МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»
Кафедра математической кибернетики и компьютерных наук

СРАВНЕНИЕ БЫСТРОДЕЙСТВИЯ ПАРАЛЛЕЛЬНОЙ И ПОСЛЕДОВАТЕЛЬНОЙ РЕАЛИЗАЦИИ ИГРЫ «ЖИЗНЬ» НА ЯЗЫКЕ JAVA

АВТОРЕФЕРАТ БАКАЛАВРСКОЙ РАБОТЫ

студента 5 курса 551 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Андреева Артёма Юрьевича

Научный руководитель
к. ф.-м. н., доцент _____ А. С. Иванова

Заведующий кафедрой
к. ф.-м. н., доцент _____ А. С. Иванов

Саратов 2020

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В современных реалиях, задачи, с которыми должны справляться создаваемые приложения, требуют все больше вычислительной мощности. Однако, рост тактовой частоты процессоров уже не может компенсировать столь высокие потребности. Дальнейшее ее увеличение затруднено из-за конечности величины частоты в рамках заданной системы и интенсивного роста тепловыделения при возрастании тактовой частоты. Для того, чтобы обеспечить рост вычислительной мощности информационной системы, применяется архитектура параллельной обработки данных [1]. Это решение основано на использовании многоядерных процессоров или нескольких физических процессоров. Наиболее распространенной, для параллельной обработки данных, в сегменте домашних настольных персональных компьютеров, является архитектура на основе многоядерного процессора. Напротив, решение с использованием нескольких физических одноядерных или многоядерных процессоров, наиболее распространено в сегменте серверных вычислительных систем, т.к. являются более дорогими, требуют более мощного охлаждения и используют специализированные комплектующие (например, материнская плата со слотами для нескольких физических процессоров).

Для построения программ, которые могли бы эффективно использовать технологии параллелизма, требуются теоретические знания в области параллельного программирования. Однако, бонус от их применения, позволяет ускорить вычисление в несколько раз, что является ощутимым фактором.

Основной целью дипломной работы является сравнение быстродействия параллельной и не параллельной (последовательной) версии алгоритма игры «Жизнь», реализованной на языке Java.

Для выполнения цели работы были поставлены следующие задачи:

- изучить механизм многопоточных вычислений, реализующий модель параллелизма на уровне Java-приложения;
- ознакомиться с понятиями и определениями, применяемыми в многопоточном программировании;
- изучить API Java для работы в многопоточной среде;
- разработать приложение, реализующее алгоритм игры «Жизнь» с использованием параллельного и последовательного подходов;
- провести измерения времени выполнения программы для каждого из

реализованных подходов и сравнить полученные данные.

Работа написана на основе материалов следующих литературных источников: Шилдт Г. «Java 8. Полное руководство.», Хорстманн К., Корнелл Г. «Java. Том 1. Основы.» и «Java. Том 2. Расширенные средства программирования.», а также на основе различных интернет-источников.

Структура дипломной работы включает: введение, две главы, заключение, список использованных источников и приложение. Во введении рассматривается актуальность темы, ставятся цели и задачи работы. Первая глава посвящена теоретическим вопросам и содержит определения, принципы многопоточного программирования, описание API Java для работы с потоками. Во второй главе проводится анализ и сравнение быстродействия параллельного и последовательного подходов на примере приложения, реализующего алгоритм игры «Жизнь». В заключении приводятся выводы, полученные в ходе исследования.

1 Теоретические основы многопоточного программирования

Существуют различные подходы, позволяющие создавать приложения для параллельной обработки данных. Основные из них:

1. основанные на сетевом взаимодействии компьютеров (GRID);
2. основанные на межпроцессном взаимодействии в рамках одной физической машины (MPI, Java RMI, CORBA);
3. основанные на взаимодействии потоков в рамках одного процесса (OpenMP, Java Multithreading);
4. различные сочетания приведённых выше подходов.

Основанная на сетевом взаимодействии архитектура параллельной вычислительной системы представляет собой виртуальный компьютер, который, в свою очередь, состоит из нескольких физических компьютеров, объединенных по удаленным каналам связи (используя, например, TCP/IP протокол) в сеть с помощью специального программного обеспечения. Такая система может сравниться по производительности с супер-компьютером. Но в отличие от настоящего супер-компьютера, может состоять из различного количества обычных настольных компьютеров (или их вариаций), обладающих разными техническими характеристиками и под управлением различных операционных систем. Данное свойство позволяет легко подключать в грид новые машины, гибко масштабируя вычислительные мощности.

