Assignment 1: Game of Life Stencil

Florian Fritzer, Markus Hickel, Leon Schwarzäugl

January 16, 2024

1 Short report

We made a two separate programs:

- sequential program, which is described in the section of Exercise 1
- parallel program, which is described in the section of Exercise 2-4

The two programs can be compiled with `make Ex1` for the sequential program and `make Ex2` for the parallel program. Our sequential implementation utilized only one process. The input parameters for the sequential program are:

- number of generations
- size of the grid $n = n_r \cdot n_c$
- initial configurations: probability that cell is alive, seed of random number generation
- repetitions

How to run the code:

`Usage: ./Ex1 <generations> <rows> <cols> <seed> <probability(%)> <repetitions>`
Build with: `make Ex1`

The program can output the final configuration of the grid after all generations, the number of dead and alive cells, the average time it took to run through all generations with the standard deviation.
The asymptotic complexity per generation is $\mathcal{O}(n_r \cdot n_c)$. Also we tested empirically that the runtime does not depend on the configuration of the input matrix, just the total number of cells $n = n_r \cdot n_c$ - this results from our chosen implementation, that checks all cells neighbors in every generation.

The parallel version can be called similarly:
`Usage: mpirun -np <numprocs> ./Ex2 <generations> <rows> <cols> <seed> <probability(%)> <repetitions> <px> <py> <method> <print>`
Build with: `make Ex2`

We added input arguments to specify the dimensions of the Cartesian grid (p_x processes in x direction, p_y processes in y direction), as well as the desired communication method. One can choose from point-to-point with separate sends for the corners (`"p2pc"`), point-to-point with just sending rows and columns (`"p2p"`) and the collective communication (`"coll"`). Providing print (`"0"`) prints the state of the board after the last iteration, the timing and the number of dead and alive cells. With print (`"0"`), only the timing and the number of dead and alive cells is printed. Of course the parallel version can be run by just one process, in which case you could consider it a serial version. We provided a small shell script that generates the output for small values. This can be done by running (`"./run_all_tests_small.sh"`). The results can be found in the directories "p2pc_small", "p2p_small" and "coll_small".

limitations

The parallel version is restricted to configurations where the columns and rows are multiples of the number of processes in the respective direction of the cartesian communicator:

$$rows \mod p_y = 0$$

$$cols \mod p_x = 0$$

2 Code explained

Exercise 1

The grid is stored as a 2d `std::vector` of type `char`. The type `char` is a memory efficient datatype to store a binary variable. A cell of this grid will be assigned `'1'` if it is alive, and `'0'` if it is dead. When initializing the *Game of Life*, the grid will be filled with an initial configuration, based on the input parameters; a pseudo-random number will be drawn from a uniform distribution and then normalized by dividing it by the maximum value. In the command line arguments the probability for a cell to be alive was given; if the normalized random number is smaller than that probability, then the cell will be set to `'1'`, else `'0'`. The seed is combined with the global indices for row and column to create a unique seed for each cell of the grid for the generator.

```
1  char get_random_value(const int row_id, const int col_id,
2                        const int n, const int seed, double probability) {
3      char r = '0';
4      int my_seed = seed + row_id * n + col_id;
5      srand(my_seed);
6      double rand_value = static_cast<double>(rand()) / RAND_MAX;
7      if (rand_value < probability) {
8          r = '1';
9      }
10     return r;
11 }
```

