Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

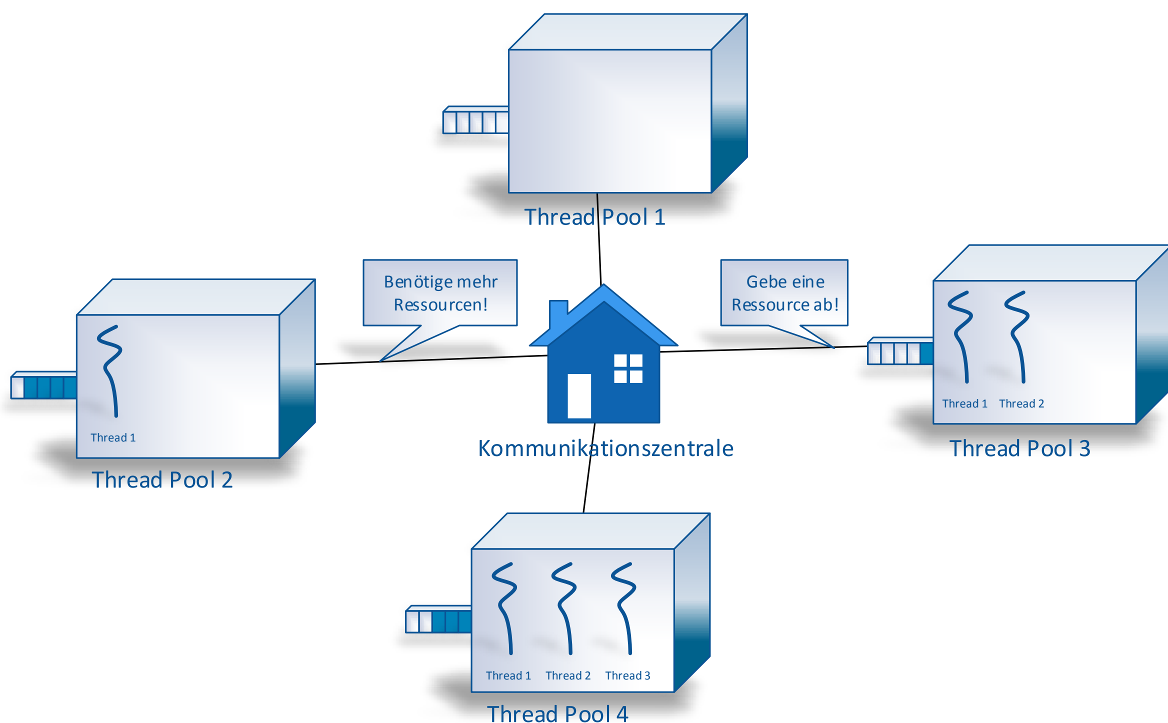
Lehrstuhl Prof. Dr.-Ing. Snelting

Kommunizierende Thread Pools

Bachelorarbeit von

Tobias Weiberg

an der Fakultät für Informatik



Gutachter: Prof. Dr.-Ing. Gregor Snelting

Betreuender Mitarbeiter: Dipl.-Inform. Andreas Zwinkau

Bearbeitungszeit: 11. Dezember 2013 – 31. März 2014

Zusammenfassung

Thread Pools sind ein verbreitetes Werkzeug zur parallelen Ausführung vieler kurzer und unabhängiger Tasks. In dieser Arbeit wurde untersucht, ob sich beim Einsatz mehrerer Thread-Pool-Instanzen eine Verbesserung der Laufzeit erreichen lässt, wenn diese miteinander kommunizieren können.
Dazu wurde eine Implementierung kommunizierender Thread Pools in der Programmiersprache Java entwickelt und im Vergleich mit einer bereits vorhandenen Thread-Pool-Implementierung evaluiert. Die kommunizierenden Thread Pools waren in der Lage, zur Laufzeit miteinander abzustimmen, wie viele Systemressourcen einer Instanz zu einem Zeitpunkt zur Verfügung stehen. Die Zuweisung einer Ressource berechtigte dabei eine Instanz zur Ausführung eines Threads.
Die Ergebnisse der Evaluation zeigen, dass durch diesen Ansatz deutliche Laufzeitverbesserungen erreicht werden können. Die Höhe der Verbesserungen hängt jedoch stark von dem Anwendungsfall und von den Eigenschaften des eingesetzten Systems ab.

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **3**
2.1 Threads . . . 3
2.1.1 Kontextwechsel . . . 3
2.1.2 Cache-Trashing . . . 3
2.2 Thread Pools . . . 4
2.2.1 Einsatzgebiete . . . 4
2.3 Thread Pools in Java . . . 4
2.3.1 Die Klasse ThreadPoolExecutor . . . 5
2.4 Verwandte Arbeiten . . . 5

3 Entwurf und Implementierung **9**
3.1 Erste Überlegungen . . . 10
3.1.1 Beziehung zu ThreadPoolExecutor . . . 10
3.1.2 Scheduling . . . 10
3.1.3 Implementierung der Kommunikation . . . 10
3.2 Erster Lösungsansatz . . . 11
3.2.1 Diskussion der Vor- und Nachteile . . . 12
3.3 Zweiter Lösungsansatz . . . 12
3.4 Die Klasse CommunicativeThreadPoolExecutor . . . 13
3.4.1 Task- und Thread-Verwaltung . . . 13
3.4.1.1 Implementierung der Warteschlange . . . 13
3.4.1.2 Repräsentation der Threads . . . 14
3.4.1.3 Berechtigungen und blockierte Threads . . . 14
3.4.1.4 Beenden von Threads . . . 14
3.4.2 Implementierung der wichtigsten Methoden . . . 15
3.4.2.1 Vererbung von AbstractExecutorService . . . 15
3.4.2.2 Die *execute*-Methode . . . 16
3.4.2.3 Die Methoden *shutdown* und *shutdownNow* . . . 16
3.4.2.4 Warten auf das Ende - die *awaitTermination*-Methode 17
3.4.2.5 Die Methoden *isShutdown* und *isTerminated* . . . 17
3.4.3 Zustände . . . 17
3.4.4 Die innere Klasse SidelinesSemaphore . . . 18
3.4.5 Ressourcenverwaltung . . . 19
3.4.6 Abweisen von Tasks . . . 19
3.5 Die Klasse ThreadPoolCommunicationCenter . . . 20
3.5.1 Verwaltung der Thread-Pool-Instanzen . . . 20
3.5.2 Methoden zur Kommunikation . . . 20

3.5.3 Ressourcenverteilung 21
3.6 Die Utility-Klasse CommunicativeExecutors 22
3.7 Synchronisation 22

4 Evaluierung 25
4.1 Aufbau der Testfälle 25
4.2 Evaluierungsmethode 26
4.3 Testumgebungen 27
4.4 Ergebnisse 27
4.4.1 Testfall 1 27
4.4.2 Testfall 2 28
4.4.2.1 Bemerkungen 29
4.4.3 Testfall 3 30
4.4.3.1 Bemerkungen 32
4.4.4 Testfall 4 33
4.5 Abschließende Bemerkungen 34

5 Zusammenfassung und Ausblick 37

Literaturverzeichnis 39

A Anhang 41
A.1 Messergebnisse - Testfall 1 und 2 41
A.2 Messergebnisse - Testfall 3 43
A.3 Messergebnisse - Testfall 4 44

1. Einleitung

Webanwendungen erfreuen sich in den letzten Jahren einer wachsenden Beliebtheit. Immer mehr Applikationen werden in das Internet ausgelagert. Die Zahl der Anfragen, denen ein Webserver standhalten muss, steigt unaufhörlich. Um diesem Trend gerecht zu werden, werden immer schnellere Prozessoren verbaut. Da es seit einiger Zeit nicht mehr effizient ist, die Taktraten von Prozessoren zu steigern, ist Parallelität die einzige Möglichkeit die steigenden Anforderungen zu erfüllen. Heutzutage werden Mehrkernprozessoren mit einer stetig steigenden Anzahl an Kernen eingesetzt. Um das mit dieser Entwicklung verbundene Potential ausschöpfen zu können, werden jedoch geeignete softwaretechnische Bausteine benötigt.

Ein Thread Pool besteht aus einer in der Regel begrenzten Menge an Threads und einer Warteschlange. Die Threads sind dabei zur Ausführung von Tasks vorgesehen und die Warteschlange zur Zwischenspeicherung dieser Tasks, wenn zum Zeitpunkt ihrer Ankunft kein Thread zur Ausführung bereit ist. Thread Pools sind ein geeignetes Werkzeug, wenn viele homogene und voneinander unabhängige Tasks ausgeführt werden sollen. Aus diesem Grund werden sie bevorzugt bei der Entwicklung von netzwerkbasierten Server-Applikationen eingesetzt. Entsprechend optimiert bieten sie eine effiziente und ressourcenschonende Möglichkeit, auch unter Last eine große Anzahl an Anfragen abzuarbeiten. Daher ist die Optimierung der Parameter von Thread Pools bereits Thema einiger Forschungsarbeiten.

In einer Anwendung können verschiedene Thread-Pool-Instanzen an unterschiedlichen Stellen zum Einsatz kommen. Dabei werden die Instanzen typischerweise unabhängig voneinander für ihren Aufgabenbereich optimiert. In dieser Arbeit soll untersucht werden, ob es die Leistung einer Anwendung positiv beeinflussen kann, wenn die darin verwendeten verschiedenen Thread-Pool-Instanzen miteinander kommunizieren können. Die Idee hierzu stammt aus [**?**]. Dort konnten unter Last Leistungsvorteile bei der Ausführung zweier Prozesse gemessen werden, wenn diese miteinander bzgl. ihrer Prozessornutzung abgestimmt waren.

Zur Untersuchung der oben genannten Frage soll zunächst eine Implementierung kommunizierender Thread Pools in einer beliebigen Programmiersprache erstellt und diese dann im Vergleich mit einer vorhandenen Thread-Pool-Implementierung evaluiert werden. Die Wahl der Programmiersprache fiel dabei auf Java, da sie in

Form der Klasse ThreadPoolExecutor des Executor-Frameworks über eine geeignete Vergleichsimplementierung verfügt.

Im folgenden Kapitel werden zunächst die für das Verständnis dieser Arbeit erforderlichen Grundlagen erläutert. Kapitel 3 beinhaltet eine ausführliche Beschreibung des verfolgten Lösungsansatzes. Außerdem werden dort Entwurfsentscheidungen und Implementierungsdetails erläutert. Danach werden in Kapitel 4 die Ergebnisse der Evaluierung präsentiert. Abschließend beinhaltet Kapitel 5 eine Zusammenfassung über die in dieser Arbeit gewonnenen Erkenntnisse.

2. Grundlagen

2.1 Threads

Der Einsatz von Mehrkernprozessoren erfordert ein Programmiermodell, mit dem die parallele Ausführung von Anwendungen möglich ist. Heutige Prozessoren unterstützen die Multithreading-Technologie. Der Code einer Anwendung kann dann in mehrere Einheiten unterteilt werden, die mit Hilfe dieser Technologie gleichzeitig auf verschiedenen Prozessorkernen ausgeführt werden können. Eine solche Einheit wird dabei als Thread bezeichnet. Um sicherzustellen, dass die Anwendung weiterhin korrekt arbeitet, ist gegebenenfalls die Synchronisation der Threads notwendig. Jeder Thread arbeitet in seinem eigenen Kontext. Darunter sind alle Informationen zu verstehen, die für die Ausführung dieses Threads benötigt werden.

2.1.1 Kontextwechsel

Das Betriebssystem entscheidet, wann ein Thread einem bestimmten Prozessorkern zugewiesen und ausgeführt wird (Scheduling). Je nach Methode wird ein Thread mehrmals während seiner Ausführung unterbrochen und ein anderer erhält Rechenzeit auf dem entsprechenden Kern. Dabei muss der Kontext des alten Threads gesichert und der des neuen wiederhergestellt werden. Dieser Vorgang wird Kontextwechsel genannt. Konkurriert eine hohe Anzahl an Threads um Rechenzeit auf vergleichsweise wenigen Prozessorkernen, leistet die Zeit, die für Kontextwechsel benötigt wird, einen relevanten Beitrag zur Gesamtlaufzeit des Programms. Aus diesem Grund kann die Verwendung einer zu hohen Anzahl an Threads die Laufzeit eines Programms verlängern.

2.1.2 Cache-Trashing

Arbeiten Threads auf verschiedenen Datensätzen, werden diese zur Zugriffsbeschleunigung in den Cache-Speicher des jeweiligen Prozessorkerns geladen. Ist die Anzahl an Threads groß und ebenso die Zahl an Kontextwechseln, steigt die Zahl der Cache-Fehlzugriffe stark an, da jeder Thread dort die Daten seines Vorgängers vorfindet. Der Fehlzugriff bewirkt, dass vorhandene Daten verdrängt werden, um Speicherplatz für die Daten des neuen Threads zu schaffen. Das hat zur Folge, dass der

ursprüngliche Thread ebenfalls nicht vom Cache-Speicher profitieren kann, sobald er fortgesetzt wird. Das häufige Laden derselben Daten in den Cache-Speicher und das Verdrängen dieser Daten, bevor sie erneut verwendet werden können, wird dabei als Cache-Trashing bezeichnet. Dadurch kann die Ausführungszeit des Programms ebenso wie durch Kontextwechsel negativ beeinträchtigt werden.

2.2 Thread Pools

Ein Thread Pool besteht aus einer Menge von Threads und einer Warteschlange. Die Threads sind zur Ausführung von Tasks vorgesehen. In die Warteschlange werden eingehende Tasks eingereiht, die zum Zeitpunkt ihrer Ankunft nicht von einem dieser Threads bearbeitet werden können. Die Tasks verweilen in der Warteschlange bis einer der Threads seine aktuelle Aufgabe beendet hat und sich ihrer annimmt. Die Wiederverwendung eines Threads soll dabei den Mehraufwand reduzieren, der bei der Erzeugung eines neuen Threads für jeden eingehenden Task entstehen würde.

Die Anzahl der Threads ist im Regelfall durch einen festen Wert, der von verschiedenen Faktoren abhängt, begrenzt. Ein wichtiger Faktor ist der Grad an Parallelität, den die Prozessorarchitektur zulässt. Beispielsweise kann ein Vierkernprozessor, der kein Hyperthreading unterstützt, maximal vier Threads gleichzeitig ausführen. Die Hyperthreading-Technologie ermöglicht es Prozessoren, mehr als einen Thread nebenläufig auf einem physikalischen Prozessorkern auszuführen. Ebenfalls relevant sind die Eigenschaften der Tasks, die dem Thread Pool zur Bearbeitung zugewiesen werden. Handelt es sich dabei ausschließlich um CPU-intensive Berechnungen, kann keine Leistungssteigerung erreicht werden, wenn eine größere Anzahl an Threads verwendet wird, als gleichzeitig ausgeführt werden kann. Muss ein Thread jedoch während seiner Ausführung auf spezielle Ereignisse warten, kann in der Zwischenzeit die Berechnung eines anderen Threads fortgesetzt werden. In diesem Fall ist auch eine Leistungssteigerung zu erwarten, wenn die Anzahl der Threads weiter erhöht wird.

Die Anzahl an Threads hat direkten Einfluss auf die Leistung, die erreicht werden kann. Ausgehend vom optimalen Wert steigt mit zunehmender Anzahl der Speicherverbrauch sowie die Anzahl an Kontextwechseln. Mit abnehmender Anzahl werden die zur Verfügung stehenden Ressourcen nicht optimal eingesetzt.

2.2.1 Einsatzgebiete

Thread Pools sind ein effizientes Werkzeug, wenn viele kurze und unabhängige Tasks parallel ausgeführt werden sollen. Daher sind sie besonders geeignet für Serveranwendungen, da in diesem Umfeld eine große Anzahl an kurzen Anfragen in möglichst kurzer Zeit verarbeitet werden muss. Ebenso können sie sinnvoll in nicht serverbasierten Anwendungen eingesetzt werden, sofern dort eine große Anzahl kurzer rechenintensiver Tasks ausgeführt werden soll.

2.3 Thread Pools in Java

Seit Java 5 ist das Executor-Framework fester Bestandteil der Java API ([**?**]) und wurde seitdem weiterentwickelt und erweitert. Die aktuelle Version Java 7 stellt drei Realisierungen von Thread Pools im Paket java.util.concurrent zur Verfügung:

1. ThreadPoolExecutor
2. ScheduledThreadPoolExecutor
3. ForkJoinPool

Dabei ist die ThreadPoolExecutor-Klasse eine vielseitig einsetzbare Implementierung mit vielen Parametern, um sie individuell an die gewünschten Anforderungen anpassen zu können. Da sie eine typische Repräsentation eines Thread Pools darstellt, wird sie in dieser Arbeit als Vergleichsimplementierung für die Evaluierung verwendet. Der folgende Abschnitt enthält deshalb eine genauere Beschreibung der Klasse ThreadPoolExecutor. Die beiden weiteren Klassen bieten eine für bestimmte Spezialfälle optimierte Implementierung der Thread-Pool-Funktionalität. Die Klasse ScheduledThreadPoolExecutor ist für die Ausführung von zeitabhängigen Tasks vorgesehen und die Klasse ForkJoinPool bietet eine für Teile-und-Herrsche-Probleme optimierte Implementierung.

