Государственное образовательное учреждение высшего профессионального образования
“Московский государственный технический университет имени Н.Э.Баумана”

Дисциплина: Анализ алгоритмов

Лабораторная работа №4

Параллельная реализация алгоритма Винограда

Студент группы ИУ7-54Б,
Котов Никита

2019 г.

Содержание

Введение

Многопоточность - способность центрального процессора (CPU) или одного ядра в многоядерном процессоре одновременно выполнять несколько процессов или потоков, соответствующим образом поддерживаемых операционной системой. Этот подход отличается от многопроцессорности, так как многопоточность процессов и потоков совместно использует ресурсы одного или нескольких ядер: вычислительных блоков, кэш-памяти ЦПУ или буфера перевода с преобразованием (TLB).

В тех случаях, когда многопроцессорные системы включают в себя несколько полных блоков обработки, многопоточность направлена на максимизацию использования ресурсов одного ядра, используя параллелизм на уровне потоков, а также на уровне инструкций. Поскольку эти два метода являются взаимодополняющими, их иногда объединяют в системах с несколькими многопоточными ЦП и в ЦП с несколькими многопоточными ядрами.

Многопоточная парадигма стала более популярной с конца 1990-х годов, поскольку усилия по дальнейшему использованию параллелизма на уровне инструкций застопорились. Смысл многопоточности - квазимногозадачность на уровне одного исполняемого процесса. Значит, все потоки процесса помимо общего адресного пространства имеют и общие дескрипторы файлов. Выполняющийся процесс имеет как минимум один (главный) поток.

Многопоточность (как доктрину программирования) не следует путать ни с многозадачностью, ни с многопроцессорностью, несмотря на то, что операционные системы, реализующие многозадачность, как правило, реализуют и многопоточность. Достоинства:

- облегчение программы посредством использования общего адресного пространства.
- меньшие затраты на создание потока в сравнении с процессами.
- повышение производительности процесса за счёт распараллеливания процессорных вычислений.
- если поток часто теряет кэш, другие потоки могут продолжать использовать неиспользованные вычислительные ресурсы.

Недостатки:

- несколько потоков могут вмешиваться друг в друга при совместном использовании аппаратных ресурсов
- с программной точки зрения аппаратная поддержка многопоточности более трудоемка для программного обеспечения
- проблема планирования потоков
- специфика использования. Вручную настроенные программы на ассемблере, использующие расширения MMX или AltiVec и выполняющие предварительные выборки данных, не страдают от потерь кэша или неиспользуемых вычислительных ресурсов. Таким образом, такие программы не выигрывают от аппаратной многопоточности и действительно могут видеть ухудшенную производительность из-за конкуренции за общие ресурсы[1].

Несмотря на существующие недостатки, многопоточная парадигма имеет огромный потенциал, поэтому данная лабораторная работа будет посвящена распараллеливанию реализованного ранее алгоритма Винограда для умножения матриц.
В рамках выполнения работы необходимо решить следующие задачи:

- изучить понятие параллельный вычислений;
- реализовать последовательный и параллельный алгоритм Винограда;
- сравнить временные характеристики реализованных алгоритмов экспериментально;
- на основании проделанной работы сделать выводы.

1 Аналитическая часть

1.1 Описание алгоритма

1.1.1 Алгоритм Винограда

Рассматривая результат умножения двух матриц очевидно, что каждый элемент в нем представляет собой скалярное произведение соответствующих строки и столбца исходных матриц. Такое умножение допускает предварительную обработку, позволяющую часть работы выполнить заранее.

Рассмотрим два вектора V = (v1, v2, v3, v4) и W = (w1, w2, w3, w4). Их скалярное произведение равно:

$$V * W = v1w1 + v2w2 + v3w3 + v4w4 \tag{1}$$

Это равенство можно переписать в виде:

$$V * W = (v1 + w2)(v2 + w1) + (v3 + w4)(v4 + w3) - v1v2 - v3v4 - w1w2 - w3w4 \tag{2}$$

Несмотря на то, что второе выражение требует вычисления большего количества операций, чем стандартный алгоритм: вместо четырех умножений - шесть, а вместо трех сложений - десять, выражение в правой части последнего равенства допускает предварительную обработку: его части можно вычислить заранее и запомнить для каждой строки первой матрицы и для каждого столбца второй, то для каждого элемента будет необходимо выполнить лишь первые два умножения и последующие пять сложений, а также дополнительно два сложения. Из-за того, что операция сложения быстрее операции умножения, алгоритм должен работать быстрее стандартного.

