

Федеральное государственное автономное образовательное учреждение
Высшего профессионального образования
Санкт-Петербургский политехнический университет
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Отчёт по лабораторным работам

Работу выполнил студент группы № 53501/3 Жук П.П.

Работу принял преподаватель _____ Стручков И.В.

Санкт-Петербург

2016

Задание

В качестве исследуемого был выбран алгоритм Форда-Беллмана для поиска путей в графе. Для данного алгоритма необходимо:

- Реализовать алгоритм последовательно;
- Реализовать алгоритм с использованием потоков;
- Реализовать алгоритм с использованием OpenMP;

Для каждого из реализованных вариантов:

- Написать юнит-тесты и протестировать программу.
- Замерить время выполнения программы для различного числа потоков.

А также провести серию экспериментов и сравнить эффективность различных реализаций.

Ход работы

Алгоритм Беллмана–Форда — алгоритм поиска кратчайшего пути во взвешенном графе. За время $O(|V| \times |E|)$ алгоритм находит кратчайшие пути от одной вершины графа до всех остальных. В отличие от алгоритма Дейкстры, алгоритм Беллмана–Форда допускает рёбра с отрицательным весом.

Описание алгоритма

Заведём массив расстояний $d[0..n-1]$, который после отработки алгоритма будет содержать ответ на задачу. В начале работы мы заполняем его следующим образом: $d[v] = 0$, а все остальные элементы $d[]$ равны бесконечности ∞ .

Сам алгоритм Форда-Беллмана представляет из себя несколько фаз. На каждой фазе просматриваются все рёбра графа, и алгоритм пытается произвести **релаксацию** (relax, ослабление) вдоль каждого ребра (a, b) стоимости c. Релаксация вдоль ребра — это попытка улучшить значение $d[b]$ значением $d[a]+c$. Фактически это значит, что мы пытаемся улучшить ответ для вершины b, пользуясь ребром (a, b) и текущим ответом для вершины a.

Утверждается, что достаточно $n-1$ фазы алгоритма, чтобы корректно посчитать длины всех кратчайших путей в графе. Для недостижимых вершин расстояние $d[]$ останется равным бесконечности ∞ .

Отдельно надо учесть случай достижимого из вершины v цикла отрицательного веса. В этом случае расстояние до любой вершины достижимой из v равно минус бесконечности и, если не ограничивать число

фаз числом $n-1$, то алгоритм будет работать бесконечно, постоянно улучшая расстояния до этих вершин.

Отсюда мы получаем критерий наличия достижимого цикла отрицательного веса: если после $n-1$ фазы мы выполним ещё одну фазу, и на ней произойдёт хотя бы одна релаксация, то граф содержит цикл отрицательного веса, достижимый из v ; в противном случае, такого цикла нет.

Для целей исследования производительности многопоточных программ задача поиска цикла отрицательного веса не существенна, поэтому в приведенных ниже реализациях учитываться не будут.

Однопоточная реализация программы.

В листинге 1 приведен фрагмент программы, отображающий однопоточную реализацию алгоритма.

Листинг 1

```
const int INF = 1000000000;

struct edge {
    int a, b, cost;
};

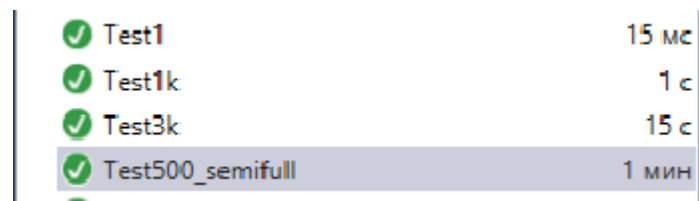
/* n - количество вершин
m - количество ребер
v - стартовая вершина
e - список ребер
*/
string BFplain(int n, int m, vector<edge> const& e, int v) {
    vector<int> d(n, INF);
    d[v] = 0;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            if (d[e[j].a] < INF) {
                if (d[e[j].b] > d[e[j].a] + e[j].cost) {
                    d[e[j].b] = max(-INF, d[e[j].a] +
e[j].cost);
                }
            }
        }
    }

    stringstream ss;
    for (int i = 0; i < n; ++i) {
        ss << d[i] << " ";
    }
    ss << endl;
    return ss.str();
}
```

Весь проект, со всеми исходными кодами можно найти в приложении к проекту или в репозитории (<https://github.com/ppzhuk/bellman-ford-plain>).

