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 №6**

**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**

# Вычислить произведение матриц С=А × В, где А матрица размера 𝑛1 × 𝑛2 и В матрица 𝑛2 × 𝑛3. Решите задачу с учетом знания принципов коллективных обменов.

# **Ход работы**

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

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

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

Анализ эффективности реализованных алгоритмов проводился на квадратных матрицах размерности 50, 100, 200 и 300. Время работы последовательного алгоритма сравнивается со временем работы параллельного алгоритма с использованием 5, 10 и 20 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

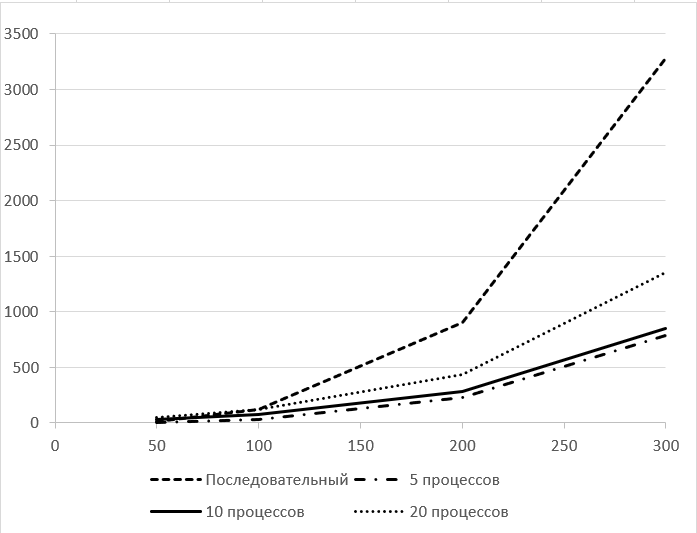


Рисунок 2 – График для результатов исследования

По итогам экспериментов видно, что параллельный алгоритм работает значительно быстрее последовательного, вне зависимости от количества процессов.

Для параллельных алгоритмов можно сделать вывод, что количество процессов не ускоряет работу алгоритма.

Код программы для последовательного алгоритма:

import random

import time

RANDOM\_MODE = True

MATRIX\_A = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9],

]

MATRIX\_B = [

[0, 0, 0, 1],

[3, 1, 2, 5],

[2, 3, 1, 7],

]

VECTORS\_LEN = 200

MIN\_NUMBER = -100

MAX\_NUMBER = 100

def init\_test\_data():

def generate\_matrix():

return [[random.randint(MIN\_NUMBER, MAX\_NUMBER) for \_ in range(VECTORS\_LEN)] for \_ in range(VECTORS\_LEN)]

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 generate\_matrix(), generate\_matrix()

matrix\_a = MATRIX\_A

matrix\_b = MATRIX\_B

width\_matrix\_a = get\_matrix\_width(matrix\_a)

\_ = get\_matrix\_width(matrix\_b)

if width\_matrix\_a != len(matrix\_b):

raise Exception('Cant multiple this matrix!')

return matrix\_a, matrix\_b

def vector\_multiply(vector\_a, vector\_b):

return sum([vector\_a[i] \* vector\_b[i] for i in range(len(vector\_a))])

if \_\_name\_\_ == '\_\_main\_\_':

matrix\_a, matrix\_b = init\_test\_data()

start\_time = time.time()

result = [

[vector\_multiply(line\_a, line\_b) for line\_b in zip(\*matrix\_b)]

for line\_a in matrix\_a

]

time\_message = 'Time: {} ms'.format(round((time.time() - start\_time) \* 1000))

print('Result:')

print(\*result, sep='\n')

print(time\_message)

Код программы для параллельного алгоритма:

import os

import logging

import random

import time

from mpi4py import MPI

# Run command

# mpiexec -np 2 py lab\_5.py

RANDOM\_MODE = True

TAG = 2

MATRIX\_A = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9],

]

MATRIX\_B = [

[0, 0, 0, 1],

[3, 1, 2, 5],

[2, 3, 1, 7],

]

VECTORS\_LEN = 300

MIN\_NUMBER = -100

MAX\_NUMBER = 100

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():

return [[random.randint(MIN\_NUMBER, MAX\_NUMBER) for \_ in range(VECTORS\_LEN)] for \_ in range(VECTORS\_LEN)]

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 generate\_matrix(), generate\_matrix(), VECTORS\_LEN // (size - 1)

matrix\_a = MATRIX\_A

matrix\_b = MATRIX\_B

width\_matrix\_a = get\_matrix\_width(matrix\_a)

\_ = get\_matrix\_width(matrix\_b)

if width\_matrix\_a != len(matrix\_b):

raise Exception('Cant multiple this matrix!')

matrix\_a\_len = len(matrix\_a)

if size - 1 > matrix\_a\_len:

raise Exception('Too many processes!')

return matrix\_a, matrix\_b, matrix\_a\_len // (size - 1)

def vector\_multiply(vector\_a, vector\_b):

return sum([vector\_a[i] \* vector\_b[i] for i in range(len(vector\_a))])

if \_\_name\_\_ == '\_\_main\_\_':

if rank == 0:

matrix\_a, matrix\_b, matrix\_split\_len = init\_test\_data()

start\_time = time.time()

matrix\_for\_send = []

for proc\_rank in range(1, size):

if proc\_rank == size - 1:

matrix\_for\_send.append(matrix\_a[matrix\_split\_len \* (proc\_rank - 1):])

else:

matrix\_for\_send.append(matrix\_a[matrix\_split\_len \* (proc\_rank - 1): matrix\_split\_len \* proc\_rank])

scatter\_matrix = [None] + matrix\_for\_send

logger.info(scatter\_matrix)

comm.scatter(

sendobj=scatter\_matrix,

root=0,

)

comm.bcast(

obj=list(zip(\*matrix\_b)),

root=0,

)

# Получение результатов от процессов

result = comm.gather(

sendobj=None,

root=0,

)

# Соединение результатов в общую матрицу

result\_matrix = []

for line in result[1:]:

result\_matrix.extend(line)

result\_message = 'Result:'

time\_message = 'Time: {} ms'.format(round((time.time() - start\_time) \* 1000))

print(result\_message)

print(\*result\_matrix, sep='\n')

print(time\_message)

else:

# Получение данных от 0 процесса

fragment\_from\_a = comm.scatter(sendobj=None, root=0)

if not RANDOM\_MODE:

print(f'{rank} A {fragment\_from\_a}')

matrix\_b = comm.bcast(obj=None, root=0)

if not RANDOM\_MODE:

print(f'{rank} B {matrix\_b}')

result = [

[vector\_multiply(line\_a, line\_b) for line\_b in matrix\_b]

for line\_a in fragment\_from\_a

]

comm.gather(

sendobj=result,

root=0,

)

# **Вывод**

В ходе лабораторной работы 6 было проведено исследование зависимости времени выполнения многопоточной программы от количества потоков и размеров матрицы. Также было проведено сравнение эффективности работы последовательной и параллельной алгоритма.