Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Отчет по лабораторной работе

Дисциплина: Параллельные вычисления

Тема: Создание многопоточных программ на языке C++ с использованием Pthreads и OpenMP

Выполнил
студент гр. 13541/2 _____ Зобков Д. А.
(подпись)

Преподаватель _____ Стручков И. В.
(подпись)

“___”_____ 2018 г.

Санкт-Петербург
2018 г.

СОДЕРЖАНИЕ

1 Постановка задачи 3
1.1 Индивидуальное задание 3
1.2 Программа работы 3

2 Сведения о системе 3

3 Структура проекта 4

4 Алгоритм решения 5
4.1 Последовательная реализация 5
4.2 Параллельная реализация с использованием pthreads 6
4.3 Параллельная реализация с использованием OpenMP 6

5 Тестирование 7
5.1 Эксперимент 1 7
5.2 Эксперимент 2 8
5.3 Эксперимент 3 9

Вывод 10

Приложение А Исходный код 11

1 Постановка задачи

1.1 Индивидуальное задание

Вариант 10, OpenMP.
Определить частоту встречи слов в тексте на русском языке.

1.2 Программа работы

1. Для алгоритма из полученного задания написать последовательную программу на языке C или C++, реализующую этот алгоритм.
2. Для созданной последовательной программы необходимо написать 3-5 тестов, которые покрывают основные варианты функционирования программы. Для создания тестов можно воспользоваться механизмом Unit-тестов среды NetBeans, или описать входные тестовые данные в файлах. При использовании NetBeans необходимо в свойствах проекта установить ключ компилятора -pthread.
3. Проанализировать полученный алгоритм, выделить части, которые могут быть распараллелены, разработать структуру параллельной программы. Определить количество используемых потоков, а также правила и используемые объекты синхронизации.
4. Согласовать разработанную структуру и детали реализации параллельной программы с преподавателем.
5. Написать код параллельной программы и проверить ее корректность на созданном ранее наборе тестов. При необходимости найти и исправить ошибки.
6. Провести эксперименты для оценки времени выполнения последовательной и параллельной программ. Проанализировать полученные результаты.
7. Сделать общие выводы по результатам проделанной работы:
   - Различия между способами проектирования последовательной и параллельной реализаций алгоритма.
   - Возможные способы выделения параллельно выполняющихся частей, Возможные правила синхронизации потоков.
   - Сравнение времени выполнения последовательной и параллельной программ.
   - Принципиальные ограничения повышения эффективности параллельной реализации по сравнению с последовательной.

2 Сведения о системе

Работа производилась на реальной системе, со следующими характеристиками:

Таблица 2.1. Сведения о системе

ОС	Windows 10 Домашняя для одного языка
Версия	Версия 1709 (Сборка ОС 16299.431)
Установленная оперативная память (RAM)	8 ГБ
Процессор	Intel(R) Core(TM) i7-3537U CPU @ 2.00 GHz 2.50 GHz, ядер: 2, логических процессоров: 4

3 Структура проекта

```
project
├─source code
│  ├─main.cpp
│  └─utils.cpp
└─header files
   └─utils.h
```

Рис. 3.1. Структура проекта

Точка входа расположена в файле **main.cpp**, в котором вызываются необходимые функции, реализованные в **utils.cpp**.

Текст для анализа загружается из файла построчно и хранится в переменной типа string.

Итоговый список встреченных слов в тексте представлен в виде unordered_map<string, int> со словом в качестве ключа и количеством встреч в тексте в качестве значения.

Стоит обратить внимание на функцию `getStringPart()` в файле **utils.cpp**, которая позволяет скорректировать подстроку таким образом, чтобы в ее конце не оказалось обрезанного слова и разделителей, а в начале разделителей. Для ее корректной работы необходимо, чтобы подстрока начиналась с начала слова либо разделителя. Таким образом, при исполнении в цикле для следующей подстроки в качестве стартового индекса используется конечный индекс предыдущей подстроки, полученный при использовании функции.

