МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет информационных технологий

Кафедра параллельных вычислений

ОТЧЕТ

О ВЫПОЛНЕНИИ ПРАКТИЧЕСКОЙ РАБОТЫ

«Игра "Жизнь" Дж. Конвея»

студентки 2 курса, группы 21207

Черновской Яны Тихоновны

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

А.Ю. Власенко

Новосибирск 2023

СОДЕРЖАНИЕ

СОДЕРЖАНИЕ 2
ЦЕЛЬ 3
ЗАДАНИЕ 4
ОПИСАНИЕ РАБОТЫ 5
ЗАКЛЮЧЕНИЕ 9
ПРИЛОЖЕНИЕ 10
Приложение 1. Полный листинг параллельной программы на C 10
Приложение 2. Скрипт для запуска параллельной программы 15

ЦЕЛЬ

Практическое освоение методов реализации алгоритмов мелкозернистого параллелизма на крупноблочном параллельном вычислительном устройстве на примере реализации клеточного автомата «Игра "Жизнь" Дж. Конвея» с использованием неблокирующих коммуникаций библиотеки MPI.

ЗАДАНИЕ

1. Написать параллельную программу на языке C/C++ с использованием MPI, реализующую клеточный автомат игры "Жизнь" с завершением программы по повтору состояния клеточного массива в случае одномерной декомпозиции массива по строкам и с циклическими границами массива. Проверить корректность исполнения алгоритма на различном числе процессорных ядер и различных размерах клеточного массива, сравнив с результатами, полученными для исходных данных вручную.

2. Измерить время работы программы при использовании различного числа процессорных ядер: 1, 2, 4, 8, 16, … . Размеры клеточного массива X и Y подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд. Построить графики зависимости времени работы, ускорения и эффективности распараллеливания от числа используемых ядер.

3. Произвести профилирование программы и выполнить ее оптимизацию. Попытаться достичь 50-процентной эффективности параллельной реализации на 16 ядрах для выбранных X и Y

Исходные данные задачи

Клеточный массив размером X × Y клеток (не менее 100 × 100) заполнен нулями. Единицами инициализированы пять клеток (1,2), (2,3), (3,1), (3,2) и (3,3), согласно рис. 6. Такая конфигурация с периодом в 4 итерации воспроизводит саму себя со смещением на одну клетку по диагонали вправо-вниз и называется парусником (глайдером).

