Московский государственный университет
имени М. В. Ломоносова
Факультет вычислительной математики и кибернетики

Задание 3. Игра "Жизнь".

Отчёт
О выполненном задании

Выполнил:
студент 523 группы
Латыпов Ш. И.
latypovshamil2001@gmail.com

Москва
2023

Описание задачи

Требуется написать параллельную программу, реализующую описанную в главе 3 пособия схему распараллеливания двумерного клеточного автомата (игра Жизнь) с циклическими граничными условиями. Программа должна принимать на вход (через аргументы командной строки) два параметра: n - размер квадратного фрагмента автомата, обрабатываемого одним процессором, t - число итераций. Если число параллельных процессов равно $P = p^2$..

Результатом работы программы должны быть два файла. В файле output.txt должно быть записано состояние автомата после выполнения последней итерации. В файле stat.txt должно быть сохранено время T работы основного цикла программы с указанием всех параметров запуска, в том числе параметр P — число параллельных процессов.

Для тестирования используется начальная конфигурация с одним глайдером, расположенным в центре автомата. Необходимо рассчитать, через сколько итераций глайдер опять окажется в этой точке.

Код программы

```
#include <mpi.h>
#include <algorithm>
#include <fstream>
#include <iostream>
#include <unordered_set>
#include <vector>

using namespace std;

int f(int* data, int i, int j, int n) {
  int state = data[i * (n + 2) + j];
  int s = -state;
  for (int ii = i - 1; ii <= i + 1; ii++)
    for (int jj = j - 1; jj <= j + 1; jj++)
      s += data[ii * (n + 2) + jj];
  if (state == 0 && s == 3)
    return 1;
  if (state == 1 && (s < 2 || s > 3))
    return 0;
  return state;
}

void update_data(int n, int* data, int* temp) {
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
      temp[i * (n + 2) + j] = f(data, i, j, n);
}

void exchange_borders(int* grid, int n, int rank, int p) {
  MPI_Status status;

  int row = rank / p;
  int col = rank % p;

  int up = (row == 0) ? ((p - 1) * p + col) : (rank - p);
```

```
  int down = (row == p - 1) ? col : (rank + p);
  int left = (col == 0) ? (rank + p - 1) : (rank - 1);
  int right = (col == p - 1) ? (rank - p + 1) : (rank + 1);

  // Обмен верхней и нижней границы
  MPI_Sendrecv(&grid[n], n, MPI_INT, up, 0, &grid[n * (n - 1)], n, MPI_INT,
               down, 0, MPI_COMM_WORLD, &status);

  MPI_Sendrecv(&grid[(n - 2) * n], n, MPI_INT, down, 1, grid, n, MPI_INT, up, 1,
               MPI_COMM_WORLD, &status);

  // Обмен левой и правой границы
  for (int i = 0; i < n; ++i) {
    MPI_Sendrecv(&grid[i * n + 1], 1, MPI_INT, left, 2, &grid[i * n + n - 1], 1,
                 MPI_INT, right, 2, MPI_COMM_WORLD, &status);
    MPI_Sendrecv(&grid[i * n + n - 2], 1, MPI_INT, right, 3, &grid[i * n], 1,
                 MPI_INT, left, 3, MPI_COMM_WORLD, &status);
  }
}

unordered_set<int> poses;

void setup_set(int n) {
  int n0 = 1 + n / 2;
  int m0 = 1 + n / 2;
  int middle_point = n0 * (n + 2) + m0;
  vector<int> points = {middle_point + 1, middle_point - (n + 2),
                        middle_point + (n + 2) + 1, middle_point + (n + 2),
                        middle_point + (n + 2) - 1};
  for (auto& tmp : points) {
    poses.insert(tmp);
  }
}

void init(int n, int* data, int local_n, int p, int rank) {
  for (int i = 0; i < (local_n + 2) * (local_n + 2); i++)
    data[i] = 0;

  int row = rank / p;
  int col = rank % p;

  int n0 = 1 + n / 2;
  int m0 = 1 + n / 2;
  int middle_point = n0 * (n + 2) + m0;

  for (int i = 0; i < local_n + 2; i++) {
    for (int j = 0; j < local_n + 2; j++) {
      int tmp = row * (n + 2) * local_n + col * local_n + i * (n + 2) + j;
      if (poses.find(tmp) != poses.end()) {
        data[i * (local_n + 2) + j] = 1;
      }
    }
```

```
  }
}

void run_life(int n, int T, int rank, int size) {
  int p = sqrt(size);
  int local_n = n / p;

  setup_set(n);
  MPI_Barrier(MPI_COMM_WORLD);

  int* local_data = new int[(local_n + 2) * (local_n + 2)];
  int* new_data = new int[(local_n + 2) * (local_n + 2)];

  init(n, local_data, local_n, p, rank);

  MPI_Barrier(MPI_COMM_WORLD);

  double start_time = MPI_Wtime();

  for (int t = 0; t < T; ++t) {
    update_data(local_n, local_data, new_data);
    swap(local_data, new_data);
    exchange_borders(local_data, local_n + 2, rank, p);
  }

  double end_time = MPI_Wtime();

  if (rank == 0) {
    int* result = new int[n * n];
    int* tmp = new int[(local_n + 2) * (local_n + 2)];
    for (int i = 0; i < size; i++) {
      int row = i / p;
      int col = i % p;
      if (i != 0) {
        MPI_Recv(tmp, (local_n + 2) * (local_n + 2), MPI_INT, i, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      } else {
        for (int j = 0; j < (local_n + 2) * (local_n + 2); j++) {
          tmp[j] = local_data[j];
        }
      }
      for (int j = 0; j < local_n; j++) {
        for (int k = 0; k < local_n; k++) {
          result[row * n * local_n + col * local_n + j * n + k] =
              tmp[(j + 1) * (local_n + 2) + k + 1];
        }
      }
    }

    ofstream f("output.dat");
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
```

```
        f << result[i * n + j];
      }
      f << endl;
    }
    f.close();

    cout << "Time = " << end_time - start_time << "\n";
    ofstream s("stat.txt");
    s << "Time = " << end_time - start_time << "\n";
    s << "n = " << n << "\n";
    s << "T = " << T << "\n";
    s.close();

    delete[] tmp;
    delete[] result;

  } else {
    MPI_Send(local_data, (local_n + 2) * (local_n + 2), MPI_INT, 0, 0,
             MPI_COMM_WORLD);
  }

  delete[] local_data;
  delete[] new_data;

  MPI_Barrier(MPI_COMM_WORLD);
}

int main(int argc, char** argv) {
  int n, T;
  MPI_Init(&argc, &argv);
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  if (argc != 3) {
    if (rank == 0) {
      std::cout << "Usage: " << argv[0] << " n T" << endl;
    }
    MPI_Finalize();
    return 1;
  }

  n = atoi(argv[1]);
  T = atoi(argv[2]);

  run_life(n, T, rank, size);

  MPI_Finalize();
  return 0;
}
```

