Министерство науки и высшего образования Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования «Московский государственный технический университет имени Н. Э. Баумана (национальный исследовательский университет)» (МГТУ им. Н. Э. Баумана)

ОТЧЕТ

По лабораторной работе №4

По курсу: «Анализ алгоритмов»

Тема: «Параллельная реализация алгоритма Копперсмита — Винограда»

Студент: Козаченко А. А.

Группа: ИУ7-54Б

Преподаватели: Волкова Л. Л.,
Строганов Ю. В.

Москва, 2020

Оглавление

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 [1];
- с программной точки зрения аппаратная поддержка многопоточности более трудоемка для программного обеспечения [2];
- 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 [3].

Несмотря на существующие недостатки, многопоточная парадигма имеет огромный потенциал, поэтому данная лабораторная работа будет посвящена распараллеливанию реализованного ранее алгоритма Винограда для умножения матриц.

Задачи работы

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 задачи

Пусть даны две прямоугольные матрицы

$$A_{lm} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{l1} & a_{l2} & \dots & a_{lm} \end{pmatrix}, \quad B_{mn} = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{pmatrix}, \tag{1.1}$$

тогда матрица C

$$C_{ln} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{l1} & c_{l2} & \dots & c_{ln} \end{pmatrix}, \tag{1.2}$$

где

$$c_{ij} = \sum_{r=1}^{m} a_{ir} b_{rj} \quad (i = \overline{1,l}; j = \overline{1,n}) \tag{1.3}$$

будет называться произведением матриц A и B [4].

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 [5].

Рассмотрим два вектора $V = (v_1, v_2, v_3, v_4)$ и $W = (w_1, w_2, w_3, w_4)$. Их скалярное произведение равно: $V \cdot W = v_1w_1 + v_2w_2 + v_3w_3 + v_4w_4$,, что эквивалентно

$$V \cdot W = (v_1+w_2)(v_2+w_1)+(v_3+w_4)(v_4+w_3)-v_1v_2-v_3v_4-w_1w_2-w_3w_4. \tag{1.4}$$

Несмотря на то, что второе выражение требует вычисления большего количества операций, чем стандартный алгоритм: вместо четырех умно-

жений - 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 [6].

В данной лабораторной работе стоит задача распараллеливания алгоритма Винограда. Так как каждый элемент матрицы C вычисляется независимо от других и матрицы A и B не изменяются, то для того, чтобы вычислить произведение параллельно, достаточно просто указать, какие элементы C какому потоку вычислять.

Вывод

В лабораторной работе стоит задача реализации многопоточной версии алгоритма Копперсмита — Винограда. Необходимо сравнить её с однопоточной версией.

2 Конструкторская часть

2.1 Разработка алгоритмов

2.1.1 Алгоритм Копперсмита — Винограда

На рисунке 2.1 представлена схема улучшенного алгоритма Копперсмита — Винограда.

2.1.2 Параллельная реализация алгоритма Копперсмита — Винограда

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 2-х, 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.

Подсчёты для каждой строки (циклы i на рисунке 2.1) выполняются независимо, а значит их можно распараллелить, выполняя цикл не от 0 до условного `R`, а поделив `R` на `n_threads` частей. Схема распараллеливания цикла приведена на рисунке 2.2

Вывод

На основе теоретических данных, полученных из аналитического раздела, была построена схема алгоритма Копперсмита — Винограда и приведены способы его распараллелить.

