Simon König, 11702826
Viktor Beck, 11713110
Michael Ketter, 11701297

184.725 High Performance Computing

TECHNISCHE UNIVERSITÄT WIEN

HPC Project: Game of Life Stencil

Contents

1 Introduction

The Game of Life is a simple time-stepping algorithm, which updates a binary 2D matrix according to predefined rules. Those rules usually concern only the value of the updated cell and its direct neighbors, making the algorithm local per definition. Due to this locality, the algorithm is trivially parallelizable by splitting the 2D domain into submatrices. Each submatrix is assigned to an available processor, which solely performs the updates of its domain. Between the update steps, each processor has to communicate its border values to neighboring processors. However, this communication step creates a bottleneck for parallel speed-up with increasing processor numbers.

The aim of this project is to implement one sequential and multiple parallel versions (utilizing the features in OpenMPI) of the game of life. To assess the versions performance, the code should be benchmarked in a weak and strong scaling parallel set-up. Findings, explanations and possible improvements should be stated and discussed in the following.

2 Project layout

The core functionality of the project is given in the files `solver.hpp` and `solverATA.hpp`. Each contains a stencil solver that differs from the other one solely in the chosen method of inter-processor communication.

For storing the 2D matrices a self-designed `C++`-class called `MatrixView` is used. A short introduction and the usage of the class in the parallel game of life is illustrated in the following section.

2.1 Code Structure and Organization

MatrixView Class

Listing 1: MatrixView Class

```
template <typename Type>
class MatrixView {
private:
  std::vector<Type> &v;
  // ... (constructor and other member functions)
public:
  const int N, M;
  MatrixView(std::vector<Type> &v, int N, int M) : v(v), N(N), M(M) {
    assert(static_cast<int>(v.size()) / N == M);
  }
  // ... (set, get, and other member functions)
};
```

Description: The MatrixView class provides a convenient interface for accessing and manipulating a 2D matrix stored in a 1D vector (`std::vector`). It abstracts the underlying vector and simplifies operations on the matrix. That is one can access Matrix elements in typical manner i.e. M[i,j] = Matrixview(i,j).

2.1.1 Domain Setup

Listing 2: Domain Setup

```
int NX, NY;
std::vector<int> domain(NX * NY, Cell::UNKNOWN);
MatrixView<int> domainView(domain, NX, NY);
```

Description: The domain vector represents the entire domain of the Game of Life, initialized with `Cell::UNKNOWN` values. `MatrixView<int>` is then used to create a view of the matrix to facilitate easy access and manipulation.

2.1.2 Setting Up Boundary Conditions

Listing 3: Setting Up Boundary Conditions

```
for (int i = 1; i != NX - 1; ++i) {
  domainView.set(i, 0) = Cell::NORTH;
  domainView.set(i, NY - 1) = Cell::SOUTH;
}
for (int j = 1; j != NY - 1; ++j) {
  domainView.set(NX - 1, j) = Cell::EAST;
  domainView.set(0, j) = Cell::WEST;
}
domainView.set(0, 0) = Cell::NORTHWEST;
domainView.set(NX - 1, 0) = Cell::NORTHEAST;
domainView.set(0, NY - 1) = Cell::SOUTHWEST;
domainView.set(NX - 1, NY - 1) = Cell::SOUTHEAST;
```

Description: This part of the code sets up boundary conditions for the domain. The values of `Cell::NORTH`, `Cell::SOUTH`, `Cell::EAST`, `Cell::WEST`, `Cell::NORTHWEST`, `Cell::NORTHEAST`, `Cell::SOUTHWEST`, and `Cell::SOUTHEAST` are assigned to the corresponding positions in the matrix. To ensure a 9 point stencil, 8 different data packages are sent and received by each processor in each iteration of the game.

Solution Initialization

Listing 4: Solution Initialization

```
std::vector<int> solution(NX * NY, 0);
MatrixView<int> solutionView(solution, NX, NY);
```

Description: The solution vector represents the initial state of the Game of Life. It is initialized with zeros, and `MatrixView<int>` is used to create a view for easy manipulation.

