Effiziente Threadkommunikation im gemeinsamen Speicher mithilfe lock-basierter und lock-freier Channel

Henrik Laubert

Bayreuth Reports on Parallel and Distributed Systems

No. 17, März 2022

University of Bayreuth
Department of Mathematics, Physics and Computer Science
Applied Computer Science 2 – Parallel and Distributed Systems
95440 Bayreuth
Germany

Phone: +49 921 55 7701
Fax: +49 921 55 7702
E-Mail: brpds@ai2.uni-bayreuth.de

Bachelorarbeit

Henrik Laubert

20.09.2021

Effiziente Threadkommunikation im gemeinsamen Speicher mithilfe lock-basierter und lock-freier Channel

Efficient thread communication in shared memory using lock-based and lock-free channels

Bachelorarbeit
Universität Bayreuth
Lehrstuhl für Angewandte Informatik 2

Betreuer: Prof. Dr. Thomas Rauber
Prof. Dr. Matthias Korch

Henrik Laubert (1571600)
Bayreuth, 20. September 2021

Zusammenfassung

In seiner Dissertation entwickelte Dr. Prell ein Laufzeitsystem zur parallelen Programmierung. Ziel dieses Laufzeitsystems ist es, dem Entwickler das Verwalten und Koordinieren von Threads abzunehmen. Der Programmierer soll nur noch Tasks spezifizieren, während das Laufzeitsystem für eine effiziente, parallele Ausführung dieser Tasks sorgt. Um die Thread-Koordination beim Zugriff auf gemeinsame Variablen vor dem Entwickler verbergen zu können, wurde eine zusätzliche Abstraktionsebene der Kommunikation geschaffen. Die Threads kommunizieren innerhalb des Laufzeitsystems ausschließlich über Channel. Durch die explizite Thread-Kommunikation im gemeinsamen Speicher ergeben sich eine Reihe von Vorteilen. Channel sind einfach zu implementieren, ermöglichen eine asynchrone Thread-Kommunikation und das Laufzeitsystem ist durch die explizite Kommunikation leicht portierbar, da es nicht länger an einen gemeinsamen Adressraum gebunden ist. Vor allem ermöglichen die Channel, die low-level Kommunikation der Threads vom Rest des Laufzeitsystems abzukoppeln und so zu verbergen.

In dieser Arbeit werden Möglichkeiten zur Verbesserung der Performance dieser Channel genauer betrachtet. Dazu werden verschiedene Channel-Varianten implementiert. Verwendet werden dazu verschiedene Lock-Strategien, die mit atomaren Operationen umgesetzt wurden. Ebenfalls wird eine alternative, interne Datenstruktur des Puffers des Channels getestet, sowie eine lock-freie Lösung eines Channels umgesetzt. Während mit der Änderung der im Puffer verwendeten Datenstruktur keine Verbesserungen erreicht werden, ergeben sich durch die Nutzung atomarer Operationen zur Thread-Koordination anstelle der Synchronisationsoperationen der `pthread.h`-Bibliothek Effizienzgewinne. Auch die lock-freie Lösung zeigt eine bessere Performance als die Ausgangsversion.

Der Datendurchsatz der Channel wird durch Messungen auf einem Intel Xeon E5-2630 CPU mit 16 physischen Kernen (je zwei Hardwarethreads pro Kern) miteinander verglichen. Als Messszenario diente ein Stresstest, in welchem eine bestimmte Anzahl von Threads so schnell wie möglich versuchen, die ihnen zugeteilten Nachrichten über den Channel zu versenden. So kann die Performance der verschiedenen Channel bei sehr hohen Zugriffsraten gemessen werden.

Abstract

In his thesis, Dr. Prell developed a runtime system for parallel programming. His goal was to create a runtime system that relieves the developer from managing and coordinating threads. The programmer should only have to specify tasks, while the runtime system ensures efficient, parallel execution of these tasks. Therefore, an additional level of abstraction was added to the thread communication, to hide the thread coordination at the access on shared variables from the developer. Within the runtime system threads exclusively communicate via channels. This explicit thread communication in shared memory results in several advantages. Channels are easy to implement, enable asynchronous thread communication, and make the runtime system easily portable due to the explicit communication not linked to shared memory. Above all, the channels decouple the low-level communication of threads from the runtime system and hide it from the developer.

This bachelor thesis investigates the possibility of optimization for channels. For this purpose, different channel variations are implemented, using various lock strategies and atomic operations. Furthermore, an alternative data structure of the channels' buffer is tested, and a lock-free channel solution is implemented. While there is no success in varying the data structure of the buffer it becomes quite clear that using atomic operations for thread coordination is more efficient than using the synchronization operations of the `pthread.h` library. The lock-free solution also shows better performance than the initial channel version.

The different channels' data throughput is compared by measurements on an Intel Xeon E5-2630 CPU with 16 physical cores and 32 hardware threads in total. The measurement scenario is a stress test, in which a certain number of threads try to send their messages via the channel as fast as possible. This test makes it possible to measure the performance of the various channels at very high access rates.

Inhaltsverzeichnis

1 Einleitung

Aufbauend auf die Dissertation „Embracing Explicit Communication in Work-Stealing Runtime Systems“ [15] von Dr. Prell soll in dieser Arbeit die explizite Kommunikation von Threads, über Channel, im gemeinsamen Adressraum weiter untersucht werden.

Mehrkernprozessoren sind bereits ein fester Bestandteil moderner Rechner, auch im privaten Bereich. Der Trend einer steigenden Anzahl von Prozessorkernen auf CPUs bricht nicht ab und Mehrkernprozessoren sind in unserem Alltag schon längst allgegenwärtig. Um auf diese Entwicklung zu reagieren und das Potenzial von Mehrkernprozessoren zu nutzen, werden Techniken zur parallelen Programmierung immer wichtiger [15]. Dazu hat Dr. Prell ein Laufzeitsystem als Alternative zur klassischen Thread-Programmierung entwickelt, da diese als zu komplex gilt, um der breiten Masse an Programmierern ein geeignetes Programmiermodell für das Entwickeln effizienter, paralleler Software zu bieten [15]. Der Fokus wurde von Dr. Prell verstärkt auf die Aufteilung der Berechnung in kleine, unabhängige Teile und deren Spezifikation als Tasks durch den Programmierer gelegt. Die Implementierung der Threads und ihre Kommunikation und Koordination soll dabei komplett von seinem Laufzeitsystem übernommen und vor dem Programmierer verborgen werden. Tasks und deren Verwaltung in Taskpools sind dabei kein völlig neues Konzept, sondern haben sich bereits als bewährte Technik zur Arbeitslastverteilung in parallelen Programmen etabliert und sind Gegenstand aktueller Forschung. [6, 7, 8, 16]

Um die Verwaltung der Tasks von der low-level Kommunikation der Threads zu trennen, wurden Channel eingesetzt. Channel sind ebenfalls eine weitverbreitete Technik für direkte Kommunikation zwischen Prozessen und sind beispielsweise ein fester Bestandteil der Programmiersprachen Go und Rust [18, 19]. Zudem bieten Channel eine Reihe von Vorteilen gegenüber der impliziten Threadkommunikation über gemeinsame Variablen [15]. Dadurch, dass die Kommunikation ausschließlich über Channel stattfindet, entfällt die aufwendige Synchronisierung von Threads beim Zugriff auf geteilte Datenstrukturen. Channel sind einfach zu implementieren und bieten eine hohe Flexibilität. Durch die explizite Kommunikation über Channel ist man nicht länger auf Cache-Kohärenz und gemeinsame Adressräume angewiesen. Dadurch wird die Laufzeitumgebung unabhängig von aktuellen Hardwarearchitekturen und zukünftigen Entwicklungen in diesem Bereich und ist deshalb leicht portierbar. Außerdem ermöglichen gepufferte Channel einen asynchronen Informationsaustausch zwischen den Threads. So blockieren Threads nicht länger bei der Kommunikation und diese kann sich mit der Ausführung von Tasks überlagern. [15]

In „Embracing Explicit Communication in Work-Stealing Runtime Systems“ spielen die Channel allerdings eine untergeordnete Rolle. Für die Threadkommunikation im Laufzeitsystem wurde eine Schnittstelle für den Zugriff auf die Channel entworfen und eine erste Channelversion, die mit Mutexvariablen des POSIX-Standards arbeitet, implementiert. Eine genauere Betrachtung der Channel, zum Beispiel eine Auswahl verschiedener Lock-Strategien und eine Untersuchung auf eventuelle Performanceunterschiede, fand aber nicht statt. Im Fokus stand vor allem das Erstellen, Verteilen und Ausführen der Tasks.

Nun soll der Fokus auf die Channel gelegt werden. Als zentraler Teil der Threadkommunikation des Laufzeitsystems haben sie einen erheblichen Einfluss auf dessen Effizienz.

Ziel der Arbeit ist es einen ersten Einblick darin zu erhalten, wie die Channel optimiert und so die Effizienz des Laufzeitsystems gesteigert werden könnte. Dazu stehen eine Vielzahl von Algorithmen, die den wechselseitigen Ausschluss von Threads bei Verwendung der Channel sicherstellen, zur Verfügung. Es wurden einige dieser Algorithmen ausgewählt und mit ihnen weitere Channelversionen implementiert. Neben den einfach zu implementierenden und deshalb weit verbreiteten, lock-basierten Algorithmen wurden auch lock-freie Algorithmen mit einbezogen. Lock-freie Algorithmen haben einige Vorteile gegenüber lock-basierten, da bei lock-freien Algorithmen langsame Threads schnellere Threads nicht ausbremsen. Lock-freie Algorithmen sind aber auch wesentlich komplexer und schwerer umzusetzen [17]. Daher ist es interessant zu betrachten, ob sich bei der Verwendung eines lock-freien Channels eine Verbesserung erkennen lässt und eine weiterführende Untersuchung sinnvoll ist. Da es sich um gepufferte Channel für den asynchronen Nachrichtenaustausch handelt, wurde ebenfalls untersucht, ob sich eine andere Datenstruktur besser als Puffer zum Zwischenspeichern der Nachrichten eignet. Anschließend wurden die Laufzeiten und Datenübertragungsraten der dabei entstandenen Channel miteinander verglichen.

2 Technischer Hintergrund

Zunächst sollen in diesem Kapitel technische Begrifflichkeiten definiert, sowie technische Hintergründe und Grundlagen erläutert werden, die nötig sind um das Vorgehen in den weiteren Kapiteln nachzuvollziehen.

Abschnitt 2.1 befasst sich mit der Motivation für die verwendung von parallelen Programmen und wie sich diese von sequentiellen unterscheiden.

Abschnitt 2.2 zeigt den Unterschied zwischen parallelen Programmen mit gemeinsamen und verteilten Speicher.

In Abschnitt 2.3 und 2.4 wird unter anderem der zentrale Begriff *Thread* eingeführt und kurz auf das zugrunde liegende Prinzip, sowie dessen Umsetzung eingegangen.

Abschnitt 2.5 erklärt was unter einem *Channel*, über den die Kommunikation der Threads abläuft, zu verstehen ist.

Die Abschnitte 2.6, 2.7 und 2.8 geben einen Überblick über *atomaren Operationen* und deren Funktionsweise. Sie werden für die Zugriffskontrolle in den meisten Channeln verwendet. Abschnitt 2.7 geht dabei kurz auf eine Problematik bei der Verwendung, der atomaren *Compare-and-swap* Operation ein.

In Abschnitt 2.9 werden die Begriffe *lock-frei* und *nicht-blockierend* definiert.

2.1 Parallele Programmierung

Eine treibende Kraft, bei der Entwicklung von neuer Hard- und Software ist die ständig steigende Nachfrage, nach mehr Rechenleistung, um größere und komplexere Berechnungen in kürzerer Zeit durchführen zu können. In der Vergangenheit wurde diese Steigerung der Rechenleistung durch die Erhöhung der Taktrate von Prozessoren erreicht. Dabei stößt man jedoch inzwischen an physikalische Grenzen, denn durch eine höhere Taktrate erhöht sich auch die Hitzeentwicklung im Prozessor. Zudem verkürzt sich die Zeit, die den Signalen zur Verfügung stehen, um Strecken innerhalb des Prozessors zurückzulegen, wodurch man gezwungen ist die Architektur möglichst eng gepackt auf kleinen Flächen zu realisieren. [16]

Die parallele Verarbeitung von Berechnungen ist deshalb eine gute Alternative um wesentlich höhere Rechenleistungen zu erreichen, als durch die Beschleunigung der sequentiellen Verarbeitung erreichen kann. Weshalb sie in den letzten Jahrzehnten die primäre Methode zur Steigerung der Prozessorleistung war [2]. Für die parallele Verarbeitung wird spezielle Hardware benötigt, die die zeitgleiche Ausführung verschiedener Rechenschritte unterstützt [16]. Ganz allgemein ist für solche Hardware charakteristisch, dass sie mehrere Verarbeitungseinheiten besitzt, die unabhängig voneinander Berechnungen durchführen können. In der Regel handelt es sich dabei um *Mulitcore-Prozessoren* also Prozessoren die aus vielen unabhängigen Recheneinheiten, sogenannten *Kernen*, bestehen [16].

Um diese Eigenschaften der Hardware auszunutzen wird ebenfalls spezielle, auf die Hardware angepasste, Software benötigt, die so geschrieben wurde, dass die Hauptbe-

rechnung in kleinere Teilberechnungen aufgeteilt wird, die unabhängig voneinander auf den einzelnen Kernen ausgeführt werden können. Dabei ist vor allem auf eine korrekte Abarbeitung der Teilberechnungen zu achten, indem die notwendige Synchronisation und der notwendige Informationsaustausch sichergestellt wird. [16]

2.2 Geteilter und gemeinsamer Adressraum

Die Herangehensweisen beim parallelen Programmieren können grob in zwei unterschiedliche Programmiermodelle unterteilt werden: dem *verteilten Adressraum* (engl. *distributed memory*) und dem *gemeinsamen Adressraum* (engl. *shared memory*). [16]

Bei einem verteilten Adressraum hat jedes Teilprogramm seinen eigenen privaten Bereich im Speicher. Dieser kann auch physikalisch getrennt sein. Für den Austausch von Informationen müssen explizite Kommunikationsoperationen genutzt werden. Details zu diesen Operationen und zur *Message-Passing-Programmierung* können Kapitel 5 aus [16] entnommen werden.

Bei einem gemeinsamen Adressraum teilen sich die Teilprogramme einen Abschnitt im Speicher, auf den jeder direkt zugreifen kann. Dieses Vorgehen unterscheidet sich stark von dem vorherigen, denn die Kommunikation zwischen den Programmen findet hier durch Variablen statt, auf die gemeinsam zugegriffen wird. Die Schwierigkeit liegt bei diesem Vorgehen also nicht darin, den Austausch von Informationen zwischen den einzelnen Programmen, mit getrennten Adressräumen, durch spezielle Operationen zu koordinieren, sondern darin den Zugriff auf diese gemeinsamen Variablen so zu steuern, dass nie der Fall auftritt in dem zwei Teilprogramme gleichzeitig versuchen dieselbe Variable zu manipulieren. Man bezeichnet solche Szenarien als *zeitkritische Abläufe* (engl. *race conditions*). Das kann zu inkonsistenten Werten und damit zu fehlerhaften Endergebnissen oder häufig auch zum Programmabsturz führen. Bereiche im Code, in denen zeitkritische Abläufe stattfinden können, werden als *kritischer Abschnitt* bezeichnet. [16]

Die Forschungen von Dr. Prell bringen die explizite Kommunikation aus dem Bereich des verteilten Adressraums mit in den Kontext des gemeinsamen Adressraums. Durch die Verwendung von *Channeln* für die Kommunikation in seinem Laufzeitsystem, findet diese ebenfalls ausschließlich durch explizite Methodenaufrufe statt und nicht durch den impliziten, direkten Zugriff auf geteilte Datenstrukturen. Zwar stellen diese Channel, die in Abschnitt 2.5 genauer erklärt werden, genaugenommen auch eine geteilte Datenstruktur dar, aber die Koordination der Teilprogramme, die auf die Channel zugreifen wird, von den Channeln selbst übernommen und ist nach außen, für den Programmierer der das Laufzeitsystem nutzt, unsichtbar.

