**Міністерство освіти і науки України**

**Національний університет "Львівська політехніка"**

**Кафедра ЕОМ**



**Звіт з лабораторної роботи №3**

**з дисципліни “** **Паралельні та розподілені обчислення ”**

**на тему: ” МОЖЛИВОСТІ ВИКОРИСТАННЯ ПАРАЛЕЛЬНИХ АЛГОРИТМІВ”**

**Виконав: студент .гр. КІ-33**

**Кіндій В. А.**

**Прийняв: викладач**

**Козак Н. Б.**

**Львів 2020 р.**

**Мета роботи:** Дослідити можливості розв’язання різноманітних задач за допомогою паралельних алгоритмів. Навчитися виділяти незалежні гілки обчислень та виконувати їх паралельно.

**Завдання:**

Варіант – 5

Запропонувати та відобразити алгоритм реалізації гри “LIFE”. Гра моделює життя деякої колонії живих клітин, які виживають, розмножуються або гинуть згідно наступних умов. Клітина виживає, якщо вона має двох або трьох сусідів з восьми можливих. Якщо у клітини лише один сусід, або немає жодного, то вона гине (від ізоляції). Якщо клітина має більше трьох сусідів, то вона гине (від перенаселення). В будь-якій порожній позиції, яка має рівно трьох сусідів у наступному поколінні з’являється нова клітина. Гра повинна починатися з довільної кількості клітин, що розташовані в ігровому полі випадковим чином (інтерактивний режим задавання вхідних даних) – так звана початкова популяція. Програма повинна коректно завершувати роботу у таких випадках: а).загинула вся популяція, б) на вимогу користувача.

**Хід роботи:**

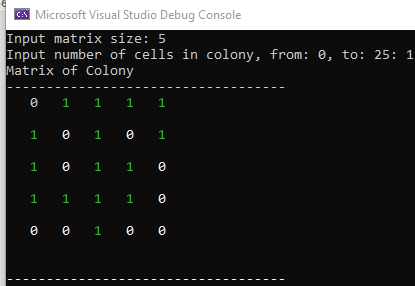
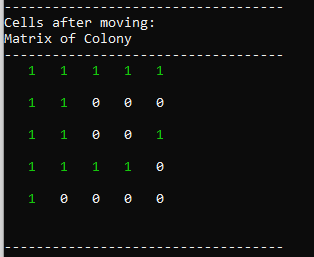
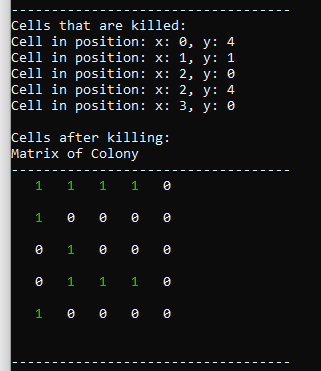
1. Аналізую завдання і виділяю в ньому незалежні гілки обчислень. Використовую метод розбиття задачі на підзадачі та виділення незалежних подій.
2. Реалізовую кожну гілку окрему
3. Обєдную всі частини для вирішення поставленої задачі і переконуюсь у правильності реалізації.
4. Оцінюю обчислювальні та часові затрати створеної програми.

Після аналізу завдання в ньому можна виділити гілки: задання початкової колонії, обчислення поточного елементу, вивід колонії. Виділивши ці гілки, оформляємо їх як окремі функції і використовуємо в програмі.

Крім того, з метою півищення продуктивності, можна також реалізувати одночасний перегляд матриці з різних сторін (наприклад, з першого елемента і з останнього елемента).

Приклад виконання програми:

Після вводу розміру матриці та кількості живих клітин, відображається матриця (елементи матриці які дорівнюють 1 відповідають за клітину). Після цього клітини рухаються у різних напрямках і здійснюється обчислення які з них залишились живі а які ні.

**Висновок**: дослідив можливості розвязання різноманітних задач за допомогою паралельних алгоритмів. Навчився виділяти незалежні гілки обчислень та виконувати їх паралельно.