Функция BFPlain принимает на вход граф в виде количества вершин, количества ребер, а также списка ребер, стартовую вершину. Затем осуществляется поиск путей, в соответствии с описанным ранее алгоритмом. В конце функции формируется и возвращается строка из длин путей до каждой из вершин.

Также для программы были составлены юнит-тесты и проведено тестирование. Тесты постепенно увеличивают свою сложность. На всех тестовых данных программа работает корректно.



✓ Test1	15 мс
✓ Test1k	1 с
✓ Test3k	15 с
✓ Test500_semifull	1 мин

Рис. 1. Юнит-тесты.

Файлы с тестовыми данными могут быть найдены в директории «\bellman-ford-plain\tests\».

Также было проведено измерение времени работы алгоритма, при этом не учитывалось время ввода данных. Измеренное таким образом время более точно по сравнению с временными показаниями юнит-тестов.

Листинг 2

```
// ....
// считывание данных

string res = "";
unsigned int start_time;

switch (mode) {
case 1:
    start_time = clock();
    res = BFplain(n, m, e, v);
    break;
case 2:
    start_time = clock();
    res = BFThreads(n, m, e, v, threads_number);
    break;
case 3:
    start_time = clock();
    res = BFopenMP(n, m, e, v, threads_number);
    break;
default:
    cout << "incorrect mode" << endl;
```

```

        return 0;
    }

    unsigned int end_time = clock();
    unsigned int work_time = end_time - start_time;
    cout << res;
    cout << "working time: " << work_time << endl;

```

Результаты для каждого из тестов приведены в таблице ниже.

Таблица 1. Время работы программы.

Название теста	Врем работы программы (мс)
Test1	0
Test1k	2410
Test3k	21645
Test500_semifull	158216

Реализация многопоточной программы с использованием потоков.

Поскольку работа проводилась в ОС Windows то, для написания данной версии программы использовались потоки, реализованные в WinAPI.

Листинг 3

```

// -----
// Потоки
// -----

vector<int> * dist;
vector<edge> const* eds;

struct thrd_param {
    int n, from, to;
    boost::barrier *barrier;

    thrd_param(int num, int f, int t, boost::barrier *bar) {
        n = num;
        from = f;
        to = t;
        barrier = bar;
    }
};

DWORD WINAPI thrd_func(LPVOID lpParam)
{
    thrd_param p = *(thrd_param*)lpParam;
    int n = p.n;
    int from = p.from;
    int to = p.to;

```

```

        boost::barrier *barrier = p.barrier;

        for (int i = 0; i < n; ++i) {
            for (int j = from; j <= to; ++j) {
                if ((*dist)[(*edgs)[j].a] < INF) {
                    if ((*dist)[(*edgs)[j].b] >
(*dist)[(*edgs)[j].a] + (*edgs)[j].cost) {
                        (*dist)[(*edgs)[j].b] = max(-INF,
(*dist)[(*edgs)[j].a] + (*edgs)[j].cost);
                    }
                }
            }
            (*barrier).wait();
        }

        ExitThread(0);
    }

string BFThreads(int n, int m, vector<edge> const& e, int v, int thrds_num) {

    edgs = &e;
    dist = new vector<int>(n, INF);
    (*dist)[v] = 0;
    // потоков не может быть больше ребер
    if (thrds_num > m) {
        thrds_num = m;
    }

    boost::barrier barrier(thrds_num);

    // создаем пул потоков
    HANDLE* thrd_pool = new HANDLE[thrds_num];

    thrd_param *p;

    // определяем какую часть списка ребер будет обрабатывать один поток
    int edges_per_thrd = e.size() / thrds_num;
    int edges_left = e.size() % thrds_num;
    int from = 0;
    int to = edges_per_thrd + edges_left - 1;

    for (int i = 0; i < thrds_num; ++i) {
        // задаем параметры потока и запускаем тред
        p = new thrd_param(n, from, to, &barrier);
        thrd_pool[i] = CreateThread(NULL, 0, &thrd_func, p, 0, NULL);
        from = to;
        to += edges_per_thrd;
    }

    WaitForMultipleObjects(thrds_num, thrd_pool, true, INFINITE);
    for (int i = 0; i < thrds_num; i++) {
        CloseHandle(thrd_pool[i]);
    }

    stringstream ss;
    for (int i = 0; i < n; ++i) {
        ss << (*dist)[i] << " ";
    }
    ss << endl;
}