Figure 1: random number generator

To start the *Game of Life*, the function `runLife` is called. For every generation this function will go through every cell and will update its value, based on the number of alive neighbors. The number of alive neighbors is calculated in the function `countNeighbours`. The function `countNeighbours` deals with the periodic boundaries by looking at the neighbors $C[i][j \ominus 1], C[i \oplus 1][j \ominus 1], C[i \oplus 1][j], C[i \oplus 1][j \oplus 1], C[i][j \oplus 1], C[i \ominus 1][j \oplus 1], C[i \ominus 1][j], C[i \ominus 1][j \ominus 1]$, where $x \oplus 1 = (x+1) \bmod m$ and $x \ominus 1 = (x-1+m) \bmod m$ (where $m = n_r$ for rows and $m = n_c$ for columns).

```
1  int plus(int x, int m) {return (x+1)%m;}
2
3  int minus(int x, int m) {return (x-1+m)%m;}
4
5  int count_neighbours(std::vector<std::vector<char>> &world, int i, int j, int rows, int cols)
       {
6      int number_of_neighbours = 0;
7      if (world[i][plus(j,cols)] == '1') number_of_neighbours++; // right
8      if (world[minus(i,rows)][plus(j,cols)] == '1') number_of_neighbours++; // down right
9      if (world[minus(i,rows)][j] == '1') number_of_neighbours++; // down
10     if (world[minus(i,rows)][minus(j,cols)] == '1') number_of_neighbours++; // down left
11     if (world[i][minus(j,cols)] == '1') number_of_neighbours++; // left
12     if (world[plus(i,rows)][minus(j,cols)] == '1') number_of_neighbours++; // up left
13     if (world[plus(i,rows)][j] == '1') number_of_neighbours++; // up
14     if (world[plus(i,rows)][plus(j,cols)] == '1') number_of_neighbours++; // up right
15     return number_of_neighbours;
16 }
```

Figure 2: counting the number of alive neighbors

Let's look at the if-clauses of `runLife` in more detail. If the cell is dead `world[i][j] == '0'` (lines 6-10 in figure 3) the cell will be brought back to life only if the number of neighbours is exactly 3. If the cell is alive `world[i][j] == '1'` (lines 11-15 in figure 3), it will die if the number of neighbours is not exactly 2 or 3, otherwise it will live on. We cannot write the updates to the same grid as we loop through all cells, because then the number of alive neighbors for the later cells could be incorrect. Therefore the updated value is written to a second grid of the same size, which has the same initial values. If all changes are done, then the original grid can be filled with the new values and the next generation continues.
After all generations are done, the final configuration is evaluated for the total number of alive and dead cells. The runtime of the *Game of Life* is the time it takes for the function `runLife` to run through all generations. The function is done a specified number of repetitions (after resetting to the initial state each time) and the mean runtime is calculated.

```
void life(std::vector<std::vector<char>> &world, std::vector<std::vector<char>> &world_copy,
    int generations, int rows, int cols) {
    for (int gen = 0; gen<generations; gen++) {
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                int neighbours = count_neighbours(world, i, j, rows, cols);
                if (world[i][j] == '0') {
                    if (neighbours == 3) {
                        world_copy[i][j] = '1';
                    }
                }
                else {
                    if (neighbours != 2 && neighbours != 3) {
                        world_copy[i][j] = '0';
                    }
                }
            }
        }
    world = world_copy;
    }
}
```

Figure 3: running the *Game of Life*

Exercise 2 & 3

The first part of this exercise was to create a new 2-d cartesian grid communicator, the dimensions are determined by passing input arguments. The whole grid is then divided into subgrids for each rank. To let the subgrids know their neighboring cells, the subgrids were extended by a ghost layer around the grid, so that the subgrid has the size $(n_r/p_y + 2) \cdot (n_c/p_x + 2)$. The boundary values of the neighboring subgrids are then communicated into this ghost layer.
The data that is communicated needs to be contiguous. However in a 2d `std::vector` it is not guaranteed to have the whole data in a contiguous block of memory. If we want to avoid copying the boundary layer data into temporary contiguous arrays before sending them, we can use a 1d vector and access the elements with a stride with derived MPI datatypes. Using temporary arrays would be not only less convenient, it also adds unnecessary computational overhead and adds to the used memory. Therefore the grid is now stored in a one dimensional `std::vector` of type `char`, where all columns are contiguous and the rows have a constant stride. For the columns and corner points we created derived datatypes as well, even tough more out of convenience as they are not strictly necessary. For the rows we used MPI_Type_vector types to have the necessary stride available, for the columns and corners we used MPI_Type_contiguous. All the derived MPI datatypes and defined communicators are initialized in the `init_mpi()` function. To make accessing the cells easier, a class was created for this container `matrix2D`, where the `()` operator is overloaded so it can accept the indices of the cell as if it were a 2D grid and return the value of the vector at the specified location.

```
template <class T>
class matrix2D {
    std::vector<T> data;
    int rows;
public:
    T &operator()(int x, int y) {
        return data[y * rows + x];
    }
    matrix2D(int x, int y, const T &initialValue = T()) : data(x * y, initialValue), rows(x)
    {}
};
```