2.3.1 Die Klasse ThreadPoolExecutor

Die ThreadPoolExecutor-Klasse weist ein flexibles Grundgerüst auf, das für die meisten Anwendungsfälle ausreichend geeignet ist. Mit den verschiedenen Parametern lässt sich der Thread Pool an die individuellen Bedürfnisse anpassen. Eine ThreadPoolExecutor-Instanz unterscheidet bei den ihr zur Verfügung stehenden Threads zwischen sogenannten Core-Threads und Nicht-Core-Threads. Im Unterschied zu Core-Threads, werden Nicht-Core-Threads beendet, wenn sie in einer durch einen Parameter festgelegten Zeitspanne keinen weiteren Task zur Ausführung erhalten. Diese Unterscheidung soll eine bessere Speichereffizienz ermöglichen. Bei Erzeugung einer ThreadPoolExecutor-Instanz müssen unter anderem Angaben über die Anzahl an Core-Threads und die maximale Anzahl an Threads gemacht werden. Zur Vereinfachung werden in der Executors-Klasse im gleichen Paket unterschiedliche Fabrikmethoden bereitgestellt, mit denen sich ein Thread Pool mit fester Anzahl an Threads, ein sogenannter Cached-Thread-Pool, der für jeden eingehenden Task einen neuen Thread erzeugt, falls zu diesem Zeitpunkt kein untätiger zur Verfügung steht, sowie ein Thread Pool, der nur einen Thread enthält und die übergebenen Aufgaben sequentiell ausführt, erzeugen lässt. In Kapitel 4 werden Instanzen dieser Klasse mit fester Anzahl an Threads als Vergleich zu den kommunizierenden Thread Pools verwendet.

2.4 Verwandte Arbeiten

Da Thread Pools ein viel eingesetztes Werkzeug in der Softwareentwicklung sind, wurden bereits viele verschiedene Ansätze zu deren Optimierung untersucht. In [**?**] wird ein theoretischer Ansatz zur Berechnung der optimalen Größe eines Thread Pools vorgestellt. Dazu werden zunächst anhand gegebener Systemfaktoren die Kosten berechnet, die bei der Verwendung eines neuen Threads für jede eingehende Anfrage entstehen. Diese werden dann mit den Kosten der Verwendung eines Thread Pools der Größe n für die gleiche Zahl an Anfragen verglichen. Die Zahl der Anfragen ist dabei durch eine Wahrscheinlichkeitsverteilung gegeben. Das Ziel ist, den optimalen Wert für n zu finden, bei dem der Leistungsgewinn durch Verwendung eines Thread Pools maximal ist.

Nach [?] sind die Faktoren, die in dieser Berechnung berücksichtigt werden, zur Laufzeit nur schwer zu erfassen. Daher wird dort ein Modell zur dynamischen Anpassung eines Thread Pools zur Laufzeit präsentiert. Dieses Modell beinhaltet zwei Thread Pools: einen Basic Thread Pool und einen Extended Thread Pool. Der Basic Thread Pool verfügt über eine festgelegte Anzahl Threads zur Bearbeitung von Anfragen, der Extended Thread Pool enthält hingegen zunächst keine Threads. Die Leistung des Modells wird anhand verschiedener Faktoren und Heuristiken zur Laufzeit von einem Pool Management Modul überwacht. Sinkt die Leistung des Modells aufgrund einer steigenden Anzahl an Anfragen, werden im Extended Thread Pool weitere Threads zur Bearbeitung dieser Anfragen erzeugt. Nimmt die Anzahl der Anfragen ab, werden Threads des Extended Thread Pools blockiert. Die Anfragen werden vor der Bearbeitung in einem speziellen Warteschlangen-System zwischengespeichert, das dazu in der Lage ist, die durchschnittliche Anzahl an Anfragen sowie die durchschnittliche Antwortzeit zu berechnen. Diese Werte werden von dem Pool Management Modul bei der Verwaltung der zwei Thread Pools berücksichtigt. In diesem Modell werden zwei Thread-Pool-Instanten zur Realisierung eines Thread Pools verwendet.

Ein ähnlicher Ansatz wird in [?] verfolgt. Dort werden verschiedene Metriken zur dynamischen Optimierung der Anzahl an Threads eines Thread Pools untersucht. Außerdem wird ein Algorithmus vorgestellt, der anhand der durchschnittlichen Wartezeit von Tasks in der Warteschlange die Anzahl an Threads zur Laufzeit solange erhöht oder vermindert bis die optimale Anzahl für die aktuelle Situation erreicht ist. Dazu wird die durchschnittliche Wartezeit einer festgelegten Anzahl an Tasks t berechnet und durch die Betrachtung der durchschnittlichen Wartezeiten der vorherigen $2t$ Tasks ermittelt, ob der Thread Pool angepasst werden soll. Der Algorithmus nimmt dann für die Situation angemessene Veränderungen an der Anzahl an Threads vor.

Verbunden mit der Größe des Thread Pools ist auch die effiziente Verwendung von Systemressourcen von zentraler Bedeutung. In [?] wird ein auf Vorhersage basierendes Schema zur Optimierung eines Thread Pools vorgestellt. Die Intention ist, die optimale Anzahl an Threads zur Bearbeitung der in Zukunft eingehenden Tasks im Voraus zu schätzen, sodass möglichst wenige Systemressourcen durch untätige Threads belegt werden und gleichzeitig eine geringe durchschnittliche Antwortzeit erreicht wird. Die Vorhersage der optimalen Anzahl an Threads und gegebenenfalls die Erzeugung weiterer Threads werden dabei von einem separaten Watcher Thread durchgeführt. Außerdem werden Threads beendet, sofern sie in einer festgelegten Zeitspanne keinen weiteren Task zur Ausführung erhalten.

In [?] wird der zuvor beschriebene Ansatz erweitert. Dazu wird der Vorhersage-Mechanismus zur Schätzung der optimalen Anzahl an Threads angepasst, sodass nach den Autoren eine genauere Schätzung erreicht werden kann. Des Weiteren wird die Zeitspanne, nach der untätige Threads beendet werden, dynamisch veränderbar realisiert und die Zerstörung von Threads an weitere Bedingungen geknüpft, um die Effizienz des Thread Pools bei der Verwendung von Systemressourcen weiter zu verbessern.

Die bisher beschriebenen Ansätze untersuchen die Optimierung einer einzigen Thread-Pool-Instanz. In [?] werden vier Entwurfsmuster zur besseren Verwendung mehrerer Instanzen vorgestellt. Die Anwendung dieser Entwurfsmuster bei der Entwicklung

von parallelen Systemen soll unter anderem eine effizientere Nutzung der Systemressourcen und eine Verringerung der Ausführungszeit ermöglichen. Die Optimierung wird dort demnach durch die strukturelle Anordnung von Thread-Pool-Instanzen realisiert. In dieser Arbeit soll ein auf Kommunikation basierender, dynamischer Ansatz zur Optimierung von Thread Pools beim Einsatz mehrerer Instanzen untersucht werden.

3. Entwurf und Implementierung

Die Anzahl der Threads, die ein Thread Pool zur Ausführung von Tasks verwenden darf, ist im Regelfall identisch mit der Anzahl der Threads, die auf dem verwendeten System gleichzeitig ausgeführt werden können. Unterstützt der Prozessor des Systems nicht die Hyperthreading-Technologie, so entspricht sie der Anzahl an verfügbaren Prozessorkernen. Jeder Thread kann dann vom Betriebssystem auf einen Kern zur Ausführung abgebildet werden. Erhöht sich die Zahl der Thread Pools und damit auch die Anzahl an Threads, konkurrieren mehrere dieser Threads miteinander um Ausführungszeit auf einem Prozessorkern. Da nun nicht mehr implizit klar ist, welcher Thread auf welchem Prozessorkern ausgeführt werden soll, steigt die Anzahl an Kontextwechseln stark an, wodurch Cache-Trashing begünstigt wird. Um dies zu verhindern, sollen kommunizierende Thread Pools in der Lage sein, sich derart abzustimmen, dass zu jedem Zeitpunkt genauso viele Threads aktiv sind, wie vom Prozessor des Systems gleichzeitig ausgeführt werden können. In [**?**] ist diese Art der Koordination für zwei verschiedene Prozesse umgesetzt worden. Die Kommunikation der Thread Pools soll demnach eine Aufteilung der Systemressourcen unter den vorhandenen Instanzen ermöglichen, sodass klar ist, welcher Thread Pool dazu berechtigt ist, bestimmte Prozessorkerne zu einem Zeitpunkt zu verwenden. Dabei soll das „Verhungern“ einer Thread-Pool-Instanz vermieden werden. Ein Thread Pool „verhungert“, wenn er über einen längeren Zeitraum nicht berechtigt ist, zumindest einen Thread auszuführen, obwohl ihm bereits Tasks zugewiesen worden sind. Sind einige Threads einer Thread-Pool-Instanz zur Ausführung berechtigt, jedoch zu diesem Zeitpunkt untätig, soll die Thread-Pool-Instanz diese Berechtigungen abgeben, um anderen Instanzen eine optimale Verwendung der Systemressourcen zu ermöglichen. Gibt diese Instanz alle ihre Berechtigungen ab, muss jedoch gewährleistet sein, dass sie zumindest eine Berechtigung erhält, falls ihr weitere Tasks zur Ausführung übergeben werden.

Die kommunizierenden Thread Pools werden in dieser Arbeit in Java entwickelt. Wie in Abschnitt 2.3 bereits beschrieben, enthält die Java API eine Thread-Pool-Implementierung in Form der Klasse ThreadPoolExecutor. Kommunizierende Thread Pools sollen mit möglichst wenig Änderungen am Programmcode in denselben Anwendungsfällen eingesetzt werden können, in denen Instanzen dieser Klasse einsetzbar sind. Sie sollen deshalb durch geeignete Parameter ebenso anpassungsfähig wie

Instanzen der Klasse ThreadPoolExecutor entworfen werden. Im Folgenden werden zunächst einige Vorüberlegungen zur Implementierung der kommunizierenden Thread Pools diskutiert. Danach werden zwei verschiedene Ansätze zu dessen Entwurf näher untersucht. Die darauffolgenden Abschnitte beschreiben dann im Detail spezielle Entwurfsüberlegungen und deren Implementierungen.

3.1 Erste Überlegungen

3.1.1 Beziehung zu ThreadPoolExecutor

Die ThreadPoolExecutor-Implementierung stellt sogenannte Hook-Methoden zur Verfügung, die von Unterklassen realisiert werden können. Dadurch kann das Verhalten vor und nach der Ausführung eines Tasks und bei der Terminierung des Thread Pools angepasst werden. Es bietet sich somit an, zunächst einen Lösungsansatz zu betrachten, bei dem die kommunizierenden Thread Pools als Unterklasse der ThreadPoolExecutor-Klasse implementiert werden. Abschnitt 3.2 erläutert die damit verbundenen Vorteile und Nachteile genauer. Ein weiterer Ansatz ist, von der Oberklasse AbstractExecutorService von ThreadPoolExecutor zu erben. Die Thread-Pool-Funktionalität muss bei diesem Ansatz selbstständig implementiert werden. In Abschnitt 3.3 wird diese Vorgehensweise genauer betrachtet.

3.1.2 Scheduling

In [**?**] wird die Ausführung eines LoadTask-Prozesses und eines Sortierprozesses auf einem Mehrkernprozessor koordiniert. Dabei wird der LoadTask-Prozess nach verschiedenen Mustern auf den einzelnen Prozessorkernen ausgeführt und der Sortierprozess zu diesen Zeitpunkten auf den nicht genutzten Kernen fortgesetzt. Das Scheduling wird demnach nicht vom Betriebssystem, sondern bewusst manuell durchgeführt. In dieser Arbeit soll es zunächst Aufgabe des Betriebssystems bleiben, Threads bestimmten Rechenkernen zur Ausführung zuzuordnen. Die Kommunikation unter den Thread Pools beschränkt sich daher auf die Anzahl an Ressourcen, die eine Instanz zu einem Zeitpunkt verwenden darf. Eine Ressource stellt dabei im Regelfall einen Prozessorkern dar, kann aber auch als Abstraktion anderer in der Anzahl begrenzter Systemressourcen gesehen werden. Für jede Ressource, die einem kommunizierenden Thread Pool zugewiesen ist, erhält dieser die Erlaubnis einen Thread zu erstellen und auszuführen. Bleibt die erhoffte Leistungssteigerung bei dieser Vorgehensweise aus, kann die Implementierung nachträglich um explizites Scheduling erweitert werden.

3.1.3 Implementierung der Kommunikation

Eine Leistungssteigerung durch Kommunikation zwischen den einzelnen Thread-Pool-Instanzen ist nur dann zu erwarten, wenn der Mehraufwand, der dadurch entsteht, gering gehalten wird. Es erscheint daher sinnvoll, die Kommunikation über eine weitere Klasse zu realisieren, die als zentraler Punkt die Aufteilung der Ressourcen unter den Thread Pools organisiert. Bei diesem Ansatz ist es nicht erforderlich, dass eine Thread-Pool-Instanz Kenntnis von weiteren besitzt. Sie muss lediglich von der Kommunikationsklasse erfahren, wie viele Ressourcen bzw. Threads sie zum aktuellen Zeitpunkt verwenden darf.

3.2 Erster Lösungsansatz

Wie in Abschnitt 3.1.1 angedeutet, soll zunächst der Ansatz geprüft werden, die vorhandene Implementierung in Form der ThreadPoolExecutor-Klasse mittels Vererbung um die gewünschte Funktionalität zu erweitern. Die neue Unterklasse (im Folgenden CommunicativeThreadPoolExecutor genannt) kann in diesem Fall die zur Verfügung stehenden Hook-Methoden implementieren und auf diesem Weg das Verhalten vor und nach jeder Taskausführung sowie bei Terminierung des Thread Pools um geeignete Kommunikationsfunktionalität ergänzen.

Die Kommunikation soll nach den Überlegungen in Abschnitt 3.1.3 von einer weiteren Klasse zentral organisiert werden. Zu diesem Zweck wird die Klasse ThreadPoolCommunicationCenter definiert. Sie soll Methoden zur Verfügung stellen, mit denen die einzelnen Thread-Pool-Instanzen Ressourcen abgeben und erfragen können. Um eine hohe Anzahl von Nachfragen zu vermeiden, erscheint es sinnvoll sie gemäß des Beobachter-Entwurfsmusters zu implementieren, wobei ThreadPoolCommunicationCenter die Rolle des Subjekts und jede Thread-Pool-Instanz die Rolle eines Beobachters einnimmt (vgl. [**?**]). Die ThreadPoolCommunicationCenter-Instanz informiert ihre Beobachter darüber, ob aktuell freie Ressourcen zur Verfügung stehen oder nicht. Wie diese Information bezüglich der Nachfrage oder Abgabe von Ressourcen seitens einer Thread-Pool-Instanz verwertet wird, soll von einer AbstractResourceHandler-Klasse gemäß des Strategie-Entwurfsmusters ([**?**]) entschieden werden. Ist eine nach Ressourcen fragende Thread-Pool-Instanz vollständig blockiert und stehen zu diesem Zeitpunkt keine freien Ressourcen zur Verfügung, die der Instanz zugewiesen werden können, soll die ThreadPoolCommunicationCenter-Instanz die insgesamt vorhandenen Ressourcen neu auf alle existierenden Thread-Pool-Instanzen verteilen. Zu diesem Zweck wird eine AbstractResourceDistributionStrategy-Klasse definiert, die ebenfalls entsprechend des Strategie-Entwurfsmusters verwendet werden soll. Die Implementierung des Strategie-Entwurfsmusters ermöglicht dabei, das Verhalten in entsprechenden Situationen flexibel anzupassen. In Abbildung 3.1 ist das soeben Beschriebene zur Veranschaulichung als UML-Klassendiagramm dargestellt.

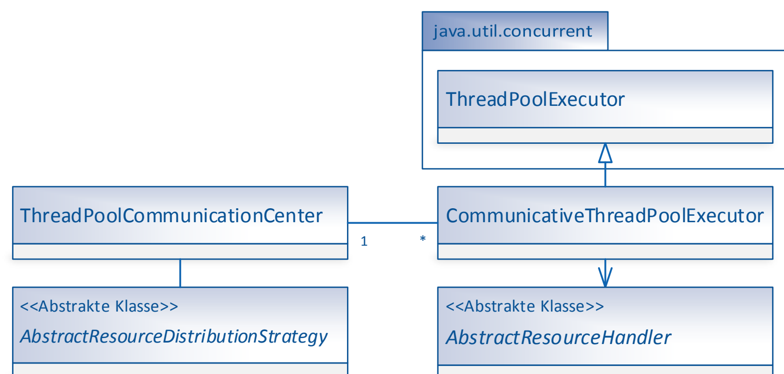


Abbildung 3.1: Die Struktur des ersten Lösungsansatzes, dargestellt in Form eines UML-Klassendiagramms. Die Klasse CommunicativeThreadPoolExecutor soll die bereits im Paket java.util.concurrent der Java API enthaltene Klasse ThreadPoolExecutor um Kommunikationsfunktionalität erweitern.