2 Конструкторская часть

2.1 Разработка алгоритмов

2.1.1 Алгоритм Винограда

На рис. 1 и рис. 2 приведена схема алгоритма Винограда

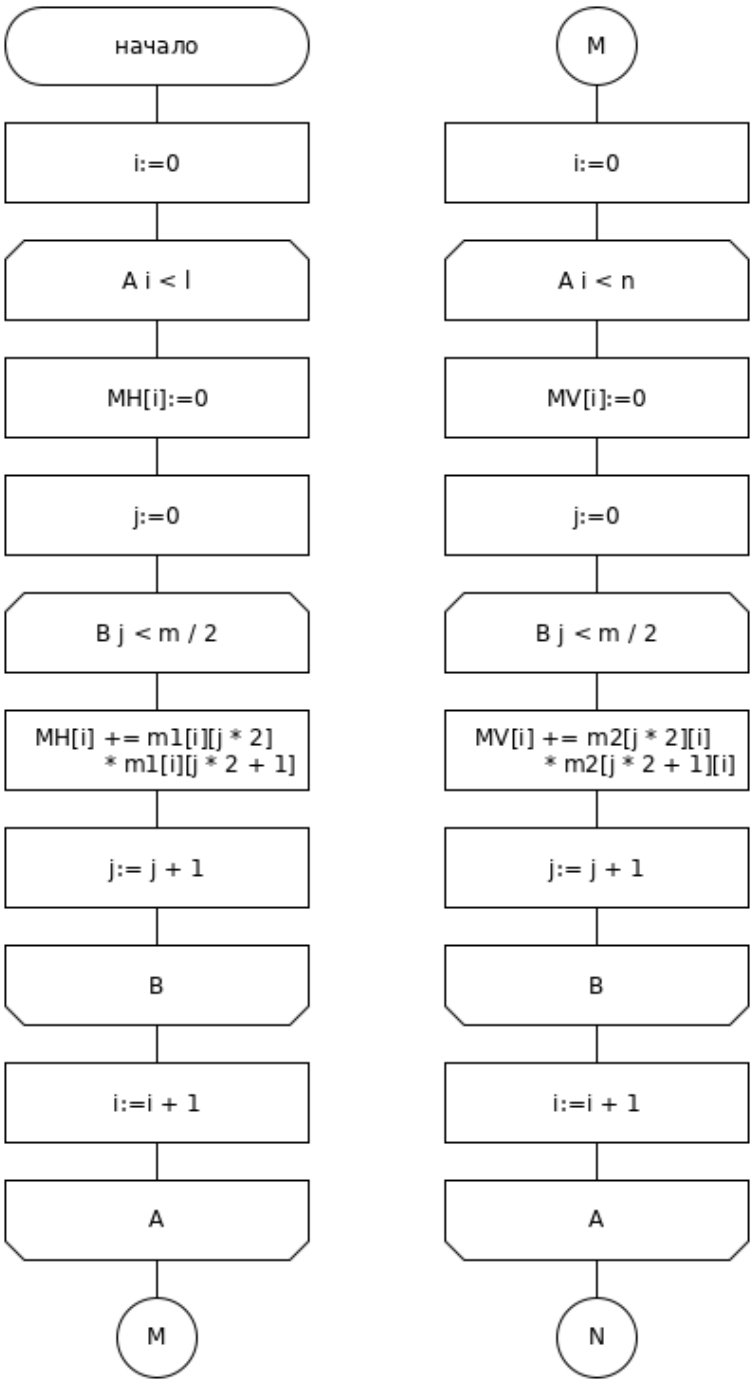

Рис. 1: Алгоритм Винограда

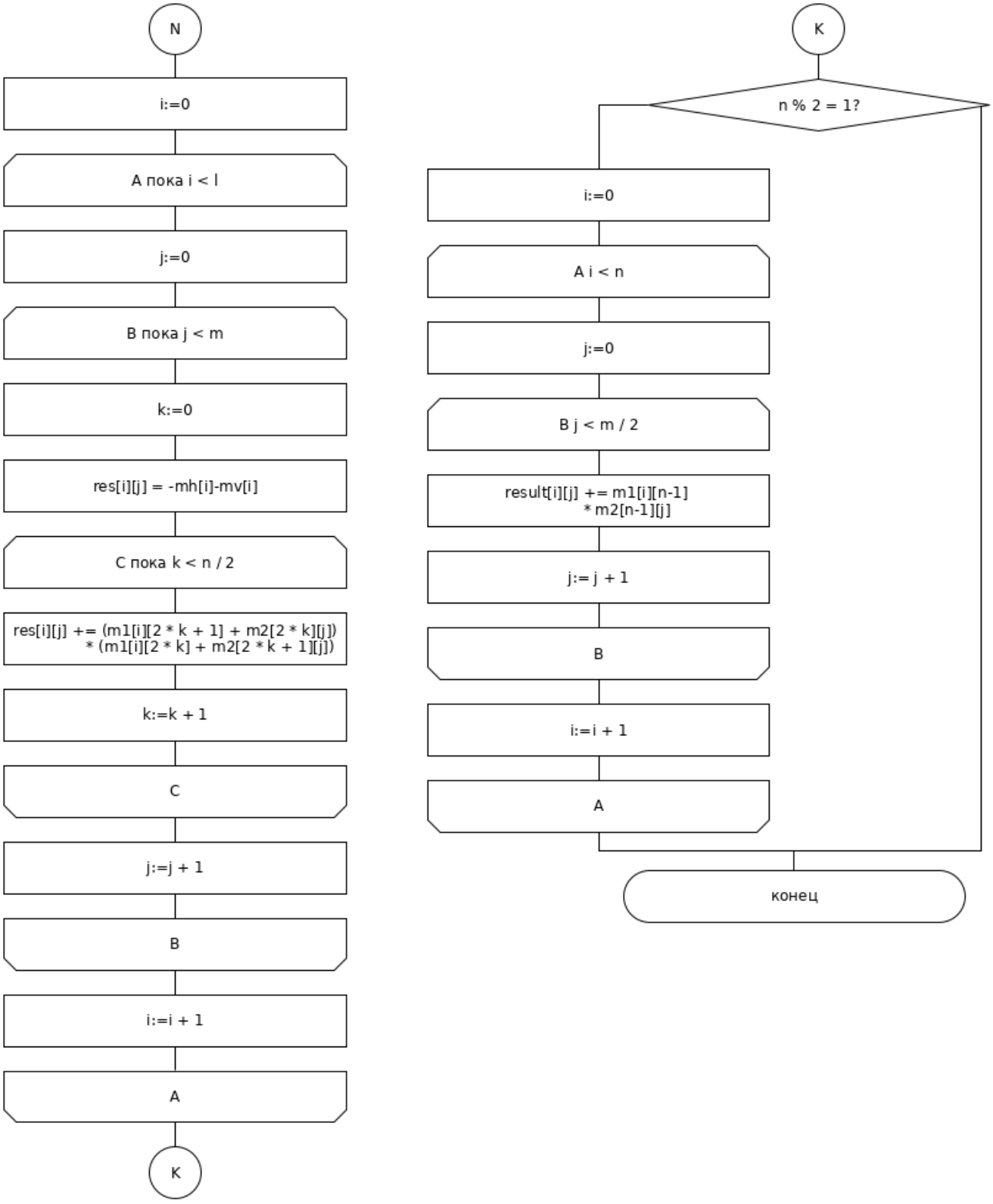

Рис. 2: Алгоритм Винограда

Видно, что для алгоритма Виноградова худшим случаем являются матрицы нечетного размера, а лучшим четного, т.к. отпадает необходимость в последнем цикле.
В качестве оптимизаций можно:

- заранее считать в MH и MV отрицательные произведения
- заменить выражения вида $a = a + ...$ на $a+ = ...$

- в циклах по k сделать шаг 2, избавившись тем самым от двух операций умножения на каждую итерацию

Рассмотрим способы распараллеливания алгоритма.

