Санкт-Петербургский государственный политехнический университет

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

ОТЧЕТ

о курсовом проекте

по дисциплине: «Параллельные вычисления»

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

Работу выполнил студент

53501/3 *Алексюк А.О.*

Преподаватель

_____ *Стручков И.В.*

Санкт-Петербург
2016

1. Задание

В рамках курсового проекта студент выбирает один из алгоритмов, для которого выполняется:

1) разработка алгоритма
2) создание последовательной программы, реализующей алгоритм
3) разработка тестового набора, оценка тестового покрытия
4) выделение частей для параллельной реализации, определение общих данных, анализ потенциального выигрыша
5) выбор способов синхронизации и защиты общих данных
6) разработка параллельной программы
7) отладка на имеющихся тестах, анализ и доработка тестового набора с учетом параллелизма
8) измерение производительности, сравнение с производительностью последовательной программы
9) анализ полученных результатов, доработка параллельной программы
10) написание отчета и презентации
11) защита проекта

Мною было выбрано задание №10: Определить частоту встречи слов в тексте на русском языке.

2. Ход работы

2.1. Разработка последовательной программы

Для разбиения текста на слова воспользуемся функцией strtok_r (реентерабельный вариант функции strtok). В качестве разделителя укажем пробел и знаки препинания. Вызывая каждый раз функцию strtok_r, мы будем получать новое слово.

Считанные слова будут помещаться в ассоциативный массив (C++ контейнер map), где ключом будет слово, а значением - число появлений этого слова.

Листинг 1: Объявление типов (файл utils.h)

```
18 typedef map<string, int> wordStat;
```

Листинг 2: Последовательная программа (файл count.h)

```
23 wordStat countWords(const char* text, size_t len) {
24     char* workArray = new char[len + 1];
25     strncpy(workArray, text, len);
26     workArray[len] = '\0';
27 
28     const char* delim = " ,. \"!-?()";
29     char *saveptr;
30 
31     wordStat stat;
32 
33     char* pch = strtok_r(workArray, delim, &saveptr);
34 
35     while (pch != NULL) {
36         int prevCount = stat.count(pch) ? stat[pch] : 0;
37         stat[pch] = prevCount + 1;
38         pch = strtok_r(NULL, delim, &saveptr);
39     }
40     delete[] workArray;
41     return stat;
42 }
```

Такой ассоциативный массив очень легко заполнять, но не очень удобно обрабатывать. Например, в нем довольно сложно найти 5 самых популярных слов. Для этого преобразуем данные в другой формат, тоже ассоциативный массив, но такой, в котором ключом будет число появлений слова, а значением - само слово. В C++ для этого подойдет контейнер multimap (обычный map не подойдет, так как в наборе может быть несколько слов с одинаковой частотой появления, т.е. несколько записей, имеющих одинаковый ключ).

Листинг 3: Преобразование формата хранения (файл utils.h)

```
27 template<typename A, typename B>
28 std::multimap<B,A> flipMap(const std::map<A,B> &src)
29 {
30     std::multimap<B,A> dst;
31     std::transform(src.begin(), src.end(),
32                    std::inserter(dst, dst.begin()), flipPair<A,B>);
33     return dst;
34 }
```

Выведем результаты на экран. Multimap обычно реализован с помощью бинарного дерева поиска, поэтому нет необходимости вручную сортировать результаты. Возьмем 5 самых популярных слов.

Листинг 4: Отображение результатов (файл utils.h)

```
36 void printStats(wordStatFlipped stat) {
37     printf("Stats: \n");
38 
39     const unsigned long showFirst = 5;
40     auto end = stat.size() > showFirst ? std::next(stat.rbegin(),
          showFirst) : stat.rend();
41 
42     for (auto it = stat.rbegin(); it != end; ++it) {
43         printf("%5d %s \n", it->first, it->second.c_str());
```

```
    }
}
```

Функция main:

Листинг 5: Функция main (файл main.cpp)

```
#include <iostream>
#include "utils.h"
#include "count.h"

using namespace std;

int main() {
    pair<const char*, size_t> text = loadTextFromFile("book.txt");

    auto counted = countWords(text.first, text.second);
    auto flipped = flipMap<string, int>(counted);
    printStats(flipped);
    return 0;
}
```

Для сборки воспользуемся утилитой CMake. Ниже приведен сценарий сборки.

Листинг 6: Сценарий сборки (итоговый вариант)

```
cmake_minimum_required(VERSION 2.8)
project(lab1)

find_package(OpenMP)
if (OPENMP_FOUND)
    set (CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${OpenMP_C_FLAGS}")
    set (CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}")
endif()

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11 -O2 -march=native"
   )

add_definitions(-DWITH_OPENMP)

set(SOURCE_FILES utils.h count.h)
add_executable(lab1 main.cpp ${SOURCE_FILES})

#################################
# GTest
#################################
ADD_SUBDIRECTORY(googletest-release-1.7.0)
enable_testing()
include_directories(${gtest_SOURCE_DIR}/include ${gtest_SOURCE_DIR})

#################################
# Unit Tests
#################################
# Add test cpp file
add_executable(runTest test.cpp ${SOURCE_FILES})
# Link test executable against gtest & gtest_main
target_link_libraries(runTest gtest gtest_main)
add_test( runTest runTest )
```

```
32
33 ################################
34 # Benchmark
35 ################################
36 add_executable(benchmark benchmark.cpp ${SOURCE_FILES})
```

В качестве тестовых данных возьмем роман-эпопею Льва Николаевича Толстого «Война и мир». Напишем скрипт, который скачивает её из Интернета, объединяет два тома и преобразует кодировку. Результат её работы - файл book.txt, содержащий чуть менее полумиллиона слов и занимающий 5 мегабайт.

Листинг 7: Скрипт скачивания книги (файл get_book.sh)

```
1 #!/bin/bash
2
3 curl http://vojnaimir.ru/files/book1.txt http://vojnaimir.ru/files/
     book2.txt | iconv -f WINDOWS-1251 -t UTF-8 > book.txt
4 sed 's-//-/g' -i book.txt
```

Результат работы программы:

```
1 Stats:
2 20304 и
3 10198 в
4 8428 не
5 7822 что
6 6453 на
```

2.2. Тестирование

Для запуска тестов воспользуемся системой Google Test. Сценарий сборки ожидает, что исходный код фреймворка будет находиться рядом с исходным кодом в директории googletest-release-1.7.0.

Напишем простейший тест - пусть программа берет данные из константной строки, считает в ней частоту появления слов и сравнивает результаты с заранее известными.

