Государственное образовательное учреждение высшего профессионального образования

**«Московский государственный технический университет имени Н. Э. Баумана»**

**(МГТУ им. Н.Э. Баумана)**

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЁТ ПО АНАЛИЗУ АЛГОРИТМОВ

к лабораторной работе №5 на тему:

«Распараллеливание вычислений»

Студент: Марабян К. ИУ7-55

Москва 2018

**Содержание**

[**Введение**](#_y1ea5kse8d8s) **2**

[**1 Аналитическая часть**](#_n91k6n79dzut) **3**

[**1.1 Постановка задачи**](#_3gqvaou3e8f5) **3**

[**1.2 Описание алгоритмов**](#_brhollpqxi2m)4

[**2 Конструкторский раздел**](#_5g4cmomvb0gf) **5**

[**2.1 Разработка алгоритмов**](#_codkpmvflfqb) **5**

[**3 Технологический раздел**](#_dgwh654ihhkp)6

[**3.1 Минимальные требования к программному обеспечению**](#_mncwt7me2c7x)6

[**3.2 Средства реализации**](#_80qaa7iwbkbc)6

[**3.3 Листинг кода**](#_l3quu9c8lbwe)8

[**4 Экспериментальная часть**](#_hlnbv34b5bsg) **12**

[**4.1**](#_445uz6nu0e23) **Постановка эксперимента 1**2

**Вывод 13**

[**Заключение**](#_1hl6wz3mwv0x)14

##### 

##### Введение

Процесс с двумя потоками выполнения на одном процессоре. Поток выполнения — наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы. Реализация потоков выполнения и процессов в разных операционных системах отличается друг от друга, но в большинстве случаев поток выполнения находится внутри процесса. Несколько потоков выполнения могут существовать в рамках одного и того же процесса и совместно использовать ресурсы, такие как память, тогда как процессы не разделяют этих ресурсов. В частности, потоки выполнения разделяют инструкции процесса (его код) и его контекст (значения переменных, которые они имеют в любой момент времени). В качестве аналогии потоки выполнения процесса можно уподобить нескольким вместе работающим поварам. Все они готовят одно блюдо, читают одну и ту же кулинарную книгу с одним и тем же рецептом и следуют его указаниям, причём не обязательно все они читают на одной и той же странице.

На одном процессоре многопоточность обычно происходит путём временного мультиплексирования (как и в случае многозадачности): процессор переключается между разными потоками выполнения. Это переключение контекста обычно происходит достаточно часто, чтобы пользователь воспринимал выполнение потоков или задач как одновременное. В многопроцессорных и многоядерных системах потоки или задачи могут реально выполняться одновременно, при этом каждый процессор или ядро обрабатывает отдельный поток или задачу.

Потоки возникли в операционных системах как средство распараллеливания вычислений.

Параллельное выполнение нескольких работ в рамках одного интерактивного приложения повышает эффективность работы пользователя. Так, при работе с текстовым редактором желательно иметь возможность совмещать набор нового текста с такими продолжительными по времени операциями, как переформатирование значительной части текста, печать документа или его сохранение на локальном или удаленном диске. Еще одним примером необходимости распараллеливания является сетевой сервер баз данных. В этом случае параллелизм желателен как для обслуживания различных запросов к базе данных, так и для более быстрого выполнения отдельного запроса за счет одновременного просмотра различных записей базы. Именно для этих целей современные ОС предлагают механизм многопоточной обработки (multithreading). Понятию «поток» соответствует последовательный переход процессора от одной команды программы к другой. ОС распределяет процессорное время между потоками. Процессу ОС назначает адресное пространство и набор ресурсов, которые совместно используются всеми его потоками.

