HPC Assignment 1: Game of Life Stencil

2023 TU WIEN, Group Project by Group 16

This is the HPC Project of Group 16 for the TU Wien couse High Performance Computing from WS23/24.

Members of Group 16:

```
-Peter Diem, 11711794
-Adam Nordenhög, 12224156
-Markus Witzko, 11709272
```

In Part I, you find all the information specifying our implementations (what they can do and what they cannot do, what the input parameters are), how to run your codes, and stating what you think works (and what not).

In Part II, you find all the information specifying our implementations: design decisions, data structures, a detailed view on the implementation of the exercises etc. Also, at the very end we present our **benchmarking** results and discuss them briefly.

PART I Short report: How to run, what it can (not) do

1. Project Structure

The project folders are:

```
- /include: hpp files
- /src: cpp files
- /build: .o files, .exe files
- /debug: .csv debug files
- /pics: .png elements for documentation use
- /plotting: benchmark plotting python scripts etc
- /benchmark: benchmark raw result data etc
- /data: .odp file for documentation use, other files used during
implementation
```

Also in the root folder, we have:

```
- Makefile
- run.sh
- Readme.pdf
```

2. How to build and run it

The project can be build with the use of a Makefile, both in the optimized version and the debug version:

```
make clean

make sequential
make parallel

make debug sequential
make debug parallel
```

The object files and executables are then located in the /build folder.

To run the program, we included a shell file **run.sh** for automation. The executable can also be called with the following commands and CL arguments for the sequential version:

```
mpirun -n 1 ./build/sequential <matrix_size_row> <matrix_size_col>
<prob_of_life> <number_of_repetitions>
```

And the following arguments for the parallel version:

```
mpirun -n <num_of_processes> ./build/parallel <matrix_size_row>
<matrix_size_col> <prob_of_life> <number_of_repetitions>
<weak_scaling_flag> <num_procs_by_col> <num_procs_by_row>
```

with the command line arguments:

```
- matrix_size_row: int
- matrix_size_col: int
- prob_of_life: float in range [0, 1.0]
- number_of_repetitions: int
- weak_scaling_flag: true | false
- num_procs_by_col: int
- num_procs_by_row: int
```

e.g. parallel execution: *mpirun -n 16 ./build/parallel 16 16 0.6 100 false 4 4*

This runs the program with 16 processes in a 4x4 grid for a matrix size of 16x16 with probability of life of 60%, 100 iterations and weak-scaling off.

Note: We did not include a command line argument to decide if we run it with collective communication or with P2P. The resepective function must be commented/uncommented in line 218 in /src/parallel/main.cpp.

3. What works (and what not)

3.1 What works

Everything is working fine. We included a variety of debug possibilities to check the results. For instance, when running in DEBUG mode, the grid gets stored at crucial moments inside of the /debug folder in .csv format, which allows for direct comparison between iterations and implementations (sequential vs parallel).

Moreover, we implemented a comparison function areGenerationsEqual(), which checks the equality of the parallel with the sequential version at the very end.

The weak_scaling is also showing the expected results. During implementation, we defined DEBUG sections to check wheter the cell distribution in the matrix for each process is how we expect it to be, which always was the case.

3.2 What does not work

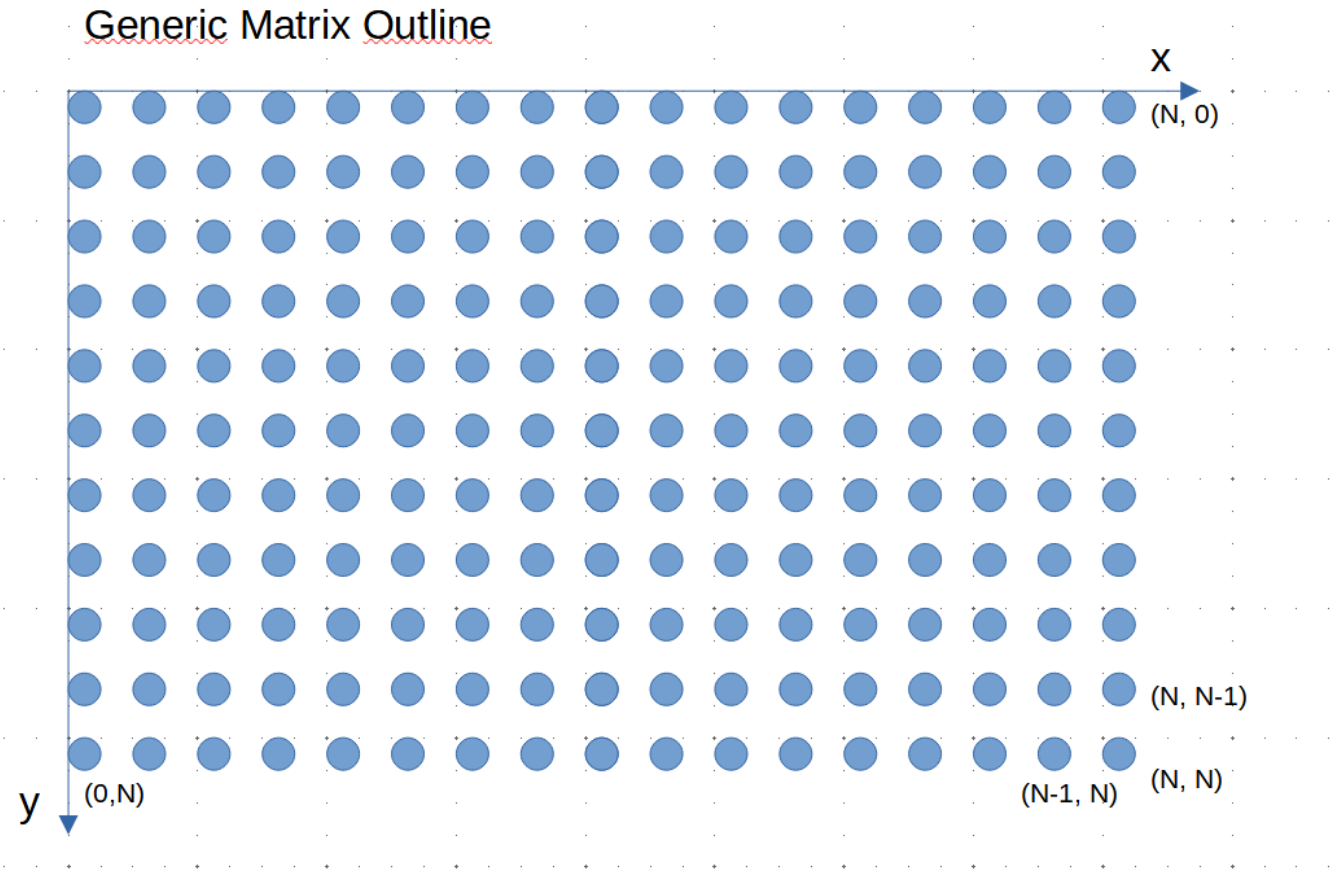
We were fortunately able to solve all problems, and therefore there is nothing to write here.

Part II Long report: Implementation and Benchmarking

In the following, we discuss general ideas on how to implement the project, discuss the exercises in detail, talk about the benchmark and finally, interpret the results.

1. Task description

We are implementing a two dimensional stencil computation on small integers, and for that purpose we choose *Game of Life*. In the *Game of Life* a so-called **Cell** can be either dead or alive. The cells are then distributed in a N x M matrix, which is illustrated in the following picture:



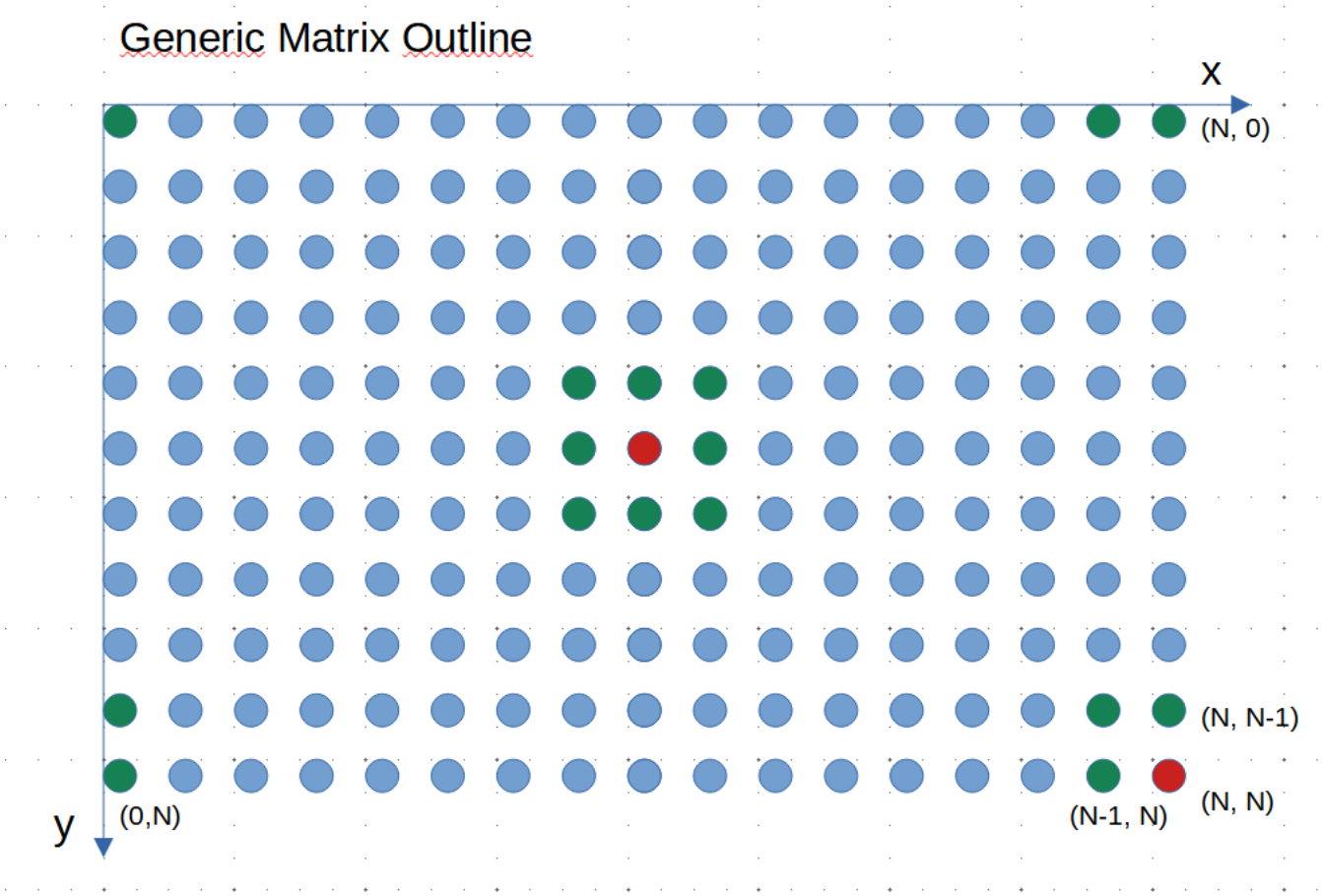
In the first generation, each cell is either dead or alive with a certain probability (input parameter in our program). In the *Game of Life* cells either live or die depending on the state of their neighbours. The following principles must be implemented in our case:

```
- each cell has 8 neighbours
- alive cell survives to next generation if:
    - 2 or 3 neigbours are alive
- dead cell gets reborn to next generation if:
    - exactly 3 neighbours are alive
- if not reborn or survive:
    - dead for next generation
```