Межпроцессное параллельное программирование базируется на свойстве операционной системы выполнять несколько задач одновременно, т.е. многозадачности. Многозадачность позволяет запускать в операционной системе несколько процессов одновременно, которые в свою очередь могут состоять из нескольких потоков, выполняющихся параллельно. Истинная многозадачность и параллельность выполнения возможна только в распределенных вычислительных системах либо системах, построенных на основе нескольких физических процессоров [2]. Большинство пользовательских информационных систем являются псевдо-многозадачными. В основе своей работы, они используют подход управления временем выполнения центрального процессора, выделяя квант процессорного времени тому или иному процессу, в зависимости от совокупности различных факторов. Такими факторами могут быть, например, приоритет процесса, загруженность центрального процессора и др. [3, 4].

Многопоточный подход при создании приложений позволяет довольно легко организовать координацию между потоками, при этом затрачивая меньше ресурсов, чем при межпроцессном и сетевом подходах [5]. Достигаются такие показатели за счет того, что все потоки одного процесса взаимодействуют с общим адресным пространством оперативной памяти, выделенным этому процессу операционной системой. В таком случае каждый поток может напрямую обратиться к общему пространству памяти, чтобы считать или записать информацию. Следовательно, нет необходимости в организации дополнительной инфраструктуры для координации потоков между собой, что существенно экономит ресурсы информационной системы [5].

Однако, сетевое и межпроцессное взаимодействие являются довольно затратными технология в плане ресурсов компьютера и операционной системы. Поэтому, далее будет рассматриваться более подробно именно многопоточная архитектура, основанная на взаимодействии между потоками в рамках одного процесса операционной системы.

1.1 Понятия и определения

Основным понятием параллельного подхода в программировании является «Поток» и «Процесс».

Процесс — это независимый объект уровня операционной системы, которому выделены такие ресурсы как процессорное время и память [5]. Каждому процессу, операционной системой, выделяется свое собственное адресное пространство. Поэтому ни один процесс не может получить прямой доступ к данным другого процесса. Если требуется наладить сообщение между процессами, то необходимо использовать специальные технологии межпроцессного взаимодействия (MPI, Java RMI, CORBA) [5].

Поток — это объект уровня приложения, который использует то же самое адресное пространство памяти [5]. Множество потоков могут совместно использовать данные, находящиеся в общем пространстве памяти. Потоки являются частью какого-либо процесса операционной системы.

Потоки используются для распараллеливания частей приложения. Такой подход обеспечивает относительную простоту обработки общих данных и синхронизации потоков.

Другими важными понятиями являются: параллельное и конкурентное программирование. Под параллельным программированием понимается такой

подход, при котором обработка данных в программе выполняется параллельно и выполняющие обработку потоки не нуждаются в синхронизации и не влияют друг на друга [3, 4]. Конкурентное программирование представляет собой подмножество параллельного программирования, при котором потоки приложения активно сообщаются между собой либо обрабатывают общий ресурс. В последнем случае может потребоваться синхронизация потоков, чтобы получить в результате корректные данные.

Синхронизация потоков — это состояние информационной системы, при котором поток не может изменить какие-либо данные, если они уже зарезервированы для работы с другим потоком [3, 4]. Чтобы понять не заблокированы ли данные, для работы с ними в текущем исполняемом потоке, применяется механизм мониторов. Если монитор объекта уже захвачен другим потоком, то исполняемый поток не может манипулировать с данными этого объекта и ждет, когда монитор освободится. После освобождения монитора, поток, который ожидал этого события, блокирует его и выполняет свои действия. Монитор — объект, обеспечивающий доступ к неразделяемым ресурсам [3, 4].

Критическая секция — участок программного кода или данные, доступ к которому ограничен одним или несколькими потоками [5].

1.2 Основные проблемы многопоточного программирования

В основном, проблемы многопоточного программирования связаны с управлением оперативной памятью нескольких потоков. Существует несколько основных проблем, усложняющих создание многопоточных программ:

1. Проблема видимости данных между потоками;
2. Состояние гонки;
3. Взаимная блокировка потоков;
4. Поточное голодание [6, 7].