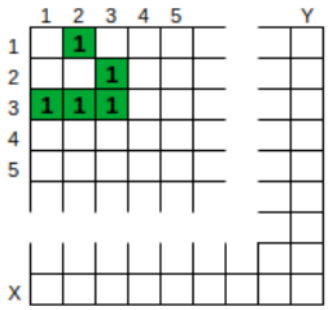


Рис. 6. Исходное заполнение клеточного массива

ОПИСАНИЕ РАБОТЫ

1. Была написана параллельная программа с использованием MPI

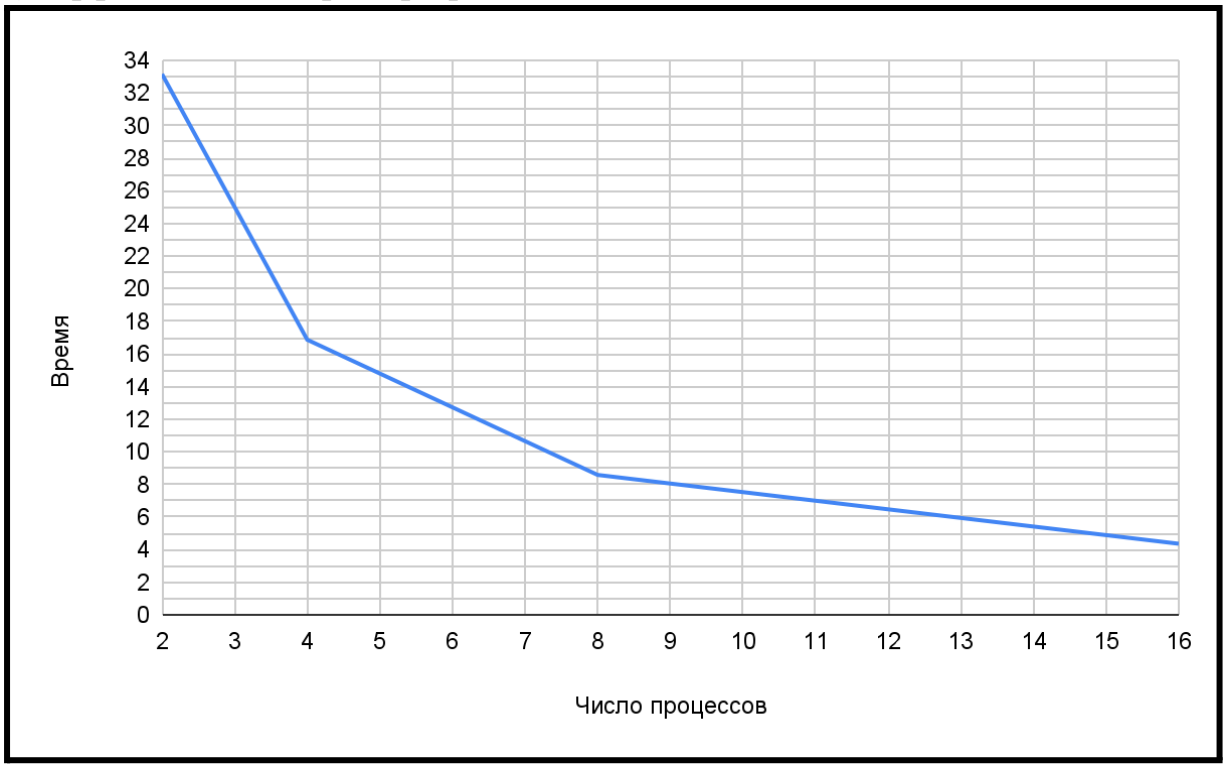
2. Было измерено время работы параллельной программы на 1, 2, 4, 8, 16 процессах

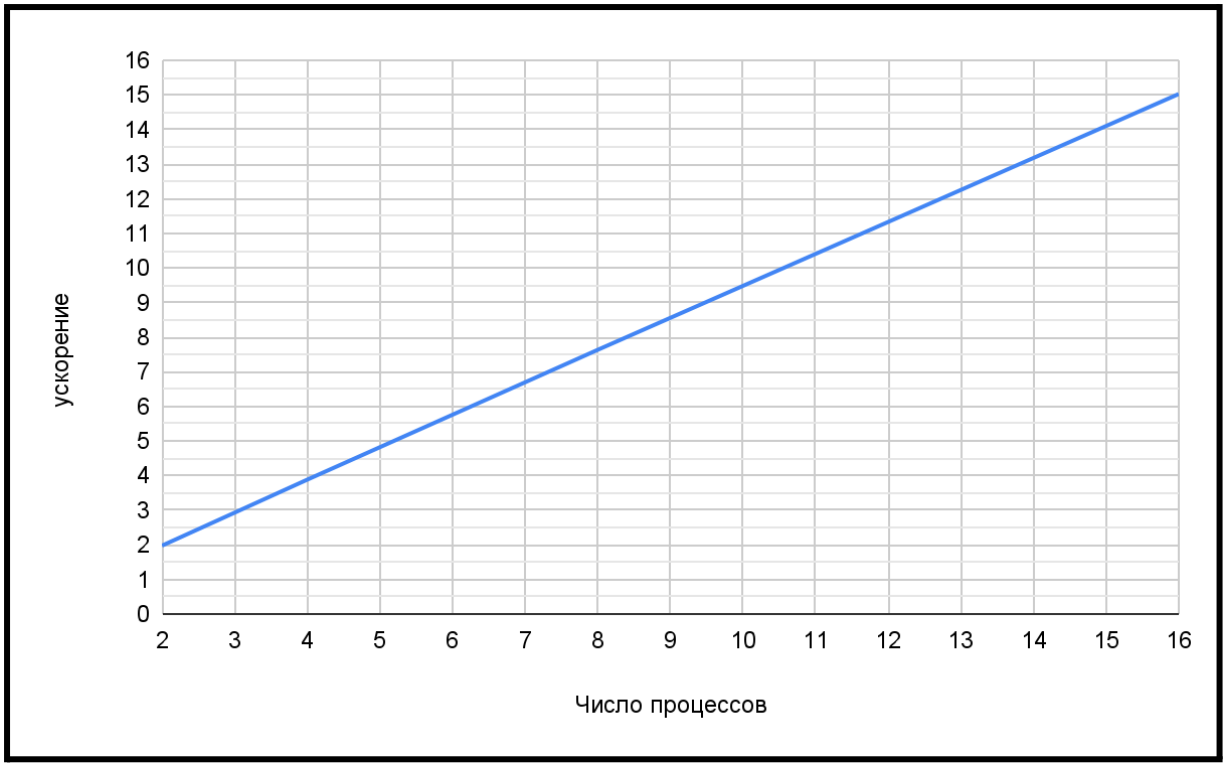
```
Total steps: 800
Total time: 65.487530
Total steps: 800
Total time: 33.169977
Total steps: 800
Total time: 16.871115
Total steps: 800
Total time: 8.577634
Total steps: 800
Total time: 4.357382
```

