UNIVERSITY
OF AMSTERDAM

High Performance Computing and Big Data
Assignment 3: MPI and OpenMP

High Performance Computing and Big Data 2024

Group: 5
Team members: Iason Christofilakis, Lars Meijer
Email: iason.christofilakis@student.uva.nl, lars.meijer@student.uva.nl

February 2, 2024

1 Implementation

In this section, we present the implementation details of the parallelized Game of Life simulation using MPI and OpenMP. The implementation focuses on distributing the computational workload among multiple MPI processes and leveraging OpenMP for parallelism within each process.

The chosen implementation path leverages a parallelized approach to simulate Conway's Game of Life using a combination of MPI (Message Passing Interface) and OpenMP (Open Multi-Processing) for distributed and shared-memory parallelism, respectively. The code begins by initializing MPI and determining the rank and size of each process within the MPI communicator. The board is divided horizontally into blocks, with each process responsible for a distinct segment. The parallelization is achieved by employing OpenMP directives, specifically in the computation of cell states during each iteration. Notably, the implementation demonstrates efficient communication between neighboring processes using MPI, ensuring synchronization and exchange of boundary data. Additionally, the code incorporates verification checks at specified iterations, comparing the population of a predefined pattern against expected values. The implementation not only achieves the parallelization of the Game of Life algorithm but also includes robust error handling and execution time measurements. The choice of this implementation path aligns with the goal of harnessing both MPI and OpenMP to attain a high-performance parallel simulation of the Game of Life across multiple processes and threads.

1.1 Header and Library Inclusions

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <mpi.h>
#include <omp.h>
#include <stdbool.h>
```

Listing 1: Header and Library Inclusions

These lines include the necessary standard C libraries and MPI (Message Passing Interface) library for parallel processing.

1.2 Constants and Definitions

```
// Constants defining the size of the board and maximum iterations
#define BOARD_SIZE 3000
#define MAX_ITERATIONS 5000

// Constants defining the starting positions of various patterns on the board
#define GROWER_START_ROW 1500
#define GROWER_START_COL 1500
#define GLIDER_START_ROW 1500
#define GLIDER_START_COL 1500
#define BEEHIVE_START_ROW 1500
#define BEEHIVE_START_COL 1500

// Population expectations for verification at specific iterations
#define GROWER_POPULATION_GEN10 49
#define GROWER_POPULATION_GEN100 138
#define GLIDER_POPULATION 5
#define BEEHIVE_POPULATION 6

```

```
// Debug printing controls
#define DEBUG_PRINT_RANK_INFO 0
#define DEBUG_PRINT_REGION 0
#define DEBUG_RUN_VERIFICATION_CHECKS 0
#define DEBUG_PRINT_TOTAL_POPULATION 0
#define DEBUG_PRINT_EXECUTION_TIME 1
```

Listing 2: Constants and Definitions

Here, we define constants for the board size, maximum iterations, starting positions of patterns, and debug controls.

1.3 Pattern Definitions

```
// Include patterns
#include "grower.h"
#include "glider.h"
#include "beehive.h"
```

Listing 3: Pattern Definitions

Imports the patterns (grower, glider, beehive) that will be placed on the board.

1.4 Utility Macros and Functions

```
// Macro to convert 2D indices to a 1D index for a flattened array
#define INDEX(row, col) ((row) * BOARD_SIZE + (col))

// Function to convert local coordinates to global coordinates
void local_coords_to_global(int *global_row, int *global_col, int row, int col, int
    startRow, int startCol) {
    *global_row = row + startRow;
    *global_col = col + startCol;
}

// Function to print a region of the board
void print_region(uint8_t *board, int startRow, int startCol, int height, int width)
    {
    for (int i = startRow; i < startRow + height; ++i) {
        for (int j = startCol; j < startCol + width; ++j) {
            printf("%c", board[INDEX(i, j)] ? '#' : '^');
        }
        printf("\n");
    }
}
```

Listing 4: Utility Macros and Functions

Defines a macro for converting 2D indices to a 1D index and functions for converting local coordinates to global coordinates and printing a region of the board.

1.5 Initialization and Population Counting