Проблема видимости данных между потоками — плавающая ошибка, возникающая в том случае, если потоки выполняются на разных физических ядрах процессора. Например, существует общий для двух потоков объект в оперативной памяти, при этом потоки выполняются на разных физических ядрах процессора. Для изменения данных объекта, процессору требуется вычитать его из оперативной памяти в регистры процессора или кэш. После чего, процессор сможет сделать изменения в объект. Однако, эти изменения не будут видны из другого потока, т.к. первый поток записал результат в свой

кэш и не опубликовал их в оперативную память. Для избежания такой ситуации, необходимо проинструктировать процессор считывать только актуальные значения. В Java это можно сделать в помощью ключевого слова volatile.

Состояние гонки — это плавающая ошибка, возникающая в многопоточных программах, когда один из потоков может остановиться, в это время другой поток изменяет данные [6, 7]. После того, как первый поток возобновит свое выполнение, в его области видимости окажутся некорректные значения данных. Для избежания такого рода ошибки применяют секции синхронизации над теми участками кода, которые могут выполняться в многопоточной среде.

Взаимная блокировка потоков — ошибка, возникающая когда два или более потока находятся в состоянии ожидания мониторов объектов, захваченных друг другом, и ни один из них не может выйти из этого состояния и продолжить выполнение [6, 7]. Взаимная блокировка является довольно сложным для выявления и устранения типом ошибки. Для избежания такой проблемы, следует следить за порядком захвата мониторов ресурсов.

Поточное голодание — проблема, возникающая, когда какой-либо поток занимает все процессорное время, не давая тем самым выполниться другому потоку [6, 7]. Такое поведение в многопоточной среде может происходить из-за неправильно расставленных приоритетов потоков. Для устранения этой ошибки, необходимо правильно расставить приоритет потоков, либо явно уведомить планировщик о том, что текущий поток готов уступить выполнение другому потоку.

1.3 API Java для работы с потоками

Язык программирования Java обладает богатыми возможностями для разработки многопоточных программ. Все эти возможности доступны в стандартном виде и не требуют дополнительной установки. К числу таких инструментов относятся стандартные возможности Java по созданию и управлению жизненным циклом потока, библиотека Java Concurrency и Fork Join Pool framework.

1.3.1 Жизненный цикл потока

Жизненный цикл потока в Java включает следующие состояния:

— вновь созданный;

— готовый к выполнению;
— выполняющийся;
— заблокированный;
— спящий;
— ожидающий;
— завершенный [3, 4].

В состоянии вновь созданного, поток оказывается после непосредственного создания потока, но до вызова метода start(). Т.е. поток был создан, но не запущен и до вызова метода start() не будет считаться активным.

Как только был вызван метод start(), поток попадает в пул потоков, его состояние изменяется на «готов к выполнению», где он ожидает, когда планировщик JVM предоставит ему процессорное время для выполнения действий [3, 4]. Поток впервые попадает в это состояние, когда был вызван метод start(), но так же, поток может вернуться к этому состоянию из других своих состояний, таких как: заблокированный, ожидающий, спящий.

Данное состояние поток достигает, когда планировщик JVM выбирает поток из пула потоков, готовых к выполнению. Поток получает квант процессорного времени и начинает выполнят свой код, его состояние переходит в «выполняющийся».

Если во время выполнения поток запрашивает уже занятый монитор, то он переходит в состояние «заблокированный». Похожая ситуация возникает, когда поток переходит в состояние «ожидающий», посредством вызова метода wait(). А так же, когда происходит вызов метода sleep() и поток переходит в состояние «спящий». Во всех трех случаях поток считается активным, но не может выполняться, т.е. не происходит выполнения никаких инструкций, прописанных в коде потока.

Когда завершается выполнение всех инструкций в методе run(), поток переходит в состояние «завершенный». Объект потока может быть все еще доступен в программе, но запустить его не получится. Достигнув состояния «завершенный», поток никогда далее не может быть перезапущен. Если попытаться выполнить метод start() для потока, достигшего состояния завершения, то будет выброшено исключение как только выполнение дойдет до данной команды. Из этого состояния возобновление потока невозможно [3–5].

