

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ  
УНИВЕРСИТЕТ» (НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет **ФИЗИЧЕСКИЙ**

Кафедра **АВТОМАТИЗАЦИИ ФИЗИКО-ТЕХНИЧЕСКИХ ИССЛЕДОВАНИЙ**

Направление подготовки **03.03.02 ФИЗИКА**

Образовательная программа: **БАКАЛАВРИАТ**

### ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Пантелеева Евгения Владимировича

(Фамилия, Имя, Отчество автора)

Тема работы «Исследование применимости сопрограмм в параллельной системах обработки данных»

«К защите допущена»

Заведующий кафедрой

канд. тех. наук

\*

Лысаков К. Ф.

...../.....  
(фамилия И., О.) / (подпись, МП)

«.....».....20...г.

Научный руководитель

канд. физ-мат. наук

заведующий лабораторией, ИСИ СО РАН

Бульонков М. А.

...../.....  
(фамилия И., О.) / (подпись, МП)

«.....».....20...г.

Дата защиты: «.....».....20...г.

Новосибирск, 2021

## СОДЕРЖАНИЕ

1	АННОТАЦИЯ . . . . .	6
2	ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ . . . . .	8
2.1	Java в параллельных системах . . . . .	8
2.2	Что такое Сопрограммы? . . . . .	8
2.3	JavaScript и C# . . . . .	10
2.4	Язык Go . . . . .	12
2.5	Проект "Loom" . . . . .	13
2.6	Применение сопрограмм . . . . .	14
3	ЦЕЛИ И ЗАДАЧИ . . . . .	16
4	ОПИСАНИЕ РЕШЕНИЯ . . . . .	18
4.1	Разработка тестов . . . . .	18
4.2	Переключение сопрограмм в Huawei JDK . . . . .	18
4.3	Сборка мусора . . . . .	20
4.4	Потребление памяти . . . . .	21
4.5	Применение сопрограмм в вычислительных задачах . . . . .	22
5	РЕЗУЛЬТАТЫ . . . . .	23
	БИБЛИОГРАФИЧЕСКИЙ СПИСОК . . . . .	24

## 1 АННОТАЦИЯ

Сопрограммы (англ. *coroutine*) — программный модуль, особым образом организованный для обеспечения взаимодействия с другими такими модулями по принципу кооперативной многозадачности[1]. Выполнение сопрограммы может быть приостановлено в точках явного планирования и передано другому модулю. При этом будет сохранено полное состояние сопрограммы (включая стек, значения регистров и счётчик команд).

Сопрограммы используются для реализации генераторов, итераторов, бесконечных списков. Они уже были реализованы во многих популярных языках программирования, таких как Go, Kotlin, C#, но не в языке Java. Сопрограммы по своему поведению очень похожи на потоки. Теоретически, с их помощью можно реализовывать параллелизм на уровне виртуальной машины. Изучению этого обстоятельства и посвящена работа, целью которой является изучение применимости сопрограмм вместо потоков в программах Java. Для достижения поставленной цели необходимо было решить следующие задачи:

- Разработать тесты для сравнения производительности потоков и сопрограмм.
- Создать базовый прототип сопрограмм.
- Сравнить производительность сопрограмм и потоков.
- Оценить целесообразность использования сопрограмм.

Работа проводилась на базе Huawei JDK, альтернативной реализации виртуальной машины Java, которая способна компилировать код перед исполнением программы (*ahead-of-time compilation*).

В ходе работы было выяснено, что отличие сопрограмм от потоков операционной системы заключается в потреблении меньшего объема системных ресурсов — адресного пространства и памяти. Что касается переключения контекста, то сопрограммы требуют меньших издержек, чем потоки. Все из-за того, что переключение осуществляется в пользовательском пространстве операционной системы средой исполнения языка.

Сопрограммы будут полезны разработчикам систем параллельной обработки данных из-за своих преимуществ, перечисленных выше.

## 2 ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

### 2.1 Java в параллельных системах

В современном мире язык Java используется для создания крупных многопоточных приложений для решения многих задач, в том числе и научных. Начиная с 90-х годов, Java используется для реализации систем управления Большим адронным коллайдером и для организации параллельной обработки результатов экспериментов, например, в библиотеке Colt Parallel[7]. Язык Java выбран из-за того, что он обладает управляемой средой исполнения, а это значительно упрощает разработку. Управляемая среда представляет собой вычислительное окружение, необходимое для настройки стека и кучи, включающее в себя функции сборки мусора, синхронизации потоков и так далее.