Figure 4: storage container for the grid

In the `runLife` function the communication with the neighbors is happening before entering the loops over the cells. Also the loops only iterate over the inner cells without the ghost layer.
The `countNeighbours` function now can always take the cell with neighboring indices, because we defined a ghost layer. so $x \oplus 1$ becomes $x + 1$ and $x \ominus 1$ becomes $x - 1$.

```
void GameOfLife::runLife(int generations, std::string method="p2p") {
    for (int gen = 0; gen < generations; gen++) {
        if (method == "p2p") {exchangePointToPoint();};
        if (method == "p2pc") {exchangePointToPointCorners();};
        if (method == "coll") {exchangeCollective();};
        for (int i = 1; i < _rows-1; ++i) {
            for (int j = 1; j < _cols-1; ++j) {
                int neighbours = countNeighbours(i, j);

                if (_world(i,j) == '0') {
                    if (neighbours == 3) {
                        _worldCopy(i,j) = '1';
                    } else {
                        _worldCopy(i,j) = '0';
                    }
                } else {
                    if (neighbours != 2 && neighbours != 3) {
                        _worldCopy(i,j) = '0';
                    } else {
                        _worldCopy(i,j) = '1';
                    }
                }
            }
        }
        _world = _worldCopy;
    }
}
```

Figure 5: running the *Game of Life*

Communication

In our first naive approach, we simply exchanged the columns and rows between the neighboring processes and used separate sends and receives for the corners. This equals to ghost layer columns of size cols-2 and ghost layer rows of size rows-2. This leads to

$$c_{\text{total}} = 2 \cdot \text{cols} + 2 \cdot \text{rows} + 4 \cdot \text{corners} = 8$$

communication exchanges, with c_{total}, the global count of communications.
Later we found another approach described here: `https://dl.acm.org/doi/pdf/10.1145/582034.582084` ([1]). Instead of sending the corners in separate communication steps, the corners are included into the columns and rows of the ghost layers. So we have columns

$$c_{\text{total}} = 2 \cdot \text{cols} + 2 \cdot \text{rows} = 4$$

This leads to a higher number of total values sent, but decreases the number of MPI calls and the respective overhead introduced by these operations.

```
    MPI_Irecv(&_world(0,_cols-1), 1, _fullColType, _right, 7, _cart, &_req[0]);
    MPI_Irecv(&_world(0,0), 1, _fullColType, _left, 8, _cart, &_req[1]);
    MPI_Isend(&_world(0,1), 1, _fullColType, _left, 7, _cart, &_req[3]);
    MPI_Isend(&_world(0,_cols-2), 1, _fullColType, _right, 8, _cart, &_req[2]);
    MPI_Waitall(4, _req, _statuses);

    MPI_Irecv(&_world(_rows-1,0), 1, _fullRowType, _bot, 5, _cart, &_req[1]);
    MPI_Irecv(&_world(0,0), 1, _fullRowType, _top, 6, _cart, &_req[0]);
    MPI_Isend(&_world(1,0), 1, _fullRowType, _top, 5, _cart, &_req[2]);
    MPI_Isend(&_world(_rows-2,0), 1, _fullRowType, _bot, 6, _cart, &_req[3]);
    MPI_Waitall(4, _req, _statuses);
```