Создание потоков требует от ОС меньших накладных расходов, чем процессов. В отличие от процессов, которые принадлежат разным, вообще говоря, конкурирующим приложениям, все потоки одного процесса всегда принадлежат одному приложению, поэтому ОС изолирует потоки в гораздо меньшей степени, нежели процессы в традиционной мультипрограммной системе. Все потоки одного процесса используют общие файлы, таймеры, устройства, одну и ту же область оперативной памяти, одно и то же адресное пространство. Это означает, что они разделяют одни и те же глобальные переменные. Поскольку каждый поток может иметь доступ к любому виртуальному адресу процесса, один поток может использовать стек другого потока. Между потоками одного процесса нет полной защиты, потому что, во-первых, это невозможно, а во-вторых, не нужно. Чтобы организовать взаимодействие и обмен данными, потокам вовсе не требуется обращаться к ОС, им достаточно использовать общую память — один поток записывает данные, а другой читает их. С другой стороны, потоки разных процессов по-прежнему хорошо защищены друг от друга.

##### 1 Аналитическая часть

В данном разделе представлены теоретическое описание алгоритмов и указание области их применения.

##### 1.1 Постановка задачи

1) Научиться писать многопоточные программы;

2) применить полученные знания на практике, а именно переписать алгоритм Копперсмита — Винограда в несколько потоков;

3) провести замеры скорости работы однопоточной и многопоточной реализаций и проанализировать полученные результаты.

##### 1.2 Описание алгоритмов

В данном разделе приведено описание обычной реализации алгоритма Копперсмита — Винограда.

Алгоритм умножения матриц Винограда

Если посмотреть на результат умножения двух матриц, то видно, что каждый элемент в нем представляет собой скалярное произведение соответствующих строки и столбца исходных матриц. Можно заметить также, что 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.

Это равенство можно переписать в виде:

V • W = (v1 + w2)(v2 + w1) + (v3 + w4)(v4 + w3) - v1v2 - v3v4 - w1w2 - w3w4.

Кажется, что второе выражение задает больше работы, чем первое: вместо четырех умножений мы насчитываем их шесть, а вместо трех сложений - десять. Менее очевидно, что 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. На практике это означает, что над предварительно обработанными элементами нам придется выполнять лишь первые два умножения и последующие пять сложений, а также дополнительно два сложения.

Улучшения алгоритма.

Возможны улучшения алгоритма с целью снизить его трудоемкость.

В 2010 Эндрю Стотерс усовершенствовал алгоритм до O(n^{2.374}).

В 2011 году Вирджиния Вильямс усовершенствовала алгоритм ещё раз —O(n^{2.3728642}).

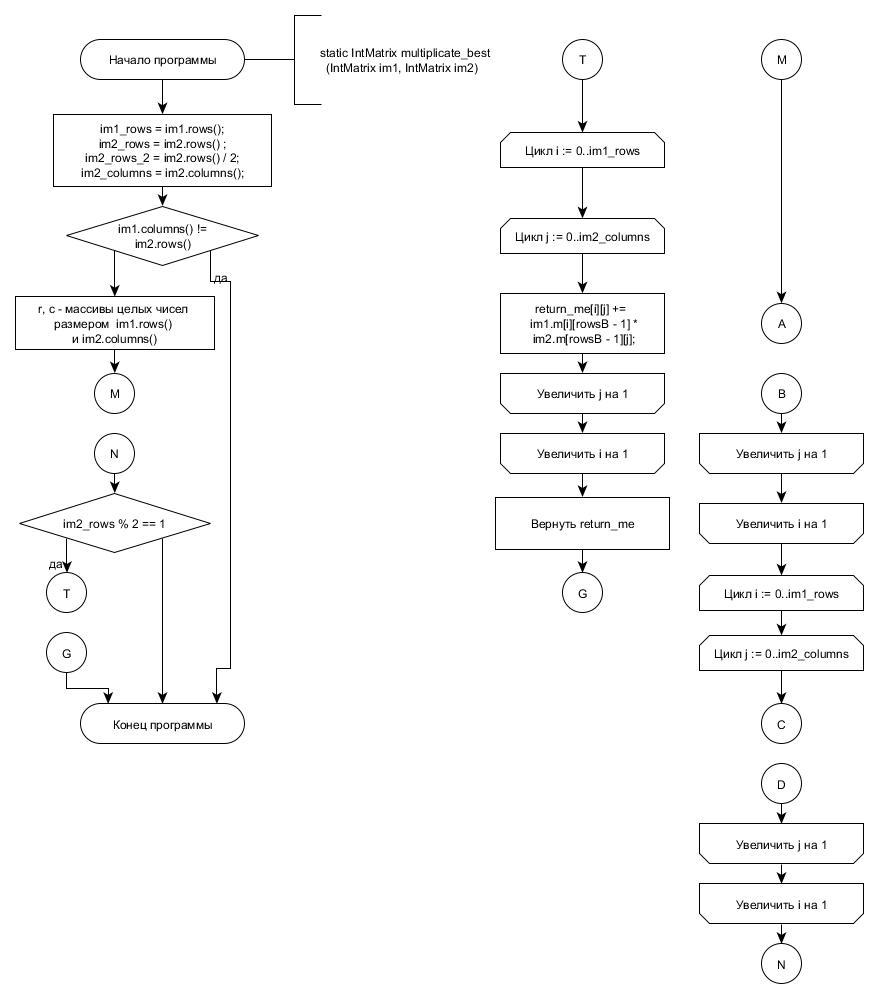
В 2014 году Франсуа Ле Галль упростил метод Уильямса и получил новую улучшенную оценку O(n^{2.3728639}).