1.4 Java Concurrency

Пакет java.util.concurrent появился в Java 1.5. Данный пакет содержит более современные и оптимизированные средства для работы с потоками, чем стандартный API Java. Сюда входят следующие элементы:

- Конкурентные коллекции — набор коллекций, более эффективно работающих в многопоточной среде нежели стандартные универсальные коллекции из java.util пакета, где в качестве оптимизации используются блокировки по сегментам данных;
- Очереди — различные виды очередей с поддержкой многопоточности;
- Синхронизаторы — вспомогательные утилиты для синхронизации потоков;
- Экзекуторы — содержит средства для создания пулов потоков, планирования работы асинхронных задач с получением результатов;
- Локи — представляет собой альтернативные и более гибкие механизмы синхронизации потоков по сравнению с базовыми synchronized, wait, notify, notifyAll;
- Атомики — классы с поддержкой атомарных операций над примитивами и ссылками [3, 4].

1.4.1 Fork Join Pool framework

Особо внимания заслуживает ForkJoinPool фреймворк, который появился в java.util.concurrent с версии Java 1.7. Основное назначение данного инструмента — решение рекурсивных задач по стратегии разделяй и властвуй. За счет разбиения на части, можно добиться эффективной параллельной обработки на нескольких потоках. В основе ForkJoinPool используется специальный алгоритм work-stealing, который хорошо проявляет себя в системах с большим количеством процессоров [4].

ForkJoinPool фреймворк состоит из следующих основных классов:

- ForkJoinPool — пул потоков и точка входа для запуска корневых задач. Подзадачи запускаются через методы задачи, от которой нужно ответвиться. По умолчанию создается пул с количеством потоков равным количеству доступных для JVM процессоров;
- ForkJoinTask — базовый класс для всех задач. Позволяет добавлять задачу в очередь текущего потока для асинхронного выполнения, запускать

задачу в текущем потоке, дожидаться завершения подзадачи с возвращением результата и др.;

— RecursiveTask — абстрактный класс унаследованный от ForkJoinTask, с объявлением метода compute(), в котором должна вычисляться основная логика;
— RecursiveAction — аналогичен классу RecursiveTask, но не возвращает результат;
— ForkJoinWorkerThread — класс рабочего потока, который будет обрабатывать задачи [4].

2 Исследование быстродействия последовательного и параллельного подходов программирования на примере алгоритма игры «Жизнь»

В качестве примера, для демонстрации различий между последовательным и параллельным подходом программирования, был выбран алгоритм игры «Жизнь».

2.1 Описание алгоритма игры «Жизнь»

Алгоритм игры «Жизнь» представляет собой форму клеточного автомата, придуманного английским математиком Джоном Конвеем в 1970 году. Он имеет регулярную сетку ячеек, где каждая ячейка может находиться в одном из конечного множества состояний, например, 1 и 0. Сетка может быть любой размерности. Для каждой ячейки определено множество соседних ячеек, называемых окрестностью [8]. Запуск клеточного автомата требует задания начального состояния всех ячеек, а так же правил перехода ячеек из одного состояния в другое. На каждой итерации, используя правила перехода и состояния соседних ячеек, определяется новое состояние каждой ячейки. Обычно правила перехода одинаковы для всех ячеек и применяются сразу ко всей сетке.

Каждая клетка может находиться в двух состояниях: быть «живой» (заполненной) или быть «мертвой» (пустой). Клетка имеет восемь соседей, окружающих ее. Распределение живых клеток в начале игры называется первым поколением [8]. Каждое следующее поколение рассчитывается на основе предыдущего по таким правилам:

— в мертвой клетке, рядом с которой ровно три живые клетки, зарождается жизнь;
— если у живой клетки есть две или три живые соседки, то эта клетка продолжает жить;
— если соседей меньше двух или больше трех, клетка умирает (от «одиночества» или «от перенаселенности»).

Игра прекращается, если на поле не останется ни одной «живой» клетки.

2.2 Реализация алгоритма игры «Жизнь»

Реализация логики игры жизнь, базируется на классе PlayGrid, который моделирует сетку ячеек. Внутри данного класса находится закрытое поле с

двумерным массивом, где каждый элемент массива представляет собой класс Cell, отвечающий за логику ячейки. Класс Cell содержит закрытое поле, характеризующее состояние ячейки — «пустая» или «заполненная». Для реализации поля состояния ячейки используется класс CellState, представляющий собой класс перечисления Java и позволяющий задать одно из двух перечисленных выше состояний [9]. В классе PlayGrid находится основной открытый метод getCellsForChangeState(), который используется как в последовательном, так и в параллельном подходе. Задача этого метода — выдать объекты ячеек, которые должны поменять свое состояние на противоположное, в рамках заданного интервала.