Figure 6: point to point version without sending the corner elements separately

```
    MPI_Irecv(&_world(1,_cols-1), 1, _colType, _right, 7, _cart, &_req[0]);
    MPI_Irecv(&_world(1,0), 1, _colType, _left, 8, _cart, &_req[1]);
    MPI_Isend(&_world(1,1), 1, _colType, _left, 7, _cart, &_req[3]);
    MPI_Isend(&_world(1,_cols-2), 1, _colType, _right, 8, _cart, &_req[2]);
    MPI_Waitall(4, _req, _statuses);

    MPI_Irecv(&_world(_rows-1,1), 1, _rowType, _bot, 5, _cart, &_req[1]);
    MPI_Irecv(&_world(0,1), 1, _rowType, _top, 6, _cart, &_req[0]);
    MPI_Isend(&_world(1,1), 1, _rowType, _top, 5, _cart, &_req[2]);
    MPI_Isend(&_world(_rows-2,1), 1, _rowType, _bot, 6, _cart, &_req[3]);


    MPI_Waitall(4, _req, _statuses);

    MPI_Isend(&_world(1,1), 1, MPI_CHAR, _upperLeftRank, 1, _cart, &_req[0]);
    MPI_Irecv(&_world(0,0), 1, MPI_CHAR, _upperLeftRank, 0, _cart, &_req[0]);

    MPI_Isend(&_world(1,_cols-2), 1, MPI_CHAR, _upperRightRank, 2, _cart, &_req[1]);
    MPI_Irecv(&_world(0,_cols-1), 1, MPI_CHAR, _upperRightRank, 4, _cart, &_req[1]);

    MPI_Isend(&_world(_rows-2,1), 1, MPI_CHAR, _lowerLeftRank, 4, _cart, &_req[2]);
    MPI_Irecv(&_world(_rows-1,0), 1, MPI_CHAR, _lowerLeftRank, 2, _cart, &_req[2]);

    MPI_Isend(&_world(_rows-2,_cols-2), 1, MPI_CHAR, _lowerRightRank, 0, _cart, &_req[3]);
    MPI_Irecv(&_world(_rows-1,_cols-1), 1, MPI_CHAR, _lowerRightRank, 1, _cart, &_req[3]);

    MPI_Waitall(4, _req, _statuses);
```

Figure 7: point to point version with sending the corner elements separately

Exercise 4

In a next step we used `MPI_Neighbor_alltoallw` for communication. Since we already had the MPI datatypes defined, `MPI_Neighbor_alltoallw` was the better choice than `MPI_Neighbor_alltoallv` for us. All the setup with the new communicator, send and receive arrays, was done in the function `init_mpi`, which is run before the `runLife` starts. With the right counts, displacements and types defined, the communication can be done with one command (figure 8), if we define a new neighborhood communicator.

```
void GameOfLife::exchangeCollective() {
    MPI_Neighbor_alltoallw(&_world(0,0),sendcount,senddisp,sendtype,
                      &_world(0,0),recvcount,recvdisp,recvtype,_coll);
}
```

Figure 8: function for communicating boundary cells with collective communication

With `MPI_Dist_graph_create_adjacent` (figure 9) the neighborhood was defined, by specifying that we need 8 neighbors and passing the ranks of these neighbors in the old, cartesian communicator. The derived MPI

```
MPI_Dist_graph_create_adjacent(_cart, 8, _neighbors, MPI_UNWEIGHTED,
                                         8, _neighbors, MPI_UNWEIGHTED,
                                         MPI_INFO_NULL, 0, &_coll);
```

