Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

Отчет по лабораторной работе №1

по дисциплине: «Параллельные вычисления»

Тема работы: «Определение частоты встречи слов в тексте на русском языке»

Работу выполнила

студентка группы 13541/3

_____ Мартюшева Н. Ю.

Преподаватель

_____ Стручков И.В.

Санкт-Петербург
2018

Оглавление

1 Постановка задачи

В рамках лабораторной работы необходимо реализовать программу, определяющую частоту встречи слов в тексте на русском языке.

Программа должна быть разработана в следующих вариантах:

1. Параллельная реализация при помощи POSIX threads;
2. Параллельная реализация при помощи технологии MPI.

Необходимо произвести оценку скорости работы различных вариантов программы в зависимости от окружения, а так же от количества потоков (процессов).

2 Реализация

Опишем входные параметры, алгоритм работы и формат результата каждого варианта программы.

2.1 Параллельная реализация при помощи POSIX threads

В качестве параметров программе передается количество потоков и имя файла с текстом на русском языке. Результатом работы программы является сообщение в следующем формате: в первой строке вывода записано *время выполнения программы в миллисекундах*, затем построчно выводится информация о количестве повторений слов в формате *"Слово Количество_повторений"*.

Программа работает по следующему алгоритму:

1. Инициализируем глобальные переменные: вектор для хранения слов, map для хранения повторений слов, мьютекс для обеспечения совместного доступа к глобальным переменным.
2. Получаем из параметра количество потоков (N).
3. Получаем из параметра имя файла.
4. Открываем файловый поток и получаем информацию о размере анализируемого файла (size), размере части файла для анализа в одном потоке (size/N).
5. Считываем входной файл в массив типа char.
6. Инициализируем таймер и получаем время начала работы программы.
7. Разбиваем входной массив на N массивов примерно одинаковой длинны (size/N). Для этого из входной строки берем подстроку длиной size/N, а затем смещаем границу до первого разделяющего символа (пробела, точки, запятой и т.д.). Таким образов формируется рабочий массив для каждого потока.
8. Запускаем функцию подсчета количества слов для каждого из N потоков, передав ему соответствующую подстроку:
   (a) Инициализируем локальные вектор и карту для подсчета слов.

(b) Последовательно анализируем каждое слово в подстроке, до тех пор, пока не закончится строка:

- Если слова нет в векторе встреченных слов, то добавляем его и в вектор *Новое_слово*, и в map в виде пары *Новое_слово 1*.
- Если слово есть в векторе встреченных слов, то увеличиваем соответствующее значение в map на 1.

(c) Переводим мьютекс в заблокированное состояние.

(d) Объединяем локальные вектор и карту с глобальными вектором и картой.

(e) Разблокируем мьютекс.

9. Ожидаем завершения всех потоков.
10. Считываем время завершения работы и находим время выполнения.
11. Выводим результат работы программы.

Исходный код программы представлен в листинге 1.

2.2 Выполнение при помощи mpi

В качестве параметров программе передается количество процессов и имя файла с текстом на русском языке. Результатом работы программы является сообщение в следующем формате: в первой строке вывода записано *время выполнения программы в миллисекундах*, затем построчно выводится информация о количестве повторений слов в формате *"Слово Количество_повторений"*.

Программа работает по следующему алгоритму:

1. Инициализируем глобальные переменные: вектор для хранения слов, map для хранения повторений слов.
2. Получаем из параметра имя файла.
3. Открываем файловый поток и получаем информацию о размере анализируемого файла (size).
4. Считываем входной файл в массив типа char.
5. Инициализируем таймер и получаем время начала работы программы.
6. При помощи функции *MPI_Init* разбиваем процесс на несколько процессов. При этом процесс с ID 0 будет "мастером а остальные процессы будут "служебными". Для каждого типа процессов будет свой алгоритм выполнения.
7. Процесс-мастер:

   (a) Разбиваем входную строку на N частей по такому же принципу, как в многопоточной программе.

   (b) Передаем каждому из служебных процессов последовательно два сообщения:

   - Размер передаваемой строки.

- Подстроку, которая будет анализироваться в этом служебном процессе.