These are the core principles to consider. Another important aspect of the task is, that the matrix is folded, so if we access an entry at a out-of-bounce position in an N x M matix:

```
Matrix[N+1, 0] = Matrix[0,0]
Matrix[0, M+1] = Matrix[0,0]
Matrix[N+1, M+1] = Matrix[0,0]
Matrix[N+1, 1] = Matrix[0, 1]
(...... and so on)
```

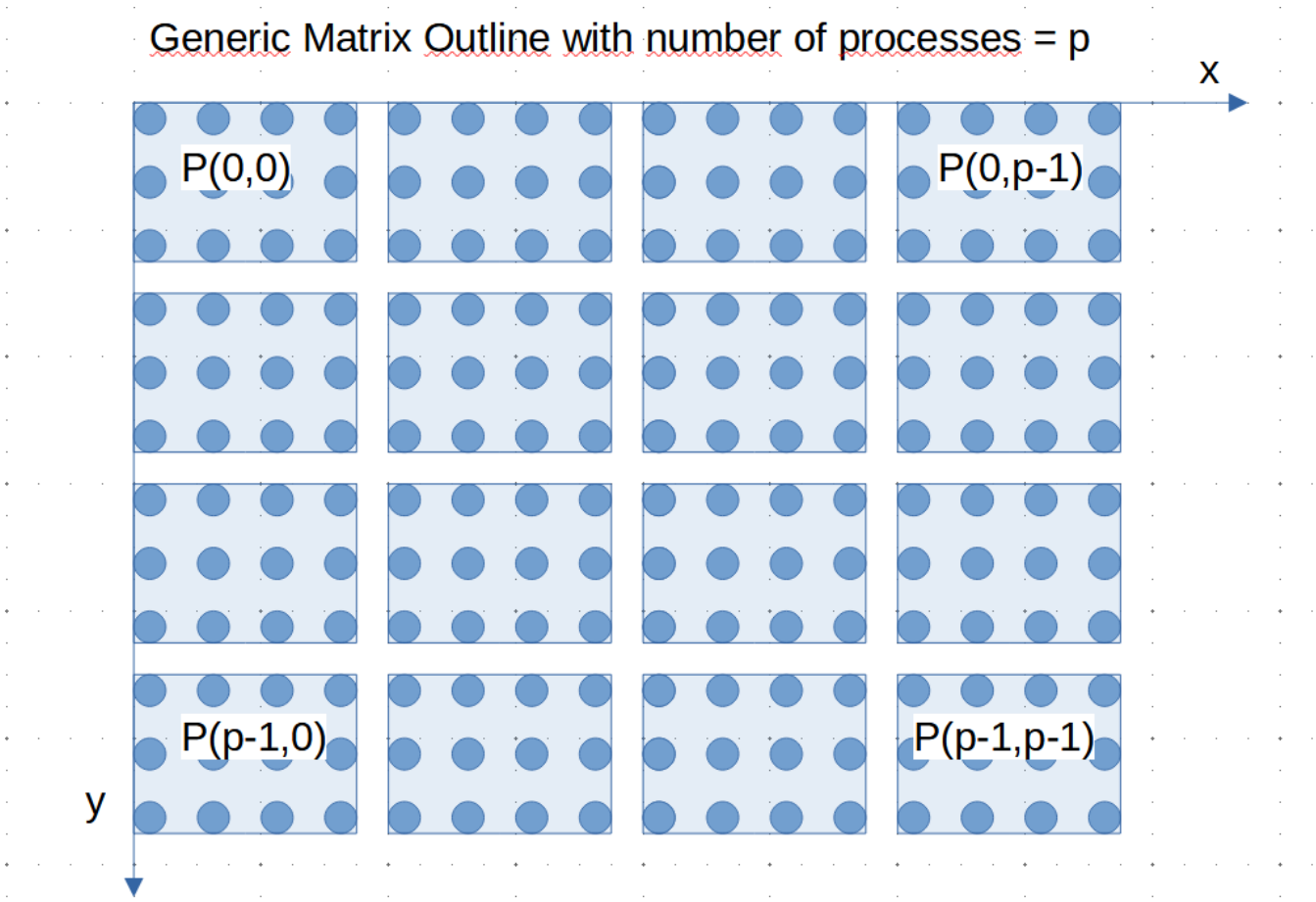
The neighbour situation corresponding to these constraints, is illustrated below:



So the goal is to calculate the state of the new generation, meaning calculating each state of each cell, a lot of times. In the task desciption its stated we should aim for at least 1000 generations. After that we should report the final state.

1.1 MPI and processes

So how does MPI come into play here? We use independent sub-sections of the matrix and assign it to processes, which are by themselves "packed" into a grid:



The calculations for these processes are only partly independent, because on the edges we need communication. However, if the block size for one process becomes big, we can expect a almost linear speed up.

2. Task implementation

2.1 Design ideas

There are two main considerations apart from the algorithms:

```
- data structure for matrix
- data type for cell state
```

We note the task description: "The cell contents should be represented by a short C data type, either char or a bit in a word. Your implementation must be space efficient in the sense of maintaining at most two generations, the current and the next, that is at most two n×n matrices."

Data Structure for Matrix:

(We followed data structure decision principles from [2], Chapter 26 here)

Given we are programming in C++, some possibilities from the std library include:

```
- vector
- array
- list
(...)
```

Vector is very often the correct choice because its feature rich, dynamic and provides data locality. Lets look at the advantages of array and list over a vector, and see if these advantages are important to us:

```
- Vector vs. List: A list might be the better choice if our vector grows
dynamically and the copying of that memory is expensive (list elements
always stay at the same
                    memory location, vector elements don´t). Also list is
better if we insert very often in the middle of the data struct.
                    --> both advantages not relevant in our case

- Vector vs. Array: An array might be the better choice if we want to
allocate on the stack.
                    --> not the case, our matrix size might lead to stack-
overflow
```

Therefore *std::vector* is a good choice.

Data Type for Cell State:

Some possibilities include:

```
- char: 8 Bit
- short int (unsigned): 16 Bit
- a single bit in a machine word, e.g. uint_64: 1 Bit
```

The advantage of the first two is obviously simplicity. However, the advantage of the single bit is space efficiency for sure. A short int uses unnecessary, precious space. It does not have obvious advantages over a char.

Char for the beginning is a good choice, but bitwise operations on a machine word seem like a interesting (but more complicated) alternative.

Do we need classes?

Probably not, but they are convenient for the Cell. Look at this example:

```
class Cell
{
private:
    char state;
```

```
public:
    Cell(char s) : state(s) {}

    void setState(char s) {
        this->state = s;
    }

    void setStateToDead() {
        this->state = 'd';
    }
    void setStateToAlive() {
        this->state = 'a';
    }
};
```

The member functions like getter, setter make things a little easier and safer. Memory concerns: Does the cell need more memory? No! member functions are not stored in the object but the class definition. Both are 1 byte!

Same goes for the Generation class consisting of a Matrix (std::vector). Note that we keep the Object oriented style to a minimum (we strongly agree with the principle "free your functions" from the conference talk *CppCon 2017: Klaus Iglberger "Free Your Functions!"*)

Compiling

As stated in [1], chapter 3.2.4, an MPI program can be compiled with a normal C/C++ compiler and a wrapper mpicc/mpicxx. We use mpicxx (mpic++ is the same thing).

2.2 Classes and functions

2.2.1 Cell (Class)

members: char state; **member-functions:** setState(), setStateAlive(), setStateToDead(), getState(), isAlive(), Constructor

2.2.2 Generation (Class)

members: std::vector (Matrix) **member-functions:** getCell(), printGeneration(), getGeneration(), countAliveNeighbours(), Constructors, countAliveNeighbours(), Parameter Constructor with probability of alive cells and the grid dimensions etc., default constructor, getRowSize(), getColSize()

2.2.3 Functions

There are also a variety of free functions. The most important ones are shortly mentioned in the following.

calculateNextGenSequentially()

input: Generation object (reference) **output:** Generation object **algorithm:** Two for-loops which iterate over the matrix and count for each cell the alive neighbours of the 8 neighbours. Based on the conditions of

the game of life the cell lives or dies. The only slightly tricky part is to "fold" the matrix, so making sure the upper_row, lower_row, left_col, right_col values are correct.

calculateNextGenParallel()

input: Generation object (reference), MPI communicator of the cartesian grid (reference), MPI cell datatype (reference), MPI datatype for the communication of the ghost layers (reference), ghost_layer_size representing the size of the ghost layer on 1 side **output:** Generation object **algorithm:** Basics are similar to sequential version except that each processor calculate the next generation of its own sub-grid together with the ghost layer gathered from the adjacent sub-grids. More in depth details in chapter 3.

calculateNextGenWCollComm() input and output the same as above **algorithm**:

Other than that we have some helper functions like **areGenerationsEqual()**, **countAliveAndDeadCells()**, **printGrid()** (...) which are all used either for debug purposes or inside one of the functions from above.

2.3 Optimizations, Asymptotic complexity

Arguably the two most relevant optimization areas are: - performance (speed) - memory efficiency

Performance:

The performance of the program can be influenced, to a great deal, by the choice of data-structures and algorithms. std::vector is a highly efficient, very robust and tested data-structure from the std::library. We use modern c++ move-semantics and pass-by-reference wherever applicable, to avoid unnecessary, expensive copies of big data.

With regards to the algorithm, it is quite likely that there is room for improvement, especially within the for loops. We do not claim to have found the "fastest" solution to the problem.

Of course, compiler flags are also important for proper optimization. E.g. -O3 -march=native. There is bunch of other things we can try, e.g. using different compilers, optimize linking etc.

Memory Efficiency:

We decided to use a short C data-type *char*. This is space-efficient, but a single bit in a machine word would suffice since we only need true/false (dead/alive). This overhead could have been reduced, but needs more implementation time and careful planning. With memory being a valueable good in HPC, we would argue that the most efficient way of implementing the task is to use a single bit of a machine word.

How many generations are in memory simultaneously? We at most keep two generations, i.e. two MxN sized vectors in the sequential version and one NxN together with one (N+ghost_layer)x(N+ghost_layer) in the parallel algorithm version, in memory.

Asymptotic Complexity, runtime

We iterate over two for loops (in the sequential variant) which would be a time complexity of O(n^2). However, we also need to check for each Cell the 8 neighbours, which adds a constant factor of 8 to the time complexity. In regards to the runtime: The runtime primarily depends on the size of the matrix.

2.4 Exercise 1

The entry point for the first, sequential MPI implementation can be found in src/sequential/main.cpp

The first exercise is mostly the implementation of the basics which were mentioned in Chapters 2.1 -2.3 above, but in the following we demonstrate how we initialized and used our Classes and functions with MPI for the sequential solution.

Step 1: We pass 4 command line arguments to the program: <matrix_dim_N> <matrix_dim_N> <prob_of_life> <number_of_repetitions>

Step 2: We initialize the first Generation based on the input parameters (row/col sizes and probability of life)

```
Generation current_gen{row_size, col_size, prob_of_life};
```

Step 3: We initialize MPI. For that we follow the script [1], page 134 - 136.

```
MPI_Init(&argc, &argv);

int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
```

Step 4: We calculate the next Generation with our sequential function calculateNextGenSequentially() and measure the time with MPI_Wtime

---

```
for (int i = 0; i < number_of_repetitions; i++)
{
    start_time = MPI_Wtime();

    next_gen = calculateNextGenSequentially(current_gen);

    end_time = MPI_Wtime();
    times.push_back(end_time - start_time);

    current_gen = next_gen;
}
```

We calculate it a certain number of repetitions, which is an input to our program as well. The code is only executed by one single process (rank == 0). The times taken by MPI_WTime() are averaged later, and we store them in a std::vector.