Функция `getCutIndex()` используется в качестве псевдонима функций find_* для большей читаемости кода.

```
1 // beginInd must be at word start or delimiter
2 std::pair<int, int> getStringPart(const std::string &str, unsigned int
    beginInd, unsigned int endInd) {
        std::pair<int, int> ind;
4
        // cut delimiter if needed from start
6       ind.first = (getCutIndex(str, beginInd, WORD_0) == beginInd) ?
    getCutIndex(str, beginInd, SPACE_0) : beginInd;
```

```
        // cut delimiter if needed or restore cut word from end
        ind.second = (getCutIndex(str, endInd, WORD_N) == endInd) ?
   getCutIndex(str, endInd, SPACE_N) : getCutIndex(str, endInd, WORD_0)
    - 1;

        return ind;
}

unsigned int getCutIndex(const std::string &str, int pos, cutType type)
    {
        std::string delim = " .,!?&@():;-|\"\r\n";
        switch(type) {
                case(SPACE_0):
                        return str.find_first_not_of(delim, pos);
                case(SPACE_N):
                        return str.find_last_not_of(delim, pos);
                case(WORD_0):
                        return str.find_first_of(delim, pos);
                case(WORD_N):
                        return str.find_last_of(delim, pos);
                default:
                        return -1;
        }
}
```

Листинг 3.1. отрывок utils.cpp

Полный исходный код приведен в листинге А.3 на с. 15.

4 Алгоритм решения

4.1 Последовательная реализация

Алгоритм заключается в вызове функции `getWordDict()`, которая проходит по строке, находит слова и формирует unordered_map.

```
std::unordered_map<std::string, int> getWordDict(const std::string &str
   , unsigned int strBegin, unsigned int strEnd) {
        std::unordered_map<std::string, int> dict;

        while((strBegin = getCutIndex(str, strBegin, SPACE_0)) < strEnd
   ){
                int cutEnd = getCutIndex(str, strBegin, WORD_0);
                int strLength = cutEnd - strBegin;
                std::string temp = str.substr(strBegin, strLength);

                strBegin = getCutIndex(str, cutEnd, SPACE_0);
                dict[temp] += 1;
        }

        return dict;
}
```

Листинг 4.1. отрывок utils.cpp

4.2 Параллельная реализация с использованием pthreads

В отличии от последовательной реализации, в данном случае накладывается ограничение на количество возможных потоков.

Общий алгоритм остается тем же, но перед запуском потоков исходная строка разбивается на число подстрок равное числу потоков и корректируются функцией `getStringPart()`, чтобы в подстроках не было обрезанных слов либо слов, попавших в обе подстроки. Затем, после получения массива unordered_map, происходит их слияние в одном потоке (функция `mergeDict()` в листинге А.3 на с. 15).

```
void *getWordDict_pthread(void *args) {
        getWordDictArgs *arg = (getWordDictArgs *)args;
        std::unordered_map<std::string, int> dict;

        if(!arg->str.empty()) {
                while((arg->strBegin = getCutIndex(arg->str, arg->strBegin, SPACE_0)) < arg->strEnd){
                        int cutEnd = getCutIndex(arg->str, arg->strBegin, WORD_0);
                        int strLength = cutEnd - arg->strBegin;
                        std::string temp = arg->str.substr(arg->strBegin, strLength);

                        arg->strBegin = getCutIndex(arg->str, cutEnd, SPACE_0);
                        dict[temp] += 1;
                }
        }

        arg->dict = dict;
        return 0;
}
```

Листинг 4.2. отрывок utils.cpp

4.3 Параллельная реализация с использованием OpenMP

Алгоритм для OpenMP аналогичен Pthreads. Благодаря использованию директив, удалось сильно сократить количество строк реализации.