3.2.1 Diskussion der Vor- und Nachteile

Der größte Vorteil des Ansatzes der direkten Vererbung der ThreadPoolExecutor-Klasse besteht darin, dass Code-Redundanz vermieden wird. Ein großer Teil des vorhandenen Codes kann wiederverwendet und somit der Aufwand für das Implementieren und Testen der kommunizierenden Thread Pools deutlich verringert werden. Des Weiteren kann eine CommunicativeThreadPoolExecutor-Instanz mit nur geringen Änderungen des bestehenden Programmcodes überall dort eingesetzt werden, wo eine ThreadPoolExecutor-Instanz verlangt wird.

Unglücklicherweise sind mit diesem Ansatz jedoch auch einige Probleme verbunden. Die ThreadPoolExecutor-Implementierung sieht es vor, dass neue Threads in der *execute*-Methode mit einem Initialtask erzeugt und der Thread-Menge hinzugefügt werden. Dieses Verhalten kann durch eine Unterklasse nicht zufriedenstellend geändert werden, da für sie die Thread-Menge und die innere Worker-Klasse, die die Threads repräsentiert, nicht sichtbar sind. Wird nun ein Thread mit Initialtask erzeugt und durch Anweisungen der ThreadPoolCommunicationCenter-Instanz blockiert, wird der Task im ungünstigsten Fall überhaupt nicht ausgeführt. Mit der Verwendung einer höheren Anzahl an Core-Threads und der Methode *prestartCoreThread* lässt sich dieses Problem zwar umgehen, aber nicht zufriedenstellend lösen. Die Methode *prestartCoreThread* erzeugt einen weiteren Core-Thread, sofern die maximale Anzahl an Core-Threads nicht bereits erreicht ist. Weiter erscheint zumindest die Möglichkeit sinnvoll, blockierte Nicht-Core-Threads nach einer gewissen Zeit zu beenden, um die Thread-Pool-Instanz möglichst ressourcenschonend zu verwenden. Die ThreadPoolExecutor-Klasse beendet Nicht-Core-Threads, wenn diese in einer durch einen Parameter festgelegten Zeitspanne keinen neuen Task zugewiesen bekommen. Threads, die in der CommunicativeThreadPoolExecutor-Implementierung blockiert sind, können zwar beendet, jedoch nicht aus der Thread-Menge entfernt werden, da die Thread-Menge wie bereits erwähnt für die Instanz der Unterklasse nicht sichtbar ist. Ein weiteres Problem ist, dass in der ThreadPoolExecutor-Klasse ein für Unterklassen nicht sichtbares explizites Lock-Objekt anstelle des ThreadPoolExecutor-Objekts zur Synchronisation der Threads verwendet wird. Für weitere Synchronisation müssen daher gegebenenfalls andere Objekte verwendet werden, sodass die Implementierung dadurch anfälliger für Fehler wird.

Aufgrund der nicht vernachlässigbaren Probleme, die bei der Implementierung des ersten Ansatzes entstehen, wird in dieser Arbeit ein anderer Ansatz verfolgt, der im folgenden Abschnitt näher erklärt werden soll.

3.3 Zweiter Lösungsansatz

Im Gegensatz zu dem in Abschnitt 3.2 beschriebenen ersten Lösungsansatz soll hier nicht von der Klasse ThreadPoolExecutor geerbt werden, sondern von deren abstrakter Oberklasse AbstractExecutorService. Sie implementiert die ExecutorService-Schnittstelle und stellt bereits Implementierungen einiger Methoden dieser Schnittstelle zur Verfügung. Die Thread-Pool-Funktionalität muss vollständig selbst implementiert werden, sodass dieser Ansatz deutlich höheren Implementierungs- und Testaufwand beinhaltet. Die Probleme des ersten Ansatzes sind aus diesem Grund jedoch nicht vorhanden. Weiter lässt sich die Implementierung durch die Vererbung der direkten Oberklasse von ThreadPoolExecutor ebenso mit wenigen Änderungen des bestehenden Programmcodes in den Anwendungsfällen einsetzen, in denen

Instanzen von ThreadPoolExecutor verwendet werden können. Die Kommunikation der Thread-Pool-Instanzen soll ebenso realisiert werden wie im ersten Lösungsansatz beschrieben. Zur besseren Übersicht zeigt Abbildung 3.2 die Struktur des zweiten Ansatzes in Form eines UML-Klassendiagramms. In den folgenden Abschnitten wird nun die Umsetzung dieses Lösungsansatzes im Detail beschrieben.

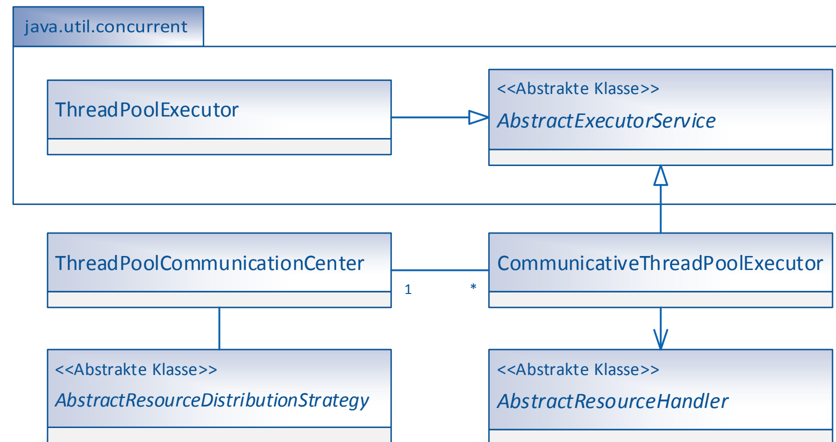


Abbildung 3.2: Die Struktur des zweiten Lösungsansatzes, dargestellt in Form eines UML-Klassendiagramms. Die Klassen ThreadPoolExecutor und AbstractExecutorService sind bereits im Paket java.util.concurrent in der Java API enthalten. Die Klasse CommunicativeThreadPoolExecutor soll AbstractExecutorService um Thread-Pool-Funktionalität und Kommunikationsfunktionalität erweitern.

3.4 Die Klasse CommunicativeThreadPoolExecutor

3.4.1 Task- und Thread-Verwaltung

Die wichtigsten Aspekte jeder Thread-Pool-Implementierung sind die Task- und die Thread-Verwaltung. Entscheidend für die Task-Verwaltung ist die Implementierung der Warteschlange, für die Thread-Verwaltung die Erzeugung und Repräsentation der Threads.

3.4.1.1 Implementierung der Warteschlange

Eingehende Tasks werden bei dieser Implementierung zunächst in eine Warteschlange eingereiht, in der sie verbleiben, bis sie von einem Thread ausgeführt werden können. Threads müssen auf die Ankunft weiterer Tasks warten, sofern alle vorhandenen Tasks abgearbeitet worden sind und sich keine weiteren in der Warteschlange befinden. Die Warteschlange kann demnach als eine Art Synchronisationspunkt angesehen werden und sollte dementsprechende Funktionen unterstützen. In Java kann zu diesem Zweck jede Realisierung der Schnittstelle BlockingQueue aus dem Paket java.util.concurrent verwendet werden.

3.4.1.2 Repräsentation der Threads

Um die Thread-Verwaltung übersichtlicher zu gestalten, wird zunächst eine innere Klasse Worker eingeführt, die die Schnittstelle Runnable implementiert. Bei der Erzeugung einer neuen Worker-Instanz wird mittels einer ThreadFactory-Instanz dann ein neues Thread-Objekt erzeugt und gestartet. Die ThreadFactory-Instanz kann als Parameter bei der Erzeugung einer Thread-Pool-Instanz festgelegt werden. Die Worker-Instanz dient zusätzlich als Runnable-Objekt, das dem Thread-Objekt bei Erzeugung übergeben wird. Der auf diesem Weg gestartete Thread führt die *run*-Methode der Worker-Instanz aus, in der er in einer Schleife solange weitere Tasks der Warteschlange entnimmt und ausführt, bis er unterbrochen wird. Die Worker-Klasse stellt außerdem Methoden zur Unterbrechung des Threads zur Verfügung. Erzeugte Worker-Instanzen werden in einer Menge gespeichert. Dazu wird ein Set-Objekt basierend auf einer ConcurrentHashMap (java.util.concurrent) verwendet.

3.4.1.3 Berechtigungen und blockierte Threads

Für jede CommunicativeThreadPoolExecutor-Instanz muss bei dessen Erzeugung eine Referenz zu einer ThreadPoolCommunicationCenter-Instanz angegeben werden, da sonst keine Kommunikation stattfinden kann. Ihre Ressourcennutzung wird dann mit den anderen Thread-Pool-Instanzen, die ebenfalls von dieser Instanz verwaltet werden, abgestimmt. Bevor ein neuer Thread erzeugt werden kann, muss die Thread-Pool-Instanz zunächst bei dieser Instanz nachfragen, ob eine Ressource zur Verfügung steht. Ist dies der Fall, wird die Anzahl der Berechtigungen dieser Thread-Pool-Instanz zur Ausführung von Threads inkrementiert. Im Laufe der Zeit kann die Anzahl der Berechtigungen zu- und abnehmen. Werden sie verringert müssen gegebenenfalls einige Threads blockiert werden. Zu diesem Zweck muss jeder Thread, bevor er der Warteschlange einen weiteren Task entnehmen kann, ein SidelinesSemaphore-Objekt passieren. Die SidelinesSemaphore ist eine veränderte Version einer gewöhnlichen Semaphore (java.util.concurrent), bei der eine Anzahl an Threads pausiert werden kann. Die Implementierungsdetails der SidelinesSemaphore-Klasse werden genauer in Abschnitt 3.4.4 beschrieben.

Zum besseren Verständnis sind in Abbildung 3.3 die Interaktionen der beteiligten Objekte bei der Erzeugung und Ausführung eines neuen Threads in Form eines Sequenzdiagramms dargestellt.

3.4.1.4 Beenden von Threads

Java-Threads sind beendet, wenn die *run*-Methode des zugehörigen Runnable-Objekts ausgeführt wurde. Sie können danach nicht erneut gestartet werden. Aus diesem Grund enthält die *run*-Methode einer Worker-Instanz eine Schleife, die bis zum Beenden des Thread Pools ausgeführt wird. Die Thread-Pool-Implementierung bietet zwei verschiedene Methoden zum asynchronen Beenden an: *shutdown* und *shutdownNow* (siehe 3.4.2.3). Da sich zum Zeitpunkt des Aufrufs dieser Methoden einige Threads im schlafenden Zustand befinden können, müssen die Threads mit Hilfe der *interrupt*-Methode unterbrochen werden. Bevor der Thread dann endgültig beendet ist, entfernt er die zugehörige Worker-Instanz aus der Menge, die der Thread Pool verwaltet.

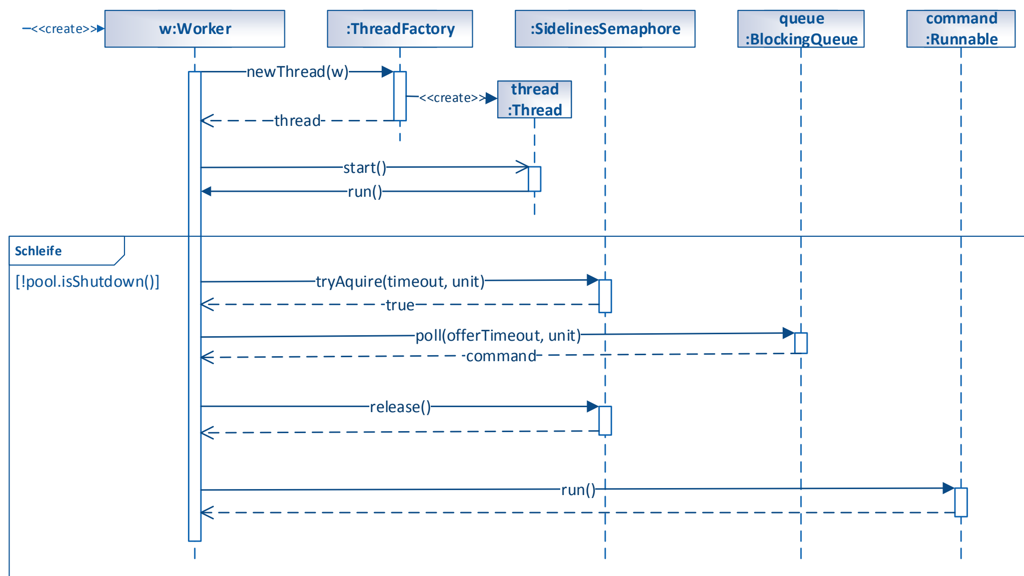


Abbildung 3.3: Vereinfachte Darstellung der Interaktionen der beteiligten Objekte bei der Erzeugung und Ausführung eines neuen Threads. Bei Konstruktion einer Worker-Instanz wird ein neuer Thread mit Hilfe einer ThreadFactory-Instanz erzeugt und gestartet. Der Thread fragt bei der Warteschlange dann solange iterativ nach weiteren Tasks bis der Thread Pool beendet wird. Außerdem prüft der Thread bei jedem Schleifendurchgang, durch Interaktion mit einem SidelinesSemaphore-Objekt, ob er momentan zur Ausführung berechtigt ist.

3.4.2 Implementierung der wichtigsten Methoden

In diesem Abschnitt soll näher auf die Implementierung der wichtigsten öffentlichen Methoden eingegangen werden.

3.4.2.1 Vererbung von AbstractExecutorService

Die Verwendung von AbstractExecutorService als Oberklasse erfordert die Implementierung der folgenden Methoden der ExecutorService-Schnittstelle:

void execute(Runnable) Methode zur Übergabe von Tasks zur Ausführung.

void shutdown() Methode zum asynchronen Beenden des ExecutorService. Nach Aufruf dieser Methode werden alle weiteren an diesen ExecutorService übergebenen Tasks abgewiesen. Die zuvor übergebenen Tasks werden ausgeführt und der ExecutorService danach beendet.

List<Runnable> shutdownNow() Methode zum sofortigen asynchronen Beenden des ExecutorService. Nach Aufruf dieser Methode werden alle weiteren an diesen ExecutorService übergebenen Tasks abgewiesen. Es wird eine Liste der Tasks zurückgegeben, die sich zum Zeitpunkt des Aufrufs dieser Methode in der Warteschlange befinden. Tasks, die zurzeit von einem Thread ausgeführt werden, können unterbrochen oder beendet werden.

boolean awaitTermination(long, TimeUnit) Methode zum Warten auf die Terminierung des ExecutorService.

boolean isShutdown() Methode zum Überprüfen, ob eine der beiden *shutdown*-Methoden aufgerufen wurde.

boolean isTerminated() Methode zum Überprüfen, ob der ExecutorService beendet ist.

Im Folgenden soll genauer auf die Implementierung dieser Methoden eingegangen werden.

3.4.2.2 Die *execute*-Methode

Die Hauptaufgabe der *execute*-Methode ist die Übermittlung von Tasks an die Thread-Pool-Instanz. Es wird daher zunächst geprüft, ob bereits eine der Methoden zum Beenden des Thread Pools aufgerufen wurde. Ist dies der Fall, wird der Task an eine RejectionHandler-Instanz (siehe 3.4.6) weitergereicht, die diesen abweist. Kann der übergebene Task akzeptiert werden, wird er in die Warteschlange eingereiht. Ein dort eventuell zu diesem Zeitpunkt wartender Thread kann ihn dann bereits entgegennehmen und mit der Ausführung beginnen. Danach wird die AbstractResourceHandler-Instanz (siehe 3.4.5) damit beauftragt durch Interaktion mit der ThreadPoolCommunicationCenter-Instanz zu ermitteln, ob ein weiterer Thread zur Ausführung von Tasks erzeugt werden kann. Gegebenenfalls wird dann eine weitere Worker-Instanz konstruiert und der bestehenden Menge hinzugefügt. Zur Verdeutlichung der Funktionsweise der *execute*-Methode werden in Abbildung 3.4 die zuvor beschriebenen Interaktionen in einem Sequenzdiagramm dargestellt.

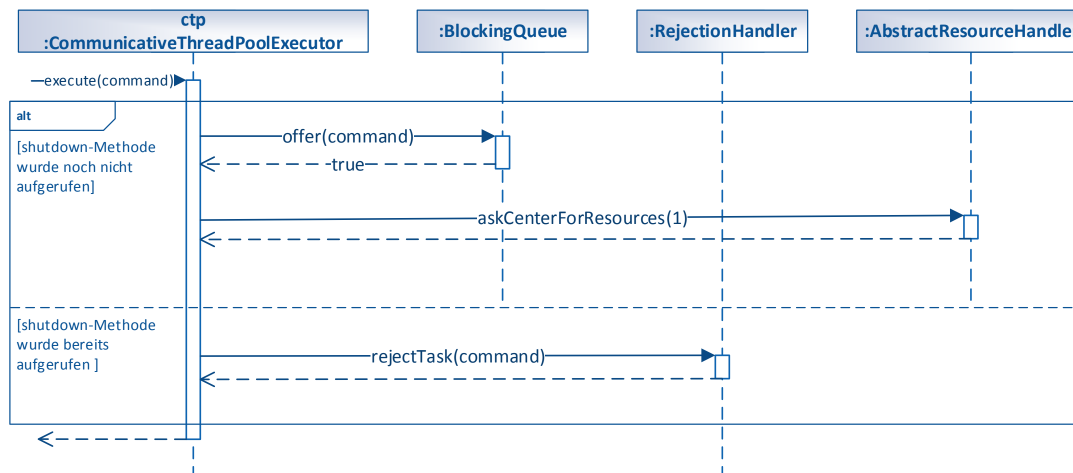


Abbildung 3.4: Vereinfachte Darstellung der Interaktionen der beteiligten Objekte bei der Übergabe eines Tasks unter Verwendung der *execute*-Methode. Wurde zum Zeitpunkt des Aufrufs der *execute*-Methode noch keine *shutdown*-Methode aufgerufen, wird der übergebene Task in die Warteschlange eingereiht und über Interaktion mit der AbstractResourceHandler-Instanz entschieden, ob nach einer weiteren Ressource gefragt werden soll. Andernfalls wird der Task unter Verwendung einer RejectionHandler-Instanz abgewiesen.