Результаты работы программы

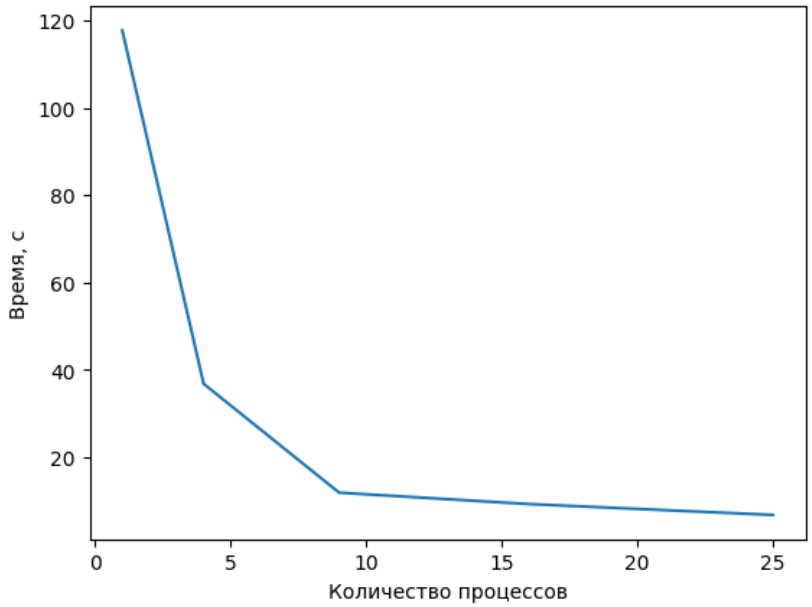
Запуск программы происходил с параметрами:
n = 600
T = 2400

