

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

Пантелеев Евгений Владимирович

(Фамилия, Имя, Отчество автора)

Тема работы **Исследование применимости сопрограмм в параллельных системах обработки данных**

**«К защите допущена»**

И. о. зав. кафедрой

**Научный руководитель**

к.т.н.

канд. физ-мат. наук

Заведующий лабораторией, ИСИ СО РАН

Лысаков К.Ф./.....  
(фамилия И., О.) / (подпись, МП)

Бульонков М.А./.....  
(фамилия И., О.) / (подпись, МП)

«.....».....20...г.

«.....».....20...г.

Дата защиты: «.....».....20...г.

Новосибирск, 2021

## СОДЕРЖАНИЕ

1	ВВЕДЕНИЕ . . . . .	3
2	ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ . . . . .	6
2.1	Отличие сопрограмм от потоков . . . . .	6
2.2	JavaScript и C# . . . . .	8
2.3	Язык Go . . . . .	9
2.4	Проект "Loom" . . . . .	11
2.5	Применение сопрограмм . . . . .	12
3	РЕАЛИЗАЦИЯ МОДУЛЯ . . . . .	15
4	ОПИСАНИЕ РЕШЕНИЯ . . . . .	16
4.1	Разработка тестов . . . . .	16
4.2	Переключение сопрограмм в Huawei JDK . . . . .	17
4.3	Сборка мусора . . . . .	20
4.4	Потребление памяти . . . . .	21
4.5	Применение сопрограмм в вычислительных задачах . . . . .	23
5	ВЫВОДЫ . . . . .	25
	БИБЛИОГРАФИЧЕСКИЙ СПИСОК . . . . .	27

## 1 ВВЕДЕНИЕ

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, которое позволяет настраивать размер доступной стековой и динамической памяти, включающее в себя функции сборки мусора, синхронизации потоков и так далее.

Традиционно, параллелизм реализуется внутри операционной системы с помощью механизма потоков, которые абстрагируют взаимодействующие между собой независимо работающие задачи. Поток (англ. thread) – наименьшая единица обработки, исполнение которой может быть запланировано ядром операционной системы[3]. Модель программирования, которая позволяет нескольким потокам выполняться в рамках одного экземпляра исполняемой программой называется многопоточностью. Потоки в контексте одной программы способны совместно использовать ее ресурсы. Многопоточная модель предоставляет разработчикам абстракцию параллельного выполнения задач. Преимущество многопоточной программы позволяет ей работать быстрее на компьютерах, имеющих несколько процессоров и/или ядер. Из-за этого, с помощью потоков можно добиться реального параллельного выполнения задач. Поток создается и планируется операционной системой, которая с помощью системных вызовов предлагает разработчику прикладных программ манипулировать ими. Параллельное программирование требует осторожности со стороны программиста во избежание состояния гонки и другого контринтуитивного поведения. Что касается планирования, то в современных операционных системах общего назначения как правило потоки планируются посредством вытесняющей многозадачности. Это предполагает, что операционная система

принимает решение о переключении между задачами по истечении некоторого выделенного кванта времени[21]. В такой системе каждой выполняемой задаче дается приоритет, в зависимости от которого принимается решение о планировании. В отличие от кооперативной многозадачности, управление операционной системе передается независимо от состояния работающего потока, благодаря чему, в частности, зависшие (ушедшие в бесконечный цикл) задачи не блокируют ядро процессора. За счёт регулярного переключения задач также улучшается отзывчивость системы, своевременность освобождения ресурсов, которые больше не используются потоком[22].

К сожалению, модель потоков имеет ряд минусов. Операционные системы представляют потоки как универсальное средство многозадачности для всех ныне существующих языков программирования. Проблема в том, что 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. Избежать накладных расходов на использование потоков можно, применяя вместо них сопрограммы.

Сопрограмма (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.

Сопрограммы уже были применены в наборе библиотек для научных расчетов Go-NEP[15]. Библиотеки написаны на языке Go, в котором нельзя использовать потоки. Если необходима альтернатива потокам, то используются сопрограммы.