Step 5: We finalize MPI and Post Process the data --> output a summary of dead and alive cells, average time of calculation

---

```
int alive_cells{0};
int dead_cells{0};
countAliveAndDeadCells(current_gen, alive_cells, dead_cells);
std::cout << "Last generation \n"
             "Alive Cells: "
          << alive_cells << " Dead Cells: " << dead_cells << std::endl;


std::cout << "Average calculation time per generation: " <<
averageVectorElements(times) << std::endl;
```

We call the executable with *mpirun 1 ./build/sequential <matrix_dim_N> <matrix_dim_M> <prob_of_life> <number_of_repetitions>*

2.5 Exercise 2

The entry point for the second, parallel MPI implementation can be found in src/parallel/main.cpp

Step 1: Setup cartesian Communicator For exercise 2, one of the main tasks is to organize the given processes in a structured way. For that purpose a Cartesian communicator will be used. The task is implemented with inspiration from [1], Chapter 3.2.8, which covers Cartesian communicators. The input arguments to our program stay the same as in Exercise 1, the number of processes is retrieved from the integer given at mpirun <num_of_processes> (...).

The generic structure of the cartesian communicator creation function is:

```
int MPI_Cart_create(MPI_Comm comm, int ndims, const int dims[], const int
periods[], int reorder, MPI_Comm *cartcomm);
```

Input arguments are:

comm: The "base" communicator. MPI_COMM_WORLD in our case. **ndim**: Number of dimensions: 2 in our case for our NxM Matrix **dims**: The size of the dimensions, in our case we choose to distribute the processes evenly in a 2dim grid (see 1.1 for the outline) **Periods**: From [1] we know that "The periods array is a Boolean (0/1) array indicating whether the grid is periodic in the ith dimension", where in our case the grid is indeed periodic in both directions. Therefore, the periods array should be initialized as periods = {1,1}. **Reorder**: We can experiment with the reorder flag, which is used so that MPI reorders the processes in the new communicator in a (potentially) more efficient fashion. **cartcomm**: The cartcomm gets returned as a pointer. We use a reference here.

This leads to our implementation:

```
MPI_Init(&argc, &argv);

int rank, size;
const int ndim = 2;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
assert(0 <= rank && rank < size);

int dims[ndim] = {0, 0};
MPI_Dims_create(size, 2, dims);

MPI_Comm cart_comm;
int periods[ndim] = {1, 1};
int reorder = 1;
MPI_Cart_create(MPI_COMM_WORLD, ndim, dims, periods, reorder, &cart_comm);
assert(cart_comm!=MPI_COMM_NULL);

int coords[ndim];
MPI_Cart_coords(cart_comm, rank, ndim, coords);
```

For setting up Cartesian communicators over an existing communicator of size p (that is, with p MPI processes), the MPI_Dims_create function can be helpful for factoring p into d factors that are close to each other. [1]. We use that to create the dimensions based on the size of MPI_COMM_WORLD (which ultimately just is the number of processes). MPI_Cart_coords is useful to get the cartesian coordinates for each process in the grid.

We defined a debug section with *#ifdef DEBUG* where we can check the cartesian coordinates of each process and its rank. E.g. for 4 processes we get in the console: " Rank 0 has coordinates (0, 0) Rank 1 has coordinates (0, 1) Rank 2 has coordinates (1, 0) Rank 3 has coordinates (1, 1) " which is exactly the outcome what we want.

Create initial Generation depening on weak/strong scaling

If weak scaling is disabled, then the dimensions provided should be divided by the amount of processors in each x-y direction based on the cartesian communicator domain representation. By doing this it defines the sizes that each sub-grid which each processor should handle.

```
Generation current_gen;

if (!weak_scaling_flag)
{
    row_size = row_size / dims[0];
    col_size = col_size / dims[1];
}

current_gen = Generation(row_size, col_size, prob_of_life);
```

Assign workload to processes

The next task is to calculate the new generation in the sub-grid which the appropriate processor handles. We use our function calculateNextGenParallel() here:

```
void calculateNextGenParallel(Generation &&current_gen, MPI_Comm
&cart_comm, MPI_Datatype &MPI_CELL, MPI_Datatype &MPI_COL_PADDING_WGHOST,
int ghost_layer_size)
{

int row_size = current_gen.getRowSize();
int col_size = current_gen.getColSize();
int row_size_whalo = row_size + 2 * halo_layer_size;
int col_size_whalo = col_size + 2 * halo_layer_size;

int dims[2];
int rank, size;
MPI_Comm_rank(cart_comm, &rank);
MPI_Comm_size(cart_comm, &size);

const int ndim = 2;
int coords[ndim];
int periods[ndim] = {1, 1};

MPI_Cart_get(cart_comm, ndim, dims, periods, coords);

    ....
}
```

The function *calculateNextGenParallel()* is called using a movable reference to the current generation, the MPI cartesian communicator, the MPI CELL datatype, the MPI vector datatype for sending/receiving the left & right hand side columns in the sub-grid, as well as the size of the halo layer. There are never more than two generations in memory, since we reuse the current generation as well as a temporary generation containing the halo which go out of scope. Within our function, we use the functions MPI_Comm_rank and MPI_Comm_size to retrieve the rank and size of the function-calling process.

Compare solution against sequential solution

Lastly we want to demonstrate our way of checking the sequential vs the parallel solution:

```
#ifdef DEBUG
    MPI_Barrier(cart_comm);

    if (rank == 0) {
        curr_gen_cells = global_grid; // As the global grid contains the
```

```
global current_gen entries
    }

    MPI_Gatherv(&next_gen.getCell(0,0), row_size * col_size, MPI_CELL,
                &global_grid[0], num_blocks_per_proc, col_padding_block,
recvGlobalGridBlock, 0,
                cart_comm);
    /*
        Print global grid and compare it with sequential version
     */
    if (rank == 0){
        next_gen_cells = global_grid;

        Generation global_next_gen(std::move(next_gen_cells),
global_row_size, global_col_size);
        Generation global_curr_gen(std::move(curr_gen_cells),
global_row_size, global_col_size);

        Generation next_gen_sequential =
calculateNextGenSequentially(global_curr_gen);

        if (!areGenerationsEqual(global_next_gen, next_gen_sequential))
        {
            std::cout << "The sequential and parallel solution are not the
same! Check the /debug folder." << std::endl;
            global_next_gen.printGeneration("parallel_gen");
            next_gen_sequential.printGeneration("sequential_gen");
        }
    }
    MPI_Barrier(cart_comm);
#endif
```

This section of debug code is executed after the calling of the *calculateNextGenParallel()*. First operation is to gather the global next generation grid (as all processes contains all the subgrids of the global grid) using MPI_Gatherv and then calling the *calculateNextGenSequentially()* using the previous global grid to be able to compare the two generations. The comparison of the two generations is then done by calling the *areGenerationsEqual(Gen1, Gen2)* function.

2.6 Exercise 3

For exercise 3 we had 3 different options to choose from on how to implement the MPI_Collective communication to run the game of life using multiple processors.

2.6.1 Option 1

Option 1 was to implement the sending and receiving of the halo elements using non-blocking communication. One advantage which can be noted directly by using this approach is that the implementation does not need to care for the order of sending and receiving the halo elements, as by example shown by the code snippet below:

```
MPI_Isend(send upper halo border to upper process);
MPI_Isend(send below halo border to below process);
MPI_Isend(send left halo border to left process);
MPI_Isend(send right halo border to right process);

MPI_Irecv(recv upper halo border to upper process);
MPI_Irecv(recv below halo border to below process);
MPI_Irecv(recv left halo border to left process);
MPI_Irecv(recv right halo border to right process);

MPI_Waitall();
```

Here the order of sending and receiving of the non-blocking communications are not strictly ordered. It is not a problem if a process successfully receives the right halo before the upper halo, or successfully sends the right halo before the upper halo.

This approach is straight forward and subsequently, by the argument above, less prone to errors, as the manual ordering of communication between the processes is not needed.

2.6.2 MPI datatypes

2.6.2.1 MPI_Cell type

The first datatype which is used in the MPI communication is the MPI_CELL type which represents the already defined Cell class. As the Cell class only contains one *char* field, the MPI_Cell datatype then becomes trivial:

```
/*
    Define the MPI_CELL type for the MPI communication.
*/
MPI_Datatype MPI_CELL;
MPI_Datatype types[1] = {MPI_CHAR};
int blocklength[1] = {1};
MPI_Aint displacement[1] = {0};
MPI_Type_create_struct(1, blocklength, displacement, types, &MPI_CELL);
MPI_Type_commit(&MPI_CELL);
```

2.6.2.2 MPI_COL_PADDING_WHALO type

The second datatype MPI_COL_PADDING_WHALO is a MPI vector type which is used when sending and receiving the left and right halo borders of the subgrid. The datatype describes the padding that is needed of two consecutive elements on different rows.

```
/*
    Define the MPI vector padding datatype for the MPI communication of the
left and right borders (for indexing).
*/
const int halo_layer_size = 1;
int col_size_whalo = col_size + 2 * halo_layer_size; // As the column is
added by one layer on each side

MPI_Datatype MPI_COL_PADDING_WHALO;
MPI_Type_vector(row_size, 1, col_size_whalo, MPI_CELL,
&MPI_COL_PADDING_WHALO);
MPI_Type_commit(&MPI_COL_PADDING_WHALO);
```

2.6.2.3 RecvGlobalGridBlock type

The last datatype is the *recvGlobalGridBlock* that is used for the collective communication Gatherv It is used to collect all subgrids that are handled by each process into a global grid. Here the *recvGlobalGridBlock* should represent the structure of the subgrids in a global grid stored into process 0.

```
/*
    Create MPI datatype for the master receiving all sub-grids and
combining them into the global grid.
*/
if (rank == 0)
{

    global_grid = std::vector<Cell>(global_grid_size, Cell('d'));

    /*
        Create a datatype of the receive block for all the sub-grids
    */
    MPI_Type_create_subarray(ndim, global_sizes_direction,
local_sizes_direction, local_start_indexes,
                             MPI_ORDER_C, MPI_CELL, &recvGridBlock);

    /*
        Resize the recvGridBlock so that when col_size elements have been
recevied/placed (1 block) by the master, start
        to place the next rank/senders elements into the next block.

        For example a 6x6 grid of 9 processors
        |block1|block2|block3|
        |--------------------|
        | A  A | B  B | C  C |
        | A  A | B  B | C  C |
        |--------------------|
        | D  D | E  E | F  F |
        | D  D | E  E | F  F |
```

```
        |--------------------|
        | G  G | H  H | I  I |
        | G  G | H  H | I  I |
        |--------------------|
    */
    int block_size = col_size * sizeof(Cell);
    MPI_Type_create_resized(recvGridBlock, 0, block_size,
&recvGlobalGridBlock);
    MPI_Type_commit(&recvGlobalGridBlock);
}

...

/*
    Collect the global grid using Gatherv
*/
MPI_Gatherv(&current_gen.getCell(0,0), row_size * col_size, MPI_CELL,
            &global_grid[0], num_blocks_per_proc, col_padding_block,
recvGlobalGridBlock, 0,
            cart_comm);
```

2.6.3 Computation of next generation

The following sequence of operations is executed when calculating the next generation

1. Create temporary Generation w. halo
2. Calculate the neighbours and the corners ranks
3. Share borders with neighbouring processors using MPI
4. Calculate and update the current Generation

1. Create temporary Generation w. halo

```
/*
    Create new generation with halo layer and move the current generation
to it:
    | halo   halo   halo |
    | halo curr_gen halo |
    | halo   halo   halo |
*/

std::vector<Cell> current_gen_cells_whalo(row_size_whalo * col_size_whalo,
Cell('d'));

for (int i = 0; i < row_size; i++) {
    int padding_1 = i*col_size;
    int padding_2 = (i+1)*col_size;
    int padding_3 = halo_layer_size + (i+1)*col_size_whalo;
    std::copy(current_gen.getGeneration().begin() + padding_1,  // Left col
starting index
```

```
                current_gen.getGeneration().begin() + padding_2,  // Right
col end index
                current_gen_cells_whalo.begin() + padding_3);     // Starting
index with halo
}

Generation current_gen_whalo(std::move(current_gen_cells_whalo),
row_size_whalo, col_size_whalo);
```

Here we create a temporary Generation object called *current_gen_whalo* containing the current generation together with the halo. This is done mainly to simplify the calculation of the next generation in the latter section of the code as well as having a central object to store the halo in.