```

```
        return ss.str();  
    }
```

В данной версии программы массив расстояний `dist` и вектор ребер `edges` объявлены глобальными, поскольку используются всеми потоками. Ускорение в работе достигается за счет параллельной обработки списка ребер – каждый поток обрабатывает лишь его часть. Отсюда получаем ограничение на максимальное количество потоков – не имеет смысла использовать потоков больше чем ребер в графе.

Основная функция `BFThreads` создает пул из `thrs_num` потоков, настраивает и запускает каждый из них, затем ждет их завершения и выводит ответ.

Логика отдельного потока реализована в функции `thrd_func`. Алгоритм аналогичен однопоточной реализации за исключением того, что по обработке своего участка списка ребер поток должен дождаться окончания обработки соответствующих участков других потоков, прежде чем перейти к новой итерации. Для этих целей используется примитив синхронизации барьер. Реализацию барьера из WinAPI использовать не удалось, поэтому была подключена библиотека `boost/threads` и барьер был взят оттуда.

Алгоритм Беллмана-Форда говорит о том, что ответ будет получен после n попыток релаксации всех ребер, при этом конкретный порядок релаксации ребер в отдельной итерации не важен. Поэтому при обновлении массива расстояний `dist` отсутствует необходимость блокировки, что положительно сказывается на производительности.

Тестирование и измерение времени выполнения проводилось для 2-3-4-5-8-9 потоков.

✓ Threads2Test1	15 мс
✓ Threads2Test1k	939 мс
✓ Threads2Test3k	8 с
✓ Threads2Test500_semifull	56 с
✓ Threads3Test1	< 1 мс
✓ Threads3Test1k	665 мс
✓ Threads3Test3k	5 с
✓ Threads3Test500_semifull	38 с
✓ Threads4Test1	1 мс
✓ Threads4Test1k	544 мс
✓ Threads4Test3k	4 с
✓ Threads4Test500_semifull	30 с
✓ Threads5Test1k	791 мс
✓ Threads5Test3k	6 с
✓ Threads5Test500_semifull	40 с
✓ Threads8Test1k	564 мс
✓ Threads8Test3k	4 с
✓ Threads8Test500_semifull	33 с
✓ Threads9Test1k	709 мс
✓ Threads9Test3k	5 с
✓ Threads9Test500_semifull	35 с

Рис. 2. Юнит-тесты реализации с потоками.

Таблица 2. Время работы реализации на потоках.

Название теста	Врем работы программы (мс)
Threads2Test1	0
Threads2Test1k	765
Threads2Test3k	8311
Threads2Test500_semifull	58502
Threads3Test1	0
Threads3Test1k	588
Threads3Test3k	5026
Threads3Test500_semifull	40654
Threads4Test1k	503
Threads4Test3k	4631
Threads4Test500_semifull	38123
Threads5Test1k	763
Threads5Test3k	6395
Threads5Test500_semifull	44985
Threads8Test1k	564
Threads8Test3k	4381
Threads8Test500_semifull	35012
Threads9Test1k	708
Threads9Test3k	5569
Threads9Test500_semifull	37550

Реализация многопоточной программы с использованием 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.