Целью данной работы является изучение применимости сопрограмм вместо потоков в параллельных системах. В частности, для изучения применяются многопоточные программы Java. Для достижения цели необходимо решить следующие задачи:

- а) Провести анализ реализаций сопрограмм в других языках. Это позволит понять причину, почему сопрограммы потребляют меньше процессорного времени и памяти.
- б) Создать базовый прототип сопрограмм. Его реализация проводилась на базе Huawei JDK.
- в) Сравнить производительность сопрограмм и потоков.
- г) Выявить ключевые плюсы использования сопрограмм.

В обзоре предметной области будут рассмотрены отличия потоков от сопрограмм. Затем, будут показаны способы реализаций сопрограмм в различных языках программирования, а после описаны этапы выполнения поставленной цели.

## 2 ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

### 2.1 Отличие сопрограмм от потоков

Концепция сопрограмм впервые появилась 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 сопроγραμμαам, 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 реализовано 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[20]. Сопрограммы являются объектом среды исполнения языка, что позволяет оптимизировать переключение контекста под конкретный язык и виртуальную машину. Это уменьшает накладные расходы на переключение. Как итог, системы, построенные на сопроγραμμαах, 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 стек 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 запретить вытеснение по таймеру потоков, которые исполняют сопрограммы, то техника блокирования сопрограмм может дать 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 невозможно в системах, построенных на потоках.

Сопрограммы уже поддерживаются многими языками программирования, такими как C++ стандарта 20[19], C#, JavaScript, Go и многими другими. Эти языки используют различные подходы к реализации сопрограмм. В следующих разделах рассмотрим способы переключения в управляемых средах.

## 2.2 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#)[17], 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 сопрограммами, как они были определены ранее. У таких "сопрограмм" 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 та же самая проблема: один API предназначен для приостановки потока, а другой для остановки новой конструкции, которая похожа на поток, но не является им[18].