2.3 Prozesse, Threads und Tasks

Für die Bezeichnung der einzelnen Teilprogramme haben sich die Begriffe *Prozess*, *Thread* und *Task* etabliert. Unter einem Prozess versteht man ein Programm, das gerade auf dem Rechner ausgeführt wird und alle weiteren Informationen die zur Ausführung

des Programms nötig sind. In UNIX-Betriebssystemen können Prozesse geklont werden. Dabei wird eine identische Kopie eines Prozesses in seinem aktuellen Zustand erstellt. Die geklonten Kindprozesse führen das gleiche Programm aus, besitzen aber nur eine *Kopie* des Adressraums des Elternprozesses und haben eine eigene Prozess-ID. Durch Anweisungen, die abhängig von der Prozess-ID ausgeführt werden, entsteht die, im vorherigen Abschnitt beschriebene, Parallelität. Zwei Prozesse bearbeiten nun verschiedene Teile des gleichen Programms. Da der Kindprozess nur eine Kopie des Adressraums besitzt und sich nicht den Adressraum mit dem ursprünglichen Prozess teilt, handelt es sich um dabei um einen verteilten Adressraum. Im Kontext von paralleler Programmierung im verteilten Adressraum werden die Teilprogramme daher bevorzugt als Prozess bezeichnet. [16]

Im Kontext eines gemeinsamen Adressraums und der Kommunikation über gemeinsame Variablen ist dabei eher von Threads die Rede. Threads sind unabhängige Kontrollflüsse innerhalb eines Prozesses. Diese teilen sich den Adressraum des Prozesses. Wenn ein Thread einen Wert im Adressraum ablegt, können die anderen Threads des gleichen Prozesses diesen lesen. Auf diese Weise können Threads sehr schnell und ohne explizite Kommunikationsmethoden untereinander Nachrichten austauschen. [16]

Tasks sind zwar kein Teil dieser Arbeit, spielen aber in der Dissertation von Dr. Prell eine große Rolle, und wurden deshalb in der Hinleitung zum Thema mehrfach erwähnt, weshalb sie nun hier kurz erklärt werden sollen. Tasks sind eine Reihe von Instruktionen, die parallel mit anderen Tasks ausgeführt werden können. Im Allgemein versteht man dabei unter einem Task einen Programmabschnitt, üblicherweise eine Funktion, und die für diesen Abschnitt benötigten Parameter [15]. Man unterteilt das Programm also in noch abzuarbeitende Programmabschnitte, die Tasks, und teilt diese dann den einzelnen Threads zu. Durch die Trennung von Threads, als Kontrollflüsse die in der Lage sind Berechnungen auszuführen, und Tasks, die den entsprechenden Code für die Berechnungen enthalten, wird vermieden, dass für jede Teilaufgabe ein extra Thread erstellt und anschließen wieder vernichtet werden muss. [16]

In dieser Arbeit wird speziell die Kommunikation von Threads im gemeinsamen Adressraum betrachtet. Dazu werden verschiedene Möglichkeiten, den Zugriff auf die gemeinsamen Variablen, innerhalb der Channel, zu steuern, implementiert und anschließend auf Effizienz bei hohen Zugriffsraten auf den Channel bewertet.

2.4 Pthreads und der POSIX Standard

Die Thread-Funktionalität wurde für diese Arbeit mit POSIX Threads, kurz *Pthreads* umgesetzt. Dafür wurde die Programmiersprache C, verwendet. Sie unterstützt den POSIX-Standard in ihrer Pthread-Bibliothek. Pthread ist eine standardisierte Schnittstelle zur Anwendungsprogrammierung mit Threads, die im POSIX-Standard (Portable Operating System Interface) seit 1995 definiert ist. Der POSIX-Standard wird von vielen UNIX-Betriebssystemen unterstützt, dazu gehören unter anderen macOS und die GNU/Linux Varianten. [14, 16]

Zusätzlich zu den Funktionen für die Erzeugung und Verwaltung von Threads bietet der Pthread-Standard auch eigene Lösungen zur Thread-Koordination und dem Thread-Scheduling. [16]

Genauere Einzelheiten zur Thread-Programmierung mit Pthreads können in Kapitel 6.2 aus [16] gefunden werden.

2.5 Channel

Anstelle der üblichen impliziten Threadkommunikation über geteilte Variablen liegt der Fokus, wie bei Dr. Prell auch, in dieser Arbeit auf dem expliziten Nachrichtenaustausch über *Channel*. Die Grundstruktur der Channel entspricht dabei der Struktur des Channels, den Dr. Prell in seiner Arbeit vorstellt [15]. Der Informationsaustausch zwischen den Threads findet nicht dadurch statt, dass ein Thread zu übermittelnde Werte direkt in eine gemeinsame Variable schreibt, auf die dann alle anderen Threads zugreifen können. Stattdessen wird der Informationsaustausch um eine zusätzliche Ebene ergänzt. Nachrichten werden durch einen expliziten Zugriff auf den Channel versendet oder empfangen. Die Channel bieten dafür eine Schnittstelle zum Versenden und Empfangen von Nachrichten, bei deren Verwendung der Programmierer nicht darauf achten muss, auf welche Weise die Threads ihre Informationen austauschen. Channel-intern, auf der untersten Ebene, findet allerdings immer noch eine Kommunikation über eine gemeinsame Datenstruktur statt. Diese Datenstruktur ist nach außen aber nicht sichtbar. Die Koordination der Threads beim Zugriff auf diese Datenstruktur erfolgt ausschließlich innerhalb des Channels und wird durch die Schnittstelle vor dem Nutzer verborgen.

Die Threads, die auf einen solchen Channel zugreifen, lassen sich in zwei Gruppen unterteilen. Die *Sender*, die Nachrichten über den Channel versenden möchten, und die *Empfänger*, die die Nachrichten erhalten sollen. Sender werden auch als *Produzenten* (engl. *producer*) bezeichnet, Empfänger auch als *Konsumenten* (engl. *consumer*). Diese Rollen sind aber nicht statisch vergeben, sondern hängen alleine davon ab in welchem Kontext ein Thread gerade auf den Channel zugreift.

Bei einem Channel handelt es ich um eine *FIFO (First In First Out)-Warteschlange* auf die, durch verschiedene Funktionen, welche die Zugriffe koordinieren, lesend und schreibend zugegriffen werden kann.

Eine FIFO-Warteschlange ist eine spezielle Liste, in der Nachrichten nur an einem Ende entnommen und nur am anderen Ende angehängt werden. So wird sichergestellt, dass Nachrichten die als Erstes in die Warteschlange geschrieben wurden, auch als Erstes aus ihr entnommen werden. [5]

In den folgenden Kapiteln wird, wenn nur die FIFO-Warteschlange alleine gemeint ist, von einem *Puffer* gesprochen. Mit *Channel* wird auf das vollständige Konstrukt aus Puffer und Zugriffsoperationen Bezug genommen.

In den meisten Channeln wurde der Puffer als Array mit einer bestimmten Länge $n, n > 0$, umgesetzt. Zusätzlich werden zwei Zähler `head` und `tail` verwaltet. Sie enthalten den Index des Anfangs, beziehungsweise des Endes des Puffers. `head` zeigt dabei immer auf

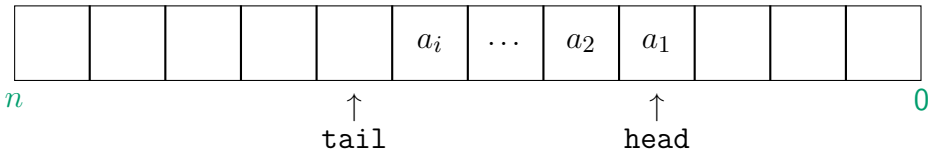

Abbildung 2.1: FIFO-Warteschleife als Array [5]

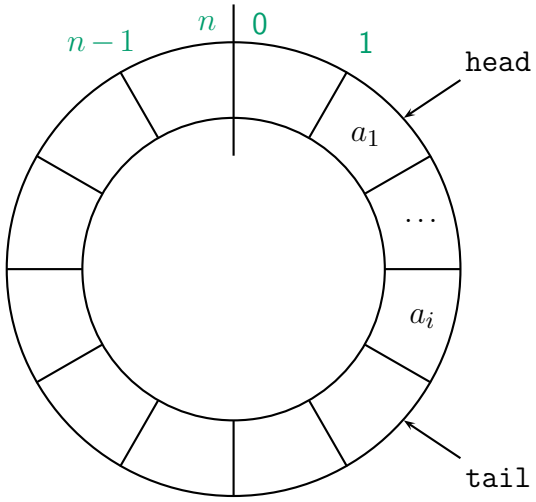

Abbildung 2.2: Zyklische FIFO-Warteschleife [5]

das erste belegte Feld des Arrays, `tail` auf das erste freie Feld (vgl. Abbildung 2.1).

Wenn ein Thread eine Nachricht über den Channel versenden möchte, ruft er die entsprechende `channel_send()`-Funktion auf. Die Funktion stellt zunächst sicher, dass gerade kein anderer Thread auf den `tail` und das entsprechende Pufferfeld zugreift, um zeitkritische Abläufe zu vermeiden. Greift gerade kein anderer Thread auf den Tail des Puffers zu, wird die Nachricht in das Feld, auf das `tail` zeigt geschrieben. Anschließend wird `tail` um eins erhöht. Das Empfangen einer Nachricht läuft analog über die `channel_receive()`-Funktion und `head` ab. Die Art wie in diesen Funktionen die Zugriffe gesteuert werden, ist zwischen den verschiedenen Channel-Versionen unterschiedlich. Diese unterschiedlichen Strategien der Zugriffskontrolle in diesem Kontext zu bewerten ist das Hauptanliegen der Arbeit.

Durch das Einfügen und Entfernen von Nachrichten stößt die Warteschlange schnell an einem Ende des Puffers an (`tail`$= n$), obwohl noch Platz für weitere Nachrichten im Puffer verfügbar wäre, da bereits alte Nachrichten wieder entfernt wurden. Dieses Problem kann einfach umgangen werden, indem man den Puffer als zyklisch auffasst, das bedeutet, wenn das Ende des Puffers erreicht wurde beginnt man wieder von vorne, unter der Voraussetzung, dass noch Plätze frei sind (vgl. Abbildung 2.2). [5] Alle als Array implementierte Puffer wurden zyklisch umgesetzt.

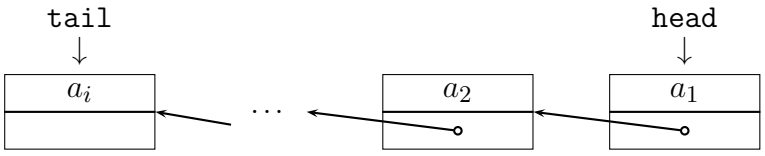

Abbildung 2.3: Eine einfach verkettete Liste [5]

Ist die Länge des Puffers $n = 1$, spricht man von einem *ungepufferten* oder *synchronen* Channel. Da im Puffer nur eine Nachricht gespeichert werden kann, muss vor dem Versenden einer weiteren Nachricht darauf gewartet werden bis der Empfänger die Nachricht gelesen hat.

Für jede Länge $n > 1$ spricht man von einem *gepufferten* oder *asynchronen* Channel. Hier muss nach dem Versenden einer Nachricht nicht gewartet werden, bis diese gelesen wurde, sondern es können direkt im Anschluss weitere Nachrichten versendet werden. Das Versenden einer Nachricht schlägt allerdings dann fehl, wenn alle Plätze des Puffers bereits mit einer Nachricht belegt sind.

Zwei Channel, die in Kapitel 3.4.1 und 3.6 ausführlicher beschrieben werden, verwenden eine andere Implementierung des Puffers. Hier wurde der Puffer nicht mit einem Array umgesetzt, sondern mit einer *einfach verketteten Liste* (engl. *singly linked list*).

Eine einfach verkette Liste besteht aus einzelnen Bausteinen, sogenannten *Elementen*, oder *Knoten* (engl. *node*), die zu einer Kette verbunden werden. Ein Element ist ein Objekt, in dem die Daten, die in der Liste gespeichert werden sollen, sowie einen Zeiger auf das nächste Element in der Kette, hinterlegt sind. Zusätzlich gibt es auch hier `head` und `tail` die auf das letzte und das erste Element der Liste zeigen (vgl. Abbildung 2.3). [5]

Um den Algorithmus zu vereinfachen zeigt dabei `head` immer auf ein Dummyelement, das keine relevante Nachricht enthält [17]. Auf diese Weise werden die Sonderfälle vermieden, die auftreten, wenn man ein Element in eine leere verkettete Liste einfügt, oder das letzte Element aus einer verketteten Liste entfernt. Ist die Liste leer zeigen `head` und `tail` auf das Dummyelement [17].

Mit der Verwendung einer einfach verketteten Liste anstatt eines Arrays wird versucht die Zeit, die ein Thread den Puffer blockieren muss, zu verkürzen, da das Listenelement unabhängig vom Puffer vorbereitet werden kann. Somit muss der Puffer nicht mehr für die Zeit, in der die Nachricht in das Element geschrieben wird, blockiert werden, sondern nur für die Zeit, die benötigt wird, um das neue Element einzuhängen. Außerdem ist bei einer einfach verketteten Liste die Anzahl der Nachrichten, die sie enthalten kann, theoretisch unbegrenzt.

2.6 Atomare Operationen

Atomare Operationen sind Operationen, deren Ausführung nicht, von anderen konkurrierenden Aktivitäten auf dem Rechner, unterbrochen werden kann. Das gilt zwar auch für

die meisten einfachen Befehle, die ein Prozessor unterstützt, wie zum Beispiel, das Lesen einer Variable aus dem Speicher, oder das Schreiben einer Variablen in den Speicher. Das bedeutet aber nicht, dass ein Thread ohne unterbrochen zu werden eine Variable aus dem Speicher lesen, manipulieren und wieder in den Speicher zurückschreiben kann. Viele moderne Architekturen unterstützen aber auch solche komplexeren Abläufe, die sich auch aus mehreren kleinen Operationen zusammensetzen lassen können, als atomare Operationen. [17]

Es ist wichtig festzuhalten, dass atomare Operationen von der Hardware explizit unterstützt werden müssen. Es ist nicht möglich auf Softwareebene eigene atomare Operationen zu implementieren, die nicht auf Hardwareebene umgesetzt wurden. [17]

Komplexe atomare Operationen sind sehr nützlich, um auf einfache Art den *wechselseitigen Ausschluss* (engl. *mutual exclusion*) von Threads umzusetzen, so dass sich die Threads untereinander daran hindern gleichzeitig auf eine Variable im Speicher zuzugreifen. [17]

Folgendes Beispiel verdeutlicht das:

Zwei Threads $T1$ und $T2$ konkurrieren gegenseitig um eine Variable x im Speicher, auf die beide Threads schreibend zugreifen möchten. Damit der Inhalt von x immer ein konsistenter Wert ist, ist es unbedingt notwendig, dass immer nur ein Thread gleichzeitig auf x zugreift. Um das zu erreichen wird der Variable ein *Lock* vorgeschaltet, das die Threads passieren müssen, bevor sie auf x zugreifen. Der erste Thread, der das Lock passiert, sperrt es, sodass der zweite Thread warten muss, bis das Lock wieder durch den ersten Thread entsperrt wird. In diesem Beispiel soll das Lock eine weitere Variable sein. Enthält die Variable den Wert 0, bedeutet das, dass gerade kein Thread auf x zugreift. Enthält sie den Wert 1, bedeutet das, dass gerade ein Thread auf x zugreift und x deshalb gesperrt ist. Das Passieren des Locks besteht für einen Thread also aus zwei Schritten. Zuerst liest er den Wert und vergleicht ihn mit 0. Ist der gelesene Wert gleich 0, schreibt der Thread den Wert 1 in das lock und beginnt x zu manipulieren. Ist der gelesene Wert gleich 1, muss der Thread warten, bis der Wert wieder auf 0 gesetzt wird. Das passiert, wenn der andere Thread seinen Zugriff auf x beendet hat. Würden die zwei Schritte aus zwei getrennten Operationen bestehen, bedeutet das, dass die Ausführung des einen Thread von dem anderen unterbrochen werden könnte. Das wäre vor allem zwischen Schritt 1 und 2 fatal. $T1$ liest den Wert des Locks, dieser ist noch 0. $T2$ unterbricht $T1$, liest ebenfalls den Wert des Locks, setzt ihn auf 1 und beginnt damit x zu manipulieren. Jetzt wacht $T1$ wieder auf, in der Annahme das Lock wäre frei, denn der Wert 0 wurde bereits gelesen. $T1$ setzt den Wert des locks ebenfalls auf 1 und beginnt nun zeitgleich mit $T2$ auf x zuzugreifen. Die Folge ist ein inkonsistenter Wert in x. Es ist also unbedingt notwendig, dass kein Thread, nachdem er den Wert des Locks gelesen hat, unterbrochen wird, bevor er Schritt 2 abgeschlossen hat. Eine atomare Operation, die beide Schritte in sich vereint, ist eine einfache denkbare Lösung.

Listing 2.1 zeigt die für diese Arbeit verwendeten atomaren Operationen, auf deren Funktionsweise und Verwendung im Folgenden kurz eingegangen werden soll.

Die *Test-and-set* Operation nimmt als Parameter eine Variable r und den Wert val

der in r gespeichert werden soll. Im Listing wird r *register* genannt, da es sich genau genommen um eine Speichereinheit des Prozessors handelt, in der der Wert von r liegt. Als atomarer Schritt wird dann der neue Wert val in r gespeichert und der alte Wert von r zurückgegeben. [17]
Diese atomare Operation würde sich auch zur Umsetzung des Locks im vorherigen Beispiel eignen. Sie wird auf eine ähnliche Weise im Channel aus Abschnitt 3.3 verwendet.