В данном случае, была использована директива `#pragma omp parallel for` для параллельного вызова функции `getWordDict()`.

```
omp_set_num_threads(threadCount);
#pragma omp parallel for
        for(int i = 0; i < threadCount; i++) {
                dicts[i] = getWordDict(args[i].str, args[i].strBegin, args[i].strEnd);
        }
```

Листинг 4.3. отрывок main.cpp

Полный исходный код приведен в листинге А.1 на с. 11.

5 Тестирование

В качестве исходного теста использовалась книга „Гарри Поттер и Орден Феникса“, увеличение текста производилось удвоением содержимого.

5.1 Эксперимент 1

Количество потоков: 4 (равно числу логических процессоров)
Количество слов в тексте: 269374 – 8619968

Таблица 5.1. Зависимость от числа слов в тексте

Число слов в тексте	Последовательный	OpenMP	Pthreads
269374	0.403	**0.206**	0.226
538748	0.813	0.433	**0.429**
1077496	1.607	**0.793**	0.831
2154992	3.387	**1.622**	1.697
4309984	6.759	**3.201**	3.281
8619968	13.078	6.391	**6.294**

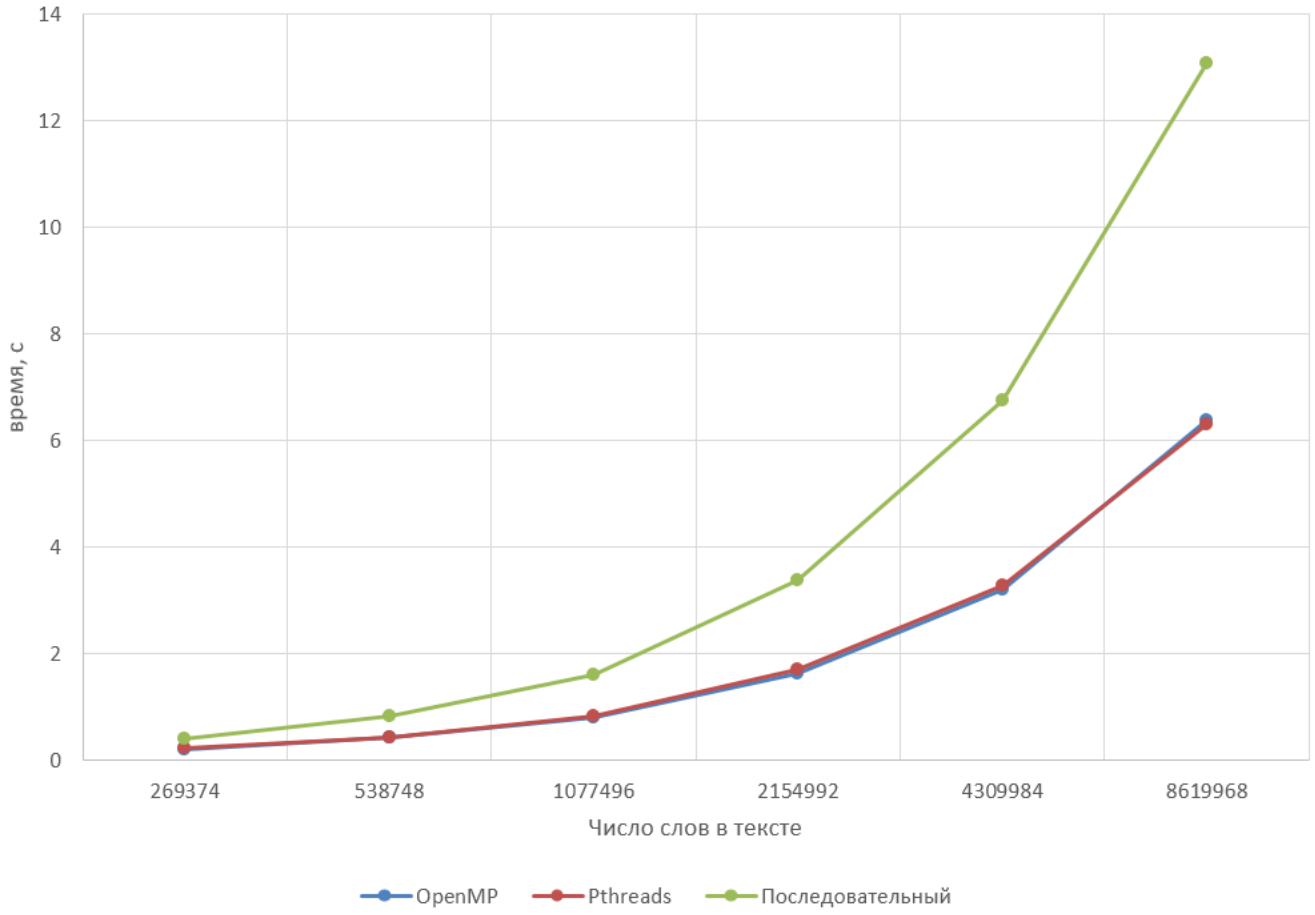

Рис. 5.1. Зависимость времени от числа слов в тексте

Из эксперимента видно, что параллельные решения работают приблизительно в два раза быстрее, чем последовательное решение. По скорости выполнения задачи параллельные решения в целом показывали близкие друг к другу результаты.

5.2 Эксперимент 2

Количество потоков: 1 – 128
Количество слов в тексте: 2154988

Таблица 5.2. Зависимость от количества потоков

Число потоков	Последовательный	OpenMP	Pthreads
1	3.345	**3.210**	3.240
2	-	2.125	**2.062**
4	-	**1.547**	1.627
8	-	**1.619**	1.672
16	-	**1.720**	1.797
32	-	**1.872**	1.941
64	-	**2.069**	2.293
128	-	**2.543**	2.688