(c) Получаем от каждого служебного процесса следующие сообщения:

- Размер получаемой строки.
- Подстроку с результатом подсчета частоты слов в подстроке в формате *Слово Количество_ повторений*.

(d) Разбираем строку и обновляем глобальные вектор и карту по аналогии с последовательной программой.

(e) Выводим результат работы программы.

8. Служебный процесс:

(a) Получаем от процесса мастера сообщение с длинной строки, которую необходимо принять и инициализируем память по эту строку.

(b) Получаем строку с текстом.

(c) Инициализируем локальные вектор и карту для подсчета слов.

(d) Последовательно анализируем каждое слово в подстроке, до тех пор, пока она не закончится:

- Если слова нет в векторе встреченных слов, то добавляем его и в вектор *Новое_ слово*, и в map в виде пары *Новое_ слово 1*.
- Если слово есть в векторе встреченных слов, то увеличиваем соответствующее значение в map на 1.

(e) Превращаем вектор и карту в строку вида *"Слово Количество_ повторений"*.

(f) Отправляем процессу-мастеру сообщение с длинной полученной строки.

(g) Отправляем процессу-мастеру сообщение с созданной строкой.

Исходный код параллельной программы приведены в листинге 2.

3 Тестирование производительности программ

3.1 Программа тестирования

Для автоматизации тестирования производительности было решено написать следующий скрипт (листинг 3).

Скрипт работает по следующему алгоритму:

1. Очищаем прошлые результаты.

2. Для каждого из тестовых файлов выполняем:

(a) Запускаем параллельную программу с 4 потоками и записываем результат выполнения в файл *Тестовая_ директория/Имя_ файла.result.thread*.

(b) Запускаем параллельную средствами mpi программу с 4 процессами и записываем результат выполнения в файл *Тестовая_ директория/Имя_ файла.result.mpi*.

3. Запускаем параллельную программу для 1, 2, 4 потоков 50 раз для сбора статистики времени работы и записываем результаты работы в файл *Тестовая_ директория/ result . threads. Количество_ потоков .repeate.*

4. Запускаем параллельную средствами mpi программу для 2, 3, 4 процессов 50 раз для сбора статистики времени работы и записываем результаты работы в файл *Тестовая_ директория/ result . mpi. Количество_ процессов .repeate.*

3.2 Результаты тестирования

Тестирование производилось на компьютере со следующими характеристиками:

- Процессор Intel Core i5 7200U
  - 2 физических ядра;
  - 4 логических ядра;
  - 4 потока;
  - Базовая тактовая частота 2.5 ГГц.
- Оперативная память DDR4, размером 8 Гб;
- Установленная ОС - Ubuntu 16.04 (версия ядра 4.2).

В качестве тестируемого файла был выбран файл достаточно большого размера (около 8Мб) для того, что бы основное время выполнения приходилось на подсчет количества слов. Результаты тестирования приведены в таблице 1.

Таблица 1: Результаты работы программ

Тип программы	**Мат. ожидание**	**СКО**
pthreads 1 поток	1525,9	24,88
pthreads 2 потока	861,1	84,82
pthreads 4 потока	592,18	28,92
pthreads 8 потоков	575,22	26,02
pthreads 16 потоков	579,58	31,61
mpi 1 служебный процесс	1604,88	133,75
mpi 2 служебных процесса	1014,36	76,07
mpi 3 служебных процесса	854,84	50,62
mpi 4 служебных процесса	920,04	92,66
mpi 8 служебных процессов	817,22	49,57
mpi 16 служебных процессов	890,66	111,0

Как видно из результатов программы, при задании минимального количества потоков (1) для pthread программы и минимального количества процессов для mpi программы (2), время выполнения отличается не сильно. Графическое отображение зависимости времени выполнения представлены соответственно на рисунке 1 и рисунке 2.