Die *Fetch-and-add* Operation nimmt ebenfalls eine Variable r und einen Wert val. Der Wert von r wird um val erhöht. Danach wird der alte Wert von r zurückgegeben. [3] Die Fetch-and-add Operation kann auf verschiedene Arten umgesetzt werden. So gibt es auch Versionen in denen der neue und nicht der alte Wert von r zurück gegeben wird [17]. In dieser Arbeit wurde aber die in Listing 2.1 vorgestellte Operation verwendet. Die Fetch-and-add Operation wird in den Channeln aus Abschnitt 5.7 verwendet, um einen Zähler in atomaren Schritten zu erhöhen.

Compare-and-swap bekommt eine Variable r, einen alten Wert old und einen neuen Wert new. Wenn der aktuelle Wert von r mit old übereinstimmt, wird new in r gespeichert und der Erfolg signalisiert. Ansonsten wird der Wert von r nicht geändert und $false$ zurückgegeben. [17]
Compare-and-swap ist eine sehr mächtige Operation, da sie zusammen mit einer Schreiboperation den Abgleich mit einem alten Wert ermöglicht. In den Channeln aus Abschnitt 3.5 und 3.6 wird sie sehr häufig verwendet. Sie wird genutzt, um den aktuellen Wert einer Variable mit einem älteren, bekannten Wert zu vergleichen und so festzustellen, ob eine Veränderung seit der letzten Aktivität des Threads vorliegt und gegebenenfalls den Zugriff auf den Puffer abzubrechen.

2.7 Das ABA-Problem von Compare-and-swap

Die Verwendung von Compare-and-swap Operationen bringt ein Problem mit sich, das bei oberflächlicher Betrachtung nicht sofort auffällt. Die Compare-and-swap Operation erhält bei ihrem Aufruf einen Wert old als Parameter (vgl. Listing 2.1). Dieser Wert wird mit dem aktuellen Wert der Variable r verglichen und abhängig vom Ergebnis des Vergleichs wird der Wert von r aktualisiert oder nicht. Der Parameter old entspricht also dem letzten bekannten, oder dem erwarteten Wert von r. Wenn sich der Wert von r von old unterscheidet bedeutet das, dass ein anderer Thread zwischenzeitlich auf r zugegriffen hat, weshalb die Compare-and-swap Operation fehlschlagen sollte. Es kann aber theoretisch passieren, dass ein Thread zwischen dem Lesen des alten Werts und dem Aufruf von Compare-and-Swap schlafen gelegt wird. In dieser Zeit könnte r von anderen Threads so manipuliert werden, dass es für den ersten Thread nicht nachvollziehbar ist, ob r in der Zwischenzeit manipuliert wurde oder nicht. Das nennt man das *ABA-Problem*.

Angenommen ein Thread $T1$ liest den Wert A aus r und wird anschließend schlafen gelegt. Nun greift Thread $T2$ auf r zu und ändert dessen Wert auf B. Anschließend setzt $T2$ den Wert von r wieder auf A. Nun wacht $T1$ auf und führt die Compare-and-swap Operation aus, da sich der Wert von r, aus der Perspektive von $T1$, scheinbar nie geändert

```
function test-and-set (r:register, val:value)
    temp := r;
    r := val;
    return(temp);
end-function

function fetch-and-add (r:register, val:value)
    temp := r;
    r := r + val;
    return(temp);
end-function

function compare-and-swap (r:register, old:value, new:value)
    if r = old then
        r := new;
        return(true);
    else
        return(false);
    fi;
end-function
```

Listing 2.1: Für die Arbeit verwendete atomare Operationen in Pseudocode (vgl. [17])

hat. In diesem Fall sollte die Compare-and-Swap Operation eigentlich fehlschlagen. Das ist aber beim ABA-Problem nicht der Fall, was zu Laufzeitfehlern führen könnte. [17]

Es besteht aber keine Möglichkeit dass das ABA-Problem in den Implementierungen der Channel auftritt, da Werte nie auf einen früheren Wert zurückgesetzt werden, oder es kein Problem darstellen würde, wenn Compare-and-swap nach unbemerkten Änderungen trotzdem ausgeführt wird.

2.8 Die Bibliothek stdatomic.h

Seit der Standardrevision C11 ist die Bibliothek `stdatomic.h` im Standard der Programmiersprache C enthalten. Diese Bibliothek bietet einige atomare Operationen und Datentypen an, ist dabei aber nicht an eine bestimmte Hardwarearchitektur gebunden. Das bringt den großen Vorteil mit sich, dass man Programme schreiben kann, die mit hardwarespezifischen, atomaren Operationen arbeiten und trotzdem portierbar sind. Die Referenzen auf atomare Funktionen werden erst beim Übersetzen des Programms durch einen Compiler aufgelöst, falls die atomaren Funktionen von der Hardware unterstützt werden. [3, 4]

Um diesen Vorteil zu nutzen, wurde für die Channel, die mit atomaren Operationen arbeiten, `stdatomic.h` verwendet, anstatt die Teile des Codes mit atomaren Operationen in plattformabhängigen Assemblercode zu schreiben.

2.9 Nicht-blockierende, lock-freie Algorithmen

Den wechselseitigen Ausschluss beim Zugriff auf geteilte Daten mithilfe von Locks umzusetzen ist sehr weit verbreitet. Das lässt sich auf das einfache Programmiermodell und die Verfügbarkeit verschiedener effizienter und skalierbarer Implementierungen zurückführen. Die Verwendung von Locks kann aber einen negativen Einfluss auf die Performance von Anwendungen haben, da Prozesse darauf warten müssen, dass Locks wieder freigegeben werden. Langsame oder innerhalb des kritischen Abschnitts gestoppte Prozesse könnten andere Prozesse sogar davon abhalten überhaupt auf die Daten zuzugreifen. Handelt es sich bei den Daten um einfache Datenstrukturen, wie zum Beispiel Warteschlangen, Stacks, Heaps, oder verkettete Listen, könnte dieses Problem vermieden werden, indem man lock-freie Datenstrukturen nutzt. [17]

Der Begriff lock-freie Algorithmen bezieht sich dabei auf Algorithmen, die in keiner Weise ein Lock verwenden. Bezüglich des Fortschritts der einzelnen Prozesse, die auf eine lock-freie Datenstrukturen zugreifen, gibt es zwei wichtige Bedingungen, *nicht-blockierend* (engl. *non-blocking*) und *wait-free*. Beide Bedingungen können nicht von lock-basierten Datenstrukturen erfüllt werden, da ein Prozess, der sich in einem kritischen Abschnitt aufhängt, alle anderen Prozesse blockieren kann.[17]

Eine Datenstruktur ist nicht-blockierend, wenn immer mindestens ein Thread in der Lage ist, seine ausstehenden Operationen in einer endlichen Anzahl von Rechenschritten auszuführen, unabhängig von der Ausführungsgeschwindigkeit anderer Prozesse. Ein nicht-blockierender Algorithmus ist per Definition lock-frei, da, wie oben beschrieben ein lock-basierter Algorithmus diese Eigenschaft nie erfüllen kann. Allerdings ist ein lock-freier Algorithmus nicht automatisch nicht-blockierend, auch wenn beide Begriffe häufig als Synonyme verwendet werden. [17]

Eine nicht blockierende Datenstruktur ist *wait-free*, wenn jeder Thread zu jeder Zeit in der Lage ist, seine ausstehenden Operationen in einer endlichen Anzahl von Rechenschritten auszuführen. [17]

Nicht-blockierende Algorithmen werden selten in Anwendungen verwendet, da sie oft sehr komplex sind [17], bieten aber den großen Vorteil, dass langsame Threads kein Hinderniss mehr für andere Threads darstellen und diese nicht mehr ausbremsen können.

3 Implementierung der Channel

Ausgehend von einer Channel-Implementierung von Dr. Prell, welche die Zugriffe auf den Puffer mit Mutexvariablen der Pthread-Bibliothek steuert, wurden im Rahmen der Arbeit noch sechs weitere Channel mit verschiedenen Lock-Strategien implementiert.

Dazu war es nötig die bereits verwendete API der Channel von Dr. Prell nach außen nicht zu verändern, sie Intern aber an die verschiedenen Channel-Implementierungen anzupassen.

Jeder Channel liegt in zwei Versionen vor. Zum einen als allgemeiner *Multiple Producer, Multiple Consumer* (MPMC) Channel, zum anderen als *Multiple Producer, Single Consumer* (MPSC) Channel. Die MPSC Versionen decken den speziellen Fall ab, in dem nur ein Thread als Empfänger auf den Puffer zugreift und deshalb mit keinem anderen Thread um die entsprechenden Variablen konkurriert. Die Funktionen zum Empfangen von Nachrichten können in diesen Versionen lock-frei umgesetzt werden. Durch dieses Vorgehen wird eine Verbesserung der Laufzeit erwartet [15].

Einen Spezialfall bildet der *Single Producer, Single Consumer* (SPSC) Channel. Weil hier jeweils nur ein Thread sendet bzw. empfängt werden weder beim Senden, noch beim Empfangen von Nachrichten Locks benötigt. Aufgrund fehlender Locks ist es nicht möglich, diesen Typ von Channel durch das Variieren von Lock-Strategien zu verbessern, weshalb er in dieser Arbeit nicht weiter betrachtet wird.

Zunächst werden in Abschnitt 3.1 und 3.2 zwei Channel vorgestellt, die Synchronisationsoperationen der `pthread.h`-Bibliothek nutzen. Danach folgen Channel die mit atomaren Operationen arbeiten.

Der Channel aus Abschnitt 3.5 ist dabei ein eigener Versuch einen Channel zu entwerfen, der teileweise lock-frei arbeitet und durch seinen Aufbau das Problem des hohen Speicheraufkommens durch busy-waiting verbessern soll.

In Abschnitt 3.6 wird schließlich noch ein lock-freier Channel vorgestellt.

3.1 Pthread Mutexlock (PT ML) Channel

Bei diesem Channel handelt es sich um die ursprüngliche Implementierung von Dr. Prell. Er nutzt für die Koordination der konkurrierenden Threads *Mutexvariablen*. Sie sind ein Hilfsmittel, das von der Pthread-Bibliothek gestellt wird [16]. Es handelt sich um eine Datenstruktur, die verwendet wird, um den wechselseitigen Ausschluss bei der Manipulation des Puffers sicherzustellen. Der Name leitet sich von dem englischen *mutual exclusion* (*wechselseitiger Ausschluss*) ab. Eine Mutexvariable kann die zwei Zustände *gesperrt* (engl. *locked*) oder *ungesperrt* (engl. *unlocked*) haben. Bevor ein Thread den Zugriff auf den Puffer startet, sperrt er die zugehörige Mutexvariable durch einen speziellen Funktionsaufruf und wird ihr Eigentümer. Nach Beendigung des Zugriffs gibt der Thread die Mutexvariable wieder frei. Ist die Mutexvariable bereits gesperrt, wird der Thread blockiert, also nicht weiter ausgeführt, bis die Mutexvariable wieder freigegeben wird. Die Reihenfolge in der blockierte Threads, nach der Freigabe der Mutexvariable, die

Kontrolle über diese erhalten, ist abhängig von ihren Prioritäten und dem verwendeten Scheduling-Verfahren. Die genaue Methode wird nicht vom Pthread-Standard vorgegeben, und hängt deshalb von der verwendeten Thread-Bibliothek ab. [10, 16]

Der Channel besitzt zwei Mutexvariablen, `tail_lock` und `head_lock`. `tail_lock` ist der `tail`-Variable des Puffers, die in Abschnitt 2.5 erläutert wurde, zugeordnet, und wird bei Zugriffen auf `tail` gesperrt. `head_lock` ist der `head`-Variable zugeordnet.

Soll eine Nachricht über den Channel verschickt werden, wird die `send()`-Funktion des Channels aufgerufen. Bevor der Thread in den *kritischen Abschnitt*, also dem Bereich im Code, in dem auf den Puffer zugegriffen wird, betritt, versucht er `tail_lock` zu übernehmen und zu sperren. Wenn das gelingt und der Puffer noch nicht voll ist, kopiert der Thread die Nachricht in den Puffer und sendet damit die Nachricht. Nun erhöht der Thread `tail`, sodass `tail` wieder auf das nächste freie Feld im Puffer zeigt. Anschließend gibt der Thread `tail_lock` wieder frei. Nun kann der nächste Thread, der gegebenenfalls schon bei dem Versuch die gesperrte Mutexvariable zu erhalten blockiert wurde, seine Ausführung fortsetzen und auf den Puffer zugreifen. In Listing 3.1 ist ein Codeausschnitt des kritischen Abschnitts und der Verwendung der Mutexvariablen in der `send()`-Funktion zu sehen. Das Kürzel `pml` ist dabei die Bezeichnung für einen Pthread Mutexlock Channel. Die Funktionen `pthread_mutex_lock()` (Zeile 2) und `pthread_mutex_unlock()` (Zeile 7 und 19) sind die Funktionen der pthread-API zum Sperren und Freigeben von Mutexvariablen. Das Makro `IS_FULL()` in Zeile 4 erhält einen Channel als Parameter und gibt `true` zurück, falls der Puffer des Channels voll ist. Die Funktion `memcpy` kopiert Byteweise Daten zwischen zwei gegebenen Adressen im Speicher. `INC` ist ebenfalls ein Macro. Es erhält als Parameter den aktuellen Wert von `tail` und die Größe des Puffers und berechnet mit $tail + 1 \mod size$ den nächsten Wert für `tail`. So wird der Puffer als zyklischer Array benutzt, wie in Abschnitt 2.5 beschrieben.

Das Empfangen einer Nachricht, mit der `recv()`-Funktion, läuft analog ab. Es wird `head_lock` gesperrt, die Daten aus dem Puffer kopiert und `head` um eins erhöht.

Das Versenden einer Nachricht über einen ungepufferten Channel stellt eine Besonderheit dar. Da der Puffer nur Raum für eine Nachricht bietet und entweder leer oder voll ist, wäre es unnötig mit `head` und `tail` zu arbeiten. Stattdessen wird nur eine der beiden Variablen, in diesem Fall `head` verwendet, um festzuhalten ob der Puffer gerade voll oder leer ist. Sowohl beim Senden, als auch beim Empfangen, wird also versucht `head_lock` zu erhalten, bevor auf den Puffer zugegriffen wird. Nach dem Senden einer Nachricht wird `head` mit dem Wert 1 belegt, was einen vollen Puffer signalisiert, bevor `head_lock` wieder freigegeben wird. Nach dem Empfangen der Nachricht wird `head` auf 0 gesetzt, um einen leeren Puffer zu signalisieren.

3.2 Pthread Spinlock (PT SL) Channel

Neben Mutexvariablen ist im POSIX-Standard auch ein *Spinlock* enthalten [11]. Dieser Channel ist fast identisch mit dem Pthread Mutexlock Channel aus Abschnitt 3.1, nutzt

```
1  // Die Mutexvariable für tail wird gesperrt
2  pthread_mutex_lock(&pml->tail_lock);
3
4  if (IS_FULL(pml)) {
5    // Die Mutexvariable für tail wird freigegeben,
6    // da der Channel bereits voll ist
7    pthread_mutex_unlock(&pml->tail_lock);
8    return false;
9  }
10
11 // Zugriff auf den Puffer
12 memcpy(pml->buffer + pml->tail * pml->itemsize,
13        data, size);
14
15 // tail wird um eins erhöht
16 pml->tail = INC(pml->tail, pml->size);
17
18 // Die Mutexvariable für tail wird freigegeben
19 pthread_mutex_unlock(&pml->tail_lock);
```

Listing 3.1: Ausschnitt der `send()`-Funktion des SHM Channels

aber anstatt der Mutexvariablen zwei dieser Spinlocks. Mit dieser Channelversion soll vor allem geprüft werden, welche Leistungsunterschiede zwischen, Mutex- und Spinlocks bestehen, die beide vom POSIX-Standard bereitgestellt werden. Zudem wurde der Channel aus Abschnitt 3.3 mit einem eigenen Spinlock, das nicht in der Pthread-Bibliothek enthalten ist, implementiert, um feststellen zu können, welche Unterschiede zur Verwendung eines Pthread-Spinlocks bestehen.

Ein Spinlock hat die gleiche Wirkung wie ein Mutexlock und garantiert, bei richtiger Verwendung, den wechselseitigen Ausschluss. Auch die Verwendung des Spinlocks unterscheidet sich kaum von der des Mutexlocks. Ersetzt man im Listing 3.1 alle `pthread_mutex_*` mit `pthread_spin_*` erhält man den entsprechenden Codeausschnitt für diesen Channel.

Der Unterschied zwischen Mutex- und Spinlock besteht darin, wie sich der Thread verhält, wenn der Versuch ein Lock zu sperren fehlschlägt. Während bei einem Mutexlock der betreffende Thread schlafen gelegt und, abhängig von der verwendeten Priorisierung und der verwendeten Scheduling-Methode, wieder aufgeweckt wird, *wartet* der Thread bei einem Spinlock bis es wieder frei ist. Das bedeutet, dass der Thread nicht aus seinem Aufruf von `pthread_spin_lock()` zurückkehrt, sondern immer wieder in einer Schleife prüft, ob das Lock noch gesperrt ist. Das nennt man *spinning* oder *busy-waiting*.