2.1.3 Random Initialization Using `get_random_value`

Listing 5: Random Initialization Using `get_random_value`

```
for (int j = 1; j != NY - 1; ++j) {
  for (int i = 1; i != NX - 1; ++i) {
    solutionView.set(i, j) = get_random_value(m_offset_r + i, m_offset_c + j, 2, 10);
  }
}
```

Description: This part of the code initializes the interior cells of the solution matrix with random values using the `get_random_value` function.

2.1.4 Solution Swap Mechanism

Listing 6: Solution Swap Mechanism

```
std::vector<int> solution2 = solution;
// ...
game(solution, solution2, NX, NY);
```

Description: `solution2` is a separate vector used as a temporary buffer for the next iteration. Before each iteration of the Game of Life (`game` function), the content of `solution2` is swapped with `solution`. This mechanism avoids the need for creating a new vector in each iteration, improving memory efficiency.

2.1.5 Game of Life Logic

Listing 7: Game of Life Logic

```
auto game = [](std::vector<int> &sol, std::vector<int> &sol2, int NX, int NY) {
  // ... (Playing the game of life on subdomain.)
  sol.swap(sol2);
};
```

Description: The `game` function takes two vectors (`sol` and `sol2`) as input and applies the rules of the Game of Life. After the update, it swaps the content of `sol` and `sol2` using `sol.swap(sol2)`, ensuring that `sol2` now contains the updated state for the next iteration.

3 Point-to-Point Communication Algorithm

3.1 Cartesian Grid Representation

Choice: A Cartesian grid is used to represent the domain.
Reasoning: This grid structure simplifies the identification of neighboring processes and facilitates communication in a structured manner.

3.2 Neighboring Ranks and Diagonal Coordinates

The provided code segment determines the ranks of neighboring cells in a Cartesian grid using MPI functions. Additionally, it calculates the ranks of cells in the diagonal directions (North East, South East, North West, and South West).

Listing 8: rank setup

```
// Get ranks of neighboring cells in the cartesian grid
MPI_Cart_shift(GRID_COMM, 0, 1, &east_rank, &west_rank);
MPI_Cart_shift(GRID_COMM, 1, 1, &south_rank, &north_rank);

MPI_Cart_coords(GRID_COMM, myrank, 2, own_coords);

  // North East Diagonal
  diag_coords[0] = (own_coords[0] - 1 + dims[0]) % dims[0];
  diag_coords[1] = (own_coords[1] + 1) % dims[1];
```

```
10  MPI_Cart_rank(GRID_COMM, diag_coords, &north_east_rank); // Norht east rank now holds the rank
      processor of the processor north east
```

Description: Every rank gets its neighboring cells in the Cartesian grid. The other diagonal elements are setup alike.
The `MPI_Sendrecv` function in MPI is a combined operation that sends a message to a destination process and simultaneously receives a message from another source. This function is particularly useful when communication involves a pair of processes that need to exchange data.

3.3 Communication: SendRecv()

Listing 9: communication

```
1 MPI_Sendrecv(NORTH_SEND.data(), NX, MPI_INT, north_rank, 1,
2              SOUTH_RECV.data(), NX, MPI_INT, south_rank, 1, GRID_COMM, MPI_STATUS_IGNORE);
3
4  // Northeast Send, Southwest Recv
5     MPI_Sendrecv(&NORTHEAST_SEND, 1, MPI_INT, north_east_rank, 5,
6                  &SOUTHWEST_RECV, 1, MPI_INT, south_west_rank, 5, GRID_COMM, MPI_STATUS_IGNORE);
```

Description: This line sends data (`NORTH_SEND`) to the process in the north (`north_rank`) and simultaneously receives data (`SOUTH_RECV`) from the process in the south (`south_rank`). Communication in other directions works alike.
Parameters:

- `NORTH_SEND.data()`,`NORTHEAST_SEND`: Pointer to the data array to be sent to the north/north-east.
- `NX/1`: Number of elements to be sent in one direction (width of the subdomain).
- `MPI_INT`: Data type of the elements being sent (integer in this case).
- `north_rank`: Rank of the destination process in the north.
- `1/5`: Tag for the sent message (used to distinguish different types of messages).
- `SOUTH_RECV.data()/SOUTH_WEST`: Pointer to the data array to store the received data from the south/southwest.
- `south_rank`: Rank of the source process in the south.

