UNIKASSEL
VERSITÄT

University of Kassel

Bachelorarbeit

Benchmarking von Virtuellen Threads in Java 19

presented to
Department for Electric Engineering/Computer Science
Research Group Programming Languages/Methodologies

Marco Spöth
35197913

Kassel, 9. März 2023

Examiners:
Prof. Dr. Claudia Fohry
Prof. Dr. Oliver Hohlfeld

Inhaltsverzeichnis

Verwendete Begriffe

jThread
(Kurz für: Java Thread)

Repräsentation eines OS-Threads in der JVM. jThreads sind in Java über die Klasse `java.lang.Thread` nutzbar, welche eine Wrapper-Klasse für OS-Threads darstellt. Jeder jThread wird auf einem OS-Thread ausgeführt (One-to-One Modell). jThreads haben verschiedene Zustände. Die beiden für diese Arbeit relevanten Zustände sind:

- *Running*: Der jThread berechnet einen Task.
- *Waiting*: Der Task eines jThreads wartet auf ein Ereignis, z. B. das Ende eines anderen Tasks.

OS Thread
(Auch: Nativer Thread)

Ressource des Betriebssystems zur Lösung nebenläufiger Aufgaben.

Task

Teil einer Programmausführung, der unabhängig zu anderen Tasks ist.

Thread-per-Task

Programmierparadigma bei dem für jeden zu berechnenden Task ein Thread gestartet wird, der diesen ausführt.

vThread
(Kurz für: Virtueller Thread)

Leichtgewichtige Alternative zu jThreads mit den selben Funktionalitäten. Beliebig viele

vThreads können auf beliebig vielen OS-Threads gestartet werden (Many-to-Many Modell). Auf OS-Threads werden die vThreads über Carrier-Threads ausgeführt. Ein Carrier-Thread pro OS-Thread, aber viele vThreads pro Carrier-Thread.

Statutory Declaration

I declare on oath that I completed this work on my own and that information which has been directly or indirectly taken from other sources has been noted as such. Neither this, nor a similar work, has been published or presented to an examination committee.

_____

Kassel, 9. März 2023

Marco Spöth

1 Einleitung

In nebenläufigen Programmen wird eine große Aufgabe in kleinere Teilaufgaben, die *Tasks*, aufgeteilt. Ein Task bezeichnet einen Teil einer Gesamtaufgabe, der parallel zu den anderen Tasks berechnet werden kann. Anwendungen aus Bereichen wie Backend Server oder High-Performance-Computing müssen eine große und wachsende Anzahl von Tasks ausführen.

Typischerweise gibt es ein Limit, wie viele Tasks gleichzeitig ausgeführt werden können. In Java ist der limitierende Faktor häufig die Anzahl der hardwareseitig verfügbaren Prozessorkerne. Auf den Kernen werden Tasks mithilfe von *OS-Threads* ausgeführt. OS-Threads sind zueinander nebenläufig, können jeweils aber immer nur einen Task gleichzeitig bearbeiten. Sobald alle OS-Threads einen Task bearbeiten, können keine weiteren Tasks mehr ausgeführt werden.

In Java kann nicht direkt auf OS-Threads zugegriffen werden. Um OS-Threads in der JVM zu verwenden, werden *jThreads* verwendet. Die jThreads sind Wrapper für OS-Threads. Eine Übersicht über jThreads ist in Abb. 1.1 dargestellt. In Java wird bei dem Start eines jThreads, ein Task übergeben, der auf einem OS-Thread ausgeführt werden soll. Ein jThread gilt als „laufend“, falls der Task des jThreads aktuell auf einem OS-Thread ausgeführt wird und als „wartend“, falls eine *blockierende Anweisung* ausgeführt wurde und die gesamte Ausführung pausiert.

Wenn ein jThread einen Task berechnet, wird der OS-Thread vollkommen ausgelastet. Das Betriebssystem kann einen laufenden jThread unterbrechen und diesen durch einen anderen, nicht laufenden jThread ersetzen. Sind alle OS-Threads belegt, kann kein weiterer jThread ausgeführt werden, bis mindestens ein laufender jThread seine Berechnung beendet oder ausgetauscht wird. Dieser

Austauschprozess wird *Preemption* genannt. Wann eine Preemption stattfindet oder welcher jThread berechnet wird, entscheidet das Betriebssystem über ein *Scheduling*.

Das verwendete Scheduling ist betriebssystemabhängig. Innerhalb der JVM hat der Entwickler keine Informationen oder Kontrolle über den Austausch von jThreads. Das Starten und das Wechseln von jThreads kostet Zeit und jeder jThread, egal ob er aktuell läuft oder nicht, verbraucht Speicher. Entwickler sollten darauf achten, nicht zu viele jThreads zu starten, um die Laufzeit der Anwendung und den Speicherverbrauch gering zu halten.

Für jThreads wurden verschiedene Techniken entwickelt, um die Anzahl der zu startenden jThreads zu minimieren und die vorhandenen OS-Threads möglichst optimal auszulasten. Um das Starten neuer jThreads zu vermeiden, werden für neue Tasks keine neuen jThreads gestartet. Gestartete jThreads werden in einer Liste, dem *Thread Pool*, gespeichert. Wenn ein Task berechnet werden soll, wird dieser an den Thread Pool übergeben, der den Task von einem freien jThread ausführen lässt. (Dieses Thema wird in Abschnitt 2.1 vertieft).

Project Loom entwickelt *vThreads* als Alternative zu jThreads. Der Begriff steht für „virtual thread“ und steht Entwicklern als Ergänzung zu jThreads zur Verfügung. Ein vThread ist „virtuell“, weil seine Implementierung vollständig von der Hardware getrennt ist. vThreads sind ausschließlich in der JVM implementiert und sind keine Wrapper für OS-Threads. Um vThreads in OS-Threads auszuführen, werden *Carrier-Threads* verwendet. Wie in Abb. 1.1 dargestellt, hat jeder OS-Thread einen Carrier-Thread und jeder Carrier-Thread hat mehrere vThreads.

Im Gegensatz zu jThreads sind vThreads unabhängig vom Scheduling des Betriebssystems und verfügen daher über keine Preemption. Ein vThread wird jedoch durch einen anderen ersetzt, sobald dieser vThread eine blockierende

Anweisung ausführt. Während ein wartender jThread den OS-Thread voll auslastet, kann bei vThreads der OS-Thread einen anderen vThread berechnen. Das Starten und Wechseln von vThreads erfolgt ausschließlich durch ein Scheduling innerhalb der JVM. Das Starten und Wechseln innerhalb der JVM ist schneller und zusätzlich wird durch die dynamischen Stacks der vThreads weniger Speicher in Anspruch genommen.

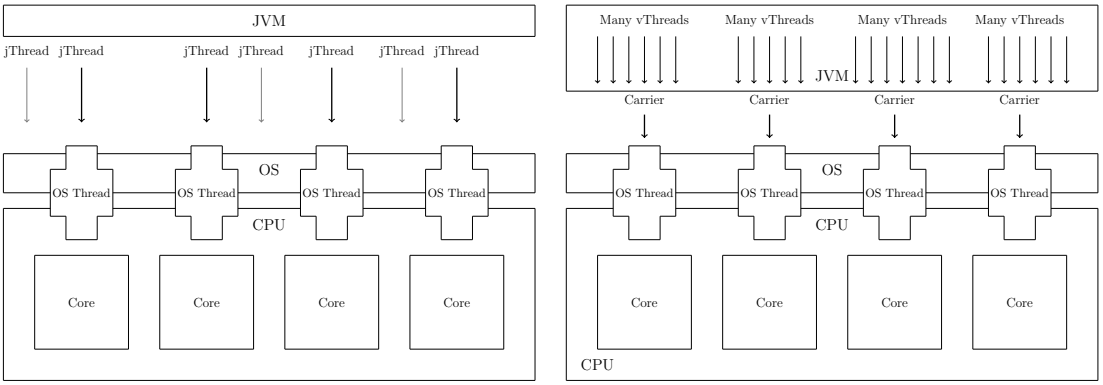
Im Gegensatz zu jThreads kann der Entwickler nahezu beliebig viele vThreads starten, ohne Bedenken für Systemressourcen. Auf diese Weise ist es möglich, in nebenläufigen Anwendungen für jeden Task einen eigenen vThread zu starten (*Thread-per-Task*). Es gibt aber auch Nachteile. So können lange laufende vThreads, ohne blockierende Anweisung, nicht ausgewechselt werden. Solange alle Carrier-Threads Tasks ausführen, welche nicht warten, können keine weiteren vThreads berechnet werden.

Wenn in Tasks blockierende Anweisungen ausgeführt werden, werden diese Tasks als *IO-gebunden* bezeichnet. Sind Tasks stattdessen ausschließlich mit Rechenarbeit beschäftigt, wird dies als *CPU-gebunden* bezeichnet. IO-gebunden beinhaltet nicht ausschließlich Ein- und Ausgaben von Nutzern, sondern auch Anweisungen wie das Warten auf Ereignisse.

Das Scheduling von jThreads führt dazu, dass jeder jThread während seiner gesamten Lebensdauer einen OS-Thread beschäftigt. Ein vThread beschäftigt nur dann einen OS-Thread, während der vThread nicht wartet. Für IO-gebundene Tasks führen vThreads zu einer deutlich besseren Performance.

Abbildung 1.1: Eine Übersicht der beiden Thread Implementierungen im Verhältnis zu den Systemressourcen.

Links: jThreads. Rechts: vThreads



In Java 19 werden die vThreads erstmals offiziell als Preview Feature verfügbar sein. Das Ziel dieser Bachelorarbeit ist es, die Ausführung von Tasks über vThreads und jThreads zu vergleichen. Es wird untersucht, unter welchen Umständen vThreads zu besserer Performance führen als jThreads und für welche Anwendungen sich der Einsatz von vThreads lohnt. Die Tests wurden mit eigenen Benchmarks und einer eigenen Ausführungsumgebung auf dem NV-Rechner der Universität Kassel durchgeführt.

Die Ergebnisse der Benchmarks aus Kapitel 5 zeigen, dass die aktuelle Implementierung der vThreads noch nicht optimal ist. Dennoch sind die vThreads bereits in Java 19 manchen Fällen schneller als die jThreads und sind in den schlechtesten Fällen von der Laufzeit vergleichbar schnell.

In Kapitel 2 werden jThreads und ihre Bedeutung in Java näher erläutert. Anschließend werden Project Loom und die vThreads erläutert. Kapitel 3 stellt die in dieser Arbeit verwendeten Benchmarks vor und Kapitel 4 deren Implementierung. Die Performanceergebnisse werden in Kapitel 5 dargestellt, interpretiert und anschließend diskutiert. Schlussfolgerungen und Ausblick auf weitere Entwicklungen mit Project Loom folgen in Kapitel 6.

