САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО
ИНСТИТУТ КОМПЬЮТЕРНЫХ НАУК
КАФЕДРА КОМПЬЮТЕРНЫХ СИСТЕМ И ПРОГРАММНЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе

Курс: «Параллельные вычисления»

Тема: «Определить вероятность появления 3-грамм в тексте на русском языке»

Работу выполнила: Шурыгина Н. А.
Группа 91501
Работу принял: Стручков И. В.

Санкт-Петербург
2020

СОДЕРЖАНИЕ

Индивидуальное задание

Определить вероятность появления 3-грамм в тексте на русском языке. Под 3-граммами будем понимать 3 подряд идущих слова, исключая цифры, знаки препинания и латиницу.

Используемое окружение

- ОС: macOS Catalina
- Версия ОС: 10.15.4
- Процессор: 1,4 GHz Quad-Core Intel Core i5
- RAM: 16 ГБ
- Python 3
- mpi4py 3.0.3

Алгоритм решения

Разработка производилась на языке Python 3. Первым делом выполнялось чтение текста из файлов. Далее производилась очистка текста от латиницы, цифр, знаков препинания и переносов строк. Затем текст разбивается на слова, и считаются все возможные комбинации из трех слов, идущих подряд. Каждая найденная комбинация слов помещается в хэш-таблицу. Так удобно считать количество вхождений отдельных 3-грамм в текст и вычислять вероятность их появления. После обнаружения и подсчета всех троек слов, данные записываются в файл.

Параллельный алгоритм решения

Чтобы ускорить поиск 3-грамм в русском тексте, можно внести в алгоритм следующую модификацию: пусть чтение из файла, очистка текста и запись в файл происходят последовательно, а поиск 3-грамм и подсчет вероятности их появления — параллельно. Это сделано с использованием методов MPI: scatter (распределим порции текста между процессами) и gather (соберем данные из процессов и объединим их). Исходный код последовательной и параллельной реализации представлен в Приложении.

Тестирование

На рисунке 1 представлен результат работы программы для небольшого (~4 KB) текста. Посчитаны все тройки слов, встречающиеся в тексте подряд и вычислена их вероятность появления.