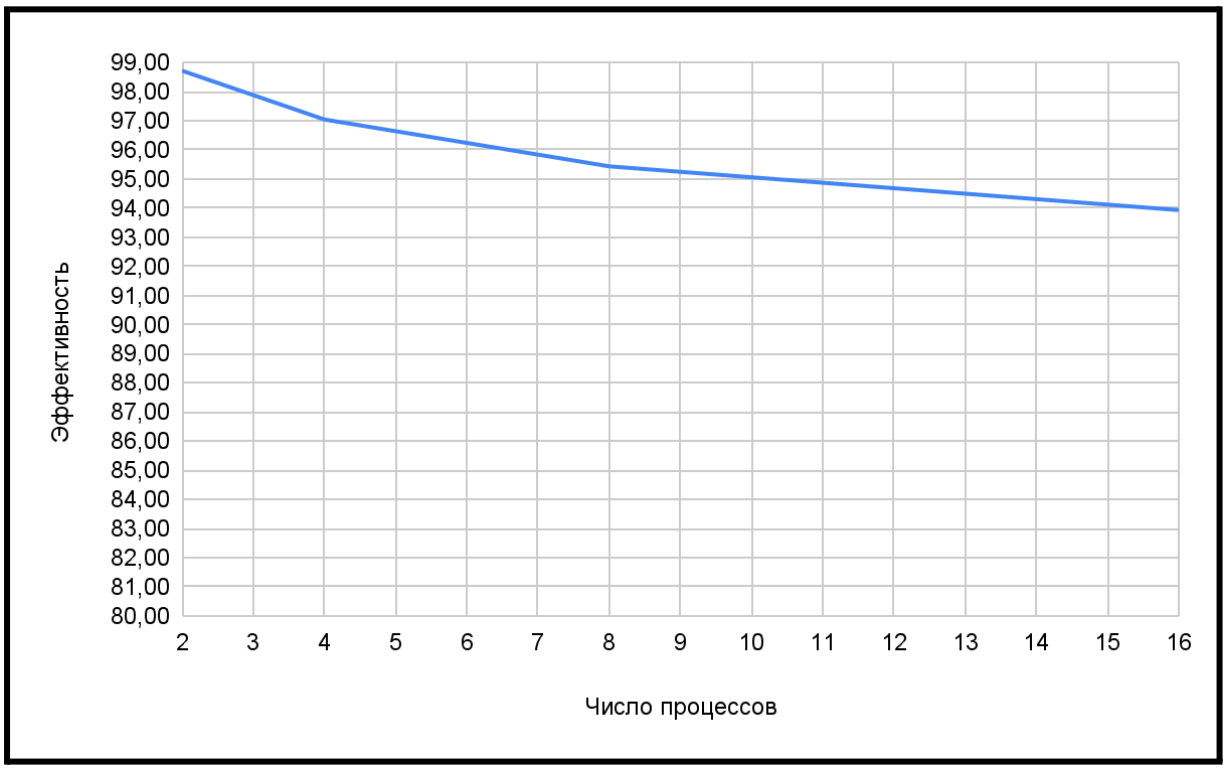
3. Были построены графики времени, ускорения и эффективности, где