Для реализации последовательного и параллельного подходов, были созданы два класса контроллеров, каждый их которых отвечает за один из подходов. Последовательный подход реализован перебором всех ячеек сетки в цикле и находится в классе CellControllerSingleThread.

Параллельный подход реализован как рекурсивная задача с использование ForkJoinPool фрейворка из стандартной библиотеки Java. Все пространство двумерного массива сетки делится на четыре части по индексам, где каждая часть будет либо обработана, либо поделена на более мелкие части, в зависимости от ее размера. Обработка каждой части пространства заключается в вызове метода getCellsForChangeState() класса PlayGrid, который вернет список ячеек для заданной индексами части пространства двумерного массива. Затем, полученный в ходе предыдущей операции список элементов так же рекурсивно делится между рабочими потоками ForkJoinPool фрейворка, которые меняют состояние ячеек на противоположное.

2.3 Сравнение быстродействия последовательной и параллельной реализации алгоритма игры «Жизнь»

Апробирование последовательного и параллельного подходов проводилась на сетке с различным количеством ячеек. В целях эксперимента намеренно не применялись специальные методы оптимизации алгоритма, чтобы более ярко продемонстрировать разницу в быстродействии. К таким методам оптимизации относится, например, отслеживание ячеек, которые меняли свое состояние, и работа со списком данных объектов, а не со всем массивом целиком.

Быстродействие измерялось временем выполнения задачи по изменению

состояния ячеек сетки на каждой итерации. Замеры были выполнены на двух физических машинах с различной конфигурацией оборудования. В обоих случаях использовалась Java версии 1.8.0 и 16 Гб оперативной памяти DDR4. Конфигурация обеих систем описана в таблице 1.

Таблица 1 – Конфигурация используемых компьютерных систем

№	ОС	Тип процессора	Кол-во потоков
1	Linux Mint 19.1 x64	Intel Core i5-6600	4
2	Windows 10 x64	Intel Core i7-6700	8

Результаты измерения быстродействия для первой конфигурации приведены в таблице 2.

Таблица 2 – Время выполнения последовательного и параллельного подходов для первой конфигурации

Кол-во ячеек	Последовательный подход (мс)	Параллельный подход (мс)
10 000	0,02	0,56
1 000 000	17,41	7,2
100 000 000	1735,37	959,03

Результаты измерения быстродействия для второй конфигурации приведены в таблице 3.

Таблица 3 – Время выполнения последовательного и параллельного подходов для второй конфигурации

Кол-во ячеек	Последовательный подход (мс)	Параллельный подход (мс)
10 000	0,0	0,42
1 000 000	16,12	7,69
100 000 000	1689,67	467,1

Время выполнения определялось как среднее арифметическое из 10 000 итераций вызова метода контроллера runTask() с заданным количеством ячеек сетки.

Так же были выполнены более подробные замеры производительности алгоритма на данных системах, иллюстрирующие динамику расхода времени с увеличением количества ячеек в поле. Для выполнения замеров использовались unit-тесты, которые выполняли расчеты времени выполнения алгоритма для различных размеров поля ячеек. График производительности на данном интервале от 100 до 250 000 ячеек для системы с Windows представлен на рисунке 1.