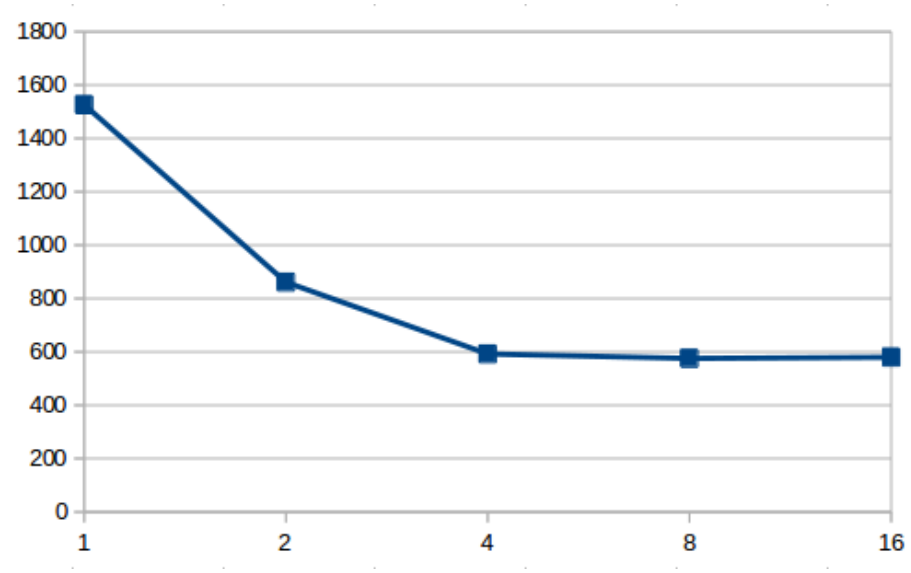

Рис. 1: Зависимость времени выполнения от количества потоков

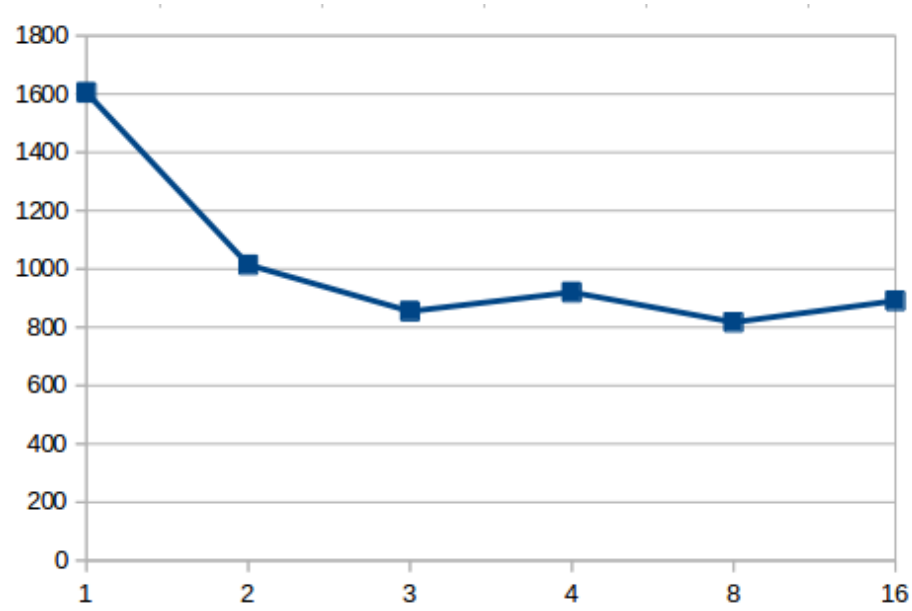

Рис. 2: Зависимость времени выполнения от количества процессов

Как видно из результатов работы, при увеличении числа процессов (потоков) до количества ядер позволяет уменьшить время выполнения программы (до 2 раз при достижении количества физических ядер и до 3 раз - логических), после чего две программы ведут себя несколько по разному. Программа, в которой используются потоки, при задании количества потоков больше, чем количество ядер процессора практически не меняет свои временные показатели (рисунок 1). Программа, в которой использовалась технология MPI (рисунок 2), напротив, при задании количества процессов, большее, чем количество ядер, значительно ухудшила свои показатели. Это, скорее всего, связано с тем, что передача сообщений между процессами занимает значительное время.

При количестве процессов больше, чем количество ядер происходит небольшой скачек,

что, вероятно, вызвано процедурой планирование (увеличивается СКО). При дальнейшем увеличении числа процессов СКО уменьшается, а так же уменьшается время выполнения программы.

4 Вывод

Целью данной лабораторной работы было показать увеличение производительности при разбиение программы на несколько параллельно работающих частей.