Figure 9: defining the neighborhood with a new communicator

datatypes are reused from the previous communication. Because we used derived MPI datatypes, the send and receive counts are 1 for all neighbors. However the hard part was figuring out how to define the send and receive displacements. They include the starting indices for the data we send to each neighbor. The order we defined the neighbors in the array that was passed to `MPI_Neighbor_alltoallw`, is: top left, top right, bottom left, bottom right, left, right, top, bottom. This order is also represented in the send displacements (figure 10). The function `c(i,j)` is used to give the index in the underlying 1d vector, if we pass the indices of the cell as if it were a 2d array.
The receive displacements were also defined in the same order, but the indices now belong to the ghost layers. In this way the right boundary of the left neighbor should be sent to the left ghost cells in the current subgrid. The send and receive locations are visualized in the schematic in figure 12. This works as expected only if the

```
senddisp[0] = c(1, 1);                sendtype[0] =  _cornerType; // top left
senddisp[1] = c(1, _cols-2);          sendtype[1] =  _cornerType; // top right
senddisp[2] = c(_rows-2, 1);          sendtype[2] =  _cornerType; // bottom left
senddisp[3] = c(_rows-2, _cols-2);    sendtype[3] =  _cornerType; // bottom left
senddisp[4] = senddisp[0];            sendtype[4] =  _colType; // left
senddisp[5] = senddisp[1];            sendtype[5] =  _colType; // right
senddisp[6] = senddisp[0];            sendtype[6] =  _rowType; // top
senddisp[7] = senddisp[2];            sendtype[7] =  _rowType; // bottom
```

Figure 10: send displacements

number of processes in both dimensions in the cartesian communicator are greater than 2. If the number of processes in one dimension is 1 or 2, then either the rank has itself as neighbor, or it has the same neighbor twice in that dimension, because we defined the communicator as periodic. In that case the receives are switched in this dimension. To better understand, we can look at an example configuration for cartesian grid communicator dimensions 1 x 3 (figure 13). Here we can see that in the x-direction the number of processes is 1. Therefore the right boundary of the left neighbor is sent to the right ghost cells of the current rank. So all lefts are switched to right and vice versa. The same goes for a 3 x 1 configuration, where top and bottom are switched. And for a configuration with ($n_x < 3$) x ($n_y < 3$) all directions are switched. We found this discussion, describing the problem we encountered: `https://github.com/mpi-forum/mpi-issues/issues/153` ([2]).

```
recvdisp[0] = c(0, 0);                recvtype[0] =  _cornerType; // top left
recvdisp[1] = c(0, _cols-1);          recvtype[1] =  _cornerType; // top right
recvdisp[2] = c(_rows-1, 0);          recvtype[2] =  _cornerType; // bottom left
recvdisp[3] = c(_rows-1, _cols-1);    recvtype[3] =  _cornerType; // bottom right
recvdisp[4] = c(1, 0);                recvtype[4] =  _colType; // left
recvdisp[5] = c(1, _cols-1);          recvtype[5] =  _colType; // right
recvdisp[6] = c(0, 1);                recvtype[6] =  _rowType; // top
recvdisp[7] = c(_rows-1, 1);          recvtype[7] =  _rowType; // bottom
```

Figure 11: receive displacements

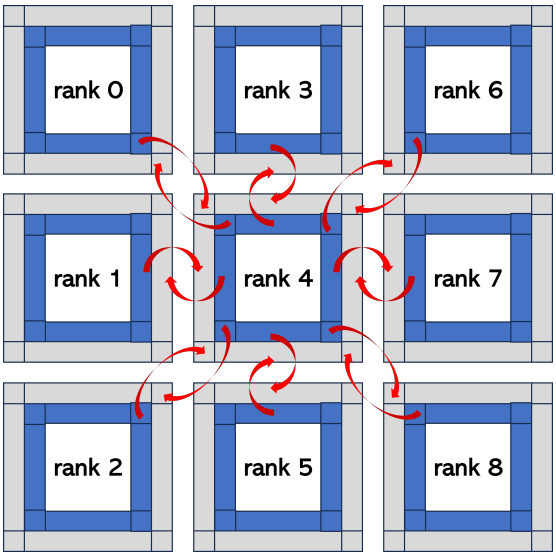

Figure 12: communication for a 3 x 3 communicator for rank 4

In the end we made a check if one (or both) of the dimensions in the communicator is less than 3 and then switched the according receive displacement entries. This did the trick and the communication was correct for all configurations.