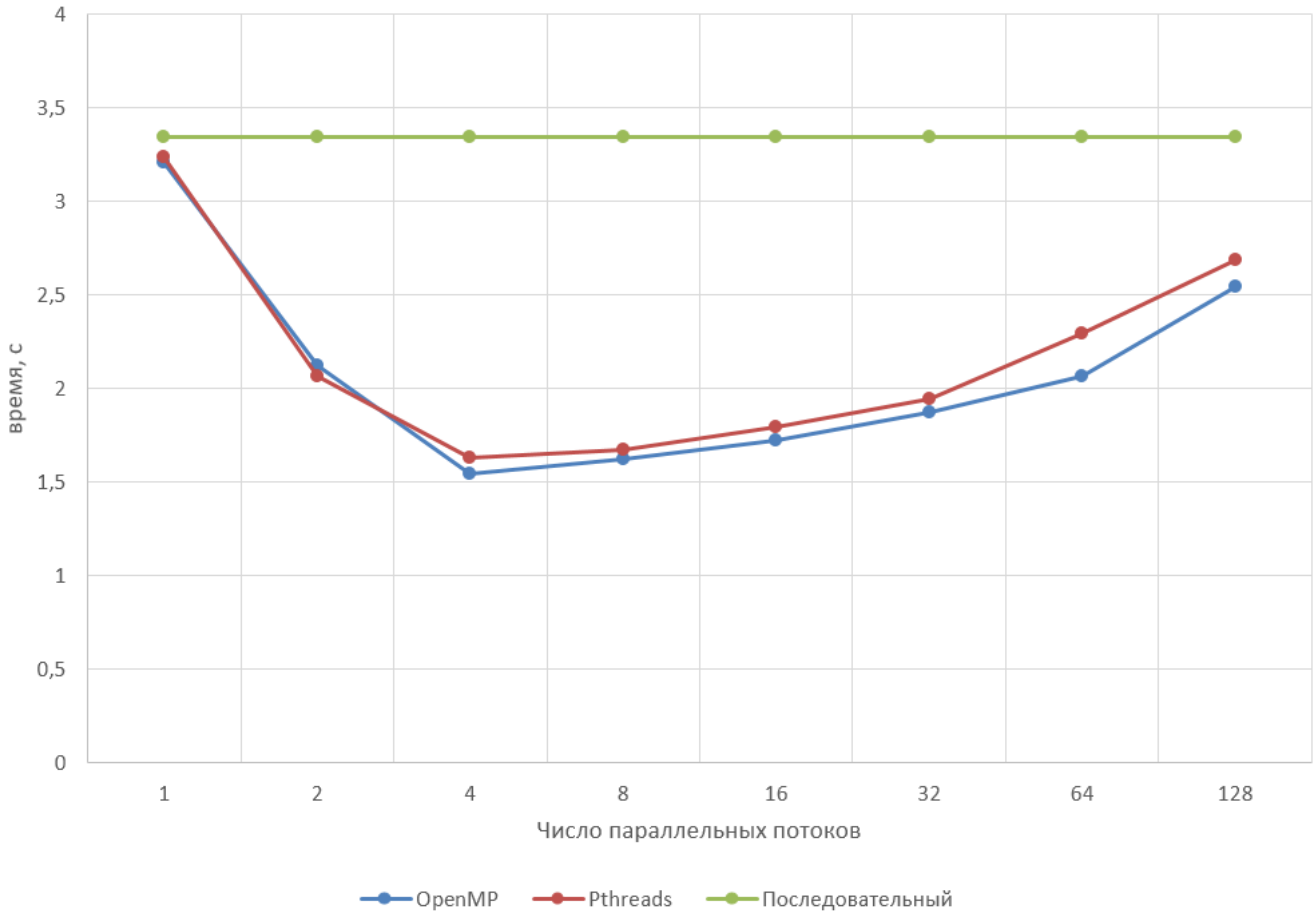

Рис. 5.2. Зависимость от числа выделенных потоков

Наилучшие показатели были получены при 4 потоках, прирост производительности при этом числе потоков составил:

- 54% — OpenMP;
- 51% — Pthreads.

5.3 Эксперимент 3

В данном эксперименте проводится многократный запуск при одних и тех же характеристиках, для того чтобы вычислить:

- математическое ожидание;
- дисперсию;
- доверительный интервал для оценки среднего,

что позволит более объективно оценить результаты алгоритмов.

Количество потоков: 4
Количество слов в тексте: 4309976

Таблица 5.3. Тестовая выборка для анализа

Последовательный	OpenMP	Pthreads
6.465	3.089	3.126
6.522	3.081	3.144
6.471	3.063	3.132
6.602	3.142	3.122
6.516	3.045	3.143
6.441	3.076	3.273
7.529	3.152	3.377
6.666	3.151	3.239
6.632	3.123	3.203
6.882	3.362	3.145

Таблица 5.4. Вероятностные характеристики

Характеристика	Последовательный	OpenMP	Pthreads
Среднее значение	6.6726	3.1284	3.1904
Дисперсия	0.00451584	0.00047524	0.000378951
Доверительный интервал (P = 0.95)	[6.469278635 - 6.669349082]	[3.072340462 - 3.127503661]	[3.138601712 - 3.189571794]

Как видно из представленных характеристик, **OpenMP** является лучшим решением. Средняя скорость выполнения с его использованием немного выше, чем у pthreads.

Вывод

В данной работе были рассмотрены методы организации параллельных вычислений в программах с использованием OpenMP и pthreads.

Реализация на OpenMP проще и занимает меньшее количество строк кода по сравнению с pthreads. Например, в pthreads необходимо использовать функцию pthread_join для синхронизации потоков, в то время как в OpenMP это контролируется самим фреймворком. Кроме того, функции, вызываемые при создании pthread, должны передавать аргументы через структуру, когда в случае с OpenMP это необязательно.