Spinlocks sind für kleine, kritische Abschnitte effizienter als ein Kontextwechsel, durch den man in der Zeit, in der auf das Lock gewartet werden muss, andere Aufgaben abarbeiten könnte [1]. Ein solcher Kontextwechsel würde auftreten, wenn ein Thread durch

ein Mutexlock schlafen gelegt würde. Bei dem Kopieren einer Nachricht, deren Größe sehr wahrscheinlich nur wenige Byte betragen wird, und dem Erhöhen eines Zählers handelt es sich beispielsweise um kleine Aufgaben. Deshalb kann der kritische Abschnitt als klein angesehen werden. Allerdings kann es zu einer schlechteren Performance führen, wenn mehrere Threads auf eine Variable warten, da alle Threads, in einer Dauerschleife, die gleiche Adresse im Speicher abfragen [17]. Es ist also durchaus interessant zu untersuchen, ob dieser Vorteil des Spinlocks den Nachteil der vielen Speicherzugriffe überwiegt.

3.3 Simple Spinlock (SL) Channel

Die Arbeiten [6] und [7] haben für andere Anwendungen ergeben, dass man durch die Verwendung von atomaren Operationen eine höhere Performance erzielen kann, als mit den Synchronisationsoperationen, Mutex- und Spinlock, des POSIX-Standards. Dieser Channel verwendet daher ein Spinlock, das mit atomaren Operationen umgesetzt wurde. Wie auch die vorherigen Channel besitzt der SL Channel zwei Locks für *head* und *tail*. Diese können ähnlich wie eine Mutexvariable entweder gesperrt oder ungesperrt sein. Sie sind als einfache *Test-and-set Locks* umgesetzt, die in einer Schleife, von den Threads, abgefragt werden, was sie zum Spinlock macht.

Wie bereits erwähnt, kennt das Test-and-set Lock zwei Zustände, gesperrt und ungesperrt [9]. Es eignen sich beispielsweise Variablen vom Typ `Boolean`, mit den Werten *true* und *false*, oder vom Typ `Integer` mit den Werten 0 und 1. In diesem Fall wurde für die Lockvariablen der Typ `atomic_flag` gewählt, ein interner Datentyp von `stdatomic.h`. Das entspricht einem Boolean auf den atomar zugegriffen werden kann. [3]

Um das Lock zu setzen, bietet sich die Compare-and-swap, oder die Test-and-set Operation an (vgl. Listing 2.1) [9]. Wichtig ist, dass die genutzte atomare Operation einen Vergleich des (alten) Zustands des Locks gegen einen Wert enthält, um festzustellen, ob das Lock bereits gesperrt ist. In diesem Fall wurde die Test-and-set Operation gewählt. Beim Versuch ein Lock zu sperren, setzt der Thread den Wert der Variable auf `true` (was *gesperrt* entspricht) und erhält den alten Wert des Locks als Rückgabewert von der Operation. Ist dieser Rückgabewert ebenfalls `true`, bedeutet das, dass das Lock bereits gesperrt ist. Der Thread tritt nicht in den kritischen Abschnitt ein, sondern wartet in einer Schleife darauf, dass das Lock freigegeben wird, das heißt, bis Test-and-Set `false` zurückgibt. Anschließend tritt er in den kritischen Abschnitt ein, greift auf den Puffer zu, erhöht `head` (beziehungsweise `tail`) und gibt das Lock wieder frei, indem er es auf `false` setzt.

Listing 3.2 zeigt noch einmal das gerade beschriebene Vorgehen in der `send()`-Methode des SL Channels. In Zeile 7 versucht der ausführende Thread das Lock zu sperren. Die Funktion `atomic_flag_test_and_set()` entspricht dabei der Test-and-set Operation wie sie in Listing 2.1 vorgestellt wurde. Sie ist Teil der `stdatomic.h`-Bibliothek und wird verwendet um Flags vom Typ `atomic_flag` zu setzen.

```
1  // Versuche in einer While-Schleife das Lock zu erhalten
2  do {
3    // Abbruch falls der Puffer währenddessen voll wird
4    if (IS_FULL(slc))
5    return false;
6    // Zugriff auf das Lock mit Test-and-set
7  } while (atomic_flag_test_and_set(&tail_lock));
8
9  // Abbruch falls der Puffer voll ist
10 if (IS_FULL(slc)) {
11   atomic_flag_clear(&tail_lock);
12   return false;
13 }
14
15 // Schreibender Zugriff auf den Puffer
16 memcpy(slc->buffer + slc->tail * slc->itemsize, data, size);
17
18 // Tail wird um eins erhöt
19 slc->tail = INC(slc->tail, slc->size);
20
21 // Das Lock wird freigegeben
22 atomic_flag_clear(&tail_lock);
```

Listing 3.2: Ausschnitt der `send()`-Funktion des SL Channels

Die Funktion `atomic_flag_clear()` ändert den Wert von Flags auf *nicht-gesetzt* und ist damit das Gegenstück zu `atomic_flag_test_and_set()`.

3.4 Ticketlock (TL) Channel

In den vorherigen Channeln, vor allem bei den Spinlockchanneln, kann es im schlechtesten Fall passieren, dass ein Thread zwar auf den Puffer zugreifen will, aber nie in den kritischen Abschnitt eintreten kann. Da nicht festgelegt ist welcher Thread als nächstes an der Reihe ist, bekommt der Thread das Lock der als erster versucht es zu sperren, nachdem es wieder freigegeben wurde. In welcher Reihenfolge die Threads Zugriff auf das Lock erhalten ist zufällig. Es könnte also passieren, dass ein Thread „Pech“ hat und immer wieder ein anderer Thread vor ihm Zugriff auf das Lock erhält. Das kann theoretisch so weit gehen, dass immer wieder neue Threads nachrücken und vor dem bereits wartenden Thread in den kritischen Abschnitt eintreten können, sodass der wartende Thread nie Zugriff auf den Channel erhält. In so einem Fall spricht man davon, das ein Thread *verhungert*.

Ein *Ticketlock* oder *Bakerylock* ist ein weiteres einfach umsetzbares Lock und im Gegensatz zu den Locks der vorherigen Channel ist es *starvation free*. Ein Algorithmus

ist *starvation free*, wenn jeder Prozess, der versucht den kritischen Abschnitt zu betreten, das auch irgendwann schafft. [17]

Die Idee hinter dem Ticketlock, ist ein Vorgehen, das man gelegentlich in einigen Ämtern, Arztpraxen oder Läden (zum Beispiel Bäckereien) beobachten kann. Kunden die den Laden betreten, ziehen ein Ticket mit einer Nummer und nehmen anschließend so lange im Wartebereich Platz, bis diese Nummer aufgerufen wird [9, 17]. Ein Ticketlock stellt also sicher, dass die Threads in der Reihenfolge, in der sie auf den Puffer zugreifen wollten, auch Zugriff auf den Puffer erhalten. Kein Thread kann einen bereits wartenden Thread überholen, und vor diesem auf den Puffer zugreifen. [17] Es ist deshalb ausgeschlossen, dass ein Thread verhungern kann.

Zur Umsetzung eines Ticketlocks werden zwei Zähler benötigt. Ein Zähler `next` enthält die Nummer des Tickets für den nächsten Thread, das entspricht der Anzahl wie oft Threads versucht haben das Lock zu sperren. Der andere Zähler `current` enthält die Nummer des Ticktes mit dem aktuell der kritische Abschnitt betreten werden darf, dieser Wert entspricht der Anzahl, wie oft die Sperre bereits aufgehoben wurde. [13]

Für die Implementierung wurden insgesamt vier Zähler verwendet, da, wie bei den anderen Channeln auch, für `head` und `tail` getrennte Locks verwendet werden. Ein Thread, der in den kritischen Abschnitt ein treten möchte, „zieht“ sich sein Ticket, indem er den aktuellen Wert in `next` in einer privaten Variable `MyTicket` speichert und den Wert von `next` um eins erhöht. Dafür wird die Fetch-and-add Operation verwendet, dieser erhöht `next` atomar um eins, sodass der nächste Thread eine neue Nummer erhält, und gibt den alten Wert zurück. Nun wartet der Thread bis er „aufgerufen“ wird. Das bedeutet er vergleicht in einer Schleife `MyTicket` mit `current`, bis beide denselben Wert haben. Nun betritt er den kritischen Abschnitt. Zuletzt erhöht er `current` um eins, und ruft so den nächsten Thread auf.

Listing 3.3 zeigt die Anwendung des Ticketlocks. Die Funktion `atomic_fetch_add()` in Zeile 2 ist die Fetch-and-add Operation von `stdatomic.h`. Sie wird verwendet um das Ticket zu ziehen und den Zähler für den nächsten Thread zu erhöhen. In der While-Schleife in Zeile 4 wartet der Thread anschließend bis er an der Reihe ist. Mit `pthread_yield()` wird er dabei nach jedem fehlgeschlagenen Versuch schlafen gelegt. `pthread_yield()` veranlasst den aufrufenden Thread, die CPU abzugeben. Der Thread wird an das Ende der Ausführungswarteschlange gestellt und ein anderer Thread wird für die Ausführung eingeplant [12]. So werden Threads, die gerade nicht an der Reihe sind, daran gehindert, das Lock mit unnötigen Abfragen zu blockieren.

Denn das Ticketlock bringt dasselbe Problem wie das Spinlock mit sich, da das Warten der Threads auch hier mit busy-waiting verbunden ist. Jedoch wirkt sich dieser Nachteil hier noch stärker aus. Während bei einem Spinlock sofort, nachdem das Lock wieder freigegeben ist, der nächste beliebige Thread, der gerade versucht das Lock zu erhalten, es auch sperren kann, ist beim Ticketlock die Reihenfolge, in der die Threads Zugang erhalten, festgelegt. Dadurch kommt eine zusätzliche Verzögerung hinzu, da darauf gewartet werden muss, dass der richtige Thread wieder Zugriff auf die Variable erhält, obwohl sich schon längst kein Thread mehr im kritischen Abschnitt befindet. [9]

Der Grund warum in diesem Channel trotzdem ein Ticketlock verwendet wird, ist, da so überprüft werden kann, wie groß das Problem des Verhungerns ist.

```
1  // Der Thread "zieht" sein Ticket und erhöht den Zähler
2  my_ticket = atomic_fetch_add(next_ticket_tail, 1);
3  // Der Thread wartet, bis sein Ticket "aufgerufen" wird
4  while(my_ticket != tlc->current_ticket_tail)
5  {
6    pthread_yield();
7  }
8  if (IS_FULL(tlc)){
9    tlc->current_ticket_tail++;
10   return false;
11 }
12 // Zugriff auf den Puffer
13 memcpy(tlc->buffer + tlc->tail * tlc->itemsize, data, size);
14 // Aktualiserung von tail
15 tlc->tail = INC(tlc->tail, tlc->size);
16 // Aufruf des Nachfolge Threads
17 tlc->current_ticket_tail++;
```

Listing 3.3: Ausschnitt der `send()`-Funktion des TL Channels

Es könnte sein, dass das Ticketlock, weil es das Problem des Verhungerns löst, einen so großen Vorteil mit sich bringt, dass dieser die Nachteile überwiegt.

3.4.1 Linked List Version

Speziell vom TL Channel wurde eine weitere Version implementiert, die zwar dasselbe Ticketlock verwendet, aber den Puffer nicht als Array, sondern als einfach verkettete Liste umsetzt. So soll verglichen werden welchen Effizienzvorteil es mit sich bringt, wenn alle Schreib- und Leseoperationen, die mit der Nachricht zusammenhängen, vor oder nach dem kritischen Abschnitt durchgeführt werden können.

Das Senden (beziehungsweise das Empfangen) einer Nachricht läuft etwas anders ab, als bei den Channeln, die einen Array als Puffer verwenden. Bevor der Thread auf das Ticketlock zugreift, wird die Nachricht vorbereitet. Das geschieht, indem ein neues Element allokiert wird und die Daten in dieses geschrieben werden. Jetzt erst greift der Thread auf das Lock zu und erhält sein Ticket.

Prüfungen, ob der Puffer voll ist, sind nicht nötig, da immer neue Elemente in die Liste eingefügt werden können, solange genug Speicherplatz verfügbar ist, um neue Elemente zu allokieren. Es gibt deshalb keine ungepufferte Version des Channels.

Sobald der Thread an der Reihe ist, hängt er das neue Element an das Ende der Liste. Dazu setzt er, im letzten Element der Liste, den Zeiger auf das nächste Element, sodass dieser auf die neue Nachricht zeigt. Anschließend aktualisiert er den Zeiger auf das Ende der Liste, sodass auch dieser auf das neue Element zeigt. Die neue Nachricht ist jetzt eingehängt und die Liste hat wieder einen konsistenten Zustand. Der Thread verlässt

den kritischen Abschnitt, in dem er den Zähler für das aktuelle Ticket erhöht.

Das Empfangen einer Nachricht verläuft analog. Der Thread versucht in den kritischen Abschnitt einzutreten. Dort entfernt er dann das erste Element aus der Liste, indem er den `head`-Zeiger auf das erste Element der Liste um ein Element weiter rückt. Anschließend kann er das aktuelle Ticket erhöhen und die Daten aus dem Element kopieren. Zuletzt gibt er den Speicher des aus der Liste entfernten Elements wieder frei.

Die Überlegungen, wie man diesen Ablauf noch optimieren könnte, führten zu der Hypothese, dass es schneller ist ein bestehendes Element aus einer Liste zu entfernen (oder einzufügen), also ein neues zu allokieren (oder freizugeben).

Diese Hypothese wurde durch das Vorgehen in [7] (Abschnitt 3.2 Memory Management) inspiriert. In diesem Artikel wird davon ausgegangen, dass die Systemaufrufe, die nötig sind, um Speicher zu allokieren oder freizugeben, einen Flaschenhals im Programm darstellen können, der umgangen werden kann, indem man schon allokierte Speicherbereiche wiederverwendet.

Es wird deshalb versucht eine Laufzeitverbesserung zu erzielen, indem „gebrauchte“ Elemente, die aus dem Puffer entfernt wurden, nach dem Kopieren der Daten, nicht freigegeben werden, sondern in einer zweiten Liste, der sogenannten *Freelist* verwaltet werden. Aus dieser Liste kann der nächste Sender das Element dann wieder entnehmen und erneut in den Puffer einhängen, ohne ein neues Element allokieren zu müssen.

Listing 3.4 zeigt den relevanten Teil der `send()`-Methode, ohne die Optimierung durch die Freelist. In Zeile 2 Wird ein neues Element angelegt. Anschließend wird in Zeile 5 die Nachricht in dieses Element kopiert. Die Zeilen 8 bis 12 stellen das Ticketlock dar. In Zeile 14 wird das neue Element mit `insert()` in die Liste eingehängt.

3.5 Parallel Access (PA) Channel

Wie schon mehrfach erwähnt, verwenden alle Channel zwei Locks, denn so wird ermöglicht, dass ein Sender und ein Empfänger gleichzeitig auf den Puffer zugreifen können. Mit nur einem Lock für den ganzen Puffer wäre das nicht möglich. Da es aber kein Problem darstellt, wenn zwei Threads auf verschiedene Variablen und getrennte Bereiche des Puffers zugreifen, ist es wesentlich besser die Bereiche, die man sperrt, möglichst klein zuhalten.

Dieser Channel ist ein eigener experimenteller Versuch diese Herangehensweise weiter auszureizen. Dazu wurden zwei Ideen kombiniert. Zum einen wurde die Idee aus Abschnitt 3.4.1 wieder aufgegriffen. Dort wurde eine einfach verkettete Liste verwendet, deren Elemente man vor dem Zugriff auf den Puffer vorbereiten konnte, um in der Zeit, in der ein Thread seine Nachricht kopiert, kein Lock zu halten. Nun soll auch hier versucht werden, bei dem Schreibzugriff auf das Array, kein Lock halten zu müssen, das den Zugriff auf `head` oder `tail` blockiert. Zum anderen ist es wichtig auf die Granularität zu achten, wenn Locks zur Synchronisation verwendet werden [17]. Mit Granularität beschreibt man, wie groß die Bereiche im Code sind, die ein einzelnes Lock sperrt. Man spricht von einer *groben Granularität*, wenn mehrere Rechenschritte, von denen eventuell

```
1     // Allokieren eines neuen Listenelements
2     message = node_alloc(size);
3
4     // Die Daten werden in das neue Element kopiert
5     memcpy(message->value, data, size);
6
7     // Ticketlock zur Threadkoodination
8     my_ticket = atomic_fetch_add(next_buffer_tail, 1);
9
10    while (my_ticket != tlll->current_buffer_tail) {
11      pthread_yield();
12    }
13    // Einfügen des Elements in den Puffer
14    insert(tlll->buffer, message);
15
16    // Freigabe des Locks
17    tlll->current_buffer_tail++;
```

Listing 3.4: Ausschnitt der `send()`-Funktion des TL LL Channels

nicht alle durch ein Lock geschützt werden müssten, in einem von einem Lock gesperrten Abschnitt liegen.