2 Hintergrund

2.1 jThreads

Wie in der Einleitung erwähnt, werden jThreads mit einem Task gestartet. Tasks werden in der JVM als Instanzen der `Runnable` bzw. `Callable<V>` Interfaces definiert und dem Thread bei seiner Instanziierung übergeben. Statt eigener Interface-Implementierungen werden die Interfaces meist durch Lambda-Ausdrücke definiert. Die folgenden Codezeilen zeigen den Start eines einzelnen jThreads:

```
1        Thread platformThread = new Thread(() -> doWork()); // Create thread
2        platformThread.start(); // Start Thread
3        platformThread.join(); // Wait for execution to finish
```

Listing 2.1: Starten eines jThread

In Zeile 1 wird der jThread instanziiert, sein Typ ist `Thread`. Das `() -> doWork()` ist der Lambda-Ausdruck für den Task der über den jThread ausgeführt wird. Der Aufruf der Methode `start()` auf dem jThread startet die Ausführung und `join()` in der folgenden Zeile lässt die Anwendung auf die Beendigung des jThreads warten.

Um zu vermeiden, dass mehr jThreads gestartet werden, als OS-Threads zur Verfügung stehen, werden Maßnahmen wie *Work-Stealing* verwendet. Bei Work-Stealing werden jThreads von einem Thread Pool gestartet und verwendet. Im Kontext von Work-Stealing werden die jThreads im Thread Pool als Worker bezeichnet und haben jeweils eine Queue von Tasks. Wenn ein Task ausgeführt werden soll, wird dieser an den Thread Pool übergeben, der diesen Task an das Ende der Queue eines zufälligen Workers stellt. Die Worker nehmen sich

fortlaufend einen Task aus der zugehörigen Task-Queue und berechnen diesen. Hat ein Worker keine Tasks zu bearbeiten, so nimmt er sich den nächsten Task aus der Queue eines zufälligen anderen Workers. Auf diese Weise wird die Auslastung der Worker gleichmäßig unter ihnen verteilt.

Ein Thread Pool kann selbstständig jThreads starten und beenden. Dabei wird bevorzugt, dass nicht mehr jThreads gestartet werden als OS-Threads verfügbar sind. So müssen keine jThreads eingewechselt werden und es kommt nicht zu Preemption.

In Java wird ein `ExecutorService` verwendet um Tasks auf jThreads ausführen zu lassen. Dabei handelt es sich um ein Interface, welches einen einheitlichen Umgang für den Entwickler gewährleistet. Über die Funktion `submit()` wird einem *ExecutorService* ein Task zur Ausführung übergeben. Java bietet verschiedene Implementierungen dieses Interfaces an, die definieren über welche Mechanismen jThreads gestartet und verwendet werden. Ein `ExecutorService` muss, über einen Aufruf der Methode `close()`, geschlossen werden. Die Klasse `Executors` besitzt verschiedene Factory-Pattern Methoden, über die ExecutorServices instanziiert werden können. Die für diese Arbeit relevante Methode ist `Executors.newWorkStealingPool()`. Wie der Methodenname impliziert, handelt es sich bei dieser Implementierung um einen ExecutorService, der jThreads als Worker nach dem Work-Stealing-Prinzip startet und verwendet. Die Implementierung basiert auf einer Klasse namens `ForkJoinPool`.

```
1        ExecutorService executor = Executors.newWorkStealingPool();
2        executor.submit(() -> doWork());
```

Listing 2.2: Ausführen von Tasks über ExecutorService

Wenn ein Task über `submit()` übergeben wird, muss die Anwendung in der Lage sein, auf die Ausführung dieses Tasks warten zu können. Die Methode `submit()`

gibt dazu eine Instanz des Interface `Future<V>` zurück. Einer Instanz von `Future` ist ein Task zugeordnet, dessen Ausführung kontrolliert wird. Über `Future` kann der aktuelle Zustand des Tasks überprüft und der Task abgebrochen werden. Mit der blockierenden Anweisung `get()` wird auf die Berechnung des Tasks gewartet und anschließend der Rückgabewert des Tasks zurückgegeben.

```
1        Future<?> future = executor.submit(() -> doWork());
2        var result = future.get();
```

Listing 2.3: Warten über `get()`

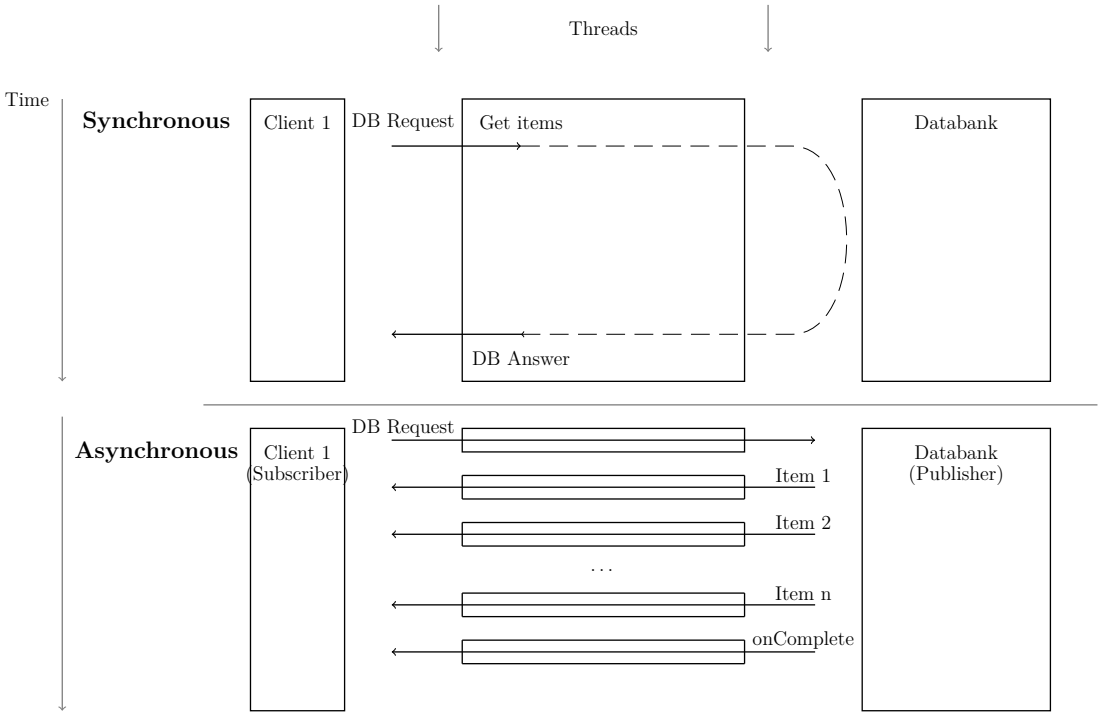
2.2 Probleme mit jThreads

Die jThreads sind für das Thread-per-Task-Modell nicht ausreichend. Wenn viele IO-gebundene Tasks gestartet werden, wird viel Laufzeit mit wartenden jThreads verbraucht. Während ein jThread wartet, wartet dieser meistens auf ein externes Ereignis, wie z. B. die Antwort auf eine Anfrage für Einträge einer Datenbank. Für den Sender der Anfrage ist der jThread und damit der gesamte OS-Thread blockiert, obwohl die Verarbeitung in der Datenbank stattfindet. Um diese Wartezeit bei dem Sender zu vermeiden haben Entwickler das Programmierparadigma der *asynchronen Programmierung* entwickelt.

Bei der asynchronen Programmierung startet der Sender keine Tasks mehr für die gesamte Datenbanktransaktion. Der Sender und die Datenbank werden in eine Subscriber-Publisher-Beziehung gesetzt. Der Sender (Subscriber) übergibt der Datenbank (Publisher) eine Benachrichtigung, dass eine Anfrage ausgeführt werden soll. Ein Task führt diese Benachrichtigung aus und erzeugt eine Subscription zwischen Subscriber und Publisher. Der Task endet und der Subscriber wartet nicht auf Ergebnisse. Über einzelne Tasks übergibt der Publisher die Ergebnisse der Anfrage (einzelne Datenbankeinträge) einzeln an

den Subscriber. Nachdem alle Ergebnisse einzeln übertragen wurden, wird ein abschließender Task vom Publisher ausgeführt, der den Subscriber darüber informiert, dass die Subscription abgeschlossen ist.

Abbildung 2.1: Übersicht über den Datenbankzugriff durch synchrone (oben) und asynchrone Programmierung (unten).
Gestrichelte Linien stellen den Thread im wartenden Zustand dar



Das Bild zeigt, wie bei synchroner Programmierung ein jThread für die gesamte Transaktion verwendet wird. Während der Transaktion wurde der OS-Thread des JThreads dauerhaft ausgelastet und nichts anderes konnte berechnet werden, obwohl der Großteil des Tasks nur aus Warten bestand.

Bei asynchroner Programmierung wird die Transaktion auf mehrere Tasks aufgeteilt, die jeweils von jThreads ausgeführt werden. Die Tasks werden nur für den Start der Subscription und den Datentransfer verwendet. Es werden keine blockierenden Anweisungen benötigt.

Der ursprüngliche Task wurde für die asynchrone Ausführung in mehrere Teil-Tasks aufgeteilt. Die Logik für die Subscription, die Auswertung der einzelnen Datenbankeinträge, die Fehlerbehandlung und die Gesamtauswertung am Ende müssen einzeln implementiert werden. Durch die Aufteilung aller Tasks können diese untereinander gemischt ausgeführt werden. So können z. B. zwischen den einzelnen Teilen einer Transaktion die Teile anderer Transaktionen ausgeführt werden.

Die asynchrone Programmierung hat jedoch mehrere Nachteile:

- Java bietet nativ wenige Mittel für asynchrone Programmierung und viele Mechanismen müssen komplett selbst implementiert oder über Libraries importiert werden.
- Die Aufteilung der Tasks erschwert die Programmierung. Entweder müssen für Tasks eigene Klassen geschrieben werden, die Interfaces implementieren oder Tasks werden durch mehrere, komplexe Lambda-Ausdrücke repräsentiert.
- Alle Teil-Tasks können von verschiedenen jThreads ausgeführt werden. Variablen und Informationen die diese Teil-Tasks sich teilen, müssen ausgelagert oder dupliziert werden.
- Eine geworfene Exception gibt dem Entwickler durch die Aufteilung keinen Überblick über den gesamten Task. Z. B. kann in Abb. 2.1, bei der asynchronen Benachrichtigung über einen Datenbankeintrag, eine Exception geworfen werden, aber der Entwickler bekommt keine Einsicht über den Zustand der restlichen Teile des Gesamttasks.
- Das Debuggen ist schwierig, wenn die einzelnen Task-Teile voneinander getrennt und auf verschiedenen jThreads ausgeführt werden.

- Aufgrund des Programmieraufwands und der unintuitiven Struktur wird der Code komplexer und schwieriger zu warten.

2.3 Project Loom

Das Ziel von Project Loom ist es, nebenläufige Anwendungen einfacher und lesbarer zu schreiben, zu debuggen und mit einem Profiler analysieren zu können. Das Projekt Loom wird von Ron Pressler geleitet.[1]

In Project Loom werden zwei zentrale Technologien entwickelt: Fibers und Structured Concurrency. Fibers ist für diese Arbeit relevant. Structured Concurrency befindet sich noch in einem frühen Entwicklungsstadium.