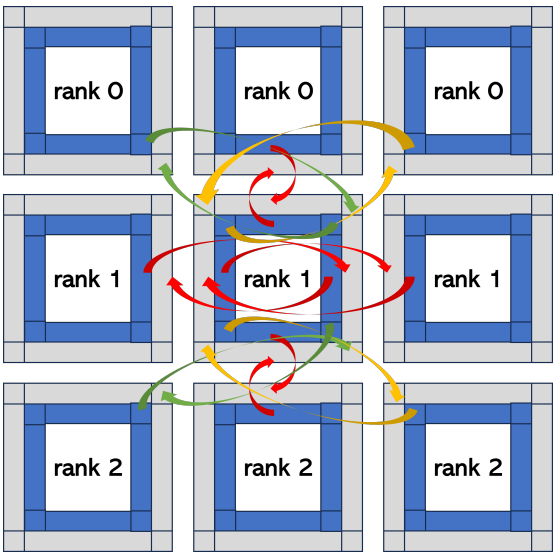

Figure 13: communication for a 1 x 3 communicator for rank 1

3 Experimental Results

All experiments that we performed were run with the runtime argument seed=0 and probability=50. For each experiment we averaged over 10 repetitions with 1000 generations, unless otherwise noted.

Table 1 shows our results for the sequential ($p = 1 \times 1$) run of the small problem with $n = 1024$.

Method	Runtime	Std. Deviation
p2p	1.457s	0.001s
p2pc	1.460s	0.004s
coll	1.459s	0.001s

Table 1: Sequential Experiments, $n = 1024$

Table 2 holds our results for the sequential run of the bigger problem with $n = 10240$. This problem had an estimated runtime of about 30 minutes, hence we were not able to run it fully on the hydra remote. In order to receive an estimate for the runtime on hydra we hence decided to run the code for 100 generations instead. A rough estimator for the runtime of 1000 generations should then be given by multiplying the runtime by a factor of 10:

Method	Runtime for 100 generations	Std. Deviation	Est. runtime, 1000 gen.
p2p	196.426s	2.004s	1964.26s
p2pc	197.169s	1.338s	1971.69s
coll	196.137s	2.738s	1961.37s

Table 2: Sequential Experiments, $n = 10240$

Figure 14 shows the results for our strong scaling experiments; we can see that the runtimes are overall very similar between our three implemented methods and show a linear behavior. For the smaller problem size, we can see a bit of a bigger deviation between the implementations for the `p2pc` method. This can be explained by the higher amount of data that is being sent by that method.
For the bigger problem size, it becomes hard to see a meaningful difference in runtime between the methods; when we ran our program through a profiling tool arm forge, we noticed that our program only spends about $8 - 15\%$ of the time overall on MPI-Communication. The rest of the time is being spent on initializing the boards as well as running through the loop in `runLife`.
In figure 15, the speedup is plotted. Of course, for the $n = 10240$ version, this is again only an estimate. We can see that we receive the biggest speedup the more processes we use in the parallel version, which is also what is to be expected.
In figure 16, the parallel efficiency is plotted. We observe a quite low parallel efficiency for 1x32 processes. We suspect a pinning issue here, since we ran all the runs in a single job on 32 nodes. It could be possible that a