Mit einem Lock immer ganze Datenstrukturen zu blockieren, und so nur einem Thread auf einmal zu erlauben auf die Datenstruktur zuzugreifen, ist ein Beispiel für eine grobe Granularität. Feine Granularität ist es, wenn kleinere Teile der Datenstruktur gesperrt werden können, und so mehrere Threads gleichzeitig auf die Datenstruktur zugreifen können. Eine grobe Granularität ist leichter zu programmieren, aber nicht so effizient wie eine feine. [17]

Je weniger Rechenschritte zwischen dem Sperren und der Freigabe eines Locks liegen und je weniger Variablen ein einzelnes Lock sperrt, desto feiner wird die Granularität.

In diesem Channel werden also die Bereiche, die ein Lock sperrt, so klein gehalten, dass jedes Feld des Arrays ein eigenes Lock bekommt, das jeweils anzeigt ob in diesem Feld gerade geschrieben oder gelesen wird.

Dieser Channel hat keine Locks für `head` und `tail`. In den anderen Channeln haben `head` und `tail` die Funktion anzuzeigen wo im Array die erste Nachricht liegt, beziehungsweise wo das erste freie Feld ist, weshalb sie nicht aktualisiert werden können, bevor Schreib- oder Lesezugriffe abgeschlossen sind und deshalb mit Locks geschützt werden müssen. Hier werden sie aber so verwendet, dass sie auf das nächste Feld zeigen in dem ein Schreib oder Lesezugriff *begonnen* werden kann. Es reicht also `head` und `tail` als einfache Zähler zu behandeln. Sie werden ähnlich wie die Tickets des Ticketlocks atomar gelesen und erhöht. Allerdings kann man `head` und `tail` nicht einfach immer nur um eins erhöhen, denn es werden zyklische Arrays verwendet. Daher muss ein Thread, der auf den Puffer zugreifen will, zunächst den aktuellen Wert in `head` (oder `tail`) lesen.

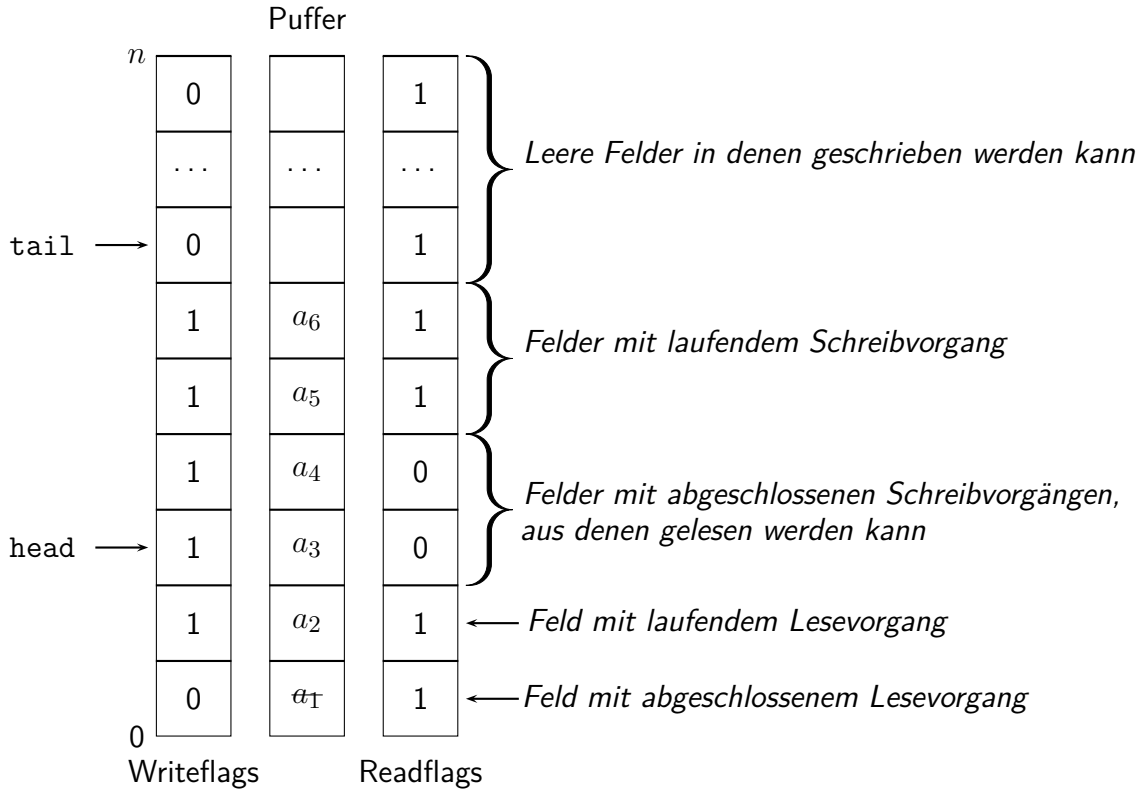

Abbildung 3.1: Aufbau des PA Channels

Dieser wird später als Index für den Arrayzugriff verwendet. Anschließend berechnet der Thread den nächsten Wert für `head` und berücksichtigt dabei, dass es sich um ein zyklisches Array handelt. Nun versucht er den Zähler mit einem Compare-and-swap Aufruf zu aktualisieren. Compare-and-swap wird deshalb verwendet, weil so geprüft werden kann, ob `head` seit dem letzten Zugriff geändert wurde. Wenn `head` beim Operationsaufruf von Compare-and-swap immer noch den gleichen Wert hat, den der Thread zu Beginn gelesen hat, wird `head` mit dem neuen Wert aktualisiert.

Schlägt Compare-and-swap fehl, bedeutet das, dass `head` in der Zwischenzeit, von einem anderen Thread geändert wurde. Ist das der Fall muss der Thread noch einmal von neuem beginnen, in dem er wieder den aktuellen Wert von `head` ließt, den neuen Wert berechnet und erneut versucht `head` mit Compare-and-swap zu aktualisieren.

Ohne die Locks für `head` und `tail` muss auf eine andere Art sichergestellt werden, dass nie zwei Threads auf denselben Bereich im Puffer zugreifen. Dazu besitzt der Channel zwei weitere Arrays mit Flags. Diese Arrays haben die gleiche Länge wie der Puffer. So sind die Flags den Feldern im Puffer direkt über den Index zugeordnet. Die Flags des ersten Flagarrays, die *Writeflags*, signalisieren in welchen Bereichen im Puffer gerade geschrieben werden kann. Die Flags des zweiten Flagarrays, die *Readflags*, signalisieren in welchen Bereichen gerade gelesen werden kann.

Abbildung 3.1 veranschaulicht die Arbeitsweise der Flagarrays. Wenn eine Flag gesetzt

```
1  do {
2      // Vorbereitung des Compare-and-swap Aufrufs
3      expected = lfp->tail;
4      new_tail = INC(expected, lfp->size);
5
6      if (IS_FULL(lfp))
7        return false;
8  // Update von tail mit Compare-and-swap
9  } while (!atomic_compare_exchange_weak(tail, &expected,
10                                        new_tail));
11
12 // Warten auf die Writeflag (vgl. Spinlock)
13 while(!atomic_flag_test_and_set(&write_flags[expected])){};
14
15 // Puffer Zugriff
16 memcpy(lfp->buffer + expected * lfp->itemsize, data, size);
17
18 // Freigabe der Readflag
19 atomic_flag_clear((volatile void *)&read_flags[expected]);
```

Listing 3.5: Ausschnitt der `send()`-Funktion des PA Channels

ist, in der Abbildung mit „1“ dargestellt, bedeutet das, dass auf das entsprechende Feld des Puffers gerade nicht zugegriffen werden kann. Gründe dafür können sein, dass bereits ein anderer Thread darauf zugreift, eine Nachricht im Puffer enthalten ist, die noch nicht gelesen wurde und deshalb nicht überschrieben werden darf, oder das der Puffer noch keine Nachricht enthält und deshalb nicht gelesen werden darf. Ist eine Flag nicht gesetzt („0“), dann bedeutet das, dass auf das entsprechende Feld zugegriffen werden kann. Writeflags signalisieren dabei einen möglichen Schreibzugriff, Readflags einen möglichen Lesezugriff. Die oberen und das unterste Feld in Abbildung 3.1 zeigen derzeit nicht benutzte Felder. Auf sie darf schreibend zugegriffen werden, weshalb die Writeflags den Wert 0 haben. Der `tail`-Zeiger verweist dabei auf das erste freie Feld. Wenn beide Flags gesetzt sind, bedeutet das, dass im Moment ein Thread auf dieses Feld zugreift. Liegt das Feld dabei zwischen `tail` und `head`, ist es ein Schreibzugriff, liegt das Feld hinter `head` handelt es sich um einen laufenden Lesevorgang. Die Felder mit dem Inhalt a_3 und a_4 enthalten Nachrichten, die noch gelesen werden müssen. Das kann man an den gesetzten Writeflags, die einen Schreibzugriff verhindern, und den nicht gesetzten Readflags, die signalisieren, dass aus diesen Feldern gelesen werden kann, erkennen.

Der Ablauf für das Senden einer Nachricht beginnt damit, dass der Thread wie gerade beschrieben versucht den `tail` Zähler um eins zu erhöhen. Dabei speichert er den alten Wert und verwendet ihn im weiteren Verlauf als Index, um auf die entsprechenden Bereiche der Flagarrays und des Puffers zuzugreifen. Das geschieht in Listing 3.5 in den Zeilen 3 bis 9. Zunächst speichert der Thread in Zeile 3 den aktuellen

Wert von `tail`. Danach berechnet er mithilfe von `INC()` den nächsten Wert, und versucht anschließend in Zeile 9 `tail` mit dem neuen Wert zu aktualisieren. Die Funktion `atomic_comapare_exchange_weak()` ist dabei eine der Compare-and-swap Operationen, die in `stdatomic.h` enthalten sind. Nun wartet der Thread in Zeile 13 auf die entsprechende Writeflag. Die Writeflag hat die gleiche Funktionsweise wie das Spinlock aus Abschnitt 3.3. Der Thread versucht in einer Whileschleife, mithilfe von Test-and-set, die Flag zu setzen. In dieser Zeit können andere Threads schon wieder auf `head` zugreifen und selbst auf ihre eigene Writeflag warten.

Die Writeflags wurden vor der ersten Verwendung des Channels als ungesperrt initialsiert. Während der Ausführung des Programms werden die Writeflags immer von Empfängerthreads freigegeben, die einen Lesevorgang abgeschlossen haben.

Allerdings tritt beim Warten auf die Writeflag nicht das typische Problem des Busywaitings auf, denn jeder Thread wartet auf eine eigene Flag. Die Flag kann also im Cache gehalten werden und es kommt zu keinem Flaschenhals im Speicher. Dazu wurde der Flagarray so programmiert, dass jede Flag in einer eigenen Cachezeile liegt, denn sonst würden Veränderungen an einer Flag wieder ein Cacheupdate für alle Threads, die auf in derselben Zeile liegenden Flags warten, auslösen.

Sobald der Thread die Writeflag setzen konnte, beginnt er damit seine Daten in den Puffer zu schreiben, ohne dabei andere Bereiche zu sperren, sodass mehrere Threads gleichzeitig in verschiedene Bereiche schreiben können. Nachdem alle Daten in den Puffer geschrieben wurden, gibt der Thread nun die entsprechende Readflag frei

Die Readflags wurden zu Beginn als gesperrt initialisiert und werden immer von Threads freigegeben, die einen Schreibzugriff beendet haben. Ein Empfänger-Thread kann nun beginnen die Nachricht zu lesen. Der Sendevorgang ist abgeschlossen.

Bei der MPSC Version des Channels entfällt für Sender-Threads das Warten auf die Writeflag. Da es hier nur einen Empfänger-Thread gibt, muss dieser `head` nicht für den nächsten Thread erhöhen, bevor er aus dem Puffer liest. Deshalb wird `head` hier erst erhöht, nachdem der Lesevorgang abgeschlossen ist, und zeigt wie in den anderen Channeln auf das erste leere Feld im Array. Sender-Threads erkennen dann nicht mehr anhand von Writeflags, ob sie schreiben können, sondern daran ob `head` und `tail` verschiedene Werte haben. Für den Empfänger-Thread entfällt in der MPSC Version der atomare Zugriff auf `head` über die Compare-and-swap Methode, denn er konkurriert mit keinem Thread um `head`.

Auch die ungepufferte Version des PA Channels hat eine Besonderheit. Ein Sender-Thread versucht `head` mit einem Compare-and-swap Aufruf auf 1 zusetzen, falls dieser davor 0 war. Ist er bereits 1 bedeutet das, dass der Puffer bereits belegt ist und der Thread bricht den Sendevorgang ab. Nach einem erfolgreichen Compare-and-swap Aufruf schreibt er seine Nachricht in den Puffer und signalisiert den Abschluss des Schreibvorgangs in dem er `tail` auf 1 setzt.

```
1  function insert(L: pointer to list, element: list element)
2    done := false;
3    repeat // Wiederhole, bis das Element eingefügt ist
4      ltail := L->tail; // Lese den aktuellen tail-Zeiger
5      lnext := ltail->next; // Lese den next-Zeiger von tail
6      // Prüfe ob sich tail geändert hat
7      if ltail = L->tail then
8        // Ist tail immernoch das letzte Element?
9        if lnext = NULL then
10         // Versuche das neue Element einzuhängen
11         if CAS(&ltail->next, lnext, element) then
12           done := true; // Element ist eingehängt
13         fi
14       else
15         // Aktualisiere L->tail (für einen anderen Thread)
16         CAS(&L->tail, ltail, lnext);
17       fi;
18     fi;
19   until done = true;
20   // Versuche L->tail zu aktualisieren
21   CAS(&L->tail, ltail, element);
22 end-function
```

Listing 3.6: Die nicht blockierende `insert()`-Funktion (vgl. [17])

Empfänger-Threads warten in einem Spinlock, mit Compare-and-swap, darauf, dass sie `tail` von 1 auf 0 setzten können. Anschließend lesen sie die Nachricht aus dem Puffer und signalisieren den abgeschlossenen Lesezugriff, indem sie `head` wieder auf 0 setzen, sodass der nächste Sender auf den Puffer zugreifen kann.

3.6 Lockfree Linked List (LF LL) Channel

Dieser Channel basiert auf dem „Non-blocking queue algorithm“ von Michael und Scott, der in [17, S. 171-175] vorgestellt wird. Der Puffer ist als einfach verkettete Liste implementiert, die nach dem Vorbild des vorliegenden Algorithmus keine Locks für die Zugriffskoordination verwendet.

Das Einfügen eines neuen Elements benötigt zwei erfolgreiche Compare-and-swap Operationen, mit denen die Zeiger aktualisiert werden. Listing 3.6 zeigt die Funktion zum Einfügen eines neuen Elements in die Liste. Für eine bessere Lesbarkeit wurde Pseudocode für die Darstellung gewählt. Der Code im Listing unterscheidet sich von der tatsächlichen Implementierung, da aufgrund der Syntax von C und der Verwendung von `stdatomic.h` einige zusätzliche Zwischenschritte nötig waren. Das Vorgehen an sich ist aber identisch. Die erste Compare-and-swap Operation in Zeile 11 hängt das neue Ele-

```
1  function get(L: pointer to List): return value
2    done := false;
3    repeat // Wiederhole, bis das Element ausgehängt ist
4      lhead := L->head; // Lese den aktuellen head-Zeiger
5      ltail := L->tail; // Lese den aktuellen tail-Zeiger
6      lnext := lhead->next; // Lese den next-Zeiger von head
7      // Prüfe ob sich head geändert hat
8      if lhead = L->head then
9        // Ist die Liste leer, oder tail nicht aktuell?
10       if lhead = ltail then
11         if lnext = NULL then // Die Liste ist leer
12           return(NULL);
13         fi
14         // Aktualisiere tail (für einen anderen Thread)
15         CAS(&L->tail, ltail, lnext);
16       else
17         // Speichere den Rückgabewert
18         lvalue := lnext->value;
19         // Versuche head zu aktualisieren
20         if CAS(&L->head, lhead, lnext) then
21           done := true; // Element wurde ausgehängt
22         fi
23       fi
24     fi
25   until done = true;
26   free(lhead); // Füge den alten Head zur Freelist hinzu
27   return(lvalue);
28 end-function
```