2. Calculate the neighbours and the corners ranks

```
// Find the ranks of the neighbours
int upper_left_rank, upper_rank, upper_right_rank, left_rank, right_rank,
lower_left_rank, lower_rank, lower_right_rank;

MPI_Cart_shift(cart_comm, 0, 1, &upper_rank, &lower_rank);
MPI_Cart_shift(cart_comm, 1, 1, &left_rank, &right_rank);

// Find ranks of the corners
int upper_left_coords[2] = {coords[0] - 1, coords[1] - 1};
int upper_right_coords[2] = {coords[0] - 1, coords[1] + 1};
int lower_left_coords[2] = {coords[0] + 1, coords[1] - 1};
int lower_right_coords[2] = {coords[0] + 1, coords[1] + 1};

MPI_Cart_rank(cart_comm, upper_left_coords, &upper_left_rank);
MPI_Cart_rank(cart_comm, upper_right_coords, &upper_right_rank);
MPI_Cart_rank(cart_comm, lower_left_coords, &lower_left_rank);
MPI_Cart_rank(cart_comm, lower_right_coords, &lower_right_rank);
```

Here the cartesian communicator operations *MPI_Cart_shift* and *MPI_Cart_rank* finds the ranks of the neighbouring processes. *MPI_Cart_shift* is in this case used to find the neighbouring processes on the vertical and horitzontal axes, and *MPI_Cart_rank* is used to find the corner processes.

3. Share borders with neighbouring processes using MPI

```
MPI_Request send_request[8], recv_request[8];
int tag[8] = {0, 1, 2, 3, 4, 5, 6, 7};

// Send (only the "inner" matrix of current_gen_whalo)
MPI_Isend(&current_gen_whalo.getCell(1,1), col_size, MPI_CELL, upper_rank,
tag[0], cart_comm, &send_request[0]); // Upper border
```

```
MPI_Isend(&current_gen_whalo.getCell(row_size_whalo-2,1), col_size,
MPI_CELL, lower_rank, tag[1], cart_comm, &send_request[1]); // Lower border
MPI_Isend(&current_gen_whalo.getCell(1,1), 1, MPI_COL_PADDING_WHALO,
left_rank, tag[2], cart_comm, &send_request[2]); // Left border
MPI_Isend(&current_gen_whalo.getCell(1,col_size_whalo-2), 1,
MPI_COL_PADDING_WHALO, right_rank, tag[3], cart_comm, &send_request[3]); //
Right border

MPI_Isend(&current_gen_whalo.getCell(1,1), 1, MPI_CELL, upper_left_rank,
tag[4], cart_comm, &send_request[4]); // Upper-left corner
MPI_Isend(&current_gen_whalo.getCell(1,col_size_whalo-2), 1, MPI_CELL,
upper_right_rank, tag[5], cart_comm, &send_request[5]); // Upper-right
corner
MPI_Isend(&current_gen_whalo.getCell(row_size_whalo-2,1), 1, MPI_CELL,
lower_left_rank, tag[6], cart_comm, &send_request[6]); // Lower-left corner
MPI_Isend(&current_gen_whalo.getCell(row_size_whalo-2,col_size_whalo-2), 1,
MPI_CELL, lower_right_rank, tag[7], cart_comm, &send_request[7]); // Lower-
right corner

// Receive (address of the halo layer indexes)
MPI_Irecv(&current_gen_whalo.getCell(0,1), col_size, MPI_CELL, upper_rank,
tag[1], cart_comm, &recv_request[0]); // Upper border
MPI_Irecv(&current_gen_whalo.getCell(row_size_whalo-1, 1), col_size,
MPI_CELL, lower_rank, tag[0], cart_comm, &recv_request[1]); // Lower border
MPI_Irecv(&current_gen_whalo.getCell(1,0), 1, MPI_COL_PADDING_WHALO,
left_rank, tag[3], cart_comm, &recv_request[2]); // Left border
MPI_Irecv(&current_gen_whalo.getCell(1,col_size_whalo-1), 1,
MPI_COL_PADDING_WHALO, right_rank, tag[2], cart_comm, &recv_request[3]); //
Right border

MPI_Irecv(&current_gen_whalo.getCell(0,0), 1, MPI_CELL, upper_left_rank,
tag[7], cart_comm, &recv_request[4]); // Upper-left corner
MPI_Irecv(&current_gen_whalo.getCell(0,col_size_whalo-1), 1, MPI_CELL,
upper_right_rank, tag[6], cart_comm, &recv_request[5]); // Upper-right
corner
MPI_Irecv(&current_gen_whalo.getCell(row_size_whalo-1,0), 1, MPI_CELL,
lower_left_rank, tag[5], cart_comm, &recv_request[6]); // Lower-left corner
MPI_Irecv(&current_gen_whalo.getCell(row_size_whalo-1, col_size_whalo-1),
1, MPI_CELL, lower_right_rank, tag[4], cart_comm, &recv_request[7]); //
Lower-right corner

MPI_Waitall(8, recv_request, MPI_STATUS_IGNORE);
```

Here the implementation utilizes non-blocking communication, as explained earlier, where each process shares its border elements with its neighbouring processes. Note that it uses the already defined Generation that contains the halos already so that there is no need to dynamically define additional send and receive buffers when sending and receiving elements to and from other processes.

Additionally, the tags are ONLY necessary if a specific process is neighbouring itself, as you then need to distinguish which of its borders it should send and receive.

4. Calculate and update the current Generation

```
// Reuse the Generation object current_gen which is already defined and
store the next generation in it.
std::vector<Cell> next_gen_cells(row_size * col_size, Cell('d'));
current_gen.setGenerationAndProperties(std::move(next_gen_cells), row_size,
col_size);

// Iterate over the inner grid of the halo layer generation and count
neighbours. Store the iterated cells into the next_gen
for (int i = 1; i < row_size_whalo - 1; ++i)
{
    int upper_row_idx = i - 1;
    int lower_row_idx = i + 1;

    for (int j = 1; j < col_size_whalo - 1; ++j)
    {
        int left_col_idx = j - 1;
        int right_col_idx = j + 1;
        int alive_neighbours_count =
current_gen_whalo.countAliveNeighbours(left_col_idx, right_col_idx,
lower_row_idx, upper_row_idx, j, i);
        bool isAlive = current_gen_whalo.getCell(i,j).isAlive();

        if (!isAlive && alive_neighbours_count == 3)
        {
            current_gen.getCell(i-halo_layer_size,j-
halo_layer_size).setStateToAlive();
        }
        else if (isAlive && (alive_neighbours_count == 3 ||
alive_neighbours_count == 2))
        {
            current_gen.getCell(i-halo_layer_size,j-
halo_layer_size).setStateToAlive();
        }
    }
}

return current_gen;
```

Using the similar loop as the sequential version, it updates the current_gen by checking the current_gen_whalo Generation. At first current_gen is reset to a default state with only dead cells. The loop then iterates over the elements of the inner grid in current_gen_whalo, where it calls the function *countAliveNeighbours* on each element and subsequently checks whether any elements in the current_gen should be set to alive.

This shows that by defining the current_gen_whalo, the loop above can iterate over the whole local grid and utilize the *countAliveNeighbours* without any additional overhead which would otherwise be needed if the implementation uses independent buffers to store the halo.

2.7 Exercise 4

In the final exercise, we want to look at a different approach of communication by using the neighborhood collective. For this we implementated a second function called *calculateNextGenParallelWCollNeighbourComm*. In this section we will only touch on the communication part of this method, as the rest is equivalent to the *calculateNextGenParallel* from section 2.6.

1. A New Communicator

To use Neighborhood methods in MPI we require a distributed Graph communicator. We can specify the edges of the graph by getting the neighboring ranks in the cartesian communicator.

```
MPI_Comm dist_graph_comm;

int sources[] = {
    upper_rank,
    lower_rank,
    left_rank,
    right_rank,
    upper_left_rank,
    upper_right_rank,
    lower_left_rank,
    lower_right_rank,
};

MPI_Dist_graph_create_adjacent(cart_comm, 8, sources, MPI_UNWEIGHTED, 8,
                               sources, MPI_UNWEIGHTED, MPI_INFO_NULL,
false, &dist_graph_comm);
```

The ordering of the sources starts from the vertical axis i.e. upper to lower and then horizontal i.e. left to right.

2. The Neighborhood Strikes Back

For the Neighborhood communication we used *MPI_Neighbor_alltoallw* method. This requires us to provide displacements for the buffer instead of seperate buffers as in section 2.6.

```
Cell *buffer_start = &current_gen_whalo.getCell(0, 0);
const MPI_Aint send_displs[] = {
    &current_gen_whalo.getCell(1, 1) - buffer_start,
    &current_gen_whalo.getCell(row_size_whalo - 2, 1) - buffer_start,
    &current_gen_whalo.getCell(1, 1) - buffer_start,
    &current_gen_whalo.getCell(1, col_size_whalo - 2) - buffer_start,
    &current_gen_whalo.getCell(1, 1) - buffer_start,
    &current_gen_whalo.getCell(1, col_size_whalo - 2) - buffer_start,
    &current_gen_whalo.getCell(row_size_whalo - 2, 1) - buffer_start,
    &current_gen_whalo.getCell(row_size_whalo - 2, col_size_whalo - 2) -
```

```
buffer_start,
};
```

The only difficulty in the implementation was to assign the correct receive buffer to the corresponding process, since we cannot use tags in this communication method.

```
MPI_Neighbor_alltoallw(buffer_start, send_counts, send_displs, send_types,
                       buffer_start, send_counts, recv_displs, send_types,
dist_graph_comm);
```

---

3. Return Of The Generation

The remaining part of the function is equivalent to everything done in 2.6, so we calculate the new generation and return it.

3. Benchmarking

The benchmarking was divided into 3 steps, following the order that was introduced in the assignment. All benchmarks were ran with the alive/death probability of 0.4.

Raw benchmarking data can be found in /benchmarking folder.