Blocking Operation: `MPI_Sendrecv` is a blocking operation, meaning it will not proceed to the next line of code until both sending and receiving are completed.`MPI_Sendrecv` is a versatile function for bi-directional communication between MPI processes, and its usage allows for efficient exchange among neighboring processes in a Cartesian grid.

4 All-to-All Communication Algorithm

4.1 Communication Pattern Setup

This algorithm relies on an MPI neighbor grid setup. The idea here is, that within the communicator each process knows its eight neighbors. This allows for send and recv operations in a single call to MPI_NeighborAlltoAllw(). The setup and execution is explained below.

Listing 10: Communication pattern setup

```
// Communication pattern setup
const short data_size = 8;
int sendcounts[data_size], recvcounts[data_size];
MPI_Aint sdispls[data_size], rdispls[data_size];
MPI_Datatype dtypes[data_size];

MPI_Comm GRAPH_COMM;
reorder = 1;
MPI_Dist_graph_create_adjacent(GRID_COMM,
                               t, sources, MPI_UNWEIGHTED,
                               t, destinations, MPI_UNWEIGHTED,
                               MPI_INFO_NULL, 0, &GRAPH_COMM);


// Concatenate send vectors into a single buffer
std::vector<int> sendBuffer;
// ... (code for preparing sendBuffer)

// Prepare receive buffer
std::vector<int> recvBuffer(NX * 2 + NY * 2 + 4, 0);

// Perform MPI_Neighbor_alltoallw communication
MPI_Neighbor_alltoallw(sendBuffer.data(), sendcounts, sdispls, dtypes,
                       recvBuffer.data(), recvcounts, rdispls, dtypes, GRAPH_COMM);
MPI_Barrier(GRAPH_COMM);
```

Adjacent Distribution Graph Explanation

1. **Communication Pattern Setup:** The code initializes arrays to define the communication pattern for the `MPI_Neighbor_alltoallw` function. The pattern involves horizontal, vertical, and diagonal communication.

2. **Neighbor Rank Calculation:** Arrays `sources` and `destinations` are calculated based on Cartesian coordinates and target offsets, determining the source and destination ranks for communication.

3. **Communication Graph Creation:** `MPI_Dist_graph_create_adjacent` creates a new communicator (`GRAPH_COMM`) based on the specified communication graph.

4. **Concatenating Send Vectors:** `sendBuffer` is created by concatenating vectors representing data to be sent in different directions (North, South, East, West, and diagonals).

5. **Buffer Preparation and Communication:** `recvBuffer` is prepared to store received data. `MPI_Neighbor_alltoallw` is then used for communication, exchanging data based on the defined communication pattern.

6. **Barrier Synchronization:** `MPI_Barrier` ensures synchronization among processes in `GRAPH_COMM`.

4.2 MPI_Neighbor_alltoallw Explanation

- `MPI_Neighbor_alltoallw` is a collective communication function that performs a generalized neighborhood all-to-all communication. It allows processes to exchange data with their neighbors based on user-defined communication patterns.

- **Parameters:**
  - `sendBuffer.data()`: Pointer to the send buffer data.
  - `sendcounts`, `sdispls`, `dtypes`: Descriptions of the send buffer.
  - `recvBuffer.data()`: Pointer to the receive buffer data.
  - `recvcounts`, `rdispls`: Descriptions of the receive buffer.
  - `new_comm`: Communicator specifying the communication topology.

5 Instructions for Compilation and Execution

Below are instructions for compiling and running the MPI code:

Listing 11: MPI Compilation and Execution

```
# Step 1: Compile the MPI code
mpic++ ./src/main.cpp -o ./bin/GameOfLifeMPI -lpthread -DUSEMPI -std=c++14 -O3 -Wall -pedantic -
    march=native

# Step 2: Run the MPI code
# Replace NPROCS, REPETITION, RESOLUTION, ITERATIONS, and CORRECTNESS_TEST with actual values
mpirun -n NPROCS --use-hwthread-cpus ./bin/GameOfLifeMPI REPETITION RESOLUTION ITERATIONS
    CORRECTNESS_TEST
```