Эксперименты показали, что наилучшие результаты проиводительности на больших объемах текста можно получить на 4 потоках, где прирост производительности составляет 54% для OpenMP и 51% для pthreads. Скорее всего, данные результаты можно значительно улучшить, если реализовать более сложный алгоритм распараллеливания (например древовидную структуру распределения текста по потокам).

Отсюда можно сделать вывод, что организация параллельных вычислений программ имеет смысл в трудоемких задачах, хотя для тривиальных задач последовательное решение будет быстрее.

Приложение А Исходный код

```
#include <iomanip>
#include <pthread.h>
#include <omp.h>
#include "utils.h"

using namespace std;

double startTime, endTime; // For timestamps
unordered_map<string, int> wordDict[3];

double defaultProcess(const string &text) {
        startTime = omp_get_wtime();
        pair<int, int> substring = getStringPart(text, 0, text.length() - 1);
        wordDict[0] = getWordDict(text, substring.first, substring.second);
        endTime = omp_get_wtime();

        return endTime - startTime;
}

double pthreadProcess(const string &text, int threadCount) {
        pthread_t *threads = new pthread_t[threadCount];
        getWordDictArgs args[threadCount];
        unordered_map<string, int> dicts[threadCount];

        int textBlock = text.length() / threadCount;
        int startIndex = 0;

        startTime = omp_get_wtime();
        for(int i = 0; i < threadCount; i++) {
                pair<int, int> substring;
                if(i != threadCount - 1){
                        substring = getStringPart(text, startIndex, (i + 1) * textBlock - 1);
                } else {
                        substring = getStringPart(text, startIndex, text.length() - 1);
                }
                startIndex = substring.second + 1;

                args[i].str = text;
                args[i].strBegin = substring.first;
                args[i].strEnd = substring.second;
        }

        for(int i = 0; i < threadCount; i++)
                pthread_create(&threads[i], NULL, getWordDict_pthread, (void *) &args[i]);
```

```
46          for(int i = 0; i < threadCount; i++) pthread_join(threads[i],
    NULL);

48          for(int i = 0; i < threadCount; i++) dicts[i] = args[i].dict;

50          wordDict[1] = mergeDict(dicts, threadCount);
            endTime = omp_get_wtime();
52
            return endTime - startTime;
54  }

56  double ompProcess(const string &text, int threadCount) {
            getWordDictArgs args[threadCount];
58          unordered_map<string, int> dicts[threadCount];

60          int textBlock = text.length() / threadCount;
            int startIndex = 0;
62
            startTime = omp_get_wtime();
64          for(int i = 0; i < threadCount; i++) {
                    pair<int, int> substring;
66                  if(i != threadCount - 1){
                            substring = getStringPart(text, startIndex, (i
    + 1) * textBlock - 1);
68                  } else {
                            substring = getStringPart(text, startIndex,
    text.length() - 1);
70                  }
                    startIndex = substring.second + 1;
72
                    args[i].str = text;
74                  args[i].strBegin = substring.first;
                    args[i].strEnd = substring.second;
76          }

78          omp_set_num_threads(threadCount);
            #pragma omp parallel for
80                  for(int i = 0; i < threadCount; i++) {
                            dicts[i] = getWordDict(args[i].str, args[i].