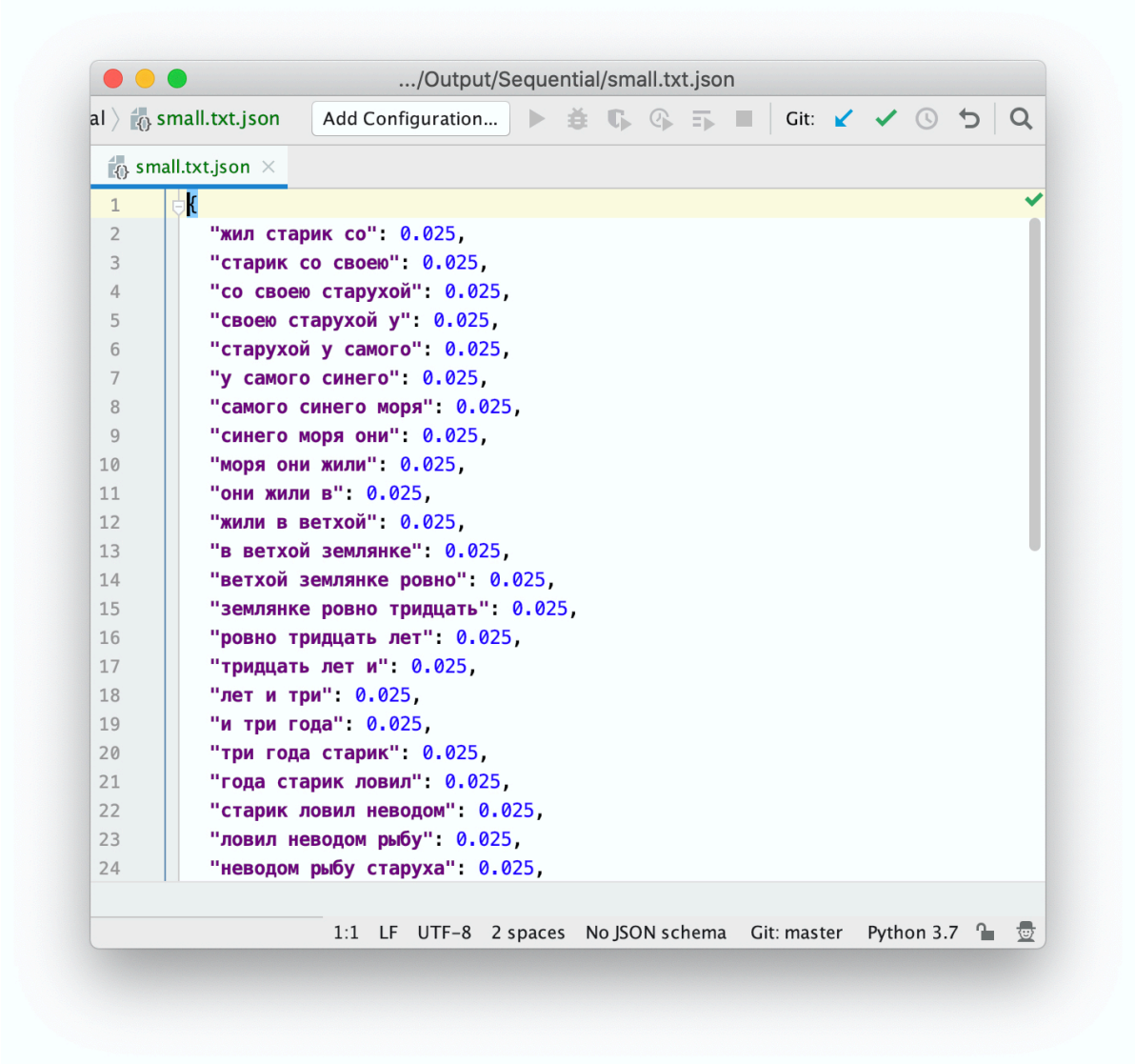

Рисунок 1. Пример выходных данных программы

Время работы программы зависит от количества процессов и объема входных данных. Опытным путем было выяснено, что при параллельном выполнении программы прирост скорости можно получить только при двух задействованных процессах. При задействовании трех и четырех уже происходит деградация. Скорее всего, это связано с особенностями ПК, на котором запускается программа. Результаты запуска при разном количестве процессов приведены в таблице 1. Размер входного текста ~20 MB.

Таблица 1. Время работы программы в зависимости от количества задействованных процессов.

Количество процессов	Время выполнения, сек
1	8.623248
2	8.295741
3	8.33155
4	8.786753

Далее было произведено по 50 запусков последовательной и параллельной (два процесса) реализации с разными объемами входных данных:

- ~ 400 KB
- ~ 20 MB
- ~ 60 MB

По результатам запусков были рассчитаны математическое ожидание времени выполнения программы, дисперсия и доверительный интервал в зависимости от объема входных данных. Графики полученных зависимостей представлены на рисунках 2 и 3.

