next_rank) = next_rank {
        guards.push(WaitGuard::from(
            world
                .process_at_rank(next_rank)
                .immediate_send(scope, unsafe {
                    std::mem::transmute::<&mut [Cell; WIDTH], &mut [u8; WIDTH]>(
                        &mut grid_a[height - 1],
                    )
                }),
        ));
        guards.push(WaitGuard::from(
            world
                .process_at_rank(next_rank)
                .immediate_receive_into(scope, unsafe {
                    std::mem::transmute::<&mut [Cell; WIDTH], &mut [u8; WIDTH]>(
                        receive_bottom,
                    )
                }),
        ));
    }
});


if previous_rank.is_some() {
    movement_even_row(receive_top, &grid_a[0], &grid_a[1], &mut grid_b[0]);
} else {
    movement_top_row(&grid_a[0], &grid_a[1], &mut grid_b[0]);
}

grid_a
    .par_windows(3)
    .zip(grid_b.par_iter_mut().skip(1))
    .enumerate()
    .for_each(|(row_index, (context, result))| {
        let above = &context[0];
```

```
61          let current = &context[1];
62          let below = &context[2];
63          if ((row_index + 1) % 2) == 0 {
64              movement_even_row(above, current, below, result);
65          } else {
66              movement_odd_row(above, current, below, result);
67          }
68      });
69
70  if next_rank.is_some() {
71      movement_odd_row(&grid_a[height-2], &grid_a[height-
    1], receive_bottom, &mut grid_b[height-1]);
72  } else {
73      movement_bottom_row(&grid_a[height-2], &grid_a[height-1], &mut grid_b[height-1]);
74  }
75
76  std::mem::swap(&mut grid_a, &mut grid_b);
```

Using this approach, we can scale our simulation to multiple nodes. We tested our implementation on the Virgo cluster.

MEASUREMENTS

The Virgo cluster has nodes with Intel Xeon Gold 6248 CPUs, which support AVX512. Each Node has two Sockets, each with 24 CPUs and 48 cores. The measurements always reserved a whole node because the measurements were really unpredictable sometimes when sharing a node with other tasks. We think that this is caused by some other job that does not use many cores but completely saturates the memory bandwidth of the processors, but we did not investigate this further. We thought using a whole node exclusively for measurements would solve this problem. It did not, but we ran enough measurements to get some good ones.

The main metric we will focus on during our measurements is the time per cell. This metric is calculated by dividing the total runtime by the total number of cells calculated (width * height * rounds). The runtime measurement starts after the grid is set up and ends after the last calculation is done, so it does not include memory allocation, program startup, or placing the initial cells into the grid. This metric is mostly independent of the actual test size and lets us compare different measurements with different grid sizes. It should be noted that the actual time per cell depends on the grid size because the processing of border cells is far slower than the processing of core cells.

If otherwise specified, measurements were done with a 10000x10000 cell grid over 1000 rounds. When starting a benchmark job, it tests if all nodes are working correctly and aborts if they are not.

COMPARING DIFFERENT CORE IMPLEMENTATIONS