3.4.2.3 Die Methoden *shutdown* und *shutdownNow*

Bei *shutdown* und *shutdownNow* handelt es sich um Methoden, die dem Thread Pool die Anweisung geben, sich zu beenden. Sie unterscheiden sich darin, dass bei

Aufruf der *shutdown*-Methode zunächst alle bisher eingegangenen Tasks ausgeführt werden sollen, bevor der Thread Pool beendet wird, wohingegen *shutdownNow* die schnellstmögliche Terminierung des Thread Pools und die Rückgabe der noch ausstehenden Tasks vorsieht. Um auch schlafende Threads über den Aufruf dieser Methoden zu informieren, müssen sie mittels *interrupt*-Methode unterbrochen werden. Da die Implementierungsdetails eines Tasks nicht bekannt sind, ist nicht klar wie dessen Ausführung durch den Aufruf der *interrupt*-Methode beeinflusst wird. Aus diesem Grund darf bei Aufruf von *shutdown* der Aufruf von *interrupt* nur unter gegenseitigem Ausschluss mit der Ausführung eines Tasks durchgeführt werden. In dieser Implementierung wird dieses Verhalten mit Hilfe des Locks der Worker-Instanz sichergestellt. Bei Verwendung der *shutdownNow*-Methode kann ein Thread hingegen auch während der Ausführung eines Tasks unterbrochen werden.

3.4.2.4 Warten auf das Ende - die *awaitTermination*-Methode

Da die Methoden *shutdown* und *shutdownNow* den Thread Pool asynchron beenden, d.h. nicht auf die Terminierung des Thread Pools warten, wird zu diesem Zweck die Methode *awaitTermination* angeboten. Hier wird sie mit Hilfe eines CountDownLatch-Objekts (java.util.concurrent) implementiert, das den aufrufenden Thread solange blockiert, bis sich der letzte Thread des Thread Pools beendet und die *countDown*-Methode dieses Objekts aufruft.

3.4.2.5 Die Methoden *isShutdown* und *isTerminated*

Die Implementierung des Thread Pools verwaltet einige Zustände, die in Abschnitt 3.4.3 näher beschrieben werden. Die Methoden *isShutdown* und *isTerminated* können dann überprüfen, ob sich der Thread Pool zum Zeitpunkt des Aufrufs in dem entsprechenden Zustand befindet und das Ergebnis zurückgeben.

3.4.3 Zustände

Wie zuvor bereits angedeutet kann der Thread Pool im Laufe der Zeit unterschiedliche Zustände erreichen, die z.B. anzeigen, ob er beendet ist oder keine weiteren Tasks mehr akzeptiert. Es wird unter den folgenden vier Zuständen unterschieden:

RUNNING Der Thread Pool wird bei Instanziierung in diesen Zustand versetzt. Es können Tasks angenommen und ausgeführt werden.

SHUTDOWN Bei Aufruf der *shutdown*-Methode wird der Thread Pool in diesen Zustand versetzt. Es werden keine weiteren ankommenden Tasks akzeptiert. Wenn die Warteschlange leer ist, werden die Threads beendet.

SHUTDOWN_NOW Wird die *shutdownNow*-Methode aufgerufen, erreicht der Thread Pool diesen Zustand. Auch in diesem Zustand wird jeder weitere eingehende Task abgewiesen. Die Threads werden, unabhängig von der in der Warteschlange vorhandenen Menge an Tasks, beendet, sobald es möglich ist.

TERMINATED Der Thread Pool ist beendet. Es werden keine weiteren Tasks akzeptiert oder ausgeführt. Alle Threads dieses Thread Pools sind ebenfalls beendet. Die Menge der Worker-Instanzen ist leer.

Es ist nicht möglich nach Verlassen des *RUNNING*-Zustands wieder in diesen zurückzukehren. Zur besseren Übersicht stellt Abbildung 3.5 die vorhandenen Zustände sowie deren Übergänge grafisch dar.

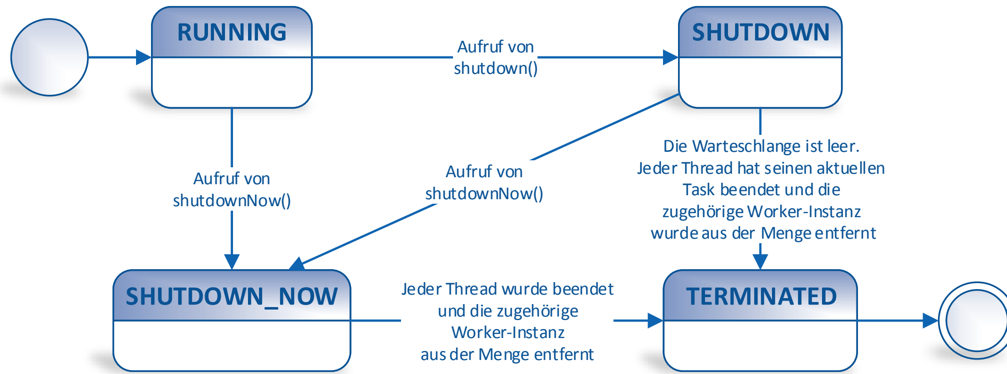


Abbildung 3.5: Die Zustände einer CommunicativeThreadPoolExecutor-Instanz und deren Übergänge

3.4.4 Die innere Klasse SidelinesSemaphore

In Abschnitt 3.4.1.3 wurde beschrieben, dass Threads zuerst ein Objekt der Klasse SidelinesSemaphore passieren müssen, bevor sie der Warteschlange einen Task entnehmen dürfen. Das SidelinesSemaphore-Objekt legt fest, wie viele Threads zum aktuellen Zeitpunkt zu der Ausführung von Tasks berechtigt sind. Die Klasse SidelinesSemaphore bietet dazu ebenso wie eine gewöhnliche Semaphore die Methoden *aquire* und *release* an. Threads, die beim Aufruf der *aquire*-Methode keine Berechtigung erhalten können, werden jedoch solange pausiert, bis die Gesamtanzahl an Berechtigungen erhöht wird. Dieses Verhalten wird durch Delegation an zwei Semaphore-Instanzen realisiert. Die erste repräsentiert die aktuelle Anzahl an Berechtigungen, die zweite blockiert Threads, die aktuell keine Berechtigung erhalten können. Ruft ein Thread die *aquire*-Methode des SidelinesSemaphore-Objektes auf, wird zunächst bei der ersten Semaphore nach einer Berechtigung gefragt. Ist zu diesem Zeitpunkt keine weitere Berechtigung zu vergeben, wird der Thread an die Seitenlinie (engl. on the sidelines), repräsentiert durch die zweite Semaphore, geschickt. Dort wird er blockiert bis die Berechtigungen von der ThreadPoolCommunicationCenter-Instanz erhöht werden. Das Ziel dabei ist die Reduzierung von Kontextwechseln, erreicht dadurch, dass Threads, die gerade einen Task ausgeführt haben, erneut eine Berechtigung zur Ausführung eines nächsten erhalten. Eine gewöhnliche Semaphore würde bei Aufruf der *release*-Methode durch einen Thread A einen zu diesem Zeitpunkt wartenden Thread B fortsetzen. Bei Aufruf der *aquire*-Methode wird A dann in den schlafenden Zustand versetzt. Es muss demnach ein unnötiger Kontextwechsel von A nach B durchgeführt werden. Da gerade die Reduzierung von Kontextwechseln ein Ziel der Kommunikation der Thread-Pool-Instanzen ist, ist eine gewöhnliche Semaphore für die Implementierung nicht geeignet. Die SidelinesSemaphore-Klasse stellt außerdem Methoden zur dynamischen Veränderung der Anzahl an Berechtigungen zur Verfügung. Bei Erhöhung der Berechtigungen werden einige der an der zweiten Semaphore-Instanz wartenden Threads mittels Aufruf der *release*-Methode fortgesetzt. Die erwachten Threads erhöhen danach die Berechtigungen der ersten Semaphore ebenfalls durch Aufruf der *release*-Methode, sodass sie bei folgendem Aufruf der *aquire*-Methode der SidelinesSemaphore-Instanz nicht erneut an die Seitenlinie geschickt werden.

3.4.5 Ressourcenverwaltung

Die Anzahl der Threads, die zu einem Zeitpunkt dazu berechtigt sind, der Warteschlange Tasks zu entnehmen und auszuführen, hängt davon ab, wie viele Ressourcen die CommunicativeThreadPoolExecutor-Instanz von ihrer ThreadPoolCommunicationCenter-Instanz zugewiesen bekommt. Die Ressourcenverwaltung übernimmt dabei eine Instanz einer Unterklasse der inneren Klasse AbstractResourceHandler. Sie entscheidet, ob eine Ressource abgegeben oder um weitere gebeten werden soll. Bei der Einreihung eines neuen Tasks in die Warteschlange und nach der erfolgreichen Ausführung eines Tasks wird sie kontaktiert, um zu prüfen, ob nach weiteren Ressourcen gefragt werden soll. Weiter entscheidet die Instanz, ob Ressourcen abgegeben werden, falls ein Thread in einer gewissen Zeitspanne keinen neuen Task erhält. Es werden drei verschiedene Unterklassen zur Verfügung gestellt: DefaultResourceHandler, KeepCoreResourceHandler und ManuallyPermitOfferingResourceHandler. Die Bedingungen, die erfüllt sein müssen, um nach weiteren Ressourcen zu fragen, sind bei diesen drei Klassen identisch:

1. Die Warteschlange darf zum Zeitpunkt des Aufrufs der Methode *askCenterForResources* nicht leer sein.

2. Die ThreadPoolCommunicationCenter-Instanz muss aktuell über freie Ressourcen verfügen.

3. Es sind aktuell weniger Threads zur Ausführung von Tasks berechtigt als der Thread Pool maximal verwenden darf.

Sind keine freien Ressourcen verfügbar und ist der Thread Pool zum Zeitpunkt des Aufrufs vollständig blockiert, wird trotzdem nach Ressourcen gefragt, um das „Verhungern“ dieses Thread Pools zu verhindern. Die ThreadPoolCommunicationCenter-Instanz wird in diesem Fall eine Neuverteilung der vorhandenen Ressourcen vornehmen (siehe 3.5.3). Bei der Ressourcenabgabe unterscheidet sich das Verhalten der drei Klassen. Eine Instanz der Klasse DefaultResourceHandler gibt bei jedem Aufruf der *offerResourcesToCenter*-Methode die angegebene Anzahl an Ressourcen ab, wohingegen eine Instanz der Klasse KeepCoreResourceHandler nur dann welche abgibt, wenn aktuell auch Nicht-Core-Threads dazu berechtigt sind, der Warteschlange Tasks zu entnehmen und auszuführen. Wird stattdessen eine ManuallyPermitOfferingResourceHandler-Instanz verwendet, kann der Anwender explizit angeben, ob Ressourcen an einem bestimmten Zeitpunkt abgegeben werden sollen.

3.4.6 Abweisen von Tasks

Verlässt eine CommunicativeThreadPoolExecutor-Instanz den Zustand *RUNNING*, beispielsweise durch Aufruf der *shutdown*-Methode, werden alle weiteren eingehenden Tasks abgewiesen. Das Verhalten beim Abweisen eines Tasks wird dabei von einer RejectionHandler-Instanz festgelegt. RejectionHandler ist eine Schnittstelle, die die Implementierung einer *rejectTask*-Methode verlangt. Die inneren Klassen CancelOnRejectionPolicy, RunsInCallerRejectionPolicy und DiscardPolicy implementieren diese Schnittstelle und stellen einige einfache Möglichkeiten zur Verfügung wie

Tasks sinnvoll abgewiesen werden können. Wird eine Instanz der Klasse CancelOnRejectionPolicy verwendet, wird bei jedem abzuweisenden Task eine RejectedExecutionException (java.util.concurrent) geworfen. Eine RunsInCallerRejectionPolicy-Instanz führt den abzuweisenden Task in dem Thread aus, der ihn gesendet hat und eine DiscardPolicy-Instanz verwirft den Task ohne weitere Konsequenzen. Für die Klasse ThreadPoolExecutor sind vergleichbare Klassen implementiert worden. Diese sind hier jedoch nicht einsetzbar, da deren Methode zum Abweisen von Tasks eine Instanz der Klasse ThreadPoolExecutor als Parameter benötigt.

3.5 Die Klasse ThreadPoolCommunicationCenter

3.5.1 Verwaltung der Thread-Pool-Instanzen

Die Klasse ThreadPoolCommunicationCenter organisiert die Kommunikation zwischen CommunicativeThreadPoolExecutor-Instanzen. Dazu benötigt eine Instanz dieser Klasse Referenzen zu den Thread-Pool-Instanzen, die miteinander abgestimmt werden sollen. Sie verwaltet daher eine Menge, der diese Instanzen hinzugefügt werden müssen. Die Menge wird hier durch ein Set-Objekt basierend auf einer ConcurrentHashMap (java.util.concurrent) realisiert. Wird eine Instanz beendet, muss sie aus dieser Menge wieder entfernt werden. CommunicativeThreadPoolExecutor-Instanzen rufen dazu die Methode *addCommunicativeThreadPoolExecutor* bei ihrer Erzeugung und die Methode *removeCommunicativeThreadPoolExecutor* bei ihrer Terminierung auf.

3.5.2 Methoden zur Kommunikation

Die ThreadPoolCommunicationCenter-Instanz unterhält eine Variable, die die Anzahl der aktuell frei zur Verfügung stehenden Ressourcen repräsentiert. Die Anzahl ist durch einen festen Wert begrenzt, der bei der Instanziierung als Parameter festgelegt werden kann. Sobald dann eine Thread-Pool-Instanz hinzugefügt wird, wird ihr von der ThreadPoolCommunicationCenter-Instanz mitgeteilt, ob aktuell freie Ressourcen zur Verfügung stehen. Der neuen Instanz werden jedoch noch keine Ressourcen zugewiesen, da nicht klar ist, wann ihr Tasks zur Ausführung gesendet werden. Wird eine Thread-Pool-Instanz beendet und aus der Menge entfernt, wird die Anzahl an frei verfügbaren Ressourcen um die Anzahl derer erhöht, die dieser Instanz zugewiesen waren. Waren zuvor keine Ressourcen verfügbar, werden die in der Menge verbliebenen Thread Pools darüber informiert, dass neue Ressourcen zur Verfügung stehen. Es steht ihnen dann frei zu entscheiden, ob sie nach Ressourcen fragen oder nicht (siehe 3.4.5).

Das Fragen nach Ressourcen ist mit Hilfe der Methode *askForAvailableResources* möglich. Die Thread-Pool-Instanz gibt dabei an, wie viele sie maximal benötigt. Es wird dann überprüft, ob freie Ressourcen zur Verfügung stehen. Ist dies der Fall werden der Thread-Pool-Instanz möglichst viele zugewiesen. Sind zu dem Zeitpunkt der Anfrage keine Ressourcen verfügbar, hängt das Verhalten der Methode davon ab, ob der anfragende Thread Pool vollständig blockiert ist, d.h. ihm aktuell keine Ressourcen zugewiesen sind. Um das „Verhungern“ dieses Thread Pools zu vermeiden, wird in diesem Fall eine neue Verteilung der Ressourcen über alle zu verwaltenden Thread Pools vorgenommen. Dies geschieht durch Delegation an eine Instanz der Klasse AbstractResourceDistributionStrategy, die in Abschnitt 3.5.3 näher beschrieben wird.

Ist der Thread Pool nicht vollständig blockiert, erhält er keine weiteren Ressourcen. Zur besseren Übersicht stellt Abbildung 3.6 das zuvor Beschriebene grafisch in Form eines UML-Sequenzdiagramms dar.