а) Ускорение: Sp = T1 / Tp, где T1 - время работы последовательной программы, Tp - время работы параллельной программы на p процессах

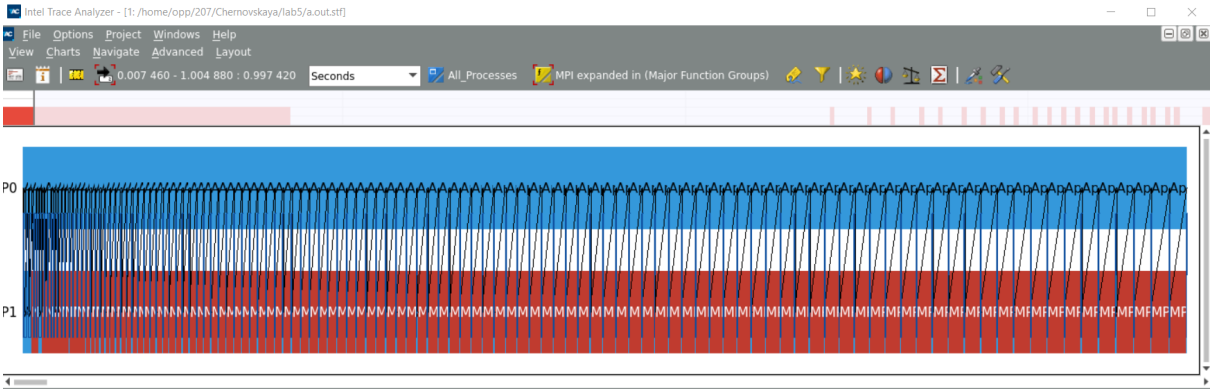
б) Эффективность Ep = Sp / p * 100%.







4. Было проведено профилирование на размере 200 на 200 на 2 процессах



Flat Profile | Load Balance | Call Tree | Call Graph

All_Processes

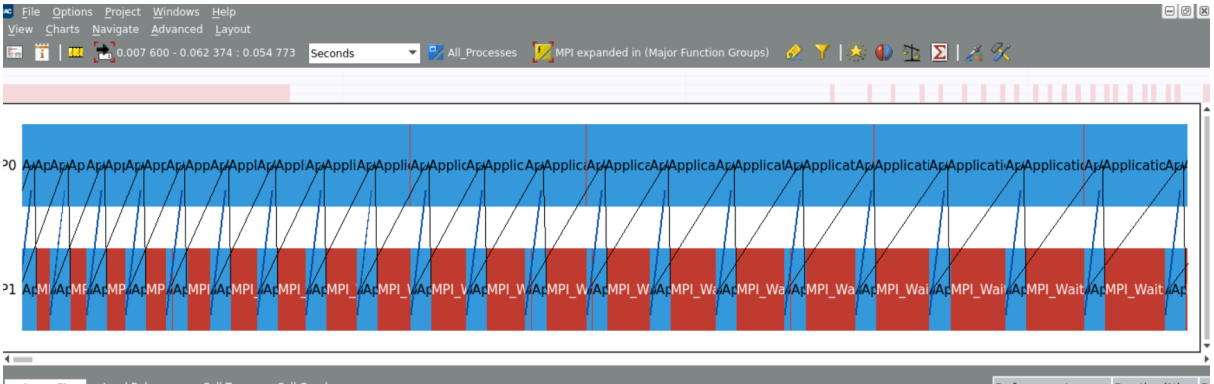
Name	TSelf	TSelf	TTotal	TTotal	#Calls	#Calls	TSelf /Call
All_Processes							
Group Application	1.10892 s		1.99484 s		0		n.a.
MPI_Iallgather	1.794e-3 s		1.794e-3 s		236		7.60169e-6 s
MPI_Isend	863e-6 s		863e-6 s		472		1.82839e-6 s
MPI_Irecv	315e-6 s		315e-6 s		472		667.373e-9 s
MPI_Wait	882.944e-3 s		882.944e-3 s		1180		748.258e-6 s

Performance Issue	Duration (%)
Wait at Barrier	4.19%
Late Receiver	0.03%

Show advanced...