Традиционно, параллелизм реализуется внутри операционной системы с помощью механизма потоков, которые абстрагируют иногда взаимодействующие между собой независимо работающие задачи. Поток (от англ. thread) – наименьшая единица обработки, исполнение которой может быть запланировано ядром операционной системы[3]. В современных реалиях их количество обычно превышает число ядер процессора: аппаратного механизма параллелизма. Операционным системам следует представлять потоки как универсальное средство многозадачности для всех ныне существующих языков программирования. Потому потоки – это достаточно ”тяжеловесный” механизм: их создание и переключение несет в себе крупные накладные расходы. Это становится особенно заметно с ростом числа потоков в программе.

Но существует возможная альтернатива потокам – сопрограммы.

### 2.2 Что такое Сопрограммы?

Сопрограммы (англ. coroutine) — программный модуль, особым образом организованный для обеспечения взаимодействия с другими модулями

по принципу кооперативной многозадачности[1]. Выполнение сопрограммы может быть приостановлено в точках явного планирования и передано другому такому модулю. При этом будет сохранено полное состояние сопрограммы: включая стек, значения регистров и счётчик команд.

Концепция сопрограмм не нова: впервые они появились в языках программирования Симула[5], Модула-2[2] и Клу[6] в 1960 — 1970-е годы и использовались как еще одно языковое средство для реализации итераторов, генераторов, бесконечных списков и так далее. В тот момент концепция не получила широкого распространения и в более поздних языках программирования, таких как Си, С++, Java, она не была применена. Если нужна была альтернатива сопрограммаам, то использовались потоки. В случае языков Си и С++ можно применять функции для переключения контекста потока для представления модуля сопрограмм в виде библиотеки.

Что касается реализации, то часто сопрограммы исполняются специально выделенными потоками операционной системы, как это применяется в языке Go. С точки зрения пользователя языка они являются обычными потоками. И это не удивительно, ведь сопрограммы, как говорилось ранее, могут сохранять контекст<sup>1</sup> и приостанавливать поток вычисления, что свойственно и потокам. Но есть ряд существенных отличий:

- а) Переключение сопрограммы происходит в пользовательском пространстве операционной системы в отличие от потоков, которые планируются ядром ОС. Сопрограммы являются объектом среды исполнения языка, что позволяет оптимизировать переключение контекста под конкретный язык и виртуальную машину. Это уменьшает накладные расходы на переключение их потока выполнения. Как итог, системы, построенные на сопрограммаах, будут иметь лучшее время отклика и лучше масштабируются.
- б) Сопрограммы как правило имеют меньший размер стека. Как говорилось в прошлом пункте, сопрограммы являются сущностями среды исполнения в отличии от потоков, а значит виртуальная машина имеет больший

---

<sup>1</sup> Контекст – регистры и указатель на стек потока или сопрограммы.

контроль над ними. Потому появляется возможность выделять сравнительно небольшие стеки под сопрограммы. Если он переполняется, то виртуальная машина может создать новый стек большего размера и содержимое старого скопировать в новый. Благодаря меньшему размеру стека, виртуальная машина способна создавать число сопрограмм, превосходящее по количеству потоков ОС на том же оборудовании.

- в) Известно, что при вызове блокирующих операций, вроде приема данных из сети, вызывающий поток вытесняется операционной системой[11]. Это делается во избежание лишнего простоя процессора, который в момент работы блокирующего вызова может переключиться на выполнение другого потока, запланированного ядром операционной системы. Сопрограммами же управляет среда исполнения языка, которая способна вытеснить сопрограмму, инициирующую операцию ввода вывода, избежав при этом блокировки потока. Если бы была возможность запретить вытеснение по таймеру потоков, которые исполняют сопрограммы, то возможно техника блокирование сопрограмм даст прирост производительности.
- г) В отличие от потоков, сопрограммы планируются средой исполнения языка. Благодаря этому, разработчику приложения дается возможность оптимизировать планировщик сопрограмм под конкретную задачу, что не возможно в системах, построенных на потоках.