6 Correctness Assessment

In general, there are a lot of possibilities to test for correctness. In this case each process generates its own randomized domain. To test the correctness a function called `getGrid()` was implemented that uses the `MPI_Gather()` function to collect all the subdomains and their corresponding position in the grid - see code listing below. The domain information is stored in a single vector that is split up and reassembled into the full domain matrix. This function is called before and after the main part of the solver starts running (the part that is timed) to get the initial domain and the domain after the specified number of life cycles. To check whether this result is correct or not the function `run_sequential()` is called which runs the game of life sequentially. Note that the file "game_sequential.hpp" and its functions like `run_sequential()` are totally separated from the parallel implementation and are only used for testing the correctness. By this, it can be ensured that mistakes in the parallel implementation can be found more easily. The parallel and the sequential domain are then printed to the terminal, where it can be assessed visually if both domains are identical. This was successfully tested for various inputs and it is therefore, assumed that the provided implementation is correct.

Listing 12: Collecting the submatrices and their coordinates in `getGrid()`

```
std::vector<int> submatrix_collection(NX * NY * dims[0] * dims[1]);

// get coordinates of submatrices
std::vector<int> all_coords(2 * numprocs);
MPI_Gather(&coords, dims[0], MPI_INT,
          all_coords.data(), dims[0], MPI_INT, 0, COMM);

// get submatrices
MPI_Gather(solutionView.getVector().data(), solutionView.getVector().size(), MPI_INT,
```

```
10          submatrix_collection.data(), solutionView.getVector().size(), MPI_INT,
11          0, COMM);
```

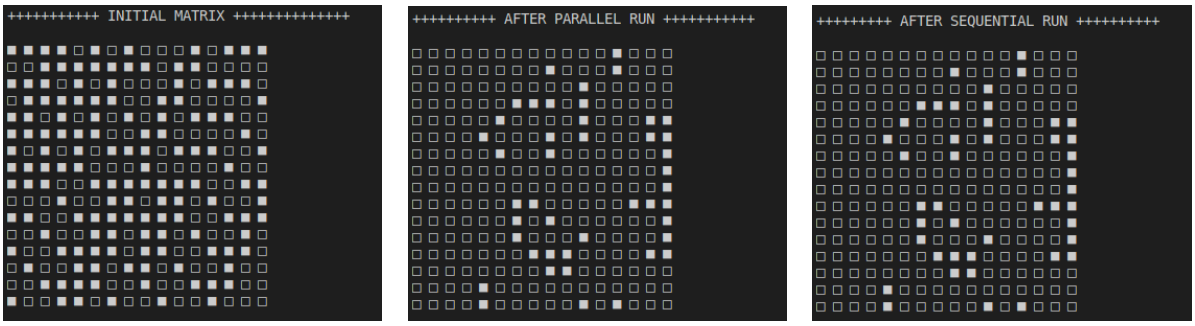

Figure 1: Domains printed to terminal for assessing correctness

7 Technical Details for benchmarking the Implementation

In general, the goal of the benchmarks was to test the developed code in terms of weak and strong scaling capabilities. In the context of the game-of-life, weak scaling describes the ability of the algorithm to simulate finer resolutions with a higher number of processors while keeping the runtime of the algorithm constant. In comparison to that, strong scaling describes the ability to speed-up the simulation of a fixed grid resolution by increasing the number of processors. In the best case, this speed-up behaves linearly in the number of processes.

The benchmarks which were conducted in this work where repeated for 20 times. Each repetition comprises of 50 game-of-life iterations. The runtime of the slowest processor of one parallel run was defined to be the runtime of the overall parallel algorithm. The resolution and the number of processes of each benchmark can be found in table 1. Each benchmark was performed utilizing both described communication strategies individually.

Table 1: Details about used settings for the benchmarks

Weak Scaling Experiments					
# processors	1	32	256	512	1024
resolution	1024×1024	5793×5793	16384×16384	23170×23170	32768×32768
Strong Scaling Experiments					
# processors	1	32	256	512	1024
resolution	all # processors benchmarked with 1024×1024 and 10240×10240				

8 Benchmark results and Discussion

8.1 Reorder Findings

In our MPI program, we utilized `MPI_Cart_create` to create a Cartesian communicator with the `reorder` parameter set to true. Subsequently, we used `MPI_Comm_compare` to compare two communicators, and the result was `MPI_IDENT`, indicating that the processes were considered identical.