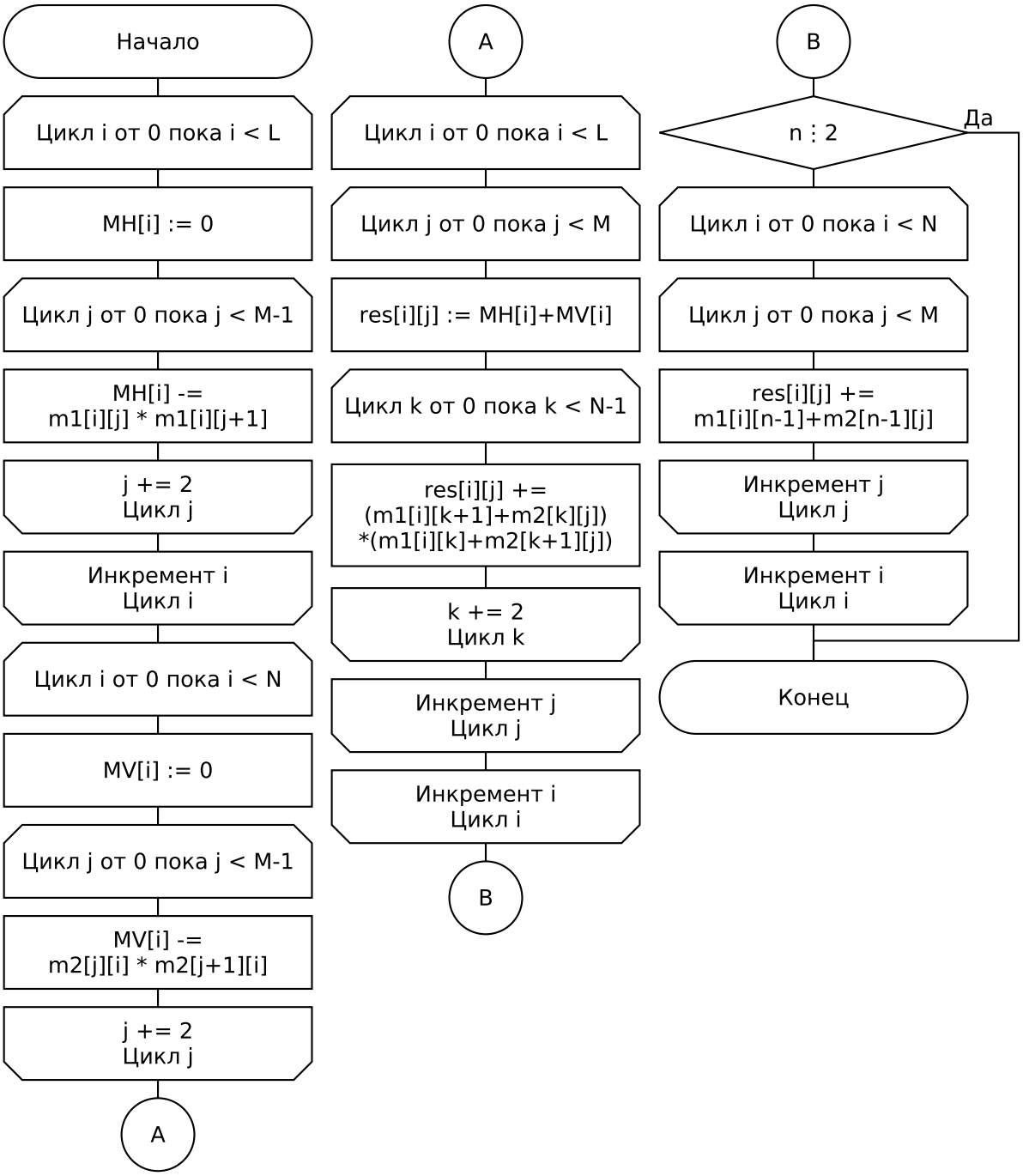

Рис. 2.1: Схема улучшенного алгоритма Копперсмита — Винограда

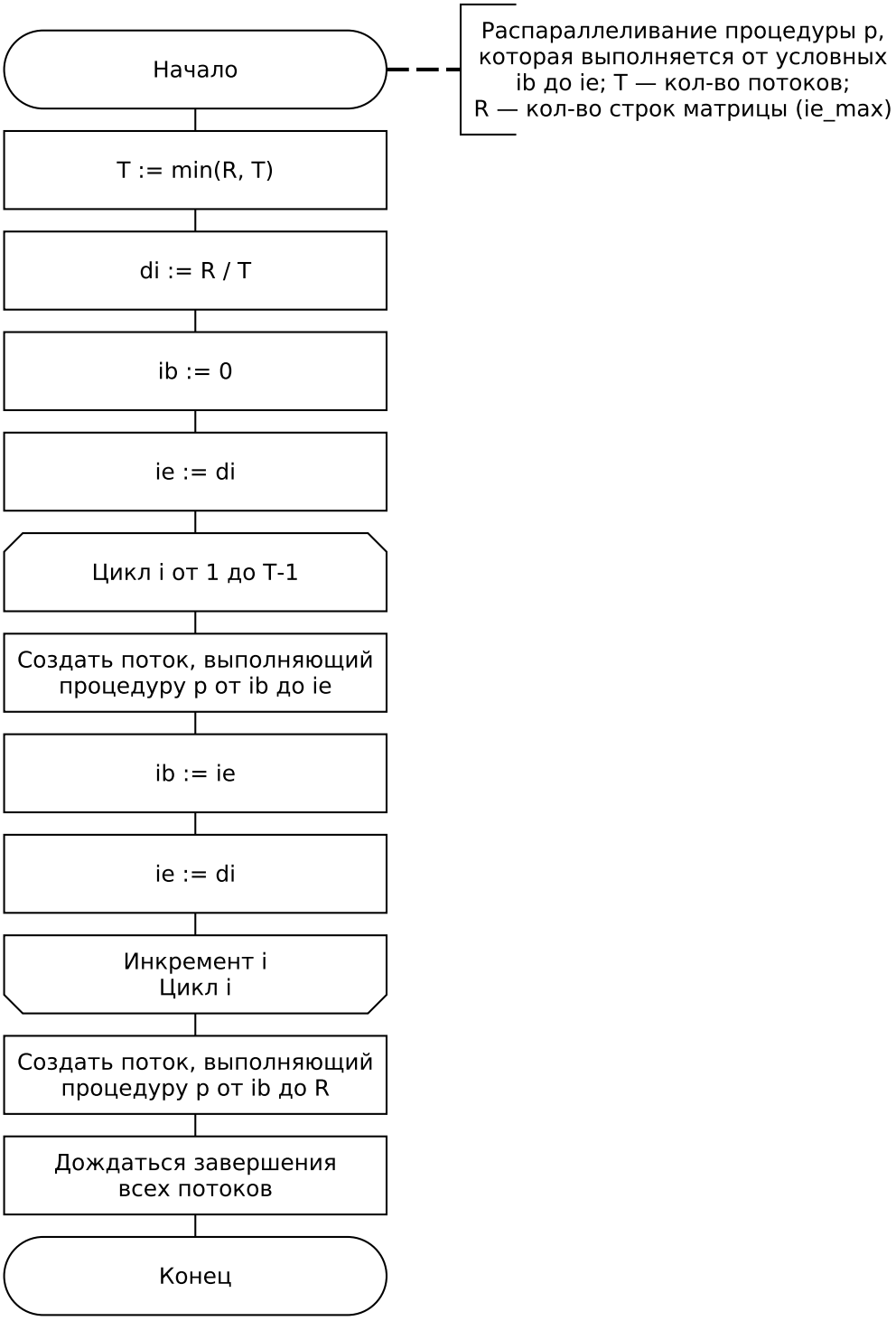

Рис. 2.2: Схема распараллеливания процедуры p

3 Технологическая часть

В данном разделе приведены средства реализации и листинг кода.

3.1 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран высокопроизводительный язык C++ [7], так как он предоставляет широкие возможности для эффективной реализации алгоритмов. Для распараллеливания алгоритма используется `std::thread` из стандартной библиотеки.

Для генерации псевдослучайных чисел использована функция из листинга 3.3.