Сопрограммы уже поддерживаются многими языками программирования, такими как C++ стандарта 20, C#, JavaScript, Go и многие другие. Эти языки используют различные подходы к реализации сопрограмм. В следующих разделах рассмотрим способы переключения в управляемых средах.

## 2.3 JavaScript и C#

В JavaScript для работы с асинхронным вводом выводом введен класс Promise. Он представляет собой обёртку для значения, неизвестного на момент создания объекта. Promise позволяет обрабатывать результаты асинхронных

операций так, как если бы они были синхронными: вместо результата асинхронного метода возвращается обещание получить результат в будущем. Для удобной работы с Promise (с Task в случае C#), существует специальный синтаксис, который называется «async/await».

```
async function f() {  
    return 100;  
}
```

Ключевое слово `async` перед функцией означает, что функция всегда возвращает Promise.

```
let value = await promise;
```

Ключевое слово `await` заставляет ждать, пока Promise не исполнится, и возвращает результат операции. Это работает только в функциях, помеченных ключевым словом `async`.

Строго говоря, блоки `async/await` не являются сопрогнрамаами, так как они были определены ранее. У таких сопрогнрамм нет отдельного стека, нет контекста, и реализуются они следующим образом. Компилятор языка превращает каждую конструкцию `async/await` в конечный автомат.

Механизм `async/await` позволяет писать масштабируемый синхронный код и решает проблему контекста, вводя его новый вид, который представляет собой поток во всем, но несовместим с потоками операционной системы. Синхронный и асинхронный код обычно не могут быть смешаны в одном блоке кода, и в результате языки с поддержкой `async/await` требуют два разных API для приостановки выполнения `async` блока кода и текущего потока. В Kotlin существует таже самая проблема: один API предназначен для приостановки потока, а другой для остановки новой конструкции, которая похожа на поток, но не является им.

## 2.4 Язык Go



Сопрограммы в Go еще называют горутинами. Это функции, которые запускаются конкурентно с другими функциями. При запуске новой сопропрограммы нужно перед вызываемой функцией вставить ключевое слово **go**.

```
package main

import "fmt"

func foo() {
    fmt.Println("Foo called.")
}

func bar() {
    fmt.Println("Bar called.")
}

func main() {
    go foo()
    go bar()
}
```

Эта программа содержит 3 сопропрограммы. Первая - это функция `main`, являющаяся неявной сопропрограммой. Вторая и третья это `foo` и `bar`. Обычно при вызове функции, наша программа выполняет все ее операторы, а затем возвращает управление на следующую строку после вызова. С помощью сопропрограммы управление немедленно переходит к следующей строке без необходимости дожидаться завершения функции. Среда исполнения языка не позволяет использовать потоки операционной системы напрямую. Разрешается создавать только сопропрограммы.

Язык Go, начиная с версии 1.3 использует непрерывный стек корутин. В прологе вызываемой функции вставляется проверка, что текущего размера стека будет достаточно для исполнения кода. Если старый стек слишком мал, то выделяется память под новый стек и содержимое старого копируется в новый. Имеющиеся указатели на данные в стеке изменяются на новые. Данная реализация позволяет иметь небольшие стеки порядка 4–8 кБ, которые могут расти в неограниченных пределах. Но проверка выхода за границы размера в прологе дает накладные расходы, что является минусом.

Механизм переключения контекста сопрограммы в языке Go похож на переключения потоков[kernel.org???]: сохраняются в отдельный буфер необходимые регистры и информация, характерная только для данного потока/горутины, а затем в указатель стека записывается другой адрес. Но в отличие от потоков, при переключении сопрограммы Go выполняется меньше операций и сохраняются не все регистры[8].

## 2.5 Проект "Loom"

В текущей версии языка Java - JDK16, поддержка сопрограмм отсутствует. Но с конца 2017 года ведется работа в этом направлении проектом "Loom". Он направлен на сокращение усилий по написанию, поддержке высокопроизводительных параллельных приложений, которые максимально используют доступное оборудование[9].

Сопрограммы Loom - это виртуальные потоки, создание и блокировка которых требует меньше накладных расходов[9]. Они управляются средой исполнения языка Java. В отличие от представленных в стандартной библиотеке потока "java.lang.Thread", виртуальные потоки не являются оболочками потоков ОС, а реализованы в JDK.