Листинг 8: (файл utils.h)

```
8  TEST(WordCountTest, ConstStringTest)
9  {
10     const char* text = "Lorem ipsum dolor sit amet, consectetur
          adipiscing elit."
11           " Curabitur at sollicitudin risus. Proin id dictum ex.
              Cum sociis"
12           " natoque penatibus et magnis dis parturient montes,
              nascetur"
13           " ridiculus mus. Donec porta felis magna, eu fringilla
              sapien porta"
14           " consectetur. Vestibulum sagittis, dui sit amet sagittis
               posuere,"
15           " augue tellus mollis mauris, id ornare.";
16
17     auto counted = countWords(text, strlen(text));
18     auto flipped = flipMap<string, int>(counted);
```

```
19      int wordsTotal = 0;
20      for (auto it: flipped) {
21          wordsTotal += it.first;
22      }
23      ASSERT_EQ(wordsTotal, 50);
24
25      wordStatFlipped m = {{2, "amet"},
26                           {2, "consectetur"},
27                           {2, "id"}};
28
29      ASSERT_TRUE(std::equal(m.begin(), m.end(), flipped.rbegin()));
30  }
```

Напишем более сложный тест - будем считывать данные из файла.

Листинг 9: (файл utils.h)

```
32  TEST(WordCountTest, FileTest)
33  {
34      pair<const char*, size_t> text = loadTextFromFile("book.txt");
35
36      auto counted = countWords(text.first, text.second);
37      auto flipped = flipMap<string, int>(counted);
38
39      wordStatFlipped m = {{20304, "и"},
40                           {10198, "в"},
41                           {8428, "не"},
42                           {7822, "что"},
43                           {6453, "на"}};
44
45      ASSERT_TRUE(std::equal(m.rbegin(), m.rend(), flipped.rbegin()));
46  }
```

Результат запуска:

```
1  artyom@artyom-H97-D3H:~/Projects/ParallelComputing$ build/gcc/runTest
2  [==========] Running 8 tests from 4 test cases.
3  [----------] Global test environment set-up.
4  [----------] 2 tests from WordCountTest
5  [ RUN      ] WordCountTest.ConstStringTest
6  [       OK ] WordCountTest.ConstStringTest (0 ms)
7  [ RUN      ] WordCountTest.FileTest
8  [       OK ] WordCountTest.FileTest (388 ms)
9  [----------] 2 tests from WordCountTest (388 ms total)
```

2.3. Разработка параллельной реализации

Перед написанием параллельной реализации создадим промежуточный, «псевдопараллельный» вариант программы. Пусть в процессе своей работы программа разбивает исходные данные на блоки, обрабатывает их последовательно, а в конце объединяет результаты.

Блоком в данном случае будет часть текста. Все блоки будут иметь примерно одинаковый размер. К сожалению, если просто взять длину текста и разделить её на число блоков, если вероятность того, что граница между блоками ляжет

посредине слова. Поэтому необходимо передвинуть каждую границу так, чтобы она не разбивала слово пополам (например, чтобы она лежала на пробеле).

Для подсчета частоты появления слов в блоке воспользуемся уже разработанной функцией countWords. Результат её работы - ассоциативный массив (словарь). Для объединения результата блоков достаточно объединить их словари, а если одно и то же слово имеется и в одном, и в другом блоке, нужно сложить количество раз, сколько они появлялись в каждом блоке. Параметр blockCount задает количество блоков.

Листинг 10: (файл count.h)

```
44 wordStat countWordsBlockwise(const char* text, size_t len, int
      blockCount) {
45     wordStat stat;
46     vector<size_t> blockStart;
47 
48     size_t blockSize = len / blockCount + 1;
49     size_t startPos = 0;
50     size_t endPos;
51 
52     // Fix borders
53     for (int i = 0; i < blockCount; i++) {
54         blockStart.push_back(startPos);
55         endPos = startPos + blockSize;
56         while (true) {
57             if ((endPos > len) || (text[endPos] == ' ')) {
58                 break;
59             }
60             endPos++;
61         }
62         startPos = endPos + 1;
63         if (startPos > len) {
64             break;
65         }
66     }
67 
68     for (int i = 0; i < blockCount; i++) {
69         startPos = blockStart[i];
70         endPos = i == (blockCount-1) ? len : blockStart[i + 1];
71         // Run countWords
72         auto localMap = countWords(text + startPos, endPos - startPos
              );
73         // Merge results
74         for (auto& it: localMap) {
75             int prevCount = stat.count(it.first) ? stat[it.first] :
                  0;
76             stat[it.first] = prevCount + it.second;
77         }
78     }
79     return stat;
80 }
```

Напишем тесты для этой реализации. Во-первых, проверим работу функции для файла. Во-вторых, сравним результаты работы новой реализации и старой. Для этого пройдем по словарю и сравним попарно все элементы (как уже говори-

лось выше, словарь заведомо отсортированный).

Листинг 11: (файл utils.h)

```
48 ивнечтона
49 TEST(WordCountBlockwiseTest, FileTest)
50 {
51     pair<const char*, size_t> text = loadTextFromFile("book.txt");
52 
53     auto counted = countWordsBlockwise(text.first, text.second,
       defaultThreadCount);
54     auto flipped = flipMap<string, int>(counted);
55 
56     wordStatFlipped m = {{20304, "и"},
57                          {10198, "в"},
58                          {8428, "не"},
59                          {7822, "что"},
60                          {6453, "на"}};
61 
62     ASSERT_TRUE(std::equal(m.rbegin(), m.rend(), flipped.rbegin()));
63 }
64 
65 TEST(WordCountBlockwiseTest, FileParAndSeqTest)
66 {
67     pair<const char*, size_t> text = loadTextFromFile("book.txt");
68 
69     auto counted1 = countWords(text.first, text.second);
70     auto flipped1 = flipMap<string, int>(counted1);
71 
72     auto counted2 = countWordsBlockwise(text.first, text.second,
       defaultThreadCount);
73     auto flipped2 = flipMap<string, int>(counted2);
74 
75     ASSERT_TRUE(std::equal(flipped1.rbegin(), flipped1.rend(),
       flipped2.rbegin()));
76 }
```

Результат тестов:

```
1 [----------] 2 tests from WordCountBlockwiseTest
2 [ RUN      ] WordCountBlockwiseTest.FileTest
3 [       OK ] WordCountBlockwiseTest.FileTest (415 ms)
4 [ RUN      ] WordCountBlockwiseTest.FileParAndSeqTest
5 [       OK ] WordCountBlockwiseTest.FileParAndSeqTest (806 ms)
6 [----------] 2 tests from WordCountBlockwiseTest (1239 ms total)
```

2.4. Распараллеливание с помощью OpenMP

"OpenMP (Open Multi-Processing) — открытый стандарт для распараллеливания программ на языках Си, Си++ и Фортран. Дает описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью."