## 2.3 Язык Go

Сопрограммы в Go еще называют горутинами (goroutines). Это функции, которые запускаются конкурентно с другими функциями[12]. При запуске новой сопрограммы 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()
}
```

Этот пример содержит 3 сопрограммы. Первая - это функция `main`, являющаяся неявной сопрограммой. 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 сопрограммы управление немедленно переходит к следующей строке без необходимости дожидаться завершения функции. Среда исполнения языка не позволяет использовать потоки операционной системы напрямую. Разрешается создавать только сопрограммы.

Язык Go, начиная с версии 1.3 использует непрерывный стек горутин. 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 переключение потоков: 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.[8].

## 2.4 Проект "Loom"

В текущей версии языка Java - JDK16<sup>2</sup>, 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]. Прототип сопрограмм реализуется на базе OpenJDK и сейчас уже можно использовать его для написания тестовых программ, но для применения в реальных проектах он еще не готов.

Сопрограммы "Loom" - это виртуальные потоки, создание и блокировка которых требует меньше накладных расходов[9]. Они управляются средой исполнения языка Java, то есть операционная система ничего не знает об их существовании. В отличие от представленных в стандартной библиотеке потоков `java.lang.Thread`, виртуальные потоки не являются оболочками потоков ОС, а реализованы в JDK. Сопрограммы спроектированы так, что в нынешних существующих приложениях в будущем будет возможно заменить потоки на сопрограммы практически один к одному с минимальными затратами усилий разработчика.

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.

---

<sup>2</sup>Java Development Kit (JDK) - разработчика приложений на языке Java, содержащий в себе базовый набор программ, необходимых для написания программ.

## 2.5 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 одномоментно исполняемых задач в несколько раз. Как было показано ранее, сопрограммы имеют меньшее потребление процессорного времени на переключение. Не трудно догадаться, что наибольший выигрыш от внедрения сопрограмм получают те задачи, в которых часто происходят блокирующие операции и блокировка потоков. А таких задач существует огромное множество.

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 клиенту. Каждый запрос не взаимодействует с другими одновременными HTTP-запросами, но он 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.

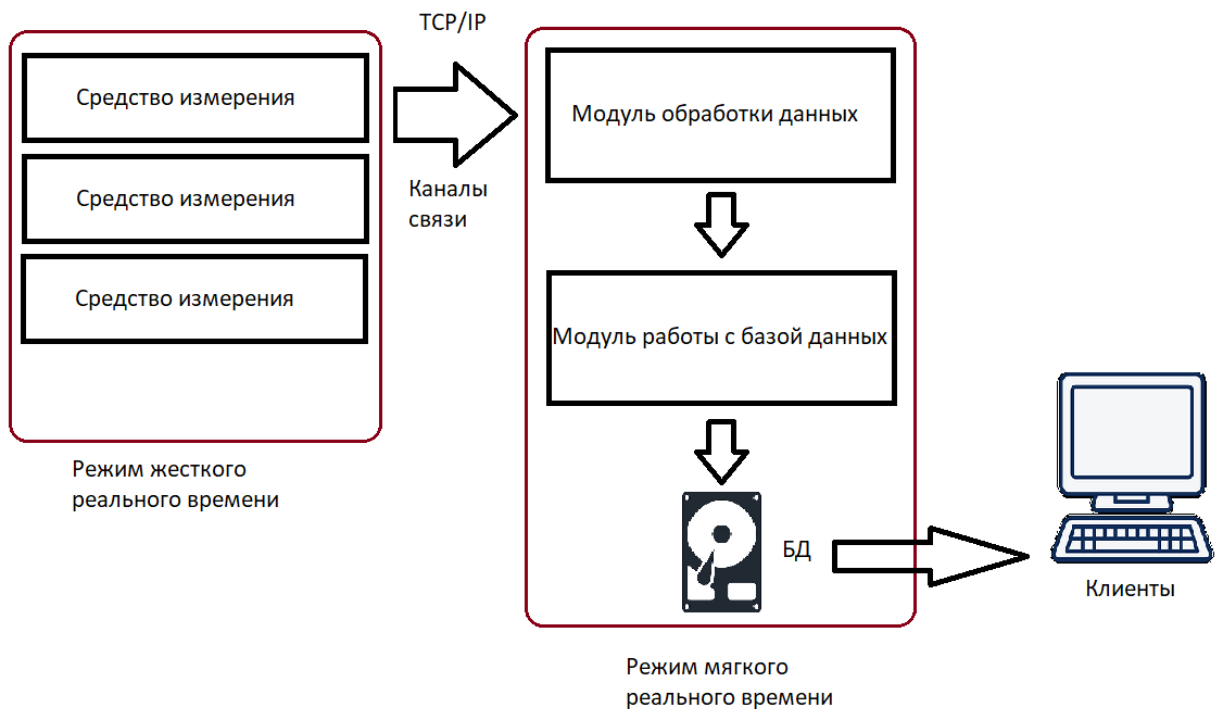
Проблема заключается в том, что поток не может соответствовать масштабу единиц параллелизма домена приложения. Потенциально сервер может обрабатывать до миллиона одновременных открытых сокетов, но операционная система не может эффективно обрабатывать более нескольких тысяч не бездействующих потоков.

Сопрограммы позволяют избежать такой проблемы. В силу своих преимуществ их может быть создано несколько миллионов, в то время как

операционная система может поддерживать всего лишь несколько тысяч активных потоков. Вдобавок они уменьшают время отклика сервера из-за большей скорости переключения.

Сопрограммы можно также эффективно применить в системах, работающих в режиме мягкого реального времени. На рисунке 2.1 изображена упро-

Рис. 2.1 – Схема системы обработки и приема данных.



щенная схема системы сбора и обработки данных. Она состоит из двух больших блоков. Левый из них отвечает за сбор сырых данных в режиме жесткого реального времени. Он состоит из набора средств для измерения, способных аккумулировать сырые данные в локальную память. Левый блок связывается с правым модулем каналами связи, которые могут работать по протоколу TCP/IP или аналогичному. Данные из средств измерения затем отправляются в правый блок, который делает их предобработку и сохранение в базу данных/на диск для последующего использования и изучения подключающимися клиентами.