Loom использует другой подход к переключению сопрограмм, отличающийся от языка Go. Когда сопрограмма начинает свою работу, она использует стек потока, на котором она запущена. При переключении сопрограммы происходит копирование части стека, используемой сопрограммой, в отдельный буфер памяти. Если нужно передать управление обратно в сопрограмму, то происходит опустошение буфера и копирование его содержимого на вершину стека потока[9]. Такая техника выбрана в целях совместимости со всеми существующими сборщиками мусора, которые есть в OpenJDK. Некоторые из них не могут поддерживать объекты на куче, которые могут хранить ссылки по смещениям в памяти, меняющиеся на протяжении всего времени существования объекта на куче.

## 2.6 Применение сопрограмм

Сопрограммы имеют ряд практических применений. При появлении они использовались как средство для создания генераторов, итераторов, бесконечных списков и так далее.

С недавнего времени, более эффективный механизм сопрограмм рассматривается как альтернатива потокам в параллельных системах, что позволяет увеличить количество исполняемых задач в несколько раз. Это может быть полезно для создания систем обработки огромного объема данных.

Сопрограммы подходят для работы со множеством относительно независимых задач.

Рассмотрим на примере http-сервера. Каждый из запросов, которые он обслуживает, в значительной степени независим от других. Для каждого из них выполняется синтаксический анализ пакета, делается запрос к базе данных и/или к другому серверу, формируется ответ, который отправляется в клиенту. Каждый запрос не взаимодействует с другими одновременными HTTP-запросами, но они конкурирует с ними за процессорное время и ресурсы ввода-вывода. Другими словами, http-сервер имеет некоторые единицы параллелизма домена приложения, специфичные для его области, причем исполнение некоторой задачи выполняется независимо от другой в то же время. В данном случае единицей параллелизма является http-запрос. Для базы данных это может быть транзакция.

Проблема заключается в том, что поток, не может соответствовать масштабу единиц параллелизма домена приложения. Потенциально сервер может обрабатывать до миллиона одновременных открытых сокетов, но операционная система не может эффективно обрабатывать более нескольких тысяч не бездействующих потоков.

Сопрограммы позволяют избежать такую проблему. В силу своих преимуществ, их может быть создано несколько миллионов, в то время как операционная система может поддерживать всего лишь несколько тысяч активных

потоков. В добавок они уменьшают время отклика сервера из-за большей скорости переключения.

Кроме того, сопрограммы позволяют легко организовать работу с асинхронным кодом. Неблокирующие операции, в том числе и асинхронный ввод-вывод, строятся на обратных обработчиках. Когда такого кода очень много, его становится очень трудно отлаживать и исправлять. Сопрограммы помогают избежать такой проблемы, представляя список обработчиков как последовательный кусок кода.

### 3 ЦЕЛИ И ЗАДАЧИ

Целью данной работы является изучение применимости сопрограмм вместо потоков в программах Java. При ее выполнении ключевой задачей становится создание модуля сопрограмм в языке Java. Его разработка проводилась на базе HuaweiJDK: альтернативной реализации виртуальной машины Java, которая поддерживает компиляцию перед исполнением. Создание сопрограмм в управляемой среде имеет ряд аспектов:

- а) Виртуальная машина должна уметь переключать контекст выполнения. Без этого механизма невозможно представить реализацию сопрограмм.
- б) Требуется проводить сборку мусора объектов, чьи ссылки лежат на стеках и в сохраненных регистрах сопрограмм.
- в) Сопрограммы должны корректно работать в критических секциях, то есть в случае Java внутри блоков кода, помеченных ключевым словом **synchronized**.
- г) Среда исполнения должна уметь вытеснять сопрограмму, которая инициировала блокирующую операцию, и запускать новую. Примером такой операции может послужить сетевой ввод-вывод. Как говорилось ранее, это поможет избежать вытеснения потока.
- д) Необходимо уметь корректно обрабатывать исключение, брошенное из сопрограммы. Причем поведение исключения может быть разным: когда исключение развернуло весь стек вызовов, оно может быть переброшено в несущий поток либо привести к приостановки сопрограммы и вывода ее стека вызовов.
- е) И наконец, HuaweiJDK с модулем сопрограмм должен проходить набор тестов JCK.<sup>2</sup>

Для того, чтобы минимальный прототип сопрограмм заработал в управляемой среде исполнения, достаточно реализовать первые два пункта. Как было показано в обзоре предметной области, существует несколько способов переключения контекста сопрограмм. Для того, чтобы выбрать оптимальный вариант,

---

<sup>2</sup>Java Compatibility Kit (JCK) - набор тестов на совместимость с Java.

