

Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

# ОТЧЕТ

о лабораторной работе №1

по дисциплине: «Параллельные вычисления»

Тема работы: «Подсчет количества русских слов в тексте»

**Работу выполнил студент**

53051/3      *Скрипаль Б.А.*

**Преподаватель**

\_\_\_\_\_ *Стручков И.В.*

Санкт-Петербург  
2016

# Оглавление

1	Постановка задачи . . . . .	2
2	Реализация . . . . .	2
2.1	Последовательное выполнение . . . . .	2
2.2	Выполнение при помощи pthreads . . . . .	6
2.3	Выполнение при помощи mpi . . . . .	12
3	Тестирование производительности программ . . . . .	18
3.1	Программа тестирования . . . . .	18
3.2	Результаты тестирования . . . . .	22

# 1 Постановка задачи

Реализовать программу, подсчитывающую количество русских слов в тексте при помощи трех технологий:

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

Сравнить времена выполнения программ, в зависимости от окружения, а так же от количества потоков (процессов) для вариантов 2 и 3.

## 2 Реализация

### 2.1 Последовательное выполнение

#### Описание алгоритма

В качестве параметров программе передается имя файла с текстом. После чего программа выдает результат в следующем формате: в первой строке вывода будет записано время выполнения программы в миллисекундах, после чего выводится количество повторений слов в формате "*Слово Количество\_повторений*".

Программа работает по следующему алгоритму:

1. Анализ входных параметров для получения имени файла.
2. Инициализируем глобальный вектор для хранения слов и тар для хранения повторений слов.
3. Считываем входной файл в массив типа char.
4. Инициализируем таймер и получаем время начала работы программы.
5. Последовательно берем каждое новое слово в строке, до тех пор, пока не закончится строка и:
  - Если слово не находится в векторе встреченных слов, то добавляем его в вектор и добавляем в тар пару типа *Новое\_слово 1*.
  - Если слово находится в векторе, то увеличиваем соответствующее значение в тар на 1.
6. Считываем время завершения работы и находим время выполнения.
7. Выводим время выполнения, а так же содержимое тар и вектора.

## Исходный код

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

/*****

/*
 * Global map with words
 */
std::map<std::string, int> *wordsMap;
/*
 * Global vector with words
 */
std::vector<std::string> *wordsVector;
/*
 * Text frame size
 */
long frameSize;
/*
 * Number of readed words
 */
long lSize;

*****/

/*
 * Add new words into global map and vector
 */
void addNewMapAndVector(std::map<std::string, int> *newCounterMap,
                        std::vector<std::string> *newKeysVector);

/*
 * Count words includes in text string
 */
void countWoldIncludes(char *workCharArr);

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

void printResult();

*****/

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

```

    }

    wordsMap = new std::map<std::string, int>;
    wordsVector = new std::vector<std::string>;

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

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

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

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

    struct timeval tvStart;
    struct timeval tvFinish;

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

    // 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();

    fclose(file);
    delete (buffer);
    delete (wordsVector);
    delete (wordsMap);
    return 0;
}

void addNewMapAndVector(std::map<std::string, int> *newCounterMap,
    std::vector<std::string> *newKeysVector) {
    if (wordsMap == NULL) {
        wordsMap = new std::map<std::string, int>;
        wordsVector = new std::vector<std::string>;

        for (std::vector<std::string>::iterator it = newKeysVector->
            begin(); it != newKeysVector->end(); ++it) {
            int bufCountB = newCounterMap->at(*it);
            wordsMap->insert(std::pair<std::string, int>(*it,
                bufCountB));
            wordsVector->push_back(*it);
        }
    }
}