Fibers

Fiber war der ursprüngliche Name von vThreads. In Java 1.1 bis 1.3 gab es, vor den jThreads, die Green Threads (GT). Alle GTs liefen auf demselben OS-Thread wie die JVM selbst. Es konnte immer nur ein GT laufen, aber wenn dieser wartete, wurde er mit einem anderen GT ausgetauscht. Während nur ein GT laufen konnte, konnten beliebig viele andere warten. In der Performance waren GTs der frühen Implementierung der jThreads unterlegen, da jThreads echte Parallelität über OS-Threads besaßen.

Die vThreads sind eine Weiterentwicklung der Grundkonzepte von GT. Wie in der Einleitung erwähnt, werden vThreads vollständig von der JVM verwaltet und über Carrier-Threads auf OS-Threads ausgeführt. Im Vergleich zu laufenden jThreads ist der Speicherverbrauch in vielen Fällen deutlich geringer und der Zeitaufwand für das Wechseln zwischen aktiven vThreads geht gegen null. Die vThreads folgen dem Many-to-Many Modell: Auf m OS-Threads werden n virtuelle Threads

ausgeführt. Statt weniger jThreads, können Tausende bis Millionen vThreads verwendet werden.

Das Scheduling zwischen vThreads wird über eine Work-Stealing Implementierung (`ForkJoinPool`) gesteuert und bleibt somit innerhalb der JVM. Der Stack eines vThreads ist von dem Stack seines OS-Threads getrennt. Der Stack eines vThreads wird getrennt, indem der Stack als Objekt im Heap angelegt wird, als *„Trace Chunk“*. Die Größe des Trace Chunks ist dynamisch, da Trace Chunks nur so viel Speicher verbrauchen, wie sie gerade benötigen. Dies ermöglicht vThreads einen geringen Speicherverbrauch bei kleinen Tasks. Im Falle von Exceptions führen Trace Chunks zu kürzeren und verständlicheren Stack Traces. Zur Ausführung von Tasks müssen vThreads nicht gepoolt werden und können im Gegensatz zu den jThreads, für ein Thread-per-Task-Modell verwendet werden.

Die vThreads sind in der Regel bei Anwendungen mit vielen, IO-gebundenen Tasks leistungsfähiger als jThreads.[2] Dies betrifft z. B. Anwendungen, die mit Webservices kommunizieren. Jedes Senden einer Nachricht oder jeder Zugriff auf eine Seite stellt eine blockierende Anweisung dar. Wenn jeder Web-Request als ein Task in einem eigenen vThread ausgeführt wird, kann die Anwendung mehr Tasks gleichzeitig ausführen als mit jThreads. Anwendungen die jThreads verwenden, müssen asynchrone Programmierung verwenden, um eine ähnliche Anzahl von Tasks zu verarbeiten.

Das Starten von vThreads ist dem Starten von jThreads ähnlich:

```
1        Thread virtualThread = Thread.ofVirtual().start(() -> doWork());
2        virtualThread.join();
```

Listing 2.4: Starten eines vThread

In Zeile 1 von Listing 2.4 wird der vThread instanziiert. Statt über einen Konstruktor wie bei jThreads, werden vThreads über eine neue Builder-Funktion der Klasse `Thread` erzeugt. Mit `Thread.ofVirtual()` definiert der Entwickler, dass der Thread-Builder einen vThread erzeugen soll. Neben `ofVirtual()` wurde auch die Funktion `ofPlatform()` hinzugefügt, die es dem Builder erlaubt jThreads zu erzeugen. Der Builder gibt Objekte der Klasse `Thread` zurück. Da sowohl jThreads als auch vThreads über `Thread` implementiert werden, verwenden beide die gleichen Funktionen. In Zeile 2 von Listing 2.4 kann man dies direkt erkennen, da genau wie in Listing 2.1 über `join()` gewartet wird. Der vThread wird in diesem Codeausschnitt auch direkt, bei der Übergabe des Tasks, gestartet. Über `unstarted()` kann der Builder auch vThreads erzeugen, welche nicht automatisch starten.

Nicht gestartete vThreads werden wie jThreads gestartet:

```
1        Thread virtualThreadSimilar = Thread.ofVirtual().unstarted(() ->
           doWork());
2        virtualThread.start();
3        virtualThread.join();
```

Listing 2.5: Starten eines vThread mit manuellem Start

Für vThreads wurde die Klasse `Executors` um einen `ExecutorService` erweitert. Durch diesen `ExecutorService` wird der Code zum Starten und Verwenden von vThreads übersichtlicher:

```
1        executor = Executors.newVirtualThreadPerTaskExecutor();
2        executor.submit(() -> doWork());
```

Listing 2.6: Ausführen von Tasks über Executor

Wie der Name impliziert, wird mit `newVirtualThreadPerTaskExecutor()` ein `ExecutorService` erzeugt, der für jeden übergebenen Task, einen neuen vThread startet, der diesen ausführt.

Das Ausführen von Tasks mit jThreads über Work-Stealing oder mit vThreads über Thread-per-Task funktioniert analog. Der Unterschied liegt in den Vorteilen von IO-gebundenen Tasks und den Nachteilen von reinen CPU-gebundenen Tasks. Ein Entwickler, der sein Programm von Work-Stealing mit jThreads zu vThreads migrieren will, muss rein syntaktisch in den meisten Fällen lediglich den `ExecutorService` austauschen. Eine Anwendung, die für die asynchrone Programmierung mit jThreads geschrieben wurde, muss größtenteils neu geschrieben werden.

2.4 Project Loom in Java 19

In Java 19 werden vThreads als Preview Feature verfügbar sein. Die API ist in einem ausgereiften Zustand, aber es ist nicht auszuschließen, dass sie sich noch verändern wird. Außerdem wird noch an der Performance der vThreads gearbeitet und Funktionalität hinzugefügt.

Eine frühe Version von Structured Concurrency ist als Incubator Feature implementiert. Incubator Features sind instabiler als Preview Features und es wird angenommen, dass die API für Structured Concurrency überarbeitet wird.

3 Übersicht der Benchmarks

3.1 Simple Sleep Benchmark

Simple Sleep ist ein Benchmark, um den grundsätzlichen Vorteil von vThreads zu testen. Es werden n Tasks gestartet und jeder Task wartet 100 ms. Das Warten wird durch eine blockierende Anweisung ausgeführt. Es wird untersucht wie sich jThreads und vThreads bei wachsendem n verhalten.

Es wird erwartet, dass die Laufzeit von jThreads deutlich langsamer sein wird als die von vThreads. Da die Tasks IO-gebunden sind und nur aus Warten bestehen, sollte die Gesamtlaufzeit mit vThreads relativ konstant bei nicht viel mehr als 100 ms bleiben. Die Laufzeit von jThreads wird erwartungsgemäß schlechter sein, da Tasks warten werden müssen, bevor diese mit der Berechnung beginnen können, während bei den vThreads alle Tasks fast sofort mit der Berechnung beginnen können.

3.2 Simple Calculation Benchmark

Analog zu Simple Sleep, das rein IO-gebundene Tasks berechnet, sind die Tasks bei Simple Calculation rein CPU-gebunden. Es werden n Tasks gestartet und jeder Task führt für 500 ms eine einfache Berechnung durch. Jeder Task merkt sich zu dem Beginn seiner Berechnung seine Startzeit und berechnet kontinuierlich die Differenz zur Systemzeit, bis 500 ms vergangen sind. Ein Task muss nicht insgesamt 500 ms berechnen. Es soll ein Warten analog zu Simple Sleep stattfinden, nur dass die vThreads nicht austauschen können. Dieser Benchmark ist als Worst-Case-Benchmark für die vThreads gedacht.

Die Laufzeit der jThreads hängt stark vom Scheduling des Betriebssystems ab. Bei den vThreads müssen immer alle gestarteten Tasks beendet sein, bevor die nächsten beginnen können. Wenn die jThreads frühzeitig ausgewechselt werden, sodass die anderen Tasks beginnen können, bevor die vorherigen vollständig berechnet wurden, sind die jThreads schneller als die vThreads. Bei größeren Werten für n sind die vThreads gegenüber den jThreads deutlich im Nachteil.

3.3 nQueens Benchmark

Das nQueens-Problem fragt, wie n Damen auf einem $n \times n$ Schachbrett platziert werden können, sodass keine der Damen eine andere Dame schlagen kann.

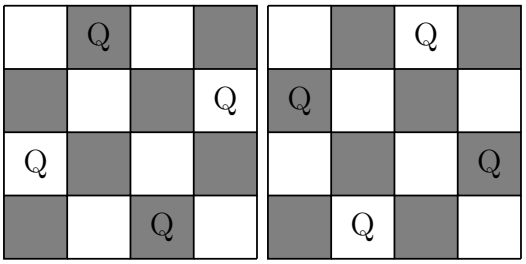


Abbildung 3.1: Die beiden Lösungen für $n = 4$

Der Benchmark testet, wie schnell für ein gegebenes n alle Lösungen für nQueens berechnet werden können. Die Anzahl der möglichen Damen-Belegungen nimmt mit steigendem n schnell zu und die Überprüfung erfordert einen moderaten Rechenaufwand. Zu Beginn sollten die vThreads etwas langsamer sein als die jThreads, da die Tasks durch den Rechenaufwand CPU-gebunden sind. Später werden die Tasks jedoch IO-gebundener, da mehr Subtasks gestartet werden müssen, auf die gewartet wird. Es ist jedoch möglich, dass sobald die Anzahl der gestarteten Tasks bei größeren n-Werten groß genug wird, die vThreads durch das Warten auf die Subtasks aufholen können.

3.4 Fibonacci Benchmark

Die Fibonacci Zahlen sind eine Zahlenfolge, die rekursiv wie folgt definiert werden kann:

$$f_0 = 1 \tag{3.1}$$

$$f_1 = 1 \tag{3.2}$$

$$f_n = f_{n-1} + f_{n-2} \tag{3.3}$$

Eine naive, rekursive Implementierung einer Fibonacci-Funktion wird als Benchmark verwendet. Die Ergebnisse werden nicht zwischengespeichert. Die rekursive Funktion berechnet $f_n = f_{n-1} + f_{n-2}$, aber für den Aufruf von f_{n-1} wird ein neuer Task gestartet.

```
1  fib(long n) {
2          if (n < 2) return 1;
3
4          a = submitTask( fib(n - 1) );
5          b = fib(n - 2);
6
7          waitFor a
8
9          return a + b;
10 }
```

Listing 3.1: Pseudo-Code Beispiel der Fibonacci-Funktion

Es wird erwartet, dass die vThreads erst bei größeren n einen Vorteil gegenüber den jThreads haben. Durch den Rechenaufwand sind die Tasks an die CPU gebunden und können bis zu einer bestimmten Anzahl von gestarteten Subtasks alle Ergebnisse ohne Preemption berechnen, während die vThreads eine kurze Zeit zum Wechseln benötigen. Somit ist bis zu einem bestimmten Wert für n die Berechnungsdauer kurz genug, sodass das Warten für den jThread nicht hinderlich für die Gesamtlaufzeit ist.

