**Stichpunkte**

**Folie 1: Titel**

Herzlich willkommen zu meinem Kolloquium zu Meiner Masterarbeit mit dem Titel:

Leistungsanalyse von Thread-Abstraktionen in verschiedenen Programmiersprachen

**Folie 2: Inhaltsverzeichnis**

Aufbau des Vortrags:

Orientiert sich am Aufbau der Arbeit

1. Ziel der Masterarbeit
2. Thread-Abstraktionen:

was versteht man darunter, wie wurden sie in den Programmiersprachen umgesetzt

1. Testaufbau
2. Benchmark 1 Mergesort
3. Benchmark 2 Bank
4. Fazit

**Folie 4: Ziel der Masterarbeit**

Umfassende Leistungsanalyse von Thread-Abstraktionen in Java, Kotlin und Go

Anlass: Veröffentlichung von Virtual Threads in Java, im September 2023

Auswahl Sprachen: sollten vergleichbar sein, Garbage Collectet, keine Skript Sprachen

Kotlin Coroutinen direkter Konkurrent in der JVM

Go Goroutinen wurde von Anfang an mit dem Fokus auf Performance und Nebenläufigkeit entwickelt, guter Referenzpunkt

Erste Hälfte der Arbeit:

Theoretischer Teil

Ziel: tiefes Verständnis Funktionsweise der Thread-Abstraktionen

Überlegung: Auswirkung auf Gesamtperformance und Effizienz von Anwendungen

Zeite Hälfte der Arbeit:

Praktischer Teil

Entwicklung und Durchführung von Benchmarks

Testen der Thread-Abstraktion

Fokus auf: Effizienz, Skalierbarkeit, Ressourcennutzung

Abschließend:

Zusammenführen der Erkenntnisse aus den zwei Teile

Praktische Empfehlungen für Entwickler, für die Auswahl geeigneter Thread-Abstraktionen

**Folie 6: Was sind Thread-Abstraktionen:**

Verweis: Kapitel 2 Masterarbeit

Thread-Abstraktionen sind Konzepte zur Vereinfachung der nebenläufigen Programmierung, die eine **Abstraktionsebene** über den traditionellen Betriebssystem-Threads bieten.

Traditionelle Thread-Verwaltung

* 1:1-Mapping zwischen Anwendungs-Threads und BS-Threads
* Threads werden direkt vom Betriebssystem verwaltet
* Hoher Ressourcenverbrauch pro Thread (2-3 MB Speicher)
* Ineffizient bei I/O-intensiven Aufgaben durch Blockieren von BS-Threads

Moderne Thread-Abstraktionen

* N:M- Mapping: N leichtgewichtige Threads auf M Betriebssystem-Threads
* Verwaltung durch die Laufzeitumgebung der Programmiersprache
* Geringerer Ressourcenverbrauch pro Thread (wenige KB Speicher)
* Effizientere Nutzung von Systemressourcen, besonders bei I/O-lastigen Anwendungen
  + Z.B. Netzwerkkommunikation, Lese-/Schreib-Zugriffe, Nutzereingaben

**Folie 7: Platform Threads**

Seit der Beginn Teil von Java

1995, etabliert und weit verbreitet

Schwergewichtige Threads

Verbrauchen viel Speicher

Dauern lange zu erstellen

wegen

1:1 Mapping

Pro PT ein BS-Thread

Wrapper, umschließen BS-Thread

Verwaltung auf Betriebssystemebene

Wenn Platform Thread blockiert, blockiert auch der BS-Thread

z.B. IO

auch BS-Thread muss raus aus dem Kern

**Folie 8: Virtual-Threads**

Leichtgewichtige Thread-Implementierung

Geringer Ressourcenverbrauch (wenige KB pro Thread)

Können schnell erstellt werden

Viele Virtual Threads teilen sich wenige Platform Threads (Carrier Threads)

M:N Mapping

Carrier Threads: Anzahl Prozessor Kerne

Millionen Virtual Threads möglich

Nicht-blockierend bei I/O-Operationen

VT blockiert, aber nicht PT

Verwaltung durch die Java Runtime

VT pro Aufgabe, bleiben die ganze Zeit verbunden

Mount/Unmount auf PT

Dafür JDK erweitert anstellen, wo Code Potenziell blockieren kann

Leichter Umstieg durch gleiche API wie Platform Threads

**Folie 9: Kotlin Coroutinen**

Leichtgewichtige Nebenläufigkeitsabstraktion

Eigene Syntax für das Erstellen und verwalten von Coroutinen:

Builder und Dispatcher: definieren die Aufgabe der Coroutine

z.B. launch, asynch

z.B. Default (rechenintensiv) IO

Basieren auf Suspendierbarkeit von Funktionen

Schlüsselwort: suspend

Continuations:

Repräsentieren den Zustand einer suspendierten Coroutine

Ermöglichen das Pausieren und Fortsetzen von Coroutinen

Ist Objekt

Werte von Variablen

An welcher Stelle vom Code wird die Ausführung unterbrochen

**Folie 10: Go Goroutine**

Rob Pike hat Go entwickelt

Unter andrem mit Ken Thompson

Eine Goroutine ist eine:

**Independently executing function, launched by a go statement.**

Schlüsselwort: go

Integraler Bestandteil der Go-Sprache

Extrem leichtgewichtige Threads

Verwaltung durch Go Runtime Scheduler (M:N Scheduling)

Herzstück für Nebenläufigkeit

Warteschlangen

Verteilt G auf logische P weißt BS Threasds M zu

Warteschlangen

Workstealing

Kommunikation über Channels (Synchronisationspunkte)

Daten an die Goroutine zu senden oder zu empfangen

Dienen als Synchronisationspunkte

Programmiermuster für effiziente Verwendung nötig

**Folie 12: Testaufbau**

Kapitel 3 Masterarbeit

Hardware:

Prozessor : 8 logische Prozessoren

Performancefaktoren:

Ausführungsdauer

Möglichst kurz

Wie skaliert bei steigender Last

CPU-Auslastung

Möglichst hohe auslastung

Bedeutet effiziente Ressourcennutzung

Arbeitsspeicherverbrauch

Möglichst niedrig

Thread-Anzahl

BS-Threads

Möglichst niedrig sein

Maximale Anzahl begrenzt

Zeigt ob Entkoppelung von leichtgewichtigen Threads funktioniert

Werkzeuge zur Datenerhebung:

Selbst entwickelte Python-Skripte

Einheitliche Messung möglich

Einheitlicher Ressourcenverbrauch

Einheitliche Messdaten

Werkzeug für Visualisierung

Alle Implementierung können mit einem Start nacheinander ausgeführt werden

**Folie 14: Benchmark 1: Mergesort**

Kapitel 4 Masterarbeit

Motivation: rekursiver Sortieralgorithmus eignet sich gut für Parallelisierung

Teile-und-herrsche-Verfahren

Implementierung:

Parallelisierung durch gleichzeitiges Sortieren

Der Teillisten

Maximale Baumebene: Schwelle, bis zu der die Teilsortierungen parallel ausgeführt werden

Ebene 1 gesamte Liste

Ebene 2 halbiert in 2 Teillisten

Ebene 3 wieder halbiert in 4 Teillisten

Usw.

Besonderheiten der Programmiersprachen

Java: Code für Platfrom Threads und Virtual Threads fast identisch,

nur unterschiedliche Executor Services

CachedThreadPool

gegen

VirtualThreadPerTaskExecutor

Kotlin: Verwendung von GlobalScope

So keine strukturierte Nebenläufigkeit

Hat Ausführungszeit verdoppelt

Go: Verwendung vom „Closing Chanel“-Muster

**Folie 15: Mergesort Messung 1**

Ist Messung 1 aus der Masterarbeit

Besonderheit:

Maximale Baumebene: Werte 1-6

Also wenige Threads werden Parallel ausgeführt

**Folie 16: Graphen Mergesort Messung 1**

Rot: Platform Threads

Orange: Virtual Threads

Blau: Coroutinen

Grün: Goroutinen

Oben Links: CPU-Auslastung in Prozent

Für maximale Baumebene

1: 12,5% 1/8

2: 25% 2/8

…

Ab 4 stagnieren die Werte, weil alle Kerne genutzt werden

Insgesamt alle Implementierungen dicht bei einander

Oben rechts: Ausführungszeiten

Gehen runter mit steigender Parallelisierung

Ab 4 stagnieren die Werte wieder

Coroutinen brauchen am längsten

VT und PT gleich auf

Goroutinen am schnellsten

Unten Mitte:

Thread Anzahl:

Bleibt konstant für neue Thread Abstraktionen

Steigt exponentiell für PT

Wegen 1:1 Mapping

**Folie 17: Mergesort Fazit**

Benchmark für Rechenlast

Alle vier Implementierungen skalieren ähnlich bei CPU-Auslastung und Ausführungsdauer

Entkoppelung von BS-Threads funktioniert

Bei einem vielen Threads die häufig blockieren neue Thread-Abstraktionen besser

Wenn sich mit wenigen Threads eine optimal CPU-Auslastung erreichen lässt, sind die neuen Technologien genau so gut wie die alten

Gedankenspiel: Benchmark wäre leicht so zu konstruieren gewesen das neue Thread-Abstraktionen besser abschneiden

Einfach listenlänge langsam steigern

Für jede Teilliste neuen Thread

Keine maximale Baumebene

Wäre nicht ehrlich gewesen

**Folie 19: Benchmark 2 Bank**

Zweite Hälfte Kapitel 4 Masterarbeit

Motivation: Realitätsnahes Szenario, das aktuelle Technologien nutzt,

blockierende I/O-Operationen

realitätsnah unter dem Aspekt verwendet Technologien

nicht realistische Bank

Technologien: PostgreSQL, Datenbanktreiber, Datenbankverbindungspools, Docker

PostgreSQL-Datenbank: Tabelle mit Konten (Kontonummer, Kontostand)

Läuft als Docker-Container

Überweisungen: Betrag wird von einem Konto auf ein anderes gebucht

Anforderung: Datenkonsistenz (Zeilensperre nötig)

Docker: Zuweisung von Hardwareressourcen

Auch Implementierungen laufen in Containern

CPU-Implementierung und CPU-PostgreSQL

Implementierung:

Java Platform/Virtual Threads: JDBC PostgreSQL-Treiber, HikariCP

Unterstützt Virtual Threads

Kotlin: R2DBC Treiber und Verbindungspool

Einzige Option die Coroutinen unterstützt

Reactive Ansatz

Go: pgx Treiber und Verbindungspool

Gilt als extrem performant

Nutzt den Net-Poller von Go

**Folie 20: Bank Messung 2**

Messung 2 aus der Masterarbeit

Besonderheit Überweisungen wurden Datenbankseitig verzögert

Zwischen 10-100 ms

pg\_sleep

80 Datenbank-Verbindungen im Pool

Letztendlich limitierender Faktor

**Folie 21: Bank Graphen**

Farben sind gleich

Oben links:

Ausführungszeiten

Steigen mit der Verzögerung der Anfragen

Entkoppelung war nicht möglich war

Bzw. nur 80 Transaktionen gleichzeitig ausgeführt werden konnten

Oben rechts:

Anzahl Threads

Entkoppelung auf Thread ebene hat funktioniert

Coroutinen pegeln sich zwischen 80-100 Threads ein, vermutlich Reactive-Ansatz

PT fixed Thread Pool

Unten Mitte:

CPU-Auslastung

Niedrig außer bei Coroutinen

Deshalb feste Hardwareressourcen mit Docker

**Folie 22: Bank Fazit**

In der Praxis: Performance von Thread-Abstraktion von verwendeten Technologien abhängig

Datenbank

Datenbanktreiber

Datenbankverbindungspool

Entkoppelung von Aufgaben (Überweisungen) und Ressourcen (Datenbankverbindungen, Threads) auf allen Ebenen nötig

Selbst wenn Technologien scheinbar Thread-Abstraktion unterstützen

Aktuell geringe Latenz wichtiger als hoher Grad an Parallelität

z.B. üblich HikariCP auch für Kotlin, weil sehr performant

**Folie 24: Fazit**

Theoretischer Teil:

Alle drei neuen Thread-Abstraktionen nutzen ähnlich Ansätze

Zusätzliche Abstraktionsebene durch Leichtgewichtige Threads

Entkoppelung von Betriebssystem-Threads

Scheduling auf Laufzeiteben

Praktischer Teil:

Stärken gar nicht so leicht auszuspielen

Mergesort: Nur besser wenn optimale CPU-Auslastung noch nicht erreicht ist

Bank: Alle verwendeten Technologien müssen das Konzept unterstützen

Performance:

neue Thread-Abstraktionen scheiden genau so gut oder besser ab

Worst-Case 1:1 Mapping

Sehr gut bei vielen Threads die lange blockieren

Goroutinen mit Abstand am performantesten

Dennoch Virtual Threads besser als Coroutinen