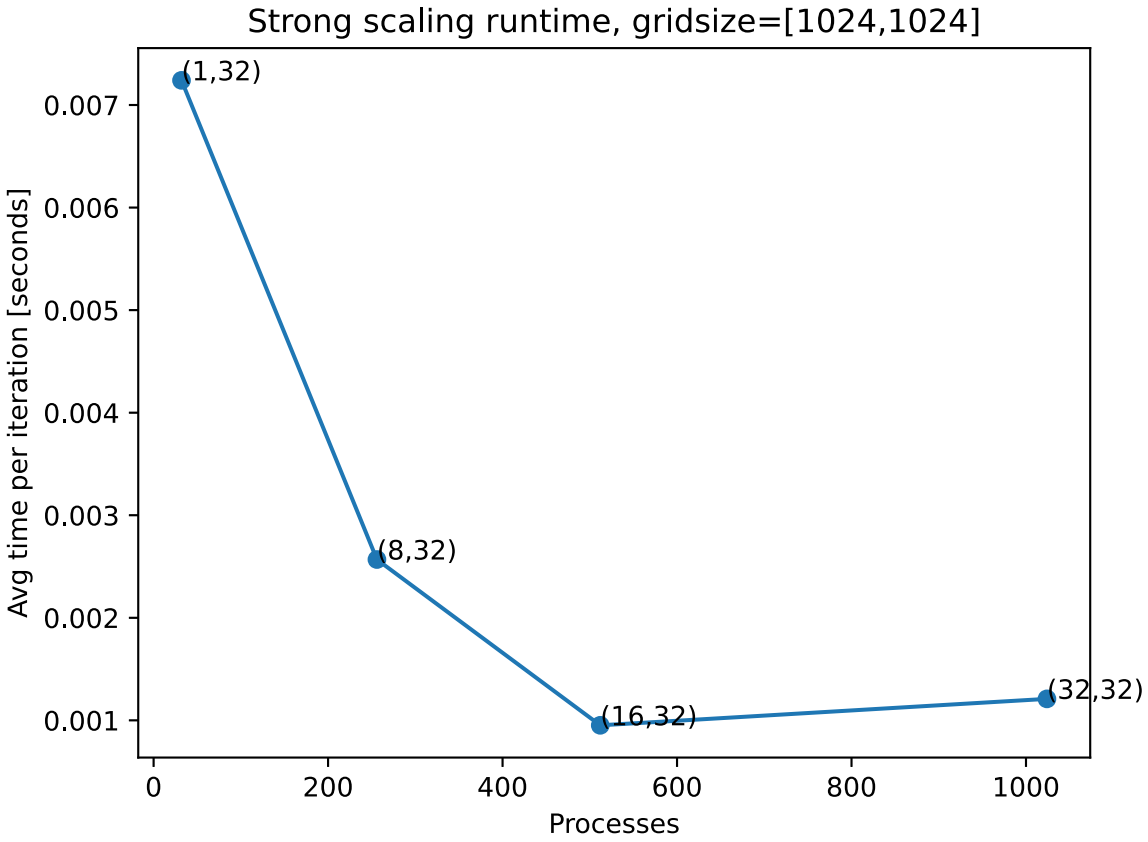
3.1 Sequential

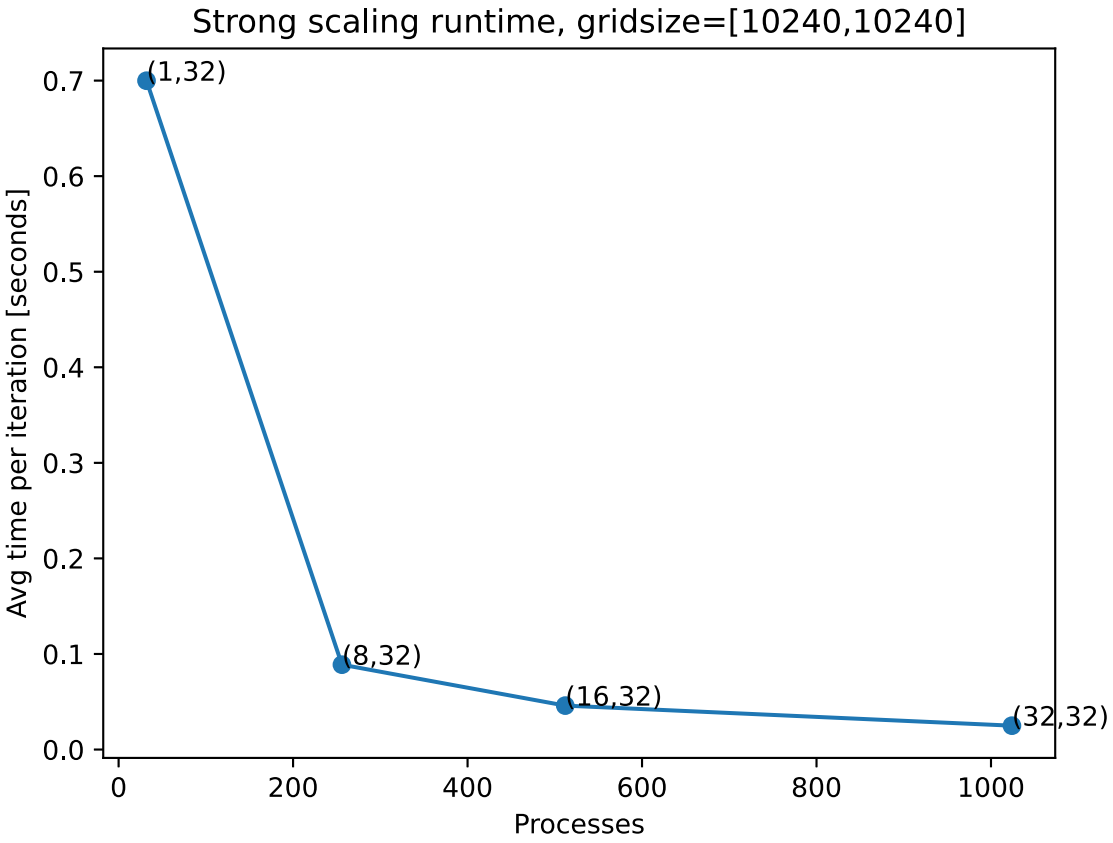
Grid sizes	**1024x1024**	**10240x10240**
Seconds	0.190515	19.0267

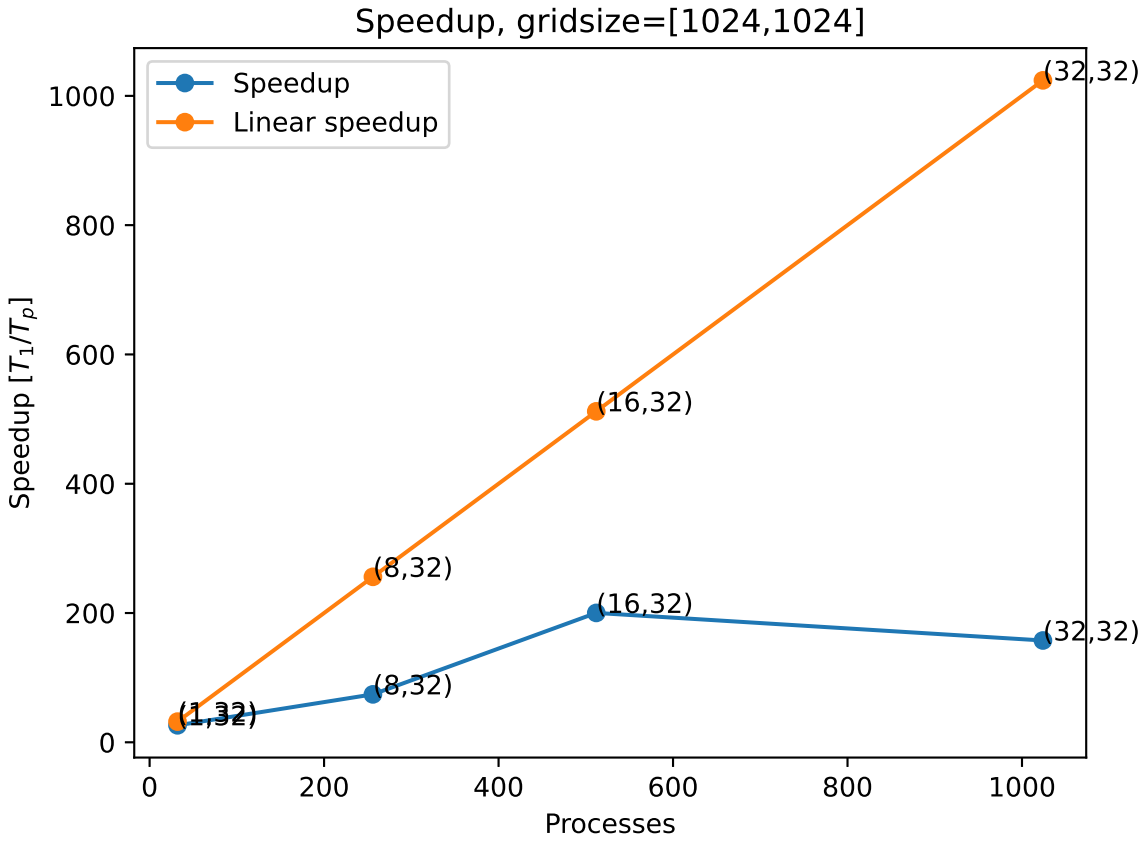
3.2 Parallel

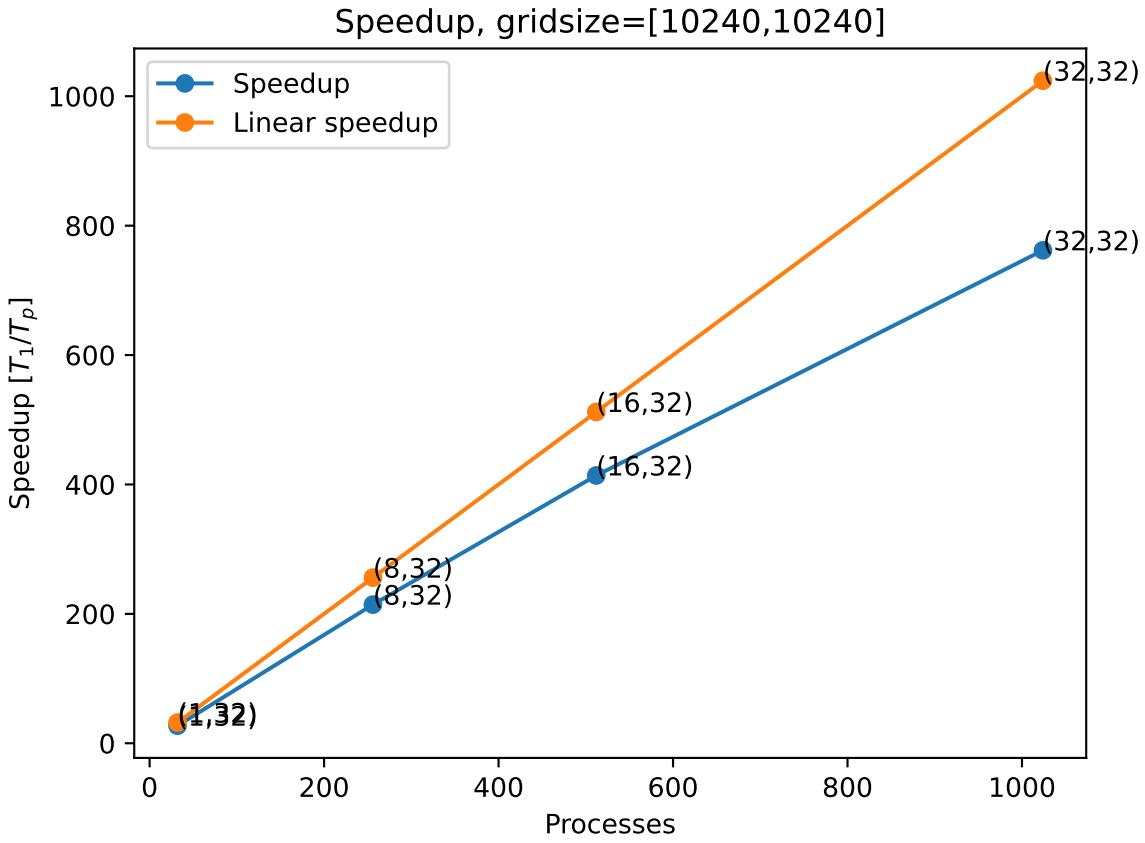
The benchmarking of the P2P parallel application was divided into two subcategories, one using the strong scaling and the other using the weak scaling.