Visual Studio поддерживает OpenMP. Для его использования необходимо включить соответствующую настройку в свойствах проекта.

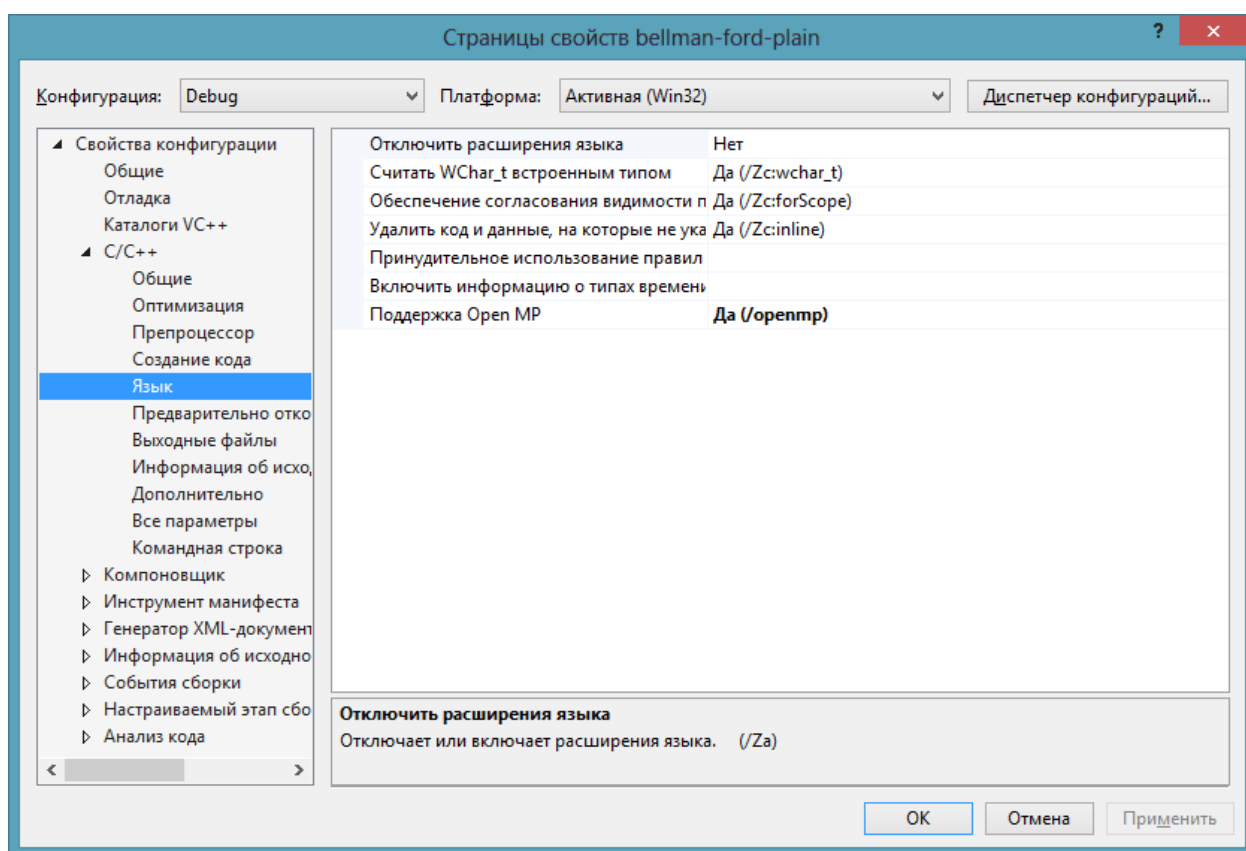


Рис. 3. Настройка OpenMP.

Также необходимо включить заголовок:

```
#include <omp.h>
```

Код соответствующего фрагмента программы выглядит следующим образом.