##### 2 Конструкторский раздел

В этом разделе приведены схемы алгоритмов.

##### 2.1 Разработка алгоритмов

Ниже приложены блок-схемы алгоритмов решения поставленных задач.

Рисунок 2.1 — Алгоритм Винограда(часть 1/2).

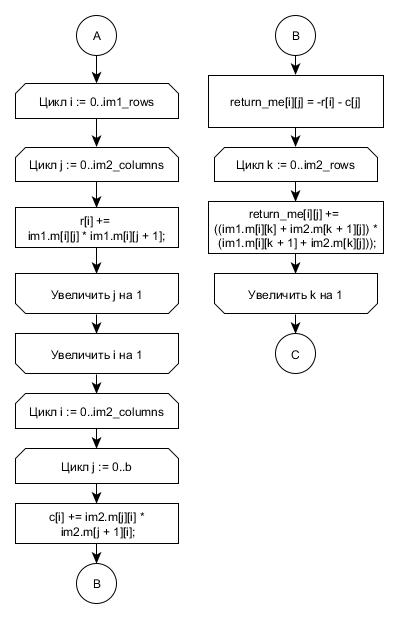


Рисунок 2.2 — Алгоритм Винограда (часть 2/2).

##### 3 Технологический раздел

Данный раздел содержит указание использованного языка программирования, а также листинг кода.

##### 3.1 Минимальные требования к программному обеспечению

Операционные системы: MacOS

Объем жесткого диска: 10 Мб.

Оперативная память: 1024 MB ОЗУ.

##### 3.2 Средства реализации

Для реализации программ мною был выбрал язык программирования - C++.

Среда разработки - Qt.

Для работы с потоками использовались библиотеки thread и mutex. Однако поскольку потоки не использовали разделяемую память, то использование семафоров не понадобилось, если не считать их использование в выводе данных для отладки. Для реализаций матриц использовалась библиотека vector.

Отключение оптимизации производилось с помощью директивы препроцессора

#pragma optimize("", off).

##### 3.3 Листинг кода

Функция замера процессороного времени:

unsigned long long tick(void)

{

unsigned long long d;

\_\_asm\_\_ \_\_volatile\_\_ ("rdtsc" : "=A" (d));

return d;

}

В листинге представлена реализация однопоточной реализации алгоритма.

#pragma optimize("", off)

static IntMatrix multiplicate\_best(IntMatrix im1, IntMatrix im2) {

if (im2.rows() != im1.columns())

return (IntMatrix());

IntMatrix return\_me(im1.rows(), im2.columns());

std::vector<int> r(im1.rows());

std::vector<int> c(im2.columns());

size\_t im1\_rows = im1.rows();

size\_t im2\_rows\_2 = im2.rows() / 2;

size\_t im2\_columns = im2.columns();

for (size\_t i = 0; i < im1\_rows; i++) {

for (size\_t j = 0; j < im2\_rows\_2; j++) {

r[i] += im1.m[i][j] \* im1.m[i][j + 1];