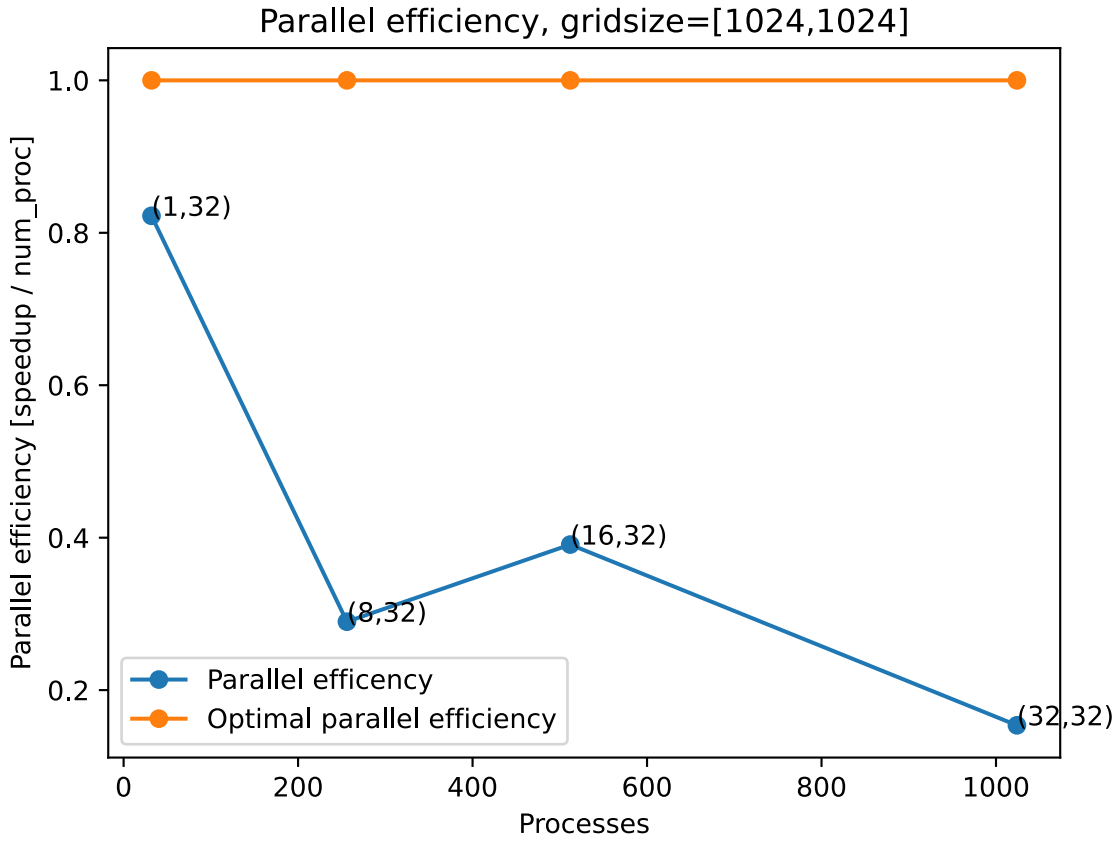
3.2.1 Strong scaling

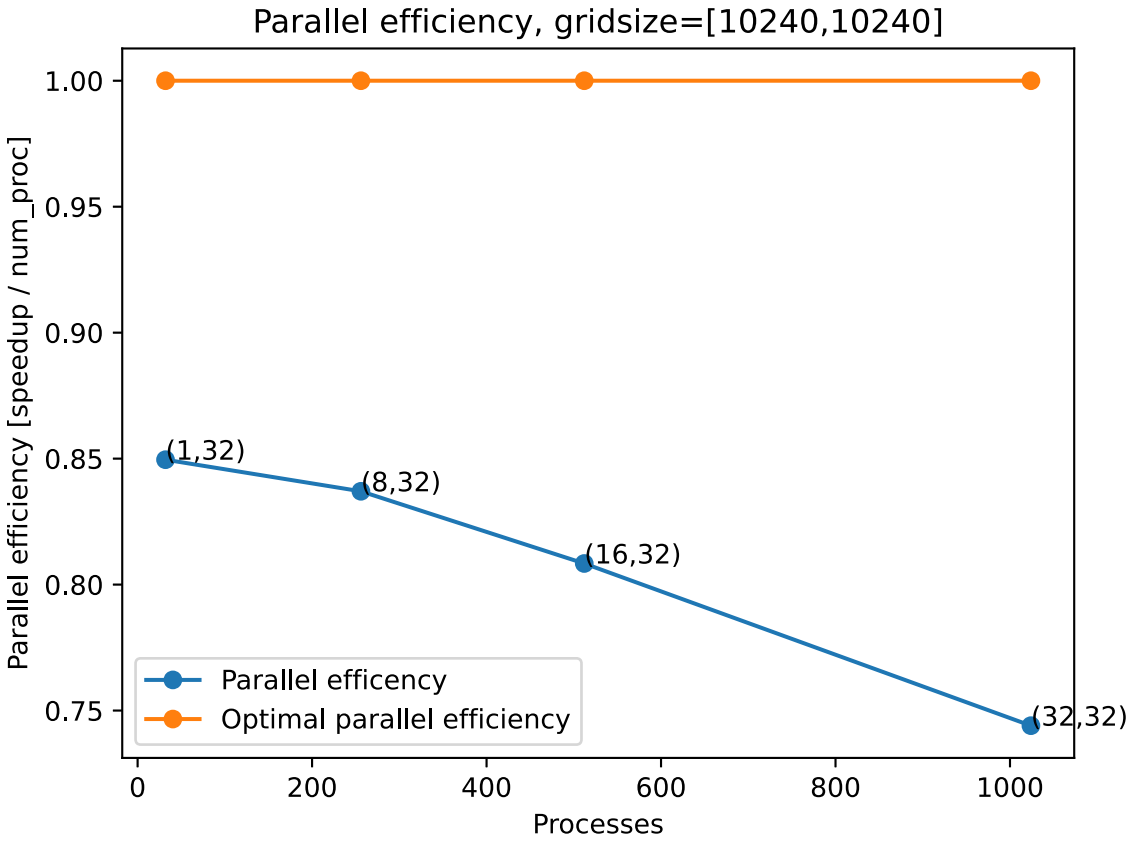

Strong scaling runtime, gridsize=[1024,1024]
Avg time per iteration [seconds]
0.007
0.006
0.005
0.004
0.003
0.002
0.001
(1,32)
(8,32)
(16,32)
(32,32)
0
200
400
600
800
1000
Processes


Strong scaling runtime, gridsize=[10240,10240]
Avg time per iteration [seconds]
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
(1,32)
(8,32)
(16,32)
(32,32)
0
200
400
600
800
1000
Processes

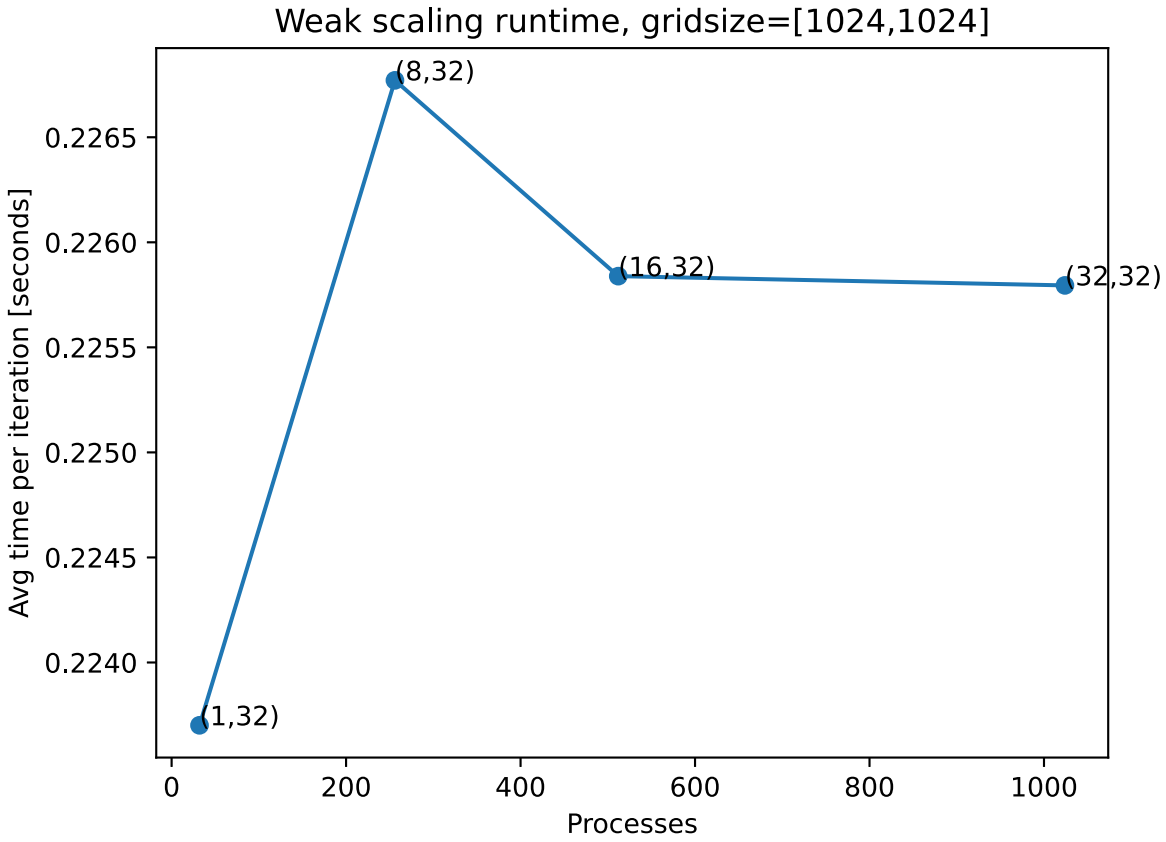

Speedup, gridsize=[1024,1024]
Speedup
Linear speedup
Speedup [T_1/T_p]
Processes
0
200
400
600
800
1000
(32,32)
(16,32)
(8,32)
(16,32)
(32,32)
(8,32)


Speedup, gridsize=[10240,10240]
Speedup
Linear speedup
Speedup [T_1/T_p]
Processes
0
200
400
600
800
1000
(32,32)
(32,32)
(16,32)
(16,32)
(8,32)
(8,32)