В работе правого блока очень важна его пропускная способность: чем она будет выше, тем быстрее данные попадут в базу данных. Эта задача очень похожа на ситуацию с http-сервером, описанным ранее. Здесь так же присутствует множество независимых событий, а именно поступающие данные со

средств измерения. Потому в этой задаче можно использовать сопрограммы. А поскольку модуль работает в системе мягкого реального времени, то допустимо использовать для его разработки языки с управляемой средой исполнения, вроде Java и Go.

По принципу, изображенном на рисунке 2.1, работает система контроля износа колес железнодорожных составов. Во время подхода поезда к станции, средства измерения проводят все требуемые замеры параметров еще движущихся в тот момент колес. Затем, пока поезд стоит на остановке, сырые данные отправляются в модуль обработки данных, находящийся в диспетчерской, и обрабатываются. Очень важно провести этот анализ до отбытия поезда, чтобы можно было успеть обнаружить колеса с дефектами. Потому система обработки должна работать в режиме мягкого реального времени.

На этом область применения сопрограмм не исчерпывается. Они позволяют легко организовать работу с асинхронным кодом. Неблокирующие операции, в том числе и асинхронный ввод–вывод, строятся на обратных обработчиках (callbacks). 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 РЕАЛИЗАЦИЯ МОДУЛЯ

Разработка прототипа сопрограмм проводилась на базе Huawei JDK: 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 приостановке сопрограммы и распечатке ее стека вызовов.
- е) И наконец, Huawei JDK с модулем сопрограмм должен проходить набор тестов JCK <sup>3</sup>.

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 определения оптимального варианта был разработан набор тестов на производительность сопрограмм в управляемых средах. О их результатах речь пойдет позднее.

---

<sup>3</sup>Java Compatibility Kit (JCK) - набор тестов на совместимость с Java.

## 4 ОПИСАНИЕ РЕШЕНИЯ

### 4.1 Разработка тестов

Для изучения существующих реализаций сопрограмм были разработаны тесты производительности для Go и OpenJDK/Loom. Тесты принимают аргументом число сопрограмм или потоков, которые при каждом запуске исполняют фиксированное число переключений. При этом гарантируется, что будет задействовано только одно ядро процессора. Каждый тест Go и Java запускается 100 раз и полученные результаты усредняются. При их исполнении измеряется RSS<sup>4</sup>. Результаты измерений для Huawei JDK, OpenJDK и Go будут представлены в следующих разделах.

---

<sup>4</sup>Resident set size (RSS) — количество памяти выделенной процессу операционной системой и в настоящее время находящейся в ОЗУ (RAM)



## 4.2 Переключение сопрограмм в Huawei JDK

В таблице 4.1 показаны результаты измерений скорости переключения сопрограмм Go и OpenJDK/Loom<sup>5</sup>. Видно, что сопрограммы Go выигрывают OpenJDK/Loom в скорости переключения.

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 первом прототипе для переключения контекста использовались функции из библиотеки glibc `getcontext` и `swarcontext`. Результат измерения первого прототипа приведен в таблице 4.2.

---

<sup>5</sup>Все измерения проводились на операционной системе Ubuntu, kernel 4.15, Intel Core i7-8700, 4.6 ГГц, 32 Гб ОЗУ

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 `swarcontext` показал, что переключение можно ускорить. Она используется для переключения потоков в операционных системах на ядре Linux[13]. 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[14], в чем нет необходимости при переключении контекста сопрограмм. Это один из факторов, побудивший реализовать аналоги функций `getcontext`, `swarcontext` внутри Huawei JDK, которые бы учитывали особенности виртуальной машины. На таблице 4.3 представлено сравнение результатов скоростей переключения.