Description | Affected Process

Select performance issue to see details.



Flat Profile | Load Balance | Call Tree | Call Graph

All_Processes

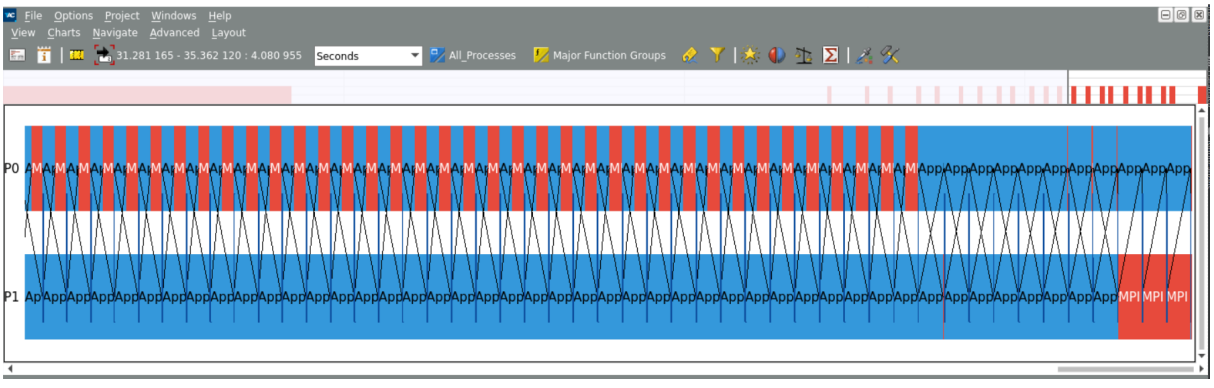
Name	TSelf	TSelf	TTotal	TTotal	#Calls	#Calls	TSelf /Call
All_Processes							
Group Application	74.4199e-3 s		109.547e-3 s		0		n.a.
MPI_Iallgather	294e-6 s		294e-6 s		41		7.17073e-6 s
MPI_Isend	146e-6 s		146e-6 s		82		1.78049e-6 s
MPI_Irecv	53e-6 s		53e-6 s		82		646.341e-9 s
MPI_Wait	34.6339e-3 s		34.6339e-3 s		210		164.923e-6 s

Performance Issue	Duration (%)
Wait at Barrier	13.44%
Late Receiver	0.08%

Show advanced...

Description | Affected Process

Select performance issue to see details.



ЗАКЛЮЧЕНИЕ

Были осовены методы реализации алгоритмов мелкозернистого параллелизма на крупноблочном параллельном вычислительном устройстве с использованием неблокирующих коммуникаций библиотеки MPI. По результатам профилирования можно сделать вывод, что, несмотря на использование неблокирующих коммуникаций, большая часть времени программы уходит на MPI_Wait.

ПРИЛОЖЕНИЕ

Приложение 1. Полный листинг параллельной программы на C

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

int ROWS;
int COLS;
#define STEPS 1000

void init_grid(int *grid) {
    grid[1] = 1;
    grid[COLS + 2] = 1;
    grid[COLS * 2 + 0] = 1;
    grid[COLS * 2 + 1] = 1;
    grid[COLS * 2 + 2] = 1;
}

void fill(int i, const int *curr_grid, int *next_grid) {
    for (int j = 0; j < COLS; j++) {
        int live_neighbors = 0;
        if (curr_grid[i * COLS + (j + 1) % COLS] == 1) live_neighbors++; // right
        if (curr_grid[i * COLS + (j - 1 + COLS) % COLS] == 1) live_neighbors++; // left

        if (curr_grid[(i + 1) * COLS + j] == 1) live_neighbors++; // down
        if (curr_grid[(i - 1) * COLS + j] == 1) live_neighbors++; // up

        if (curr_grid[(i - 1) * COLS + (j + 1) % COLS] == 1) live_neighbors++;// up right
        if (curr_grid[(i - 1) * COLS + (j - 1 + COLS) % COLS] == 1) live_neighbors++;// up left