Для замера времени работы алгоритма использованы точнейшие функции библиотеки `std::chrono`

3.2 Листинг кода

В листингах 3.1 и 3.2 приведены однопоточная и многопоточная реализации алгоритмов Копперсмита — Винограда. Функция замера времени работы алгоритма приведена в листинге 3.4.

```
matrix_t create_matrix(size_t rows, size_t cols) {
	return matrix_t(rows, row_t(cols, 0));
}

inline namespace detail {

row_t negative_row_products(const matrix_t& matrix, size_t rows, size_t cols) {
	auto result = row_t(rows, 0);
	for (size_t i = 0; i != rows; ++i) {
		for (size_t j = 0; j < cols - 1; j += 2) {
			result[i] -= matrix[i][j] * matrix[i][j + 1];
		}
	}

	return result;
}
```

```
row_t negative_col_products(const matrix_t& matrix, size_t rows, size_t cols) {
	auto result = row_t(rows, 0.);
	for (size_t j = 0; j < cols - 1; j += 2) {
		for (size_t i = 0; i != rows; ++i) {
			result[i] -= matrix[j][i] * matrix[j + 1][i];
		}
	}

	return result;
}

} // namespace detail

matrix_t coppersmith_winograd_product(const matrix_t& m1, const matrix_t& m2) {
	const size_t l = m1.size();
	const size_t m = m2.size();

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

	const size_t n = m2[0].size();

	const auto mh = negative_row_products(m1, l, m);
	const auto mv = negative_col_products(m2, n, m);

	auto result = create_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 & 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 3.1: Последовательный алгоритм Копперсмита — Винограда

```
inline namespace detail {

void parallelize(const std::function<void(size_t, size_t)>& function, size_t
    thread_count, size_t rows) {
        if (rows < thread_count) {
                thread_count = rows;
        }

        std::vector<std::thread> threads(thread_count);
        const size_t delta_i = rows / thread_count;
        size_t i_begin = 0;
        size_t i_end = delta_i;
        for (size_t i = 0; i + 1 < thread_count; ++i, i_begin = i_end, i_end += delta_i) {
                threads[i] = std::thread(function, i_begin, i_end);
        }

        threads.back() = std::thread(function, i_begin, rows);

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

row_t negative_row_products(const matrix_t& matrix, size_t rows, size_t cols, size_t
    thread_count) {
        auto result = row_t(rows, 0);
        auto lambda = [&](size_t i_begin, size_t i_end) {
                for (size_t i = i_begin; i < i_end; ++i) {
                        for (size_t j = 0; j < cols - 1; j += 2) {
                                result[i] -= matrix[i][j] * matrix[i][j + 1];
                        }
                }
        };

        parallelize(lambda, thread_count, rows);
        return result;
}

row_t negative_col_products(const matrix_t& matrix, size_t rows, size_t cols, size_t
    thread_count) {
        auto result = row_t(rows, 0.);
        auto lambda = [&](size_t i_begin, size_t i_end) {
                for (size_t i = i_begin; i < i_end; i++) {
```

```
                        for (size_t j = 0; j < cols - 1; j += 2) {
                                result[i] -= matrix[j][i] * matrix[j + 1][i];
                        }
                }
        };

        parallelize(lambda, thread_count, rows);
        return result;
}

} // namespace detail

matrix_t coppersmith_winograd_product(const matrix_t& m1, const matrix_t& m2, size_t
    thread_count) {
        const size_t l = m1.size();
        const size_t m = m2.size();

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

        const size_t n = m2[0].size();

        row_t mh, mv;
        if (thread_count > 2) {
                std::thread mh_thread([&]() { mh = negative_row_products(m1, l, m); });
                std::thread mv_thread([&]() { mv = negative_col_products(m2, n, m); });
                mh_thread.join();
                mv_thread.join();
        } else {
                mh = negative_row_products(m1, l, m);
                mv = negative_col_products(m2, n, m);
        }

        auto result = create_matrix(l, n);
        auto lambda = [&](size_t i_begin, size_t i_end) {
                for (size_t i = i_begin; i < i_end; ++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]);
                                }
                        }
                }
        };

        parallelize(lambda, thread_count, l);
```

```
        if (n & 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 3.2: Параллельный алгоритм Копперсмита — Винограда

```
int dont_try_to_guess() {
        static thread_local std::mt19937 generator(std::random_device{}());
        static thread_local std::uniform_int_distribution<int> distribution(-1000, 1000);
        return distribution(generator);
}
```