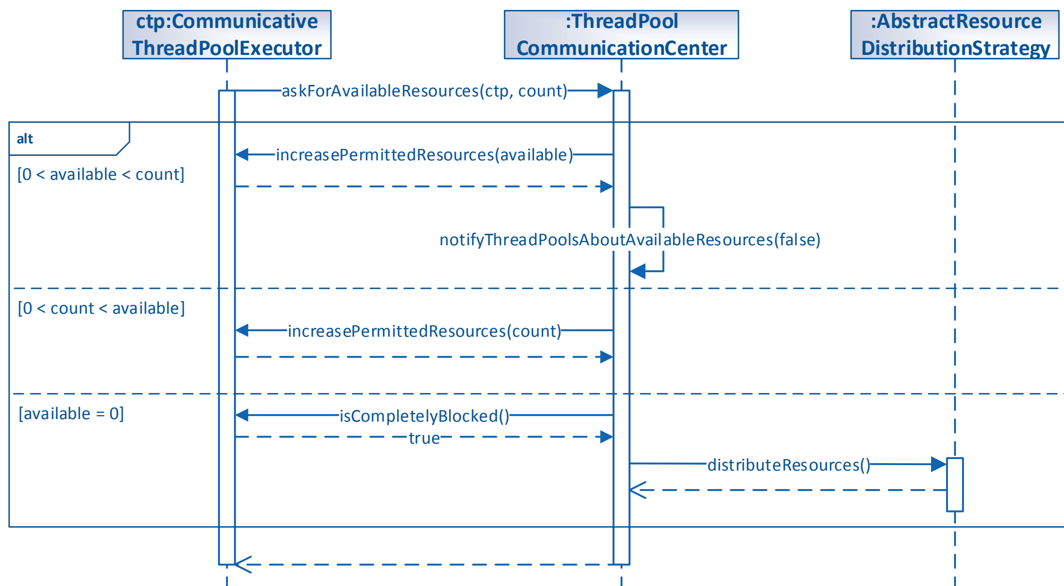


Abbildung 3.6: Vereinfachte Darstellung der Interaktionen der beteiligten Objekte bei der Anfrage eines Thread-Pools nach weiteren Ressourcen. Sofern welche zur Verfügung stehen, werden der Thread-Pool-Instanz möglichst viele zugewiesen. Sind daraufhin keine weiteren Ressourcen verfügbar, werden die anderen Instanzen darüber informiert. Sind zum Zeitpunkt des Aufrufs keine Ressourcen verfügbar und ist die anfragende Thread-Pool-Instanz vollständig blockiert, werden die vorhandenen Ressourcen durch Interaktion mit einer AbstractResourceDistributionStrategy-Instanz über alle Thread-Pool-Instanzen neu verteilt.

Die Abgabe von Ressourcen ist mit der Methode *offerResources* möglich. Über einen Parameter kann angegeben werden, wie viele Ressourcen abgegeben werden. Die ThreadPoolCommunicationCenter-Instanz addiert die Anzahl der nun wieder frei zur Verfügung stehenden Ressourcen zu der bisherigen hinzu. Waren bei Aufruf keine Ressourcen verfügbar, werden die aktuell zu verwaltenden Thread-Pool-Instanzen über die neu verfügbaren informiert.

3.5.3 Ressourcenverteilung

Wie in Abschnitt 3.5.2 bereits angedeutet, sollen die Ressourcen in bestimmten Situationen neu verteilt werden, um das „Verhungern“ einzelner Thread Pools zu verhindern. Für die Neuverteilung ist eine Instanz der Klasse AbstractResourceDistributionStrategy verantwortlich. Gemäß des Strategie-Entwurfsmusters können konkrete Unterklassen dieser abstrakten Klasse implementiert und eingesetzt werden. Im Standardfall wird eine Instanz der bereits implementierten inneren Klasse DefaultResourceDistributionStrategy eingesetzt, die bei Aufruf ihrer *distributeResources*-Methode eine möglichst gleichmäßige Verteilung über alle Thread Pools, die in dieser ThreadPoolCommunicationCenter-Instanz verwaltet werden, realisieren soll. Jedem Thread Pool wird dabei zunächst die gleiche Anzahl an Ressourcen

zugewiesen. Übersteigt die Anzahl die Grenze der maximal einsetzbaren Threads eines Thread Pools, wird der verbleibende Rest der Anzahl freier Ressourcen hinzugefügt. Nach der Verteilung werden die Thread Pools dann davon in Kenntnis gesetzt, ob weitere freie Ressourcen zur Verfügung stehen oder nicht. Nach diesen kann dann mittels der in 3.5.2 beschriebenen Methode *askForAvailableResources* gefragt werden.

3.6 Die Utility-Klasse CommunicativeExecutors

Um die Erzeugung einer ThreadPoolExecutor-Instanz zu erleichtern, werden in der Utility-Klasse Executors (java.util.concurrent) verschiedene Fabrikmethoden angeboten. Sie erzeugen Instanzen, die für die meisten Anwendungsfälle geeignet sind. Da die Klasse CommunicativeThreadPoolExecutor ebenfalls über viele Parameter verfügt, bietet es sich auch hier an, eine Utility-Klasse zu implementieren, die Fabrikmethoden zur einfacheren Verwendung bereitstellt. Zu diesem Zweck wird die Klasse CommunicativeExecutors implementiert. Neben den Fabrikmethoden stellt sie außerdem eine Methode zur Verfügung, die eine ThreadPoolCommunicationCenter-Instanz mit Standardeinstellungen zurückgibt. Die Anzahl an verfügbaren Ressourcen ist dabei identisch mit der Anzahl an Prozessorkernen des Systems. Jede Instanz der Klasse CommunicativeThreadPoolExecutor, die ohne Angabe einer Instanz der Klasse ThreadPoolCommunicationCenter erzeugt wird, wird von dieser Instanz verwaltet. Daher muss in den meisten Anwendungsfällen, in denen anstelle einer ThreadPoolExecutor-Instanz nun eine Instanz der Klasse CommunicativeThreadPoolExecutor verwendet werden soll, nur eine Zeile des Programmcodes geändert werden.

3.7 Synchronisation

Der Aufruf der Methoden *askForAvailableResources* und *offerResources* der Klasse ThreadPoolCommunicationCenter kann bereits im Fall der Verwendung eines einzigen Thread Pools durch mehrere Threads gleichzeitig erfolgen. Aus diesem Grund ist hinreichende Synchronisation zwingend erforderlich, um die korrekte Bearbeitung dieser Anfragen zu gewährleisten. In Java können zu diesem Zweck explizite Objekte der Klasse ReentrantLock (java.util.concurrent) oder intrinsisches Locking mit Hilfe des Schlüsselwortes *synchronized* verwendet werden. Nach [**?**] resultiert die Verwendung eines ReentrantLock-Objekts in einer signifikanten Leistungsverbesserung gegenüber der Verwendung von *synchronized*, wenn Java in der Version 5 eingesetzt wird. Bei dem Einsatz höherer Java Versionen seien diese Vorteile jedoch deutlich geringer. Da in dieser Arbeit die Leistung von Thread-Pool-Implementierungen verglichen werden soll, sollte zur Implementierung der bestmögliche Synchronisationsmechanismus verwendet werden.

Es wurden daher im Laufe der Arbeit verschiedene Implementierungen der Synchronisation getestet. Neben der Verwendung von ReentrantLock und intrinsischem Locking wurde außerdem der Einsatz von nicht blockierenden Synchronisationskonstrukten mit Hilfe von atomaren Variablen betrachtet. Nach [**?**] können solche Konstrukte, richtig eingesetzt, in Leistungsverbesserungen resultieren, wenn eine hohe Anzahl an Threads um einen Lock konkurriert. In den getesteten Anwendungsfällen war jedoch kein Vorteil bei der Verwendung von ReentrantLock oder nicht

blockierenden Synchronisationskonstrukten gegenüber der Verwendung von intrinsischem Locking zu erkennen. Da außerdem in [**?**] empfohlen wird, in den Fällen, in denen die erweiterte Funktionalität eines ReentrantLock-Objekts nicht erforderlich ist, intrinsisches Locking zu verwenden, wird in dieser Arbeit ausschließlich dieser Synchronisationsmechanismus verwendet.

4. Evaluierung

In diesem Kapitel werden die Ergebnisse der Evaluierung der kommunizierenden Thread Pools vorgestellt und diskutiert. In 4.1 wird dazu beschrieben, wie die Testfälle zum Vergleich der Leistung zwischen Instanzen der Klasse ThreadPoolExecutor und Instanzen der Klasse CommunicativeThreadPoolExecutor aufgebaut sind. Im Anschluss daran wird in 4.2 die Vorgehensweise bei der Erfassung der Messwerte näher erläutert und in 4.3 dargelegt, welche Testumgebungen eingesetzt werden. Abschnitt 4.4 geht dann ausführlich auf die Ergebnisse der Evaluierung ein.

4.1 Aufbau der Testfälle

Zur Leistungsevaluierung sollen die kommunizierenden Thread Pools in typischen Anwendungsfällen mit Instanzen der Klasse ThreadPoolExecutor verglichen werden. Hauptaufgabengebiet für Thread Pools ist die parallele Ausführung einer großen Anzahl homogener und unabhängiger Tasks. In den in dieser Arbeit untersuchten Testfällen werden zwei Typen von Tasks unterschieden. Zur Simulation von CPU-intensiven Tasks werden sogenannte π-Tasks verwendet, in denen die Zahl π iterativ approximiert wird. Die Anzahl der Iterationen ist dabei als Parameter einstellbar. Außerdem werden Tasks verwendet, in denen zwei zufällig konstruierte Matrizen miteinander parallel multipliziert werden. Dabei wird die Multiplikation in mehrere unabhängige Tasks aufgeteilt, die dann von den Threads einer Thread-Pool-Instanz gleichzeitig ausgeführt werden können. Bei diesen Tasks kann der Cache-Speicher sinnvoll genutzt werden, wodurch eventuelle Auswirkungen von Cache-Trashing auf die Laufzeit untersucht werden können. Zur Vereinfachung werden nur quadratische Matrizen verwendet. Die Größe der Matrizen, die Anzahl an Tasks und die Anzahl der Thread-Pool-Instanzen, die diese Tasks gleichzeitig ausführen sollen, sind als Parameter variierbar. Die Anzahl der Thread Pools darf dabei die Anzahl der maximal parallel ausführbaren Threads nicht überschreiten.

Zum Leistungsvergleich werden Instanzen der Klasse ThreadPoolExecutor verwendet. Diese entsprechen in ihren Einstellungen denen, die von der Fabrikmethode *Executors.newFixedThreadPool(int)* erzeugt und zurückgegeben werden. Die maximale Anzahl der Threads wird bei den CommunicativeThreadPoolExecutor- und

den ThreadPoolExecutor-Instanzen auf die Anzahl der maximal parallel ausführbaren Threads gesetzt, die durch die Testumgebung (siehe 4.3) gegeben ist.

4.2 Evaluierungsmethode

Eine korrekte Leistungsmessung in Java wird durch dynamische Kompilierung und den Garbage Collection Mechanismus erschwert. In [**?**] präsentieren die Autoren eine Methode mit der Java-Programme aussagekräftig evaluiert werden können. Es wird dabei zunächst zwischen zwei Formen der Leistungsmessung unterschieden:

steady-state Die Leistung soll im stabilen Zustand gemessen werden. Zur Anwendung dieser Form muss der Code bereits vor der eigentlichen Zeitmessung mehrmals ausgeführt werden, damit häufig aufgerufene Methoden kompiliert sind. Die dynamische Kompilierung soll dabei die Messergebnisse möglichst wenig beeinflussen.

start-up Die Leistung soll bei der ersten Ausführung gemessen werden. Bei dieser Form ist es erwünscht, dass die Messergebnisse durch dynamische Kompilierung beeinflusst werden, da diese beim Start eines Programmes erheblich zur Laufzeit beitragen kann.

In dieser Arbeit wird die *steady-state*-Leistung der entwickelten kommunizierenden Thread Pools untersucht. Zur Umsetzung dieser Form der Leistungsmessung sind nach [**?**] die folgenden Schritte anzuwenden:

1. Innerhalb eines Aufrufs der Java Virtual Machine (JVM) wird zunächst der untersuchende Anwendungsfall in einer Schleife so oft ausgeführt bis ein stabiler Zustand, erkennbar z.B. an der Abweichung der letzten k Messwerte, erreicht ist.

2. Ist der stabile Zustand erreicht, wird von den letzten k Messwerten der Mittelwert gebildet. Er stellt das Ergebnis dieses JVM-Aufrufs dar.

3. Es werden p Aufrufe der JVM durchgeführt und für dessen Ergebnisse Konfidenzintervalle berechnet. Überschneiden sich die Konfidenzintervalle der verschiedenen Thread-Pool-Implementierungen für den untersuchten Testfall, werden die Laufzeiten als identisch betrachtet.

Zur Berechnung der Konfidenzintervalle mittels Normalverteilung empfehlen die Autoren, dass die Bedingung $p >= 30$ erfüllt sein sollte. In dieser Arbeit werden Konfidenzintervalle mit 95 Prozent Wahrscheinlichkeit berechnet und dazu je Testfall 30 JVM-Aufrufe betrachtet. Das Erreichen eines stabilen Zustands innerhalb eines JVM-Aufrufs soll, wie von den Autoren in [**?**] vorgeschlagen, durch die Betrachtung des Variationskoeffizienten (CoV) der letzten k Messwerte ermittelt werden. Abhängig von bestimmten Parametern (z.B. Anzahl Iterationen bei der Berechnung von π) sind stärkere Schwankungen in den Ausführungszeiten innerhalb eines JVM-Aufrufs zu erwarten. Deshalb werden der Parameter k und der Wert des CoV, der für einen stabilen Zustand mindestens erreicht werden muss, für jeden Testfall individuell festgelegt. Außerdem soll nach [**?**] ein maximaler Wert für die Anzahl an

Iterationen innerhalb eines JVM-Aufrufs festgelegt werden. Dieser Wert wird in dieser Arbeit ebenfalls individuell für jeden Testfall festgelegt. Wird er bei der Suche nach k stabilen Messwerten überschritten, wird dieser Aufruf bei der Bestimmung der Konfidenzintervalle nicht berücksichtigt.

4.3 Testumgebungen

Es wurden zwei unterschiedliche Testumgebungen zur Evaluierung verwendet: ein selbst zusammengestellter Desktop-PC und ein ASUS G73JH Notebook. Der Desktop-PC ist dabei mit einem Intel Core i5-4670 Prozessor der Haswell-Architektur ausgestattet. Dieser verfügt über vier Prozessorkerne mit je einer Taktfrequenz von 3,40 GHz. Es wird keine Hyperthreading-Technologie unterstützt. Das bedeutet, dass maximal vier Threads parallel ausgeführt werden können. Weiter verfügt das Testgerät über 8 GB Arbeitsspeicher. Als Betriebssystem wird die 64-Bit Version von Windows 8.1 Pro verwendet. Das ASUS G73JH ist mit einem Intel Core i7-740QM Prozessor der Clarksfield-Architektur ausgestattet, dessen vier Prozessorkerne mit je 1,73 GHz getaktet sind. Dieser Prozessor unterstützt die Hyperthreading-Technologie, sodass bis zu acht Threads gleichzeitig ausgeführt werden können. Auch dieses Testgerät verfügt über 8 GB Arbeitsspeicher. Als Betriebssystem wird Windows 7 Home Premium in einer 64-Bit Version verwendet. Weiter kommen auf beiden Testgeräten ein Java Compiler in der Version 1.7 und die Hotspot JVM von Sun zum Einsatz.

4.4 Ergebnisse

In diesem Abschnitt sollen die Ergebnisse von verschiedenen Testfällen präsentiert werden. Jeder Abschnitt gibt eine kurze Einführung über den betrachteten Testfall. Als Erstes ist die Anwendung einer einzelnen Thread-Pool-Instanz vorgesehen, um zunächst die Leistung eines kommunizierenden Thread Pools ohne Kommunikation mit weiteren Instanzen zu untersuchen.

4.4.1 Testfall 1

In diesem Anwendungsfall soll die Laufzeit einer einzelnen Thread-Pool-Instanz untersucht werden. Der Instanz werden 100 π-Tasks mit je 1000 Iterationen zugewiesen und dabei gemessen, wie viel Zeit zur Ausführung dieser Tasks benötigt wird. Die Zeitmessung beginnt bei Übergabe des ersten Tasks und endet nach Ausführung des letzten Tasks. Abbildung 4.1 stellt die Ergebnisse dieses Testfalls grafisch für die zwei Testumgebungen dar.

Auf dem Desktop-PC können die Tasks durch den kommunizierenden Thread Pool im Durchschnitt schneller abgearbeitet werden, wohingegen auf dem ASUS Notebook die Verwendung einer ThreadPoolExecutor-Instanz deutliche Vorteile zu haben scheint. Ein möglicher Erklärungsansatz für diese Beobachtungen ist, dass für jeden Thread, der zur Bearbeitung der Tasks erzeugt werden muss, die Instanz der Klasse CommunicativeThreadPoolExecutor zuerst Rücksprache mit ihrer ThreadPoolCommunicationCenter-Instanz halten muss. Dagegen kann eine Instanz der Klasse ThreadPoolExecutor sofort einen weiteren Thread erzeugen. Je höher die Anzahl der verwendbaren Threads, desto deutlicher ist dieser Vorteil ersichtlich. In den folgenden Abschnitten soll nun untersucht werden, wie sich die Verwendung mehrerer Instanzen, die miteinander kommunizieren können, auf die Laufzeit auswirkt.