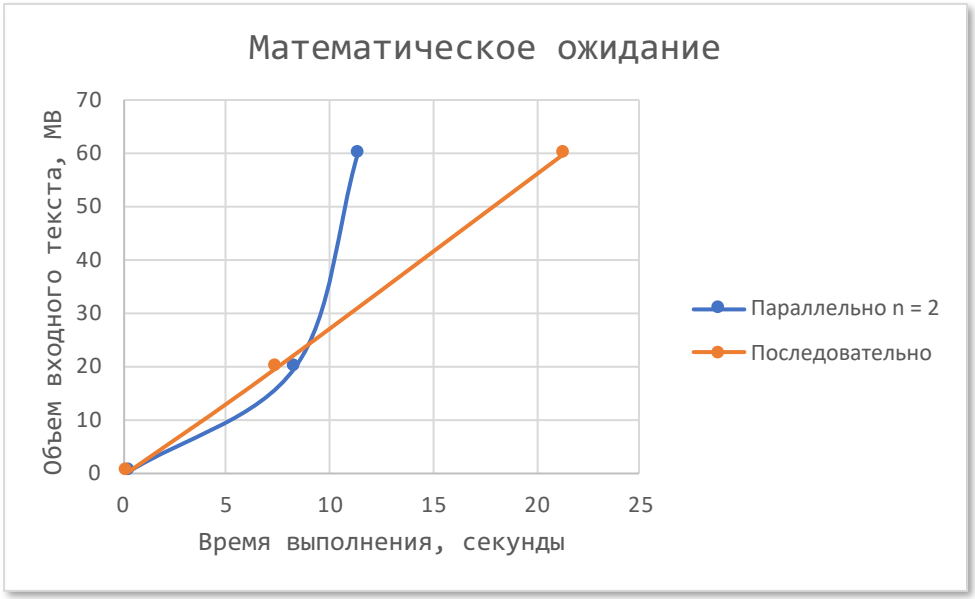

Рисунок 2. Математическое ожидание времени выполнения программы

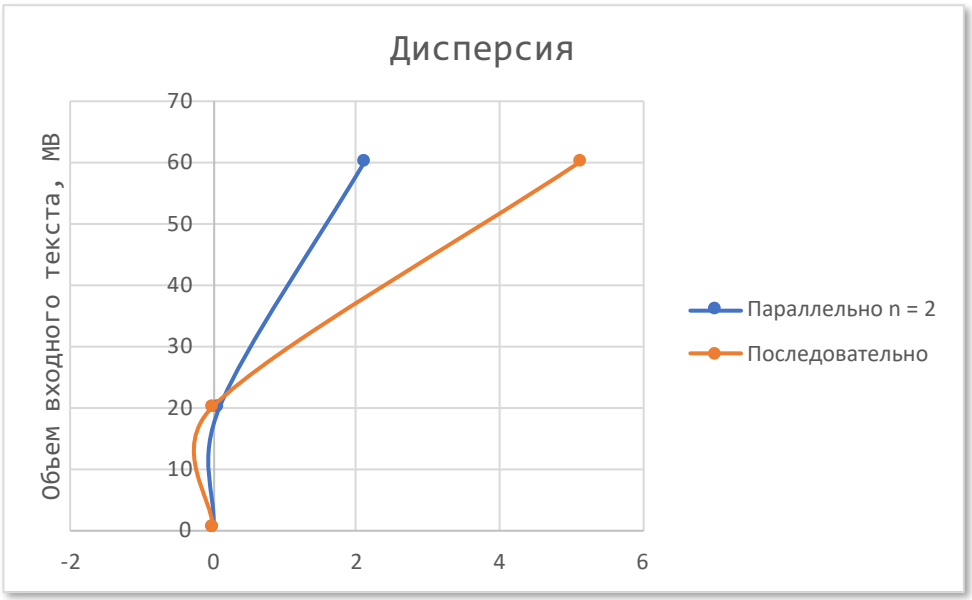

Рисунок 3. Дисперсия времени выполнения программы

Из рисунков 2-3 видно, что при маленьких объемах входных данных более эффективно работает последовательный алгоритм: затраты, связанные с распределением данных по процессам, не оправдывают себя. Однако при большом объеме входных данных (~25 MB) параллельный алгоритм справляется быстрее, и от запуска к запуску имеет более стабильный характер — дисперсия значительно меньше.

Таблица 2. Рассчитанные статистические характеристики по результатам 50 запусков. Объем входных данных ~ 400 KB

Характеристика	Последовательная реализация	Параллельная реализация
Математическое ожидание	0.200648 сек	0.289253 сек
Дисперсия	0.000869	0.005450
Доверительный интервал (P = 0.95)	[0.199660 сек, 0.201637 сек]	[0.286778 сек, 0.291729 сек]

Таблица 3. Рассчитанные статистические характеристики по результатам 50 запусков. Объем входных данных ~ 20 MB

Характеристика	Последовательная реализация	Параллельная реализация
Математическое ожидание	7.404066 сек	8.340439 сек
Дисперсия	0.003106	0.093601
Доверительный интервал (P = 0.95)	[7.402197 сек, 7.405934 сек]	[8.330180 сек, 8.350697 сек]

Таблица 4. Рассчитанные статистические характеристики по результатам 50 запусков. Объем входных данных ~ 60 MB

Характеристика	Последовательная реализация	Параллельная реализация
Математическое ожидание	21.270766 сек	11.363841сек
Дисперсия	5.136735	2.131938
Доверительный интервал (P = 0.95)	[21.194770 сек, 21.346761 сек]	[11.314882 сек, 11.412800 сек]

Выводы

В рамках данной лабораторной работы:

- разработана программа на Python 3, вычисляющее вероятность появления N-грамм в русском тексте;
- для распараллеливания исходной задачи использована библиотека OpenMPI;
- выполнено 50 запусков параллельной и последовательной реализации с разными объёмами входных данных, и на основе полученных результатов были рассчитаны статистические характеристики: математическое ожидание, дисперсия и доверительный интервал;
- при больших объемах входных данных параллельная реализация осуществила поиск 3-грамм в тексте в 2 раза быстрее, чем последовательная.