```
// Function to initialize the local board with a specific pattern
void initialize_local_board(uint8_t *board, int block_start, int block_size) {
    // Fill the local board with the grower pattern
    // Only the part of the pattern that is within the local board boundaries will be
     copied
    for (int i = 0; i < block_size + 2; ++i) {
```

```
        for (int j = 0; j < BOARD_SIZE; ++j) {
            if (i == 0 || i == block_size + 1) {
                // Set boundary rows to 0
                board[INDEX(i, j)] = 0;
                continue;
            }
            int row_global, col_global;
            local_coords_to_global(&row_global, &col_global, i, j, block_start, 0);
            if (row_global >= GROWER_START_ROW && row_global < GROWER_START_ROW +
    GROWER_HEIGHT &&
                col_global >= GROWER_START_COL && col_global < GROWER_START_COL +
    GROWER_WIDTH) {
                // Copy the grower pattern to the local board
                board[INDEX(i, j)] = grower[row_global - GROWER_START_ROW][col_global
     - GROWER_START_COL];
            } else
                // Set cells outside the grower pattern to 0
                board[INDEX(i, j)] = 0;
        }
    }
}

// Function to calculate the population of the local board
int local_board_population(uint8_t *board, int block_size) {
    int population = 0;
    for (int i = 1; i <= block_size; ++i) {
        for (int j = 0; j < BOARD_SIZE; ++j) {
            population += board[INDEX(i, j)];
        }
    }
    return population;
}
```

Listing 5: Initialization and Population Counting

Functions for initializing the local board with the grower pattern and counting the population of the local board.

1.6 Game of Life Rules

```
// Function to apply the Game of Life rules to a cell
uint8_t apply_rules(int current_state, int neighbors) {
    if (current_state == 1) {
        // Cell is alive
        if (neighbors < 2 || neighbors > 3) {
            // Rule 1 and 3: Cell dies
            return 0;
        } else {
            // Rule 2: Cell survives
            return 1;
        }
    } else {
        // Cell is dead
        if (neighbors == 3) {
            // Rule 4: Cell becomes alive
            return 1;
        } else {
            // Cell remains dead
            return 0;
        }
```

```
    }
}
```

Listing 6: Game of Life Rules

Defines the rules for the Game of Life, determining whether a cell survives, dies, or becomes alive (as described here).

1.7 Game of Life Iterations

```
// Function to run one iteration of the Game of Life for the local region of the board
void run_game_of_life(uint8_t *current, uint8_t *next, int block_size) {
#pragma omp parallel for collapse(2)
    for (int i = 1; i <= block_size; ++i) {
        for (int j = 0; j < BOARD_SIZE; ++j) {
            // Count neighbors for each cell
            int neighbors = 0;
            for (int ni = -1; ni <= 1; ++ni) {
                for (int nj = -1; nj <= 1; ++nj) {
                    if (ni == 0 && nj == 0) continue;

                    int ni_ = i + ni;

                    // Check if the neighbor is within the local board boundaries
                    int nj_ = j + nj;

                    // Check if the neighbor is within the board boundaries
                    if (nj_ >= 0 && nj_ < BOARD_SIZE) {
                        neighbors += current[INDEX(ni_, nj_)];
                    }
                }
            }

            // Apply Game of Life rules
            next[INDEX(i, j)] = apply_rules(current[INDEX(i, j)], neighbors);
        }
    }
}
```

Listing 7: Game of Life Iterations

Includes the parallelized code for running one iteration of the Game of Life for the local region of the board using OpenMP.

1.8 Global Population Calculation

```
// Function to calculate the population of the entire board across all processes
int total_board_population(uint8_t *local_board, int block_size) {
    int local_population = local_board_population(local_board, block_size);

    // Use MPI_Allreduce to perform a reduction operation across all processes
    int total_population;
    if (MPI_Allreduce(&local_population, &total_population, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD) != MPI_SUCCESS) {
        fprintf(stderr, "Error in MPI_Allreduce operation.\n");
        exit(EXIT_FAILURE);
    }

```