В результате работы было опробованы две технологии распараллеливания программы: pthreads и MPI. В результате, при оптимальных параметрах каждая из технологий дала прирост в производительности примерно в 3 раз (при 4 рабочих потоках (процессах)), что является хорошим результатом.

Для синхронизации параллельной работы в Pthreads были использованы мьютексы, а в MPI работа была организована при помощи создания главного процесса, который организует работу подчиненных процессов.

При увеличении количества потоков (процессов) скорость выполнения программы растет нелинейно из-за синхронизации потоков, а так же, экспериментально было выявлено, что наилучшие показатели, как для многопоточной, так и для MPI наилучшие показатели достигаются тогда, когда количество потоков (процессов) равно количеству ядер в компьютере. Так же стоит отметить, что при большом количестве потоков, многопоточная программа практически не получает прироста производительности. Программа, использующая технологию MPI, напротив, значительно замедляется при большом количестве процессов, что вызывается задержками при передачи сообщений.

Для правильной оценки времени работы программы были произведены многократные запуски (около 50 раз для каждого случая), что бы исключить погрешности, например, вызванные процедурой планирования.

5 Листинги

Листинг 1: Параллельная программа при помощи pthreads

```
#include <map>
#include <vector>
#include <cstring>
#include <stdio.h>
#include <string>
#include <unistd.h>
#include <sys/time.h>
#include <pthread.h>
#include <stdlib.h>

using namespace std;

// Global variables
// Map with words: <word, number>
map<string, int> *wordsFreqMap;

// Vector with words: keys from map
vector<string> *wordsVector;
```

```
// Counter of created threads
int createThreads;

// Counter of finish threads
int finishThreads;

//Mutex
pthread_mutex_t lock;

// Frame size
long frameSize;

// Size of file
long fileSize;

// Number of threads
int threadNumber;


// Functions for counting frequency of words in text
// Add word frequency in one frame into global map and vector
void addFreqForFrame(map<string, int> *newFreqMap,
        vector<string> *newKeysVector);

// Counting words frequency in one text frame
void *countFreqForFrame(void *arg);

// Generation text frames for each threads
void generateWordsFreq(const char *inputString);

void printResult();


int main(int argc, char *argv[]) {
        if (argc < 3) {
                printf("Not enough arguments!\n");
                return 1;
        }

        wordsFreqMap = new map<string, int>;
        wordsVector = new vector<string>;
        createThreads = 0;
        finishThreads = 0;

        threadNumber = atoi(argv[1]);

        FILE *file = fopen(argv[2], "r");

        if (file == NULL) {
                perror("File error");
                return 2;
        }

        fseek(file, 0, SEEK_END);
        fileSize = (size_t)ftell(file);
```

```
        frameSize = fileSize / threadNumber + 1;
        rewind(file);

        char *buffer = (char *)malloc((size_t)fileSize);
        fread(buffer, 1, fileSize, file);

        // init mutex
        if (pthread_mutex_init(&lock, NULL) != 0) {
                printf("\n mutex init failed\n");
                return 1;
        }

        struct timeval tvStart;
        struct timeval tvFinish;

        // Start time
        gettimeofday(&tvStart, NULL);

        // Count frequency of words
        generateWordsFreq(buffer);

        // Finish time
        gettimeofday(&tvFinish, NULL);

        // Operating time (in milisecond)
        long int msStart = tvStart.tv_sec * 1000 + tvStart.tv_usec / 1000;
        long int msFinish = tvFinish.tv_sec * 1000 + tvFinish.tv_usec / 1000;

        printf("%ld\n", msFinish - msStart);
        printResult();

        // Delete mutex and other variables
        pthread_mutex_destroy(&lock);
        fclose(file);
        delete (buffer);
        delete (wordsVector);
        delete (wordsFreqMap);
        return 0;
}

void addFreqForFrame(map<string, int> *newFreqMap, vector<string> *
   newKeysVector) {