Листинг 3.3: Продвинутый генератор псевдослучайных чисел

```
double count_time(parallel_product_func_type parallel_product_func, const matrix_t& m1,
    const matrix_t& m2, size_t thread_count) {
        constexpr size_t N = 5;

        const auto start = std::chrono::high_resolution_clock::now();
        for (int i = 0; i != N; ++i) {
                parallel_product_func(m1, m2, thread_count);
        }
        const auto end = std::chrono::high_resolution_clock::now();

        return
            static_cast<double>(std::chrono::duration_cast<std::chrono::nanoseconds>(end
            - start).count()) / 1.0e9 / N;
}
```

Листинг 3.4: Функция замера времени работы алгоритмов

3.3 Тестирование функций

В таблице 3.1 приведены тесты для функций, реализующих однопоточный и многопоточный алгоритмы Копперсмита — Винограда. Тесты пройдены успешно.

Матрица 1	Матрица 2	Ожидаемый результат
$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 6 & 12 & 18 \\ 6 & 12 & 18 \\ 6 & 12 & 18 \end{pmatrix}$
$\begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix}$	$\begin{pmatrix} 3 & 6 \\ 3 & 6 \end{pmatrix}$
$\begin{pmatrix} 2 \end{pmatrix}$	$\begin{pmatrix} 2 \end{pmatrix}$	$\begin{pmatrix} 4 \end{pmatrix}$
$\begin{pmatrix} 1 & -2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} -1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 0 & 4 & 6 \\ 4 & 12 & 18 \\ 4 & 12 & 18 \end{pmatrix}$
$\begin{pmatrix} 1 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 \end{pmatrix}$	Не могут быть перемножены

Таблица 3.1: Тестирование функций

Вывод

Правильный выбор инструментов разработки позволил эффективно реализовать алгоритмы, настроить модульное тестирование и выполнить исследовательский раздел лабораторной работы.

4 Исследовательская часть

4.1 Технические характеристики

- Операционная система: Ubuntu 19.10 64-bit.
- Память: 3,8 GiB.
- Процессор: Intel® Core™ i3-6006U CPU @ 2.00GHz с 2 физическими и 4 логическими ядрами

4.2 Время выполнения алгоритмов

Алгоритмы тестировались с помощью функции замера процессорного времени `std::chrono::high_resolution_clock::now()`. Чтобы уменьшить случайные отклонения в измерениях, время считалось среднее для 5 запусков функций.

Результаты замеров приведены в таблицах 4.1 и 4.2. На рисунках 4.1 и 4.2 приведены графики зависимостей времени работы алгоритмов от размеров матриц на различном количестве потоков.

	Время при кол-ве потоков, мс						
Размер	**1**	**2**	**4**	**8**	**16**	**32**	**64**
100	0.0198	0.0326	0.0480	0.0600	0.0579	0.0521	0.0739
200	0.5801	0.0808	0.0768	0.0784	0.0805	0.0793	0.2336
300	0.6798	0.2791	0.2673	0.2661	0.2626	0.2608	0.2959
400	1.3757	0.6953	0.6106	0.6178	0.6166	0.6377	0.6393
500	2.7741	1.4006	1.2142	1.2408	1.2327	1.2748	1.3683
600	4.9342	2.5748	2.1330	2.1516	2.1663	2.2405	2.2287
700	8.1183	4.3020	3.5571	3.5634	3.6199	3.7177	3.9012
800	12.4279	6.5367	5.4672	5.5282	5.5552	5.5068	5.7741
900	18.2714	9.4057	8.0896	8.2161	8.2604	8.2606	8.1459
1000	25.8421	12.9612	11.1452	11.4822	11.6378	11.6719	12.0267