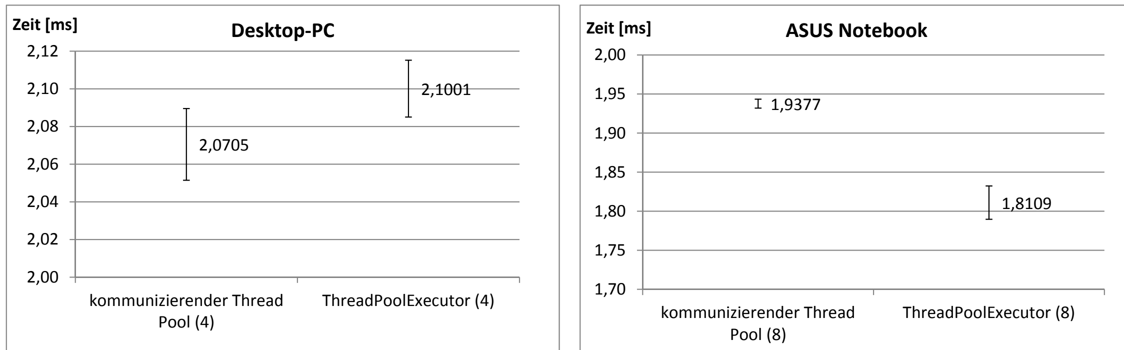


Abbildung 4.1: Mittelwerte und Konfidenzintervalle bei der Ausführung von Testfall 1 auf dem Desktop-PC (links) und dem ASUS Notebook (rechts). Die Konfidenzintervalle werden grafisch durch Fehlerindikatoren dargestellt. Der Mittelwert ist daneben angegeben. Die Zahl in den Klammern beschreibt die maximale Anzahl an Threads, die eine Thread-Pool-Instanz verwenden konnte.

4.4.2 Testfall 2

In Abschnitt 4.4.1 wurde die Leistung einer einzelnen Thread-Pool-Instanz betrachtet. Hier soll nun die Verwendung mehrerer gleichzeitig arbeitender Instanzen untersucht werden. Wie in Testfall 1 erhält jede Instanz 100 π-Tasks mit je 1000 Iterationen zur Ausführung zugewiesen. Die Zeitmessung beginnt bei Übergabe des ersten Tasks und endet, wenn alle Instanzen ihre 100 Tasks abgearbeitet haben. In Abbildung 4.2 werden zunächst die Ergebnisse der Verwendung von zwei Instanzen dargestellt.

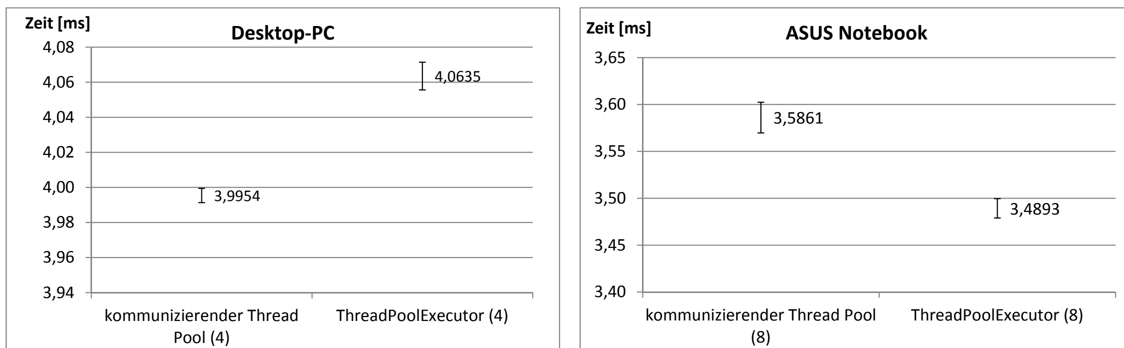


Abbildung 4.2: Mittelwerte und Konfidenzintervalle bei der Ausführung von Testfall 2 mit **zwei Instanzen** auf dem Desktop-PC (links) und dem ASUS Notebook (rechts). Die Konfidenzintervalle werden grafisch durch Fehlerindikatoren dargestellt. Der Mittelwert ist daneben angegeben. Die Zahl in den Klammern beschreibt die maximale Anzahl an Threads, die eine Thread-Pool-Instanz verwenden konnte.

Hier zeichnet sich ein ähnliches Bild wie bei der Verwendung einer einzelnen Instanz ab. Auf dem Desktop-PC zeigen die kommunizierenden Thread Pools eine bessere Leistung, auf dem ASUS Notebook die ThreadPoolExecutor-Instanzen. Die Threads der zwei Thread-Pool-Instanzen konkurrieren offenbar noch nicht stark genug um die Prozessorkerne des Systems, um einen Vorteil aus der Kommunikation miteinander zu gewinnen. Deswegen soll nun der gleiche Anwendungsfall bei der Verwendung von

vier Thread-Pool-Instanzen untersucht werden. Die Ergebnisse werden in Abbildung 4.3 präsentiert.

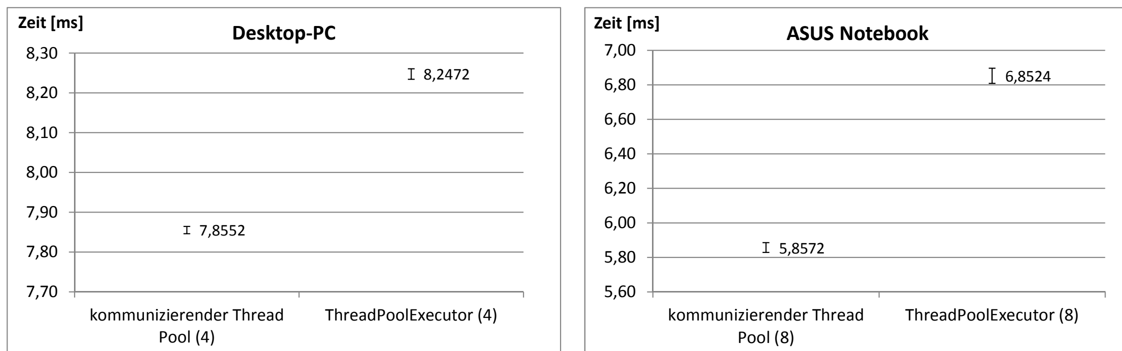


Abbildung 4.3: Mittelwerte und Konfidenzintervalle bei der Ausführung von Testfall 2 mit **vier Instanzen** auf dem Desktop-PC (links) und dem ASUS Notebook (rechts). Die Konfidenzintervalle werden grafisch durch Fehlerindikatoren dargestellt. Der Mittelwert ist daneben angegeben. Die Zahl in den Klammern beschreibt die maximale Anzahl an Threads, die eine Thread-Pool-Instanz verwenden konnte.

Hier benötigen die kommunizierenden Thread Pools auf beiden Testgeräten eine deutlich geringere Laufzeit als die Instanzen der Klasse ThreadPoolExecutor. Im Gegensatz zu den vorherigen Testfällen scheint sich die Kommunikation der Thread-Pool-Instanzen hier auszuzahlen. Auf dem Desktop-PC konnte im Durchschnitt eine Laufzeitverbesserung von $4,75$ Prozent, auf dem ASUS Notebook eine von $14,5$ Prozent, durch Verwendung von kommunizierenden Thread Pools erreicht werden.

Auf dem ASUS Notebook kann Testfall 2 mit acht Instanzen untersucht werden. Da der Prozessor des Desktop-PCs maximal vier Threads parallel ausführen kann, ist der Einsatz von acht Instanzen dort nicht sinnvoll. Abbildung 4.4 zeigt die Ergebnisse dieses Testfalls.

Im Vergleich zu der Verwendung von vier Instanzen ist in diesem Fall der Vorteil der kommunizierenden Thread Pools noch deutlicher zu erkennen. Im Durchschnitt konnte eine 26-prozentige Laufzeitverbesserung erreicht werden. Wahrscheinlichste Ursache für dieses Ergebnis ist die Reduzierung von Kontextwechseln. Bei der Verwendung von acht ThreadPoolExecutor-Instanzen konkurrieren 64 Threads um die verfügbaren Ressourcen, wohingegen die kommunizierenden Thread Pools die Ressourcen untereinander aufteilen. Dies führt zu deutlich weniger Kontextwechseln und damit zu weniger Mehraufwand für die Verwaltung der Threads durch das Betriebssystem.

4.4.2.1 Bemerkungen

Es wurden weitere Testfälle nach dem oben beschriebenen Schema mit höherer Anzahl an Tasks oder höherer Anzahl an Iterationen durchgeführt. Die Ergebnisse dieser Testfälle sind tabellarisch in Anhang A.1 aufgeführt. In einigen Fällen konnte ebenfalls eine Leistungsverbesserung bei der Verwendung kommunizierender Thread Pools gemessen werden. Diese sind jedoch nicht derart hoch wie in den zuvor beschriebenen Fällen. Beispielsweise ließ sich bei Erhöhung der Iterationen von

1000 auf 10000 auf dem ASUS Notebook unter Verwendung von acht Thread-Pool-Instanzen im Durchschnitt eine 2,4-prozentige Verbesserung erreichen. Die Erhöhung der Anzahl der Tasks auf 1000 bei 1000 Iterationen auf dem gleichen System resultierte in einer Laufzeitverbesserung von 2,5 Prozent. Auf dem Desktop-PC mit vier Thread-Pool-Instanzen ergaben sich in diesen Fällen Verbesserungen von 0,2 und 1,9 Prozent.

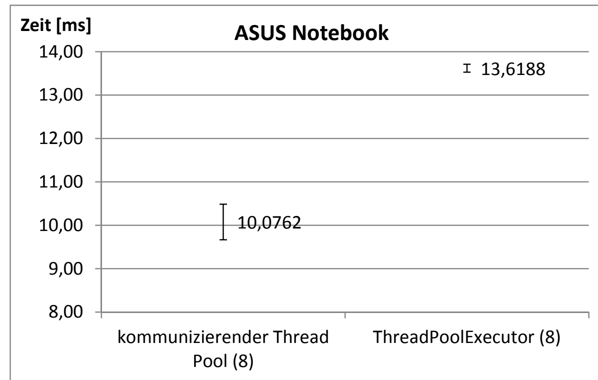


Abbildung 4.4: Mittelwerte und Konfidenzintervalle bei der Ausführung von Testfall 2 mit **acht Instanzen** auf dem ASUS Notebook. Die Konfidenzintervalle werden grafisch durch Fehlerindikatoren dargestellt. Der Mittelwert ist daneben angegeben. Die Zahl in den Klammern beschreibt die maximale Anzahl an Threads, die eine Thread-Pool-Instanz verwenden konnte.

4.4.3 Testfall 3

Im dritten Testfall soll nun untersucht werden, wie sich die Kommunikation der Thread-Pool-Instanzen auf die Laufzeit eines einzigen Thread Pools auswirkt, wenn gleichzeitig weitere Instanzen mit der Ausführung anderer Tasks beschäftigt sind. Hierzu soll die Thread-Pool-Instanz, dessen Laufzeit gemessen wird, die parallele Multiplikation zweier Matrizen durchführen. Gleichzeitig sollen weitere Instanzen π-Tasks mit 1000 Iterationen berechnen. Die Anzahl an π-Tasks ist für jede dieser Instanzen gleich. Die Zeitmessung beginnt mit der Übergabe des ersten Tasks und endet, wenn die Tasks der parallelen Matrixmultiplikation ausgeführt sind. Weiter soll eine CommunicativeThreadPoolExecutor-Instanz in diesem Testfall einen Task pro Matrix-Zeile erhalten, eine ThreadPoolExecutor-Instanz hingegen genauso viele Tasks wie gleichzeitig ausgeführt werden können.

Abbildung 4.5 zeigt die Messergebnisse dieses Testfalls zunächst für zwei Thread-Pool-Instanzen auf den beiden Testsystemen. Auf dem Desktop-PC wird die Matrixgröße auf 500 gesetzt. Das bedeutet, dass der kommunizierende Thread Pool 500 Tasks (1 Task pro Zeile) ausführen muss, die ThreadPoolExecutor-Instanz hingegen 4 Tasks. Die Anzahl an π-Tasks, die der zweite Thread Pool gleichzeitig ausführen soll, wird dabei variiert. Auf dem ASUS Notebook wird die Größe der Matrizen zur Vereinfachung auf 504 erhöht, sodass für die ThreadPoolExecutor-Instanz acht homogene Tasks erzeugt werden können.

In den Diagrammen lässt sich erkennen, dass der Vorteil der Verwendung von kommunizierenden Thread Pools in diesem Testfall maßgeblich von der Beschäftigung der zweiten Instanz abhängt. Auf beiden Testsystemen ist der Laufzeitunterschied

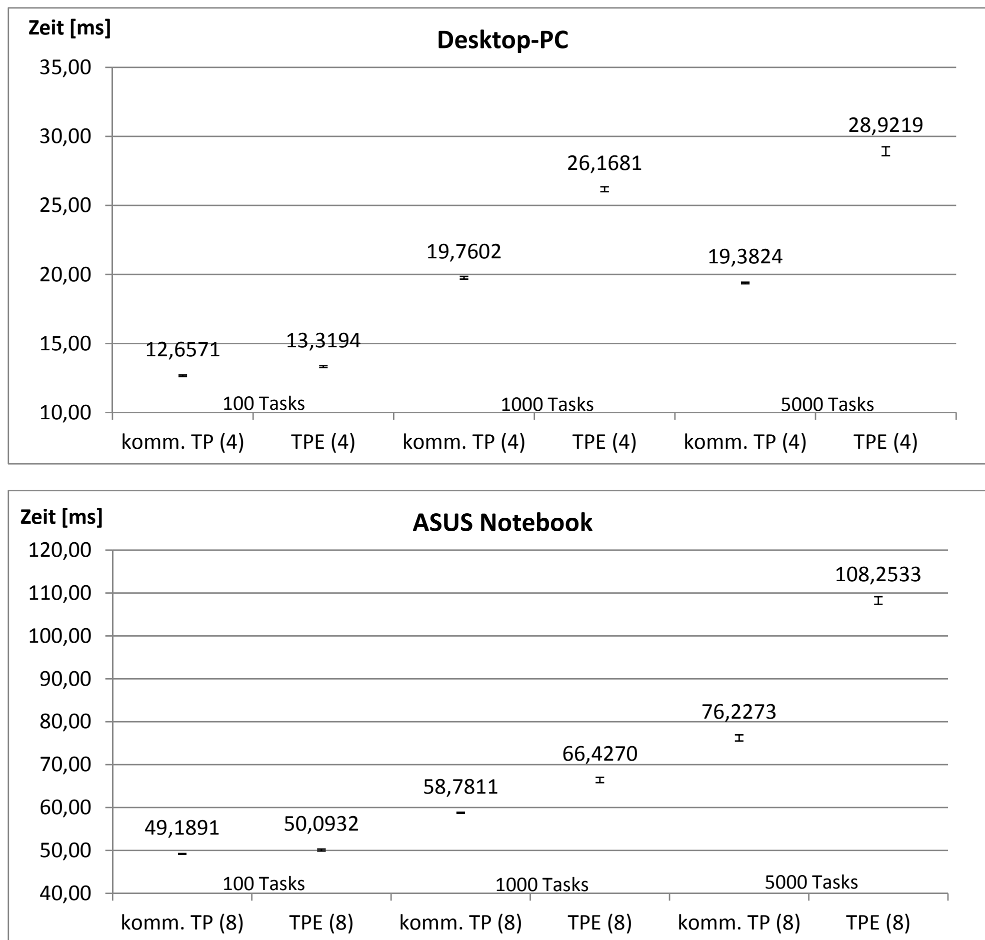


Abbildung 4.5: Mittelwerte und Konfidenzintervalle bei der Ausführung von Testfall 3 mit **zwei Instanzen** auf dem Desktop-PC (oben) und dem ASUS Notebook (unten). Die Konfidenzintervalle werden grafisch durch Fehlerindikatoren dargestellt. Der Mittelwert ist darüber angegeben. Die Anzahl an π-Tasks der zweiten Thread-Pool-Instanz ist durch die Beschriftung an der x-Achse gegeben. Die Zahl in den Klammern beschreibt die maximale Anzahl an Threads, die eine Thread-Pool-Instanz verwenden konnte.

gering, wenn der zweite Thread Pool 100 π-Tasks ausführt. Steigt deren Anzahl jedoch auf 1000 oder auf 5000, lässt sich ein deutlicher Unterschied feststellen. Interessanterweise ist die Laufzeit des kommunizierenden Thread Pools auf dem Desktop-PC ungefähr gleich, unabhängig davon, ob die zweite Instanz 1000 oder 5000 Tasks ausführt. Dies deutet darauf hin, dass der Scheduler des Systems - wie erhofft - die Prozessorkerne auf die zwei Instanzen aufteilt. Auf dem ASUS Notebook ist dies mit den verwendeten Parameterwerten nicht erkennbar.

Als Nächstes soll die Leistung bei der Verwendung von vier Thread-Pool-Instanzen untersucht werden. Der erste Thread Pool führt dabei erneut eine parallele Matrixmultiplikation auf zwei Matrizen der Größe 500 bzw. 504 durch. Die drei weiteren Thread-Pool-Instanzen sollen je 1000 π-Tasks mit 1000 Iterationen gleichzeitig ausführen. Abbildung 4.6 präsentiert die Messergebnisse dieses Testfalls.