3.5 Chat Simulation Benchmark

Für diesen Benchmark wird ein künstlicher Chatserver erstellt. Der Benchmark wird aus der Sicht von n Clients ausgeführt, die sich mit dem Mockup-Server verbinden. Alle Clients und der Server befinden sich auf dem gleichen Computer, in der gleichen Anwendung. Es werden 50 zufällige Clients ausgewählt, die jeweils eine Chat-Nachricht an den Server senden. Das Senden der Nachrichten erfolgt parallel über Tasks. Wenn der Server eine Nachricht empfängt, analysiert er diese (wartet 50 ms) und sendet sie an alle Clients weiter. Der Server kann eine bestimmte Anzahl von Nachrichten parallel verarbeiten. Sobald jeder Client alle 50 Nachrichten erhalten hat, wird der Server beendet.

Beim Senden einer Chatnachricht wartet ein Client, bis der Server diese empfangen und analysiert hat. Wie genau der Server die Nachrichten analysiert ist für einen Client unwichtig. Wie in Abb. 3.2 dargestellt, hat jeder Client eine zufällige Distanz zum Server, die bestimmt, wie lange es dauert, die Nachricht zu senden. Das Senden und Verarbeiten der Nachrichten ist künstlich implementiert. Für das Senden der Nachricht wartet der Server entsprechend der Distanz zum Client, während für das Analysieren immer 50 ms verwendet werden.

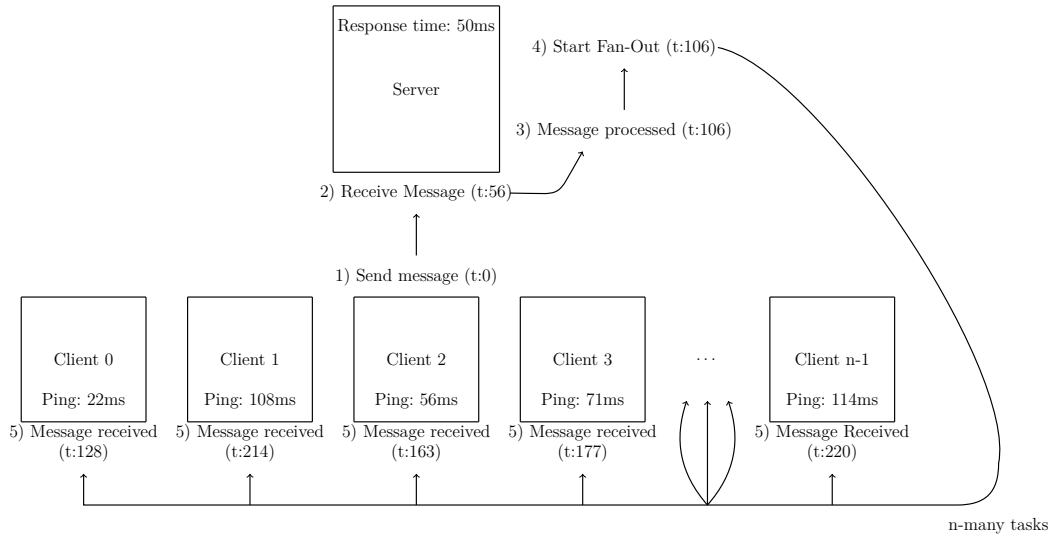


Abbildung 3.2: Näherungsweise Übersicht des Verlaufes von einer gesendeten Client-Nachricht

Das Senden, das Verarbeiten und das Verteilen einer Nachricht besteht aus blockierenden Anweisungen. Wie bei echten Serverinteraktionen ist dieses Warten ein Vorteil für vThreads. Wenn eine Nachricht an alle Clients weitergeleitet wird, können vThreads den Versand vieler Nachrichten starten. Die Gesamtlaufzeit mit vThreads sollte kürzer sein, als mit jThreads.

3.6 Textfile Server Benchmark

Ähnlich wie bei der Chat-Simulation, wird mit dem Textfile Server auch ein Mockup-Server erstellt. Der Server simuliert Datenbankabfragen. Der Server hat Zugriff auf Textdateien und n Clients fragen den Inhalt dieser Dateien ab. Jeder Client zählt dann die Anzahl der Wörter des erhaltenen Textes. Das Zählen der Wörter nimmt einige Zeit in Anspruch, da die Textdateien mehrere MB groß sind.

Der Server hat Threads, die ständig die eingehenden Anfragen auswerten und die Textdateien lesen, während die Clients den Inhalt der Dateien über Tasks

anfordern. Der Server und die Clients verwenden jeweils eigene Thread-Pools, da die Arbeit der Client Tasks nicht von den Server-Threads übernommen werden soll.

Mit diesem Benchmark soll untersucht werden, wie sich die vThreads bei größeren Tasks verhalten. Außerdem handelt es sich bei diesem Benchmark um ein Szenario, in dem der Zugriff auf den Server begrenzt ist und eine höhere Anzahl von vThreads nicht unbedingt mehr Dateien auf einmal lesen können. Das Zählen von Wörtern dauert lange genug, wodurch das Fehlen von vThread-Preemption einen spürbaren Einfluss auf die Gesamtlaufzeit haben kann. Das Lesen von Dateien und das Zählen von Wörtern stellen die IO- und CPU-gebundenen Aspekte der Tasks dar. Es wird erwartet, dass die vThreads hier langsamer sind als die jThreads. Das Lesen der Dateien ist jedoch aufgrund der Hardware ein inkonsistenter Prozess in Bezug auf die Laufzeit und führt zu schwankenden Gesamtlaufzeiten. Durch mehrmaliges Ausführen des Benchmarks sollte sich ein Mittelwert ergeben.

4 Implementierung der Benchmarks

Für die Ausführung der Benchmarks wurde eine eigene Ausführungsumgebung geschrieben. Über diese können alle Benchmarks einfach von einer Konsole aus gestartet werden. Die Ausführung verschiedener Benchmarks mit unterschiedlichen Parametern unterscheidet sich nur durch Kommandozeilenargumente.

Die Ausführung aller Benchmarks erfolgt durch die Klasse `BenchmarkRunner`. Diese Klasse nimmt verschiedene Kommandozeilenargumente entgegen und definiert über diese Argumente, welcher Benchmark mit welchen Parametern ausgeführt werden soll.

Alle Implementierungen der Benchmarks erben von der abstrakten Klasse `Benchmark`. Mit `Benchmark` wird das Verhalten vor und nach der Ausführung des Benchmarks standardisiert und festgelegt, welche Parameter ein Benchmark haben muss, damit neue Benchmarks einheitlich implementiert werden können.

Nach der Ausführung speichert jeder Benchmark alle gesammelten Daten in einem JSON-Objekt zur weiteren Verwendung. Diese JSON-Objekte können nach der Ausführung in einer Datei gespeichert werden. Aus einer Datei mit den JSON-Einträgen rendert die Klasse `GraphPlotter` Laufzeitgraphen. Mehrfach ausgeführte Benchmarks mit den gleichen Parametern werden beim Sammeln zu einem Mittelwert zusammengefasst. Die JSON-Einträge können als Graphen gerendert werden, die die Laufzeit und einen Parameter des Benchmarks in Beziehung setzen.

Für die Ausführung der Benchmarks über den `BenchmarkRunner` wurden Bash Skripte geschrieben, die der Benutzer zum Kompilieren, Ausführen und Plotten

verwenden kann. Diese Skripte ermöglichen die automatische Ausführung aller Benchmarks nacheinander von einer Linux-Konsole aus.

4.1 Die Benchmark Klasse

Alle Implementierungen der einzelnen Benchmarks erben von der abstrakten Klasse `Benchmark`. Die Klasse `Benchmark` enthält die zentrale Methode `runBenchmark()`. Jede Implementierung führt den Benchmark mit `runBenchmark()` aus, misst die Zeiten und protokolliert alle gemessenen Ergebnisse.

Jede Implementierung definiert die Bezeichner und Parameternamen des Benchmarks, indem sie abstrakte Methoden von `Benchmark` implementiert. Diese Bezeichner werden verwendet, um die Ergebnisse in den resultierenden JSON-Einträgen zu markieren und um die Graphen automatisch zu beschriften.

Jede Instanz von `Benchmark` hat einen „Concurrency-Type“. Der Concurrency Type nimmt einen von 3 Werten an. Abhängig vom Typ wird für den Benchmark festgelegt, ob dieser die Tasks sequentiell, über jThreads oder über vThreads ausführt. Die Benchmarks werden nur für jThreads und vThreads ausgeführt.

Für den Start der Benchmarks wird jeder `Benchmark` Instanz ein `ExecutorService` instanziiert und zugewiesen. Ob der `ExecutorService` über `newWorkStealingPool()` oder `newVirtualThreadPerTaskExecutor()` erzeugt wird (Kapitel 2), hängt vom Concurrency Type ab. Sequentielle Benchmarks benötigen keinen ExecutorService.

Die gemessene Laufzeit beinhaltet nicht die Instanziierung des ExecutorService oder die Verarbeitung der Ergebnisse.

4.2 Benchmarks

4.2.1 Simple Sleep Benchmark

Wie der Name des Benchmarks verrät, ist der Benchmark einfach aufgebaut. Es wird nur ein Wert als Parameter abgefragt: die Anzahl der zu startenden Threads.

```
1         try (ExecutorService executor = createExecutorService()) {
2             for (int i = 0; i < threadCount; ++i) {
3                 executor.submit(SimpleSleep::sleepThread);
4             }
5         }
```

Listing 4.1: Der gesamte Benchmark

Über `try-with-resource` in Zeile 1 von Listing 4.1 wird der Executor gestartet, der dem Concurrency Type entspricht. Dazu wird die Funktion `createExecutorService` aus der Superklasse `Benchmark` verwendet, die den entsprechenden `ExecutorService` instanziiert. Mit Java 19 werden `ExecutorServices` am Ende von `try-with-resource` automatisch geschlossen und es wird auf die Beendigung aller übergebenen Tasks gewartet.

Die Methode `sleepThread()` ist ein einfacher Wrapper für `Thread.sleep(100)`, der Exceptions abfängt.

4.2.2 Simple Calculation Benchmark

Dieser Benchmark ist ähnlich aufgebaut wie der SimpleSleep-Benchmark. Die grundsätzliche Ausführung entspricht Listing 4.1, jedoch wird statt SimpleSleep::sleepThread folgende Methode von den Tasks ausgeführt:

```
1     private static void calcThread() {
2         long startTime = System.currentTimeMillis();
3         long timePassed = 0L;
4
5         while (timePassed < calcDuration) {
6             timePassed = System.currentTimeMillis() - startTime;
7         }
8     }
```

Listing 4.2: Der CPU-gebundene Task

In der `while`-Schleife wird in jedem Schleifendurchlauf eine Subtraktion ausgeführt. Der gesamte Task hat keine blockierenden Anweisungen und ist CPU-gebunden.