Таблица 4.3 – Сравнение числа переключений

<i>Шм.</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 сопрограме. 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 границы стеков всех сопрограмм в некотором буфере. Но в отличие от потоков, нужно еще сохранять регистры приостановленных сопрограмм для корректной сборки.

#### 4.4 Потребление памяти

После реализации прототипа были проведены измерения потребления физической памяти.

Таблица 4.4 – Измерение потребления физической памяти

<i>Шт.</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, Huawei JDK обходит OpenJDK/Loom 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>6</sup> в памяти во время выполнения процесса.

---

<sup>6</sup>Jist-in-time (JIT) компиляция - компиляция программы во время ее исполнения.

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>

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.

#### 4.5 Применение сопрограмм в вычислительных задачах

Параллельные системы обработки данных играют большую роль в вычислительных задачах. Поэтому было бы правильным решением проверить применимость сопрограмм для решения такого рода задач.

Для этого был модифицирован фрагмент библиотеки Colt Parallel, отвечающий за многопоточное перемножение матриц больших размеров (свыше 1000 элементов) и вычисление дискретного преобразования Фурье. В библиотеке выделена отдельная часть, которая отвечает за параллельное выполнение задач. Она была переписана с использованием сопрограмм OpenJDK/Loom. В таблице 4.7 представлены результаты измерения времени перемножения двух матриц размером 6000x7000 и 7000x6000 соответственно. Каждый элемент имеет тип double. Замеры проводились на операционной системе CentOS ядро версии 4.18.0 с процессором Intel® Xeon® Gold 6130 с тактовой частота которого 2.10 ГГц.

Таблица 4.6 – Время перемножения матриц.

Потоки	Сопрограммы
45.6 $\pm$ 1.8 сек.	24.0 $\pm$ 5.6 сек.

Как видно, сопрограммы имеют лучший результат, чем потоки. Это можно связать с тем, что оригинальный алгоритм плохо оптимизирован. Ожидалось, что время перемножения матриц на потоках и сопрограммах будет одинаковым в пределах погрешности, как это вышло в задаче вычисления дискретного Фурье преобразования.

Таблица 4.7 – Время вычисления дискретного преобразования Фурье.

Потоки	Сопрограммы
77.2 $\pm$ 0.3 мсек.	77.4 $\pm$ 0.4 мсек.

Переделать фрагменты библиотеки Colt Parallel на сопрограммы оказалось гораздо проще, чем ожидалось. С точки зрения программирования, сопрограммы и потоки операционной системы это одна и та же сущность. В библиотеке потоки были подменены на сопрограммы практически один к одному. Сложность заключалась лишь в том, что алгоритмы ограничивали количество потоков числом доступных ядер, что не нужно в случае сопрограмм.



## 5 ВЫВОДЫ

В результате данной работы был проведен анализ реализаций сопрограмм в других языках, в частности Go, Kotlin, C#. Затем был реализован базовый прототип сопрограмм в Huawei JDK, который способен переключать контекст и поддерживает сборку мусора объектов, ссылки которых находятся на стеках сопрограмм. Затем было проведено измерение производительности с помощью ранее разработанных тестов. Созданный в рамках этой работы прототип сопрограмм обходит в скорости переключения OpenJDK в 3–8 раз и потребляет меньше физической памяти.

Прототип сопрограмм Huawei JDK еще не до конца реализован, что бы тот был применен в вычислительных задачах. Поэтому были применены сопрограммы из Loom/OpenJDK вместо потоков Java в вычислительных задачах на примере многопоточного перемножения матриц и вычисления дискретного Фурье–преобразования из библиотеки для научных расчетов Colt Parallel. Для этой цели были модифицированы соответствующие фрагменты этой библиотеки, которые отвечают за параллельную обработку. Измерения времен выполнения этих вычислительных задач показали, что применение сопрограмм в такого рода задачах возможно. Сопрограммы при перемножении матриц показали себя лучше, чем потоки, а в вычислении преобразования Фурье они имеют одинаковый результат в пределах погрешности. Поскольку уже сейчас Huawei JDK обходит в скорости переключения OpenJDK, то, предположительно, можно ожидать большего прироста производительности от применения сопрограмм, чем сейчас.