Листинг 4

```
// -----  
// OpenMP  
// -----
```

```

string BFopenMP(const int n, const int m, vector<edge> const& e, int v, int thrds_num) {
    vector<int> d(n, INF);
    d[v] = 0;
    if (thrds_num > m) {
        thrds_num = m;
    }
    const int edges_per_thrd = e.size() / thrds_num + e.size() % thrds_num;

    omp_set_num_threads(thrds_num);
    int i;
    #pragma omp parallel private (i) shared (d, e)
    {
        for (i = 0; i < n; ++i) {
            #pragma omp for schedule(dynamic, edges_per_thrd)
            for (int j = 0; j < m; ++j) {
                if (d[e[j].a] < INF) {
                    if (d[e[j].b] > d[e[j].a] + e[j].cost) {
                        d[e[j].b] = max(-INF, d[e[j].a] + e[j].cost);
                    }
                }
            }
        }
    }

    stringstream ss;
    for (int i = 0; i < n; ++i) {
        ss << d[i] << " ";
    }
    ss << endl;
    return ss.str();
}

```

Структура кода очень похожа на последовательную версию программы. `edges_per_thrd` определяет сколько ребер должно обрабатываться в каждом потоке. `omp_set_num_threads(thrds_num)` задает количество потоков.

Директива

```
#pragma omp parallel private (i) shared (d, e)
```

говорит о том, что следующий блок кода надо выполнить параллельно. При этом переменная `i` создается своя для каждого потока, в то время как `d` и `e` являются общими.

Директива

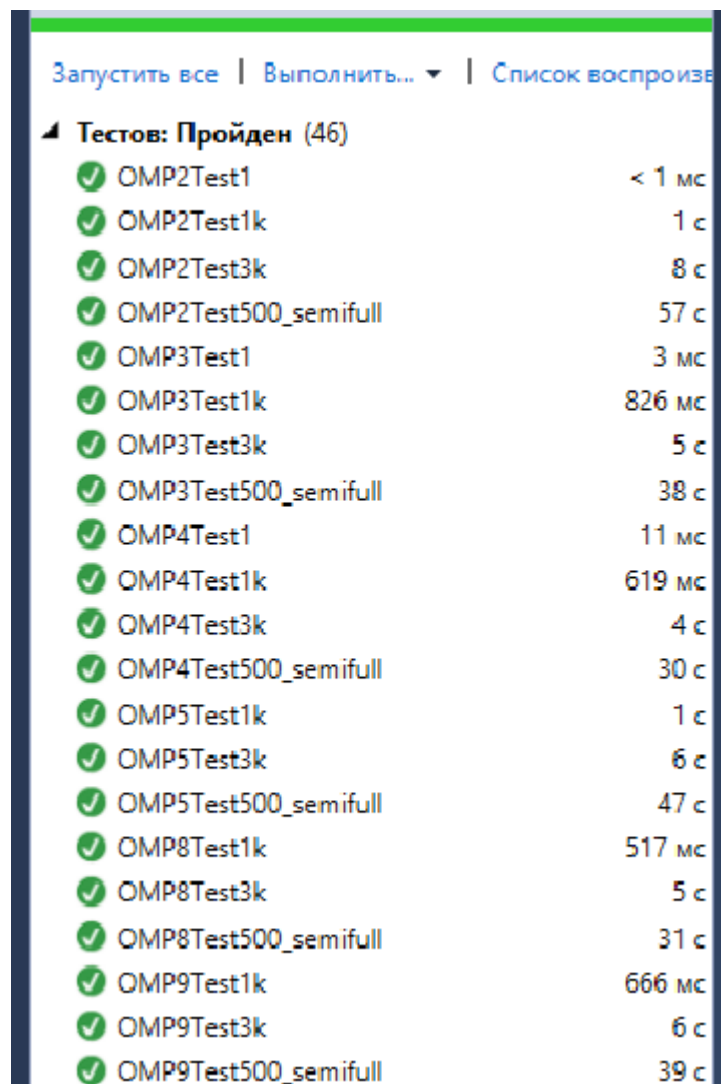
```
#pragma omp for schedule(dynamic, edges_per_thrd)
```

порционно делит идущий следом цикл на `thrds_num` потоков. Режим `dynamic` говорит о том, что если один из потоков обработал свой блок, то ему будет передан следующий, при его наличии.