Для управления многопоточным выполнением в OpenMP используется директива #pragma. Например,

Код этой реализации практически аналогичен предыдущей.

Листинг 12: (файл count.h)

```
#ifdef WITH_OPENMP

wordStat countWordsOpenMP(const char* text, size_t len, int threadCount) {
    wordStat stat;
    vector<size_t> blockStart;

    size_t blockSize = len / threadCount + 1;
    size_t startPos = 0;
    size_t endPos;

    for (int i = 0; i < threadCount; i++) {
        blockStart.push_back(startPos);
        endPos = startPos + blockSize;
        while (true) {
            if ((endPos > len) || (text[endPos] == ' ')) {
                break;
            }
            endPos++;
        }
        startPos = endPos + 1;
        if (startPos > len) {
            break;
        }
    }

#pragma omp parallel for
    for (int i = 0; i < threadCount; i++) {
        startPos = blockStart[i];
        endPos = i == (threadCount-1) ? len : blockStart[i + 1];
        auto localMap = countWords(text + startPos, endPos - startPos);
#pragma omp critical(merge)
        for (auto& it: localMap) {
            int prevCount = stat.count(it.first) ? stat[it.first] : 0;
            stat[it.first] = prevCount + it.second;
        }
    }
    return stat;
}

#endif
```

Тесты аналогичны предыдущей реализации. Для получения количества процессоров воспользуемся функцией omp_get_num_procs(). Результаты запуска тестов:

```
[----------] 2 tests from WordCountOpenMPTest
[ RUN      ] WordCountOpenMPTest.FileTest
[       OK ] WordCountOpenMPTest.FileTest (180 ms)
[ RUN      ] WordCountOpenMPTest.FileParAndSeqTest
[       OK ] WordCountOpenMPTest.FileParAndSeqTest (589 ms)
[----------] 2 tests from WordCountOpenMPTest (769 ms total)
```

2.5. Распараллеливание с помощью POSIX Threads

Листинг 13: (файл count.h)

```
123 struct threadData {
124     const char* text;
125     size_t len;
126     size_t start;
127     size_t end;
128 
129     wordStat* stat;
130     pthread_mutex_t* mutex;
131 
132     pthread_t thread_id;
133 };
134 
135 wordStat countWordsPthreads(const char* text, size_t len, int
       threadCount) {
136     wordStat stat;
137     vector<size_t> blockStart;
138     pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
139 
140     size_t blockSize = len / threadCount + 1;
141     size_t startPos = 0;
142     size_t endPos;
143     threadData data[threadCount];
144 
145     for (int i = 0; i < threadCount; i++) {
146         blockStart.push_back(startPos);
147         endPos = startPos + blockSize;
148         while (true) {
149             if ((endPos > len) || (text[endPos] == ' ')) {
150                 break;
151             }
152             endPos++;
153         }
154         startPos = endPos + 1;
155         if (startPos > len) {
156             break;
157         }
158     }
159 
160     for (int i = 0; i < threadCount; i++) {
161         data[i].text = text;
162         data[i].len = len;
163         data[i].start = blockStart[i];
164         data[i].end = i == (threadCount-1) ? len : blockStart[i + 1];
165 
166         data[i].stat = &stat;
167         data[i].mutex = &mutex;
168 
169         int s = pthread_create(&data[i].thread_id, NULL,
170                                &countThread, &data[i]);
171         assert (s == 0);
172     }
173 
```

```
174     for (int i = 0; i < threadCount; i++) {
175         pthread_join(data[i].thread_id, NULL);
176     }
177     pthread_mutex_destroy(&mutex);
178 
179     return stat;
180 }
181 
182 void* countThread(void* arg) {
183     threadData data = *(threadData*) arg;
184 
185     size_t startPos = data.start;
186     size_t endPos = data.end;
187 
188     auto localMap = countWords(data.text + startPos, endPos -
        startPos);
189 
190     pthread_mutex_lock(data.mutex);
191     for (auto& it: localMap) {
192         int prevCount = data.stat->count(it.first) ? (*data.stat)[it.
            first] : 0;
193         (*data.stat)[it.first] = prevCount + it.second;
194     }
195     pthread_mutex_unlock(data.mutex);
196     return NULL;
197 }
```

Код тестов аналогичен предыдущим реализациям. Так как в Pthreads нет функции для получения количества процессоров, используется либо соответствующая функция из OpenMP (если он присутствует в системе), либо функция std::thread::hardwa из состава C++11. Помимо этих способов, при желании можно использовать и другие источники - WinAPI, Linux-специфичные вызовы и т.д. Результаты запуска тестов:

```
1 [----------] 2 tests from WordCountPthreadsTest
2 [ RUN      ] WordCountPthreadsTest.FileTest
3 [       OK ] WordCountPthreadsTest.FileTest (188 ms)
4 [ RUN      ] WordCountPthreadsTest.FileParAndSeqTest
5 [       OK ] WordCountPthreadsTest.FileParAndSeqTest (583 ms)
6 [----------] 2 tests from WordCountPthreadsTest (774 ms total)
```

2.6. Измерение производительности

Для измерения производительности был создан отдельный модуль проекта. Основа модуля - функция doBench, занимающаяся измерением времени выполнения. В процессе своей работы функция использует класс std::chrono::steady_clock из C++11, предоставляющий доступ к монотонному и высокоточному системному таймеру.

В качестве аргумента функции передается функтор (например, лямбда-выражение), благодаря чему можно использовать эту функцию для измерения различных реализаций. Для повышения точности измерения функтор вызывается несколько раз, конкретное количество итераций цикла задается в помощью глобальной перемен-

ной runTimes и в данный момент равняется 100. Время выполнения каждой итерации записывается в массив.

