Numerical Methods :
Rate equations or MC approaches for modelling growth
- Technical Report -

Léo BECHET, M2 CompuPhys 2024-2025

The follwing is a technical report on the implementation of the simulation used in Tasks 2 and 3. It discusses the single-threaded implementation as well as the modifications added to obtain a parallelizable system. Evolution and switch of algorithms are discussed, namely Depth-First Search (DFS) and Union-Find algorithms.

I. Single-Threaded Simulation

The current simulation spans up to task 3, with task 4 focusing on different objectives.

I.1 Implementation Overview

The simulation is modeled on a discretized grid of size $N \times N$N × N, where each cell can either be empty or occupied by a monomer. Monomers are represented by instances of the `Monomer` class, which holds the necessary information for each cell, including properties such as type and aggregation status.

I.2 Class Structure

I.2.a `Monomer` Class

The `Monomer` class defines the core object for monomers in the simulation. Each monomer has two primary attributes: `type` and `aggregated`. The type (either `"A"` or `"B"`) differentiates the monomers, and the `aggregated` property indicates whether a monomer has aggregated and thus is no longer allowed to move.

```
class Monomer:
    def __init__(self, type):
        self.type = type
        self.aggregated = False

    def aggregate(self):
        self.aggregated = True
```

When a monomer is marked as aggregated using the `aggregate()` method, it becomes fixed, meaning it will not participate in further movement. This also serves to assist in counting the islands formed during the simulation.

I.2.b `Simulation` Class

The `Simulation` class encapsulates the entire simulation process, including initialization and iteration through simulation steps.

```
class Simulation:
    def __init__(self, size, ndif_ratio, debug=False):
        self.debug = debug
        self.size = size
        self.ndif_ratio = ndif_ratio
        self.stepNo = 1

        # Initialize the grid as an array of objects
        self.grid = np.zeros(size, dtype=object)

        self.indices = []
        for offset in pass_offsets:
            i_offset, j_offset = offset
            for i in range(i_offset, size[0]):
                for j in range(j_offset, size[1]):
                    if (i % 3 == i_offset) and (j % 3 == j_offset):
                        self.indices.append((i, j))
```

- **`size`** : Tuple specifying the grid dimensions.
- **`ndif_ratio`** : Ratio governing the diffusion coefficients along different axes.
- **`debug`** : If set to `True`, enables debug printing for detailed information during simulation steps.
- **`stepNo`** : Tracks the current step of the simulation.
- **`grid`** : Initialized as an empty grid, where each cell will be filled with a monomer or left empty.
- **`indices`** : Initialized as the list of offsets used when updating the simulation.

I.3 Simulation Step: Movement and Aggregation

The `Step()` function is responsible for iterating through the grid and updating the position of monomers that have not yet aggregated.

```
self.stepNo += 1  # Increment step counter
for (i, j), cell in np.ndenumerate(self.grid):
    if isinstance(cell, Monomer) and not cell.aggregated and cell.lastMoved != self.
```

This loop iterates through each cell in the grid using `np.ndenumerate()`. The cell will only be processed if it contains a `Monomer`, the monomer is not aggregated, and it has not already moved during the current step. This ensures that no monomer moves more than once in a single step, a constraint that will later aid in parallelization.

I.3.a Movement Logic

Movement is determined by selecting a random axis (`"i"` for rows or `"j"` for columns) and a direction (`-1` or `1`), which is controlled by the diffusion ratio `ndif_ratio`.

```
axis = np.random.choice(["i", "j"], p=[self.ndif_ratio, 1.0-self.ndif_ratio])
value = np.random.choice([-1, 1])
```

A `match` statement is then used to perform the actual movement of the monomer along the selected axis.

```
match axis:
    case "i":
        self.grid[i, j].moves -= 1
        self.grid[i, j].lastMoved = self.stepNo
        self.grid[i + value, j], self.grid[i, j] = self.grid[i, j], self.grid[i + va
        new_i, new_j = i + value, j

    case "j":
        self.grid[i, j].moves -= 1
        self.grid[i, j].lastMoved = self.stepNo
        self.grid[i, j + value], self.grid[i, j] = self.grid[i, j], self.grid[i, j +
        new_i, new_j = i, j + value
```

Depending on whether the chosen axis is `"i"` or `"j"`, the monomer will move up/down or left/right, respectively. The `moves` attribute ensures the movement count is reduced accordingly, and the monomer's `lastMoved` property is updated to the current step number.