        if (wordsFreqMap == NULL) {
                wordsFreqMap = new map<string, int>;
                wordsVector = new vector<string>;

                for (vector<string>::iterator it = newKeysVector->begin(); it
                   != newKeysVector->end(); ++it) {
                        int freqNew = newFreqMap->at(*it);
                        wordsFreqMap->insert(pair<string, int>(*it, frqNew));
                        wordsVector->push_back(*it);
                }
                return;
        }

        if (newFreqMap == NULL)
```

```
                return;

        if (newKeysVector == NULL)
                return;

        for (vector<string>::iterator it = newKeysVector->begin(); it !=
           newKeysVector->end(); ++it) {
                if (wordsFreqMap->count(*it)) {
                        int freq = wordsFreqMap->at(*it);
                        int freqNew = newFreqMap->at(*it);
                        map<string, int>::iterator itMap;
                        itMap = wordsFreqMap->find(*it);
                        wordsFreqMap->erase(itMap);
                        wordsFreqMap->insert(pair<string, int>(*it, freq +
                           freqNew));
                }
                else {
                        int freqNew = newFreqMap->at(*it);
                        wordsFreqMap->insert(pair<string, int>(*it, freqNew));
                        wordsVector->push_back(*it);
                }
        }
}

void *countFreqForFrame(void *arg) {
        char *frame = (char *)arg;

        map<string, int> *wMap = new map<string, int>;
        vector<string> *wVector = new vector<string>;

        char *ptr;

        char *word = strtok_r(frame, " ,.:; \"!?()\n", &ptr);

        int i = 0;
        while (word != NULL) {
                i++;
                if (wMap->count(word)) {
                        int bufCount = wMap->at(word);
                        map<string, int>::iterator itMap = wMap->find(word);
                        wMap->erase(itMap);
                        wMap->insert(pair<string, int>(word, ++bufCount));
                }
                else {
                        wMap->insert(pair<string, int>(word, 1));
                        wVector->push_back(word);
                }
                word = strtok_r(NULL, " ,.:; \"!?()\n", &ptr);
        }

        // on mutex
        pthread_mutex_lock(&lock);
        addFreqForFrame(wMap, wVector);
        finishThreads++;
        // off mutex
        pthread_mutex_unlock(&lock);
```

```
        delete (wMap);
        delete (wVector);
        delete (frame);
        delete (word);
}

void generateWordsFreq(const char *inputString) {
        if (inputString == NULL)
                return;

        long from = 0;
        long to = 0;

        int i = 0;
        while (to < (fileSize - 1)) {
                if (to == (fileSize - 1))
                        break;

                to += frameSize;
                while (inputString[to] != ' ' && to < fileSize)
                        to++;

                if (to > fileSize)
                        to = fileSize - 1;

                char *frame = new char[to - from + 1];

                for (int i = 0; i < to - from + 1; i++) {
                        frame[i] = 0;
                }

                strncpy(frame, inputString + from, to - from);
                from = to + 1;

                pthread_t thread;
                // create threads
                createThreads++;
                // counting words frequency for frame
                pthread_create(&thread, NULL, countFreqForFrame, (void *)frame
                    );
                // detach threads
                pthread_detach(thread);
        }

        while (finishThreads < createThreads)
                usleep(10);
}

void printResult() {
        for (vector<string>::iterator it = wordsVector->begin(); it !=
            wordsVector->end(); ++it) {
                string bufName = *it;
                int bufCount = wordsFreqMap->at(*it);
                printf("%s %d\n", bufName.c_str(), bufCount);
        }
```

```
}
```