Possible Reason for unchanged Process Ordering The MPI library may have a default behavior that retains the original order of processes if it is deemed optimal for the given communicator. This default behavior might result in processes not being reordered even when `reorder` is set to true.

8.2 Strong Scaling Experiments

As it can be seen in table 1, the strong scaling experiments for both communication strategies were performed for two different resolutions. The generated runtime plots can be found in figure 2. In the case of smaller resolution, the amount of computation that has to be done by each processor to perform one game-of-life iteration gets small quite quickly, leading to a communication bound runtime for already few processors. However, in the case of larger resolution the problem is much more computational bound and therefore a higher maximum speed-up is anticipated.

In fact, this is exactly what can be observed when looking at figure 2. While the runtime in the low resolution case is already converged when utilizing 256 processors, a steady decrease over the whole processor range is found in the high resolution case. Additionally, it can be observed that both communication strategies behave quite similar. The only notable difference lies in the runtime increase in the small resolution case for large processor counts when utilizing point-to-point communication. In contrast to that, using all-to-all communication does not increase the runtime.

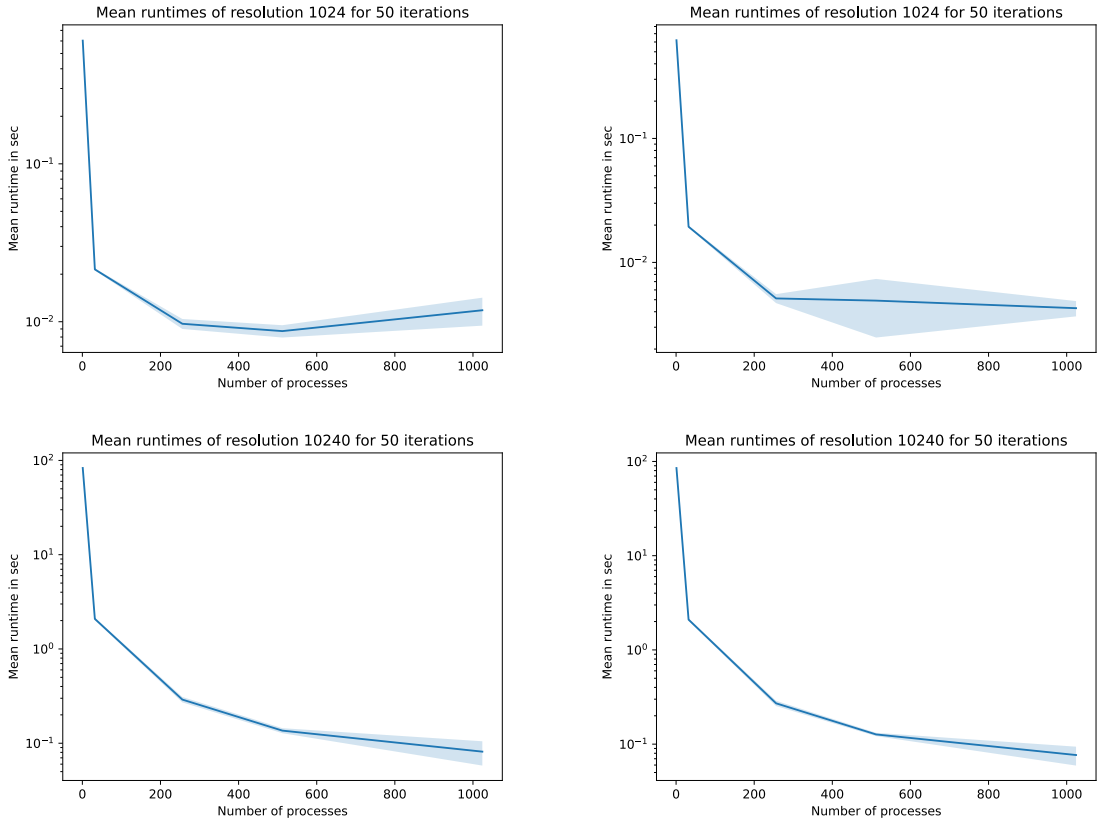

Figure 2: Mean runtime plots of the strong scaling benchmarks with point-to-point communication (left) and all-to-all communication (right) for the resolutions 1024×1024 (top) and 10240×10240 (bottom)

The same observations can be made when looking at the parallel efficiency and speed-up plots in figure 3. Both speed-up and parallel efficiency of the high resolution case behave roughly ideal over the whole range of processor counts. However, in the low resolution case parallel efficiency and speed-up converge at quite low values. Additionally, only very small differences were measured between the two communication strategies.