Note that the current simulation implementation simply does nothing if a cell tries to leave, meaning it will just stay in place.

I.3.b Aggregation

After the move, the algorithm checks if the neighboring cells (up, down, left, and right) contain monomers. If they do, both the current monomer and the neighbor are marked as aggregated.

```
for di, dj in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
    try:
        if isinstance(self.grid[new_i + di, new_j + dj], Monomer):
            self.grid[new_i, new_j].aggregate()
            self.grid[new_i + di, new_j + dj].aggregate()
    except IndexError:
        pass  # Handle out-of-bounds cells
```

This aggregation check is done in all four cardinal directions, ensuring that monomers that come into contact aggregate, preventing further movement.

I.4 Deposition

The `deposition` function randomly places monomers within a specified area on the grid. If no specific center or side lengths are provided, the function defaults to using the entire grid.

```
if x_center is None:
    x_center = self.grid.shape[0] / 2
if y_center is None:
    y_center = self.grid.shape[1] / 2

# Set default side lengths if not provided
if x_side is None:
    x_side = self.grid.shape[0]
if y_side is None:
    y_side = self.grid.shape[1]

# Choose random position within the specified area
i = int(np.random.random() * x_side + x_center - x_side / 2)
j = int(np.random.random() * y_side + y_center - y_side / 2)

# Place a monomer of the specified type
self.grid[i, j] = Monomer(type)
```

The `deposition` method selects random grid coordinates based on the provided `x_center`, `y_center`, `x_side`, and `y_side` values, and places a `Monomer` at the chosen position.

II. Multi-Threaded Simulation

II.0.a Multi-threaded vs Single-threaded Simulation

The transition from a single-threaded to a multi-threaded simulation introduces several new challenges, particularly in terms of **thread safety** and **data integrity**. In a single-threaded simulation, updating each cell is a sequential process, ensuring that only one cell is modified at any given time. However, in a multi-threaded simulation, where different parts of the grid may be processed concurrently, ensuring that two or more threads do not update the same or nearby cells simultaneously is crucial.

In the single-threaded approach, the simulation iterates over each cell using a double loop. Each cell is only updated if it is not aggregated and has not been moved during the current step. The axis and direction for movement are chosen randomly using probabilities skewed by the diffusion ratio (`ndif_ratio`), and neighboring cells are checked to determine whether aggregation occurs.

However, when optimizing with Numba, certain Python objects are not supported, which necessitates a workaround in representing the simulation's state. Instead of using objects to denote the different states of the cells, we utilize a numerical encoding scheme: `0` represents empty cells, `1`, `2`, and `3` represent the non-aggregated states of objects A, B, and C respectively, while `11`, `12`, and `13` are used to denote the aggregated states of these objects. This encoding allows for efficient computation while maintaining clarity in representing the different states of the grid cells.

II.0.a.i Difficulties Encountered in Multi-threading

Multi-threading, by design, aims to divide computational tasks across multiple cores, but this can lead to several problems:

- **Concurrent updates**: If two threads attempt to update neighboring cells simultaneously, it can lead to race conditions, resulting in inconsistent states.
- **Thread safety**: Ensuring that a cell is not updated more than once by different threads during the same pass is crucial.

The solution adopted for these problems was to divide the grid into separate **update zones**, ensuring that cells being updated in one pass do not interfere with each other. Specifically, the grid was divided into a **3x3 update scheme**.

II.0.b 3x3 Update Grid: Nine-Pass Strategy

To ensure thread safety, the grid is divided into smaller groups that are updated in **nine distinct passes**. Each pass targets a specific subset of cells, ensuring that no neighboring cells are updated during the same pass. Here's the reasoning behind the 3x3 division:

- **Neighbor checking**: Each cell checks its neighbors (left, right, above, and below) to determine if aggregation should occur. This means that an update requires a look at a **radius of 1** from the cell.
- To prevent any two cells within this radius from being updated in the same pass, a **3x3 grid** division was adopted, where each pass updates cells that are 2 cells apart, ensuring thread safety.

The diagram below illustrates how the 3x3 grid strategy divides the updates:

```
Grid layout:
+---+---+---+
| A | B | C |
+---+---+---+
| D | E | F |
+---+---+---+
| G | H | I |
+---+---+---+

- Pass 1 updates cells in A (Top-left)
- Pass 2 updates cells in B (Top-center)
- Pass 3 updates cells in C (Top-right)
- Pass 4 updates cells in D (Center-left)
- Pass 5 updates cells in E (Center)
- Pass 6 updates cells in F (Center-right)
- Pass 7 updates cells in G (Bottom-left)
- Pass 8 updates cells in H (Bottom-center)
- Pass 9 updates cells in I (Bottom-right)
```

Each pass only updates cells located at a particular offset within the grid, ensuring that no neighboring cells are updated at the same time. This approach prevents conflicts between threads when updating cells in parallel.