Листинг 14: (файл count.h)

```
60 stats doBench(std::function<void ()> functor) {
61     stats res;
62     cout << "Run number ";
63     for (int i = 0; i < runTimes; i++) {
64         cout << i << " " << flush;
65         auto start = std::chrono::steady_clock::now();
66         functor();
67         auto end = std::chrono::steady_clock::now();
68         auto diff = end - start;
69         res.durations[i] = duration(diff).count();
70     }
71     cout << endl;
72     res.calc();
73 
74     return res;
75 }
```

Над собранными данными проводится статистическая обработка. Вычисляются следующие параметры выборки:

- Математическое ожидание (из предположения о том, что данные описываются нормальным распределением)
- Среднеквадратичное отклонение
- Доверительный интервал

Листинг 15: (файл count.h)

```
1  //
2  // Created by artyom on 23.03.16.
3  //
4  
5  #include <chrono>
6  #include "utils.h"
7  #include "count.h"
8  
9  #include <iostream>
10 #include <fstream>
11 #include <sstream>
12 
13 typedef chrono::duration<double, milli> duration;
14 
15 const int runTimes = 100;
16 
17 struct stats {
18     double durations[runTimes];
19 
20     double mean;
21     double stDev;
22     double stErr;
```

```
23      double margin;
24
25      void calc() {
26          const double z = 1.96; // 95%
27          const int n = runTimes;
28
29          double sum = 0;
30          for (int i = 0; i < n; i++) {
31              sum += durations[i];
32          }
33          mean = sum/n;
34          double variance = 0;
35          for (int i = 0; i < n; i++) {
36              variance += pow(durations[i] - mean, 2);
37          }
38          variance = variance/(n-1);
39          stDev = sqrt(variance);
40          stErr = stDev/sqrt(n);
41          margin = z*stErr;
42      }
43
44      void print(std::string method, int threadNum) {
45          std::string upperCase = method;
46          for (auto & c: upperCase) c = (char) toupper(c);
47          cout << " -- " << upperCase << " " << std::to_string(
                threadNum) << " threads" << " --" << endl;
48          cout << "Mean " << mean << endl;
49          cout << "StDev " << stDev << endl;
50          cout << "Margin " << margin << endl;
51      }
52
53      std::string plotFormat(int threadNum) {
54          std::stringstream ss;
55          ss << threadNum << " " << mean << " " << margin << " " <<
                stDev << endl;
56          return ss.str();
57      }
58  };
```

Сначала измеряется время выполнения последовательной реализации, после чего происходит замер времени выполнения реализаций на основе OpenMP и POSIX Threads. Результаты выводятся на консоль и сохраняются в файл.

Листинг 16: (файл count.h)

```
77  int main() {
78      pair<const char*, size_t> text = loadTextFromFile("book.txt");
79
80      ofstream seqPlotData("stats/seq.txt");
81      ofstream mpPlotData("stats/openmp.txt");
82      ofstream pthPlotData("stats/pthreads.txt");
83
84      stats sequential = doBench([&text]() {
85          countWords(text.first, text.second);
86      });
87      sequential.print("sequential", 1);
88
```

```
89      std:array<int, 8> threadCases;
90      std::iota(threadCases.begin(), threadCases.end(), 1);
91      for (int threadNum: threadCases) {
92          stats openMP = doBench([&text, threadNum]() {
93              countWordsOpenMP(text.first, text.second, threadNum);
94          });
95          openMP.print("openMP", threadNum);
96          mpPlotData << openMP.plotFormat(threadNum);
97
98          stats pthread = doBench([&text, threadNum]() {
99              countWordsPthreads(text.first, text.second, threadNum);
100         });
101         pthread.print("pthread", threadNum);
102         pthPlotData << pthread.plotFormat(threadNum);
103
104         seqPlotData << threadNum << " " << sequential.mean/threadNum
105         << " " << sequential.margin/threadNum << endl;
106     }
107
108     auto counted = countWordsOpenMP(text.first, text.second, 8);
109     stats flip = doBench([&counted]() {
110         flipMap<string, int>(counted);
111     });
112     flip.print("flipMap", 1);
113 }
```

Для первого замера воспользуемся компьютером с процессором Intel Core i5 4690 (Haswell, 4 ядра, 4 потока, 3,5 ГГц, двухканальная память DDR3 8 ГБ). Используется ОС Ubuntu 15.10 с ядром 4.2, компилятор GCC 5.2.1.