4.2.3 nQueens Benchmark

Das nQueen Problem wurde mit Constraints gelöst. Die Damen werden nacheinander platziert. Nach jeder Dame werden alle Felder bestimmt, auf die die nächste Dame platziert werden kann, ohne von einer vorhergehenden Dame geschlagen zu werden. Da die Damen horizontal schlagen, befindet sich in jeder Reihe des Schachbretts nur eine Dame. Wenn es nur eine Dame pro Reihe gibt, kann das zweidimensionale Brett als eine eindimensionale Liste aufgefasst werden:

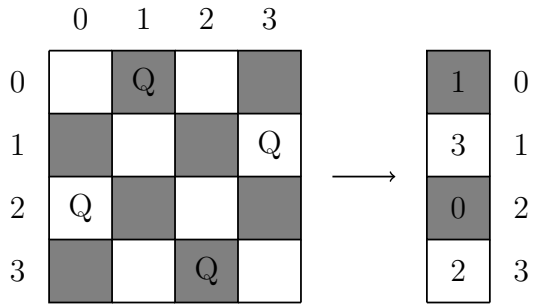


Abbildung 4.1: Übersetzung des Schachbretts zu einer Liste

Als „Konfiguration“ wird die Liste platzierter Damen und die Liste an möglichen Folge-Felder-Indexe verstanden. Wird z. B. auf einem 4×4 Brett eine Dame in das linke obere Feld platziert, so sind nur die Felder 2 und 3 der nächsten Reihe als Optionen offen. Aus diesen beiden Folge-Optionen lassen sich zwei Folge-Konfigurationen erschließen.

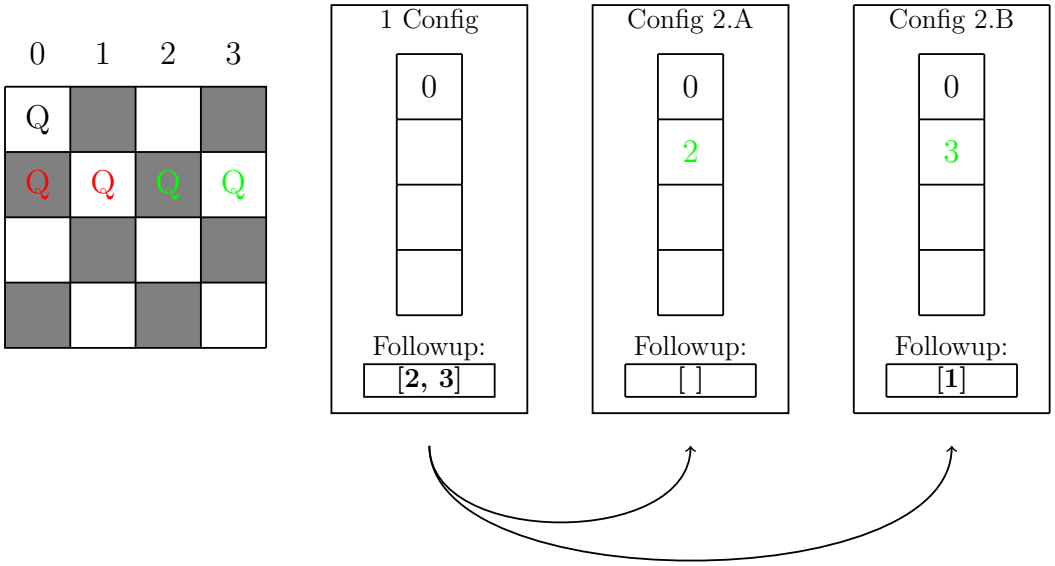


Abbildung 4.2: Eine Veranschaulichung der Folge-Konfigurationen

Eine Konfiguration wird als End-Konfiguration bezeichnet, wenn alle n Damen in der Liste platziert sind und gilt als eine Lösung für das nQueens-Problem. Wenn eine Konfiguration noch Damen zu platzieren hat, aber keine Folge-Optionen besitzt, wird die Konfiguration als invalide bezeichnet.

Eine Konfiguration wird durch die Klasse `Configuration` repräsentiert. Jede Instanz von `Configuration` besitzt Felder für die Liste der Damen und die Liste der Nachfolge-Optionen. Der einzige öffentliche Konstruktor erzeugt eine leere Konfiguration ohne platzierte Damen. Wenn die Funktion `placeQueen()` auf die `Configuration` C aufgerufen wird, wird eine neue `Configuration` instanziiert und zurückgegeben, die alle Werte von C besitzt. Erst auf diese neue

`Configuration` wird dann die Dame platziert. Die `Configuration` C wird dabei nicht verändert. Es ist somit möglich, `placeQueen()` mehrmals auf C aufzurufen um alle Nachfolge-Konfigurationen zu instanziieren.

Die Implementierung des Benchmarks ist im folgenden Pseudo-Code verdeutlicht:

```
1  List results = new List();
2
3  runBenchmark(args) {
4          Configuration startConfig = new Configuration();
5
6          for (i = 0; i < n; ++i) {
7              submitTask( buildConfig( startConfig.placeQueen(i) ) );
8          }
9
10         waitFor tasks;
11         return results.size();
12 }
13
14 buildConfig(configuration) {
15         for (int next : configuration.getNextOptions()) {
16             Configuration followUp = configuration.placeQueen(next);
17
18             if (followUp.isNQueensSolution()) {
19                 results.add(followUp);
20
21             } else if (followUp.canBePlacedOn()) {
22                 submitTask( buildConfig( followUp ) );
23             }
24         }
25
26         waitFor tasks
27     }
28 }
```

Listing 4.3: Pseudo-Code Beispiel der nQueen Implementierung

Ein Task prüft welche Nachfolger-Konfigurationen der übergebenen Konfigurationen eine valide Lösung darstellen. Jeder valide Nachfolger, der keine Lösung ist, wird dann von einem Subtask überprüft. Wenn eine Konfiguration

eine valide End-Konfiguration ist, wird diese an eine bestehende Ergebnis-Liste übermittelt. Am Ende des Benchmarks enthält die Ergebnis-Liste alle Lösungen des nQueens-Problems für das gegebene n.

Da für eine Lösung genau n Damen platziert werden, besteht ein Benchmark-Lauf aus maximal n „Task-Subtask-Generationen“. In jeder Generation wurde eine Dame platziert.

Es wurde ein „Sequentieller Cut-Off“ implementiert. Über den optionalen Cut-Off Parameter (CO) wird die Anzahl von Damen bestimmt, die parallel über Tasks platziert werden. Die Nachfolger-Konfigurationen für die restlichen Damen werden sequentiell berechnet. Die Anzahl Nachfolge-Optionen von Konfigurationen werden tendenziell mit jeder Generation kleiner. Ab einem bestimmten Punkt wird der Rechenaufwand für diese letzten Tasks so gering, dass die sequentielle Ausführung schneller ist. Durch den Cut-Off wird der Overhead für das Starten und Ausführen dieser Tasks vermieden, wodurch die Laufzeit verbessert wird.

4.2.4 Fibonacci Benchmark

Die Implementierung des Fibonacci Benchmarks ist analog zum Pseudo-Code in Listing 3.1. Zusätzlich wurde wie bei nQueens Benchmark ein sequentieller Cut-Off implementiert. Die Funktion `fib(n)` führt für ein n, das kleiner als der Cut-Off ist, die gesamte restliche Berechnung sequentiell durch.

4.2.5 Chat Simulation Benchmark

Wie in Abschnitt 3.5 beschrieben werden 50 Nachrichten von zufällig ausgewählten Clients an den Server gesendet.

Der Chat Simulation Benchmark verwendet 2 Parameter, für die Anzahl an Clients das n und den Parameter ThreadCount. Wenn ein Client eine Nachricht

an den Server sendet, wird die Nachricht an eine Queue nicht analysierter Nachrichten angehängt. Der Server startet ThreadCount viele Tasks, die in einer Endlosschleife jeweils eine Nachricht aus dieser Queue nehmen, diese analysiert und an alle Clients weiterleitet. Die Anzahl der Nachrichten, die der Server parallel verarbeiten kann, wird also durch ThreadCount definiert.

Um den Empfang eines HTTP-Antwortcodes zu simulieren, wartet ein Client, der eine Nachricht gesendet hat, bis der Server diese analysiert hat. Hierfür wurde eine eigene Implementierung des `Future` Interfaces geschrieben. Der Client erhält nach dem Senden eine solche `Future` Instanz zurück und der `get()` Aufruf wartet, bis der Server diese Nachricht verarbeitet hat.

4.2.6 Textfile Server Benchmark

Der Textfile Server Benchmark hat, ähnlich wie der Chat Simulation Benchmark, die Parameter n und ThreadCount. Zur Bearbeitung der Anfragen werden ThreadCount-viele Tasks gestartet. Ein solcher Task liest bei der Bearbeitung einer Anfrage den Inhalt einer Datei und gibt ihn zurück. Im Gegensatz zur Chat-Simulation wird nicht gewartet, sondern eine Datei auf dem System gelesen.

Clients senden über einen Task einen Dateinamen als Anfrage an den Server. Diese Anfrage wird wie bei der Chat Simulation an eine Queue angehängt. Der Client erhält eine Instanz von `Future` zurück, die über `get()` den Inhalt der Datei abfragen kann, sobald dieser verfügbar ist. Wird die Anfrage von einem Server-Task bearbeitet, wird die Anfrage von der Queue entnommen und der Inhalt der angeforderten Datei ausgelesen. Der Client wartet über `get()`, bis die Anfrage verarbeitet wurde und zählt anschließend die Anzahl der Wörter des Dateiinhalts.

5 Ergebnisse

Die Benchmarks werden mit dem Skript `batchRun.sh` ausgeführt. Dieses Skript wiederholt die Ausführung des Benchmarks, setzt aber iterativ neue Werte für die Parameter.

Mit folgender beispielhaften Skript-Aufruf wird der Simple Sleep Benchmark ausgeführt:

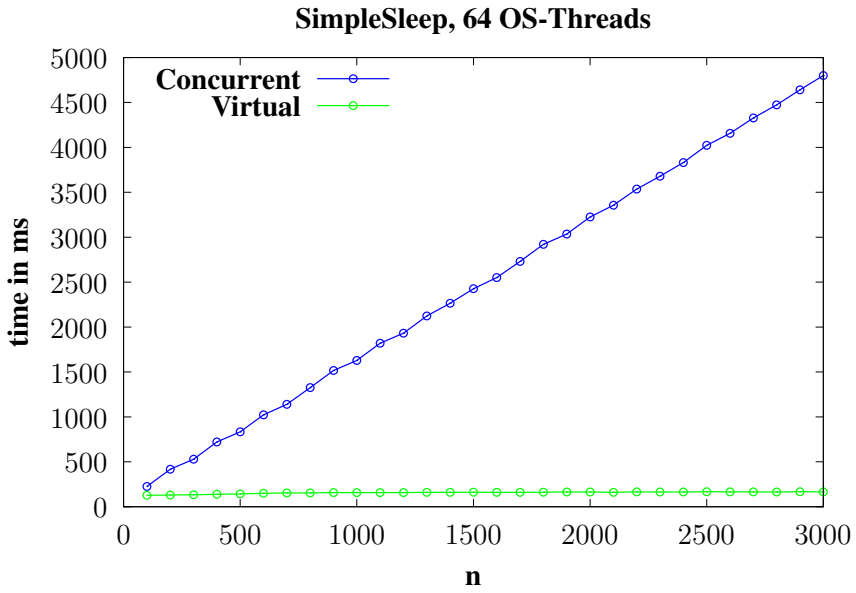
```
1    ./batchRun.sh -from 500 -to 3000 -steps 500 -repeat 20 -bm simpleSleep
```

Dabei werden für den Parameter n nacheinander die Werte 500, 1000, 1500 ... 3000 verwendet. Die Ausführung wird 20-mal wiederholt.