II.0.c 2x3 Division: Optimizing the Passes

While the 3x3 grid ensures thread safety by maintaining a sufficient distance between concurrently updated cells, we believe it's possible to optimize further by reducing the number of passes to **6 passes** using a **2x3 division**. This would involve strategically interlacing the update zones while maintaining the necessary spacing between the cells being updated simultaneously. This reduction in passes could lead to performance improvements, as it would lessen the overhead associated with managing 9 passes.

II.0.c.i 6-Pass Grid Layout (Hypothetical Optimization)

The idea is to shift rows by offsets, which could potentially reduce the number of passes from 9 to 6. This would still respect the necessary distance between updated cells to avoid conflicts, improving computational efficiency.

II.0.c.ii 5-Pass Theory

In theory, it's possible to further reduce the number of passes to **5 passes**. This would rely on a more advanced approach that takes into account the fact that each cell update involves **5 cells** (the cell itself and its four neighboring cells). If we can find a pattern where these updates don't overlap, this would result in even faster execution. However, this remains speculative as we haven't yet identified such a pattern.

When increasing the number of passes, fewer cells are updated simultaneously, which could balance the load across the processing cores. This can be beneficial in scenarios with larger simulations, where the number of active threads could be lower. In systems with a high number of cores, such as GPUs, this means that increasing the grid size might not increase the overall computation time.

II.0.d Axis and Direction Selection: Enhancing Readability

The choice of **axis** (whether the monomer moves along the x or y-axis) and **direction** (positive or negative) is done using random selection, with a probability weighted by the `ndif_ratio`. The new code separates axis selection from the move direction, improving readability. The use of random choices for both axis and direction has been simplified:

```
# Random axis selection using np.random.rand()
if np.random.rand() < ndif_ratio:  # 0 for x-axis
    axis = 0
else:
    axis = 1

# Random direction
value = 1 if np.random.rand() > 0.5 else -1
```

This structure allows for easier understanding and maintenance of the code, compared to more complex in-line random operations.

II.0.e Precomputed Indices and Parallelization

In the current implementation, the indices for each pass are precomputed before the simulation step begins. This ensures that the parallel loop can efficiently iterate over the relevant cells without needing to recalculate indices on every iteration.

The use of `prange` allows us to loop over these precomputed indices in parallel:

```
# Parallel loop over the precomputed indices
for index in prange(len(indices)):
    i, j = indices[index]
    ...
```

Although we could **cache** the precomputed indices across multiple steps to improve performance, this was avoided for the sake of code clarity and maintainability. For small to medium simulations, the overhead of precomputing the indices is negligible compared to the overall simulation time.

II.0.f Conclusion

The multi-threaded implementation leverages Numba's `@njit(parallel=True)` decorator to achieve parallelism across grid cells. By adopting a 3x3 grid division, thread safety is ensured, preventing race conditions when updating neighboring cells. Future optimizations, such as reducing the number of passes, offer promising directions for further improving the performance of the simulation.

II.1 Side functions

This section describes functions which are not part of the simulation itself but help in gathering information about it.

II.1.a Counting free Monomers function `NumMonomers`

```
def NumMonomers(self):
    # Clone of the grid, could use true/false instead of strings but oh well
    grid = [
                [0 if isinstance(x, str) or x == 0 or getattr(x, 'aggregated', False
                for row in self.grid
            ]
    return np.sum(grid)
```

We check the number of free monomers by first recreating the grid and replacing non-aggregated `Monomers` by 11 and other cells with 00. A simple sum is done, which returns the number of free monomers in the simulation.

II.1.b Counting Free Monomers *in Parallel*: `num_monomers_parallel`

```
@njit(parallel=True)
def num_monomers_parallel(grid):
    count = 0
    for r in prange(grid.shape[0]):
        for cell in grid[r]:
            if cell in (1, 2):  # Count only non-aggregated monomers
                count += 1
    return count
```

In the parallel version of the `NumMonomers` function, the grid is processed concurrently using Numba's `prange` to parallelize the outer loop. Instead of reconstructing a grid and replacing non-aggregated `Monomers` with integers (as done in the single-threaded approach), this version directly counts cells that are either `1` or `2`, which represent non-aggregated monomers. By leveraging parallelism, we can efficiently handle larger grid sizes and speed up the counting process.

