Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Факультет компьютерных технологий и прикладной математики**

**Кафедра вычислительных технологий**

**ЛАБОРАТОРНАЯ РАБОТА №5**

**Дисциплина: Распределенные задачи и алгоритмы**

Работу выполнила: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Д. Н. Баева

Направление подготовки: 02.03.02 Фундаментальная информатика и информационные технологии

Преподаватель: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ В. И. Шиян

**Тема работы:** Задачи с графами.

**Ход работы:**

В задании лабораторной работы необходимо было разработать алгоритм, который вычисляет диаметр произвольного неориентированного графа. Входными данными является матрица смежности.

Определим что необходимо вычислить. Диаметр графа – максимальная длина кратчайшего пути между двумя вершинами.

Первым шагом в программе происходит инициализация MPI ( определение ранга и размера коммуникатора). Затем выбирается корневой процесс (необходим для операции коллективной рассылки). Далее инициализируются переменная размера матрицы смежности, сама матрица смежности, матрица кратчайших расстояний между вершинами и переменная для диаметра графа.

В условии задания сказано, что граф неориентированный и произвольный, поэтому следующим шагом является создание матрицы смежности графа размером . Она заполняется случайными значениями от 0 до 9. Матрица симметрична и её диагональные элементы равны 0, так как граф неориентированный. После этого она выводится в консоль. На рисунке 1 представлен фрагмент кода, выполняющий это.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рисунок 1 – Инициализация матрицы смежности графа.

Полученная матрица в дальнейшем будет использоваться в операции коллективной рассылки. Необходимо её предварительно преобразовать одномерный массив для корректной работы методов рассылки. Фрагмент кода, выполняющий это, представлен на рисунке 2.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рисунок 2 – Преобразование двумерной матрицы смежности в одномерный массив.

После этого программа выполняет операцию коллективной рассылки массива смежности графа. Для этого используется соответствующий метод со следующими аргументами – MPI.COMM\_WORLD.Bcast(array\_matrix, 0, m, MPI.INT, root). Каждый процесс получает копию массива.

Для нахождения кратчайших путей в графе используется алгоритм Флойда-Уоршелла.

Алгоритм Флойда-Уоршелла — это алгоритм нахождения кратчайших путей между всеми парами вершин взвешенного ориентированного или неориентированного графа. Он работает во временную сложность где - количество вершин в графе. Алгоритм сравнивает все возможные пути через граф между каждой парой вершин. Он может сделать это за сравнений в графе, даже если в графе может быть до ребер, и каждая комбинация ребер проверяется. Это достигается путем постепенного улучшения оценки кратчайшего пути между двумя вершинами, пока оценка не станет оптимальной. Для каждой пары вершин может быть либо (1) путь, который не проходит через промежуточную вершину, либо (2) путь, который проходит через промежуточную вершину. При работе алгоритма используется динамическое программирование для вычисления кратчайших путей между всеми парами вершин.

Предварительно двумерная матрица смежности была скопирована в матрицу расстояний, необходимую для алгоритма.

Цикл for (k = 0; k < n; k++) перебирает все вершины графа. Внутри этого цикла выполняется двойной цикл for (i = rank; i < n; i += size) и for (j = 0; j < n; j++), который перебирает все пары вершин графа. Если существует путь через вершину k, то вычисляется длина этого пути с помощью формулы int newDist = dist[i][k] + dist[k][j]. Если длина этого пути короче текущего пути или текущего пути нет, то значение в матрице расстояний обновляется. Если i == j, то диагональный элемент матрицы расстояний устанавливается в 0. Каждый процесс отвечает за свою часть матрицы и для согласования шагов алгоритма используется барьерная синхронизация. Алгоритм представлен на рисунке 3.

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

Рисунок 3 – Поиск кратчайшего расстояния между двумя вершинами.

Аналогично тому как это было описано выше, полученная двумерная матрица кратчайших расстояний между двумя вершинами преобразуется в одномерный массив.

Затем с помощью метода MPI.COMM\_WORLD.Barrier() выполняется барьерная синхронизация (блокировка выполнения процессов до тех пор, пока все процессы не достигнут барьера) с другими процессами. Это гарантирует, что все процессы будут находиться в одной точке программы и не продолжат выполнение до тех пор, пока все процессы не достигнут этой точки. После этого метод MPI.COMM\_WORLD.Allgather() собирает данные от всех процессов и отправляет их также всем процессам. Полученная матрица кратчайших расстояний между двумя вершинами выводится в консоль.

В финальной части программы в матрице расстояний выполняется поиск максимального значения – диаметра (рисунок 4).