необходимо разработать набор тестов на производительность сопрограмм в управляемых средах Go, Java/Loom, так как на текущий момент не существует подходящих тестов, которые бы измеряли скорость переключения контекста и размер физической памяти в этих языках. Требуется так же выявить точные значения этих параметров и для потоков операционной системы, чтобы выделить их ключевые отличия от сопрограмм. В дальнейшем, разработанные тесты можно будет использовать для сравнение производительности сопрограмм в HuaweiJDK от OpenJDK и Go.

В итоге, полный список задач, поставленных для достижения поставленной цели выглядит следующим образом:

- а) Разработать тесты для сравнения производительности потоков и сопрограмм.
- б) Создать базовый прототип сопрограмм.
- в) Сравнить производительность сопрограмм и потоков.
- г) Выявить ключевые плюсы использования сопрограмм.

## 4 ОПИСАНИЕ РЕШЕНИЯ

### 4.1 Разработка тестов

Пред началом реализации сопрограмм в Huawei JDK были разработаны тесты производительности для языков Go и ранней версией проекта "Loom". Первый тест измеряет скорость переключение контекста сопрограмм в языках Go и Java с "Loom". Тест принимает в аргументах число сопрограмм или потоков, которые при каждом запуске исполняют фиксированное число переключений. При этом гарантируется, что программа будет исполнена только одним ядром процессора. Каждый тест Go и Java запускается 100 раз и полученные результаты усредняются. При их исполнении измеряется RSS<sup>3</sup>. Результаты измерений для Huawei JDK, OpenJDK и языка Go будут представлены в следующих разделах.

### 4.2 Переключение сопрограмм в Huawei JDK

В таблице 4.1 показаны результаты измерений скорости переключения сопрограмм Go и OpenJDK/Loom<sup>4</sup>. Видно, что сопрограммы из языка Go выигрывают OpenJDK в скорости переключения.

---

<sup>3</sup>Resident set size (RSS) — размер памяти, выделенных процессу операционной системой и в настоящее время находящееся в ОЗУ (RAM)

<sup>4</sup>Все измерения проводились на операционной системе Ubuntu, kernel 4.15, Intel Core i7-8700, 4.6 ГГц, 32 Гб ОЗУ

Таблица 4.1 – Число переключений сопрограмм

<i>Шт.</i>	<i>Переключений, тыс./сек.</i>	
	<i>OpenJDK/"Loom"</i>	<i>Go</i>
<i>100</i>	<i>1 900 ± 20</i>	<i>18 187 ± 219</i>
<i>1 000</i>	<i>1 775 ± 20</i>	<i>17 934 ± 332</i>
<i>5 000</i>	<i>1 703 ± 30</i>	<i>12 892 ± 339</i>
<i>10 000</i>	<i>1 924 ± 235</i>	<i>8 307 ± 80</i>
<i>20 000</i>	<i>1 863 ± 217</i>	<i>7 045 ± 72</i>
<i>30 000</i>	<i>1 772 ± 182</i>	<i>6 391 ± 94</i>
<i>40 000</i>	<i>1 606 ± 194</i>	<i>5 790 ± 67</i>
<i>50 000</i>	<i>1 503 ± 157</i>	<i>5 292 ± 122</i>

Поэтому, для реализации в HuaweiJDK был выбран подход языка Go. Из-за простоты использования, в раннем прототипе для переключения контекста использовались функции из библиотеки glibc `getcontext` и `swapcontext`. Результат измерения раннего прототипа приведен в таблице 4.2.