- Прежде всего предварительные вычисления MH и MV не зависят друг от друга, значит, их можно вычислить параллельно;
- Если количество потоков больше 2ух, то вычисления и MH, и MV можно распараллеть, выделив каждому из $n\_threads/2$ потоков, где n_threads - кол-во доступных потоков, вычисление своего участка длинной $l/n\_threads$ и $n/n\_threads$ соответственно;
- Аналогичным способом распараллеливается вычисление матрицы.

2.2 Выводы по конструкторскому разделу

В результате работы над конструкторским разделом была разработана схема алгоритма Винограда и выбран способ распараллеливания алгоритма.

3 Технологическая часть

3.1 Средства реализации

В качестве языка программирования был выбран C++, так как он предоставляет широкие возможности для эффективной реализации алгоритмов. Для распараллеливания алгоритма выбраны std::thread, предоставляемые стандартно библиотекой C++. Они имеют достаточно удобный интерфейс для поставленных в данной работе задач.

3.2 Листинг кода

Листинг 1: последовательный Алгоритм Винограда

```
row_d get_negative_row_products(matrix_d &matrix, size_t m, size_t n) {
    auto result = row_d(m, 0.);
    for (size_t i = 0; i < m; i++) {
        for (size_t j = 0; j < n - 1; j += 2) {
            result[i] -= matrix[i][j] * matrix[i][j + 1];
        }
    }

    return result;
}

row_d get_negative_col_products(matrix_d &matrix, size_t m, size_t n) {
    auto result = row_d(m, 0.);
    for (size_t i = 0; i < m; i++) {
        for (size_t j = 0; j < n - 1; j += 2) {
            result[i] -= matrix[j][i] * matrix[j + 1][i];
        }
    }

    return result;
}

matrix_d vinograd_multiplication(matrix_d &m1, matrix_d &m2) {
    auto l = m1.size();
    auto m = m2.size();
    auto n = m2[0].size();

    if (m1[0].size() != m) {
        throw std::exception();
    }

    auto mh = get_negative_row_products(m1, l, m);
    auto mv = get_negative_col_products(m2, n, m);

    auto result = init_matrix(l, n);
    for (size_t i = 0; i < l; i++) {
```

```
        for (size_t j = 0; j < m; j++) {
            result[i][j] = mh[i] + mv[j];
            for (size_t k = 0; k < n - 1; k += 2) {
                result[i][j] += (m1[i][k + 1] + m2[k][j]) * (m1[i][k] + m2[k +
                    1][j]);
            }
        }
    }

    if (n % 2 == 1) {
        for (size_t i = 0; i < l; i++) {
            for (size_t j = 0; j < m; j++) {
                result[i][j] += m1[i][n - 1] * m2[n-1][j];
            }
        }
    }

    return result;
}
```