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Автоматически созданное описание

Рисунок 4 – Поиск диаметра графа.

На рисунке 5 представлена зависимость временных затрат от разного количества процессов.

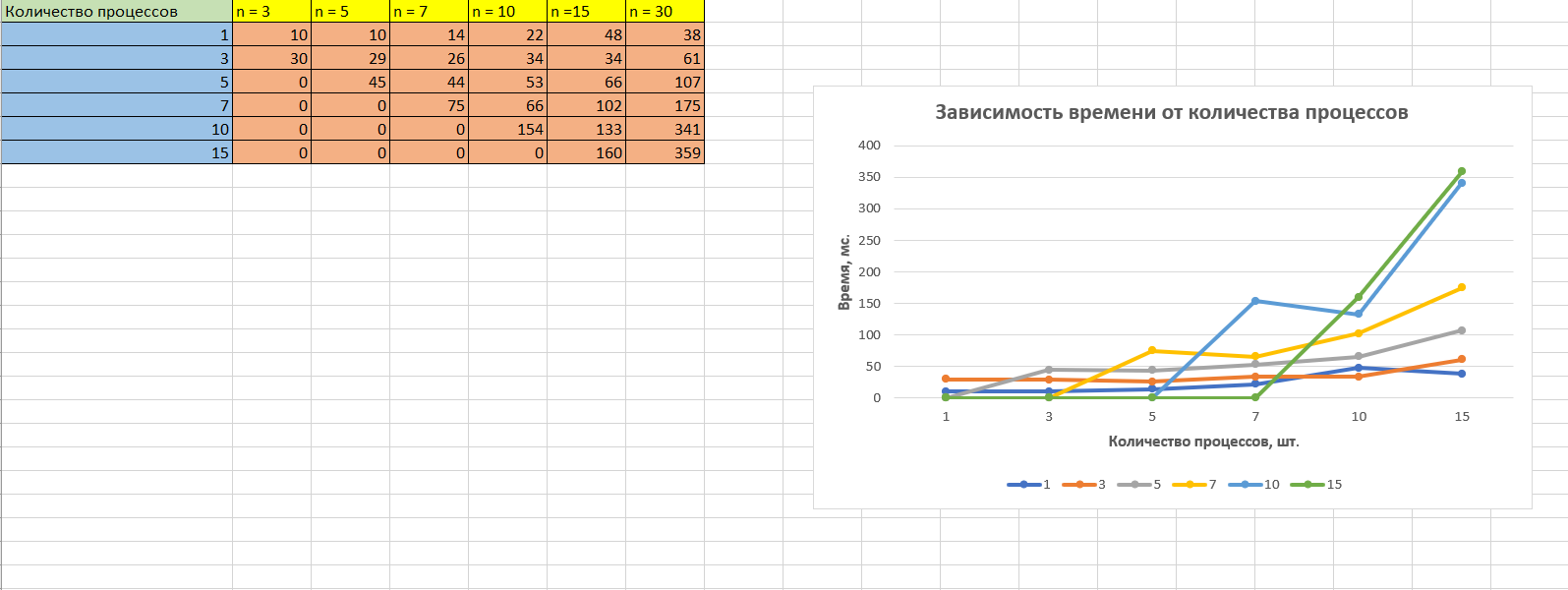


Рисунок 5 – Временная зависимость с разным количеством процессов.

Как видно из графика, время работы программы увеличивается с ростом количества процессов для каждого возможного случая размерности графа. Однако, если размерность графа меньше, чем количество процессов, то программа не выполняется, поскольку невозможно выполнение операции коллективной рассылки.

**Листинг программы**

Файл LW\_5.java

package LW\_5;  
  
import mpi.\*;  
  
import java.util.concurrent.TimeUnit;  
import com.google.common.base.Stopwatch;  
  
// вычисление диаметра произвольного неориентированного графа  
public class LW\_5 {  
 public static void main(String[] args) throws MPIException {  
 MPI.*Init*(args);  
 int rank = MPI.*COMM\_WORLD*.Rank();  
 int size = MPI.*COMM\_WORLD*.Size();  
 // выбираем корневой процесс  
 int root = 0;  
  