        if (curr_grid[(i + 1) * COLS + (j - 1 + COLS) % COLS] == 1) live_neighbors++; // down left
        if (curr_grid[(i + 1) * COLS + (j + 1) % COLS] == 1) live_neighbors++; // down right

        if (curr_grid[i * COLS + j] == 1) { // cell is alive
            if (live_neighbors < 2 || live_neighbors > 3) {
                next_grid[i * COLS + j] = 0; // cell dies
            } else {
                next_grid[i * COLS + j] = 1; // cell stays alive
            }
        } else { // cell is dead
            if (live_neighbors == 3) {
                next_grid[i * COLS + j] = 1; // cell becomes alive
            } else {
                next_grid[i * COLS + j] = 0; // cell stays dead
            }
```

```
45.             }
46.
47.         }
48.     }
49.
50.     int get_flag(int index, int rows_per_proc, const int *copy_array, const int *curr_grid) {
51.
52.         for (int i = 0; i < index; i++) {
53.             int equals = 1;
54.             for (int k = 1; k <= rows_per_proc; k++) {
55.                 for (int j = 0; j < COLS; j++) {
56.                     if (copy_array[i * ROWS * COLS + k * COLS + j] != curr_grid[k * COLS + j]) {
57.                         equals = 0;
58.                         break;
59.                     }
60.                 }
61.             }
62.
63.             if (equals) { return 1; }
64.
65.         }
66.
67.
68.         return 0;
69.     }
70.
71.     void set_matrix_part(int *send_counts, int *displs, int size, int num_proc) {
72.         int offset = 0;
73.         for (int i = 0; i < num_proc; ++i) {
74.             int line_counts = size / num_proc;
75.
76.             if (i < size % num_proc) {
77.                 ++line_counts;
78.
79.             }
80.
81.             displs[i] = offset * COLS;
82.             offset += line_counts;
83.             send_counts[i] = line_counts * COLS;
84.         }
85.     }
86.
87.     bool check_flags(int *recv_buffer_flags, int size, int index) {
88.         for (int i = 0; i < size; i++) {
89.             if (recv_buffer_flags[i * STEPS + (index)] == 0) {
90.                 return false;
91.             }
92.         }
```

```
93.         return true;
94.     }
95.
96.     void create_cart_comm(int size, MPI_Comm *cart_comm) {
97.         int dims[2] = {size, 1};
98.         int periods[2] = {true, false};
99.
100.        MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, cart_comm);
101.    }
102.
103.    int main(int argc, char **argv) {
104.
105.        int rank, size, step;
106.        int index = 0;
107.
108.        int *flags;
109.        int *copy_array;
110.        int *recv_buffer_flags;
111.        int *grid;
112.        int *send_counts;
113.        int *displs;
114.
115.        int rows_per_proc, real_size;
116.
117.        MPI_Comm cart_comm;
118.
119.        double start_time, end_time;
120.        MPI_Init(&argc, &argv);
121.        MPI_Comm_size(MPI_COMM_WORLD, &size);
122.        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
123.
124.        MPI_Status status;
125.
126.        ROWS = atoi(argv[1]);
127.        COLS = atoi(argv[2]);
128.
129.        send_counts = malloc(sizeof(int) * size);
130.        displs = malloc(sizeof(int) * size);
131.
132.        set_matrix_part(send_counts, displs, ROWS, size);
133.
134.        rows_per_proc = send_counts[rank] / COLS;
135.        real_size = rows_per_proc + 2;
136.
137.        if (rank == 0) grid = (int *) calloc(ROWS * COLS, sizeof(int));
138.
139.        int *curr_grid = (int *) malloc(sizeof(int) * real_size * COLS);
140.        int *next_grid = (int *) malloc(sizeof(int) * real_size * COLS);
```