Затем с параметрами:
n = 1200
T = 4800

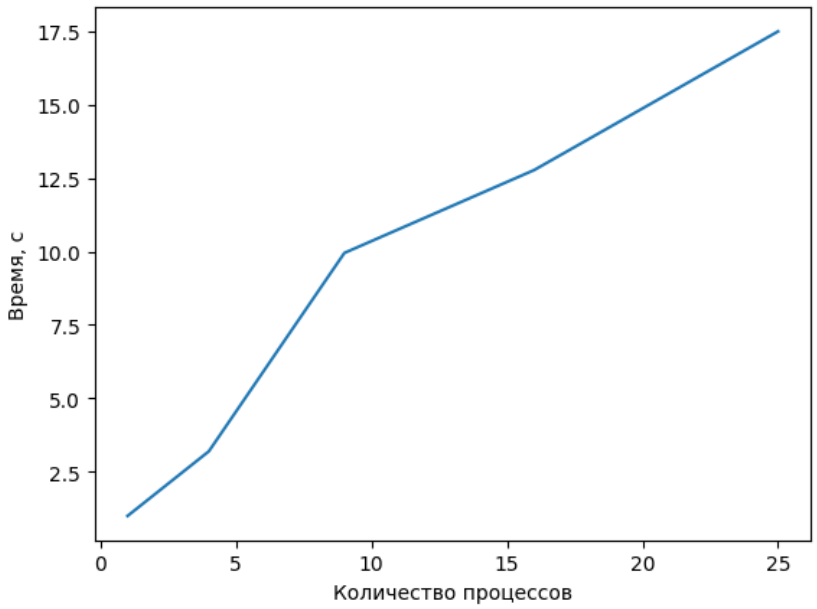
Количество MPI процессов = 1, 4, 9, 16, 25

Результат работы с n = 600 и T = 2400:

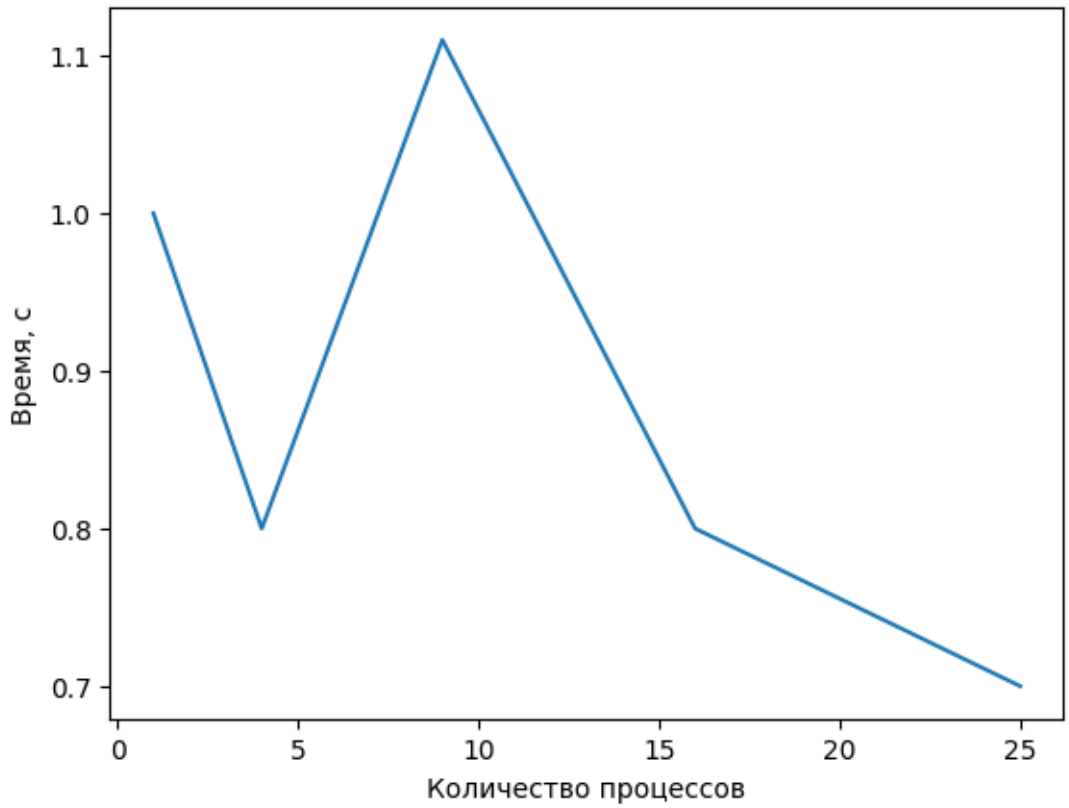
Время работы



Ускорение

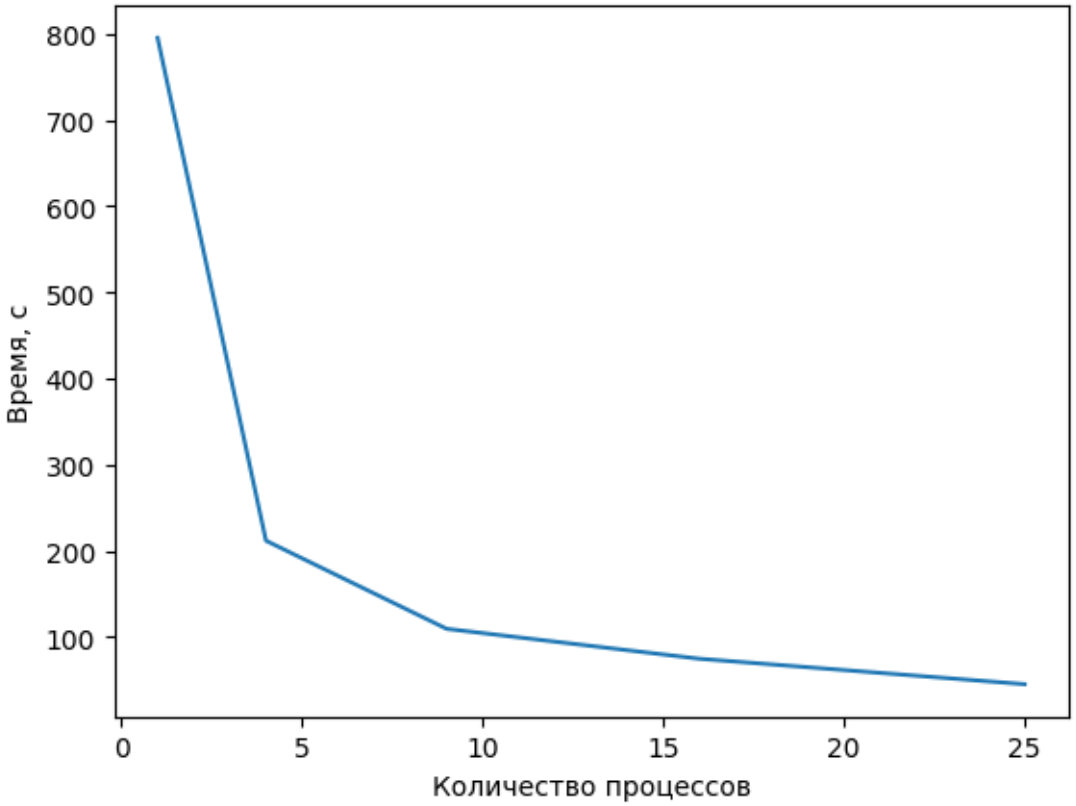


Эффективность

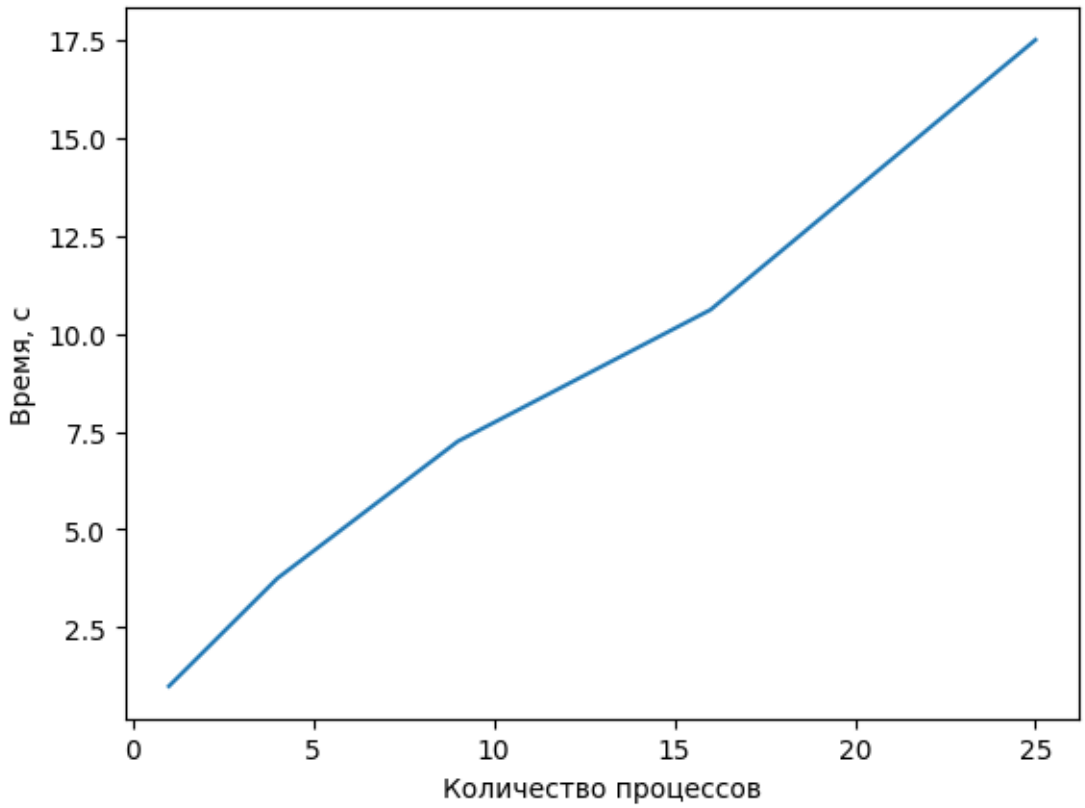


Результат работы с n = 1200 и T = 4800:

Время работы



Ускорение



Эффективность