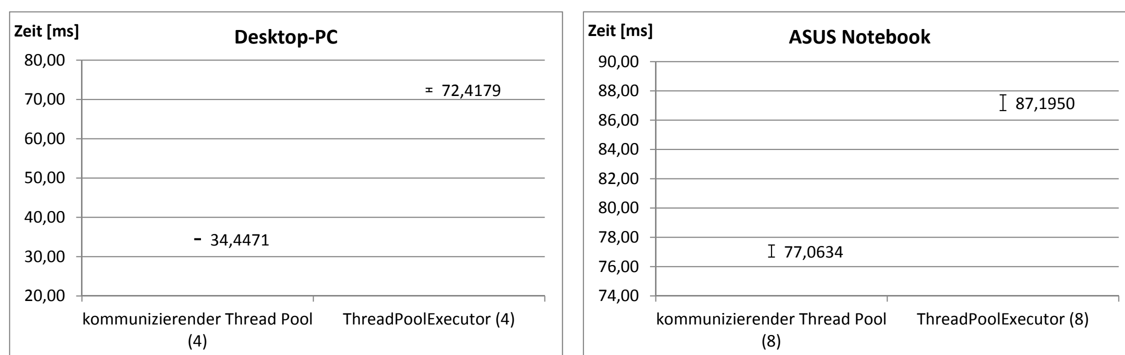


Abbildung 4.6: Mittelwerte und Konfidenzintervalle bei der Ausführung von Testfall 3 mit **vier Instanzen** auf dem Desktop-PC (links) und dem ASUS Notebook (rechts). Die Konfidenzintervalle werden grafisch durch Fehlerindikatoren dargestellt. Der Mittelwert ist daneben angegeben. Die Zahl in den Klammern beschreibt die maximale Anzahl an Threads, die eine Thread-Pool-Instanz verwenden konnte.

Bei der Ausführung auf dem Desktop-PC ist hier im Durchschnitt eine besonders deutliche Laufzeitverbesserung von $52,4$ Prozent zu erkennen. Auf dem ASUS Notebook fällt sie mit $11,6$ Prozent geringer aus. Bei Vergleich dieser Werte mit denen, die bei Anwendung von zwei Thread-Pool-Instanzen gemessen wurden, fällt auf, dass die Laufzeitverbesserung auf dem Desktop-PC um 100 Prozent gestiegen ist. Auf dem ASUS Notebook ist sie hingegen fast identisch geblieben. Zum Abschluss dieses Abschnitts soll dieser Testfall mit acht Thread-Pool-Instanzen auf dem ASUS Notebook durchgeführt werden. Abbildung 4.7 stellt die Ergebnisse grafisch dar. Im Vergleich zur Anwendung von vier Instanzen ist die gemessene Laufzeitverbesserung hier mit $4,6$ Prozent deutlich geringer.

4.4.3.1 Bemerkungen

Der Unterschied der Laufzeitverbesserungen bei der Ausführung auf dem Desktop-PC und auf dem ASUS Notebook wird erst nachvollziehbar, wenn nach Abschluss der Zeitmessung die Anzahl der fertiggestellten Tasks der weiteren Thread-Pool-Instanzen betrachtet wird. Auf beiden Testgeräten werden bei Verwendung von ThreadPoolExecutor-Instanzen alle π-Tasks innerhalb der gemessenen Zeit beendet. Werden kommunizierende Thread Pools verwendet, ist auf dem ASUS Notebook in dieser Hinsicht keine nennenswerte Veränderung festzustellen, auf dem Desktop-PC

hingegen beendet der erste Thread Pool seine Tasks deutlich früher als die anderen. Der Scheduler des Betriebssystems scheint in diesem Fall die verfügbaren Prozessorkerne an die kommunizierenden Thread Pools wie gewünscht aufzuteilen, da der erste Thread Pool offenbar nicht durch die anderen bei der Ausführung seiner Tasks behindert wird.

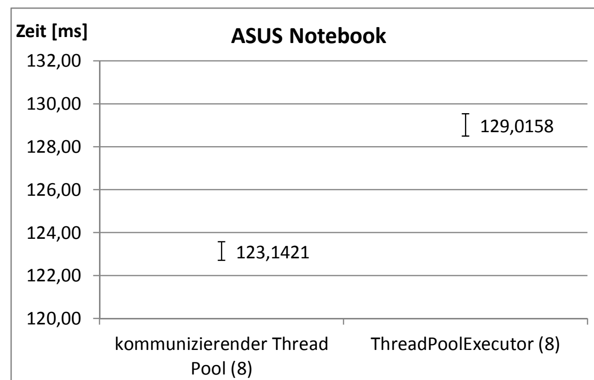


Abbildung 4.7: Mittelwerte und Konfidenzintervalle bei der Ausführung von Testfall 3 mit **acht Instanzen** auf dem ASUS Notebook. Die Konfidenzintervalle werden grafisch durch Fehlerindikatoren dargestellt. Der Mittelwert ist daneben angegeben. Die Zahl in den Klammern beschreibt die maximale Anzahl an Threads, die eine Thread-Pool-Instanz verwenden konnte.

4.4.4 Testfall 4

In Testfall 3 (Abschnitt 4.4.3) wird für eine Instanz eines kommunizierenden Thread Pools ein Task pro Zeile bei der parallelen Matrixmultiplikation erstellt. Dagegen werden für eine ThreadPoolExecutor-Instanz genauso viele Tasks erzeugt, wie maximal gleichzeitig ausführbar sind. Im vierten Testfall sollen beide Typen nun einen Task pro Matrixzeile ausführen und dabei erneut nur die Laufzeit der Instanz gemessen werden, die diese Tasks ausführt. Weitere Thread-Pool-Instanzen werden wie in Testfall 3 mit π-Tasks mit je 1000 Iterationen beschäftigt, um Konkurrenz bei der Verwendung der vorhandenen Prozessorkerne des Systems zu schaffen. Die Matrixmultiplikation soll auf Matrizen der Größe 500 bzw. 504 durchgeführt werden. Die Anzahl der π-Tasks wird auf 1000 je weiterer Thread-Pool-Instanz gesetzt. Diese Werte sind weitestgehend identisch mit denen in Testfall 3 und ermöglichen daher einen Vergleich der beiden Testfälle. Begonnen werden soll hier zunächst mit zwei Thread-Pool-Instanzen. Abbildung 4.8 zeigt die Messergebnisse für diesen Fall auf beiden Testgeräten.

Bei Vergleich dieser Werte mit denen aus Testfall 3 (Abbildung 4.5) fällt auf, dass die ThreadPoolExecutor-Instanzen hier auf dem ASUS Notebook eine bessere Laufzeit abliefern. Die kommunizierenden Thread Pools besitzen nur noch sehr leichte Vorteile auf diesem Testgerät. Auf dem Desktop-PC haben sich die Messwerte hingegen durch die Verwendung kleinerer Tasks bei der parallelen Matrixmultiplikation nicht maßgeblich verändert. Die Laufzeitverbesserung durch Verwendung von kommunizierenden Thread Pools beträgt dort im Durchschnitt $28,8$ Prozent, auf dem ASUS Notebook dagegen nur $1,1$ Prozent.

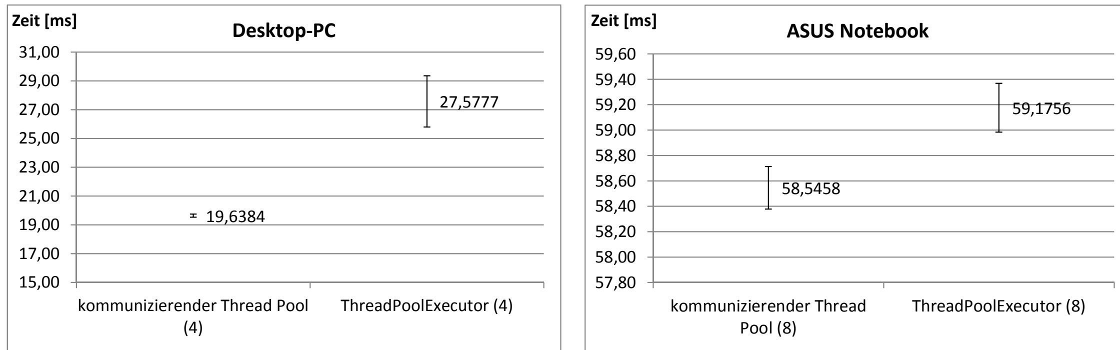


Abbildung 4.8: Mittelwerte und Konfidenzintervalle bei der Ausführung von Testfall 4 mit **zwei Instanzen** auf dem Desktop-PC (links) und dem ASUS Notebook (rechts). Die Konfidenzintervalle werden grafisch durch Fehlerindikatoren dargestellt. Der Mittelwert ist daneben angegeben. Die Zahl in den Klammern beschreibt die maximale Anzahl an Threads, die eine Thread-Pool-Instanz verwenden konnte.

Werden vier Thread-Pool-Instanzen verwendet, ergeben sich die in Abbildung 4.9 grafisch dargestellten Ergebnisse. Die durchschnittliche Laufzeitverbesserung auf dem Desktop-PC hat sich hier fast verdoppelt. Sie beträgt nun 51, 8 Prozent und ist damit etwas geringer als die 52, 4 Prozent, die in Testfall 3 gemessen wurden. Die Verbesserung auf dem ASUS Notebook ist mit 4, 6 Prozent deutlich geringer als die in Testfall 3 erreichten 11, 6 Prozent.

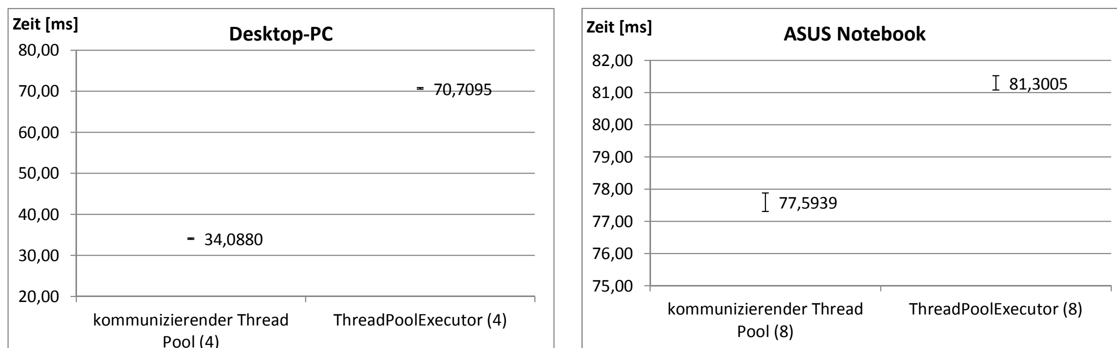


Abbildung 4.9: Mittelwerte und Konfidenzintervalle bei der Ausführung von Testfall 4 mit **vier Instanzen** auf dem Desktop-PC (links) und dem ASUS Notebook (rechts). Die Konfidenzintervalle werden grafisch durch Fehlerindikatoren dargestellt. Der Mittelwert ist daneben angegeben. Die Zahl in den Klammern beschreibt die maximale Anzahl an Threads, die eine Thread-Pool-Instanz verwenden konnte.

Zum Abschluss dieses Abschnitts soll Testfall 4 mit acht Instanzen auf dem ASUS Notebook durchgeführt werden. Abbildung 4.10 zeigt die Ergebnisse dieser Messung. In diesem Fall konnte im Durchschnitt eine geringe Laufzeitverbesserung von 1, 2 Prozent durch die Verwendung kommunizierender Thread Pools erreicht werden. Dieser Wert ist ebenfalls deutlich geringer als der der in Testfall 3 gemessenen 4, 6 Prozent. Testfall 4 verdeutlicht damit, dass es zumindest auf dem ASUS Notebook stark vom Anwendungsfall abhängt, ob eine Leistungsverbesserung bei der Verwendung von kommunizierenden Thread Pools anstelle von ThreadPoolExecutor-Instanzen erreicht werden kann.

4.5 Abschließende Bemerkungen

Die in diesem Abschnitt durchgeführten Testfälle decken selbstverständlich nicht jeden Anwendungsfall ab, in dem sich die Verwendung von Thread Pools anbietet. Sie simulieren jedoch eines der Haupteinsatzgebiete von Thread Pools: die Ausführung von kurzen, homogenen und unabhängigen Tasks. Die teilweise sehr hohe Laufzeitverbesserung auf dem Desktop-PC zeigt daher, dass es sich durchaus lohnen kann für eine gegebene Situation den Einsatz kommunizierender Thread Pools in Betracht zu ziehen. Ob tatsächlich eine Leistungssteigerung erreicht werden kann oder wie hoch diese ausfällt, scheint unter anderem stark von dem System abzuhängen, auf dem das Programm ausgeführt werden soll.

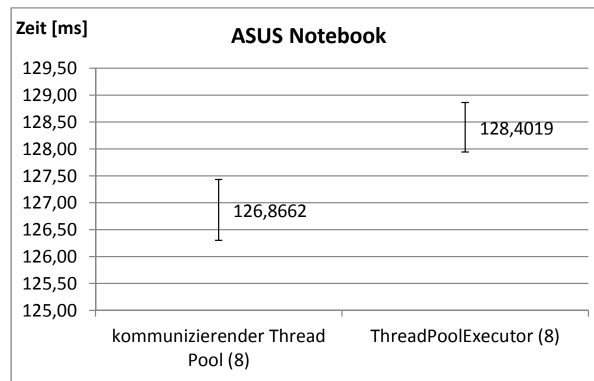


Abbildung 4.10: Mittelwerte und Konfidenzintervalle bei der Ausführung von Testfall 4 mit **acht Instanzen** auf dem ASUS Notebook. Die Konfidenzintervalle werden grafisch durch Fehlerindikatoren dargestellt. Der Mittelwert ist daneben angegeben. Die Zahl in den Klammern beschreibt die maximale Anzahl an Threads, die eine Thread-Pool-Instanz verwenden konnte.

Im Anhang sind die Ergebnisse aller durchgeführten Tests tabellarisch aufgeführt. Neben den in diesem Kapitel vorgestellten finden sich dort ebenfalls die Ergebnisse weiterer Tests mit variierenden Parametern.

5. Zusammenfassung und Ausblick

In dieser Arbeit wurde ein auf Kommunikation basierender, dynamischer Ansatz zur Laufzeitoptimierung bei der gleichzeitigen Verwendung mehrerer Thread-Pool-Instanzen untersucht. Dazu wurde eine Implementierung von kommunizierenden Thread Pools entwickelt und in typischen Testfällen mit einer bereits vorhandenen Thread-Pool-Implementierung getestet.

Die einzelnen Instanzen wurden dabei in die Lage versetzt, die zur Verfügung stehenden Systemressourcen untereinander aufzuteilen. Diese Aufteilung wurde mit Hilfe einer zentralen Klasse organisiert, bei der die Thread Pools nach verfügbaren Ressourcen fragen und verwendete abgeben konnten. Die Zuweisung einer Ressource zu einer Instanz erlaubte es dieser, einen Thread auszuführen. Der erhoffte Effekt dieser Vorgehensweise war die Reduzierung der Anzahl an Threads, die zu einem Zeitpunkt um eine Systemressource (z.B. einen Prozessorkern) konkurrieren und damit die Vermeidung von Kontextwechseln und Cache-Trashing.

Die Implementierung der kommunizierenden Thread Pools wurde in der Programmiersprache Java durchgeführt. Zur Evaluierung wurde daher die Laufzeit im Vergleich mit Instanzen der Klasse ThreadPoolExecutor der Java API auf zwei verschiedenen Testgeräten untersucht. Auf beiden Testgeräten waren in einigen betrachteten Testfällen deutliche Vorteile bei der Verwendung kommunizierender Thread Pools zu erkennen. Die Höhe der Laufzeitverbesserung variierte jedoch stark abhängig vom eingesetzten Testgerät. Verschiedene Faktoren, darunter das eingesetzte Betriebssystem (Scheduling-Methode) und der eingesetzte Prozessor (Anzahl der Kerne, Unterstützung der Hyperthreading-Technologie), könnten dafür verantwortlich sein.

Die Threads der Thread-Pool-Instanzen wurden in dieser Arbeit nicht explizit auf einen Prozessorkern abgebildet, sondern es wurde lediglich die Anzahl an Threads festgelegt, die eine Instanz zu einem bestimmten Zeitpunkt verwenden durfte. Auf welchem Prozessorkern ein bestimmter Thread ausgeführt wurde, konnte weiterhin von dem Scheduler des Betriebssystems bestimmt werden. In einer zukünftigen Arbeit könnte daher der Einfluss des Betriebssystems bei der Verwendung kommunizierender Thread Pools untersucht werden, indem explizites Scheduling der Threads durch Festlegung von Prozessoraffinitäten implementiert wird. Aus Zeitgründen konnte dieser Ansatz in dieser Arbeit nicht umgesetzt werden.

Weiter wurden die zur Verfügung stehenden Systemressourcen in dieser Arbeit möglichst gleichmäßig auf die vorhandenen Thread-Pool-Instanzen verteilt. Für spezielle Anwendungsfälle ist es denkbar dieses Verhalten zu ändern. Es könnte beispielsweise die Priorisierung einzelner Instanzen realisiert werden, sodass Thread Pools mit höherer Priorität bei der Vergabe von Ressourcen bevorzugt werden. Sofern im Voraus erkennbar, könnten ebenfalls die Eigenschaften der zu bearbeitenden Tasks eines Thread Pools berücksichtigt werden, um die Leistung des Systems zu verbessern. Ferner ist momentan jede Thread-Pool-Instanz selbst dafür verantwortlich zu entscheiden, wann Systemressourcen wieder freigegeben werden. Ausschließlich im Falle einer Neuverteilung der Ressourcen auf alle Thread-Pool-Instanzen werden einer Instanz Berechtigungen entzogen. In einer zukünftigen Arbeit könnten daher verschiedene Strategien für das Ressourcenmanagement integriert und deren Auswirkungen auf die Laufzeit untersucht werden.