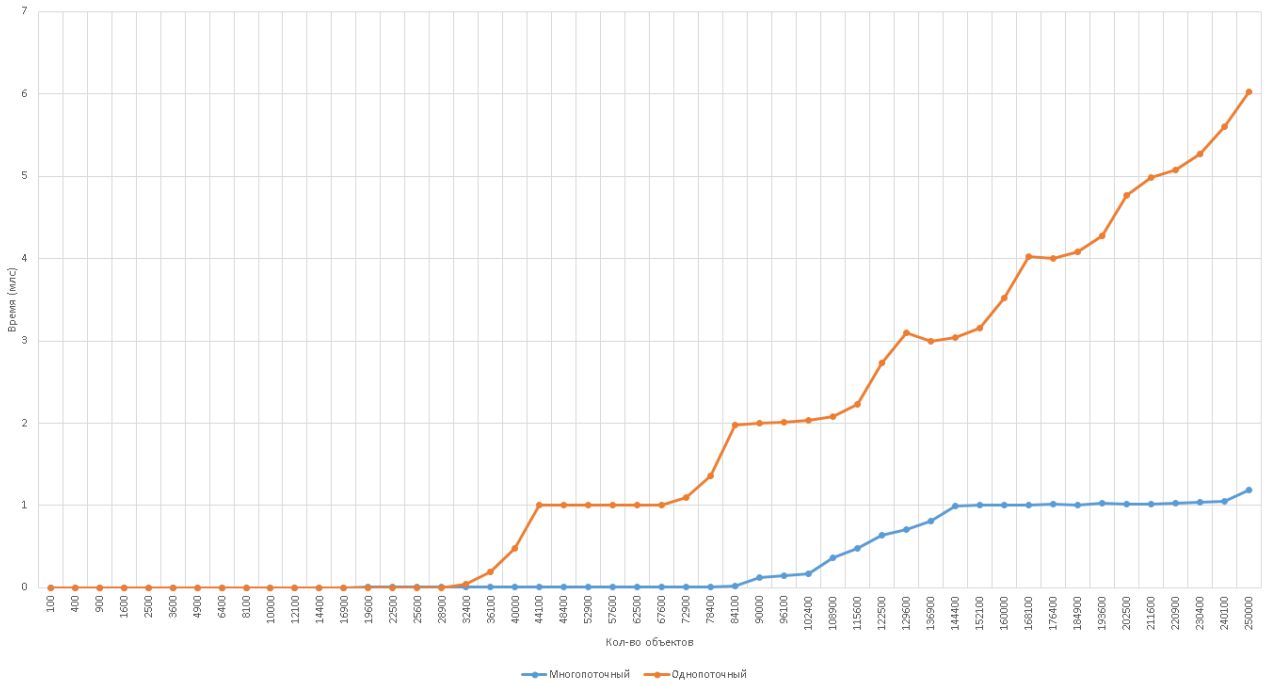

Рисунок 1 – График производительности системы второй конфигурации (Windows, процессор i7) на интервале от 100 до 250 000 ячеек

Исходя из анализа данных, можно заметить, что использование многопоточного программирования на небольшом наборе данных (до 30 000 объектов в конкретном примере) не эффективно. Время выполнения однопоточной программы для таких данных ниже, чем многопоточный вариант за счет того, что нет необходимости в координации потоков и выделения памяти для служебных объектов, которые требуются при многопоточном программировании. Как следствие, потребление памяти при однопоточном выполнении меньше.

Однако, с ростом количества обрабатываемых данных, картина меняется. График производительности на данном в интервале от 250 000 до 100 000 000 ячеек для системы с Windows представлен на рисунке 2.

Исходя из полученных данных, можно сделать следующие выводы.

1. Многопоточный подход имеет более медленную производительность по сравнению с последовательным на небольших объемах данных. Если объем данных не велик, то накладные расходы по созданию и управлению потоками будут сказываться на эффективности алгоритма. Так на 10 000 объектов последовательный подход выполняется быстрее параллельного примерно на половину миллисекунд.
2. Параллельный подход значительно выигрывает в быстродействии по сравнению с последовательным на больших объемах данных. Если кон-