Таблица 4.2 – Число переключений сопрограмм

<i>Шт.</i>	<i>Переключений, тыс./сек.</i>
	<i>HuaweiJDK</i>
<i>100</i>	<i>1 956 ± 38</i>
<i>1 000</i>	<i>1 829 ± 12</i>
<i>5 000</i>	<i>1 578 ± 39</i>
<i>10 000</i>	<i>1 316 ± 20</i>
<i>20 000</i>	<i>1226 ± 8</i>
<i>30 000</i>	<i>1068 ± 7</i>
<i>40 000</i>	<i>928 ± 7</i>
<i>50 000</i>	<i>881 ± 5</i>

Анализ исходного кода функции `swapcontext` показал, что переключение можно ускорить. Она используется для переключения потоков в операционных



системах на ядре Linux[ref]. Функция `getcontext` предоставляет информацию о пользовательском контексте, описывающую состояние потока перед активацией обработчика сигнала, в том числе и предшествующую маску сигналов и сохраненные значения регистров, в частности, программный счетчик и указатель стека[11]. Функция `swarcontext` предварительно делает системный вызов для сохранения текущей маски сигналов потока, в чем нет необходимости при переключении контекста сопрограмм. Это один из факторов, побудивший реализовать аналоги функций `getcontext`, `swarcontext` внутри HuaweiJDK, которые бы учитывали особенности виртуальной машины. На таблице 4.3 представлено сравнение результатов скоростей переключения.

Таблица 4.3 – Сравнение числа переключений

<i>Шт.</i>	<i>Число переключений, тыс./сек.</i>	
	<i>getcontext/setcontext</i>	<i>Функции из HuaweiJDK</i>
<i>100</i>	<i>1 956 ± 38</i>	<i>12 980 ± 540</i>
<i>1 000</i>	<i>1 829 ± 12</i>	<i>11 420 ± 694</i>
<i>5 000</i>	<i>1 578 ± 39</i>	<i>5 875 ± 183</i>
<i>10 000</i>	<i>1 316 ± 20</i>	<i>4 459 ± 162</i>
<i>20 000</i>	<i>1226 ± 8</i>	<i>3 604 ± 93</i>
<i>30 000</i>	<i>1068 ± 7</i>	<i>3 031 ± 94</i>
<i>40 000</i>	<i>928 ± 7</i>	<i>2 653 ± 87</i>
<i>50 000</i>	<i>881 ± 5</i>	<i>2 315 ± 60</i>

Видно, что функции из `glibc` проигрывают в несколько раз при любом количестве сопрограмм.

### 4.3 Сборка мусора

Следующим шагом работы стала сборка мусора объектов, расположенных в зоне видимости функций, вызванных в сопрограмме. Виртуальная машина Java хранит список начал и вершин всех потоков, созданных в процессе работы.

Это необходимо для того, чтобы при сборке мусора стало возможным нахождение всего корневого множества живых объектов.

В случае сопрограмм необходимо повторить данную логику: требуется хранить все адреса начал и вершин в некотором буфере. Но в отличие от потоков, нужно еще сохранять регистры приостановленных сопрограмм для корректной сборки.

#### 4.4 Потребление памяти

После реализации базового прототипа, наступил этап измерения потребления физической памяти.

Таблица 4.4 – Измерение потребления физической памяти

<i>Шм.</i>	<i>Резидентная память</i>		
	<i>HuaweiJDK</i>	<i>OpenJDK/"Loom"</i>	<i>Go</i>
<i>100</i>	<i>18 Мб</i>	<i>130 Мб</i>	<i>3,040 Мб</i>
<i>1000</i>	<i>22 Мб</i>	<i>161 Мб</i>	<i>3,105 Мб</i>
<i>5000</i>	<i>32 Мб</i>	<i>187 Мб</i>	<i>3,156 Мб</i>
<i>10000</i>	<i>37 Мб</i>	<i>193 Мб</i>	<i>3,308 Мб</i>
<i>20000</i>	<i>45 Мб</i>	<i>196 Мб</i>	<i>3,320 Мб</i>
<i>30000</i>	<i>49 Мб</i>	<i>197 Мб</i>	<i>3,350 Мб</i>
<i>40000</i>	<i>51 Мб</i>	<i>200 Мб</i>	<i>3,390 Мб</i>
<i>50000</i>	<i>57 Мб</i>	<i>202 Мб</i>	<i>3,407 Мб</i>

Как видно из таблицы 4.4, HuaweiJDK обходит альтернативную реализацию Java OpenJDK в потреблении памяти на всех измерениях, но проигрывает языку Go. Огромное потребление ОЗУ программы, исполняемой виртуальной машиной OpenJDK, можно связать с наличием JIT-компилятора<sup>5</sup> в памяти во время выполнения процесса.