Für die Erstellung der Graphen wird der Mittelwert aller vermerkten Ausführungen der Benchmarks verwendet.

Die Graphen werden mit dem Namen des Benchmarks, der Anzahl der verfügbaren OS-Threads und, je nach Benchmark, dem Namen und Wert des zweiten Parameters, wie z. B. dem Cut-Off für nQueens, betitelt. Die Werte des Graphen sind mit „Concurrent“, für die Ausführung mit jThreads, und „Virtual“ für vThreads beschriftet. Der sequentielle Cut-Off (CO) wird als „cut-off“ und der ThreadCount (TC) als „accesslimit“ bezeichnet.

5.1 Simple Sleep Benchmark



Der Simple Sleep Benchmark ist speziell auf die Vorteile von vThreads ausgelegt. Abschnitt 5.1 zeigt, dass ein Vergleich mit jThreads ist nicht unbedingt sinnvoll ist, aber die enormen Laufzeitunterschiede unter optimalen Bedingungen sind gut zu erkennen.

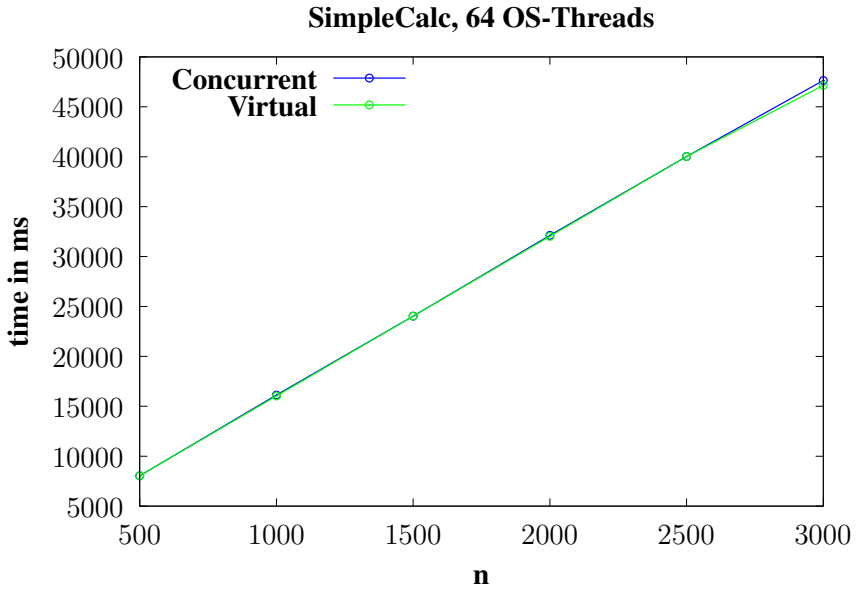
Die Tasks sind vollständig IO-gebunden. Für vThreads bedeutet dies, dass, wenn ein Task zu warten beginnt, dieser mit einem anderen vThread getauscht wird und der nächste mit dem Warten beginnen kann. Sehr schnell hat jeder Task mit der Berechnung und somit mit dem Warten begonnen. Nach 100 ms kann jeder wartende Task beenden. Die Gesamtlaufzeit bei den vThreads bleibt daher sehr konstant bei den 100 ms Wartezeit der Tasks und steigt mit steigender Anzahl der Tasks kaum an.

Bei den jThreads ist die Laufzeit deutlich langsamer. Die Tasks, die zu Beginn des Benchmarks nicht mit der Berechnung beginnen konnten, müssen warten, bis

die gesamte Berechnung der aktuell laufenden Tasks abgeschlossen ist, bevor sie die vollen 100 ms warten können.

Die Vermutung aus Abschnitt 4.2.1 scheint bestätigt worden zu sein und die Ausführung mit vThreads ist deutlich schneller als die mit jThreads.

5.2 Simple Calculation Benchmark



Die Ergebnisse der beiden Concurrency Types scheinen nahezu identisch zu sein. Da es sich um einen Worst-Case-Benchmark für vThreads handelt, wurde für diese eine schlechtere Laufzeit erwartet. Selbst bei mehrfacher Ausführung des Benchmarks, um die Varianz auszugleichen, kann höchstens ein Unterschied von wenigen Millisekunden vermutet werden. Scheinbar kam es für die jThreads nicht zur Preemption. Es kann vermutet werden, dass vThreads schneller werden könnten, wenn sie in Zukunft eine Preemption-Implementierung erhalten.

In der aktuellen Implementierung sind vThreads im Worst-Case gleich schnell wie jThreads.

5.3 nQueens Benchmark

Der nQueens Benchmark besitzt neben n einen zweiten Parameter: Cut-Off (CO). Für die Benchmarks, die einen zweiten Parameter besitzen, muss vor der Ausführung des Benchmarks ein Wert für diesen zweiten Parameter gewählt werden.

Zwei Ausführungen mit demselben konstanten CO, aber unterschiedlichen n führen zu einer unterschiedlichen Anzahl von Tasks die nicht gestartet werden. Damit ist für nQueens bei der Wahl von CO klar, dass der CO relativ zum gewählten n sein muss.

Um den CO für die Ausführung des Benchmarks zu testen, wurden COs im Bereich $n - 1$ bis $n - 10$ untersucht. Das heißt, es wurden die COs für eine parallel platzierte Dame bis 10 parallel platzierten Damen untersucht. Die Abb. 5.1 zeigt beispielhafte Laufzeiten für die Ausführungen.

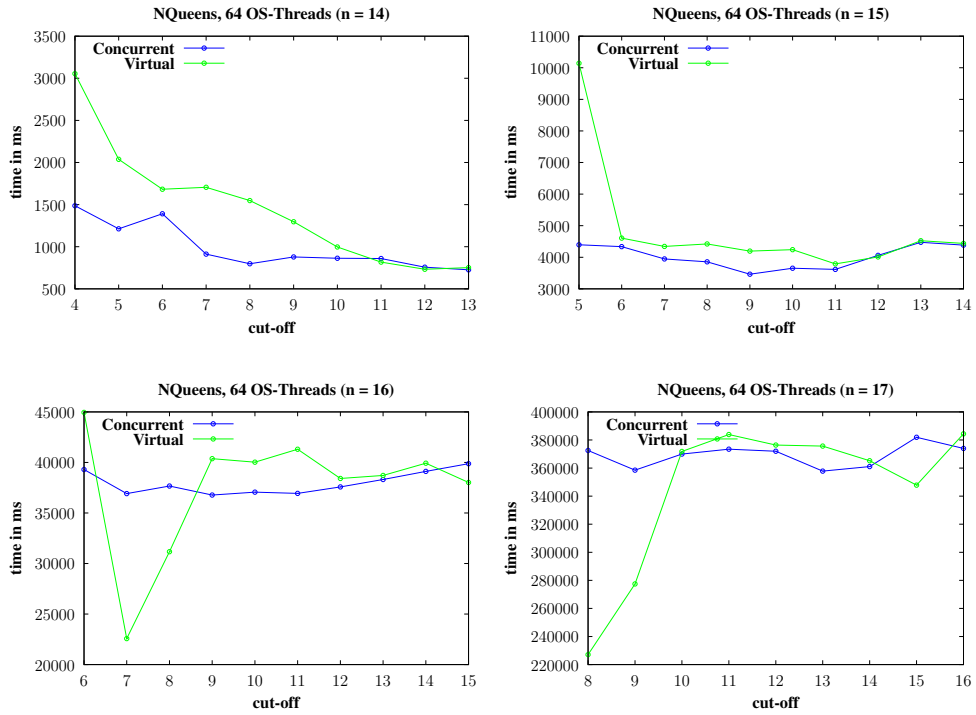


Abbildung 5.1: Untersuchung verschiedenen Cut-Off Werte für nQueens

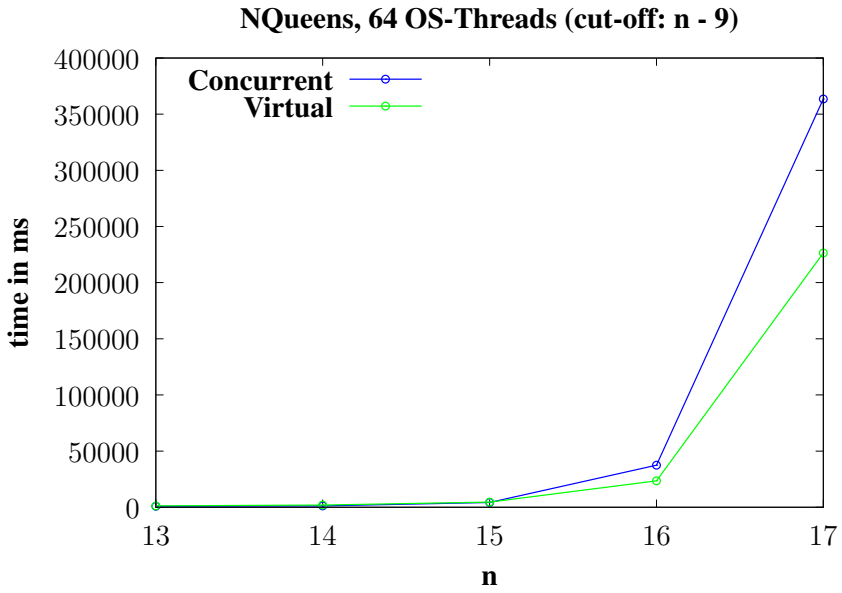
Abb. 5.1 zeigt ein relativ konsistentes Verhalten für jThreads. Abgesehen von einigen Schwankungen bleiben die jThread Werte horizontal relativ konstant.

Die Werte der vThreads zeigen dagegen ein unerwartetes Muster. Jeder Graph hat einen Cut-Off Wert, ab dem sich die vThread Werte den Werten der jThreads anzunähern beginnen. Für die verschiedenen Werte für n scheint dieser Punkt bei $CO_{near} = n - 4$ zu liegen. Ein höherer Cut-Off bedeutet, dass weniger Damen parallel platziert werden. Bei CO_{near} werden die letzten 4 Generationen sequentiell platziert.

Interessanter ist $CO_{div} = n - 9$. Vor CO_{div} ist die Berechnung mit vThreads deutlich langsamer und erst ab CO_{div} beginnen sich die Werte zu verbessern. Für Werte für $n > 15$ beginnt sich das Verhalten zu ändern. Cut-Offs kleiner als CO_{div} sind hier ebenfalls langsamer, aber ab $n > 15$ ist CO_{div} schneller als alle

anderen Werte. Die Laufzeit der vThreads ist mit CO_{div} etwa 1.6, $\times$ schneller als mit jThreads.

Die Benchmarks wurden mit CO_{div} ausgeführt. Dies ist ein interessanter Vergleich und führt zu einer Berechnung von nQueens mit einer höheren Anzahl parallel ausgeführter Tasks. Die Laufzeit unterscheidet sich für jThreads kaum.