```

```

        return;
    }

    if (newCounterMap == NULL)
        return;

    if (newKeysVector == NULL)
        return;

    for (std::vector<std::string>::iterator it = newKeysVector->begin();
        it != newKeysVector->end(); ++it) {
        if (wordsMap->count(*it)) {
            int bufCountA = wordsMap->at(*it);
            int bufCountB = newCounterMap->at(*it);
            std::map<std::string, int>::iterator itMap;
            itMap = wordsMap->find(*it);
            wordsMap->erase(itMap);
            wordsMap->insert(std::pair<std::string, int>(*it,
                bufCountA + bufCountB));
        }
        else {
            int bufCountB = newCounterMap->at(*it);
            wordsMap->insert(std::pair<std::string, int>(*it,
                bufCountB));
            wordsVector->push_back(*it);
        }
    }
}

void countWoldIncludes(char *workCharArr) {

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

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

    while (pch != NULL) {
        if (wMap->count(pch)) {
            int bufCount = wMap->at(pch);
            std::map<std::string, int>::iterator itMap = wMap->
                find(pch);
            wMap->erase(itMap);
            wMap->insert(std::pair<std::string, int>(pch, ++
                bufCount));
        }
        else {
            wMap->insert(std::pair<std::string, int>(pch, 1));
            wVector->push_back(pch);
        }
        pch = strtok(NULL, " ,.: \\"!?()\\n\\t");
    }

    addNewMapAndVector(wMap, wVector);

    delete (wMap);
    delete (wVector);
}

```

```

}

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

    long counterFrom = 0;
    long counterTo = 0;

    while (counterTo < (lSize - 1)) {

        counterTo += frameSize;
        while (inputString[counterTo] != ' ' && counterTo < lSize)
            counterTo++;

        if (counterTo > lSize)
            counterTo = lSize - 1;

        char *workArray = new char[counterTo - counterFrom + 1];

        for (int i = 0; i < counterTo - counterFrom + 1; i++)
            workArray[i] = 0;

        strncpy(workArray, inputString + counterFrom, counterTo -
            counterFrom);
        counterFrom = counterTo + 1;
        countWoldIncludes(workArray);
        delete (workArray);
    }
}

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

```

## 2.2 Выполнение при помощи pthreads

### Описание алгоритма

В качестве параметров программе передается количество потоков и имя файла с текстом. После чего программа выдает результат в следующем формате: в первой строке вывода будет записано время выполнения программы в миллисекундах, после чего выводится количество повторений слов в формате *"Слово Количество\_повторений"*.

Программа работает по следующему алгоритму:

1. Анализ входных параметров для получения имени файла и количество потоков.
2. Инициализируем глобальный вектор для хранения слов и map для хранения повторений слов, а так же мьютекс для обеспечения совместного доступа к глобальным переменным.

3. Считываем входной файл в массив типа `char`.
4. Инициализируем таймер и получаем время начала работы программы.
5. Разбиваем входной массив на  $n$  ( $n$  - количество потоков), массивов примерно одинаковой длины. Для этого входную строку делим на  $n$  равных частей, после чего смещаем каждую границу до первого разделяющего символа (пробела, точки, запятой и т.д.).
6. Для каждого из потоков запускаем функцию подсчета количества слов и переводим его в отсоединенный режим:
  - (a) Инициализируем локальные вектор и карту для подсчета слов.
  - (b) Последовательно берем каждое новое слово в строке, до тех пор, пока не закончится строка и:
    - Если слово не находится в векторе встреченных слов, то добавляем его в вектор и добавляем в `map` пару типа *Новое\_слово 1*.
    - Если слово находится в векторе, 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.
7. Ожидаем завершения всех потоков.
8. Считываем время завершения работы и находим время выполнения.
9. Выводим время выполнения, а так же содержимое `map` и вектора.