    strBegin, args[i].strEnd);
82                  }

84          wordDict[2] = mergeDict(dicts, threadCount);
            endTime = omp_get_wtime();
86
            return endTime - startTime;
88  }

90  void showDictsContent() {
            for(int i = 0; i < 3; i++) {
92                  cout << i << " RESULT_____________________________________"
    << endl;
                    for(auto word: wordDict[i]) {
```

```
94                      cout << left << setw(15) << word.first << "\t" <<
   word.second << endl;
                }
96              cout << endl;
        }
98 }

100 long getWordCount(unordered_map<string, int> dict) {
        long counter = 0;
102     for(auto it: dict) counter += it.second;
        return counter;
104 }

106 int main(int argc, char* argv[]) {
        setlocale(LC_CTYPE, "Russian");
108
        string fileName;
110     string text;

112     if(argc > 1) fileName = argv[1];
        else {
114             cout << "Usage: " << argv[0] << " <filename>" << endl;
                return 0;
116     }

118     text = readFromFile(fileName);
        text = trimAndLowerString(text);
120     if(text == "") {
                cout << "File read failed or text is empty!" << endl;
122             return 1;
        }
124
        cout << endl << "Increase words with same thread count" << endl
   ;
126     cout << "[Default]  [Pthread]  [  OMP  ]      Words  Threads"
   << endl;
        double time[3];
128     int threadCount = omp_get_num_procs();
        string str;
130     long wordCount;
        for(int i = 0; i < 6; i++){
132             str += (i == 0) ? text + " " : str + " ";
                time[0] = defaultProcess(str);
134             time[1] = pthreadProcess(str, threadCount);
                time[2] = ompProcess(str, threadCount);
136             wordCount = (i == 0) ? getWordCount(wordDict[0]) :
   wordCount * 2;

138             cout << fixed << setw(9) << time[0] << "  " << fixed <<
    setw(9) << time[1] << "  " << fixed << setw(9) << time[2] << "  "
   << fixed << setw(9) << wordCount << fixed << setw(9) << threadCount
   << endl;
        }
140
```

```
        cout << endl << "Increase threads with same words count" << endl;
        cout << "[Default]  [Pthread]  [  OMP  ]      Words  Threads" << endl;
        str = text;
        wordCount /= 32;
        for(int i = 0; i < 9; i++){
                threadCount = (i == 0) ? 1 : threadCount * 2;
                time[0] = defaultProcess(str);
                time[1] = pthreadProcess(str, threadCount);
                time[2] = ompProcess(str, threadCount);

                cout << fixed << setw(9) << time[0] << "  " << fixed << setw(9) << time[1] << "  " << fixed << setw(9) << time[2] << "  " << fixed << setw(9) << wordCount << fixed << setw(9) << threadCount << endl;
        }

        cout << endl << "Same threads count with same words count" << endl;
        cout << "[Default]  [Pthread]  [  OMP  ]      Words  Threads" << endl;
        for(int i = 0; i < 4; i++) str += str + " ";
        threadCount = 4;
        for(int i = 0; i < 10; i++){
                time[0] = defaultProcess(str);
                time[1] = pthreadProcess(str, threadCount);
                time[2] = ompProcess(str, threadCount);
                wordCount = (i == 0) ? getWordCount(wordDict[0]) : wordCount;

                cout << fixed << setw(9) << time[0] << "  " << fixed << setw(9) << time[1] << "  " << fixed << setw(9) << time[2] << "  " << fixed << setw(9) << wordCount << fixed << setw(9) << threadCount << endl;
        }

        // showDictsContent();
}
```