```
    return total_population;
}
```

Listing 8: Global Population Calculation

Calculates the population of the entire board across all processes using MPI_Allreduce.

1.9 Verification Checks

```
// Function to perform verification checks at specific iterations
void run_verification(uint8_t *local_board, int iter, int block_size) {
    // Check total population for specific iterations
    if (iter == 10) {
        int total_population = total_board_population(local_board, block_size);
        printf("Generation 10: Grower pattern population = %d\n", total_population);
        if (total_population != GROWER_POPULATION_GEN10) {
            printf("ERROR: Incorrect population for generation 10\n");
            exit(1);
        }
    }

    if (iter == 100) {
        int total_population = total_board_population(local_board, block_size);
        printf("Generation 100: Grower pattern population = %d\n", total_population);
        if (total_population != GROWER_POPULATION_GEN100) {
            printf("ERROR: Incorrect population for generation 100\n");
            exit(1);
        }
    }
}
```

Listing 9: Verification Checks

Includes checks for total population at specific iterations and terminates with an error if the population is incorrect.

1.10 Main Function and MPI Initialization

```
int main(int argc, char *argv[]) {
    int rank, size;

    // Start up MPI
    if (MPI_Init(&argc, &argv) != MPI_SUCCESS) {
        fprintf(stderr, "Error initializing MPI.\n");
        exit(EXIT_FAILURE);
    }
    if (MPI_Comm_rank(MPI_COMM_WORLD, &rank) != MPI_SUCCESS) {
        fprintf(stderr, "Error in MPI_Comm_rank.\n");
        exit(EXIT_FAILURE);
    }
    if (MPI_Comm_size(MPI_COMM_WORLD, &size) != MPI_SUCCESS) {
        fprintf(stderr, "Error in MPI_Comm_size.\n");
        exit(EXIT_FAILURE);
    }

    const bool am_master = 0 == rank;

    // Divide the board into blocks for parallel processing (horizontal segments)
    int block_size = BOARD_SIZE / size;
    int block_start = rank * block_size;
```

```
    int block_end = block_start + block_size;
    if (DEBUG_PRINT_RANK_INFO) {
        // Print the block start and end indices for each process
        printf("Rank %d: Block start = %d, Block end = %d\n", rank, block_start,
    block_end);
    }

    // Allocate memory for the local and next generation boards
    uint8_t *local_board = (uint8_t *) malloc(BOARD_SIZE * (block_size + 2) * sizeof(
    uint8_t));
    uint8_t *next_gen_board = (uint8_t *) malloc(BOARD_SIZE * (block_size + 2) *
    sizeof(uint8_t));

    if (local_board == NULL || next_gen_board == NULL) {
        fprintf(stderr, "Error allocating memory.\n");
        exit(EXIT_FAILURE);
    }

    // Initialize the local board with the grower pattern
    initialize_local_board(local_board, block_start, block_size);

    // Record the start time for measuring execution time
    double start_time = MPI_Wtime();

    for (int iter = 0; iter < MAX_ITERATIONS; ++iter) {
        if (DEBUG_PRINT_RANK_INFO) {
            // Print the population of the local board for each iteration
            printf("Rank %d: Iteration %d, Population = %d\n", rank, iter,
                   local_board_population(local_board, block_size));
        }

        if (DEBUG_PRINT_REGION) {
            print_region(local_board, GROWER_START_ROW, GROWER_START_COL,
    GROWER_HEIGHT, GROWER_WIDTH);
        }

        if (DEBUG_RUN_VERIFICATION_CHECKS) {
            // Verification checks for specific iterations
            run_verification(local_board, iter, block_size);
        }

        if (DEBUG_PRINT_TOTAL_POPULATION && am_master) {
            // Calculate and print the total population of the entire board
            int total_population = total_board_population(local_board, block_size);
            printf("Total population after iteration %d: %d\n", iter,
    total_population);
        }