We wanted to know if AVX512 (512bit vector operations) brings improvement over using AVX2 (256bit vector operations). We also wanted to know if our focus on vectorization at the cost of true randomness was bringing the expected performance improvements. The results are shown in Figure 3.

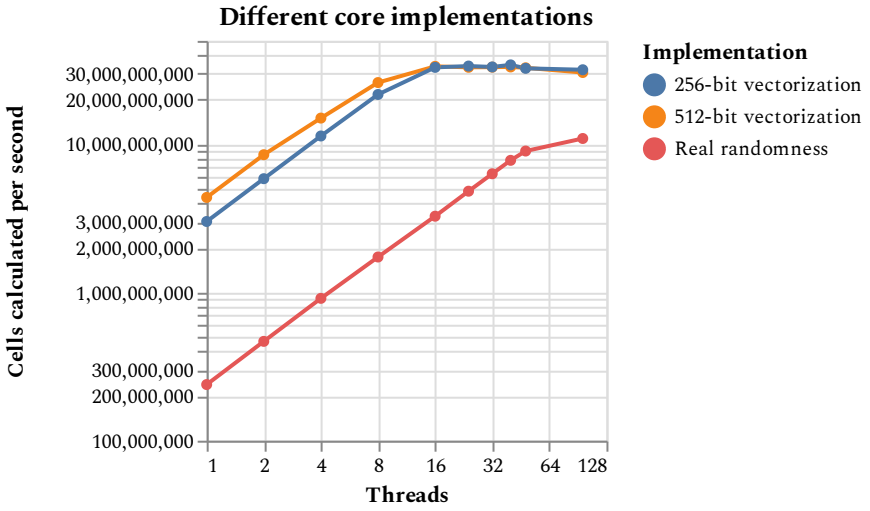

Figure 3. Comparison of different implementations for calculating core cells

The vectorized implementations process roughly ten times more cells per second than the real-random implementation. Of the vectorized implementations, the AVX512 implementation is about 30% faster than the AVX2 implementation. This is a bit less than we expected, but it is still a significant improvement. The non-vectorized implementation scales linearly with the number of threads, but from 48 to 96 threads, the performance only improves by 20%. This is because the node has only 48 CPUs. SMT is used when we use more threads and resources are shared between two cores on the same CPU.

Both vectorized implementations scale about linearly with the number of cores but hit a plateau when using more than 16 cores. With more cores, the speed stays at around 35 Gigacells per second. A memory throughput bottleneck probably causes this. As one cell is 1 byte in size, the CPU is loading and storing at 35 GB/s. The theoretical maximum memory bandwidth per CPU is 143.36 GB/s (6 channels of 2933 MHz memory). The node has two sockets, so the maximum bandwidth is around 281 GB/s. Our actual maximum is about an eighth of the theoretical maximum. This is probably because we did not optimize memory access patterns to keep the code simple. We could analyze the number of page faults to improve it and ensure we are aligned with the cache lines. We could also improve caching, as our cores are split over two sockets, so they are not sharing the L3 cache. We could use Slurm to make sure every cores is on the same socket and run two tasks per node

Because the AVX512 implementation is the fastest, the rest of the measurements will be done with that implementation.

We will also use 48 threads to avoid SMT. It should not make a big difference as we reach the memory bandwidth bottleneck at that number of threads anyway.

DIFFERENT GRID SIZES

We assume that the grid size significantly impacts the performance because bigger grids allow for more sequential processing of cells. We measured the performance for different grid sizes and got the results shown in Figure 4.

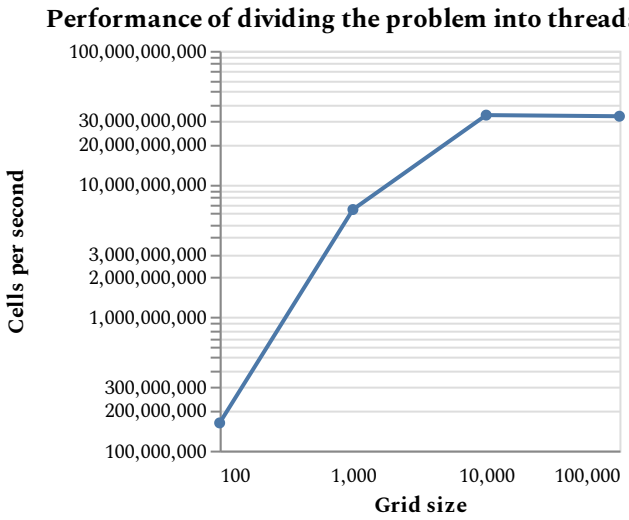

Figure 4. Different grid sizes compared to performance

For a small 100x100 grid, we process a lot fewer cells per second than for large grids. This is probably because the core of the row is so small that the overhead is a lot bigger than the actual work.

For bigger grids, the performance increases, but it stagnates after 10000x10000 cells. We assume that, at this point, the overhead is negligible compared to the time spent doing the vectorized calculations.

For the rest of the measurements, we will use a 10000x10000 grid.

THREADS VS TASKS

We want to find out how running multiple openMP tasks with a low number of threads compares to running a single openMP task with a high number of threads. The results for a single node are shown in Figure 5.

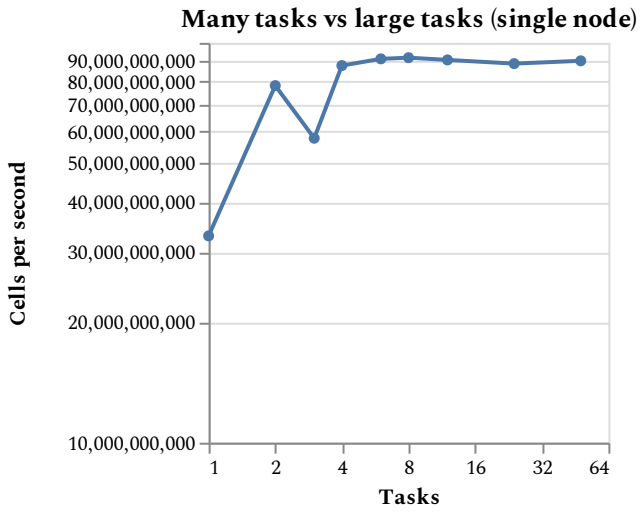

Figure 5. Threads compared to tasks for a single node