Listing 3.7: Die nicht blockierende `get()`-Funktion (vgl. [17])

ment in die Liste ein, indem sie den `next`-Zeiger des aktuell letzten Elements der Liste auf das neue Element zeigen lässt. Sollte diese Compare-and-swap Operation fehlschlagen wird sie solange in der While-Schleife von Zeile 3 bis 19 wiederholt bis sie erfolgreich ist. Um das Element in die Liste einzuhängen wird zunächst die Adresse des letzten Elements, und die Adresse des darauffolgenden Elements `next` geladen (Zeile 4 und 5). Erwartungsgemäß sollte `next` `NULL` sein wenn das Element auf das `tail` zeigt wirklich das letzte Element in der Liste ist, da aber keine Locks verwendet werden kann nicht davon ausgegangen werden, dass der Thread die Liste in einem konsistenten Zustand vorfindet. Deshalb wird in den Zeilen 7 und 9 geprüft ob der Wert in Zeile 4 geladen wurde immer noch aktuell ist und ob es sich tatsächlich um das letzte Element der Liste handelt. Das ist nötig, da der Thread jeder Zeit schlafen gelegt werden könnte. In dieser Zeit ist es möglich das andere Threads Elemente erfolgreich in die Liste einhängen

und so die Werte die in Zeile 4 und 5 gelesen wurden nicht mehr aktuell sind. Wenn die Liste in einem konsistenten Zustand ist und die gelesenen Werte nicht aktuell sind wird nun in Zeile 9 versucht den `next`-Zeiger zu aktualisieren. Diese Compare-and-swap Operation prüft dabei nochmal, ob `next` `NULL` ist und es sich tatsächlich um das letzte Element handelt. Wenn die Operation erfolgreich ist wird die Schleife verlassen, und mit der zweiten Compare-and-swap Operation in Zeile 21 fortgefahren. Mit dieser Operation wird `tail` aktualisiert, da dieser noch auf das falsche Element zeigt. Sollte die zweite Compare-and-swap Operation fehlschlagen, wird sie nicht wiederholt, denn das bedeutet, dass ein anderer Thread bereits ausgeholfen und die Aktualisierung von `tail` stellvertretend durchgeführt hat, als er selbst auf einen inkonsistenten Zustand der Liste gestoßen ist. Dieses Aushelfen geschieht in Zeile 16, nachdem in Zeile 9 festgestellt wurde, das `tail` nicht auf das letzte Element der Liste zeigt. Dort wird mit einer Compare-and-Swap Operation versucht `tail` zu aktualisieren. Dieses gegenseitige Aushelfen unter den Threads ist das Schlüsselelement damit der Algorithmus nicht-blockierend ist. Da ein Thread nach Abschluss der ersten Compare-and-swap Operation blockieren könnte, muss jeder Thread damit rechnen auf eine Liste mit unvollständig eingefügtem Element zu treffen, und das Einfügens selbst abschließen zu müssen. [17]

Listing 3.7 zeigt die Funktion für das Entfernen eines Elements aus der Liste. Der Code ist ebenfalls in Pseudocode gegeben und unterscheidet sich auch hier teilweise von der tatsächlichen Implementierung. Das Entfernen eines Elements aus der Liste benötigt nur eine erfolgreiche Compare-and-swap Operation. Mit dieser wird der `head`-Zeiger aktualisiert, sodass dieser auf das nachfolgende Element des aktuell ersten Elements in der Liste, zeigt. [17]

Dazu werden zunächst die Werte von `head` und dessen `next`-Zeiger gelesen. Zusätzlich wird auch `tail` gelesen, denn in Zeile 10 und 11 muss geprüft werden, ob es ich bei diesem Element um das letzte Element in der Liste handelt. Zeigen `head` und `tail` auf das gleiche Element und zeigt dieses Element auf kein weiteres, befindet sich nur noch ein Element in der Liste. Dieses Element wird als Dummyelement verwendet und soll nicht aus der Liste entfernt werden, da sich der Algorithmus sehr viel einfacher umsetzten lässt, wenn sich immer mindestens ein Element in der Liste befindet und so die Ausnahmefälle für leere Listen nicht beachtet werden müssen. Zeigen `head` und `tail` auf das gleiche Element, aber dessen `next`-Zeiger ist nicht `NULL`, bedeutet das, dass die Liste in einem inkonsistenten Zustand ist und `tail` aktualisiert werden muss. Das wird in Zeile 15 versucht. Wenn `head` und `tail` aber nicht auf das gleiche Element zeigen, kann der Thread, in Zeile 20, versuchen den `head`-Zeiger ein Element weiter zurücken. Damit ist das Element aus der Liste entfernt und die zuvor aus dem Element gelesenen Nachricht (Zeile 18) kann nun zurückgegeben werden.

In der MPSC Version ist keine Compare-and-swap Operation nötig, da nur ein Thread auf `head` zugreift.

Wie schon in Abschnitt 3.4.1, wird auch hier versucht mit einer Freelist, in der bereits allokierte, aber nicht genutzte Listenelemente verwaltet werden, Systemaufrufe zum Reservieren von Speicherplatz zu reduzieren und so eine bessere Laufzeit zu erzielen.

4 Experimentdetails

Um die Effizienz der einzelnen Channel zu messen, wurde sich für einen Stresstest entschieden. Dabei versucht eine gegebene Anzahl an Sender-Threads, so schnell wie möglich, eine gegebene Anzahl von Nachrichten über den Channel zu versenden, ohne dass zwischen den einzelnen Zugriffen auf den Channel andere Berechnungen stattfinden. So lässt sich der maximal mögliche Datendurchsatz der Channel messen. Die Gesamtanzahl der Nachrichten wird dazu gleichmäßig auf alle Threads aufgeteilt.

Neben der verwendeten Channelversion t gibt es weitere fünf veränderliche Größen. Die Anzahl der Sender-Threads p, die Anzahl der Empfänger-Threads c, die Größe der einzelnen Nachrichten s, die Gesamtanzahl zu versendender Nachrichten n, sowie die Puffergröße b. Für die beiden Channel, die eine einfach verkettete Liste als Puffer verwenden ist, die Größe des Puffers unbegrenzt.

Die Zeitmessung wird begonnen, nachdem der letzte Thread erstellt wurde. Bis dahin warten alle Threads und beginnen noch nicht mit dem Senden. So wird vermieden, dass die Zeit, die für das Starten der Threads benötigt wird, fälschlicherweise zu der Zeit, die benötigt wird, um die Nachrichten zu senden gezählt wird, was die Messergebnisse verfälschen würde. Außerdem würden sich sonst auch die Zugriffe auf den Channel zeitlich entzerren, da die Threads sequentiell gestartet werden. Da aber gemessen werden soll, wie sich die Effizienz bei einer bestimmten Anzahl von Threads verhält, die *gleichzeitig* versuchen Nachrichten über den Channel zu versenden und zu empfangen, gilt es das zu vermeiden.

Der Datendurchsatz d in Byte pro Sekunde, berechnet sich aus der Gesamtanzahl versendeter Nachrichten n, der Größe der einzelnen Nachrichten in Byte s und der gemessenen Laufzeit Δt in Sekunden.

$$d = \frac{n \cdot s}{\Delta t}$$

Aufgrund der vielen veränderlichen Größen ist es zu zeitaufwendig alle möglichen Kombinationen zu testen. Stattdessen wurde sich auf drei Bereiche verstärkt konzentriert. Dazu gehört zum einen die Skalierbarkeit bei steigender Anzahl der beteiligten Threads, der Einfluss der Nachrichtengröße und die Skalierbarkeit bei steigender Nachrichtenanzahl und steigender Puffergröße. Um den Einfluss der Anzahl der beteiligten Threads auf die Laufzeit, und den Datendurchsatz zu messen, wurde bei MPSC Channeln p schrittweise bis 64 erhöht, bei MPMC Channeln p und c jeweils bis 32. Dabei wurde jede Kombination aus 32x32 gemessen. Die Nachrichtengröße wurde von vier Byte bis vier Kilobyte variiert. Bei der Anzahl der versendeten Nachrichten und der Größe des Puffers wurde ebenfalls das Verhältnis beider Größen betrachtet. Dabei wurden ungepufferte Channel mit $b = 0$, bis hin zu Channeln mit $b > n$ gemessen. In Kapitel 5 werden die relevanten Ergebnisse dieser Messungen diskutiert.

Die Messungen wurden auf baugleichen Knoten eines Clustersystems durchgeführt. Auf diesen Knoten wurde ein Intel Xeon E5-2630 CPU mit 16 physikalischen Kernen

verwendet. Die Kerne unterstützen Multithreading mit zwei Threads pro Kern, bei einer Taktfrequenz von bis zu 2,4 GHz. Die CPU besitzt 4 Caches: L1i (32 KB), L1d (32 KB), L2 (256 KB) und L3 (20480 KB).

5 Auswertung der Messdaten

Im Laufe der Arbeit wurden einige Thesen und Überlegungen, über den Einfluss der verschiedenen Locks und Vorgehensweisen auf die Effizienz der Channel bei der Datenübertragung geäußert. Diese sollen nun durch Messungen überprüft werden.

5.1 Die MPSC Channel

Zunächst soll ein grober Überblick über das gesamte Ergebnis gewonnen werden. Dazu zeigen Abbildung 5.1 und 5.2 die Laufzeiten und die Übetragungsraten aller MPSC Channel im direkten Vergleich. Auf die einzelnen Channel wird im Laufe dieses Kapitels noch genauer eingegangen. Man kann aber bereits feststellen, dass das Ticketlock eindeutig am schlechtesten abgeschnitten hat. Im Gegensatz dazu liegen die Laufzeiten der anderen Channel relativ nah beieinander. Besonders gut schneidet das Spinlock ab. Auch der PA Channel und der lock-freie Channel zeigen eine Verbesserung gegenüber dem ursprünglichen Channel.

Bei Locks auf die so oft zugegriffen wird, wie es in diesem Stresstest der Fall ist, ist das beste, was man erwarten kann, dass der Datendurchsatz mit steigender Threadanzahl konstant bleibt [9]. Deshalb ist es nicht verwunderlich, dass sich die Laufzeit, oder der Datendurchsatz der Channel bei steigender Threadanzahl nicht verbessert, wie man es eigentlich erwartet, wenn man ein Programm parallelisiert.

5.2 Vorteil durch die Unterscheidung von MPMC und MPSC Channeln

Jeder Channel wurde in einer MPMC und einer MPSC-Version implementiert. Dadurch, dass für den Empfänger-Thread in der MPSC-Version keine Locks benötigt werden, ist die intuitive Erwartung, dass die Einsparung dieser Operationen eine Verbesserung der Laufzeit mit sich bringen. Um das zu Überprüfen wurden die gemessenen Laufzeiten der MPMC und MPSC Versionen der einzelnen Channeln bei identischen Messparametern miteinander verglichen.

Abbildung 5.3 zeigt die entsprechenden Daten von vier ausgewählten Channeln. Die beiden Channel die ein Spinlock verwenden erfahren, eine Verbesserung. Diese ist aber mit weniger als 0,04 Sekunden Unterschied nur sehr klein. Auch beim Mutexlock Channel lässt sich eine Verbesserung erkennen, diese ist aber kleiner und schwankt stärker als bei den Spinlocks. Die Kurve des MPSC Lock-free linked List Channels schwankt ebenfalls stark, ab 15 Threads lässt sich aber auch hier eine Verbesserung gegenüber der MPMC-Version erkennen.

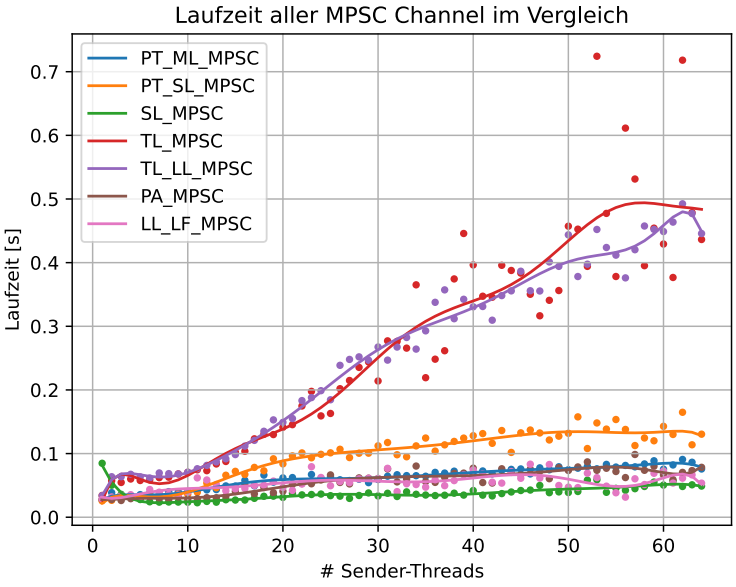

Abbildung 5.1: Die Laufzeit aller MPSC Channel im Vergleich, für $s = 4$, $n = 100.000$, $b = 10$

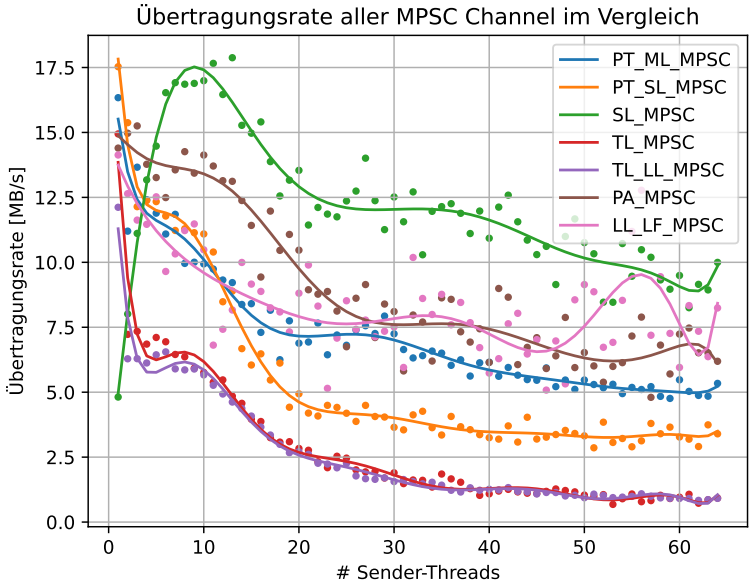

Abbildung 5.2: Die Übertragungsrate aller MPSC Channel im Vergleich, für $s = 4$, $n = 100.000$, $b = 10$

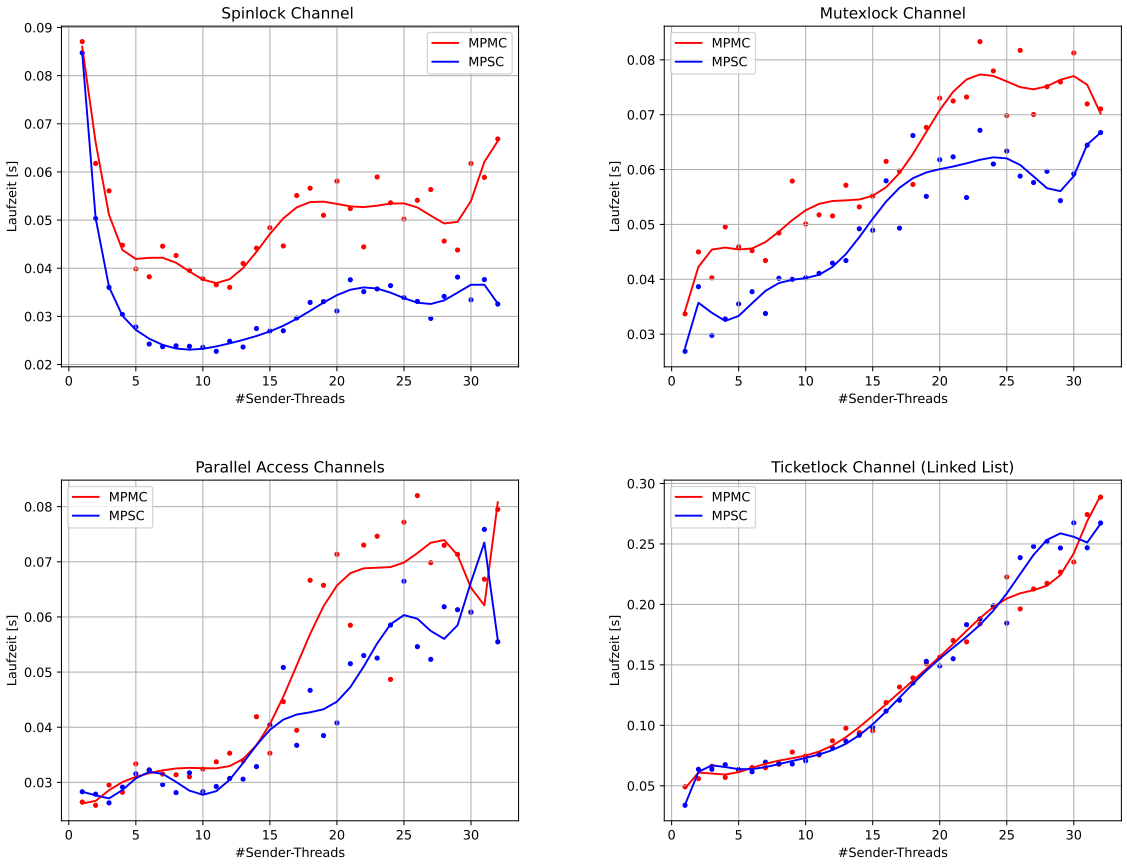

Abbildung 5.3: Laufzeitvergleich für MPMC und MPSC-Versionen ausgewählter Channel, für $c = 1$, $s = 4$, $n = 100.000$, $b = 10$

Bei dem nicht abgebildeten Parallel Access Channel lassen sich nur teilweise Verbesserungen erkennen, zwischen 15 und 30 Sender-Threads erkennen. Bei den Channeln die ein Ticketlock verwenden konnte überhaupt keine Verbesserung festgestellt werden.