## Исходный код

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

/*****
 * Global map with words
 */
std::map<std::string, int> *wordsMap;
/*
 * Global vector with words
 */
std::vector<std::string> *wordsVector;
/*
 * Количество созданных потоков
```



```

    */
int createThreadsCounter;
/*
    * Number of finish threads
    */
int finishThreadsCounter;
/*
    * Mutex
    */
pthread_mutex_t lock;
/*
    *      Frame size
    */
long frameSize;
/*
    * Number of readed words
    */
long lSize;
/*
    * Number of threads
    */
int threadNumber;

/*****/
/*
    * Add new words into global map and vector
    */
void addNewMapAndVector(std::map<std::string, int> *newCounterMap,
                        std::vector<std::string> *newKeysVector);

/*
    * Count words includes in text string
    */
void *countWoldIncludes(void *arg);

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

void printResult();

/*****/
int main(int argc, char *argv[]) {
    if (argc < 3) {
        printf("Please write filename in parameter\n");
        return 1;
    }

    wordsMap = new std::map<std::string, int>;
    wordsVector = new std::vector<std::string>;
    createThreadsCounter = 0;
    finishThreadsCounter = 0;

    threadNumber = atoi(argv[1]);

```

```

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

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

fseek(file, 0, SEEK_END);
lSize = (size_t)ftell(file);
frameSize = lSize / threadNumber + 1;
rewind(file);

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

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

struct timeval tvStart;
struct timeval tvFinish;

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

// 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();

// Delete mutex
pthread_mutex_destroy(&lock);
fclose(file);
delete (buffer);
delete (wordsVector);
delete (wordsMap);
return 0;
}

void addNewMapAndVector(std::map<std::string, int> *newCounterMap,
    std::vector<std::string> *newKeysVector) {
    if (wordsMap == NULL) {
        wordsMap = new std::map<std::string, int>;
        wordsVector = new std::vector<std::string>;

        for (std::vector<std::string>::iterator it = newKeysVector->
            begin(); it != newKeysVector->end(); ++it) {
            int bufCountB = newCounterMap->at(*it);
            wordsMap->insert(std::pair<std::string, int>(*it,
                bufCountB));
        }
    }
}

```

```

        wordsVector->push_back(*it);
    }
    return;
}

if (newCounterMap == NULL)
    return;

if (newKeysVector == NULL)
    return;

for (std::vector<std::string>::iterator it = newKeysVector->begin();
     it != newKeysVector->end(); ++it) {
    if (wordsMap->count(*it)) {
        int bufCountA = wordsMap->at(*it);
        int bufCountB = newCounterMap->at(*it);
        std::map<std::string, int>::iterator itMap;
        itMap = wordsMap->find(*it);
        wordsMap->erase(itMap);
        wordsMap->insert(std::pair<std::string, int>(*it,
            bufCountA + bufCountB));
    }
    else {
        int bufCountB = newCounterMap->at(*it);
        wordsMap->insert(std::pair<std::string, int>(*it,
            bufCountB));
        wordsVector->push_back(*it);
    }
}

}

void *countWoldIncludes(void *arg) {
    char *workCharArr = (char *)arg;

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

    char *saveptr;

    char *pch = strtok_r(workCharArr, " ,.: \"!?()\\n", &saveptr);

    int i = 0;
    while (pch != NULL) {
        i++;
        if (wMap->count(pch)) {
            int bufCount = wMap->at(pch);
            std::map<std::string, int>::iterator itMap = wMap->
                find(pch);
            wMap->erase(itMap);
            wMap->insert(std::pair<std::string, int>(pch, ++
                bufCount));
        }
        else {
            wMap->insert(std::pair<std::string, int>(pch, 1));
            wVector->push_back(pch);
        }
    }
}

```

```

        pch = strtok_r(NULL, " ,.: \"'!?()\\n", &saveptr);
    }

    // on mutex
    pthread_mutex_lock(&lock);
    addNewMapAndVector(wMap, wVector);
    finishThreadsCounter++;
    // off mutex
    pthread_mutex_unlock(&lock);

    delete (wMap);
    delete (wVector);
    delete (workCharArray);
    delete (pch);
}