Листинг 2: параллельный Алгоритм Винограда

```
row_d parallel_negative_row_products(matrix_d &matrix, size_t m, size_t n,
    size_t threads_num) {
    auto result = row_d(m, 0.);
    auto lambda = [&](size_t i_start, size_t i_end) {
        for (size_t i = i_start; i < i_end; i++) {
            for (size_t j = 0; j < n - 1; j += 2) {
                result[i] -= matrix[i][j] * matrix[i][j + 1];
            }
        }
    };

    std::vector<std::thread> threads(threads_num);
    auto line_for_thread = m / threads_num;
    if (line_for_thread == 0) {
        line_for_thread = 1;
    }

    for (size_t i = 0; i < threads_num && i < m; i++) {
        size_t start = i * line_for_thread;
        size_t end = i == threads_num - 1? m: (i + 1) * line_for_thread;
        auto thread = std::thread(lambda, start, end);
        threads[i] = std::move(thread);
    }

    for (auto &thr: threads) {
        thr.join();
    }

    return result;
}

row_d parallel_negative_col_products(matrix_d &matrix, size_t m, size_t n,
```

```
    size_t threads_num) {
    auto result = row_d(m, 0.);
    auto lambda = [&](size_t i_start, size_t i_end) {
        for (size_t i = i_start; i < i_end; i++) {
            for (size_t j = 0; j < n - 1; j += 2) {
                result[i] -= matrix[j][i] * matrix[j + 1][i];
            }
        }
    };

    std::vector<std::thread> threads(threads_num);
    auto line_for_thread = m / threads_num;
    if (line_for_thread == 0) {
        line_for_thread = 1;
    }

    for (size_t i = 0; i < threads_num && i < m; i++) {
        size_t start = i * line_for_thread;
        size_t end = i == threads_num - 1? m: (i + 1) * line_for_thread;
        auto thread = std::thread(lambda, start, end);
        threads[i] = std::move(thread);
    }

    for (auto &thr: threads) {
        if (thr.joinable()) {
            thr.join();
        }
    }

    return result;
}

matrix_d parallel_vinograd_mult(matrix_d &m1, matrix_d &m2, size_t threads_num)
    {
    auto l = m1.size();
    auto m = m2.size();
    auto n = m2[0].size();

    if (m1[0].size() != m) {
        throw std::exception();
    }

    row_d mh, mv;
    if (threads_num > 2) {
        std::thread mh_thread([&](){ mh = get_negative_row_products(m1, l, m);
            });
        std::thread mv_thread([&](){ mv = get_negative_col_products(m2, n, m);
            });
        mh_thread.join();
        mv_thread.join();
    } else {
        mh = parallel_negative_row_products(m1, l, m, threads_num / 2);
        mv = parallel_negative_col_products(m2, n, m, threads_num - threads_num
```

```
            / 2);
    }

    std::vector<std::thread> threads(threads_num);
    auto line_for_thread = l / threads_num;
    if (line_for_thread == 0) {
        line_for_thread = 1;
    }

    auto result = init_matrix(l, n);
    for (size_t i = 0; i < threads_num && i < l; i++) {
        size_t start = i * line_for_thread;
        size_t end = i == threads_num - 1? l: (i + 1) * line_for_thread;
        auto thread = std::thread(
            [&](size_t i_start, size_t i_end) {
                for (size_t local_i = i_start; local_i < i_end; local_i++) {
                    for (size_t j = 0; j < m; j++) {
                        result[local_i][j] = mh[local_i] + mv[j];
                        for (size_t k = 0; k < n-1; k += 2) {
                            result[local_i][j] += (m1[local_i][k + 1] + m2[k][j])
                                * (m1[local_i][k] + m2[k + 1][j]);
                        }
                    }
                }
            }, start, end);
        threads[i] = std::move(thread);
    }

    for (auto &thr: threads) {
        if (thr.joinable()) {
            thr.join();
        }
    }

    if (n % 2 == 1) {
        for (size_t i = 0; i < l; i++) {
            for (size_t j = 0; j < m; j++) {
                result[i][j] += m1[i][n - 1] * m2[n-1][j];
            }
        }
    }

    return result;
}
```

3.3 Тестирование фунций

В таблице 1 приведены тесты для последовательной и параллельной реализаций алгоритма Винограда.

Таблица 1: Тестирование функций

Матрица 1	Матрица 2	Ожидаемый результат	Посл. алг.	Парал. алг.
$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 6 & 12 & 18 \\ 6 & 12 & 18 \\ 6 & 12 & 18 \end{pmatrix}$	$\surd$	$\surd$
$\begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix}$	$\begin{pmatrix} 3 & 6 \\ 3 & 6 \end{pmatrix}$	$\surd$	$\surd$
$\begin{pmatrix} 2 \end{pmatrix}$	$\begin{pmatrix} 2 \end{pmatrix}$	$\begin{pmatrix} 4 \end{pmatrix}$	$\surd$	$\surd$
$\begin{pmatrix} 1 & -2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} -1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 0 & 4 & 6 \\ 4 & 12 & 18 \\ 4 & 12 & 18 \end{pmatrix}$	$\surd$	$\surd$

4 Экспериментальная часть

4.1 Тестирование времени работы функций

Для измерения времени использовались функции std::chrono, предоставляемые стандартной библиотекой C++. Чтобы исключить случайные отклонения в измеренном времени, измерялось время работы 10 запусков функции и делилось на 10. Замеры проводятся на процессоре с 2 физическими и 4 логическими ядрами i5 7200u.

В таблице 2 приведены замеры времени работы на квадратных матрицах последовательной и параллельной реализаций алгоритма Винограда, на основе них построены графики рис 3.