Fazit: Der Laufzeitvorteil, der durch die Verwendung eines speziellen MPSC Channels, anstatt eines MPMC Channels erreicht wird, ist mit weniger als 0,04 Sekunden verschwindend gering und tritt nicht bei allen Channeln auf. Während Spinlocks eine klare Verbesserung zeigen, zeigen die Ergebnisse des Mutexlocks, des Parallel Access Channels und des Lock-free Linked List Channels nur teilweise Verbesserungen. Im Falle des Ticketlocks konnte überhaupt keine Verbesserung festgestellt werden.

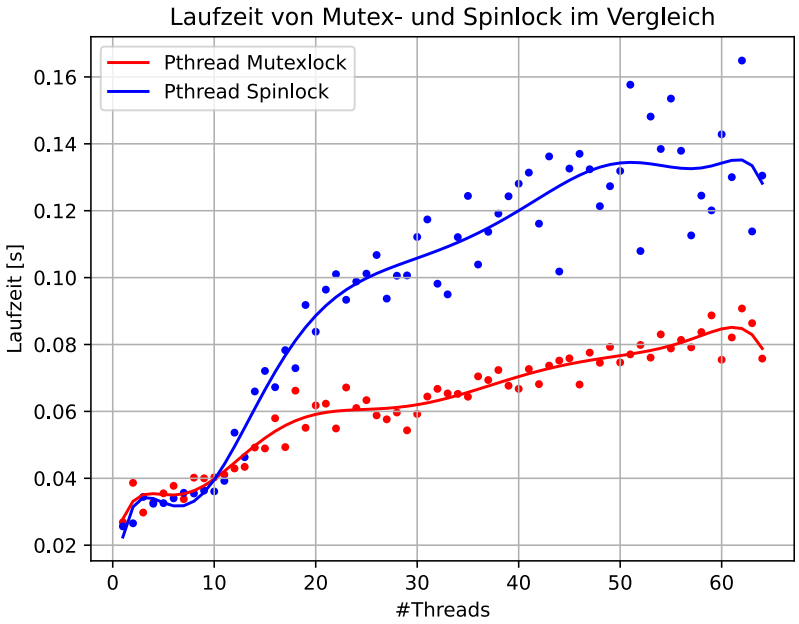

Abbildung 5.4: Laufzeit der MPSC Pthread Channel bei wachsender Anzahl von Sender-Threads, für $s = 4$, $n = 100.000$, $b = 10$

5.3 Vergleich der Pthread Locks

In Abschnitt 3.1 und 3.2 wurden zwei Channel vorgestellt, die beide Synchronisationsoperationen der `pthread.h` Bibliothek verwenden. Nun soll durch einige Messungen festgestellt werden, welche Performanceunterschiede zwischen Mutex- und Spinlock bestehen und welches sich deshalb besser für die Verwendung im Channel eignet. Gemessen wurde dazu wie lange p Sender-Threads brauchen, um 100000 Nachrichten zu je 4 Byte über einen gepufferten Channel zu versenden. Der Puffer fasst dabei 10 Nachrichten.

Abbildung 5.4 zeigt die Messdaten der MPSC Versionen beider Channel. Obwohl die Messewerte des PT SL Channels stark streuen, ist deutlich zu erkennen, dass der Channel, der das Mutexlock verwendet, bei steigender Threadanzahl besser abschneidet. Bis elf Sender-Threads gibt es keine deutlichen Unterschiede zwischen den beiden Channeln. Danach steigt die Laufzeit des PT SL Channels stark an und liegt immer über der Laufzeit des Mutex Channels.

Dieses Verhalten wird sehr wahrscheinlich von der Arbeitsweise der Locks abhängen. Bei der Verwendung von Mutexlocks werden die blockierten Threads schlafen gelegt und anschließend, abhängig von dem verwendeten Schedulingverfahren der verwendeten POSIX Implementierung, wieder aufgeweckt werden. In dieser Zeit erzeugen die Threads keinen Traffic im Speicher. Anders verhält es sich bei dem Spinlock. Hier warten die Threads aktiv in einer Schleife darauf, dass das Lock wieder freigegeben wird. Das ist

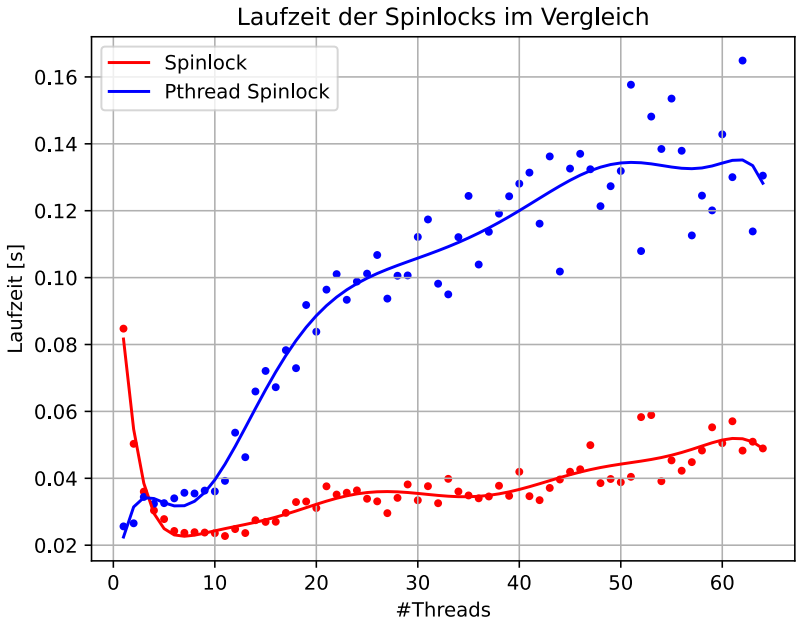

Abbildung 5.5: Laufzeit der MPSC Versionen der Spinlocks bei wachsender Anzahl von Sender-Threads, für $s = 4$, $n = 100.000$, $b = 10$

mit vielen Speicherzugriffen verbunden.

Ab einer Anzahl von etwa elf Threads hat dieser Nachteil anscheinend einen messbaren, negativen Einfluss auf die Effizienz, der sich mit jedem dazukommenden Thread vergrößert.

Fazit: Zwischen den beiden Locks der `pthread.h` Bibliothek gibt es große Unterschiede in Bezug auf die Effizienz. Es ist nicht sinnvoll, das Spinlock für Channel einzusetzen, die von vielen Threads verwendet werden.

5.4 Vergleich der Spinlocks

Der Spinlock Channel aus Abschnitt 3.3 wurde mit der atomaren Test-and-set Operation umgesetzt. Besonders interessant ist es jetzt nachzuprüfen, ob diese einfache Umsetzung irgendwelche Vorteile gegenüber dem POSIX-Spinlock hat. Dazu wurden erneut 100000 Nachrichten zu je 4 Byte über gepufferte Channel versendet und dabei die Anzahl der Sender-Threads nach jedem Durchlauf erhöht.

Abbildung 5.5 zeigt die Messdaten der beiden MPSC Spinlockchannel. Bei einem und zwei Threads schneidet das Test-and-set Spinlock überraschend schlecht ab, erfährt dann aber eine Laufzeitverbesserung. Schon ab fünf Threads liegt die Laufzeit des Test-and-set

Spinlocks deutlich unter dem des POSIX-Spinlocks. Danach steigt die Laufzeit des Test-and-set Spinlocks fast linear und sehr flach, während die Laufzeit des Pthread Spinlocks ab elf beteiligten Sender-Threads wieder sehr stark ansteigt.

Im Vergleich mit den Messdaten des Mutexlocks aus Abbildung 5.4 fällt auf, dass das Test-and-set Spinlock sogar schneller ist als beide POSIX-Locks. Was die These, aus Abschnitt 3.3, dass es besser ist auf atomaren Operationen zur Threadkoordination zurückzugreifen, als auf die Synchronisationsoperationen des POSIX-Standards, weiter stützt.

Darüber warum das Test-and-Set Lock als einziges Lock so untypisch verhält, lässt sich keine gesicherte Aussage machen. Durch wiederholte Messungen, die alle ein ähnliches Ergebnis lieferten, lässt sich aber ein Messfehler ausschließen. Eine mögliche Erklärung ist es, dass der Overhead, der entsteht, wenn immer mehr Threads auf ein Lock zugreifen möchten, bei dem Test-and-set Spinlock sehr klein ist. Dann könnte der Vorteil, den man erhält, wenn die Arbeitslast auf mehrere Threads aufgeteilt wird, groß genug sein, sodass die Laufzeit erst abnimmt und später, wenn das Problem des Busy-waitings größer wird, mit jedem weiteren Thread, wieder ansteigt.

Dafür spricht, dass sich die Laufzeit beim Wechsel von einem auf zwei Sender-Threads fast halbiert, was einer optimalen Parallelisierung sehr nahekommt. Dieser Faktor nimmt aber sehr schnell ab. So verbessert sich die Laufzeit von zwei auf drei Threads nur noch um ein $\frac{1}{5}$, für sechs bis 13 Threads kaum merklich, oder gar nicht und beginnt danach wieder leicht anzusteigen.

Da es sich bei `pthread.h` um eine häufig verwendete Standardbibliothek handelt, liegt die Vermutung, dass die Pthread-Implementierung des Spinlocks für solche Grenzfälle, in denen nur wenige Threads auf das Lock zugreifen, optimiert wurde nahe. Das könnte ein Grund sein, weshalb das POSIX-Spinlock bei der Verwendung von einem, beziehungsweise zwei, Sender-Threads so gut abschneidet. Im Gegensatz dazu wurden an dem Test-and-Set Spinlock keine weiteren Optimierungen vorgenommen. Diese Vermutung kann aber nicht bestätigt werden, da eine genauere Untersuchung der spezifischen Bibliotheksimplementierung nötig wäre.

Fazit: Die Ergebnisse von [6] und [7], die in Abschnitt 3.3 kurz erwähnt wurden, bestätigen sich auch hier. Mit steigender Threadanzahl skaliert das Test-and-set Spinlock besser als das POSIX-Spinlock oder das Mutexlock.

5.5 Der Einfluss des Puffers

In Abschnitt 2.5 wurde, abhängig von der Größe des Puffers, zwischen gepufferten und ungepufferten Channeln unterschieden. Der Unterschied liegt dabei nicht alleine in der Größe des Puffers, sondern auch in der Implementierung wie in Kapitel 3 deutlich wurde.

Die Messungen haben, für alle Channel, bis auf den Ticketlock-Channel, ein eindeutiges Ergebnis. Schon bei einem kleinen Puffer ($b = 10$) sind erhebliche Unterschiede und der

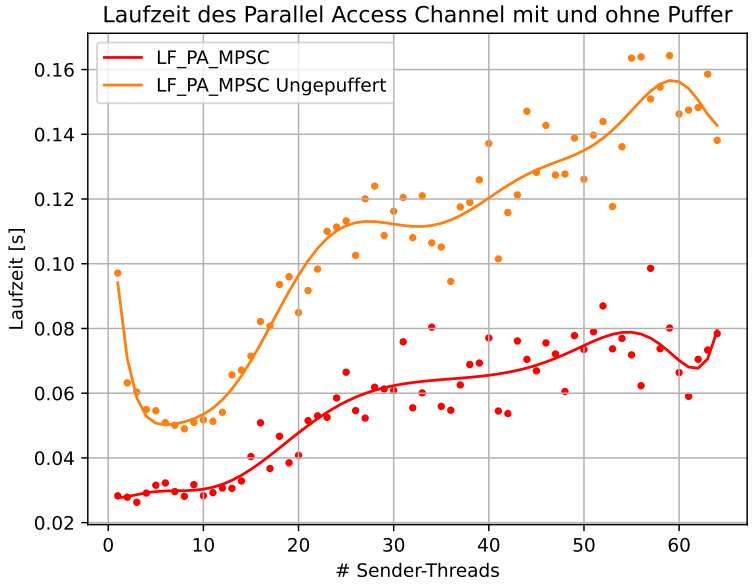

Abbildung 5.6: Vergleich der Laufzeit zwischen einem gepufferten und einem ungepufferten Parallel Access Channel, für $s = 4$, $n = 100.000$, $b = 0$ bzw. $b = 10$

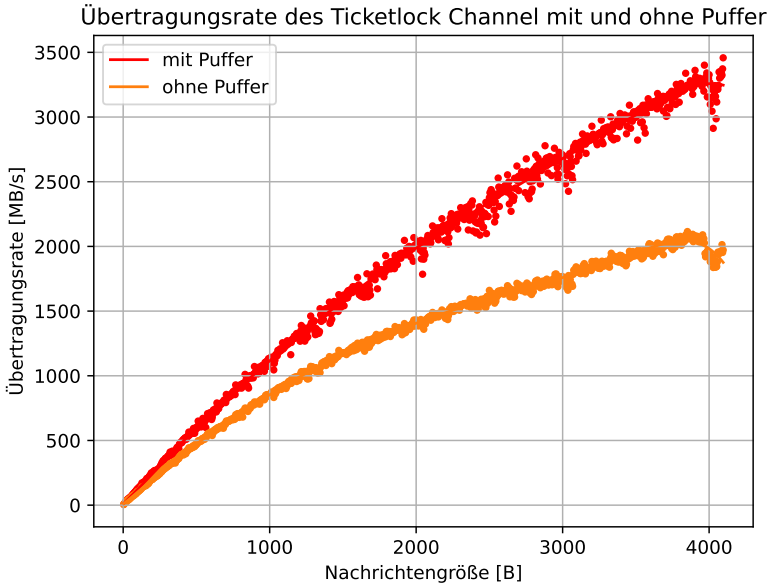

Abbildung 5.7: Vergleich der Übertragungsrate zwischen einem gepufferten und einem ungepufferten MPSC Ticketlock-Channel, für $p = 10$, $n = 100.000$, $b = 0$ bzw. $b = 10$

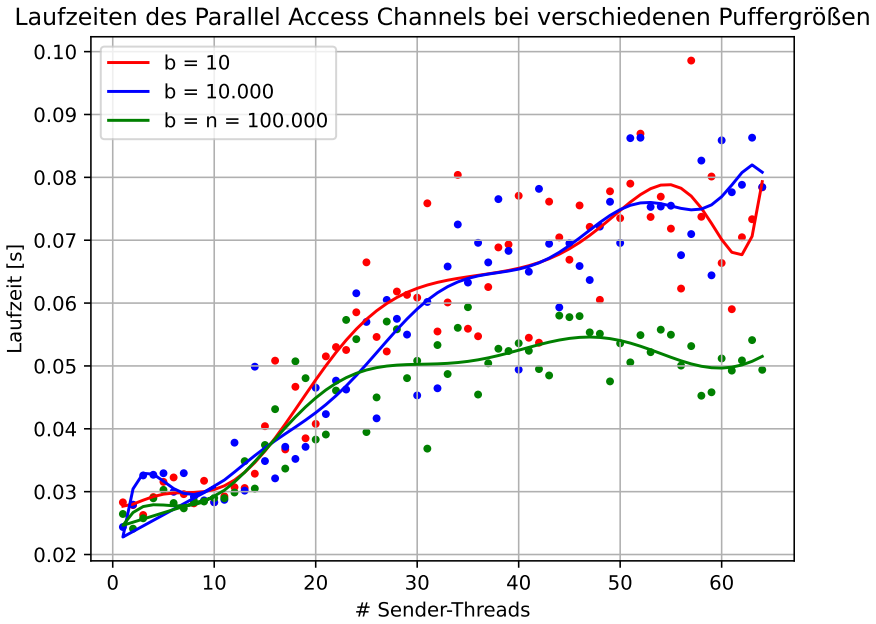

Abbildung 5.8: Vergleich der Laufzeit des MPSC Parallel Access Channel mit verschiedenen Puffergrößen, für $p = 10$, $s = 4$, $n = 100.000$

Laufzeit zu erkennen. Bei der Datenrate bewirkt ein kleiner Puffer ebenfalls eine große Verbesserung Das gilt auch für den Ticketlock-Channel.

Die Ergebnisse sind für alle Channel gleich und unterscheiden sich nur geringfügig im Betrag der Laufzeitersparnis, deshalb werden in Abbildung 5.6 stellvertretend die Ergebnisse für den Parallel Access Channel gezeigt. Der durchgängige Unterschied zwischen gepuffert und ungepuffert ist deutlich zu erkennen.

Abbildung 5.7 zeigt die Laufzeit des Ticketlock-Channels bei steigender Nachrichtengröße. Auch hier ist der durchgängige Unterschied deutlich zu sehen. Die Größe des Puffers ist dabei zweitrangig und hat keinen großen Einfluss auf die Laufzeit. Das lässt sich zum einen daraus ableiten, dass sich für verschiedene Puffergrößen in Abbildung 5.8 erst dann eine Laufzeitverbesserung erkennen lässt, wenn die Größe des Puffers der Anzahl der insgesamt versendeten Nachrichten entspricht. Zum anderen lässt es sich aus dem Vergleich der verschiedenen Puffertypen ableiten. Während bei Channeln die einen Array als Puffer verwenden, die Größe des Puffers begrenzt ist, haben Channel mit einer einfach verketten Liste keine Begrenzung darin, wie viele Elemente der Puffer enthalten kann. In Abbildung 5.9 wird der Ticketlock-Channel der einen Array als Puffer der Größe 10 verwendet, mit dem Ticketlock-Channel mit einer einfach verkettete Liste als Puffer, verglichen. Zwischen diesen beiden Channeln kann kein Unterschied bei der Laufzeit festgestellt werden. Auch hier bringt ein größerer Puffer selbst dann keinen Vorteil, wenn er alle Nachrichten, die insgesamt versendet werden, fassen kann.