Eine weitere offene Frage ist, welchen Effekt es auf die Leistung der in dieser Arbeit entwickelten Implementierung hat, wenn die Anzahl an Thread-Pool-Instanzen und damit die Anzahl an Threads stark zunimmt. Auf den eingesetzten Testgeräten war es maximal möglich acht Instanzen mit je acht Threads zu verwenden. Da die Zahl der Prozessorkerne - und damit auch die Anzahl an Threads und Thread Pools, die eingesetzt werden können - in den nächsten Jahren voraussichtlich stark zunehmen wird, ist diese Frage äußerst relevant. Die Kommunikation der Thread-Pool-Instanzen ist hier durch eine zentrale Kommunikationsinstanz realisiert, die sich bei steigender Anzahl an Thread Pools zu einem Flaschenhals entwickeln könnte.

Die in dieser Arbeit entwickelte Implementierung realisiert eine Form der Kommunikation zwischen verschiedenen Thread-Pool-Instanzen innerhalb eines Prozesses. In heutigen Systemen werden für gewöhnlich mehrere Prozesse gleichzeitig ausgeführt. Ein möglicher nächster Schritt wäre daher die Implementierung von Thread Pools, die über die Prozessgrenzen hinaus miteinander kommunizieren können. Da Kontextwechsel auf Prozessebene deutlich aufwendiger als auf Thread-Ebene sind, könnte die Kommunikation von Thread-Pool-Instanzen verschiedener Prozesse erheblichen Einfluss auf die Laufzeit dieser Prozesse besitzen.

Fazit: Die im Rahmen dieser Arbeit ermittelten Messergebnisse zeigen, dass der Einsatz von kommunizierenden anstelle von gewöhnlichen Thread Pools deutliche Vorteile für die Laufzeit eines Programmes haben kann. Die unterschiedliche Höhe der gemessenen Laufzeitverbesserungen auf den beiden Testsystemen deutet jedoch daraufhin, dass die Vorteile stark von dem Anwendungsfall und dem eingesetzten System abhängig sind. In dieser Arbeit konnten auf einem Testgerät Laufzeitverbesserungen von teilweise sogar über 50 Prozent erreicht werden. Es kann sich daher lohnen, die Verwendung kommunizierender Thread Pools bei der Implementierung eines Programms mit mehreren Thread-Pool-Instanzen in Betracht zu ziehen.

Literaturverzeichnis

[] P. Flick, P. Sanders und J. Speck. Malleable Sorting. *IEEE 27th International Symposium on Parallel and Distributed Processing*, Mai 2013, S. 418–426.

[] A. Georges, D. Buytaert und L. Eeckhout. Statistically Rigorous Java Performance Evaluation. *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, 2007, S. 57–76.

[] E. Gamma, R. Helm, R. Johnson und J. Vlissides. *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software.* Addison-Wesley. 2. Auflage, Juli 2004.

[] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes und D. Lea. *Java Concurrency in Practice.* Addison-Wesley. Mai 2006.

[] Oracle Website. Java Application Programming Interface (API). http://docs.oracle.com/javase/7/docs/api/, 2014.

[] N. jiang Chen und P. Lin. A Dynamic Adjustment Mechanism with Heuristic for Thread Pool in Middleware. *Third International Joint Conference on Computational Science and Optimization* Band 1, Mai 2010, S. 369–372.

[] D. Kang, S. Han, S. Yoo und S. Park. Prediction-based Dynamic Thread Pool Scheme for Efficient Resource Usage. *IEEE 8th International Conference on Computer and Information Technology Workshops*, Juli 2008, S. 159–164.

[] K. Lee, H. N. Pham, H. Kim, H. Y. Youn und O. Song. A Novel Predictive and Self-Adaptive Dynamic Thread Pool Management. *IEEE 9th International Symposium on Parallel and Distributed Processing with Applications*, Mai 2011, S. 93–98.

[] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley. 2. Auflage, Oktober 1999.

[] Y. Ling, T. Mullen und X. Lin. Analysis of Optimal Thread Pool Size. *Operating Systems Review* 34(2), Februar 2000, S. 42–55.

[] M. Welsh, S. D. Gribble, E. A. Brewer und D. Culler. A Design Framework for Highly Concurrent Systems. Technischer Bericht, University of California, Berkeley, 2000.

[] D. Xu und B. Bode. Performance Study and Dynamic Optimization Design for Thread Pool Systems. Technischer Bericht, Iowa State University, 2004.

Hiermit erkläre ich, Tobias Weiberg, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

____________________ ____________________________________

Ort, Datum Unterschrift

A. Anhang

A.1 Messergebnisse - Testfall 1 und 2

Die Tabellen A.1 und A.2 enthalten die Messergebnisse der Ausführung von Testfall 1 (4.4.1) und 2 (4.4.2) mit verschiedenen Parametern. Jede Thread-Pool-Instanz führte dabei die in der Spalte *Tasks/ Instanz* angegebene Anzahl an π-Tasks mit der in der Spalte *Iterationen* angegebenen Anzahl an Iterationen durch. Die Zeitmessung begann mit der Übergabe des ersten Tasks und endete, wenn jede Instanz alle ihr zugewiesenen Tasks ausgeführt hatte. Die Spalte *#Instanzen* gibt die Anzahl der gleichzeitig arbeitenden Thread-Pool-Instanzen an. Die Einheit der angegebenen Zeitwerte für die Mittelwerte und Konfidenzintervalle ist Millisekunden (ms). Bei jedem Test wurden die Ausführungszeiten bei Verwendung von Instanzen der Klasse CommunicativeThreadPoolExecutor sowie von Instanzen der Klasse ThreadPoolExecutor gemessen. Die Erzeugung der Thread-Pool-Instanzen erfolgte mit Hilfe der Fabrikmethode *newFixedSizeCommunicativeThreadPool* bzw. *newFixedThreadPool* der Klasse CommunicativeExecutors bzw. Executors. Jeder Thread-Pool-Instanz standen dabei auf dem Desktop-PC maximal vier, auf dem ASUS Notebook maximal acht Threads zur Ausführung von Tasks zur Verfügung.

			kommunizierender Thread Pool		ThreadPoolExecutor	
#Instanzen	Tasks/ Instanz	Iterationen	Mittelwert	Konfidenzintervall(95%)	Mittelwert	Konfidenzintervall(95%)
1	100	1000	2,0705	[2,0515;2,0896]	2,1001	[2,0850;2,1152]
2	100	1000	3,9954	[3,9913;3,9994]	4,0635	[4,0556;4,0714]
4	100	1000	7,8552	[7,8461;7,8643]	8,2472	[8,2340;8,2605]
4	1000	1000	76,6642	[76,6117;76,7168]	78,1263	[77,8862;78,3663]
4	100	10000	76,5138	[76,3863;76,6413]	76,6761	[76,6343;76,7179]
4	1000	10000	751,7810	[751,1975;752,3646]	746,9883	[746,2038;747,7727]

Tabelle A.1: Messwerte bei der Ausführung der Testfälle 1 und 2 mit verschiedenen Parametern auf dem Desktop-PC

			kommunizierender Thread Pool		ThreadPoolExecutor	
#Instanzen	Tasks/ Instanz	Iterationen	Mittelwert	Konfidenzintervall(95%)	Mittelwert	Konfidenzintervall(95%)
1	100	1000	1,9377	[1,9320;1,9433]	1,8109	[1,7896;1,8323]
2	100	1000	3,5861	[3,5697;3,6024]	3,4893	[3,4790;3,4996]
4	100	1000	5,8572	[5,8290;5,8853]	6,8524	[6,8086;6,8962]
4	1000	1000	43,6940	[43,5171;43,8708]	44,5498	[44,2130;44,8866]
4	100	10000	42,0443	[41,9557;42,1328]	43,0599	[42,7362;43,3836]
4	1000	10000	403,0446	[401,0364;405,0527]	408,0949	[403,3814;412,8084]
8	100	1000	10,0762	[9,6666;10,4859]	13,6188	[13,5278;13,7099]
8	1000	1000	85,6115	[85,2376;85,9853]	87,7859	[87,4102;88,1615]
8	100	10000	82,0359	[81,6462;82,4256]	84,0760	[83,3357;84,8164]
8	1000	10000	814,7784	[811,6131;817,9438]	810,9634	[804,2905;817,6364]

Tabelle A.2: Messwerte bei der Ausführung der Testfälle 1 und 2 mit verschiedenen Parametern auf dem ASUS-Notebook

A.2 Messergebnisse - Testfall 3

Die Messergebnisse der Ausführung von Testfall 3 (4.4.3) mit verschiedenen Parametern sind in den Tabellen A.3 und A.4 angegeben. Gemessen wurde ausschließlich die Laufzeit der ersten Thread-Pool-Instanz, die eine parallele Matrixmultiplikation auf zwei zufälligen Matrizen der in der Spalte *Matrixgröße* angegebenen Größe ausführte. Jede weitere Instanz führte gleichzeitig die in *Tasks/ Instanz* angegebene Anzahl an π-Tasks aus. Die Anzahl der Iterationen ist in eckigen Klammern in derselben Spalte aufgeführt. Die Zeitmessung begann mit Übergabe des ersten Tasks und endete, nachdem alle Tasks der Matrixmultiplikation ausgeführt wurden. Die Einheit der angegebenen Zeitwerte für die Mittelwerte und Konfidenzintervalle ist auch hier Millisekunden (ms). Für jeden Test wurde die in *#Instanzen* angegebene Anzahl an Thread-Pool-Instanzen auf die gleiche Art wie in Abschnitt A.1 erzeugt. Bei der Verwendung von Instanzen der Klasse ThreadPoolExecutor erhielt jede Instanz bei der parallelen Matrixmultiplikation genauso viele Tasks wie maximal gleichzeitig ausgeführt werden konnten, wohingegen eine Instanz der Klasse CommunicativeThreadPoolExecutor einen Task pro Matrixzeile erhalten hat.

			kommunizierender Thread Pool		ThreadPoolExecutor	
#Instanzen	Matrixgröße	Tasks/ Instanz	Mittelwert	Konfidenzintervall(95%)	Mittelwert	Konfidenzintervall(95%)
2	300 × 300	1000[1000]	5,8593	[5,1494;6,5692]	5,8986	[5,8689;5,9283]
2	300 × 300	5000[1000]	13,0389	[13,0110;13,0667]	13,7690	[13,6557;13,8823]
2	500 × 500	100[1000]	12,6571	[12,6016;12,7127]	13,3194	[13,2499;13,3888]
2	500 × 500	1000[1000]	19,7602	[19,6630;19,8573]	26,1681	[25,9898;26,3463]
2	500 × 500	5000[1000]	19,3824	[19,3244;19,4403]	28,9219	[28,5985;29,2453]
2	500 × 500	100[5000]	19,8017	[19,6232;19,9801]	25,0572	[24,9686;25,1459]
2	500 × 500	1000[5000]	19,4369	[19,3549;19,5190]	25,4550	[25,3084;25,6016]
4	300 × 300	1000[1000]	7,7813	[7,7433;7,8193]	60,1768	[60,0214;60,3322]
4	500 × 500	1000[1000]	34,4471	[34,3268;34,5674]	72,4179	[71,9933;72,8424]
4	500 × 500	2000[1000]	33,7458	[33,6093;33,8824]	92,6819	[91,2974;94,0664]

Tabelle A.3: Messwerte bei der Ausführung von Testfall 3 mit verschiedenen Parametern auf dem Desktop-PC

			kommunizierender Thread Pool		ThreadPoolExecutor	
#Instanzen	Matrixgröße	Tasks/ Instanz	Mittelwert	Konfidenzintervall(95%)	Mittelwert	Konfidenzintervall(95%)
2	304 × 304	1000[1000]	21,1207	[21,0647;21,1768]	23,7967	[23,6223;23,9712]
2	504 × 504	100[1000]	49,1891	[49,0954;49,2828]	50,0932	[49,8619;50,3245]
2	504 × 504	1000[1000]	58,7811	[58,6475;58,9147]	66,4270	[65,8179;67,0360]
2	504 × 504	5000[1000]	76,2273	[75,4997;76,9548]	108,2533	[107,3715;109,1352]
2	504 × 504	100[5000]	54,9551	[54,4060;55,5043]	56,8732	[56,5328;57,2136]
4	304 × 304	1000[1000]	39,3233	[38,5441;40,1026]	44,3115	[44,0437;44,5792]
4	504 × 504	1000[1000]	77,0634	[76,6525;77,4744]	87,1950	[86,6582;87,7318]
4	504 × 504	2000[1000]	77,1393	[76,7235;77,5551]	122,6361	[121,8306;123,4416]
8	304 × 304	1000[1000]	75,6117	[75,4152;75,8081]	86,1210	[85,7986;86,4434]
8	504 × 504	1000[1000]	123,1421	[122,7100;123,5742]	129,0158	[128,4955;129,5360]
8	504 × 504	2000[1000]	196,3376	[194,9751;197,7001]	202,9775	[201,7586;204,1965]

Tabelle A.4: Messwerte bei der Ausführung von Testfall 3 mit verschiedenen Parametern auf dem ASUS Notebook

A.3 Messergebnisse - Testfall 4

Die Tabellen A.5 und A.6 enthalten die Messergebnisse bei Anwendung des vierten Testfalls (4.4.3) mit verschiedenen Parametern. Es gilt hier weitestgehend die Beschreibung der Parameter aus Abschnitt A.2. Einziger Unterschied ist, dass Instanzen der Klasse ThreadPoolExecutor ebenso wie Instanzen der Klasse CommunicativeThreadPoolExecutor einen Task pro Matrixzeile bei der parallelen Matrixmultiplikation erhalten haben.

			kommunizierender Thread Pool		ThreadPoolExecutor	
#Instanzen	Matrixgröße	Tasks/ Instanz	Mittelwert	Konfidenzintervall(95%)	Mittelwert	Konfidenzintervall(95%)
2	500 × 500	100[1000]	12,7165	[12,6718;12,7612]	12,7265	[12,6772;12,7757]
2	500 × 500	1000[1000]	19,6384	[19,5303;19,7465]	27,5777	[25,7984;29,3570]
2	1000 × 1000	2000[1000]	228,6615	[227,9732;229,3499]	225,6798	[225,3102;226,0494]
4	300 × 300	300[1000]	7,7984	[7,7587;7,8381]	21,2516	[21,1874;21,3159]
4	300 × 300	1000[1000]	7,7908	[7,7468;7,8348]	62,5267	[62,3554;62,6980]
4	500 × 500	100[1000]	16,5773	[16,5323;16,6223]	17,1749	[17,0993;17,2504]
4	500 × 500	1000[1000]	34,0880	[33,9290;34,2469]	70,7095	[70,5067;70,9123]
4	500 × 500	100[5000]	34,3397	[34,0509;34,6286]	38,3966	[38,2676;38,5257]

Tabelle A.5: Messwerte bei der Ausführung von Testfall 4 mit verschiedenen Parametern auf dem Desktop-PC

			kommunizierender Thread Pool		ThreadPoolExecutor	
#Instanzen	Matrixgröße	Tasks/ Instanz	Mittelwert	Konfidenzintervall(95%)	Mittelwert	Konfidenzintervall(95%)
2	504 × 504	100[1000]	49,0532	[48,9691;49,1372]	49,0758	[49,0079;49,1437]
2	504 × 504	1000[1000]	58,5458	[58,3782;58,7134]	59,1756	[58,9834;59,3677]
2	504 × 504	3000[1000]	76,4075	[75,5334;77,2816]	79,5784	[79,1200;80,0368]
4	304 × 304	1000[1000]	39,4888	[39,1404;39,8373]	43,0304	[42,6394;43,4214]
4	304 × 304	3000[1000]	40,1306	[39,9304;40,3307]	77,2297	[75,5166;78,9429]
4	504 × 504	1000[1000]	77,5939	[77,3081;77,8798]	81,3005	[81,0777;81,5234]
4	504 × 504	3000[1000]	126,5454	[125,7178;127,3729]	144,3802	[143,2761;145,4843]
4	504 × 504	5000[1000]	141,0398	[136,4847;145,5948]	189,1572	[185,8163;192,4980]
8	160 × 160	500[1000]	12,1113	[12,0822;12,1404]	39,7575	[38,6887;40,8263]
8	160 × 160	1000[1000]	12,1555	[12,1270;12,1840]	71,9671	[70,4177;73,5165]
8	304 × 304	1000[1000]	75,5876	[75,4354;75,7398]	84,7733	[84,1446;85,4020]
8	304 × 304	3000[1000]	83,3189	[82,9887;83,6491]	165,0761	[163,6315;166,5208]
8	504 × 504	1000[1000]	126,8662	[126,3001;127,4322]	128,4019	[127,9419;128,8619]

Tabelle A.6: Messwerte bei der Ausführung von Testfall 4 mit verschiedenen Parametern auf dem ASUS Notebook