One observation which has to be analyzed in more detail is the occurrence of parallel efficiencies ≥ 1 in the high resolution case. This effect can be explained by the large amount of memory needed to story the high resolution 2D grid. While one processor is probably not able to store the whole 2D matrix in the cache memory and therefore needs more time for editing the matrix, splitting the matrix to many processors enables storing the data in the cache again and therefore increases the computation even further.

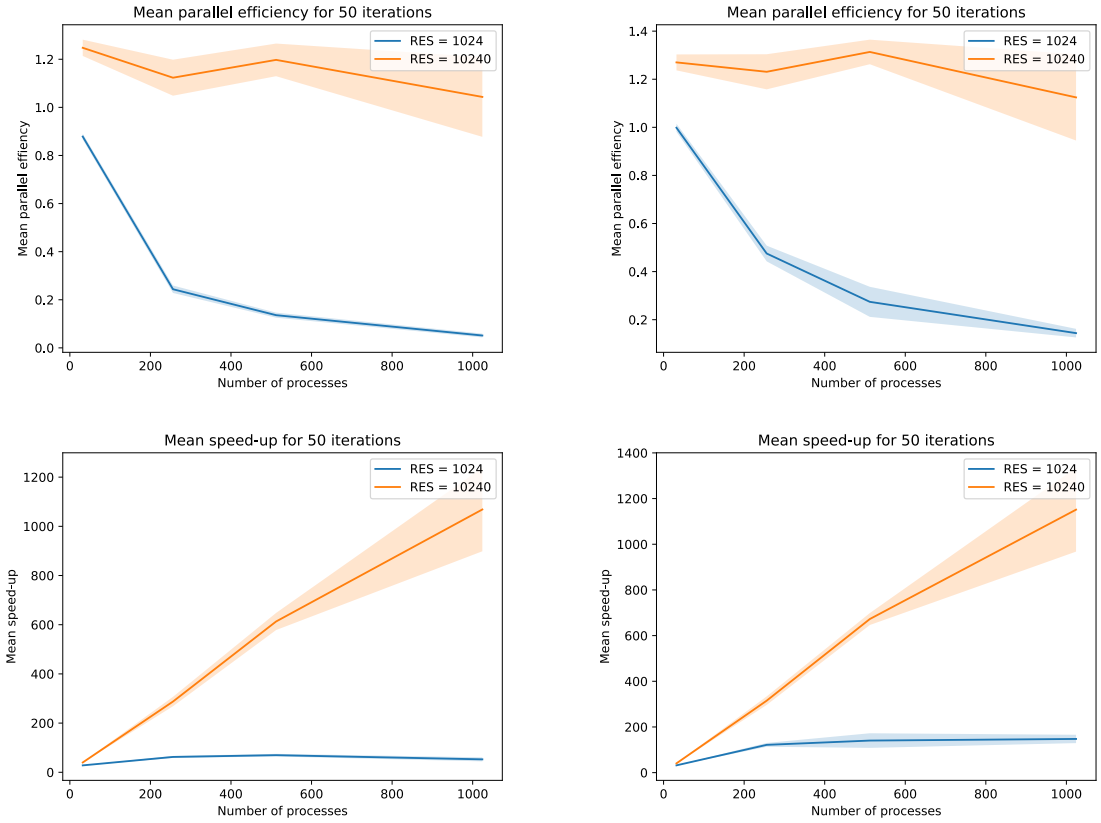

Figure 3: Parallel efficiency (top) and speed-up plots (bottom) of the strong scaling benchmarks with point-to-point (left) and all-to-all communication (right) for the resolutions 1024×1024 and 10240×10240 (bottom)

8.3 Weak Scaling Experiments

The weak scaling experiments were designed in way that the resolution is increased in a way so that the overall runtime stays constant when utilizing additional processors. However, as it can be seen in figure 4, this ideal behavior was only observed up to 256 processors. This behavior can be observed for both communication strategies. One explanation for the decreasing parallel efficiency above 256 processes could be a saturation of the bandwith of the interconnect between the nodes

due to communication of many processors at once.