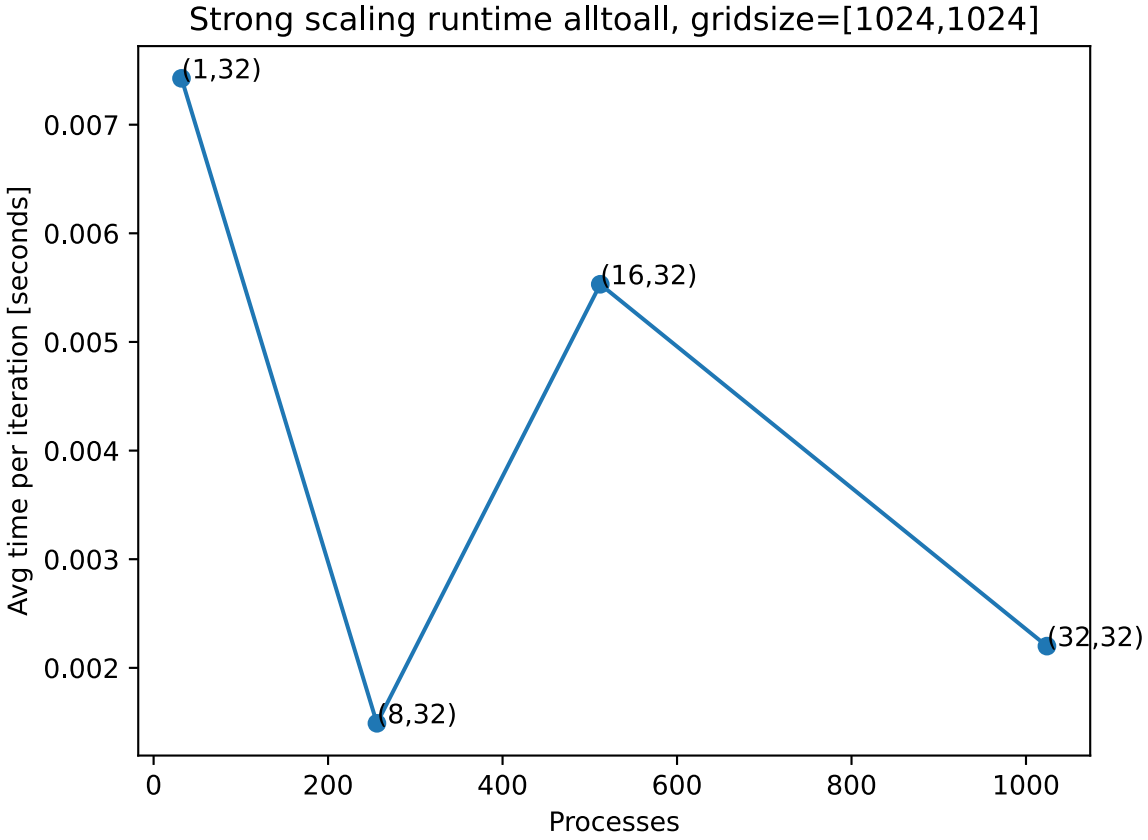


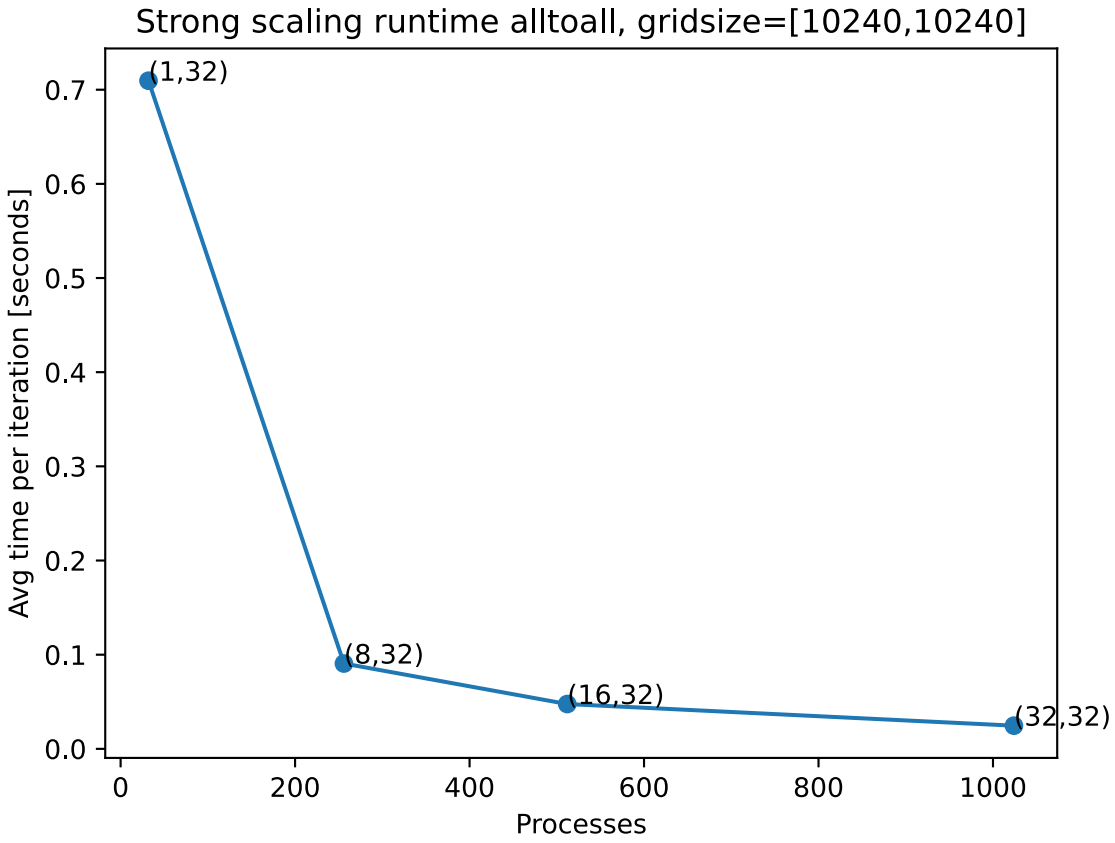
3.2.2 Weak scaling

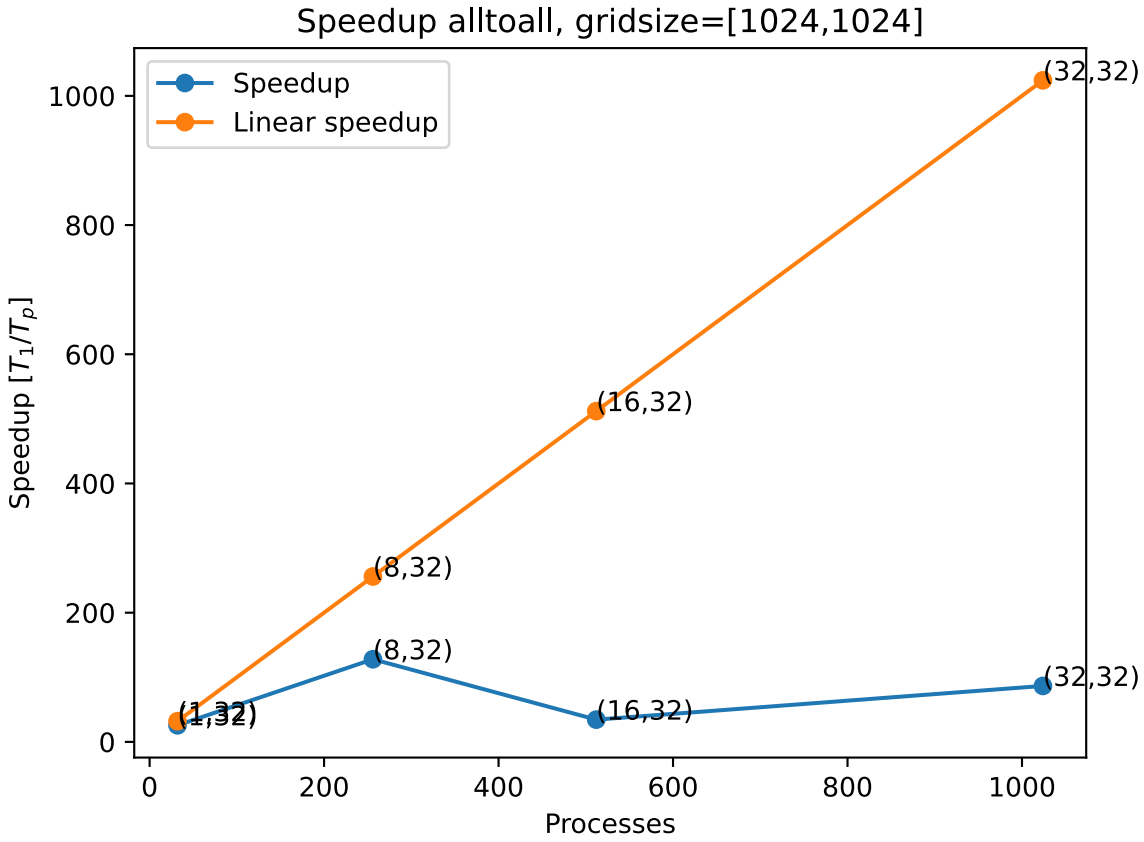


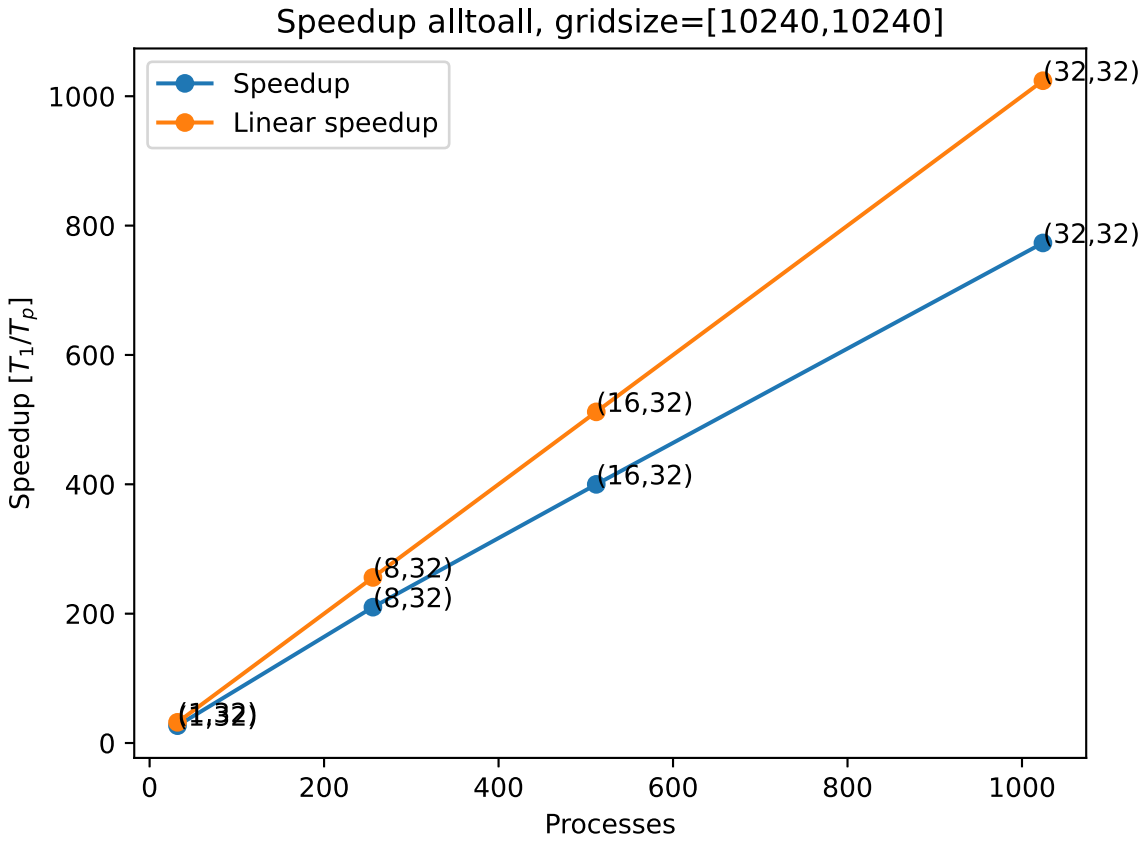
3.3 Parallel with AllToAll Neighbor collective communication