}

}

for (size\_t i = 0; i < im2\_columns; i++) {

for (size\_t j = 0; j < im2\_rows\_2; j++) {

c[i] += im2.m[j][i] \* im2.m[j + 1][i];

}

}

for (size\_t i = 0; i < im1\_rows; i++) {

return\_me.m[i].clear();

for (size\_t j = 0; j < im2\_columns; j++) {

return\_me.m[i].push\_back(0);

return\_me.m[i][j] = -r[i] - c[j];

for (size\_t k = 0; k < im2\_rows\_2; k++) {

return\_me.m[i][j] += ((im1.m[i][k] + im2.m[k + 1][j]) \*

(im1.m[i][k + 1] + im2.m[k][j]));

}

}

}

size\_t im2\_rows = im2.rows();

if (im2\_rows % 2) {

for (size\_t i = 0; i < im1\_rows; i++) {

for (size\_t j = 0; j < im2\_columns; j++) {

im2\_columns \*

return\_me.m[i][j] += im1.m[i][im2\_rows - 1] \* im2.m[im2\_rows - 1][j];

}

}

}

return return\_me;

}

Для реализации многопоточной реалиазация функция была разбита на 3 части. В листинге показана первая часть, отвечающая за создание вспомогательной строки. Вторая функция создает вспомогательный столбец. Непосредственно умножение матриц происходит в крайней функции.

#pragma optimize("", off)

void work\_1\_3(IntMatrix im1,

size\_t im1\_rows\_begin,

size\_t im1\_rows,

size\_t im2\_rows\_2,

std::vector<int> &row){

size\_t i\_begin = im1\_rows\_begin;

size\_t i\_size = im1\_rows;

size\_t j\_size = im2\_rows\_2;

for (size\_t i = i\_begin;u i < i\_size; i++) {

for (size\_t j = 0; j < j\_size; j++) {

row[i] += im1.m[i][j] \* im1.m[i][j + 1];

}

}

}

#pragma optimize("", off)

void work\_2\_3(IntMatrix im2,

size\_t im2\_columns\_begin,

size\_t im2\_columns,

size\_t im2\_rows\_2,

std::vector<int> &column){

size\_t i\_begin = im2\_columns\_begin;

size\_t i\_size = im2\_columns;

size\_t j\_size = im2\_rows\_2;

for (size\_t i = i\_begin; i < i\_size; i++) {

for (size\_t j = 0; j < j\_size; j++) {

column[i] += im2.m[j][i] \* im2.m[j + 1][i];

}

}

}

#pragma optimize("", off)

void work\_3\_3(IntMatrix im1, IntMatrix im2,

std::vector<int> c,

std::vector<int> r,

size\_t im1\_rows\_begin,

size\_t im1\_rows,

size\_t im2\_columns,

size\_t im2\_rows\_2,

IntMatrix &return\_me)

{

size\_t im2\_rows = im2.rows();

size\_t i\_begin = im1\_rows\_begin;

size\_t i\_size = im1\_rows;

size\_t j\_size = im2\_columns;

size\_t k\_size = im2\_rows\_2;

for (size\_t i = i\_begin; i < i\_size; i++) {

return\_me.m[i].clear();

for (size\_t j = 0; j < j\_size; j++) {

return\_me.m[i].push\_back(0);

return\_me.m[i][j] = -r[i] - c[j];

for (size\_t k = 0; k < k\_size; k++) {

return\_me.m[i][j] += (

(im1.m[i][k] + im2.m[k + 1][j]) \*

(im1.m[i][k + 1] + im2.m[k][j])

);

}

if (im2\_rows % 2) {

return\_me.m[i][j] += im1.m[i][im2\_rows - 1] \*

im2.m[im2\_rows - 1][j];

}

}

}

}

Далее представлена реализация двупоточной версии алгоритма. 4-ех, 6-и, 8-и поточные реализации сделаны аналогичным образом.