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

    long counterFrom = 0;
    long counterTo = 0;

    int i = 0;
    while (counterTo < (lSize - 1)) {
        if (counterTo == (lSize - 1))
            break;

        counterTo += frameSize;
        while (inputString[counterTo] != ' ' && counterTo < lSize)
            counterTo++;

        if (counterTo > lSize)
            counterTo = lSize - 1;

        char *workArray = new char[counterTo - counterFrom + 1];

        for (int i = 0; i < counterTo - counterFrom + 1; i++) {
            workArray[i] = 0;
        }

        strncpy(workArray, inputString + counterFrom, counterTo -
            counterFrom);
        counterFrom = counterTo + 1;

        pthread_t thread;
        // create threads
        createThreadsCounter++;
        pthread_create(&thread, NULL, countWoldIncludes, (void *)
            workArray);
        // detach threads
        pthread_detach(thread);
    }

    while (finishThreadsCounter < createThreadsCounter)
        usleep(10);
}

```

```

}

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

```

## 2.3 Выполнение при помощи mpi

### Описание алгоритма

В качестве параметров программе передается количество процессов и имя файла с текстом. После чего программа выдает результат в следующем формате: в первой строке вывода будет записано время выполнения программы в миллисекундах, после чего выводится количество повторений слов в формате *"Слово Количество\_повторений"*.

Программа работает по следующему алгоритму:

1. Анализ входных параметров для получения имени файла и количества процессов.
2. Инициализируем глобальный вектор для хранения слов и map для хранения повторений слов.
3. Считываем входной файл в массив типа char.
4. Инициализируем таймер и получаем время начала работы программы.
5. При помощи функции *MPI\_Init* разбиваем процесс на несколько процессов. При этом процесс с ID 0 будет "мастером а остальные процессы будут "служебными". После чего для каждого из типов процессов будет свой алгоритм выполнения.
6. Процесс-мастер:
  - (a) Разбиваем входную строку на n (n - количество служебных процессов) по такому же принципу, как в многопоточной программе.
  - (b) Передаем каждому из служебных процессов последовательно два сообщения:
    - Размер строки, которую собираемся передать.
    - Подстроку, которую будет обрабатывать этот служебный процесс.
  - (c) Получаем от каждого из служебного процесса последовательно следующие сообщения:
    - Размер строки, которую собираемся передать.
    - Подстроку с результатом подсчета слов формата *Слово Количество\_повторений*.
  - (d) Разбираем строку и обновляем глобальные вектор и карту по аналогии с последовательной программой.
  - (e) Выводим время завершения обработки.

(f) Выводим результаты.

#### 7. 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 берем каждое новое слово в строке, до тех пор, пока не закончится строка и:
  - Если слово не находится в векторе встреченных слов, то добавляем его в вектор и добавляем в map пару типа *Новое\_слово 1*.
  - Если слово находится в векторе, 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

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

/*****
 * Global map with words
 */
std::map<std::string, int> *wordsMap;
/*
 * Global vector with words
 */
std::vector<std::string> *wordsVector;
/*
 * Text frame size
 */
long frameSize;
/*
 * Number of readed words
 */
long lSize;

int rank, size;
```

```

MPI::Status status;

/*****
/*
 * Add new words into global map and vector
 */
void addNewMapAndVector(std::map<std::string, int> *newCounterMap,
                        std::vector<std::string> *newKeysVector);

/*
 * Count words includes in text string
 */
void countWoldIncludes(char *workCharArr);

/*
 * Generation text frames
 */
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;
    }

    wordsMap = new std::map<std::string, int>;
    wordsVector = new std::vector<std::string>;

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

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

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

    char *buffer = (char *)malloc((size_t)lSize);
    fread(buffer, 1, lSize, 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 = lSize / (size - 1);

    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 (wordsMap);
    return 0;
}

void addNewMapAndVector(std::map<std::string, int> *newCounterMap,
    std::vector<std::string> *newKeysVector) {
    if (wordsMap == NULL) {
        wordsMap = new std::map<std::string, int>;
        wordsVector = new std::vector<std::string>;

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

    if (newCounterMap == NULL)
        return;

    if (newKeysVector == NULL)
        return;

    for (std::vector<std::string>::iterator it = newKeysVector->begin();
        it != newKeysVector->end(); ++it) {
        if (wordsMap->count(*it)) {
            int bufCountA = wordsMap->at(*it);
            int bufCountB = newCounterMap->at(*it);
            std::map<std::string, int>::iterator itMap;