Die Werte bis $n = 15$ liegen wie beschrieben sehr nahe beieinander. Die Tasks über vThreads sind bis zu diesem n-Wert langsamer als über jThreads. Ab $n \geq 16$ machen sich die oben beschriebenen Laufzeiten der vThreads bei CO_{div} bemerkbar. Bei der ersten Betrachtung scheinen diese Werte Anomalien zu sein. Es könnten Lösungen für das nQueens-Problem fehlen oder falsch berechnet worden sein, aber die Ergebnisse bleiben über mehrfache Durchläufe konsistent und die Lösungen für das nQueens Problem sind für diese n korrekt.

Durch die Suche nach Lösungen mittels einem Constraint-Schemas (Abschnitt 3.3) bleiben nach jeder platzierten Dame weniger Optionen für die nächste übrig. Daher werden bei dieser Implementierung von nQueens auch für jede folgende

Dame weniger Tasks gestartet. Dennoch ist die Anzahl der Tasks, auch bei CO_{div}, enorm. Allein für $n = 16$ gibt es mehr als 14 Millionen Lösungen.

Es ist nicht ganz klar, weshalb bei CO_{div} ein solcher Unterschied auftritt. Es scheint, dass bei CO_{div} die Balance zwischen der Wartezeit von Task zu Subtask und die CPU-Gebundenheit der letzten, parallelen Generation optimal ist.

Wenn mehr Generationen parallel platziert werden, so wartet ein Eltern-Task länger auf seine Subtasks und dieser Eltern-Task wird länger vom Scheduler der vThreads verwaltet. Werden mehr Generationen sequentiell platziert, so müssen die letzten vThreads längere, CPU-gebundene Tasks berechnen und beschäftigen ihre Carrier-Threads länger.

5.4 Fibonacci Benchmark

Wie bei nQueens verwendet der Fibonacci Benchmark den Cut-Off-Parameter (CO). Im Gegensatz zu nQueens ist der gesuchte CO ein konstanter Wert. Ausführungen mit verschiedenen n, aber gleichem CO führen zur gleichen Anzahl sequentiell berechneter Fibonacci-Zahlen für alle $n \geq CO$.

Es wurde ein Cut-Off mit Werten bei $n \geq 50$ gesucht, da sich in diesem Bereich Laufzeitunterschiede zwischen jThreads und vThreads bemerkbar machen. Aus den ersten groben Experimenten konnte der Bereich für CO auf $CO \in [20,\ 40]$ eingegrenzt werden.

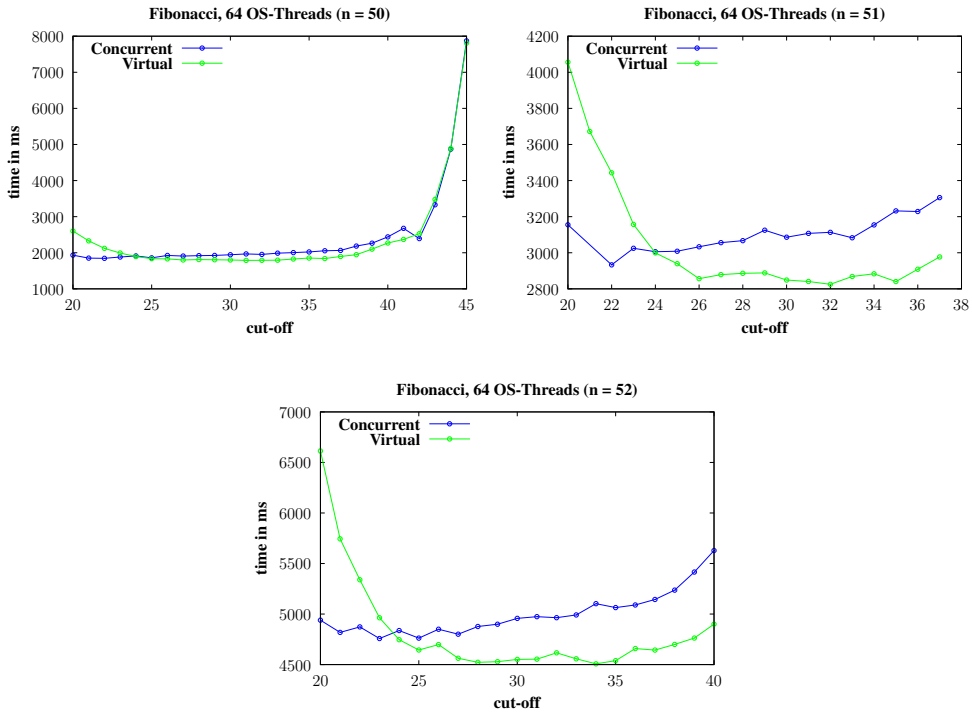
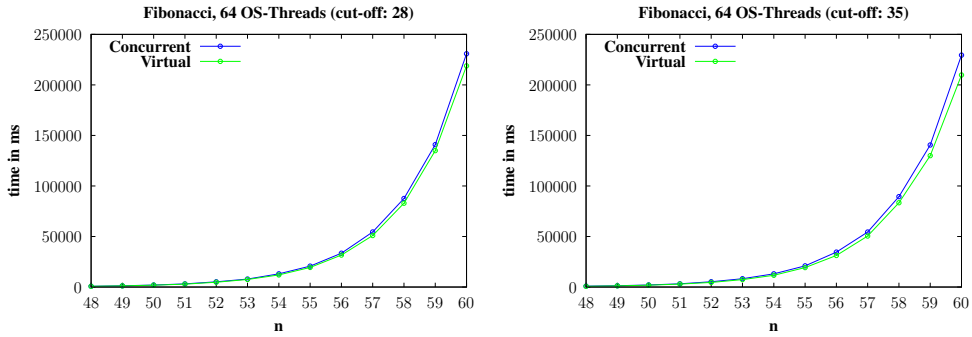


Abbildung 5.2: Beispielhafte Untersuchung verschiedener Cut-Off Werte für Fibonacci

Es wurde ein Cut-Off mit Werten bei $n \geq 50$ gesucht, da sich in diesem Bereich Laufzeitunterschiede zwischen jThreads und vThreads bemerkbar machen. Aus Abb. 5.2 ist kein optimaler Wert direkt ersichtlich. Bei $CO > 30$ scheint sich die Laufzeit für jThreads zu verschlechtern. Die vThreads hingegen haben bei $27 \leq CO \leq 37$ die schnellste Laufzeit. Die Laufzeit für vThreads scheint sich im getesteten Bereich parabolisch zu verhalten, mit einem Tiefpunkt um $CO = 35$.

Der Benchmark wird sowohl bei $CO = 28$ als auch bei $CO = 35$ ausgeführt. Bei $CO = 28$ handelt es sich um einen Mittelwert, zwischen den optimalen Werten für jThreads und vThreads. Steigende Cut-Off-Werte mit $CO \geq 28$ verschlechtern die Laufzeit für jThreads stärker, als das die vThreads schneller werden. Die vThreads haben jedoch die beste Laufzeit bei $CO = 35$.

Abb. 5.2 zeigt, dass für den praktischen Einsatz von vThreads in Anwendungen ein anderer Cut-Off verwendet werden muss als bisher bei jThreads. Durch den Vergleich der beiden Ergebnisse kann die Bedeutung des Cut-Offs untersucht werden.



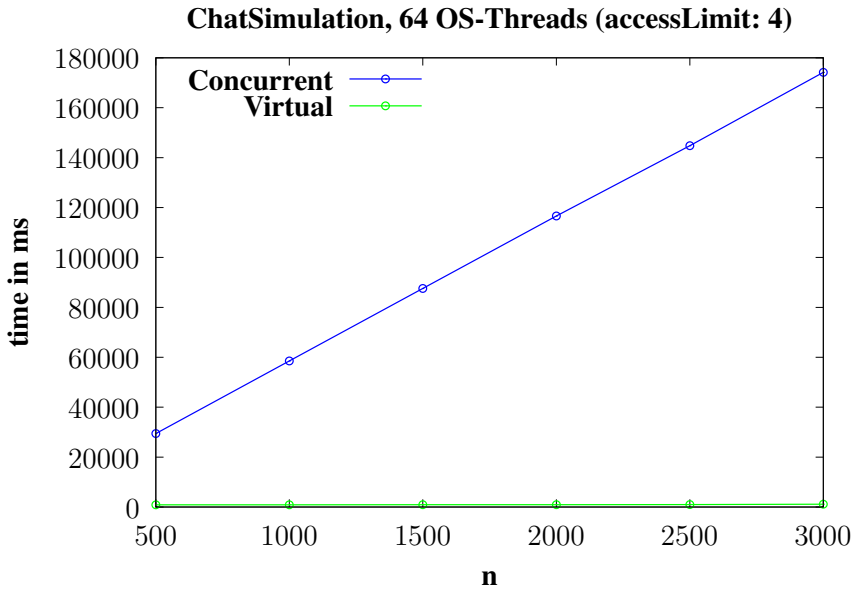
Die vThreads überholen die jThreads bei beiden Cut-Offs. Für $CO = 28$ scheint dies bei $n = 54$ der Fall zu sein und für $CO = 35$ bei $n = 51$. Die Laufzeit der jThreads verschlechtert sich bei $CO = 35$ nur minimal, aber die vThreads werden deutlich schneller. Bei $n = 60$ wurden die Ausführungen mit vThreads etwa 9.000 ms schneller und mit jThreads 1.300 ms langsamer.

Im Gegensatz zu nQueen werden deutlich weniger Tasks benötigt, aber die Tasks bei nQueen sind fertig berechnet, sobald diese ihre Konfigurationen überprüft haben. Die Berechnung eines Tasks bei Fibonacci kann erst beendet werden, wenn die gesamte Berechnung bis zum sequentiellen Cut-Off abgeschlossen wurde. Bei größeren Werten für n führt dies zu vielen Tasks, die auf Subtasks warten müssen. Die Berechnung eines rekursiven Schrittes für Fibonacci ist kurz, aber sobald die Anzahl der wartenden Tasks groß genug wird, haben vThreads den Vorteil.

5.5 Chat Simulation Benchmark

Wie in Abschnitt 4.2.5 beschrieben hat der Mock-Server eine Begrenzung, wie viele Chat-Nachrichten parallel verarbeitet werden können. Dieser Parallelitätsgrad wird durch den zweiten Parameter ThreadCount (TC) bestimmt.