The chart does not show what we had expected initially. I would have thought that the performance would be the same for 1 task with 48 threads and 1 thread with 48 tasks. However, the performance is usually at 90 Gigacells per second, except for 3 tasks with 16 threads and 1 task with 48 threads. This could be caused by the scheduler having to split single tasks over the two available sockets for these two combinations. If a task is split among two sockets, it may have issues with caching. It seems like tasks with a large number of threads. It would also explain the split at 16 threads, as two tasks with 16 threads are not split and would run at 90, and one task is split over the two sockets and runs at 30. Taking the average (90 + 90 + 30) / 3 results in 70, which is close to the actual result of 60.

SCALING OVER MULTIPLE NODES

We want to see how big the performance impact of distributing over multiple nodes is. We measured the performance for 1, 2, and 4 nodes. Figure 6 shows the results for different numbers of tasks per node. When a node runs a low number of tasks, the tasks have more threads. In total, there are always 48 threads per node.

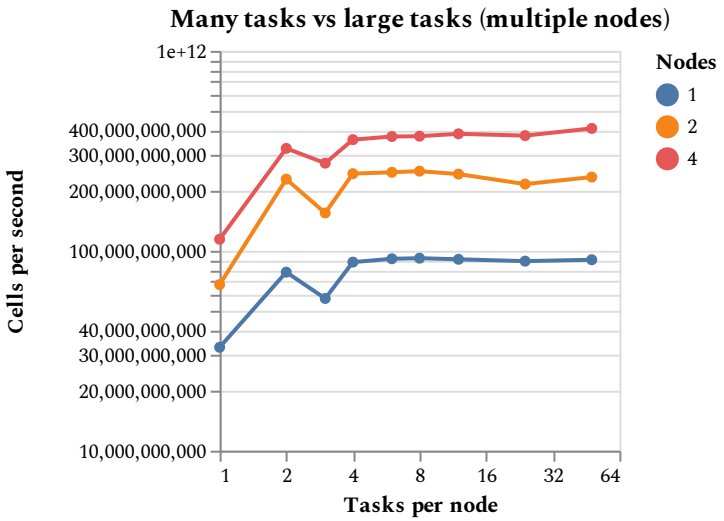

Figure 6. Performance in relation to the number of nodes

The chart shows that the performance roughly doubles when the number of nodes is doubled. The step from one to two nodes seems to be even more extreme, as it nearly triples in value. We suspect that we got a particularly good node for that measurement. Because we always use the whole node exclusively, we must wait a long time until our jobs are run. We did ten runs for the tests with one node and 5 runs with 2 and 4 nodes.

We analyzed the speedup and efficiency of distributing over multiple machines in Figure 7. The speedup is the performance gain compared to a single node. Efficiency is the performance gain in relation to the resources used.

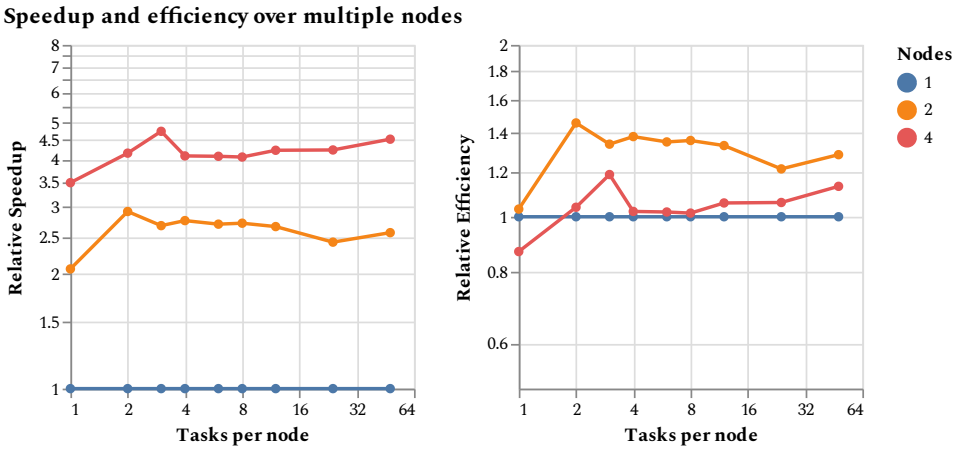

Figure 7. Speedup and efficiency

It is interesting to see that the efficiency for 2 and 4 nodes is actually higher than for 1 node. Again that is probably caused by the fact that we got particularly good nodes for the 2 measurements. For 4 nodes, the efficiency is lower than for 2 nodes, but it is still lower than for 1 node. If all nodes were

the same speed, we would expect the efficiency to be a tiny bit lower than one because of the communication overhead. However, as we are not transmitting a lot of data compared to the amount of work and only need to sync between rounds, it should not be much compared to running everything locally.

RESULTS

We were able to implement a fast and scalable simulation of the LGCA. We were able to scale the simulation over multiple nodes and got a near-linear speedup for 4 nodes. We were also able to scale the simulation over multiple threads, where we achieved linear speedup as well. Our fastest run (4 nodes, 48 tasks per node) was able to **process 492 Gigabytes per second** of data, which is nearly half the theoretical maximum set by the memory bandwidth of 1126GB/s (140GB/s * 2 sockets * 4 nodes).

LIST OF ABBREVIATIONS

LGCA
Lattice gas cellular automaton

AVX512
Advanced Vector Extensions 512 for x86

SMT
Simultaneous multithreading

SIMD
Single instruction, multiple data

APPENDIX

Listing 13. Collision calculation using match without randomness