Листинг 2: Параллельная программа при помощи MPI

```
#include <map>
#include <vector>
#include <cstring>
#include <stdio.h>
#include <string>
#include <unistd.h>
#include <sys/time.h>
#include <mpi.h>
#include <stdlib.h>

using namespace std;

// Global variables
// Map with words: <word, number>
map<string, int> *wordsFreqMap;

// Vector with words: keys from map
vector<string> *wordsVector;

// Frame size
long frameSize;

// Size of file
long fileSize;

int rank, size;

MPI::Status status;


// Functions for counting frequency of words in text
// Add word frequency in one frame into global map and vector
void addFreqForFrame(map<string, int> *newFreqMap,
        vector<string> *newKeysVector);

// Counting words frequency in one text frame
void countFreqForFrame(char *workCharArr);

// Generation text frames for each threads
void generateWordsFreq(const char *inputString);

void printResult();



int main(int argc, char *argv[]) {
        if (argc < 2) {
                printf("Please write filename in parameter\n");
                return 1;
        }

        wordsFreqMap = new map<string, int>;
        wordsVector = new vector<string>;
```

```
        FILE *file = fopen(argv[1], "r");

        if (file == NULL) {
                perror("File error");
                return 2;
        }

        fseek(file, 0, SEEK_END);
        fileSize = (size_t)ftell(file);
        rewind(file);

        char *buffer = (char *)malloc((size_t)fileSize);
        fread(buffer, 1, fileSize, file);

        struct timeval tvStart;
        struct timeval tvFinish;

        // Get time of start programm
        gettimeofday(&tvStart, NULL);

        MPI_Init(&argc, &argv);                         // starts MPI
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);           // get current process id
        MPI_Comm_size(MPI_COMM_WORLD, &size);           // get number of processes

        frameSize = fileSize / (size - 1);

        // Count frequency of words
        generateWordsFreq(buffer);

        if (rank == 0) {
                // Get time of finish programm
                gettimeofday(&tvFinish, NULL);
                long int msStart = tvStart.tv_sec * 1000 + tvStart.tv_usec /
                   1000;
                long int msFinish = tvFinish.tv_sec * 1000 + tvFinish.tv_usec
                   / 1000;

                printf("%ld\n", msFinish - msStart);

                printResult();
        }

        MPI_Finalize();

        fclose(file);


        delete (buffer);
        delete (wordsVector);
        delete (wordsFreqMap);
        return 0;
}

void addFreqForFrame(map<string, int> *newFreqMap, vector<string> *
   newKeysVector) {
```

```
        if (wordsFreqMap == NULL) {
                wordsFreqMap = new map<string, int>;
                wordsVector = new vector<string>;

                for (vector<string>::iterator it = newKeysVector->begin(); it
                   != newKeysVector->end(); ++it) {
                        int freqNew = newFreqMap->at(*it);
                        wordsFreqMap->insert(pair<string, int>(*it, freqNew));
                        wordsVector->push_back(*it);
                }
                return;
        }

        if (newFreqMap == NULL)
                return;

        if (newKeysVector == NULL)
                return;

        for (vector<string>::iterator it = newKeysVector->begin(); it !=
           newKeysVector->end(); ++it) {
                if (wordsFreqMap->count(*it)) {
                        int freq = wordsFreqMap->at(*it);
                        int freqNew = newFreqMap->at(*it);
                        map<string, int>::iterator itMap;
                        itMap = wordsFreqMap->find(*it);
                        wordsFreqMap->erase(itMap);
                        wordsFreqMap->insert(pair<string, int>(*it, freq +
                           freqNew));
                }
                else {
                        int freqNew = newFreqMap->at(*it);
                        wordsFreqMap->insert(pair<string, int>(*it, freqNew));
                        wordsVector->push_back(*it);
                }
        }
}

void countFreqForFrame(char *workCharArr) {

        map<string, int> *wMap = new map<string, int>;
        vector<string> *wVector = new vector<string>;

        char *word = strtok(workCharArr, " ,:. \"!?;()\n");

        while (word != NULL) {
                if (wMap->count(word)) {
                        int bufCount = wMap->at(word);