        // Communication between neighboring processes using MPI
        // Identify left and right neighbors, set MPI_PROC_NULL if at the boundary
        int left_neighbour = (rank == 0) ? MPI_PROC_NULL : (rank - 1);
        int right_neighbour = (rank == size - 1) ? MPI_PROC_NULL : (rank + 1);

        // Define MPI request objects for non-blocking communication
        MPI_Request left_send_request, right_send_request;
        MPI_Request left_recv_request, right_recv_request;

        // Initiate non-blocking send operations to left and right neighbors
        if (block_start > 0) {
            // Communication with the left neighbor
```

```
257             if (MPI_Isend(&local_board[INDEX(1, 0)], BOARD_SIZE, MPI_UINT8_T,
    left_neighbour, 0, MPI_COMM_WORLD,
258                           &left_send_request) != MPI_SUCCESS) {
259                 fprintf(stderr, "Error in MPI_Isend to left neighbor.\n");
260                 exit(EXIT_FAILURE);
261             }
262             if (MPI_Irecv(&local_board[INDEX(0, 0)], BOARD_SIZE, MPI_UINT8_T,
    left_neighbour, 0, MPI_COMM_WORLD,
263                           &left_recv_request) != MPI_SUCCESS) {
264                 fprintf(stderr, "Error in MPI_Irecv from left neighbor.\n");
265                 exit(EXIT_FAILURE);
266             }
267         }
268 
269         if (block_end < BOARD_SIZE) {
270             // Communication with the right neighbor
271             if (MPI_Isend(&local_board[INDEX(block_size, 0)], BOARD_SIZE, MPI_UINT8_T
    , right_neighbour, 0,
272                           MPI_COMM_WORLD,
273                           &right_send_request) != MPI_SUCCESS) {
274                 fprintf(stderr, "Error in MPI_Isend to right neighbor.\n");
275                 exit(EXIT_FAILURE);
276             }
277             if (MPI_Irecv(&local_board[INDEX(block_size + 1, 0)], BOARD_SIZE,
    MPI_UINT8_T, right_neighbour, 0,
278                           MPI_COMM_WORLD, &right_recv_request) != MPI_SUCCESS) {
279                 fprintf(stderr, "Error in MPI_Irecv from right neighbor.\n");
280                 exit(EXIT_FAILURE);
281             }
282         }
283 
284         // Wait for completion of all non-blocking communication operations
285         if (block_start > 0) {
286             if (MPI_Wait(&left_send_request, MPI_STATUS_IGNORE) != MPI_SUCCESS) {
287                 fprintf(stderr, "Error in MPI_Wait for left_send_request.\n");
288                 exit(EXIT_FAILURE);
289             }
290             if (MPI_Wait(&left_recv_request, MPI_STATUS_IGNORE) != MPI_SUCCESS) {
291                 fprintf(stderr, "Error in MPI_Wait for left_recv_request.\n");
292                 exit(EXIT_FAILURE);
293             }
294         }
295         if (block_end < BOARD_SIZE) {
296             if (MPI_Wait(&right_send_request, MPI_STATUS_IGNORE) != MPI_SUCCESS) {
297                 fprintf(stderr, "Error in MPI_Wait for right_send_request.\n");
298                 exit(EXIT_FAILURE);
299             }
300             if (MPI_Wait(&right_recv_request, MPI_STATUS_IGNORE) != MPI_SUCCESS) {
301                 fprintf(stderr, "Error in MPI_Wait for right_recv_request.\n");
302                 exit(EXIT_FAILURE);
303             }
304         }
305 
306         // Run one iteration of the Game of Life for the local region
307         run_game_of_life(local_board, next_gen_board, block_size);
308 
309         // Swap pointers to update the local board for the next iteration
310         uint8_t *temp = local_board;
311         local_board = next_gen_board;
312         next_gen_board = temp;
```