 Stopwatch stopwatch = Stopwatch.*createStarted*();  
 // размер матрицы смежности (количество вершин графа)  
 int n = 5;  
 // матрица смежности графа  
 int[][] matrix = new int[n][n];  
 // матрица кратчайших расстояний между вершинами  
 int[][] dist = new int[n][n];  
 // диаметр графа - максимальная длина кратчайшего пути между двумя вершинами  
 int diameter = 0;  
  
 if (rank == root)  
 {  
 stopwatch.reset().start();  
 // заполнение матрицы смежности случайными значениями от 0 до 9  
 // матрица симметрична, так как граф неориентированный  
 int i,j;  
 for (i = 0; i < n; i++)  
 {  
 for (j = i; j < n; j++)  
 {  
 if (i == j)  
 {  
 // диагональные элементы равны 0  
 matrix[i][j] = 0;  
 }  
 else  
 {  
 matrix[i][j] = (int) (Math.*random*() \* 10);  
 // симметричный элемент  
 matrix[j][i] = matrix[i][j];  
 }  
 }  
 }  
 System.*out*.println("Матрица смежности графа:");  
 *printMatrix*(matrix);  
 }  
  
 // преобразование двумерной матрицы смежности графа в одномерный массив (для операции коллективной рассылки)  
 int[] array\_matrix = null;  
 // определение размера нового массива  
 int m = matrix.length \* matrix[0].length;  
 array\_matrix = new int[m];  
 int i,j,l = 0;  
 for (i = 0; i < matrix.length; i++)  
 {  
 for (j = 0; j < matrix[i].length; j++)  
 {  
 array\_matrix[l] = matrix[i][j];  
 l++;  
 }  
 }  
  
 // рассылаем матрицу смежности всем процессам  
 MPI.*COMM\_WORLD*.Bcast(array\_matrix, 0, m, MPI.*INT*, root);  
  
 // копируем матрицу смежности в матрицу расстояний  
 for (i = 0; i < n; i++)  
 {  
 System.*arraycopy*(matrix[i], 0, dist[i], 0, n);  
 }  
  
 // применение алгоритма Флойда-Уоршелла для нахождения кратчайших путей  
 // каждый процесс отвечает за свою часть матрицы  
 // используем барьерную синхронизацию для согласования шагов алгоритма  
 int k;  
 for (k = 0; k < n; k++)  
 {  
 for (i = rank; i < n; i += size)  
 {  
 for (j = 0; j < n; j++)  
 {  
 // если есть путь через вершину k  
 if (dist[i][k] > 0 && dist[k][j] > 0)  
 {  
 // вычисляем длину этого пути  
 int newDist = dist[i][k] + dist[k][j];  
 // если он короче текущего пути или текущего пути нет  
 if (dist[i][j] == 0 || newDist < dist[i][j])  
 {  
 // обновляем значение в матрице расстояний  
 dist[i][j] = newDist;  
 }  
 }  
 if (i == j)  
 {  
 dist[i][j] = 0;  
 }  
 }  
 }  
  
 // преобразование двумерной матрицы кратчайших расстояний в одномерный массив (для операции коллективной рассылки)  
 int[] array\_dist = null;  
 // определение размера нового массива  
 int d = dist.length \* dist[0].length;  
 array\_dist = new int[d];  
 int t = 0;  
 for (i = 0; i < dist.length; i++) {  
 for (j = 0; j < dist[i].length; j++) {  
 array\_dist[t] = dist[i][j];  
 t++;  
 }  
 }  
  
 // синхронизируемся с другими процессами  
 MPI.*COMM\_WORLD*.Barrier();  
 // сбор данных от всех процессов и отправка их всем процессам  
 MPI.*COMM\_WORLD*.Allgather(array\_dist, 0, size,MPI.*INT*, array\_dist, 0,size, MPI.*INT*);  
 }  
  
 if (rank == root)  
 {  
 System.*out*.println("Матрица кратчайших расстояний между вершинами:");  
 *printMatrix*(dist);  
  
 // поиск максимального значения в матрице расстояний - диаметра  
 for (i = 0; i < n; i++) {  
 for (j = i + 1; j < n; j++)  
 {  
 if (dist[i][j] > diameter)  
 {  
 diameter = dist[i][j];  
 }  
 }  
 }  
 stopwatch.stop();  
 long elapsedTime = stopwatch.elapsed(TimeUnit.*MILLISECONDS*);  
 System.*out*.println("Диаметр графа: " + diameter);  
 System.*out*.println("Затраченное время: " + elapsedTime + " milliseconds");  
  
 }  
  
 MPI.*Finalize*();  
 }  
 // вспомогательный метод для вывода матрицы на экран  
 public static void printMatrix(int[][] matrix) {  
 for (int[] row : matrix) {  
 for (int element : row) {  
 System.*out*.print(element + " ");  
 }  
 System.*out*.println();  
 }  
 }  
}