```
141.
142.       recv_buffer_flags = (int *) malloc(sizeof(int) * size * STEPS);
143.       copy_array = (int *) malloc(sizeof(int) * STEPS * ROWS * COLS);
144.
145.       flags = calloc(STEPS, sizeof(int));
146.
147.       srand(rank);
148.       if (rank == 0) init_grid(grid);
149.
150.       MPI_Scatterv(grid, send_counts, displs, MPI_INT, (curr_grid + COLS),
151.                    send_counts[rank], MPI_INT, 0, MPI_COMM_WORLD);
152.
153.       create_cart_comm(size, &cart_comm);
154.
155.    // Determine neighbor ranks
156.       int left_rank, right_rank;
157.       MPI_Cart_shift(cart_comm, 0, 1, &left_rank, &right_rank);
158.
159.       start_time = MPI_Wtime();
160.
161.       for (step = 0; step < STEPS; step++) {
162.          MPI_Request reqs[5];
163.          MPI_Isend(curr_grid + COLS, COLS,
164.                    MPI_INT, left_rank,
165.                    MPI_ANY_TAG, MPI_COMM_WORLD, &reqs[0]);
166.          MPI_Isend(curr_grid + (real_size - 2) * COLS, COLS, MPI_INT, right_rank,
167.                    MPI_ANY_TAG, MPI_COMM_WORLD, &reqs[1]);
168.
169.          MPI_Irecv(curr_grid + (real_size - 1) * COLS, COLS, MPI_INT, right_rank,
170.                    MPI_ANY_TAG, MPI_COMM_WORLD, &reqs[3]);
171.          MPI_Irecv(curr_grid, COLS, MPI_INT, left_rank,
172.                    MPI_ANY_TAG, MPI_COMM_WORLD, &reqs[2]);
173.
174.          flags[index] = get_flag(index, rows_per_proc, copy_array, curr_grid);
175.
176.          MPI_Iallgather(flags, STEPS, MPI_INT, recv_buffer_flags, STEPS,
177.                         MPI_INT, MPI_COMM_WORLD, &reqs[4]);
178.
179.          for (int i = 2; i <= real_size - 3; i++) fill(i, curr_grid, next_grid);
180.
181.          MPI_Wait(&reqs[0], &status);
182.          MPI_Wait(&reqs[2], &status);
183.
184.          fill(1, curr_grid, next_grid);
185.
186.          MPI_Wait(&reqs[1], &status);
187.          MPI_Wait(&reqs[3], &status);
188.
```

```
189.            fill(real_size - 2, curr_grid, next_grid);
190.
191.            MPI_Wait(&reqs[4], &status);
192.            if (check_flags(recv_buffer_flags, size, index)) break;
193.
194.            for (int i = 1; i <= rows_per_proc; i++)
195.                for (int j = 0; j < COLS; j++)
196.                    copy_array[index * ROWS * COLS + i * COLS + j] = curr_grid[i * COLS + j];
197.
198.            index++;
199.
200.            int *temp = curr_grid;
201.            curr_grid = next_grid;
202.            next_grid = temp;
203.        }
204.
205.        if (rank == 0) {
206.            end_time = MPI_Wtime();
207.            printf("Total steps: %d\n", index);
208.            printf("Total time: %f\n", end_time - start_time);
209.            free(grid);
210.        }
211.
212.        free(curr_grid);
213.        free(next_grid);
214.        free(flags);
215.        free(copy_array);
216.        free(recv_buffer_flags);
217.        free(send_counts);
218.        free(displs);
219.
220.        MPI_Finalize();
221.        return 0;
222.    }
223.
```

Приложение 2. Скрипт для запуска параллельной программы

```
1.  #!/bin/bash
2.
3.  #PBS -l walltime=00:01:00
4.  #PBS -l select=2:ncpus=8:mpiprocs=8:mem=2000m,place=scatter
5.  #PBS -m n
6.
7.  cd $PBS_O_WORKDIR
8.
9.  MPI_NP=$(wc -l $PBS_NODEFILE | awk '{ print $1 }')
10. echo "Number of MPI process: $MPI_NP"
11.
12. mpirun -hostfile $PBS_NODEFILE -perhost 1 -np 1 ./life 200 200
13. mpirun -hostfile $PBS_NODEFILE -perhost 1 -np 2 ./life 200 200
14. mpirun -hostfile $PBS_NODEFILE -perhost 1 -np 4 ./life 200 200
15. mpirun -hostfile $PBS_NODEFILE -perhost 1 -np 8 ./life 200 200
16. mpirun -hostfile $PBS_NODEFILE -perhost 1 -np 16 ./life 200 200
```