In comparison, the single-threaded version involved a grid transformation, which adds a small overhead, whereas the multithreaded version directly inspects each cell of the original grid without extra steps. This parallel approach is particularly beneficial in large simulations where the workload can be distributed across multiple cores, significantly reducing computation time. However, as with any multithreaded approach, care must be taken to ensure correct handling of concurrent operations, although in this case, counting operations on non-aggregated cells remain thread-safe.

II.1.c `NumIslands` Function Explanation

The `NumIslands` function is designed to count the number of "islands" in a grid and return the number of islands along with the size of each one in terms of the number of cells. The function interprets a grid where cells are either land (`'1'`) or water (`'0'`). Here's a step-by-step explanation:

II.1.c.i 1. Grid Construction

The grid is cloned and transformed such that land cells are `'1'` and water cells are `'0'`. This transformation is applied based on the properties of the elements in `self.grid`, where a cell that is either a string, zero, or has no `aggregated` attribute is considered water (`'0'`). Otherwise, it is treated as land (`'1'`).

```
grid = [
    ['0' if isinstance(x, str) or x == 0 or not getattr(x, 'aggregated', False) else
    for row in self.grid
]
```

II.1.c.ii 2. DFS (Depth First Search) Setup

The DFS function, `dfs(r, c)`, is used to explore the grid from a specific cell. It works recursively to visit all connected land cells (cells adjacent up, down, left, or right). If the DFS encounters a water cell (`'0'`) or goes out of bounds, it returns `1`. Each visited land cell is marked as water (`'0'`) to prevent it from being visited again, essentially "erasing" the island as it is explored.

```
def dfs(r, c):
    # Boundary and check if it's land
    if r < 0 or c < 0 or r >= rows or c >= cols or grid[r][c] == '0':
        return 1

    # Mark the land as visited by setting it to '0'
    grid[r][c] = '0'

    s = 0  # Counter for cell number

    # Visit all adjacent cells (up, down, left, right)
    s += dfs(r + 1, c)  # down
    s += dfs(r - 1, c)  # up
    s += dfs(r, c + 1)  # right
    s += dfs(r, c - 1)  # left
    return s
```

II.1.c.iii 3. Main Loop

The grid is traversed row by row. Whenever an unvisited land cell (`'1'`) is found, a new island is discovered. The DFS is triggered, and its result, which indicates the size of the island, is added to the `cells_per_island` list. After the DFS finishes for an island, the `island_count` is incremented by one.

```
for r in range(rows):
    for c in range(cols):
        # Start a DFS if we find an unvisited land cell
        if grid[r][c] == '1':
            cells_per_island.append(dfs(r, c))
            island_count += 1  # Increase the island count after finishing the DFS
```

II.1.c.iv 4. Return

The function returns two values: the total number of islands (`island_count`) and a list (`cells_per_island`) containing the number of cells in each island.

```
return island_count, cells_per_island
```

II.1.d `num_islands_parallel` : Counting Islands in Parallel

```
@njit(parallel=True)
def num_islands_parallel(grid):
    rows, cols = grid.shape
    parent = np.arange(rows * cols)  # Each cell is its own parent
    size = np.ones(rows * cols, dtype=np.int32)  # Initialize sizes to 1
    rank = np.zeros(rows * cols, dtype=np.int32)

    # Iterate over the grid to perform union operations
    for r in prange(rows):
        for c in range(cols):
            if grid[r, c] == 1:
                if r + 1 < rows and grid[r + 1, c] == 1:
                    union(parent, size, rank, r * cols + c, (r + 1) * cols + c)
                if c + 1 < cols and grid[r, c + 1] == 1:
                    union(parent, size, rank, r * cols + c, r * cols + (c + 1))

    # Count distinct roots (islands) and their sizes
    root_set = set()
    island_sizes = {}

    for r in range(rows):
        for c in range(cols):
            if grid[r, c] == 1:
                root = find(parent, r * cols + c)
                root_set.add(root)
                if root in island_sizes:
                    island_sizes[root] += 1
                else:
                    island_sizes[root] = 1

    return len(root_set), [island_sizes[root] for root in root_set]  # Return number
```

In the parallelized version of the `NumIslands` function, the algorithm shifts from a depth-first search (DFS) approach to a **union-find (disjoint-set)** strategy for identifying and counting islands. This method leverages Numba's `prange` to parallelize the process across rows, making it more efficient for large grid sizes.