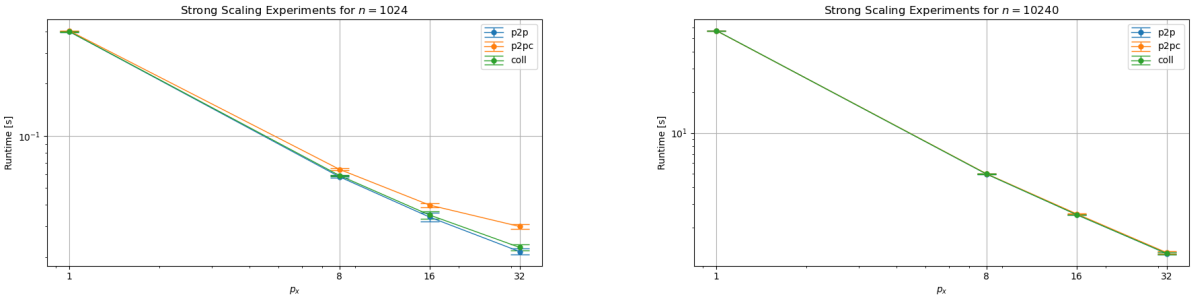

Figure 14: Strong scaling experiments, $n \in \{1024^2, 10240^2\}$

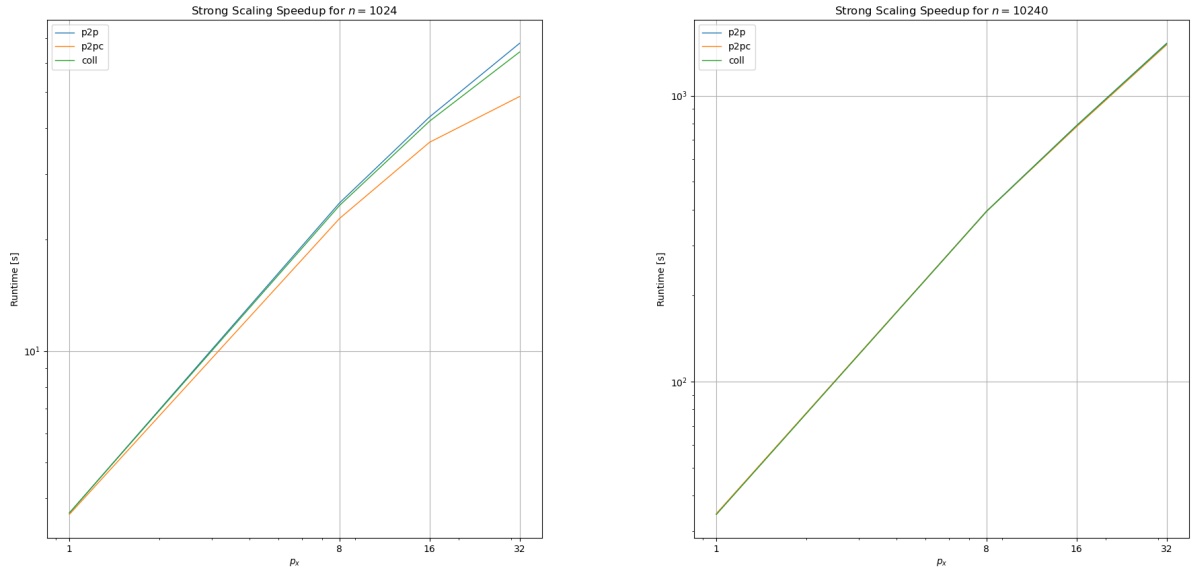

Figure 15: Strong scaling speedup, $n \in \{1024^2, 10240^2\}$

single process was run on every node. This would lead to a lot of communication over the network and therefor bad performance. For the higher process count, we observe a reasonable cut in half of the runtime for twice the amount of processes.

Figure 17 shows the results for our weak scaling experiments. In an ideal case with instantaneous communication, we would have expected here the runtimes to be constant over different process numbers, as the relation between global gridsize and size of the subgrids stays the same for all configurations. However, we can see that this is not the case. Of course, communication introduces an overhead in the practical application, which is why we would have assumed the runtimes to have a monotone increase for rising process numbers. However, we can see two particularities: The first one is that the constellation $p = 16 \times 32$ is faster than the constellation $p = 8 \times 32$, which is surprising. Even after a lot of consideration, we are not certain how this results comes to be.

The other interesting aspect is that the runtimes of each repetition of the $p = 32 \times 32$ case differ a lot. Again, we are not sure why this happens.