```



```

        itMap = wordsMap->find(*it);
        wordsMap->erase(itMap);
        wordsMap->insert(std::pair<std::string, int>(*it,
            bufCountA + bufCountB));
    }
    else {
        int bufCountB = newCounterMap->at(*it);
        wordsMap->insert(std::pair<std::string, int>(*it,
            bufCountB));
        wordsVector->push_back(*it);
    }
}

void countWoldIncludes(char *workCharArr) {

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

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

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

    std::string sendString = "";
    for (std::vector<std::string>::iterator it = wVector->begin(); it !=
        wVector->end(); ++it) {
        std::string bufString = *it;
        char bufNumber[20];
        int bufNum = wMap->at(*it);
        sprintf(bufNumber, "%d", bufNum);
        std::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 counterFrom = 0;
    long counterTo = 0;

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

            counterTo += frameSize;
            while (inputString[counterTo] != ' ' && counterTo <
                lSize)
                counterTo++;

            if (counterTo > lSize)
                counterTo = lSize - 1;

            char *workArray = new char[counterTo - counterFrom +
                1];

            for (int i = 0; i < counterTo - counterFrom + 1; i++)
                workArray[i] = 0;

            strncpy(workArray, inputString + counterFrom,
                counterTo - counterFrom);

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

            counterFrom = counterTo + 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);
        countWoldIncludes(i_buffer);
        delete(i_buffer);
    }

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

```

```

        std::vector<std::string> *wVector = new std::vector<
            std::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 *pch = std::strtok(i_buffer, " ,. \"!?()\n");
        while (pch != NULL) {
            std::string b(pch);
            wVector->push_back(b);
            pch = strtok(NULL, " ,. \"!?()\n");

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

        addNewMapAndVector(wMap, wVector);

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

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

```

## 3 Тестирование производительности программ

### 3.1 Программа тестирования

Для автоматизации тестирования производительности было решено написать следующий скрипт:

```
#!/bin/bash
```

```

RDIR='pwd'
TEST_DIR="$RDIR/testFiles"
TEST_FILES="test1.txt"

```

```

REPORT_DIR="$RDIR/testReports"

# Количество повторений
let COUNTER_VAR=50

# Подготовка директорий

echo "Create report dir $RDIR"

if [ -d $REPORT_DIR ] ; then
rm -R $REPORT_DIR
fi
mkdir $REPORT_DIR

# Сборка задач
make clean
make

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

# Сравнение результатов
for i in $TEST_FILES ; do
cat $REPORT_DIR/"$i".result.serial | sort > $REPORT_DIR/"$i".result.serial.tmp
mv $REPORT_DIR/"$i".result.serial.tmp $REPORT_DIR/"$i".result.serial

cat $REPORT_DIR/"$i".result.thread | sort > $REPORT_DIR/"$i".result.thread.tmp
mv $REPORT_DIR/"$i".result.thread.tmp $REPORT_DIR/"$i".result.thread

cat $REPORT_DIR/"$i".result.mpi | sort > $REPORT_DIR/"$i".result.mpi.tmp
mv $REPORT_DIR/"$i".result.mpi.tmp $REPORT_DIR/"$i".result.mpi

diff $REPORT_DIR/"$i".result.serial $REPORT_DIR/"$i".result.thread >
    $REPORT_DIR/"$i".diff.serial.thread
diff $REPORT_DIR/"$i".result.serial $REPORT_DIR/"$i".result.mpi >
    $REPORT_DIR/"$i".diff.serial.mpi
done

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