                        map<string, int>::iterator itMap = wMap->find(word);
                        wMap->erase(itMap);
                        wMap->insert(pair<string, int>(word, ++bufCount));
                }
                else {
                        wMap->insert(pair<string, int>(word, 1));
                        wVector->push_back(word);
                }
```

```
                word = strtok(NULL, " ,:. \"!?()\n");
        }

        string sendString = "";
        for (vector<string>::iterator it = wVector->begin(); it != wVector->
           end(); ++it) {
                string bufString = *it;
                char bufNumber[20];
                int bufNum = wMap->at(*it);
                sprintf(bufNumber, "%d", bufNum);
                string intString(bufNumber);
                sendString = sendString + ' ' + bufString + ' ' + intString;
        }

        int string_lenght = sendString.size() + 1;
        MPI::COMM_WORLD.Send(&string_lenght, 1, MPI::INT, 0, 0);
        MPI::COMM_WORLD.Send(sendString.c_str(), string_lenght, MPI::CHAR, 0,
           1);

        delete (wMap);
        delete (wVector);
}

void generateWordsFreq(const char *inputString) {
        if (inputString == NULL)
                return;

        long from = 0;
        long to = 0;

        if (rank == 0) {
                int i = 1;
                while (i < size) {

                        to += frameSize;
                        while (inputString[to] != ' ' && to < fileSize)
                                to++;

                        if (to > fileSize)
                                to = fileSize - 1;

                        char *workArray = new char[to - from + 1];

                        for (int i = 0; i < to - from + 1; i++)
                                workArray[i] = 0;

                        strncpy(workArray, inputString + from, to - from);

                        int string_lenght = to - from + 1;
                        MPI::COMM_WORLD.Send(&string_lenght, 1, MPI::INT, i,
                           0);
                        MPI::COMM_WORLD.Send(workArray, string_lenght, MPI::
                           CHAR, i, 1);

                        from = to + 1;
                        i++;
```

```
                        delete (workArray);
                }
        }
        else {
                int frameLenght;
                MPI::COMM_WORLD.Recv(&frameLenght, 1, MPI::INT, 0, 0, status);
                char *i_buffer = new char[frameLenght];
                MPI::COMM_WORLD.Recv(i_buffer, frameLenght, MPI::CHAR, 0, 1,
                   status);
                int count = status.Get_count(MPI::CHAR);
                countFreqForFrame(i_buffer);
                delete(i_buffer);
        }

        if (rank == 0) {
                for (int i = 1; i < size; i++) {
                        map<string, int> *wMap = new map<string, int>;
                        vector<string> *wVector = new vector<string>;

                        // Get text string
                        int frameLenght;
                        MPI::COMM_WORLD.Recv(&frameLenght, 1, MPI::INT, i, 0,
                           status);
                        char *i_buffer = new char[frameLenght];
                        MPI::COMM_WORLD.Recv(i_buffer, frameSize, MPI::CHAR, i
                           , 1, status);

                        char *word = strtok(i_buffer, " ,. \"!?;()\n");
                        while (word != NULL) {
                                string b(word);
                                wVector->push_back(b);
                                word = strtok(NULL, " ,. \"!?()\n");

                                string buf = wVector->back();
                                wMap->insert(pair<string, int>(buf, atoi(word)
                                   ));
                                word = strtok(NULL, " ,. \"!?()\n");
                        }

                        addFreqForFrame(wMap, wVector);

                        delete(i_buffer);
                        delete(wMap);
                        delete(wVector);
                }
        }
}

void printResult() {
        for (vector<string>::iterator it = wordsVector->begin(); it !=
           wordsVector->end(); ++it) {
                string bufName = *it;
                int bufCount = wordsFreqMap->at(*it);
                printf("%s %d\n", bufName.c_str(), bufCount);
        }
```

```
}
```