```rust
pub fn process_collision(cell: u8) -> u8 {
    match cell {
        // Two opposing particles
        0b00001001 => 0b00100100,
        0b00010010 => 0b00001001,
        0b00100100 => 0b00010010,

        // Three particles
        0b00101010 => 0b00010101,
        0b00010101 => 0b00101010,

        // Four particles with opposing holes
        0b00110110 => 0b00011011,
        0b00011011 => 0b00101101,
        0b00101101 => 0b00011011,

        // Everything else
        x => x,
    }
}
```

Listing 14. Collision calculation using a lookup table

```rust
const COLLISION_TABLE: [u8; 64] = [
    0b0000_0000,
    0b0000_0001,
    0b0000_0010,
    0b0000_0011,
    0b0000_0100,
    0b0000_0101,
    0b0000_0110,
    0b0000_0111,
    0b0000_1000,
    0b0001_0010,
    0b0000_1010,
    0b0000_1011,
    0b0000_1100,
    0b0000_1101,
    0b0000_1110,
    0b0000_1111,
    0b0001_0000,
    0b0001_0001,
    0b0010_0010,
    0b0001_0011,
    0b0001_0100,
    0b0010_1010,
```

```
    0b0001_0110,
    0b0001_0111,
    0b0001_1000,
    0b0001_1001,
    0b0001_1010,
    0b0010_1101,
    0b0001_1100,
    0b0001_1101,
    0b0001_1110,
    0b0001_1111,
    0b0010_0000,
    0b0010_0001,
    0b0010_0010,
    0b0010_0011,
    0b0000_1001,
    0b0010_0101,
    0b0010_0110,
    0b0010_0111,
    0b0010_1000,
    0b0010_1001,
    0b0001_0101,
    0b0010_1011,
    0b0010_1100,
    0b0011_0110,
    0b0010_1110,
    0b0010_1111,
    0b0011_0000,
    0b0011_0001,
    0b0011_0010,
    0b0011_0011,
    0b0011_0100,
    0b0011_0101,
    0b0001_1011,
    0b0011_0111,
    0b0011_1000,
    0b0011_1001,
    0b0011_1010,
    0b0011_1011,
    0b0011_1100,
    0b0011_1101,
    0b0011_1110,
    0b0011_1111,
];

pub fn process_collision(cell: u8) -> u8 {
    COLLISION_TABLE[cell as usize]
}
```

Listing 15. Collision calculation using if-else statements

```rust
    if *result == 0b00101010 {
        *result = 0b00010101;
    } else if *result == 0b00010101 {
        *result = 0b00101010;
    }else if *result == 0b00100100 {
        *result = 0b00010010;
    }

    if *result == 0b00010010 {
        *result = 0b00001001;
    } else if *result == 0b00001001  {
        *result = 0b00100100;
    }

    if *result == 0b00110110  {
        *result = 0b00011011;
    } else if *result == 0b00101101 {
        *result = 0b00110110;
    } else if *result == 0b00011011 {
        *result = 0b00101101;
    }
```