```

```

TEST_FILE=test1.txt

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

# Последовательная программа
COUNTER=0

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

# Параллельная программа с 1 потоком
COUNTER=0

echo "Start pthreads programm with 1 thread repeating..."
while [ $COUNTER -lt $COUNTER_VAR ] ; do
$RDIR/workThreads 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/workThreads 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 2 thread repeating..."
while [ $COUNTER -lt $COUNTER_VAR ] ; do
$RDIR/workThreads 4 $TEST_DIR/$TEST_FILE | head -n 1 >>

```

```

    $REPORT_DIR/result.threads.4.repeate
let COUNTER=COUNTER+1
done
echo "Done"
echo ""

# MPI с 1 рабочим процессом
COUNTER=0

echo "Start mpi programm with 1 process repeating..."
while [ $COUNTER -lt $COUNTER_VAR ] ; do
mpirun -np 2 $RDIR/workMPI $TEST_DIR/$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 2 process repeating..."
while [ $COUNTER -lt $COUNTER_VAR ] ; do
mpirun -np 3 $RDIR/workMPI $TEST_DIR/$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/workMPI $TEST_DIR/$TEST_FILE | head -n 1 >>
    $REPORT_DIR/result.mpi.4.repeate
let COUNTER=COUNTER+1
done
echo "Done"
echo ""

```

Скрипт работает по следующему алгоритму:

1. Очищаем прошлые результаты.
2. Подготавливаем директории для результатов теста.
3. Выполняем сборку задач.

4. Для каждого из тестовых файлов выполняем:
  - (а) Запускаем последовательную программу и записываем результат выполнения в файл *Тестовая\_директория/Имя\_файла.result.serial*.
  - (б) Запускаем параллельную программу с 4 потоками и записываем результат выполнения в файл *Тестовая\_директория/Имя\_файла.result.thread*.
  - (с) Для проверки корректности работы программы находим разницу между результатами выполнения между последовательной и параллельной и последовательной и mpi программами и выводим их в файлы *Тестовая\_директория/Имя\_файла.diff.serial.thread* и *Тестовая\_директория/Имя\_файла.diff.serial.mpi* соответственно.
5. Запускаем последовательную программу 50 раз для сбора статистики времени работы и записываем результаты работы в файл *Тестовая\_директория/result.serial.repeate*.
6. 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*.
7. Запускаем параллельную средствами mpi программу для 2, 3, 4 потоков 50 раз для сбора статистики времени работы и записываем результаты работы в файл *Тестовая\_директория/result.threads.Количество\_процессов.repeate*.

## 3.2 Результаты тестирования

Тестирование производилось на компьютере с 4-х ядерным процессором Intel, под управлением операционной системы Ubuntu. В качестве тестируемого файла был выбран файл большого размера (около 17Мб) 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	Дов. интервал
Последовательная	2253,34	25,48	7,06
pthread 1 поток	2295,96	20,18	5,59
pthread 2 потока	1195,1	12,41	3,44
pthread 4 потока	651,76	7,14	1,98
pthread 8 потоков	643,8	10,11	2,96
pthread 16 потоков	619,22	13,71	3,8
mpi 1 служебный процесс	2272,9	7,38	2,04
mpi 2 служебных процесса	1190,96	6,71	1,86
mpi 3 служебных процесса	835,58	5,03	1,40
mpi 4 служебных процесса	963,52	82,84	22,96
mpi 8 служебных процессов	1342,82	190,79	52,89
mpi 16 служебных процессов	2496,78	216,54	60,20