Таблица 4.1: Время работы алгоритмов при чётных размерах матриц

Размер	Время при кол-ве потоков, мс						
	1	**2**	**4**	**8**	**16**	**32**	**64**
101	0.0204	0.0104	0.0100	0.0105	0.0106	0.0107	0.0146
201	0.1712	0.0858	0.0779	0.0787	0.0800	0.0797	0.0803
301	0.5697	0.2863	0.2616	0.2644	0.2716	0.2688	0.3011
401	1.3884	0.6974	0.6224	0.6394	0.6297	0.6557	0.6459
501	2.7986	1.4080	1.2261	1.2469	1.2381	1.2791	1.4008
601	4.9809	2.5786	2.1551	2.1444	2.1755	2.2501	2.2561
701	8.1308	4.3173	3.5954	3.5647	3.6465	3.7569	3.9364
801	12.5641	6.6001	5.5447	5.5310	5.5825	5.5924	5.7739
901	18.4556	9.5447	8.1706	8.2530	8.3038	8.2822	8.2451
1001	26.0820	13.0804	11.3136	11.5817	11.7595	11.7445	12.1095

Таблица 4.2: Время работы алгоритмов при нечётных размерах матриц

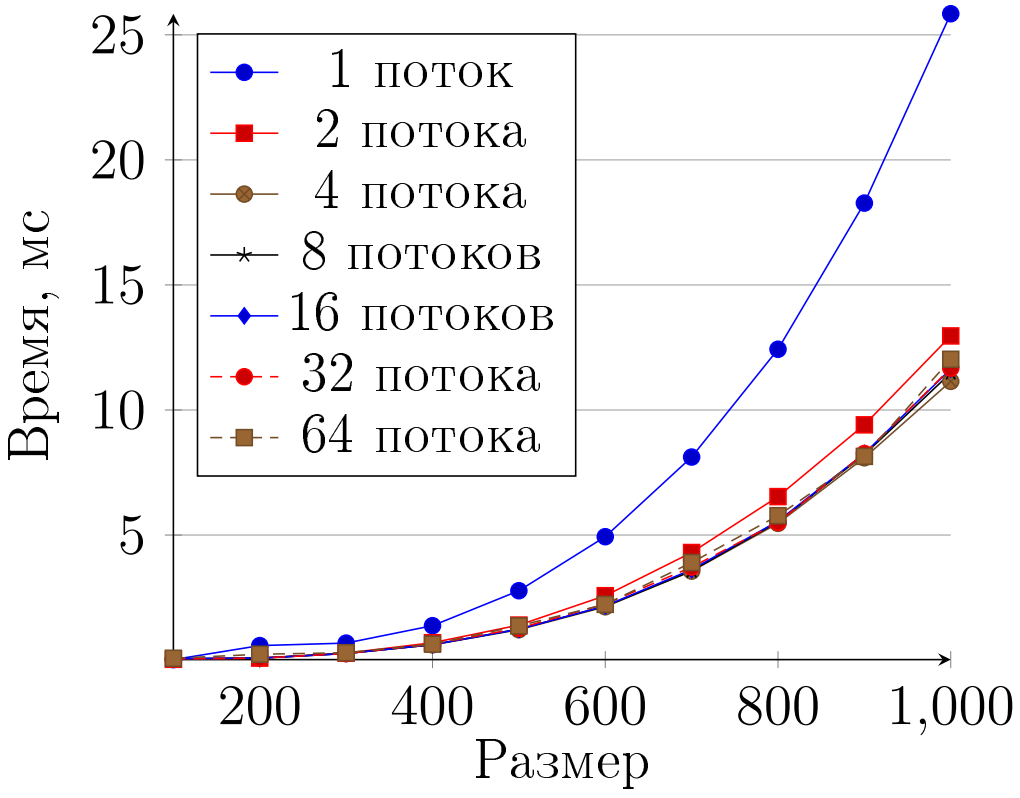

Рис. 4.1: Зависимость времени работы алгоритма от размера квадратной матрицы (чётного)