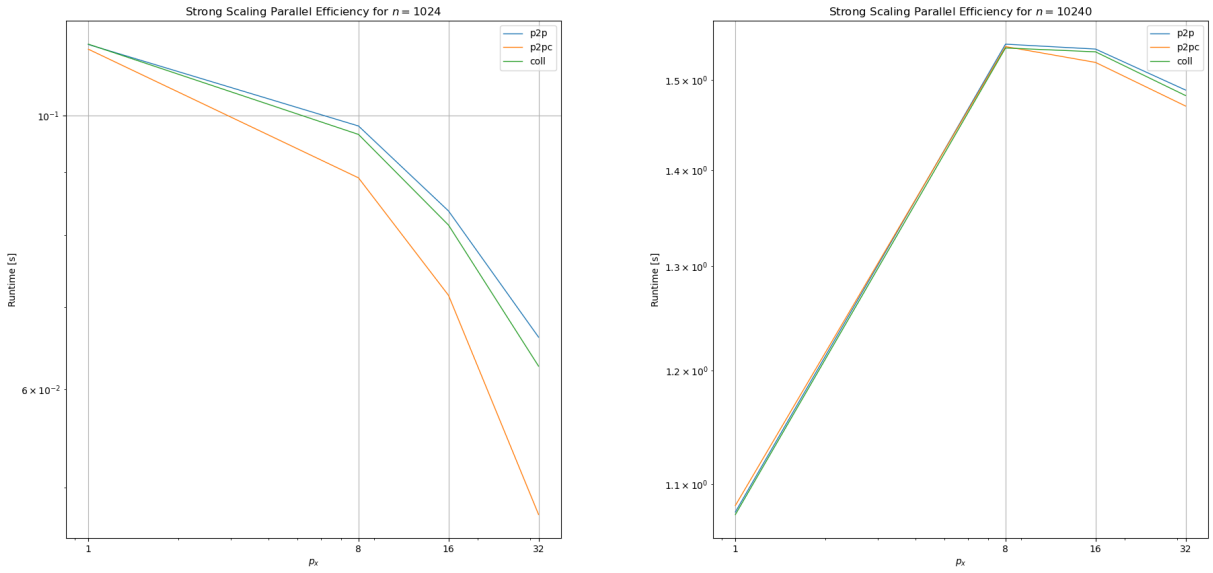

Figure 16: Strong scaling parallel efficiency, $n \in \{1024^2, 10240^2\}$

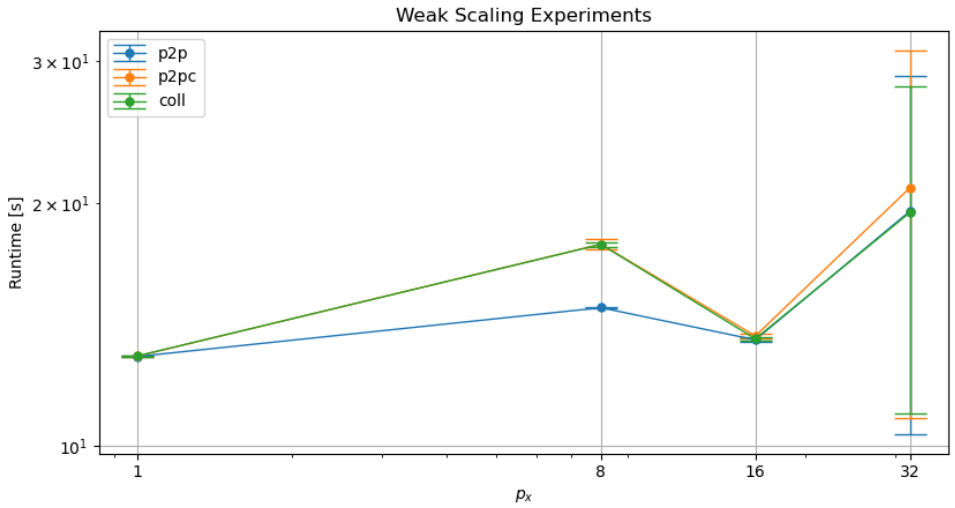

Figure 17: Weak scaling experiments

References

[1] Y. H. Chris Ding, "A ghost cell expansion method for reducing communications in solving pde problems." Accessed: 16.01.2024.

[2] R. R. Tony Skjellum, "Bug in mpi_neighbor_alltoall–1 or 2 processes in the cyclic cartesian case." Accessed: 16.01.2024.