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

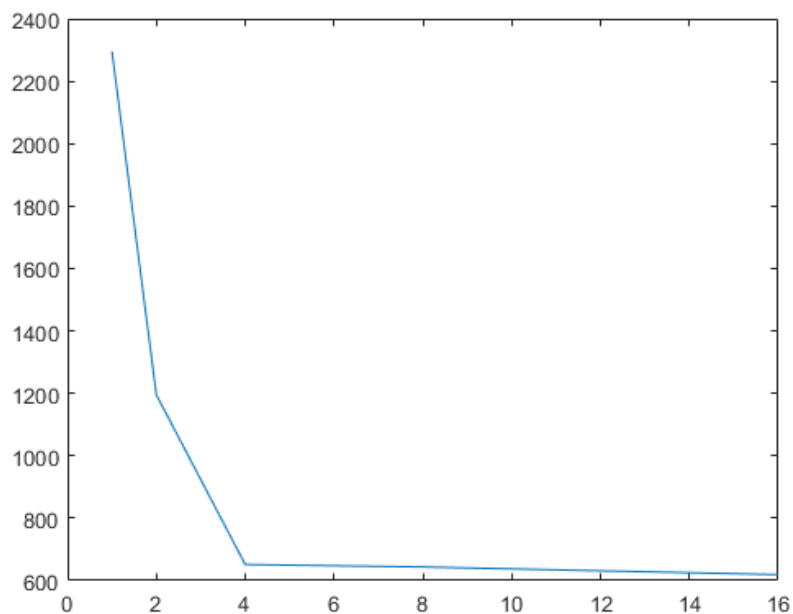


Рис. 1: Зависимость времени выполнения от количества потоков.

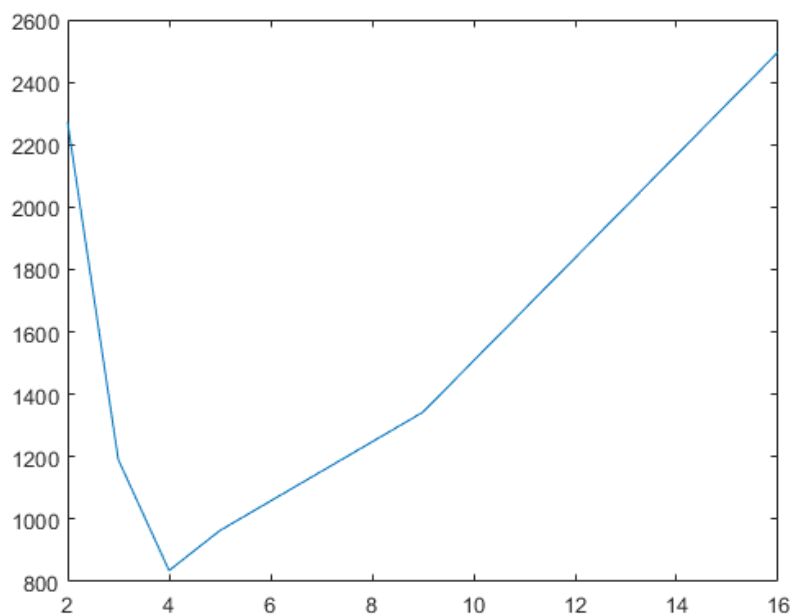


Рис. 2: Зависимость времени выполнения от количества процессов.



Как видно из результатов работы, при увеличении числа процессов (потоков) до количества ядер позволяет значительно уменьшить время выполнения программы (до 3.5 раз), после чего две программы ведут себя несколько по разному. Вторая программа (в которой используются потоки) при задании количества потоков больше, чем количество ядер процессора практически не ухудшает свои временные показатели (рисунок 1). Программа, в которой использовалась технология MPI (рисунок 2), напротив, при задании количества процессов, больше, чем количество ядер значительно ухудшила свои показатели. Это, скорее всего, связано с процедурой планирования в ОС Ubuntu: так как в ней производится планирование на уровне процессов, при создании большого количества процессов, каждый из которых сильно загружает ЦП, необходимо частое переключение контекстов, что значительно замедляет работу.