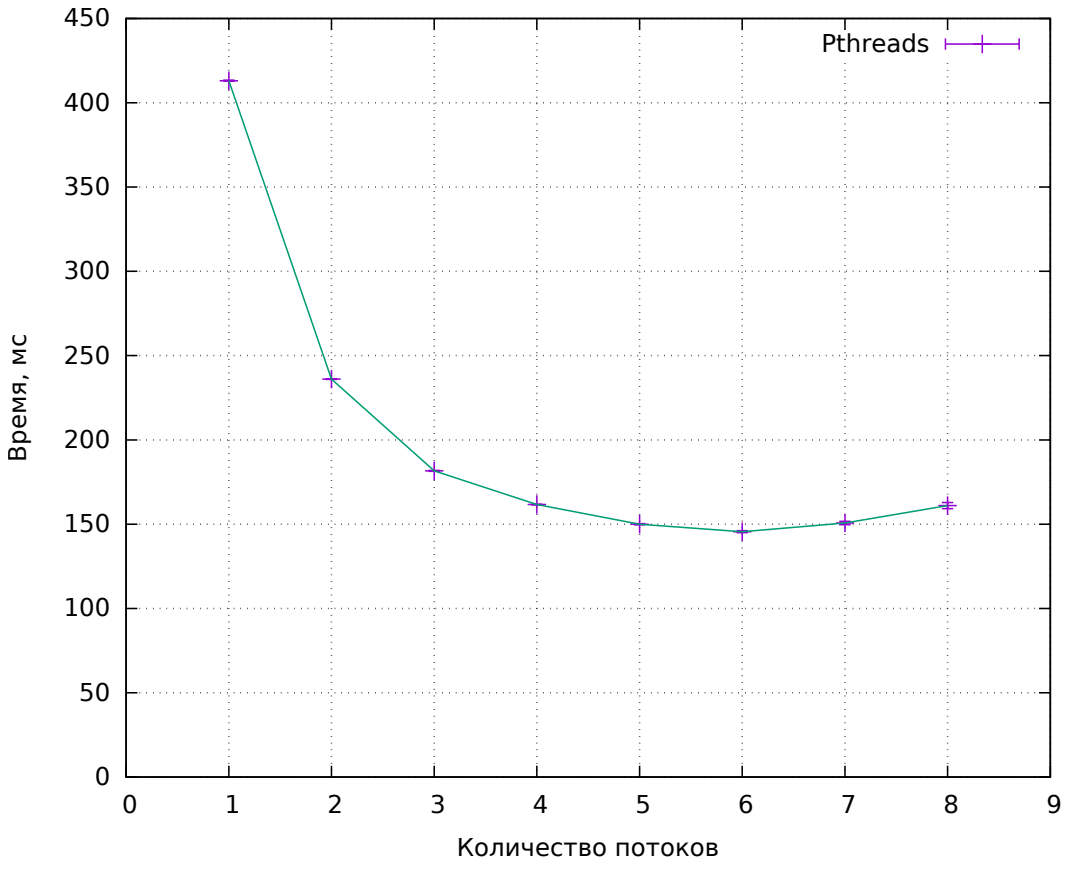

Рис. 1

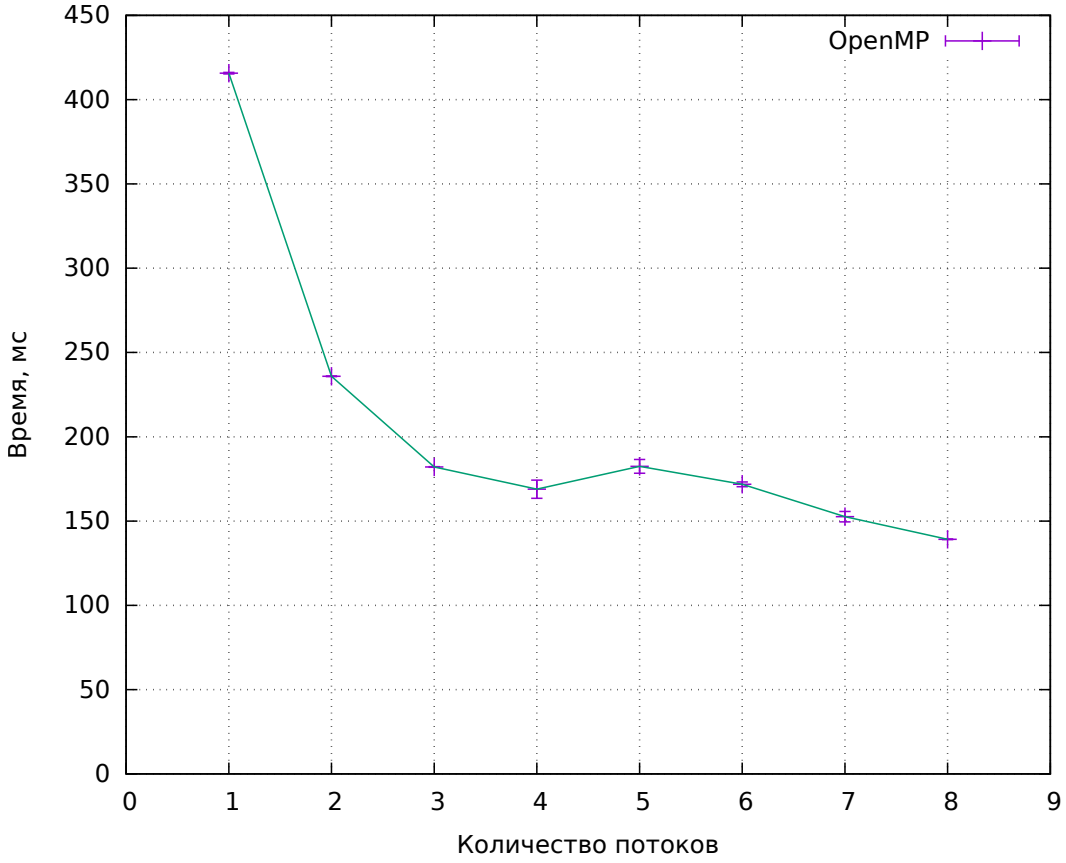

Рис. 2

OpenMP:

Кол-во потоков	Время, мс	Margin	СКО	Ускорение
1	415.70	0.42	2.15	1.00
2	235.92	0.19	0.95	1.76
3	182.10	0.20	1.00	2.28
4	168.89	5.38	27.47	2.46
5	182.44	4.11	20.98	2.28
6	171.79	1.39	7.09	2.42
7	152.63	3.08	15.74	2.72
8	139.19	0.18	0.93	2.99

Pthreads:

Кол-во потоков	Время, мс	Margin	СКО	Ускорение
1	413.10	0.31	1.57	1.00
2	236.06	0.17	0.88	1.75
3	181.63	0.23	1.19	2.27
4	161.65	0.21	1.08	2.56
5	149.97	0.38	1.93	2.75
6	145.56	0.60	3.05	2.84
7	150.71	1.00	5.11	2.74
8	161.04	1.85	9.43	2.57

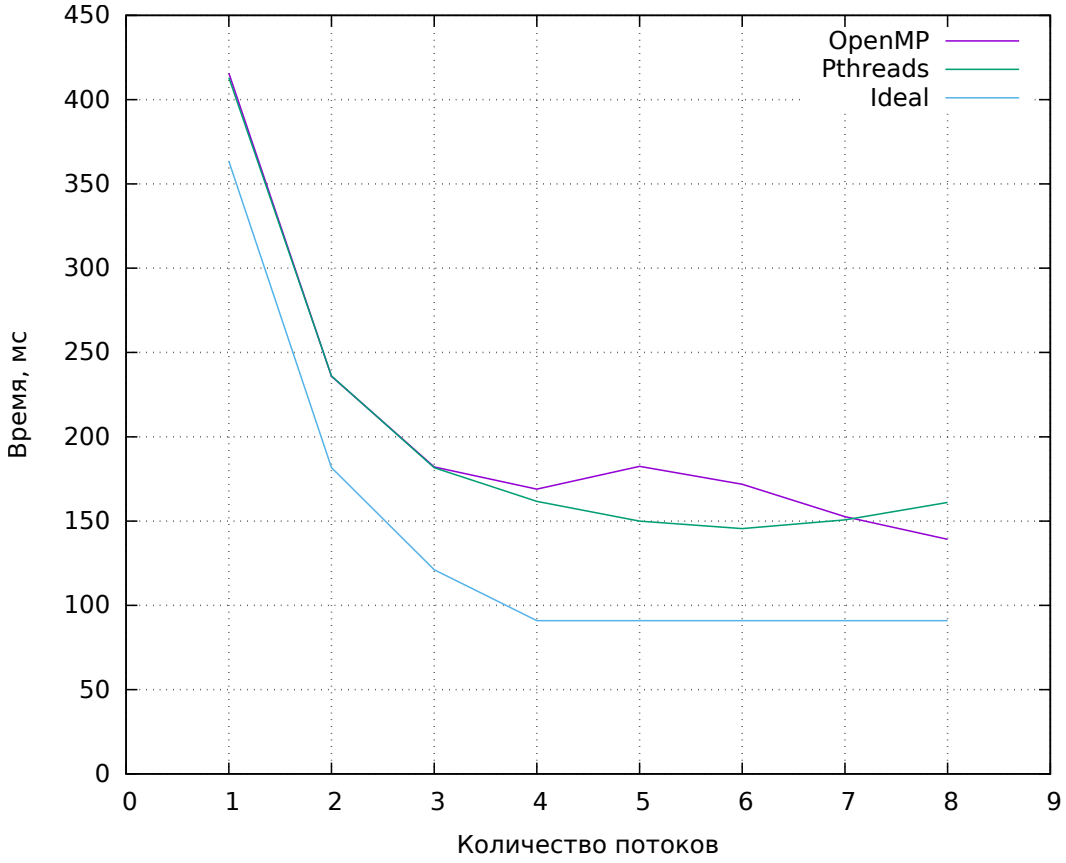

Рис. 3

Проведем аналогичный тест, но с использованием компилятора Clang 3.8.