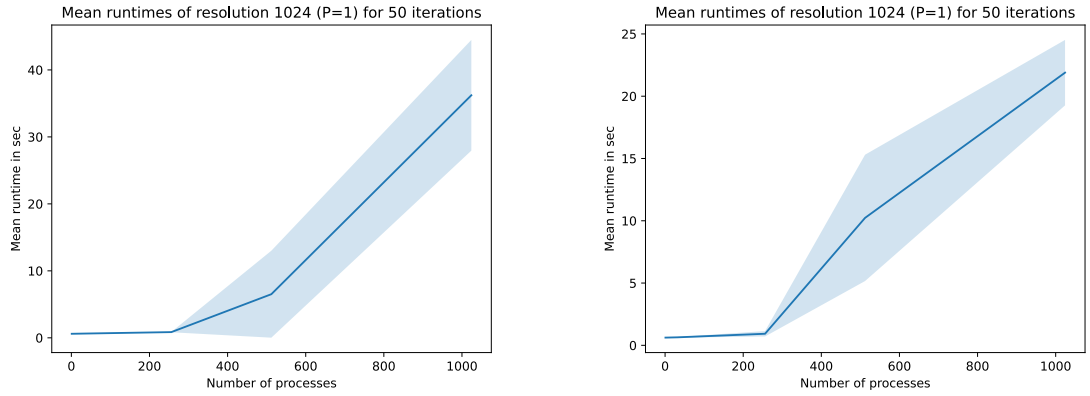

Figure 4: Mean runtime plots of the weak scaling benchmarks with point-to-point (left) and all-to-all communication (right) for a resolution of $\left(\sqrt{p} \cdot 1024\right) \times \left(\sqrt{p} \cdot 1024\right)$. The number of processes are encoded with p.

9 Problems encountered and possible improvements

One of the inconveniences we had was the timing of the sequential run with resolution 10240. We wanted to have 20 repetitions for each resolution but we ran out of time after the second run (the first two runs took more than 5 minutes). So we had to run it 20 times separately. The problem itself is the random domain generation that simply takes a lot of time. Giving each process a predefined configuration would be much faster.

We had a weird problem in the solver for the all-to-all communication. There a new communicator is created based on the grid communicator from `MPI_Cart_Create()`. For some reason this new communicator caused a lot of MPI related errors. Through trial and error we found that simply copying the new communicator into a new variable solves the problem. It would be interesting to find out what exactly happened and caused this spurious error.

For the correctness assessment we visually inspected the whole domain by printing it to the terminal. A nice improvement would be to automate the correctness test (e.g. summing the numbers of all living cells in each domain and check against sequentially computed domain).

Technical Details of the Implementation

The GitHub repository can be found under `https://github.com/simon-koenig/Parallel-Game-of-Life`.

Requirements

For the main part of the code:

- C++ compiler; e.g. on the cluster: gcc (Debian 10.2.1-6) 10.2.1 20210110
- MPI: openmpi@4.1.5

For plotting the results with Python3:

- argparse
- os
- pathlib import Path
- itertools
- matplotlib
- numpy

Used system: TU Wien Cluster System Hydra

Hydra is a cluster system containing a headnode and 36 compute nodes. It is powered by Intel Xeon Gold 6130F and (inter)connected by Intel Omnipath. Currently it is running Debian 10. More details on `https://doku.par.tuwien.ac.at/docs/machines/`.

Listing 13: Computenodes hydra01-hydra36 - important specs

```
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
Address sizes:       46 bits physical, 48 bits virtual
CPU(s):              32
Thread(s) per core:  1
Core(s) per socket:  16
Socket(s):           2
NUMA node(s):        2
Model name:          Intel(R) Xeon(R) Gold 6130F CPU @ 2.10GHz
CPU MHz:             1000.460
CPU max MHz:         2100,0000
CPU min MHz:         1000,0000
L1d cache:           32K
L1i cache:           32K
L2 cache:            1024K
L3 cache:            22528K
```