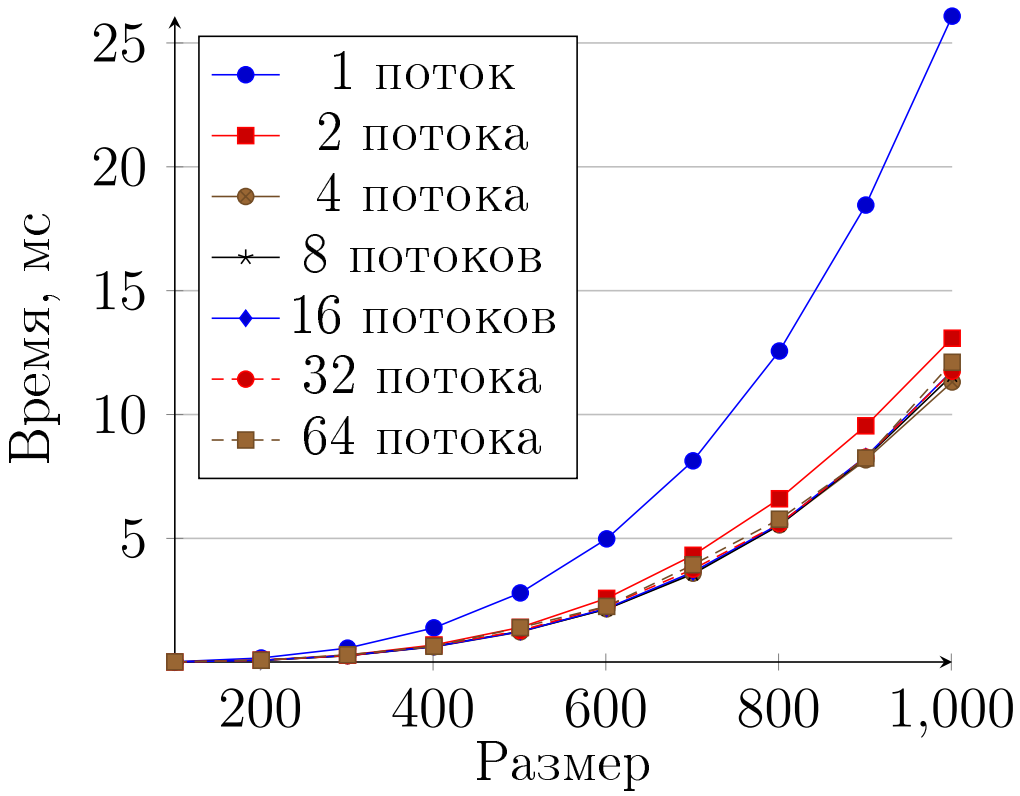

Рис. 4.2: Зависимость времени работы алгоритма от размера квадратной матрицы (нечётного)

Вывод

Наилучшее время алгоритм показал на 4 потоках, соответствующих количеству логических ядер, уменьшив время однопоточной реализации в 2,3 раза. Большее количество потоков в итоге немногим замедляет время необходимостью переключения контекста.

Заключение

В рамках лабораторной работы было изучено понятие параллельных вычислений. Реализована параллельная версия алгоритма Винограда и произведены замеры времени её 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 логических ядер процессора, превосходящая время однопоточной реализации на 57%.

Литература

[1] Mario Nemirovsky D. M. T. Multithreading Architecture // Morgan and Claypool Publishers. 2013.

[2] Olukotun K. Chip Multiprocessor Architecture — Techniques to Improve Throughput and Latency // Morgan and Claypool Publishers. 2007. p. 154.

[3] Введение в технологии параллельного программирования. URL: https://software.intel.com/ru-ru/articles/writing-parallel-programs-a-multi-language-tutorial-introduction/ (дата обращения: 24.11.2019).

[4] Group-theoretic Algorithms for Matrix Multiplication / H. Cohn, R. Kleinberg, B. Szegedy et al. // Proceedings of the 46th Annual Symposium on Foundations of Computer Science. 2005. October. P. 379–388.

[5] Coppersmith D., Winograd S. Matrix multiplication via arithmetic progressions // Journal of Symbolic Computation. 1990. no. 9. P. 251–280.

[6] Погорелов Дмитрий Александрович. Оптимизация классического алгоритма Винограда для перемножения матриц // Журнал №1. 2019. Т. 49.

[7] Working Draft, Standard for ProgrammingLanguage C++. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4713.pdf. 2017.