Instead of visiting neighboring land cells in a recursive manner, this algorithm unites connected land cells into the same "island" using the union-find structure. The grid is treated as a flattened 1D array, where each cell is initially its own parent. The union function merges connected land cells, and path compression is applied in the `find` function to optimize root identification. The parent-child relationships are adjusted as

the algorithm iterates, allowing cells in the same island to share a common root.

At the end of the process, the algorithm counts distinct root elements, which correspond to individual islands, and calculates the size of each island by summing the sizes of all cells connected to the same root. The result is returned as the total number of islands and a list of island sizes.

Compared to the DFS approach, the parallel version distributes the workload across multiple cores, reducing the time required to process large grids. This method is highly scalable, especially for grid-based simulations with significant data.

II.1.d.i `find` Function

```
@njit
def find(parent, x):
    while parent[x] != x:
        parent[x] = parent[parent[x]]  # Path compression
        x = parent[x]
    return x
```

The `find` function is responsible for determining the root representative of a given cell in the union-find structure. It employs **path compression** to optimize the search process, making subsequent queries faster by updating the parent pointers of nodes along the path to the root. This function ensures that all connected components can be efficiently accessed and identified.

II.1.d.ii `union` Function

```
@njit
def union(parent, size, rank, x, y):
    rootX = find(parent, x)
    rootY = find(parent, y)

    if rootX != rootY:
        if rank[rootX] > rank[rootY]:
            parent[rootY] = rootX
            size[rootX] += size[rootY]  # Update size of rootX
        elif rank[rootX] < rank[rootY]:
            parent[rootX] = rootY
            size[rootY] += size[rootX]  # Update size of rootY
        else:
            parent[rootY] = rootX
            size[rootX] += size[rootY]  # Update size of rootX
            rank[rootX] += 1
```

The `union` function merges two distinct sets represented by their root cells. It first identifies the roots of both cells and then links them based on their ranks, which helps keep the tree structure flat and efficient. The size of the combined islands is updated accordingly to reflect the total number of connected land cells.

These helper functions are integral to efficiently managing the relationships between cells and identifying connected components within the grid. Their optimized implementations significantly enhance the overall performance of the `num_islands_parallel` function.

III. Used Algorithms

III.1 Depth-First Search (DFS) Principle

Depth-First Search is a graph traversal algorithm that explores as far as possible along a branch before backtracking. It is particularly effective for problems like finding connected components in a grid. In this case, each land cell (`'1'`) can be seen as a node in a graph, and an edge exists between nodes that are directly adjacent (up, down, left, right).

When DFS starts from a land cell, it recursively visits all reachable land cells, marking them as visited. Once all the cells in the connected component (or island) are explored, DFS backtracks, and the algorithm continues searching for the next unvisited land cell.

III.1.a DFS Illustration

Let’s assume the following grid:

```
1 1 0 0 0
1 0 0 1 1
0 0 0 1 0
1 0 1 0 1
```

DFS starts at the first land cell (top-left corner):

```
▉ 1 0 0 0
▉ 0 0 1 1
0 0 0 1 0
1 0 1 0 1
```

It moves right and down to explore all connected land cells:

```
▉ ▉ 0 0 0
▉ 0 0 1 1
0 0 0 1 0
1 0 1 0 1
```

Once all reachable land cells are explored, it backtracks, marking the first island as fully explored. It will then start DFS from the next unvisited land cell to find the second island, and so on.

III.2 Union-Find Principle

Union-Find, also known as Disjoint Set Union (DSU), is a data structure that efficiently keeps track of a partition of a set into disjoint subsets. It supports two primary operations: **find** and **union**. These operations enable efficient queries and modifications of the connected components in a graph, making Union-Find particularly effective for problems involving connectivity, such as counting islands in a grid.

III.2.a Find Operation

The **find** operation determines the root representative of a particular element. It identifies which subset a particular element belongs to and can be optimized using **path compression**, which flattens the structure of the tree whenever `find` is called. This optimization reduces the time complexity of future queries by keeping the tree as flat as possible.

III.2.b Union Operation

The **union** operation merges two subsets into a single subset. It connects two elements and ensures that they share the same root. Union-Find employs a **rank** heuristic to keep the tree shallow by always attaching the smaller tree under the root of the larger tree. This approach minimizes the height of the trees, leading to more efficient operations.