Приложение

Приложение 1. Последовательная реализация поиска 3-грамм в тексте

```
import json
import os
import re
import sys
import time

"""This method list of all inputs.
Should be changed as per the input format"""

startTime = time.time()

def buildInput(basePath):
    """Generate list of all possible files in directory
    and subdirectory, with depth 1"""
    pathList = []
    for dir in os.listdir(basePath):
        for file in os.listdir(os.path.join(basePath, dir)):
            pathList.append(os.path.join(basePath, dir, file))
    return pathList


def readFile(path):
    with open(path, 'r', errors='surrogateescape') as inputFile:
        # ("Executing for " + inputFile.name)
        text = inputFile.read()
    return text


def writeToFile(path, text):
    with open(path, 'w') as outputFile:
        outputFile.write(text)


def cleanText(text):
    """Convert text to lower case"""
    text = text.lower()
    """Remove special characters + email addresses + alpha numeric entries"""
    text = re.sub(r'\S*@\S*\s?|([^\s\w]|_)+|\w*\d\w*|[^A-Яа-я0-9\s]|^\d+\s|\s\d+\s|\s\d+$',
'', text)
    """remove new lines"""
    text = text.replace("\n", " ")
    """Replace more than one tabs with space"""
    text = re.sub('\t+', ' ', text)
    """Finally remove more than one spaces with space"""
    text = re.sub(' +', ' ', text)
    return text


def generateNormalizedTermFrequency(tokens, documentFrequency):
    termFrequency = {}
    tokenCount = len(tokens)
    for token in tokens:
        if token in termFrequency:
            """Division by tokenCount in document will normalize the termfrequency"""
            termFrequency[token] = termFrequency[token] + 1.0 / tokenCount
        else:
            termFrequency[token] = 1.0 / tokenCount

            if token in documentFrequency:
                documentFrequency[token] = documentFrequency[token] + 1
            else:
                documentFrequency[token] = 1
    return termFrequency, documentFrequency
```

```
def wordgramGenerator(tokens, n):
    ngrams = []
    for index in range(0, len(tokens)):
        candidate = tokens[index:index + n]
        ngram = ' '.join(candidate)
        ngrams.append(ngram)
    return ngrams


def ngramGenerator(text, ngramRange):
    tokens = text.split(" ")
    ngrams = []
    for i in range(ngramRange[0], ngramRange[1] + 1):
        ngrams.extend(wordgramGenerator(tokens, i))
    return ngrams

"""read input path as command line parameter"""
inputPath = sys.argv[1]
outputPath = sys.argv[2]
ngramS = int(sys.argv[3])
ngramE = int(sys.argv[4])

pathList = None

documentFrequency = {}

if not os.path.exists(outputPath):
    os.makedirs(outputPath)

if not os.path.exists(os.path.join(outputPath, "Sequential")):
    os.makedirs(os.path.join(outputPath, "Sequential"))

pathList = buildInput(inputPath)

"""Calculate term frequency and the relative document frequency
Here first I am reading text from the input file.
Followed by clean up of text file. The cleanText procedure will remove
characters."""
for file in pathList:
    text = readFile(file)
    text = cleanText(text)
    tokens = ngramGenerator(text, (ngramS, ngramE))
    tokenTermFrequency, documentFrequency = generateNormalizedTermFrequency(tokens,
documentFrequency)
    writeToFile(os.path.join(outputPath, "Sequential", os.path.basename(file) + ".json"),
                json.dumps(tokenTermFrequency, indent=2, ensure_ascii=False))

endTime = time.time()
print(str((endTime - startTime)))
```