**Додаток А**

|  |
| --- |
| #include <iostream>  #include <vector>  #include <random>  #include <time.h>  #include <windows.h>  using namespace std;  // Output matrix  void OutputMatrix(vector<vector<int>> matrix) {  HANDLE color = GetStdHandle(STD\_OUTPUT\_HANDLE); //just once  cout << "Matrix of Colony";  cout << "\n-----------------------------------\n";  for (int i = 0; i < matrix.size(); i++) {  for (int j = 0; j < matrix.size(); j++) {  cout << " ";  if (matrix[i][j] == 1) {  SetConsoleTextAttribute(color, 10);  cout << matrix[i][j];  SetConsoleTextAttribute(color, 15);  }  else  {  cout << matrix[i][j];  }    }  cout << "\n\n";  }  cout << "\n-----------------------------------\n";  }  // init colony with cells  vector<vector<int>> InitColony(int matrixSize) {  srand(time(NULL));  vector<vector<int>> matrix;  matrix.resize(matrixSize);  for (int i = 0; i < matrixSize; i++) {  matrix[i].resize(matrixSize);  }    cellsNumberInput:  cout << "Input number of cells in colony, from: " << 0 << ", to: " << matrixSize\*matrixSize << ": ";  int colonyCellsNumber = 0;  cin >> colonyCellsNumber;  if (colonyCellsNumber < 0 || colonyCellsNumber > matrixSize\* matrixSize) {  cout << "Incorrect cells number:" << colonyCellsNumber;  goto cellsNumberInput;  }  while (colonyCellsNumber != 0) {  int x = rand() % matrixSize;  int y = rand() % matrixSize;  if (matrix[x][y] == 0) {  matrix[x][y] = 1;  colonyCellsNumber--;  }  else  {  continue;  }  }  OutputMatrix(matrix);  return matrix;  }  // move cells to different positions  vector<vector<int>> CreateStep(vector<vector<int>> matrix, int x, int y) {  int xNewPos = 0; int yNewPos = 0;  int attempts = 5;  // 0 1 2  // 3 x 4  // 5 6 7  starStep:  int randomPosition = rand() % 8;  switch (randomPosition) {  case 0: {  xNewPos = x - 1; yNewPos = y - 1;  break;  }  case 1: {  xNewPos = x - 1; yNewPos = y;  break;  }  case 2: {  xNewPos = x - 1; yNewPos = y + 1;  break;  }  case 3: {  yNewPos = y - 1; xNewPos = x;  break;  }  case 4: {  yNewPos = y + 1; xNewPos = x;  break;  }  case 5: {  xNewPos = x + 1; yNewPos = y - 1;  break;  }  case 6: {  xNewPos = x + 1; yNewPos = y;  break;  }  case 7: {  xNewPos = x + 1; yNewPos = y + 1;  break;  }  }  if ((xNewPos >= 0 && xNewPos < matrix.size()) && (yNewPos >= 0 && yNewPos < matrix.size()) && matrix[xNewPos][yNewPos] == 0) {  matrix[xNewPos][yNewPos] = 1;  matrix[x][y] = 0;  }  else if (attempts > 0) {  attempts--;  goto starStep;  }  return matrix;  }  // for positions storing  class Position {  public:  int x;  int y;  Position(int x, int y) {  this->x = x;  this->y = y;  }  };  // to kill cells  vector<vector<int>> KillCells(vector<vector<int>> matrix) {  vector<Position> makredToDelete;  for (int i = 0; i < matrix.size(); i++) {  for (int j = 0; j < matrix.size(); j++) {  if (matrix[i][j] == 1) {  // check each node  int neighbors = 0;  if ((i - 1 > 0 && j - 1 > 0) && matrix[i - 1][j - 1] == 1) {  ++neighbors;  }  if ((i - 1 > 0) && matrix[i - 1][j] == 1)  {  ++neighbors;  }  if ((i - 1 > 0 && j + 1 < matrix.size()) && matrix[i - 1][j + 1] == 1)  {  ++neighbors;  }  if ((j - 1 > 0) && matrix[i][j - 1] == 1)  {  ++neighbors;  }  if ((j + 1 < matrix.size()) && matrix[i][j + 1] == 1)  {  ++neighbors;  }  if ((i + 1 < matrix.size() && j - 1 > 0) && matrix[i + 1][j - 1] == 1)  {  ++neighbors;  }  if ((i + 1 < matrix.size()) && matrix[i + 1][j] == 1)  {  ++neighbors;  }  if ((i + 1 < matrix.size() && j + 1 < matrix.size()) && matrix[i + 1][j + 1] == 1)  {  ++neighbors;  }  if (neighbors <= 1 || neighbors > 3) {  Position\* pos = new Position(i, j);  makredToDelete.push\_back(\*pos);  }  }  }  }  cout << "Cells that are killed:\n";  for (int i = 0; i < makredToDelete.size(); i++) {  cout << "Cell in position: x: " << makredToDelete[i].x << ", y: " << makredToDelete[i].y << endl;  matrix[makredToDelete[i].x][makredToDelete[i].y] = 0;  }  return matrix;  }  // check if colony is alive  bool CheckIfColonyIsAlive(vector<vector<int>> matrix) {  bool isAlive = false;  for (int i = 0; i < matrix.size(); i++) {  for (int j = 0; j < matrix.size(); j++) {  if (matrix[i][j] == 1) {  return true;  }  }  }  return isAlive;  }  // start colony life  void StartColonyLife(vector<vector<int>> matrix) {  bool isAlive = true;  while (isAlive) {  for (int i = 0; i < matrix.size(); i++) {  for (int j = 0; j < matrix.size(); j++) {  if (matrix[i][j] == 1) {  // need to chamge current position of cell, move it  matrix = CreateStep(matrix, i, j);  }  }  }  cout << "Cells after moving:\n";  OutputMatrix(matrix);  matrix = KillCells(matrix);  cout << "\nCells after killing:\n";  OutputMatrix(matrix);  // check if colony is alive  isAlive = CheckIfColonyIsAlive(matrix);  if (!isAlive) {  cout << "Colony die\n";  return;  }  cout << "Continue colony life, enter 1: ";  int choice = 0;  cin >> choice;  if (choice != 1)  isAlive = false;  }  }  void main() {  int size = 0;  cout << "Input matrix size: ";  cin >> size;  vector<vector<int>> matrix = InitColony(size);  StartColonyLife(matrix);  } |