Листинг А.1. main.cpp

```
#pragma once

#include <iostream>
#include <fstream>
#include <string>
#include <unordered_map>
#include <algorithm>

struct getWordDictArgs {
    std::string str;
    unsigned int strBegin;
    unsigned int strEnd;
```

```
    std::unordered_map<std::string, int> dict;
};

enum cutType {SPACE_0, SPACE_N, WORD_0, WORD_N};

std::string readFromFile(std::string fileName);
std::string trimAndLowerString(std::string str);
std::pair<int, int> getStringPart(const std::string &str, unsigned int
   beginInd, unsigned int endInd);
unsigned int getCutIndex(const std::string &str, int pos, cutType type)
   ;
std::unordered_map<std::string, int> mergeDict(const std::unordered_map
   <std::string, int> dicts[], int size);
std::unordered_map<std::string, int> getWordDict(const std::string &str
   , unsigned int strBegin, unsigned int endInd);
void *getWordDict_pthread(void *args);
```

Листинг А.2. utils.h

```
#include "utils.h"

std::string readFromFile(std::string fileName) {
        std::string str;
        std::ifstream fin(fileName.c_str());

        if(fin.is_open()) {
                str.assign(std::istreambuf_iterator<char>(fin), std::
   istreambuf_iterator<char>());
                fin.close();

                return str;
        } else return "";
}

std::string trimAndLowerString(std::string str) {
        int strBegin = getCutIndex(str, 0, SPACE_0);
        int strLength = getCutIndex(str, str.length() - 1, SPACE_N) -
   strBegin + 1;
        str = str.substr(strBegin, strLength);
        transform(str.begin(), str.end(), str.begin(), ::tolower);
        return str;
}

// beginInd must be at word start or delimiter
std::pair<int, int> getStringPart(const std::string &str, unsigned int
   beginInd, unsigned int endInd) {
        std::pair<int, int> ind;

        // cut delimiter if needed from start
        ind.first = (getCutIndex(str, beginInd, WORD_0) == beginInd) ?
   getCutIndex(str, beginInd, SPACE_0) : beginInd;
        // cut delimiter if needed or restore cut word from end
```

```
30          ind.second = (getCutIndex(str, endInd, WORD_N) == endInd) ?
    getCutIndex(str, endInd, SPACE_N) : getCutIndex(str, endInd, WORD_0)
     - 1;

32          return ind;
}
34
unsigned int getCutIndex(const std::string &str, int pos, cutType type)
     {
36          std::string delim = " .,!?&@():;-|\"\r\n";
            switch(type) {
38                  case(SPACE_0):
                            return str.find_first_not_of(delim, pos);
40                  case(SPACE_N):
                            return str.find_last_not_of(delim, pos);
42                  case(WORD_0):
                            return str.find_first_of(delim, pos);
44                  case(WORD_N):
                            return str.find_last_of(delim, pos);
46                  default:
                            return -1;
48          }
}
50
std::unordered_map<std::string, int> mergeDict(const std::unordered_map
    <std::string, int> dicts[], int size){
52          std::unordered_map<std::string, int> mergedDict;

54          for(int i = 0; i < size; i++) {
                    for(auto it: dicts[i]) mergedDict[it.first] += it.
    second;
56          }

58          return mergedDict;
}
60
std::unordered_map<std::string, int> getWordDict(const std::string &str
    , unsigned int strBegin, unsigned int strEnd) {
62          std::unordered_map<std::string, int> dict;

64          while((strBegin = getCutIndex(str, strBegin, SPACE_0)) < strEnd
    ){
                    int cutEnd = getCutIndex(str, strBegin, WORD_0);
66                  int strLength = cutEnd - strBegin;
                    std::string temp = str.substr(strBegin, strLength);
68
                    strBegin = getCutIndex(str, cutEnd, SPACE_0);
70                  dict[temp] += 1;
            }
72
            return dict;
74  }

76  void *getWordDict_pthread(void *args) {
```

```
        getWordDictArgs *arg = (getWordDictArgs *)args;
78      std::unordered_map<std::string, int> dict;

80      if(!arg->str.empty()) {
                while((arg->strBegin = getCutIndex(arg->str, arg->
    strBegin, SPACE_0)) < arg->strEnd){
82                      int cutEnd = getCutIndex(arg->str, arg->
    strBegin, WORD_0);
                        int strLength = cutEnd - arg->strBegin;
84                      std::string temp = arg->str.substr(arg->
    strBegin, strLength);

86                      arg->strBegin = getCutIndex(arg->str, cutEnd,
    SPACE_0);
                        dict[temp] += 1;
88              }
        }
90
        arg->dict = dict;
92      return 0;
}
```

Листинг А.3. utils.cpp