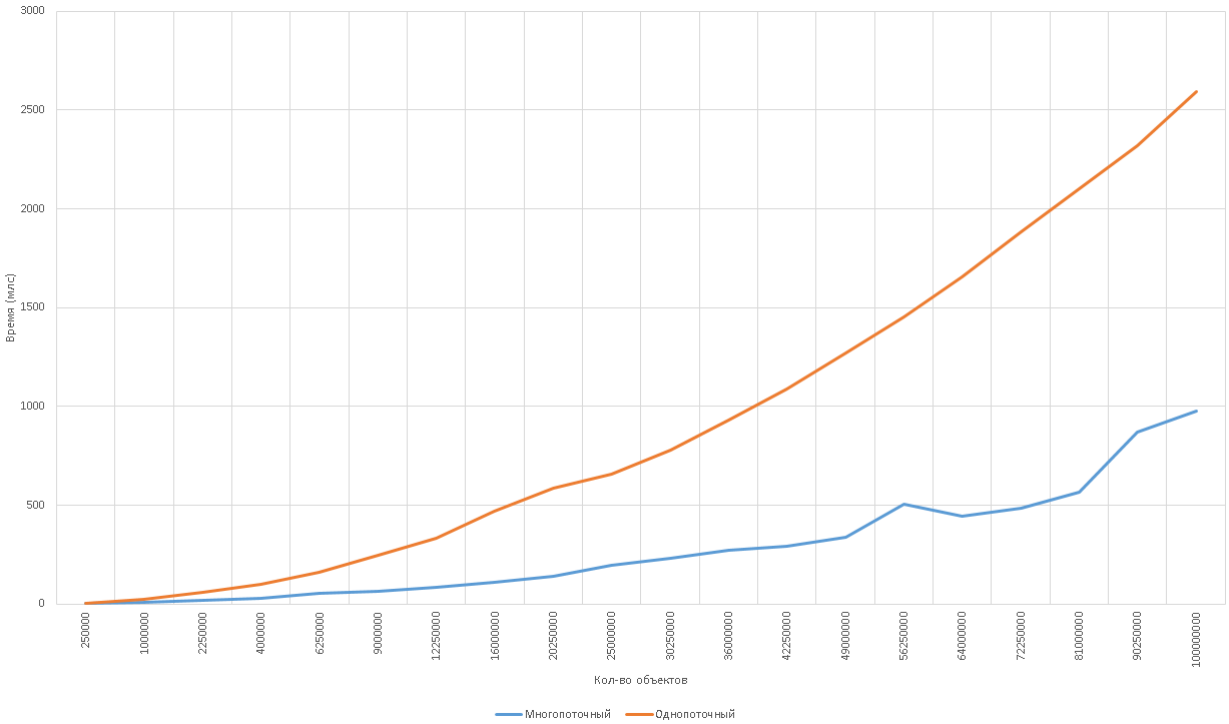

Рисунок 2 – График производительности системы второй конфигурации (Windows, процессор i7) на интервале от 250 000 до 100 000 000 ячеек

троллер, реализующий последовательный подход выполняет обработку 100 000 000 объектов примерно за полторы-две секунды, то параллельный подход показывает время выполнения менее одной секунду, вплоть до половины секунды. В результате выигрыш производительности составляет более 50 процентов.

3. Быстродействие параллельного подхода увеличивается с количеством потоков, доступных процессору. Для первой конфигурации с четырьмя потоками увеличение производительности составило около 50 процентов, а для второй конфигурации с восемью потоками прирост производительности составил уже около 70 процентов.

ЗАКЛЮЧЕНИЕ

В ходе данной научной работы были выполнены все поставленные цели и задачи. Были изучены основные понятия и определения, применяемые в многопоточном программировании, рассмотрены основные средства языка Java для работы в многопоточной среде, разработано приложение, демонстрирующее последовательную и параллельную обработку данных и проведен сравнительный анализ быстродействия реализованных подходов.

На основании проведенных исследований можно сделать вывод, что параллельное многопоточное программирование позволяет добиться существенного прироста производительности, без необходимости увеличения тактовой частоты процессора. Прирост в быстродействии сильно зависит от объема обрабатываемых данных и количества потоков, доступных процессору.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 *Андрейченко, Д. К.* Теоретические основы параллельного программирования / Д. К. Андрейченко, В. М. Велиев, А. А. Ерофтиев, М. С. Портенко. — Саратов: Саратовский национальный исследовательский государственный университет имени Н. Г. Чернышевского, 2014.

2 *Таненбаум, Э.* Современные операционные системы / Э. Таненбаум, Х. Бос. — Санкт-Петербург: Питер, 2015.

3 *Хорстманн, К.* Java. Основы / К. Хорстманн, Г. Корнелл. — Москва: Вильямс, 2014.

4 *Хорстманн, К.* Java. Расширенные средства программирования / К. Хорстманн, Г. Корнелл. — Москва: Вильямс, 2014.

5 *Шилдт, Г.* Java 8. Полное руководство / Г. Шилдт. — Москва: Вильямс, 2015.

6 *Мартин, Р.* Чистая архитектура. Искусство разработки программного обеспечения / Р. Мартин. — Санкт-Петербург: Питер, 2018.

7 *Эккель, Б.* Философия Java / Б. Эккель. — Санкт-Петербург: Питер, 2015.

8 Журнал Квант [Электронный ресурс]. — URL: http://kvant.mccme.ru/1974/09/igra_zhizn.htm (Дата обращения 11.04.2020). Загл. с экр. Яз. рус.

9 *Лафоре, Р.* Структуры данных и алгоритмы Java / Р. Лафоре. — Санкт-Петербург: Питер, 2013.