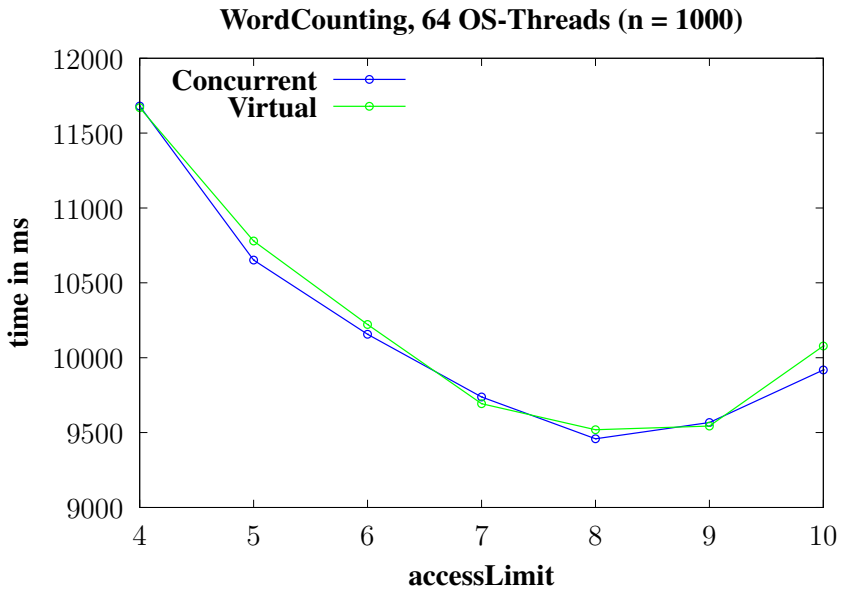
Im Gegensatz zu nQueens und Fibonacci hat die Wahl von TC nur einen geringen Einfluss auf die Laufzeiten. Bei $TC = 1$ findet keine parallele Verarbeitung der Nachrichten statt und die Laufzeit von jThreads und vThreads ist am langsamsten. Mit steigendem TC werden die Ergebnisse zunächst besser, stagnieren aber sehr schnell. Der gewählte TC für die Ausführung liegt bei $TC = 4$, wobei die Laufzeit für jThreads bevorzugt wird, die bei größeren TC Werten minimal ansteigt.



Die Ähnlichkeit mit dem Ergebnis des Simple Sleep Benchmark ist erkennbar. Wie bei echten Chat-Servern besteht ein großer Teil der Kommunikation aus dem Warten auf Antworten. Der eigentliche CPU-gebundene Teil des

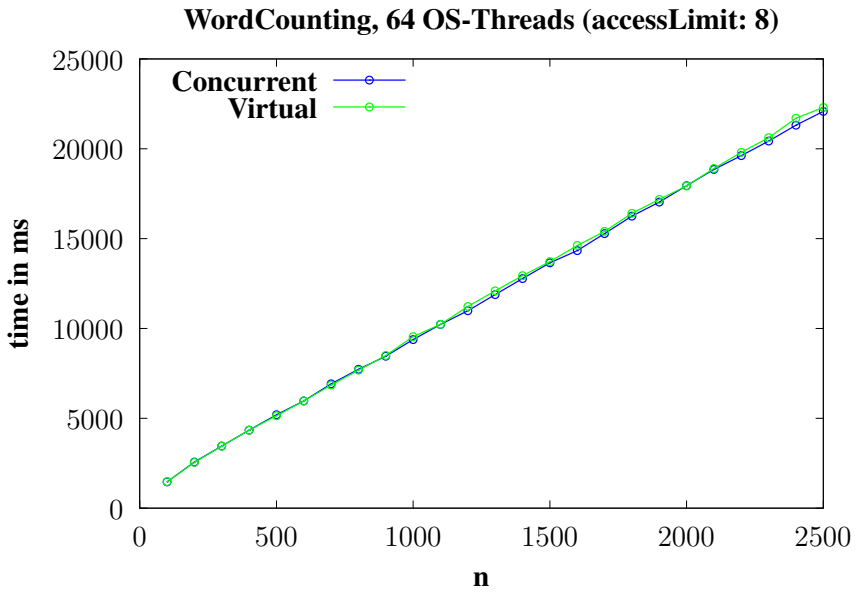
Kommunikationsvorgangs besteht aus dem Server, seiner Verarbeitung der Nachrichten und der Erstellung der Antwortnachricht. Für den Client ist dieser Vorgang allerdings irrelevant und er wartet nur. Aus diesen Gründen verfügt der Chat Server über genügend blockierende Anweisungen, sodass die vThreads einen klaren Vorteil gegenüber den jThreads haben.

5.6 Textfile Server Benchmark



Die Schwierigkeit beim Textfile Server Benchmark ist die Abhängigkeit von der Hardware. Das Lesen von Dateien hat keine konstante Geschwindigkeit und schwankt. Erst bei großen Werten für n und mehrfacher Wiederholung des Benchmarks lässt sich ein Mittelwert erkennen.

Die Laufzeit für jThreads und vThreads bleibt für alle gewählten TCs sehr ähnlich. Der beste Wert für beide Concurrency Types ist bei $TC = 8$ zu erkennen.



Ähnlich wie bei demm Ergebnis des SimpleCalc Benchmarks liegen die Laufzeiten dicht beieinander. Im Gegensatz zu SimpleCalc sind die vThreads konstant etwas langsamer als die jThreads. In diesem Benchmark können die vThreads ihren großen Vorteil, das Warten, nicht ausnutzen. Der Benchmark wurde mit $TC = 8$ durchgeführt. Der Textfile-Server konnte also nur 8 Dateien parallel lesen. Wenn jeweils 8 vThreads Dateiinhalte anfordern, wird auf die Antwort des Servers gewartet. Während die ersten 8 warten, können andere vThreads ihre Berechnungen nicht durchführen, da der Server voll ausgelastet ist.

Die vThreads versuchen bei jedem Warten auszuwechseln, aber können nur zu vThreads wechseln, die nicht berechnen können. Die Zeit für das Auswechseln von vThreads ist gering, führt aber bei der hohen Anzahl von Tasks zu einer schlechteren Laufzeit im Vergleich zu den jThreads.

5.7 Zusammenfassung

Die vThreads in ihrer jetzigen Form sind ein Preview Feature in Java 19. In zukünftigen Java-Versionen kann sich die Performance und Funktionalität noch ändern, aber die Ergebnisse aus Kapitel 5 haben gezeigt, dass die vThreads bereits Stärken gegenüber den bisherigen jThreads aufweisen.

Entwickelt werden vThreads von Project Loom, mit dem Ziel eine jThread-Alternative zu schaffen, die den OS-Thread nur während der aktiven Berechnung belegt. Die Benchmarks mit besonders IO-gebundenen Tasks, Simple Sleep und Chat Simulation, zeigen die Stärken von vThreads am besten. Anwendungen aus Bereichen wie z. B. Backend Server, die auf eine große Anzahl IO-gebundener Tasks angewiesen sind, können mit den vThreads Serveranfragen beantworten, ohne auf Paradigmen wie die asynchrone Programmierung zurückgreifen zu müssen.

Bei den Benchmarks nQueen und Fibonacci können mit der richtigen Wahl des Cut-Off beeindruckende Laufzeiten erreicht werden. Benchmarks wie SimpleCalc zeigen, dass die vThreads auch bei rein CPU-gebundenen Tasks mit den jThreads vergleichbar sein können.

Die von Project Loom angestrebte nahtlose Portierung von jThreads auf vThreads ist jedoch noch nicht möglich. Wie die Cut-Off Untersuchungen für den nQueens Benchmark zeigen, kann die Laufzeit von vThreads in bestimmten Anwendungen stark von der Wahl anderer Parameter abhängen. Setzt man den Cut-Off bei nQueens zu klein, so wird die Laufzeit deutlich schlechter. Ein kleiner Unterschied in den Parametern führt zu einer stark schwankenden Performance der vThreads. Wir wissen nicht, warum es zu solchen Schwankungen kommt, aber es ist möglich, dass die aktuelle Implementierung von vThreads nicht optimal ausgereift ist. Dagegen scheint die Laufzeit bei jThreads konsistenter zu sein. Für den

Übergang von jThreads zu vThreads müssen Anwendungen zunächst auf solche entscheidenden Parameter untersucht werden.

Wie man an dem parabolischen Verlauf der Graphen bei den Cut-Off Untersuchungen bei Fibonacci erkennen kann, weisen die vThreads bei steigender Anzahl Tasks eine deutlich langsamere Performance als jThreads auf. Es gilt also nicht generell, dass nahezu beliebig viele vThreads gestartet werden können. Dieses Ziel von Project Loom scheint nur mit besonders IO-gebundenen Tasks erfüllbar zu sein.

Anwendungen, bei denen Tasks warten, aber vThreads aus externen Gründen nicht berechnet werden können, wie z. B. beim Textfile-Server Benchmark, zeigen, dass das Scheduling der vThreads in der aktuellen Version zu Laufzeitnachteilen führen kann. Obwohl die Implementierung der vThreads vollständig innerhalb der JVM liegt, fehlen dem Entwickler die Kontrolle über deren Verhalten. Solche Anwendungen zeigen auch, dass ein hybrider Einsatz von jThread und vThread sinnvoll sein kann.

6 Schlussbemerkungen

Da die aktuelle Implementierung der vThreads noch recht neu ist, konnten zum Zeitpunkt der Erstellung dieser Arbeit keine verwandten Arbeiten oder Papers gefunden werden, bei denen jThreads und vThreads verglichen wurden.

6.1 Fazit

Das Schreiben von Anwendungen mit vielen IO-gebundenen Tasks wird durch Project Loom mit den vThreads erleichtert. Ohne asynchrone Programmierung, mit der bisher blockierende Anweisungen umgangen wurden, kann der Programmfluss fast so einfach gehalten werden wie bei sequentieller Programmierung. Entwickler vermeiden umständliches Debugging, schwer verständliche Exceptions und den Zwang zu externen Libraries. Allerdings ist diese Version von vThreads in Java 19 noch eine Vorschau und nicht final. Anwendungen müssen für vThreads angepasst werden und es kann sein, dass dies mehrere Experimente erfordert.

Die Forschung und Entwicklung von Project Loom hat gezeigt, dass in zukünftigen Versionen von Java eine starke Alternative zu jThreads verfügbar sein wird, die es Entwicklern erleichtern wird, nebenläufige Software zu schreiben.

6.2 Ausblick

Da es sich bei vThreads um ein Preview Feature handelt, werden an viele Aspekte wie die Laufzeit und das Scheduling noch gearbeitet. Darüber hinaus sind weitere Funktionalitäten für vThreads geplant.

In Project Loom wird eine Implementierung für „forced Preemption“ für die vThreads untersucht. Lange laufende vThreads können durch forced Preemption unterbrochen und durch andere vThreads ersetzt werden. Die aktuelle Performance von CPU-gebundenen Tasks bietet ohne Preemption keinen Vorteil gegenüber jThreads. Forced Preemption wäre unabhängig vom Scheduling des Betriebssystems. Z. B. könnten im SimpleCalc Benchmark, wo es nicht zu Preemption bei den jThreads kam, die vThreads ausgewechselt werden und eine bessere Laufzeit erzielen.

In Zukunft soll es Entwicklern möglich sein, vThreads ein eigenes Scheduling zuzuweisen. Bisher ist nur das Scheduling über `ForkJoinPool` möglich. Entwickler können versuchen, die Laufzeitnachteile durch eigenes Scheduling in Anwendungen wie dem Textfile-Server Benchmark zu umgehen.

Literatur

[1] R. Pressler. *Project Loom: Fibers and Continuations for the Java Virtual Machine*. 19. Feb. 2023. URL: https://cr.openjdk.org/~rpressler/loom/Loom-Proposal.html.

[2] R. Pressler und A. Bateman. *JEP 425: Virtual Threads (Preview)*. 19. Feb. 2023. URL: https://openjdk.org/jeps/425#Using-virtual-threads-vs--platform-threads.