Проверка применимости сопрограмм в реальных системах обработки и сбора данных в рамках этой работы оказалась невозможной. Однако, проведенные измерения позволяют предположить, что благодаря в разы большей скорости переключения, применение сопрограмм Huawei JDK может дать существенный прирост пропускной способности по сравнению с потоками, а это особенно будет заметно с ростом числа каналов связи.

Итак, ключевыми отличиями сопрограмм от потоков являются меньшее потребление памяти, благодаря динамически увеличивающемуся стеку, и лучшая скорость переключения контекста из-за реализации их в среде исполнения языка, а не внутри ядра операционной системы.

В дальнейшем планируется усовершенствовать прототип, добавив возможности синхронизации сопрограмм, исполняемых разными потоками одновременно. Также следует реализовать вытеснение сопрограмм, которые инициируют блокирующие операции. Затем необходимо будет внедрить сопрограммы в реальную систему сбора и обработки данных для проверки их применимости. Реализация этого проекта на сопроγραμμαх позволит в дальнейшем получить точные значения пропускной способности, насколько сопрограммы будут полезны в конкретном таком случае.

## БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Melvin E. Conway. Design of a separable transition-diagram compiler // Communications of the ACM. – 1963. – Т. 6, № 7.
2. Вирт Н. Программирование на языке Модула-2. Перевод с англ. В. А. Серебрякова, В. М. Ходукина; Под ред. В. М. Курочкина. — М. : Мир, 1987. — 222 с.
3. David R. Butenhof. Programming with POSIX Threads. Addison-Wesley
4. ЛТ–компиляция [Электронный ресурс] // [https://docs.oracle.com/cd/E13150\\_01/jrockit\\_jvm/jrockit/geninfo/diagnos/underst\\_jit.h](https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/underst_jit.h)
5. Симула [Электронный ресурс] // <https://en.wikipedia.org/wiki/Simula>
6. Barbara Liskov. CLU Reference Manual (англ.) / edited by G. Goos, J. Hartmanis. — Berlin — Heidelberg — New York: Springer-Verlag, 1981.
7. Библиотека Colt Parallel [Электронный ресурс] // <https://sites.google.com/site/piotrwendykier/software/parallelcolt>
8. Спецификация языка Go. [Электронный ресурс] // <https://golang.org/ref/spec>
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.
12. Язык программирования Go. : Пер. с англ. — М. : ООО “И.Д. Вильямс”, 2016. — 432 с.
13. Роберт Лав. Ядро Linux: описание процесса разработки = Linux Kernel Development. — 3-е изд. — М.: Вильямс, 2012. — 496 с.

14. Роберт Лав. Сопрограммы и фиберы // Linux. Системное программирование. — 2-е изд.. — СПб.: Питер, 2014. — С. 261. — 448 с.
15. Официальный сайт набора библиотек Go-HEP [Электронный ресурс] // <https://go-hep.org/>
16. Bruce Belson. A Survey of Asynchronous Programming Using Coroutines in the Internet of Things and Embedded Systems // ACM Transactions on Embedded Computing Systems. – 2019.
17. Язык программирования C# [Электронный ресурс] // <https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/>
18. Venkat Subramaniam. Programming Kotlin. Book version: P1.0—September 2019
19. Сопрограммы в C++20 [Электронный ресурс] // <https://en.cppreference.com/w/cpp/language/coroutines>
20. Документация ядра linux [Электронный ресурс] // <https://www.kernel.org/doc/html/latest/>
21. Дорот В. Л. Вытесняющая многозадачность // Толковый словарь современной компьютерной лексики. — 3 изд.. — БХВ-Петербург, 2004. — С. 143. — 608 с.
22. Янг С. Алгоритмические языки реального времени. Перевод с англ. Л. В. Ухова; Под редакцией В.В Мартынюка — М. : Мир, 1985.