<sup>5</sup>Jist-in-time (JIT) компиляция - компиляция программы во время ее исполнения. Эта техника позволяет проводить более смелые оптимизации, поскольку JIT-компилятор владеет большей информацией о выполняемой программе, чем статический.

В таблице 4.5 представлены результаты измерения потребления памяти потоками операционной системы. Как и говорилось ранее, потоки тратят больший объем ОЗУ, чем сопрограммы. Причины этому были рассмотрены в обзоре предметной области.

Таблица 4.5 – Сравнение потребления памяти сопрограмм и потоков.

<i>Шт.</i>	<i>Размер физической памяти</i>	
	<i>Сопрограммы</i>	<i>Потоки</i>
<i>100</i>	<i>18 Мб</i>	<i>34 Мб</i>
<i>1000</i>	<i>22 Мб</i>	<i>35 Мб</i>
<i>5000</i>	<i>32 Мб</i>	<i>37 Мб</i>
<i>10000</i>	<i>37 Мб</i>	<i>40 Мб</i>
<i>20000</i>	<i>45 Мб</i>	<i>49 Мб</i>
<i>30000</i>	<i>49 Мб</i>	<i>56 Мб</i>
<i>40000</i>	<i>51 Мб</i>	<i>63 Мб</i>
<i>50000</i>	<i>57 Мб</i>	<i>72 Мб</i>

#### 4.5 Применение сопрограмм в вычислительных задачах

Todo. Раздел о том, как перемножались матрицы и вычислялось фурье преобразование на потоках/сoproграммах. И что их этого получилось.

## 5 РЕЗУЛЬТАТЫ

В результате данной работы был разработан набор тестов для сравнения производительности потоков и сопрограмм в управляемых средах языка Go и OpenJDK/Loom. Тесты позволяют измерить скорость переключения и потребление физической памяти [укажи что RES???!!?]. Так же был реализован базовый прототип сопрограмм в HuaweiJDK, который способен переключать контекст и поддерживает сборку мусора объектов, ссылки которых находятся на стеках сопрограмм. Далее, учитывая особенности виртуальной машины Java, была оптимизирована скорость переключения более чем в 3 раза.

С помощью ранее разработанных тестов были получены результаты измерения производительности сопрограмм в HuaweiJDK, и проведено сравнение результатов с OpenJDK/Loom и языком Go. Созданные в рамках этой работы сопрограммы в HuaweiJDK обходят производительность OpenJDK в 3–8 раз в зависимости от их числа.

В дальнейшем планируется:

- а) Поддержка `synchronized` блоков языка Java.
- б) Реализация вытеснения сопрограмм, иницирующих блокирующие операции.

## БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Сопрограммы [Электронный ресурс]  
<https://en.wikipedia.org/wiki/Coroutine>
2. Модуля-2 [Электронный ресурс] <https://en.wikipedia.org/wiki/Modula-2>
3. Поток [Электронный ресурс] [https://ru.wikipedia.org/wiki/Поток\\_выполнения](https://ru.wikipedia.org/wiki/Поток_выполнения)
4. JIT-компиляция [https://docs.oracle.com/cd/E13150\\_01/jrockit\\_jvm/jrockit/gen](https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/gen)
5. Симула [Электронный ресурс] <https://en.wikipedia.org/wiki/Simula>
6. Язык программирования CLU [Электронный ресурс]  
[https://en.wikipedia.org/wiki/CLU\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/CLU_(programming_language))
7. Библиотека Colt Parallel [Электронный ресурс]  
<https://sites.google.com/site/piotrwendykier/software/parallelcolt>
8. Переключение контекста в языке Go. [Электронный ресурс]  
[https://github.com/golang/go/blob/master/src/runtime/asm\\_amd64.s](https://github.com/golang/go/blob/master/src/runtime/asm_amd64.s)
9. Проект Loom [Электронный ресурс]  
<https://wiki.openjdk.java.net/display/loom/Main>
10. Исходный код библиотеки Glibc 2.33 [Электронный ресурс]  
<https://www.gnu.org/software/libc/>
11. Майкл Керриск. Linux API. Исчерпывающее руководство. — СПб.: Питер, 2018.