```
313     }
314 
315     // Record the end time for measuring execution time
316     double end_time = MPI_Wtime();
317 
318     // Print total execution time
319     if (DEBUG_PRINT_EXECUTION_TIME && am_master) {
320         printf("Total execution time: %f seconds\n", end_time - start_time);
321     }
322 
323     // Print the final time : the time taken by the slowest process to complete the
    execution
324     double max_time;
325     if (MPI_Reduce(&end_time, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD)
    != MPI_SUCCESS) {
326         fprintf(stderr, "Error in MPI_Reduce operation.\n");
327         exit(EXIT_FAILURE);
328     }
329     printf("Max execution time: %f seconds\n", max_time - start_time);
330 
331 
332     // Free allocated memory
333     free(local_board);
334     free(next_gen_board);
335 
336     // Finalize MPI
337     if (MPI_Finalize() != MPI_SUCCESS) {
338         fprintf(stderr, "Error finalizing MPI.\n");
339         exit(EXIT_FAILURE);
340     }
341 
342     return 0;
343 }
```

Listing 10: Main Function and MPI Initialization

The main function initializes MPI, divides the board into blocks for parallel processing, allocates memory, runs iterations, performs MPI communication, and finalizes MPI.

2 Data distribution and communication method

In the provided code, the data distribution strategy involves dividing the Game of Life board into horizontal segments, creating a strip-like decomposition. Each MPI process is assigned a specific segment of the board, responsible for computing the Game of Life rules within that local region. The primary communication occurs between neighboring processes to exchange boundary information necessary for the computation of the next generation. For each process, the left and right neighbors are identified, and non-blocking MPI send and receive operations are employed to facilitate data exchange.

Specifically, communication involves sending and receiving entire rows of cells between neighboring processes. The left neighbor sends its rightmost row to the current process, and the right neighbor sends its leftmost row. This horizontal exchange of boundary data ensures that each process has the required information to apply the Game of Life rules accurately within its local segment. The MPI_Wait function is utilized to synchronize these non-blocking communication operations, ensuring proper coordination and data consistency throughout the parallel simulation. This communication strategy effectively balances computation and communication, contributing to the scalability of the simulation across multiple MPI processes.