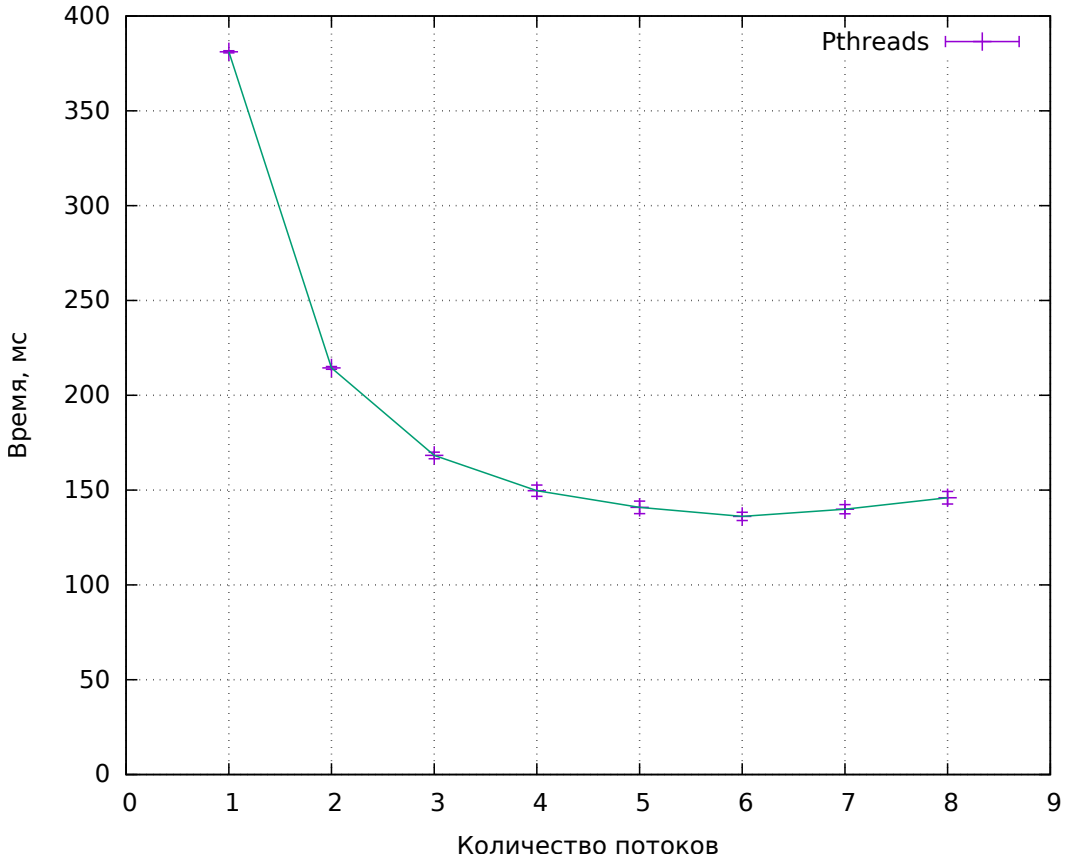

Рис. 4

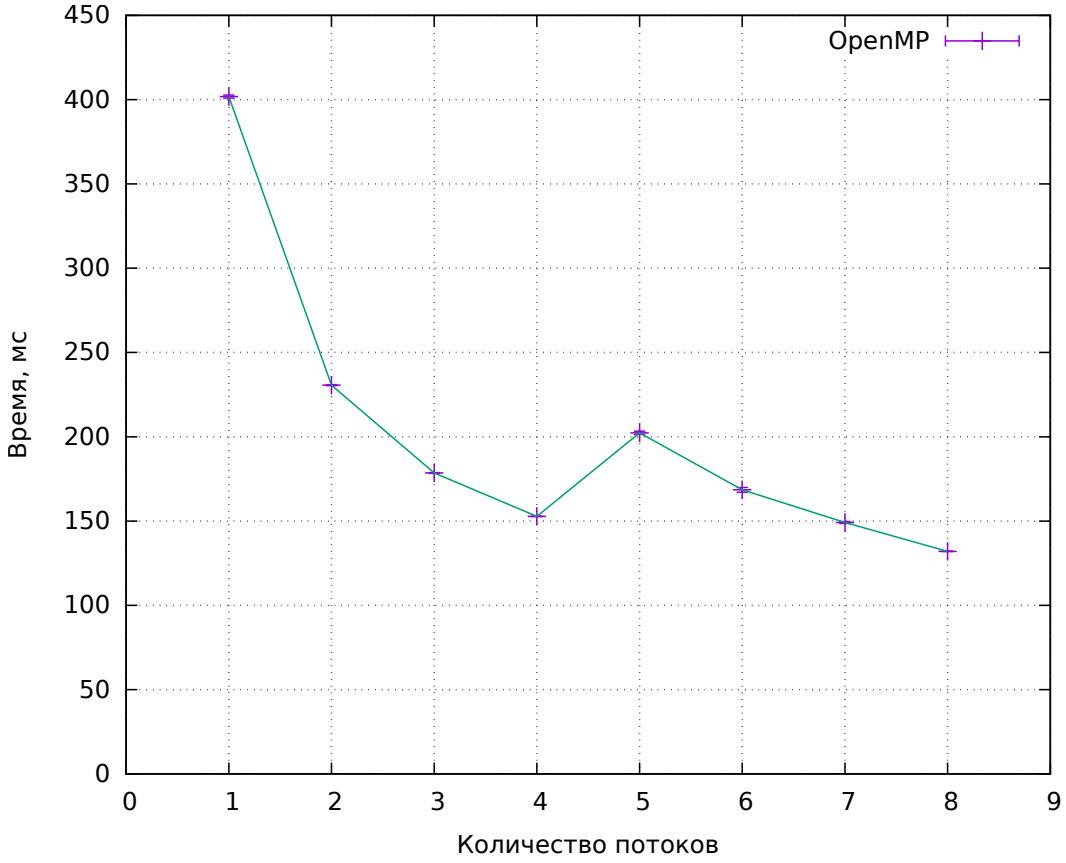

Рис. 5

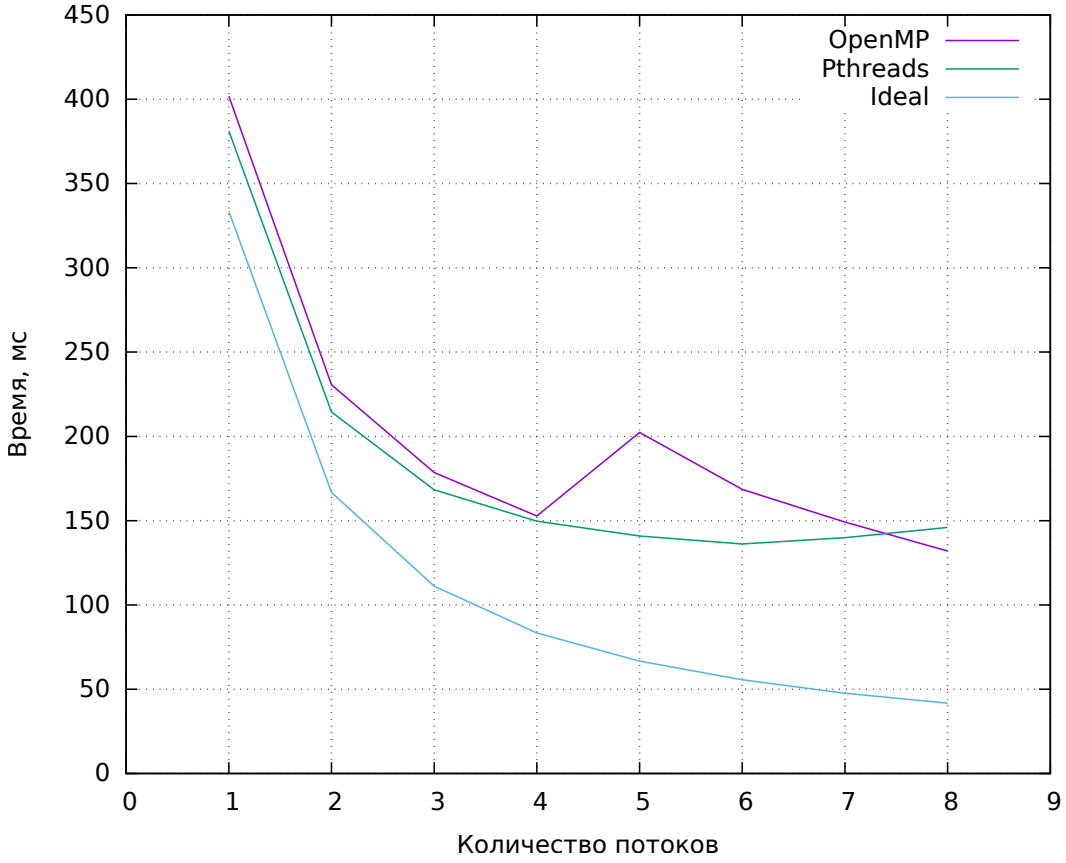

Рис. 6

По сравнению с GCC немного увеличилась производительность, но соотношения остались примерно теми же.

Тестирование на машине с Core i3 5010u (Broadwell, 2 ядра, 4 потока, 2,1 GHz, одноканальная память DDR3 4 Гб) с аналогичным набором ПО:

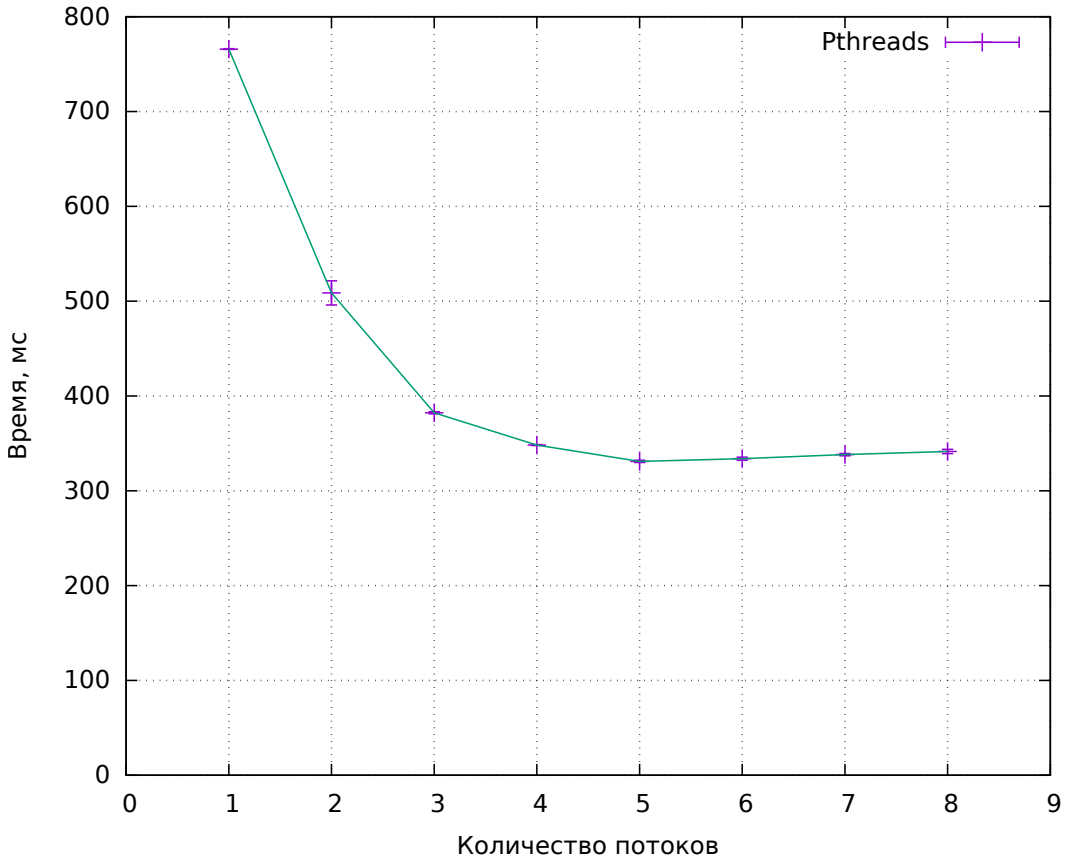

Рис. 7

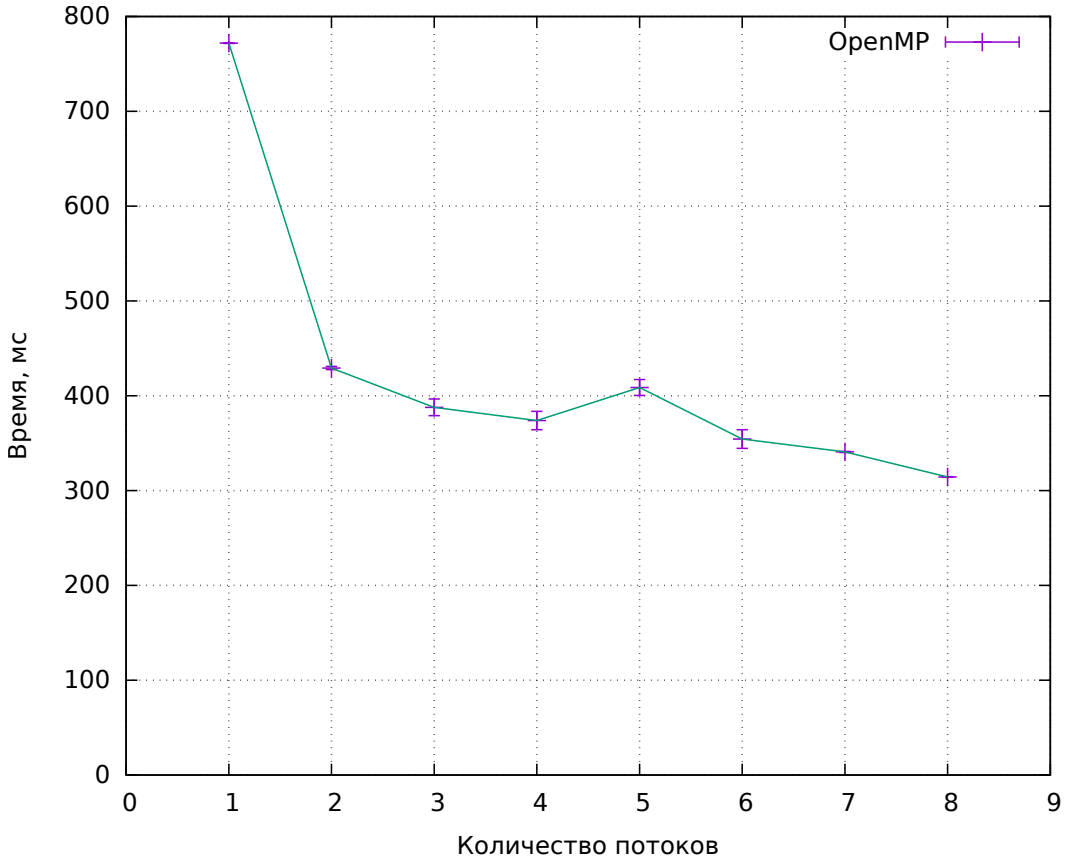

Рис. 8

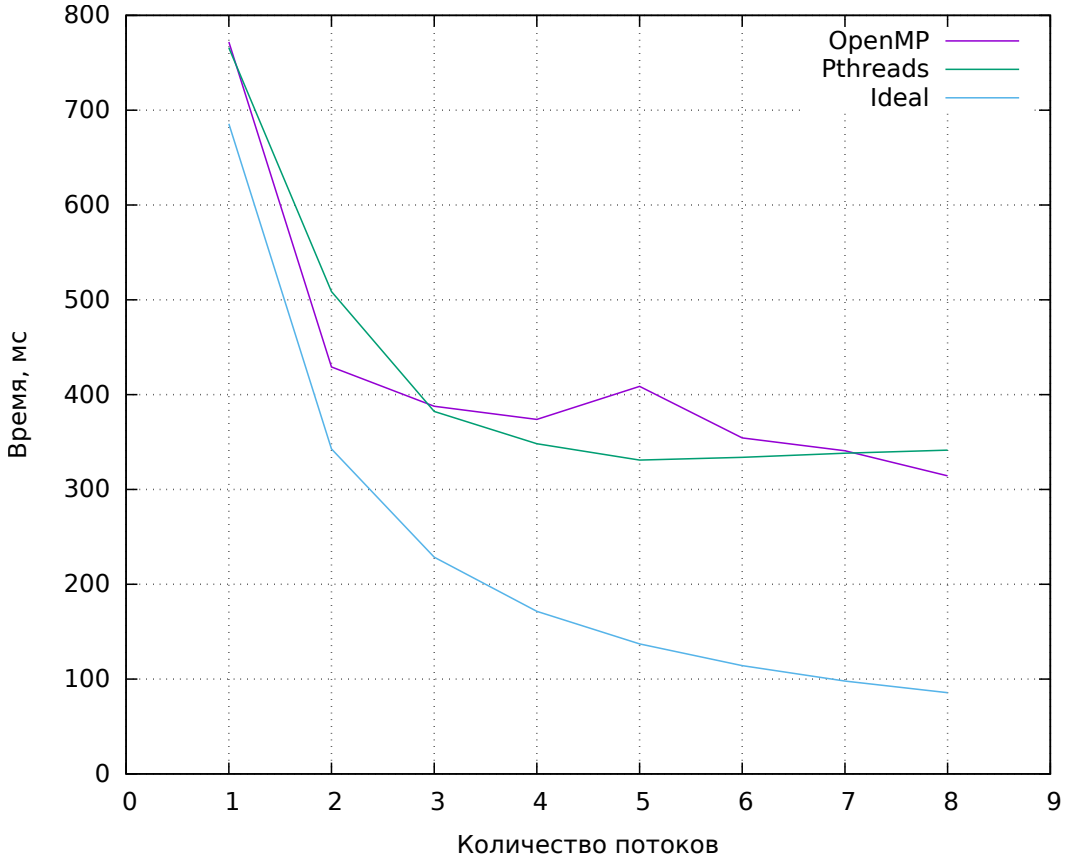

Рис. 9

Что можно заметить:

- HyperThreading дает определенные результаты, скорость работы 4 ядер выше, чем 2. Таким образом, удается более эффективно использовать блоки ЦПУ и память.
- Производительность i3 в целом меньше в 1,5 раза по сравнению с i5 за счет более низкой частоты.
- OpenMP имеет проблемы при 5 потоках. Возможно это связано со сложностями планирования.
- При 8 потоках OpenMP быстрее Pthreads. Возможно это связано с тем, что OpenMP использует некоторый пул потоков, а в случае Pthreads потоки создаются каждый раз заново и на это требуется время

3. Выводы

- Основная идея, примененная при решении этой задачи - разбить задачу на несколько частей (блоков), обрабатывать задачи параллельно, а потом объединить результаты
- Для предоставление эксклюзивного доступа к разделяемому массиву в OpenMP были использованы критические секции, а в Pthreads - мьютексы.

- При увеличении количества потоков производительность растет не линейно - появляются некоторые затраты на синхронизацию. Кроме того, некоторые узлы компьютера (например, интерфейс к памяти) становятся «бутылочным горлышком» и препятствуют дальнейшему ускорению. Решение - переход к других архитектурам, например, NUMA, а также переработке программ.

- Для правильной оценки производительности нужно использоваться стат. обработку

- OpenMP значительно проще в программировании, но допускает меньше контроля и в некоторых ситуациях медленнее