III.2.c Union-Find Illustration

Consider the following scenario with a grid of land cells represented as follows:

```
1 1 0 0 0
1 0 0 1 1
0 0 0 1 0
1 0 1 0 1
```

Initially, each land cell is its own parent, represented by indices corresponding to their positions in the grid:

```
[0, 1, 2, 3, 4]
[5, 6, 7, 8, 9]
[10, 11, 12, 13, 14]
[15, 16, 17, 18, 19]
```

III.2.d First Iteration: Union Operations

1. **Processing Cell (0, 0)**:
   - Current cell value: `1` (land).
   - Union with its neighbor `(0, 1)` (also `1`).
   - Perform union: `union(parent, size, rank, 0, 1)`.
2. **Current Parent Structure After Union**:
   - Cell (0, 0) and Cell (0, 1) are now connected. We update the parent of cell `(0, 1)` to point to `(0, 0)`.

   Parents after first union:

   ```
   Parent:  [0, 0, 2, 3, 4]
            [5, 6, 7, 8, 9]
            [10, 11, 12, 13, 14]
            [15, 16, 17, 18, 19]
   ```

3. **Union with Cell (1, 0)**:
   - Current cell value: `1` (land).
   - Perform union: `union(parent, size, rank, 0, 5)` (connecting to the cell below).

   Parents after second union:

   ```
   Parent:  [0, 0, 2, 3, 4]
            [0, 6, 7, 8, 9]
            [10, 11, 12, 13, 14]
            [15, 16, 17, 18, 19]
   ```

III.2.e Result After First Iteration

At the end of the first iteration (processing the first row), the parent structure shows that cells `(0, 0)`, `(0, 1)`, and `(1, 0)` are all connected through their common root `(0, 0)`. The grid is evolving, forming one connected component (island) in the upper left part of the grid. As we continue iterating through the rest of the grid, further union operations will merge additional components.

The Union-Find structure allows us to quickly identify the roots and connected components in the grid, facilitating efficient counting of distinct islands.

In summary, Union-Find is a powerful algorithm for managing dynamic connectivity and efficiently

handling problems that require the identification of connected components in a graph or grid.

III.2.f Parallelization of Union-Find

Parallelizing the Union-Find algorithm can significantly enhance performance, especially for large datasets where connectivity checks and unions are frequent. The main challenge in parallelizing Union-Find lies in the need to maintain consistency in the data structure while multiple threads perform operations simultaneously. Here's how the method can be parallelized effectively:

III.2.f.i Parallel Union-Find Strategy

1. **Initial Setup**:
   - Each thread can independently initialize its own local Union-Find structure to represent the grid, where each land cell is its own parent. This initialization can occur in parallel without any conflicts since it involves writing to separate memory locations.
2. **Concurrent Union Operations**:
   - While processing the grid, each thread can perform union operations for the cells it processes in parallel. For example, if a thread encounters a land cell that connects to its neighboring land cells, it can execute the union operation independently.
3. **Handling Shared Data**:
   - To avoid race conditions when multiple threads perform union operations on the same cells, we can use atomic operations or locks:
     - **Atomic Operations**: Use atomic functions for the union to ensure that only one thread can modify the parent array at a time for a given cell.
     - **Locking**: Use mutex locks to protect the union operation. However, excessive locking can lead to contention and reduce the benefits of parallelization.
4. **Path Compression and Rank Management**:
   - Path compression and rank management should still be performed, but with care to ensure that they do not introduce inconsistencies:
     - Path compression can be performed after the union operations are complete, or threads can locally compress paths and update the parent array at the end of their operations to reduce conflicts.
     - Ranks can also be managed with care to ensure that merges are consistent across threads.

By carefully managing the union operations and using parallel processing features provided by libraries like Numba, Union-Find can be effectively parallelized. This parallelization can lead to significant performance gains, particularly when processing large grids with many land cells. However, developers need to be cautious about data consistency and thread safety to avoid potential pitfalls associated with concurrent data access.

IV. Known Bugs and Limitations

IV.1 Fill ration error

There is an inconsistency in the DFS implementation, where the number of cells per islands seems to be wrong. Thus we recommend using the parallelized version of the code.

IV.2 Deposition spots

Deposition doesn't check whether a monomer is already present somewhere. This could potentially lead to issues, however we haven't change the implementation. One way to do it would be to create a list of all available positions in th grid and randomly pull one.