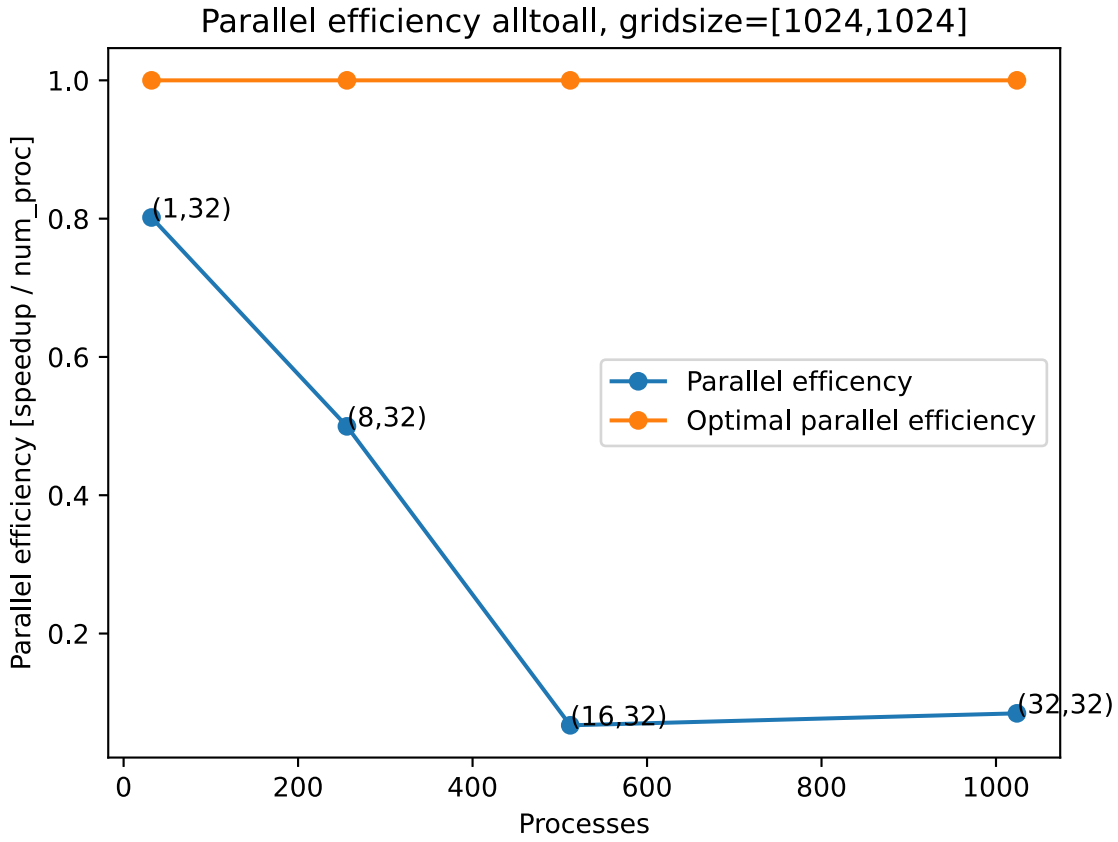
3.3.1 Strong scaling alltoall Neighbor

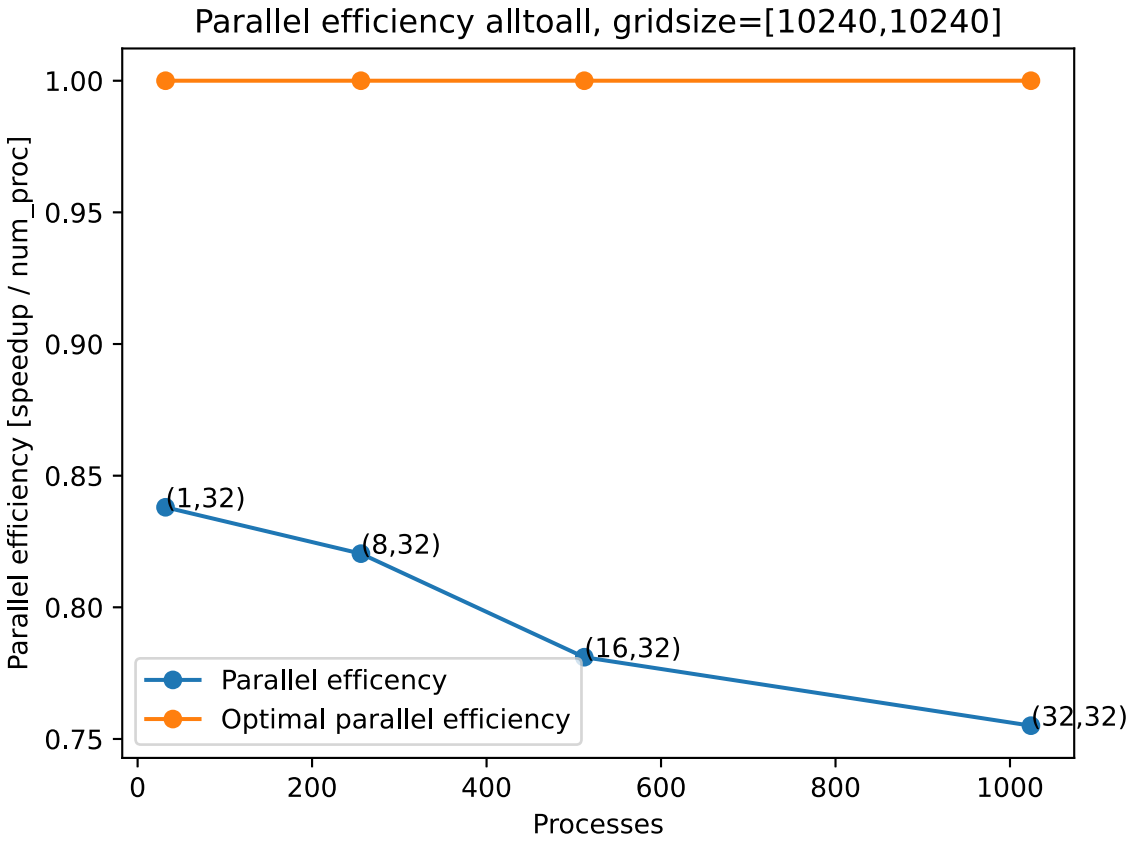

Strong scaling runtime alltoall, gridsize=[1024,1024]
Avg time per iteration [seconds]
0.002
0.003
0.004
0.005
0.006
0.007
(1,32)
(8,32)
(16,32)
(32,32)
0
200
400
600
800
1000
Processes


Strong scaling runtime alltoall, gridsize=[10240,10240]
Avg time per iteration [seconds]
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
(1,32)
(8,32)
(16,32)
(32,32)
0
200
400
600
800
1000
Processes

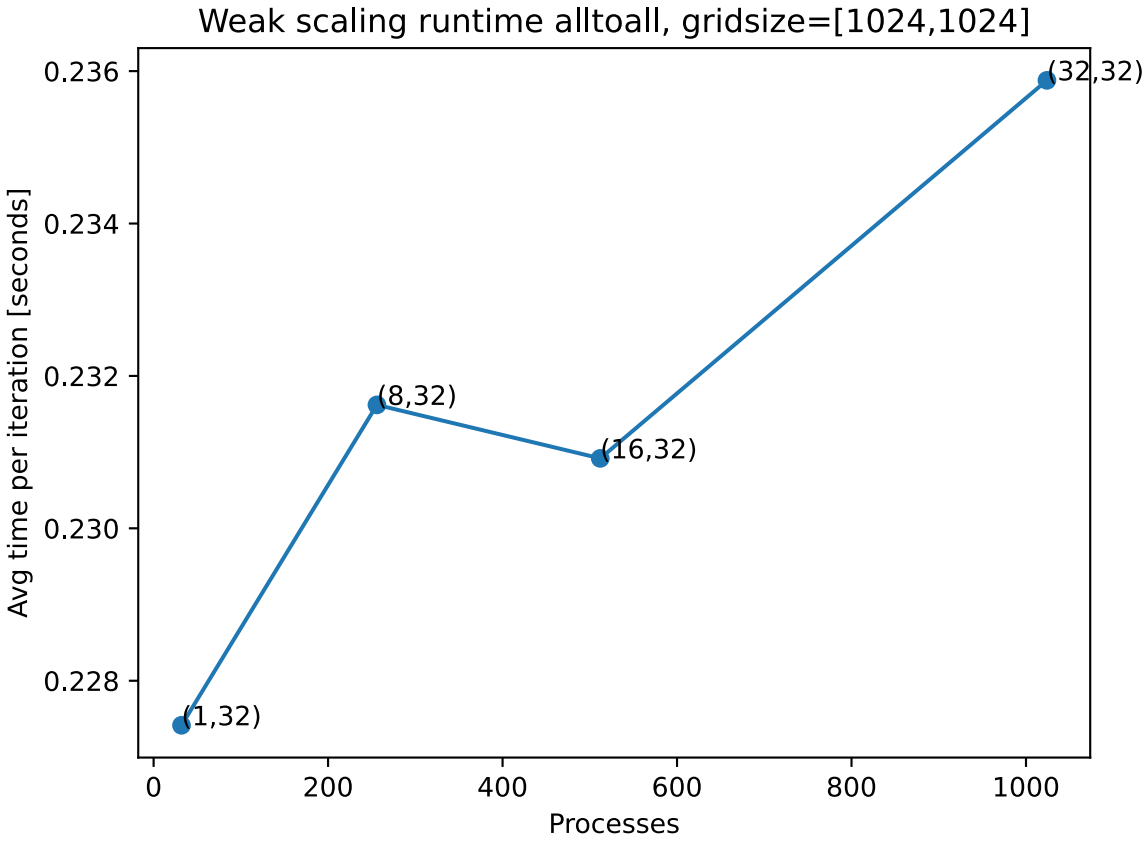

Speedup alltoall, gridsize=[1024,1024]
Speedup
Linear speedup
Speedup [T_1/T_p]
Processes
0
200
400
600
800
1000
(32,32)
(16,32)
(8,32)
(8,32)
(16,32)
(32,32)


Speedup alltoall, gridsize=[10240,10240]
Speedup
Linear speedup
Speedup [T_1/T_p]
Processes
0
200
400
600
800
1000
(32,32)
(32,32)
(16,32)
(16,32)
(8,32)
(8,32)





3.3.2 Weak scaling alltoall Neighbor



4. Conclusion and Interpretation of results

Based on the benchmarking section, we can conclude the following things:

1. The speedup of the larger grid size version (10240^2) is close to the linear speedup. This shows that the both the implementation as well as the 9-stencil grid setup seems to be a structure that is parallelizable in an optimal way. The code can be optimized for sure, which can result in an almost linear speedup for large matrizes.
2. The parallel efficiency for the larger grid sizes (10240^2) of approximately 80%, which is what the speedup also shows, is optimal.
3. By looking at the speedup and parallel efficiency plots with the different grid sizes of 1024^2 and 10240^2, it can be seen that when the grid size is larger, the more the application follows the most optimal performance that you can get from the hardware. This means that the performance gets influenced by the parallel overhead for smaller grids in comparison to when the grid sizes are larger.
4. Looking at the plots for the weak scaling runtimes, it can be seen that the runtimes stays approximately the same even when increasing the amount of processes. This has its explanation by the fact that the each process handles a grid of size 1024^2 independently of how many other processes are ran concurrently.
5. Observing the difference in runtimes, speedup and parallel efficiency of both the non-blocking communication and the Alltoall neighbor collective communication, it can be seen that they differ by a very small margin. The only larger difference is the speedup when the grid size is 1024^2.

Literature

[1]: Lectures on Parallel Computing, SS2023, Jesper Larsson Träff

[2]: C++ Das Umfassende Handbuch, Rheinwerk Computing, 2.Auflage 2020