#pragma optimize("", off)

IntMatrix multi2(IntMatrix im1, IntMatrix im2) {

if (im2.rows() != im1.columns())

return (IntMatrix());

IntMatrix return\_me(im1.rows(), im2.columns());

std::vector<int> r(im1.rows());

std::vector<int> c(im2.columns());

size\_t im1\_rows = im1.rows();

size\_t im2\_rows\_2 = im2.rows() / 2;

size\_t im2\_columns = im2.columns();

thread thread1\_1(work\_1\_3,

im1,

0, im1\_rows,

im2\_rows\_2,

ref(r));

thread thread2\_1(work\_2\_3, im2,

0, im2\_columns,

im2\_rows\_2,

ref(c));

thread1\_1.join();

thread2\_1.join();

thread thread3\_1(work\_3\_3, im1, im2

c, r, 0, im1\_rows / 2, im2\_columns, im2\_rows\_2,

ref(return\_me));

thread thread3\_2(work\_3\_3, im1, im2,

c, r, im1\_rows / 2, im1\_rows, im2\_columns, im2\_rows\_2,

ref(return\_me)));

thread3\_1.join();

thread3\_2.join();

return return\_me;

}

##### 

##### 4 Экспериментальная часть

В данном разделе будет сравнительный анализ реализаций алгоритма Винограда на основе экспериментальных данных. Во время проведения замеров использовался ноутбук c 2 физическими ядрами и 4 логическими процессорами.

##### 4.1 Постановка эксперимента

Для экспериментов использовались массивы, размер которых варьируется от 100 до 500 с шагом 100.

Количество повторов каждого эксперимента = 100. Результат одного эксперимента рассчитывается как средний из результатов проведенных испытаний с одинаковыми входными данными. Результаты замеров времени сведены в таблице. Таблица состоит из 6 столбцов. Первый столбец представляет собой количество элементов. Второй - обычный оптимизированный алгоритм Винограда. Третий, четвертый, пятый и шестой - 2, 4, 6 и 8 поточные реализации. Те же результаты представлены на рис.

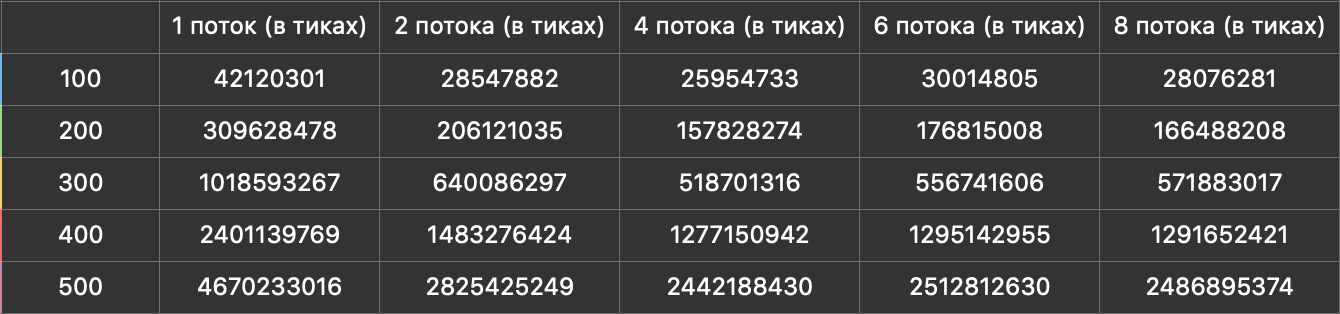


Рис. 1 – Таблица с результатами замеров.