Директивы OpenMP подразумевают наличие неявных барьеров по окончании параллельных блоков. Благодаря чему нам нет необходимости

использовать барьер по окончании внутреннего цикла или в явном виде ожидать завершения всех потоков в конце параллельного участка программы.

Таким образом, как видно по тестам, программа с OpenMP, за счет динамического распределения нагрузки работает немного быстрее, чем с обычными потоками, при этом код прост и практически не отличается от последовательной программы.



Запустить все Выполнить... ▾ Список воспроизведения	
▲ Тестов: Пройден (46)	
✓ OMP2Test1	< 1 мс
✓ OMP2Test1k	1 с
✓ OMP2Test3k	8 с
✓ OMP2Test500_semifull	57 с
✓ OMP3Test1	3 мс
✓ OMP3Test1k	826 мс
✓ OMP3Test3k	5 с
✓ OMP3Test500_semifull	38 с
✓ OMP4Test1	11 мс
✓ OMP4Test1k	619 мс
✓ OMP4Test3k	4 с
✓ OMP4Test500_semifull	30 с
✓ OMP5Test1k	1 с
✓ OMP5Test3k	6 с
✓ OMP5Test500_semifull	47 с
✓ OMP8Test1k	517 мс
✓ OMP8Test3k	5 с
✓ OMP8Test500_semifull	31 с
✓ OMP9Test1k	666 мс
✓ OMP9Test3k	6 с
✓ OMP9Test500_semifull	39 с

Рис. 4. Юнит-тесты реализации с OpenMP.

Таблица 3. Время работы реализации с OpenMP.

Название теста	Врем работы программы (мс)
OMP2Test1	0
OMP2Test1k	1013
OMP2Test3k	8419
OMP2Test500_semifull	57369
OMP3Test1	0
OMP3Test1k	788
OMP 3Test3k	4604
OMP 3Test500_semifull	37048
OMP 4Test1k	592
OMP 4Test3k	4268
OMP 4Test500_semifull	36718
OMP 5Test1k	971
OMP 5Test3k	6457
OMP 5Test500_semifull	48921
OMP 8Test1k	498
OMP 8Test3k	5180
OMP 8Test500_semifull	31457
OMP 9Test1k	663
OMP 9Test3k	5493
OMP 9Test500_semifull	3895

Проведение эксперимента

Для проведения точных измерений был проведен эксперимент, в котором каждая из реализаций была запущена 50 раз на тесте Test500_semifull.

Данный тест представляет собой граф из 500 вершин, каждая из которых имеет 250 исходящих ребер с различными весами. Располагается тест в bellman-ford-plain\tests.

Для проведения эксперимента была написана небольшая утилита (test_script), которая последовательно запускала нужное количество раз каждую из реализаций, замеряла время и протоколировала результат.

Затем была составлена таблица результатов (отчет.xlsx), посчитаны мат. ожидание, с.к.о., построена гистограмма.

