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

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 предано 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.
– Выявить ключевые плюсы 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 – памяти и процессорного времени. 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]. Модель программирования, которая позволяет нескольким потокам выполняться в рамках одного экземпляра исполняемой программой называется многопоточностью. Потоки в контексте одного процесса способны совместно использовать его ресурсы, но могут работать и независимо от него. 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 добится 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 контр интуитивного поведения. Что касается планирования, то в

современных операционных системах общего назначения как правило потоки планируются посредством вытесняющей многозадачности. Она предполагает, что операционная система принимает решение о переключении между задачами по истечении кванта времени, выделенного на выполнение задачи. В такой системе каждой выполняемой задаче дается приоритет, в зависимости от которого принимается решение о планировании. В отличие от кооперативной многозадачности, управление операционной системе передаётся 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.

В современных реалиях количество потоков 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 предано другому такому модулю. При этом будет сохранено полное состояние сопрограммы: включая стек, значения регистров и счётчик команд.

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, C++, Java, она не была применена. Если нужна была альтернатива сопрограммаам, то использовались потоки. В случае языков Си и C++ 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[1] 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 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.

[1] Контекст – регистры и указатель на стек потока или сопрограммы.

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 переключится на выполнение другого потока, запланированного ядром операционной системы. Сопрограммаами 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 возможно техника блокирование сопрограмм даст прирост производительности.

г) В отличии 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 следущих 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 сопрограммаами, так 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 одином 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 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()
}
```

Эта программа 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 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 с 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 переключение контекста сопрограммы в языке Go похож на переключения 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 отличии 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[2], 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 раннюю версию для написания тестовый программ, но в реальных проектах применять не стоит.

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 потока ”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 небольшими затратами усилий.

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

[2]Java Development Kit (JDK) - разработчика приложений на языке Java, содержащий в себе базовый набор программ, необходимых для написания программ.

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 получат 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 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 созданно 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, строится 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 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 раннее, 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, Huawei JDK с модулем сопрограмм должен проходить набор тестов JCK [3].

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 необходимо

[3] Java Compatibility Kit (JCK) - набор тестов на совместимость с Java.

разработать набор тестов на производительность сопрограмм в управляемых средах Go, OpenJDK/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 определить, насколько скорость переключения и потребление памяти лучше у сопрограмм. Может быть, полученные значения будут равными в пределах погрешности, и все отличия, перечисленные в обзоре области, не будут играть роли в реальных программах?

В дальнейшем разработанные тесты можно будет использовать для сравнение производительности сопрограмм в Huawei JDK от OpenJDK и Go. Так же стоит применить сопрограммы для решения вычислительных задач. Для этого потребуется модифицировать библиотеку Colt Parallel, добавив туда возможность производить вычисления с помощью сопрограмм. Поскольку эта библиотека довольно велика, то нет смысла изменять ее всю. Достаточно применить сопрограммы в алгоритме многопоточного перемножения матриц и дискретного Фурье преобразования, поскольку эти примеры будут довольно показательными в плане применимости сопрограмм.

В итоге, полный список задач, поставленных для достижения поставленной цели выглядит следущим 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[4]. 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[5]. Видно, что сопрограммы из языка Go выигрывают OpenJDK в скорости переключения.

[4]Resident set size (RSS) — размер памяти, выделенных процессу операционной системой и в настоящее время находящееся в ОЗУ (RAM)

[5]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

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

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

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

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

системах на ядре Linux. 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 swapcontext 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, swapcontext внутри Huawei JDK, которые бы учитывали особенности виртуальной машины. На таблице 4.3 представлено сравнение результатов скоростей переключения.

Таблица 4.3 – Сравнение числа переключений

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

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

4.3 Сборка мусора

Следущим 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 отличии 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

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

Как видно из таблицы 4.4, Huawei JDK 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[6] в памяти во время выполнения процесса.

[6]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.

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

4.5 Применение сопрограмм в вычислительных задачах

Параллельные системы обработки данных играют большую роль в вычислительных задачах. Потому 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 можно выделить отдельную часть, которая отвечает за параллельное выполнение задач. Она же и была переписана с использованием сопрограмм, доступных в ранней версии проекта ”Loom”. В таблице 4.7 представлены результаты измерения времени перемножения двух матриц размером 6000x7000 и 7000x6000 соотвественно. Каждый элемент имеет тип double. Замеры проводились на

операционной системе CentOS ядро версии 4.18.0 с процессором Intel (R) Xeon(R) Gold 6130, частота которого 2.10 ГГц.

Таблица 4.6 – Время перемножения матриц.

Потоки	Сопрограммы
18.3 ±0.3 сек.	24.0 ±5.6 сек.

Как видно, потоки имеют лучший результат, чем сопрограммы. Это связано с тем, что при перемножении матриц потоки не приостанавливаются при блокирующих операциях и не взаимодействуют между собой. Другими словами, в этой задаче не осуществляется частого переключения потока управления, в чем сопрограммы имеют преимущество перед потоками как это было показано ранее.

Таблица 4.7 – Время вычисления дискретного Фурье преобразования.

Потоки	Сопрограммы
77.2 ±0.3 мсек.	77.4 ±0.4 мсек.

Однако в задаче вычисления Фурье преобразования время выполнения программы на сопрограммаах и потоках одинаково в пределах погрешности.

Переделать фрагменты библиотеки Colt Parallel на сопрограммы оказалось гораздо проще, чем ожидалось. С точки зрения программы, сопрограммы и потоки операционной системы это одна и таже сущность. Потому в библиотеке потоки были подменены. Сложность заключалась лишь в том, что алгоритмы ограничивали количество нитей числом доступных ядер, что не нужно в случае сопрограмм.

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. Был реализован базовый прототип сопрограмм в HuaweiJDK, 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.

Наконец была изучена применимость сопрограмм 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 модифицированны соответствующие фрагменты этой библиотеки, которые отвечают за параллельную обработку. В них потоки заменены на сопрограммы практически один к одному. Измерения времен выполнения тестов показали, что применение сопрограмм в такого рода задачах с целью повышения производительности не имеет смысла. Потоки при перемножении матриц показали себя лучше, чем сопрограммы, а в вычислении преобразования Фурье они имеют одинаковый результат в пределах погрешности.

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 иници-

ируют блокирующие операции. Это поможет избежать лишнего простоя потока, выполняющих код сопрограмм.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Сопрограммы [Электронный ресурс] https://en.wikipedia.org/wiki/Coroutine

2. Модула-2 [Электронный ресурс] https://en.wikipedia.org/wiki/Modula-2

3. Поток [Электронный ресурс] https://ru.wikipedia.org/wiki/Поток_выполнения

4. JIT–компиляция https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/ger

5. Симула [Электронный ресурс] https://en.wikipedia.org/wiki/Simula

6. Язык программирования CLU [Электронный ресурс] 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

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.