3 Speedup

Table 1 summarizes the execution times of the parallel Game of Life implementation across various process configurations that were run on the Snellius cluster, using one node. Each row corresponds to a specific total number of processes/threads (specifically the product of the two), and the columns represent different configurations within that total. The "Max Execution Time" column indicates the highest execution time observed for a given configuration. The table provides insights into the scalability of the parallel implementation, demonstrating improved performance as the number of processes increases.

The speedup (S) is calculated using the formula:

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$$

where T_{serial} is the execution time in a serial (non-parallel) setting, and T_{parallel} is the (maximum) execution time in a parallel setting.

Total Processes*Threads	**#Processes**	**#Threads per process**	**Execution Time (s)**	**Max Exec Time (s)**	**Speedup**
1	1	1	206.133195	206.133195	-
2	1	2	103.484263	103.491485	1.99
	2	1	103.491485		
4	1	4	51.930277	51.930277	3.97
	2	2	51.929517		
	4	1	51.907608		
8	1	8	26.050067	26.050067	7.91
	2	4	26.005198		
	4	2	26.036027		
	8	1	25.957004		
16	1	16	13.020182	13.020182	15.83
	2	8	13.044554		
	4	4	13.068875		
	8	2	13.057330		
	16	1	13.000273		
32	1	32	6.519567	6.519567	31.62
	2	16	6.532024		
	4	8	6.513430		
	8	4	6.515856		
	16	2	6.492571		
	32	1	6.438125		
64	1	64	3.341772	3.341772	61.68
	2	32	3.307247		
	4	16	3.267373		
	8	8	3.318634		
	16	4	3.294378		
	32	2	3.262157		
	64	1	3.162100		
128	1	128	9.926397	9.926397	20.77
	2	64	2.182159		
	4	32	6.728790		
	8	16	7.027649		
	16	8	7.674845		
	32	4	2.285173		
	64	2	1.958015		
	128	1	7.117786		

Table 1: Benchmark Results for Parallel Game of Life

The speedup analysis is visualized in Figure 1, where the speedup (last column of Table 1) is plotted against the total number of processes/threads (first column of Table 1). The rounded speedup values are annotated on the data points for clarity.

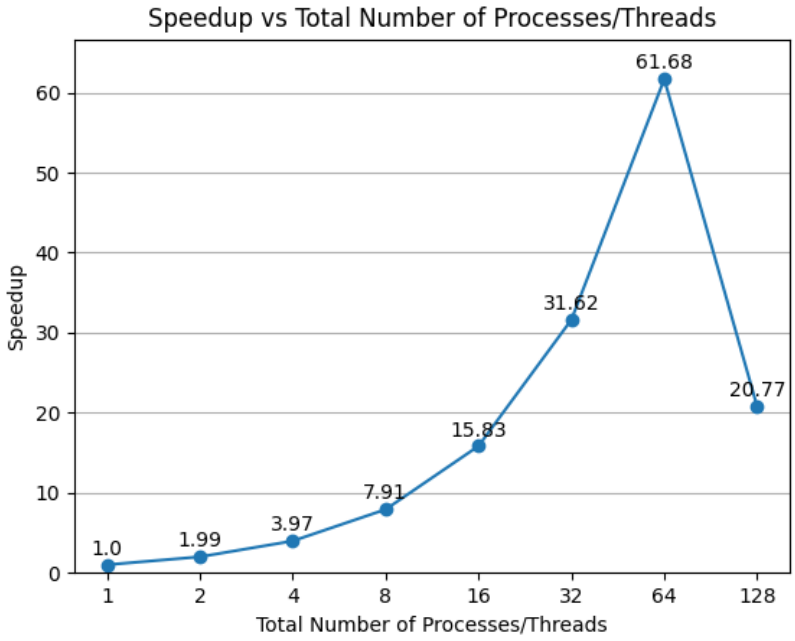

Figure 1: Speedup vs Total Number of Processes/Threads

4 Population Growth

In the context of the provided Game of Life simulation, it is noteworthy that after 5000 generations, the final population is at 3647 living cells.

The growth chart (Figure 2), visually encapsulates this evolution, showcasing the population fluctuations until reaching the state of 3647 cells after the specified number of iterations.

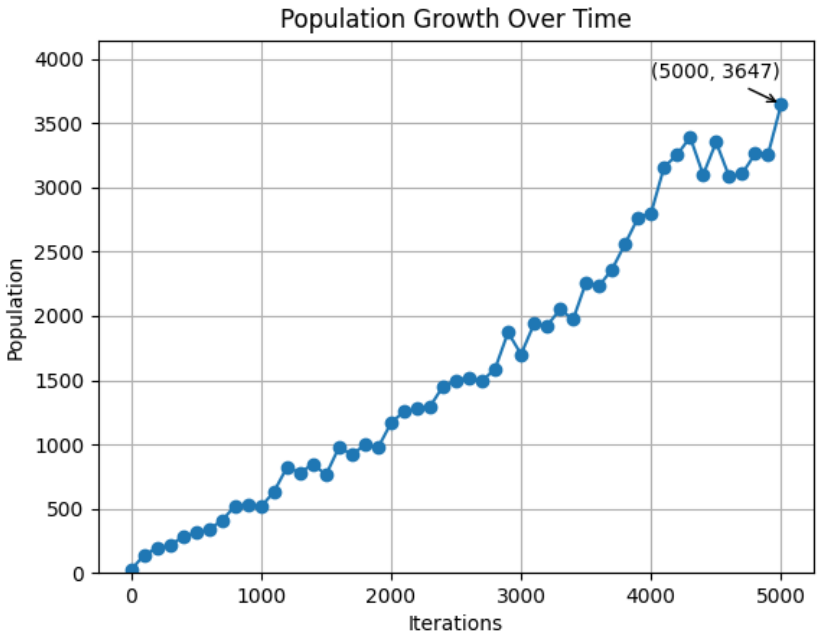

Figure 2: Population Growth Over 5000 Generations