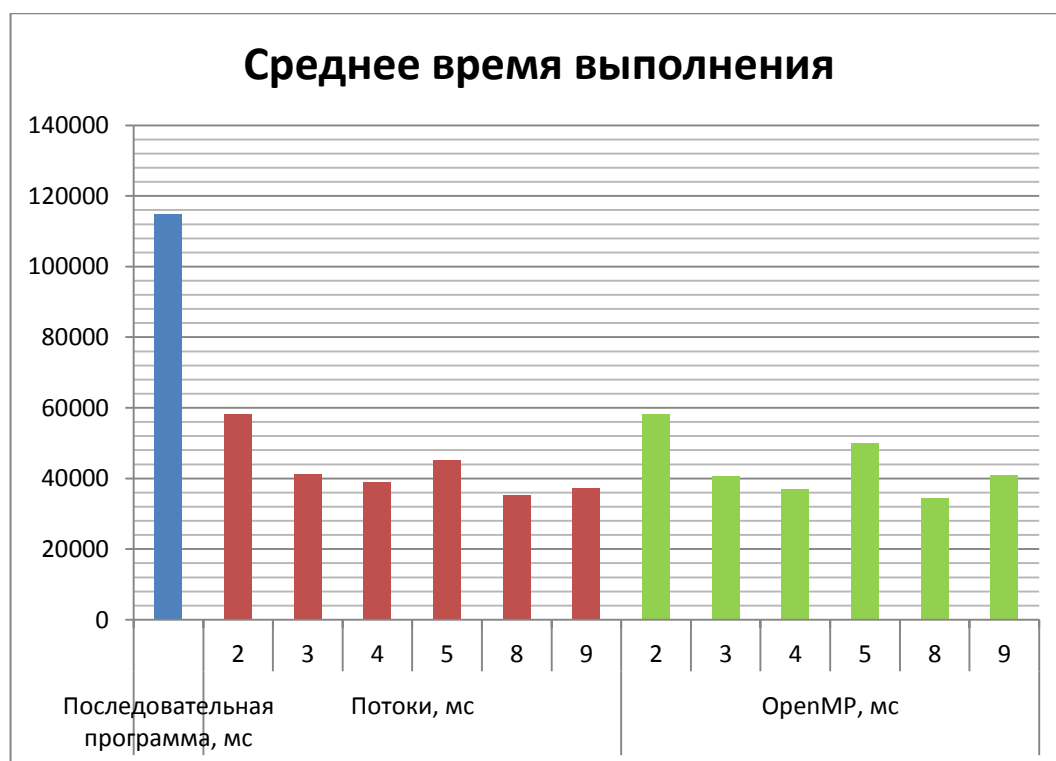


Рис. 5. Гистограмма среднего времени работы.

Стоит отметить общую закономерность для многопоточных реализаций. Ускорение происходит с каждым увеличением количества потоков до тех пор, пока не будут задействованы все ядра (система, на которой проводилось тестирование, имеет 4 ядра). Затем, каждое нечетное число потоков работает медленнее, чем предыдущее четное.

Несмотря на то, что в системе установлено 4 ядра, максимальной скорости работы удалось достичь на 8 потоках. Что может говорить о том, что данное количество потоков эффективно использует простой процессора.

OpenMP в одних случаях работает быстрее обычных потоков, в других – медленнее, но наибольшей производительности удалось достичь именно с OpenMP на 8 потоках – 34454 мс. В то время как однопоточная программа в среднем выполнялась за 114893 мс.

СКО в среднем составило 601 мс. Дополнительная информация приведена в таблице ниже.

Таблица 4. Статистика.

Среднее время выполнения однопоточной программы, мс	114893
Лучшее среднее время выполнения – потоки, мс	35384(8 потоков)
Лучшее среднее время выполнения – OpenMP, мс	34454(8 потоков)
Лучшее время выполнения – потоки, мс	35152(8 потоков)
Лучшее время выполнения – OpenMP, мс	30369(8 потоков)
Минимальный прирост производительности, раз	1,97 (2 потока)
Максимальный прирост производительности, раз	3,33 (8 потоков, OpenMP)

Дополнительную информацию можно посмотреть в файле отчет.xlsx, который приложен к отчету.

Вывод

Наилучшего результата по времени выполнения удалось достичь с 8 потоками с применением OpenMP. Время выполнения программы сократилось с 114893 мс до 30369 мс. При этом стоит отметить, что OpenMP требует внесения минимума изменений в исходную последовательную программу (по сравнению с обычными потоками).

На протяжении многих лет, производители процессоров постоянно увеличивали тактовую частоту и параллелизм на уровне инструкций, так что на новых процессорах старые однопоточные приложения исполнялись быстрее без каких-либо изменений в программном коде. Сейчас по разным причинам производители процессоров предпочитают многоядерные архитектуры, и для получения всей выгоды от возросшей производительности ЦП программы должны переписываться в соответствующей манере.