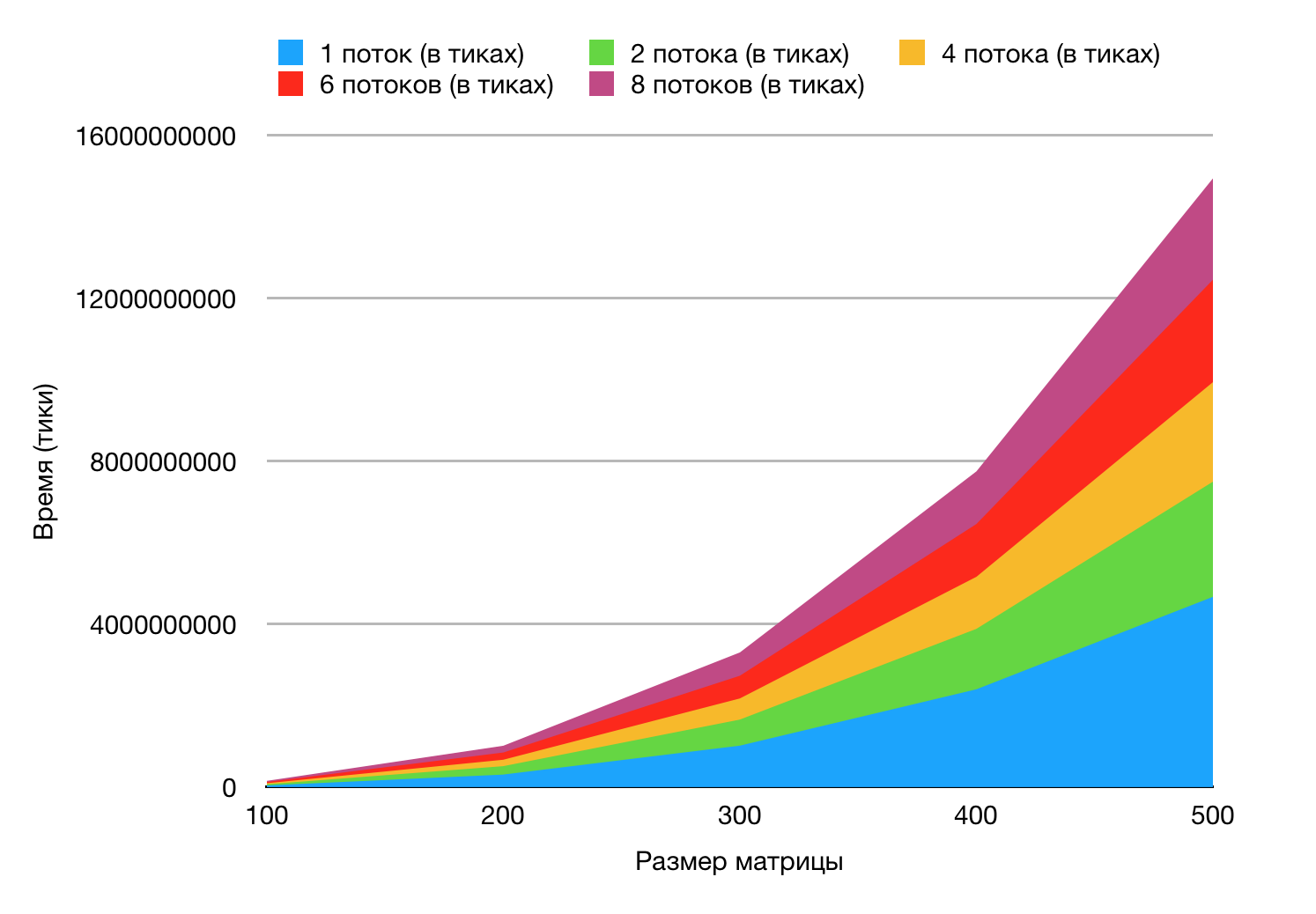


Рис. 2 – График с результатами замеров.

**Вывод**

Как и ожидалось, многопоточные программы работают быстрее однопоточных решений, причем разница достигает почти двух раз (зависит от нагруженности процессора. Чем больше нагрузка, тем меньше эффективность нескольких потоков, поскольку им приходится простаивать в очереди). Наиболее эффективным количеством потоков оказалось число 4, равное количеству логических процессоров. При дальнейшем увеличении количества потоков увеличения скорости нет, зато увеличивается затрачиваемое время, что связано с издержками ожидания процессорного времени для потоков.

##### 

##### Заключение

В ходе работы были изучена и реализована многопоточность. Поскольку было выбрано разделение алгоритма так, чтобы не было разделяемой памяти, использования семафоров не было. Был сделан вывод, что наиболее оптимальное количество потоков в программе должно соответствовать количеству логических процессоров используемого компьютера.