Приложение 2. Параллельная реализация поиска 3-грамм в тексте

```
import json
import os
import re
import sys

import numpy as np
from mpi4py import MPI

"""This method list of all inputs.
Should be changed as per the input format"""


def buildInput(basePath):
    """Generate list of all possible files in directory
    and subdirectory, with depth 1"""
    pathList = []
```

```
    for dir in os.listdir(basePath):
        for file in os.listdir(os.path.join(basePath, dir)):
            pathList.append(os.path.join(basePath, dir, file))
    return pathList


def readFile(path):
    with open(path, 'r', errors='surrogateescape') as inputFile:
        # print("Executing for " + inputFile.name)
        text = inputFile.read()
    return text


def writeToFile(path, text):
    with open(path, 'w') as outputFile:
        outputFile.write(text)


def cleanText(text):
    """Convert text to lower case"""
    text = text.lower()
    """Remove special characters + email addresses + alpha numeric entries"""
    text = re.sub(r'\S*@\S*\s?|([^\s\w]|_)+|\w*\d\w*|[^A-Яа-я0-9\s]|^\d+\s|\s\d+\s|\s\d+$',
'', text)
    """remove new lines"""
    text = text.replace("\n", " ")
    """Replace more than one tabs with space"""
    text = re.sub('\t+', ' ', text)
    """Finally remove more than one spaces with space"""
    text = re.sub(' +', ' ', text)
    return text


def generateNormalizedTermFrequency(tokens, documentFrequency):
    termFrequency = {}
    tokenCount = len(tokens)
    for token in tokens:
        if token in termFrequency:
            """Division by tokenCount in document will normalize the termfrequency"""
            termFrequency[token] = termFrequency[token] + 1.0 / tokenCount
        else:
            termFrequency[token] = 1.0 / tokenCount

            """Update copy of local document frequency"""
            if token in documentFrequency:
                documentFrequency[token] = documentFrequency[token] + 1
            else:
                documentFrequency[token] = 1
    return termFrequency, documentFrequency


def wordgramGenerator(tokens, n):
    ngrams = []
    for index in range(0, len(tokens)):
        candidate = tokens[index:index + n]
        ngram = ' '.join(candidate)
        ngrams.append(ngram)
    return ngrams


def ngramGenerator(text, ngramRange):
    tokens = text.split(" ")
    ngrams = []
    for i in range(ngramRange[0], ngramRange[1] + 1):
        ngrams.extend(wordgramGenerator(tokens, i))
    return ngrams


"""MPI variables initialization"""
comm = MPI.COMM_WORLD
processorCount = comm.Get_size()
```

```
currentRank = comm.Get_rank()
root = 0

"""read input path as command line parameter"""
inputPath = sys.argv[1]
outputPath = sys.argv[2]
ngramS = int(sys.argv[3])
ngramE = int(sys.argv[4])

taskChunks = None
pathList = None

termFrequencies = []
documentFrequency = {}

"""Only root will execute the following code to generate possible inputs paths.
These input paths will be stored on a list. This list is
further broken into smaller parts
so that it can be passed on to workers"""
if currentRank == root:
    startTime = MPI.Wtime()
    if not os.path.exists(outputPath):
        os.makedirs(outputPath)

    if not os.path.exists(os.path.join(outputPath, "Parallel")):
        os.makedirs(os.path.join(outputPath, "Parallel"))

    pathList = buildInput(inputPath)
    taskChunks = np.array_split(pathList, processorCount)

"""Divide task among workers"""
inputFiles = comm.scatter(taskChunks, root)

"""Calculate term frequency and the relative document frequency
Here first I am reading text from the input file.
Followed by clean up of text file. The cleanText procedure will remove
characters"""
for file in inputFiles:
    text = readFile(file)
    text = cleanText(text)
    tokens = ngramGenerator(text, (ngramS, ngramE))
    tokenTermFrequency, documentFrequency = generateNormalizedTermFrequency(tokens,
documentFrequency)
    writeToFile(os.path.join(outputPath, "Parallel", os.path.basename(file) + ".json"),
                json.dumps(tokenTermFrequency, indent=2, ensure_ascii=False))
    termFrequencies.append(tokenTermFrequency)

"""Gather all the relative document frequency"""
relativeDocumentFrequencies = comm.gather(documentFrequency, root)

if currentRank == root:

    documentFrequency = {}
    vectorCount = 0
    totalDocuments = len(pathList)

    for relativeDocumentFrequency in relativeDocumentFrequencies:
        for token in relativeDocumentFrequency:
            if token in documentFrequency:
                documentFrequency[token] = documentFrequency[token] +
relativeDocumentFrequency[token]
            else:
                documentFrequency[token] = relativeDocumentFrequency[token]
                vectorCount += 1

if currentRank == root:
    endTime = MPI.Wtime()
    print(str((endTime - startTime)))
```