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 №7**

**Дисциплина: Распределенные программные системы**

Работу выполнил: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Р. Р. Посевин

Направление подготовки: 02.03.03 Математическое обеспечение и администрирование информационных систем

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

Краснодар

2021

**Постановка задачи**

# Разработать алгоритм вычисления количества ребер в графе. Входные данные – матрица смежности. Выполнить подсчет временных затрат с разным количеством процессов, привести графики.

# **Ход работы**

# В целях исследования эффективности параллельного алгоритма, был реализован последовательный и параллельный алгоритмы.

# Параллельный алгоритм реализован с использованием методов коллективного для обмена (Scatter, Reduce). Использование других методов коллективного обмена в данной задачи не является целесообразным.

# Управляющий процесс в параллельном алгоритме разбивает матрицу смежности на равные, если возможно, части и отправляет всем остальным процессам. Если матрицу смежности не удается разбить на равные части, то остаток строк будет отправлен последнему подчиненному процессу.

# Каждый подчиненный процесс вычисляет сумму единиц (ребер) в полученной части матрицы смежности, после чего, управляющий процесс собирает сумму всех ребер с помощью метода Reduce.

Анализ эффективности реализованных алгоритмов проводился на квадратных матрицах размерности 3000, 5000 и 7500. Время работы последовательного алгоритма сравнивается со временем работы параллельного алгоритма с использованием 3, 5 и 10 процессов.

Результаты замеров времени представлены на рисунке 1. На рисунке 2 представлен график для результатов на рисунке 1.

Изображение выглядит как стол

Автоматически созданное описание

Рисунок 1 – Результаты исследования

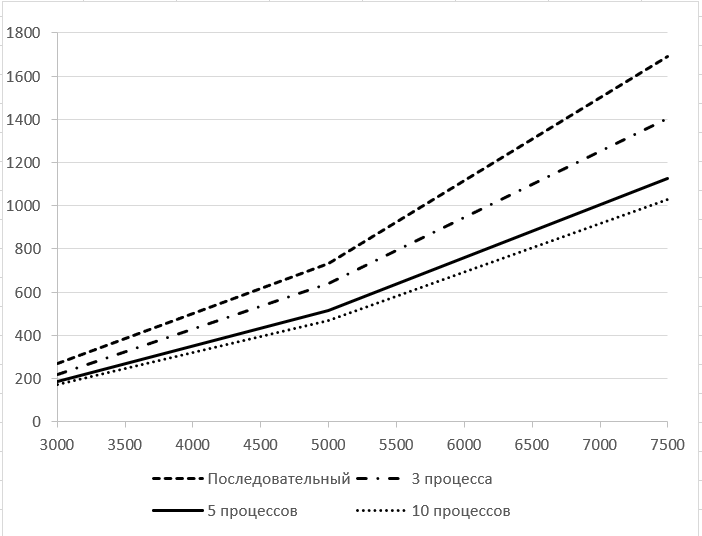


Рисунок 2 – График для результатов исследования

По итогам экспериментов видно, что параллельный алгоритм работает быстрее последовательного, вне зависимости от количества процессов. Однако разница не так велика.

В проведенном эксперименте количество процессов ускоряет работу программы.

Код программы:

import os  
import logging  
import random  
import time  
  
from mpi4py import MPI  
  
# Run command  
# mpiexec -np 2 py lab\_7.py  
  
RANDOM\_MODE = True  
TAG = 2  
# 2  
GRAPH = [  
 [0, 1, 1],  
 [1, 0, 0],  
 [1, 0, 0],  
]  
  
GRAPH\_SIZE = 7500  
  
file\_name = os.path.basename(\_\_file\_\_)  
logger = logging.getLogger(file\_name)  
logging.basicConfig(  
 filename='logs/logs.log',  
 level=logging.INFO,  
 format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',  
)  
comm = MPI.COMM\_WORLD  
rank = comm.Get\_rank()  
size = comm.Get\_size()  
  
  
def init\_test\_data():  
 def generate\_matrix(matrix\_size):  
 return [[random.randint(0, 1) for \_ in range(matrix\_size)] for \_ in range(matrix\_size)]  
  
 def get\_random\_graph(graph\_size):  
 graph = generate\_matrix(graph\_size)  
 for i in range(graph\_size):  
 for j in range(graph\_size):  
 if i == j:  
 graph[i][j] = 0  
 continue  
  
 graph[i][j] = graph[j][i]  
  
 return graph  
  
 def get\_matrix\_width(matrix):  
 lines\_width = {len(line) for line in matrix}  
 if len(lines\_width) > 1:  
 raise Exception('Incorrect matrix!')  
  
 return list(lines\_width)[0]  
  
 if RANDOM\_MODE:  
 return get\_random\_graph(GRAPH\_SIZE)  
  
 matrix = GRAPH  
  
 width = get\_matrix\_width(matrix)  
 if width != len(matrix):  
 raise Exception('Not correct matrix!')  
  
 if size > width:  
 raise Exception('Too many processes!')  
  
 return matrix  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
 if rank == 0:  
 graph = init\_test\_data()  
 formatted\_graph\_matrix = [  
 [x for x in line[:index]]  
 for index, line in enumerate(graph[1:], 1)  
 ]  
 matrix\_split\_len = len(formatted\_graph\_matrix) // (size - 1)  
  
 start\_time = time.time()  
 matrix\_for\_send = []  
 for proc\_rank in range(1, size):  
 if proc\_rank == size - 1:  
 matrix\_for\_send.append(formatted\_graph\_matrix[matrix\_split\_len \* (proc\_rank - 1):])  
 else:  
 matrix\_for\_send.append(formatted\_graph\_matrix[matrix\_split\_len \* (proc\_rank - 1): matrix\_split\_len \* proc\_rank])  
  
 scatter\_matrix = [None, \*matrix\_for\_send]  
 comm.scatter(  
 sendobj=scatter\_matrix,  
 root=0,  
 )  
  
 # Получение результатов от процессов  
 # Сумма в reduce по умолчанию  
 result = comm.reduce(  
 sendobj=0,  
 root=0,  
 )  
  
 result\_message = f'Result: {result}'  
 time\_message = 'Time: {} ms'.format(round((time.time() - start\_time) \* 1000))  
 print(result\_message)  
 print(time\_message)  
 else:  
 # Получение данных от 0 процесса  
 graph\_fragment = comm.scatter(sendobj=None, root=0)  
  
 if not RANDOM\_MODE:  
 print(f'{rank} A {graph\_fragment}')  
  
 result = sum(  
 [  
 sum(line)  
 for line in graph\_fragment  
 ]  
 )  
 comm.reduce(  
 sendobj=result,  
 root=0,  
 )

# **Вывод**

В ходе лабораторной работы 7 было проведено исследование зависимости времени выполнения многопоточной программы от количества потоков и размеров матрицы. Также было проведено сравнение эффективности работы последовательного и параллельного алгоритма.