Листинг 3: bash version

```
#!/bin/bash

RDIR=`pwd`
TEST_DIR="$RDIR/TestFiles"
TEST_FILES="test3.txt"
REPORT_DIR="$RDIR/Reports"

# Количество повторений
let COUNTER_VAR=50

# Запуск задач
for i in $TEST_FILES ; do
    echo "Start thread programm for test file $i"
    $RDIR/progThread 4 $TEST_DIR/$i > $REPORT_DIR/"$i".result.thread
    echo "Start mpi programm for test file $i"
    mpirun -np 4 $RDIR/progMPI $TEST_DIR/$i > $REPORT_DIR/"$i".result
        .mpi
done

# Многократный запуск для последующего рассчета СКО и т.д.

TEST_FILE=test3.txt

# Подготовка
find -name *.repeate | xargs rm -f

# Параллельная программа с 1 потоком
#COUNTER=0

#echo "Start pthreads programm with 1 thread repeating..."
while [ $COUNTER -lt $COUNTER_VAR ] ; do
    $RDIR/progThread 1 $TEST_DIR/$TEST_FILE | head -n 1 >>
        $REPORT_DIR/result.threads.1.repeate
    let COUNTER=COUNTER+1
done
echo "Done"
echo ""

# Параллельная программа с 2 потоками
COUNTER=0

echo "Start pthreads programm with 2 thread repeating..."
while [ $COUNTER -lt $COUNTER_VAR ] ; do
    $RDIR/progThread 2 $TEST_DIR/$TEST_FILE | head -n 1 >>
```

```
        $REPORT_DIR/result.threads.2.repeate
    let COUNTER=COUNTER+1
done
echo "Done"
echo ""

# Параллельная программа с 4 потоками
COUNTER=0

echo "Start pthreads programm with 4 thread repeating..."
while [ $COUNTER -lt $COUNTER_VAR ] ; do
    $RDIR/progThread 4 $TEST_DIR/$TEST_FILE | head -n 1 >>
        $REPORT_DIR/result.threads.4.repeate
    let COUNTER=COUNTER+1
done
echo "Done"
echo ""

# Параллельная программа с 8 потоками
COUNTER=0

echo "Start pthreads programm with 8 thread repeating..."
while [ $COUNTER -lt $COUNTER_VAR ] ; do
    $RDIR/progThread 8 $TEST_DIR/$TEST_FILE | head -n 1 >>
        $REPORT_DIR/result.threads.8.repeate
    let COUNTER=COUNTER+1
done
echo "Done"
echo ""

# Параллельная программа с 16 потоками
COUNTER=0

echo "Start pthreads programm with 16 thread repeating..."
while [ $COUNTER -lt $COUNTER_VAR ] ; do
    $RDIR/progThread 16 $TEST_DIR/$TEST_FILE | head -n 1 >>
        $REPORT_DIR/result.threads.16.repeate
    let COUNTER=COUNTER+1
done
echo "Done"
echo ""

# MPI с 1 рабочим процессом
COUNTER=0

echo "Start MPI programm with 2 process repeating..."
while [ $COUNTER -lt $COUNTER_VAR ] ; do
```

```
    mpirun -np 2 $RDIR/progMPI $TEST_DIR/$TEST_FILE | head -n 1 >>
      $REPORT_DIR/result.mpi.1.repeate
    let COUNTER=COUNTER+1
done
echo "Done"
echo ""

# MPI с 2 рабочими процессами
COUNTER=0

echo "Start MPI programm with 3 process repeating..."
while [ $COUNTER -lt $COUNTER_VAR ] ; do
    mpirun -np 3 $RDIR/progMPI $TEST_DIR/$TEST_FILE | head -n 1 >>
      $REPORT_DIR/result.mpi.2.repeate
    let COUNTER=COUNTER+1
done
echo "Done"
echo ""

# MPI с 3 рабочими процессами
COUNTER=0

echo "Start MPI programm with 4 process repeating..."
while [ $COUNTER -lt $COUNTER_VAR ] ; do
    mpirun -np 4 $RDIR/progMPI $TEST_DIR/$TEST_FILE | head -n 1 >>
      $REPORT_DIR/result.mpi.3.repeate
    let COUNTER=COUNTER+1
done
echo "Done"
echo ""

# MPI с 4 рабочими процессами
COUNTER=0

echo "Start MPI programm with 5 process repeating..."
while [ $COUNTER -lt $COUNTER_VAR ] ; do
    mpirun -np 5 $RDIR/progMPI $TEST_DIR/$TEST_FILE | head -n 1 >>
      $REPORT_DIR/result.mpi.4.repeate
    let COUNTER=COUNTER+1
done
echo "Done"
echo ""

# MPI с 8 рабочими процессами
COUNTER=0

echo "Start MPI programm with 9 process repeating..."
```

```
while [ $COUNTER -lt $COUNTER_VAR ] ; do
    mpirun -np 9 $RDIR/progMPI $TEST_DIR/$TEST_FILE | head -n 1 >>
       $REPORT_DIR/result.mpi.8.repeate
    let COUNTER=COUNTER+1
done
echo "Done"
echo ""

# MPI с 16 рабочими процессами
COUNTER=0

echo "Start MPI programm with 17 process repeating..."
while [ $COUNTER -lt $COUNTER_VAR ] ; do
mpirun -np 17 $RDIR/progMPI $TEST_DIR/$TEST_FILE | head -n 1 >>
   $REPORT_DIR/result.mpi.16.repeate
let COUNTER=COUNTER+1
done
echo "Done"
echo ""
```