Таблица 2: Время работы реализаций алгоритмов

Размер матрицы	1 поток	2 потока	4 потока	8 потоков	16 потоков	32 потока	64 потока
100	0.0028	0.0024	0.0020	0.0023	0.0026	0.0036	0.0069
200	0.0283	0.0226	0.0155	0.0158	0.0163	0.0181	0.0200
300	0.1208	0.0668	0.0843	0.0601	0.0619	0.0680	0.0692
400	0.3061	0.1551	0.1690	0.1491	0.1587	0.1469	0.1657
500	0.7427	0.4486	0.3058	0.3075	0.3058	0.3118	0.3594
600	1.7297	1.0926	1.0023	0.7592	0.7123	0.6617	0.6699
700	2.8254	1.6326	1.2640	1.3399	1.2311	1.2575	1.3509
800	4.6763	2.4869	2.9249	2.6242	2.5122	2.4741	2.5021
900	7.2538	3.5246	5.6764	5.3094	5.2693	5.2410	5.2160
1000	10.1448	4.9473	8.6905	8.7452	8.7369	8.7173	8.7413

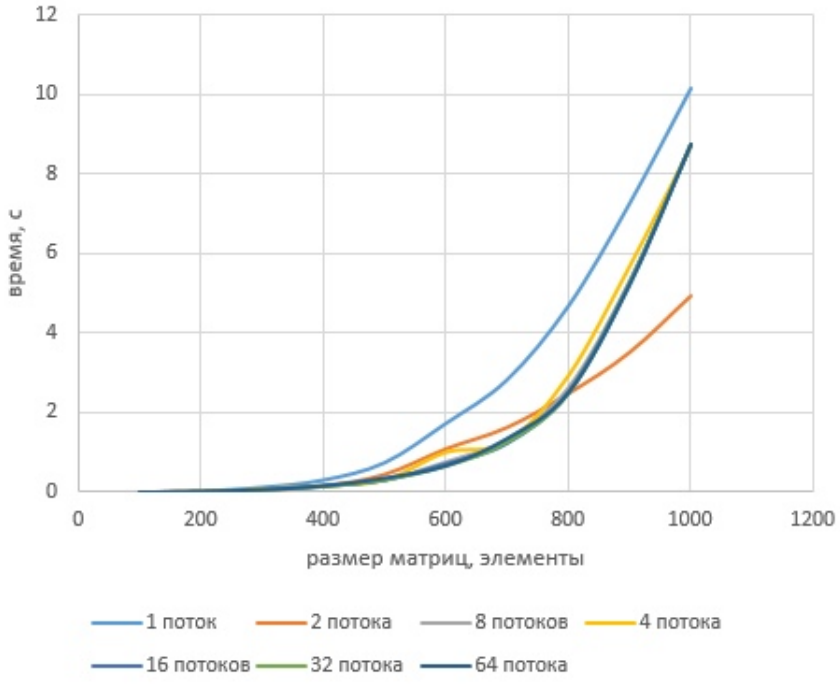

Рис. 3: Замеры времени

На графике рис. 3 видно, что на 2ухпоточный вариант показывает наилучший результат, обгоняя в 2 раза однопоточную реализацию и на 40% реализации на 4-х и более потоках при размерах матриц от 900x900 и более. Из этого можно сделать вывод, что наиболее эффективно количество потоков равное количеству физических ядер, так как при этом затраты на смену контекстов не снижают производительность.

Заключение

В рамках лабораторной работы было изучено понятие параллельных вычислений. Была реализована параллельная версия алгоритма Винограда и произведены замеры времени ее работы с различным количеством потоков. На основании этого произведено сравнение эффективности параллельной и последовательной версий алгоритма. Наиболее эффективной оказалась параллельная реализация алгоритма Винограда при количестве потоков равном количеству физических ядер процессора, имеющая выйгрыш в 2 раза по скорости по сравнению с последовательной реализации (т.е. в количество физических ядер).

Список литературы

[1] Введение в технологии параллельного программирования // INTEL: сайт. URL: https://software.intel.com/ru-ru/articles/writing-parallel-programs-a-multi-language-tutorial-introduction/ (дата обращения: 14.10.2019)