Fazit: Es gibt sehr große Unterschiede zwischen einem gepufferten und einem ungepufferten Channel. Es ist wesentlich besser einen Channel mit einem Puffer zu verwenden. Vor allem bei großen Nachrichten steigert ein Puffer die Übertragungsrate um einige MB/s. Die Größe des Puffers hat dabei keinen erkennbaren Einfluss, so lange nicht $b \approx n$ oder $b > n$.

5.6 Vergleich zwischen Spin- und Ticketlock

In Abschnitt 3.4 der Ticketlock Channel zusammen mit der Überlegung, ob die fehlende Fairness und das mögliche Verhungern von Threads ein so großes Problem darstellen, dass die Fairness des Ticketlocks die Nachteile überwiegt.

Die Ergebnisse zeigen aber, dass die in Abschnitt 3.4 erläuterten Nachteile des Ticketlocks deutlich überwiegen. Kein anderer Channel schneidet so schlecht ab wie die beiden Ticketlock Channel. In den Abbildungen 5.1 und 5.2 ist klar zu sehen, wie die beiden Ticketlock Channel deutlich schlechter skalieren.

Fazit: Die Fairness des Ticketlocks bringt keine Vorteile gegen über dem Spinlock. Die Nachteile überwiegen deutlich, sodass das Ticketlock insgesamt am schlechtesten abschneidet (vgl. Abschnitt 5.1).

5.7 Der Vorteil einer einfach verketteten Liste als Puffer

Neben den Algorithmen, die auf den Puffer zugreifen, kann auch der Puffer selbst einen Einfluss auf die Performance des Channels haben. Deshalb wurde versucht die Laufzeit zu optimieren, indem man als Datenstruktur für den Puffer eine einfach verkettete Liste verwendet, die es zulässt die Nachrichten vor dem eigentlichen Zugriff auf den Puffer „vorzubereiten“. Dazu wurde der Ticketlock Channel ausgewählt, und in einer weiteren Version, mit einer verketteten Liste als Puffer, implementiert (vgl. Abschnitt 3.4.1).

In Abbildung 5.9 sind die Daten der beiden Versionen gegeneinander aufgetragen. Wie man erkennen kann, hat die Verwendung einer einfach verketteten Liste nicht zu einer Laufzeitverbesserung beigetragen. Allerdings führte sie auch zu keiner Verschlechterung. Die Nachteile der einfach verketteten Liste, die aufwendigeren Methoden zum korrekten Einfügen und Entfernen von Elementen und die Speicherverwaltung beim Erstellen und Freigeben von Elementen, halten sich mit dem Vorteil, dass der Puffer nur für das Einfügen eines Elements gesperrt werden muss die Waage.

Fazit: Das Verwenden einer einfach verketteten Liste als Puffer hatte keinen Effekt auf die Performance des Channels. In Anbetracht des erheblichen Mehraufwands bei der Implementierung eines Channels mit einer verketteten Liste ist ein einfacher Array die bessere Wahl.

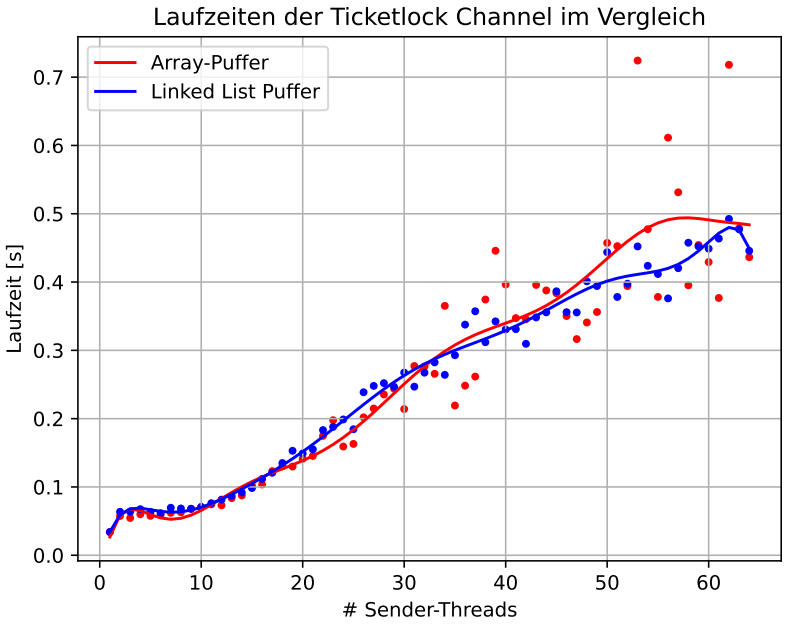

Abbildung 5.9: Die beiden Ticketlock Channel im Vergleich, für $c = 1$ (MPSC), $s = 4$, $n = 100.000$, $b = 10$ bzw. $b = \infty$

5.8 Der Vorteil durch die Freelist

Bei der Verwendung einer einfach verketteten Liste als Puffer kommt es vermehrt zu Systemaufrufen für die Reservierung und Freigabe von Speicher, da die Listenelemente erst bei Bedarf erstellt werden. Im Gegensatz dazu wird ein Array einmalig zu Beginn angelegt, danach sind keine Systemaufrufe mehr nötig. Diese zusätzlichen Systemaufrufe können einen Flaschenhals im Programm darstellen [7], und damit die Effizienz des Channels negativ beeinträchtigen.

Deshalb wurde für die beiden Channel, die eine verkettete Liste als Puffer verwenden eine weitere Liste programmiert. In dieser Liste werden bereits allokierte aber im Moment nicht verwendete Listenelemente verwaltet und können bei Bedarf wiederverwendet werden. Die Anzahl der nötigen Systemaufrufe hängt direkt mir der Anzahl der zu sendenden Nachrichten zusammen, denn für jede Nachricht muss ein Listenelement angelegt werden. Deshalb wurde zunächst der Unterschied zwischen den beiden Versionen bei steigender Nachrichtenanzahl betrachtet. Abbildung 5.10 zeigt die Ergebnisse der beiden Channel. Es wurde jeweils die Laufzeit der Version mit einer Freelist, gegen die ohne Freelist aufgetragen. Besonders auffällig ist, dass im Falle des Ticketlock-Channels überhaupt keine Veränderung erkennbar ist. Die Freelist hat hier keinen Einfluss auf die Laufzeit. Das gilt zunächst auch für den Lock-free Linked List Channel. Bis 400.000 Nachrichten gibt es keinen Unterschied zwischen den beiden Versionen. Danach, im Be-

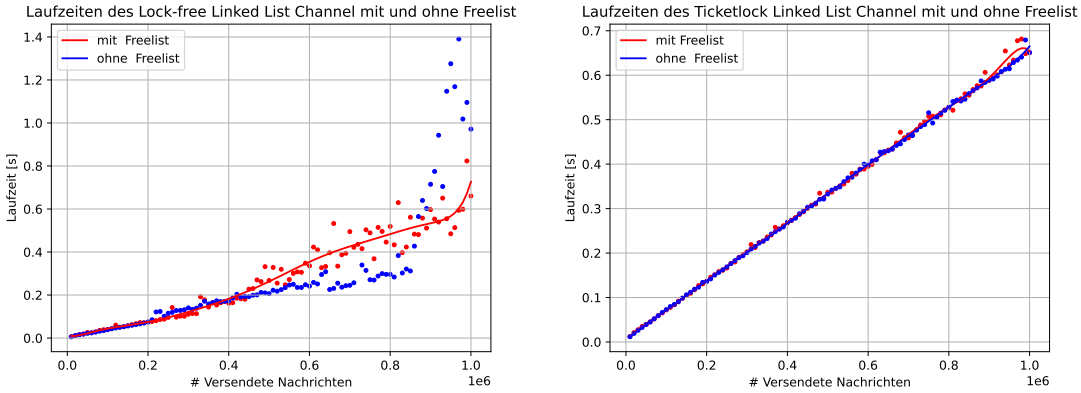

Abbildung 5.10: Die Laufzeit des Linked List Channel, mit und ohne Freelist, in Abhängigkeit der insgesamt versendeten Nachrichten, $n = 10.000$ bis $1.000.000$, $p = 10$, $c = 1$ (MPSC), $s = 4$

reich zwischen 400.000 und 800.000, hat der Channel mit Freelist sogar eine schlechtere Performance. Ab ungefähr 800.000 Nachrichten steigt die Laufzeit des Channels ohne Freelist plötzlich Sprunghaft an. Das kann durch die Verwendung einer Freelist verhindert werden. Der sprunghafte Anstieg ist auf einen Cache-Effekt zurückzuführen. Bei 800.000 Nachrichten zu je 4 Byte ist die größe des L1 Caches mit 32 KByte erreicht. Durch die Freelist müssen nicht für alle 800.000 Nachrichten ein Element allokiert werden. So ist der Cache nicht vollständig mit alten Listenelementen belegt und der Anstieg der Laufzeit bleibt aus. Deshalb hat die Freelist, ab 800.000 Nachrichten, einen Vorteil, nachdem sie bei weniger Nachrichten keinen oder und ab 400.000 Nachrichten einen negativen Effekt auf die Laufzeit hatte.

Abbildung 5.11 zeigt die Laufzeiten der beiden Channel bei steigender Threadanzahl. Im Falle des Ticketlock Channels sind die beiden Messkurven erneut fast deckungsgleich. Ab ungefähr 45 beteiligten Sender-Threads zeigt sich eine minimale Verbesserung, wenn eine Freelist verwendet wird.

Die Messdaten für den Lock-free Linked List Channel streuen sehr stark, was es schwer macht eine eindeutige Aussage aus den Daten abzuleiten. Im Bereich von einem bis 20 Sender-Threads liegen die Messpunkte noch relativ dicht beieinander. Es ist kein Unterschied zwischen den beiden Versionen erkennbar. Im Bereich von 30 bis 64 Threads ist die Streuung zwar sehr stark, aber es lässt sich erkennen, dass einige Messwerte, der Version die keine Freelist verwendet, über dem Streubereich der Version mit Freelist liegen. Bei hohen Threadanzahlen scheint die Freelist einen kleinen Vorteil zu bringen.

Fazit: Die in [7] geäußerten Bedenken, dass häufige Systemaufrufe zur Speicherplatzverwaltung ein Problem darstellen führten zu der These aus Abschnitt 3.4.1, dass es Effizienter ist ungenutzte Listenelemente nicht wieder freizugeben, sondern in einer Freelist

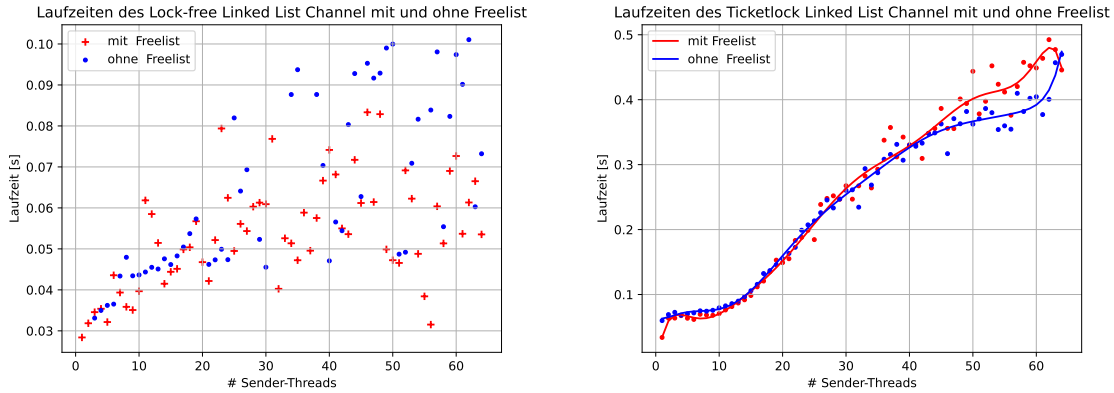

Abbildung 5.11: Die Laufzeit des Linked List Channel, mit und ohne Freelist, in Abhängigkeit der beteiligten Sender-Threads, $c = 1$ (MPSC), $s = 4$, $n = 100.000$

zu verwalten. Das konnte für den Kontext dieser Arbeit jedoch nicht festgestellt werden. Das Vermeiden von Systemaufrufen durch die Wiederverwendung „gebrauchter“ Listenelemente hat, im Falle des Ticketlock-Channels, keinen Einfluss auf die Laufzeit. Für den Lock-free Linked List Channel lässt sich bei hohen Threadanzahlen, beziehungsweise nur bei sehr vielen versendeten Nachrichten eine Verbesserung feststellen.

6 Zusammenfassung und Ausblick

Ziel der Arbeit war es einen ersten Einblick darin zu erhalten, wie die Channel optimiert und so die Effizienz des Laufzeitsystems gesteigert werden kann. Dazu wurden weitere Channelversionen implementiert, die dann mit der ursprünglichen Version des Channels von Dr. Prell verglichen wurden. Dazu wurden verschiedene Locks, die mit atomaren Operationen umgesetzt wurden, verwendet. Anschließend wurde die Skalierbarkeit bei steigender Thread- und Nachrichtenanzahl, sowie bei steigender Nachrichten- und Puffergröße gemessen. Der sehr einfache Test-and-set Spinlock Channel schneidet dabei am besten ab. Der betrachtete Lock-freie Algorithmus schnitt ebenfalls besser ab als der ursprüngliche Mutexlock Channel. Auch der experimentelle Parallel Access Channel, bei dem die Lockgranularität stark verfeinert wurde, indem jedes Arrayfeld eigene Locks erhielt erzielt eine erkennbar bessere Laufzeit und Übertragungsrate. Nur das Ticketlock brachte keine Verbesserung.

In „*Synchronization algorithms and concurrent programming*“ [17] werden einige Variationen des Tickelocks vorgestellt. Diese arbeiten mit der Methode des local-spinnings. Das wäre eine mögliche Antwort auf das massive Problem des busy-waitings, dass als Ursache für das schlechte Abschneiden der Ticketlockchannel vermutet wird. Voraussetzung dafür ist aber, dass die Anzahl der Threads, die den Channel benutzen werden, bei der Kompilierung bekannt sind. Dafür wären große Änderungen an der Channel API nötig, weshalb gegebenenfalls auch deren Verwendung im übrigen Quellcode der Laufzeitumgebung angepasst werden müsste. Diese aufwendige Überarbeitung könnte sich aber durchaus lohnen, da sich so auch für die übrigen Channel ebenfalls die Möglichkeit zur Optimierung, mit local-spinning, bieten würde.

Mit dem lock-free Algorithmus von Michael und Scott konnte eine Verbesserung erzielt werden. Eine genauere Betrachtung speziell von möglichen lock-freien Lösungen könnte daher durchaus interessant sein.

Die Verwendung einer einfach verketteten Liste hat in der aktuellen Umsetzung zu keiner Verbesserung geführt. Es hat aber auch zu keiner Verschlechterung geführt und das, obwohl sich der Einsatz einer einfach verketteten Liste wesentlich komplexer gestaltet, als bei einem Array. Vielleicht ist es möglich durch weitere Optimierungen bei der Verwendung einer einfach verketteten Liste doch noch einen Laufzeitvorteil zu erzielen. Eine weitere Untersuchung könnte auch deshalb sinnvoll sein, da der Vergleich zwischen den verschiedenen Pufferimplementierungen nur beim Ticketlock Channel vorgenommen wurde. Jedoch hat sich leider gerade das Ticketlock als sehr schlechte Lösung herausgestellt. Außerdem ist es auffallend, dass verschiedene Ansätze der Laufzeitoptimierung auf Channel mit einem Ticketlock scheinbar keinen Einfluss hatten. So war bei den Ticketlock-Channeln kein Unterschied zwischen MPMC und MPSC Implementierungen erkennbar. Die Verwendung einer einfach verketten Liste, sowie deren Optimierung durch eine Freelist hatten ebenfalls keinen Effekt und das, obwohl bei anderen Channeln mit diesen Ansätzen durchaus Veränderungen festgestellt werden konnten. Um die einfach verkettete Liste als geeignete Datenstruktur für den Puffer endgültig auszuschließen ist

eine weitere Untersuchung mit anderen Locks nötig.

Der Parallel Access Channel, der in seiner aktuellen Implementierung eher als Prototyp gesehen werden sollte, erzielt trotzdem bessere Ergebnisse als der ursprüngliche Mutexlockchannel. Die Weiterentwicklung dieser Idee und die Reduzierung des Overheads ist daher vielversprechend. Vorstellbar ist die Verwendung von Flags mit mehr als zwei Zuständen, um einen der zusätzlichen Arrays einzusparen. Vielleicht ist es auch möglich durch kleine Änderungen ganz auf die Verwendung von `head` und `tail` zu verzichten. Ob diese Änderungen eine weitere Leistungssteigerung bringen können und welche weiteren Möglichkeiten zur Optimierung bestehen, müsste eine genauere Betrachtung zeigen.

Literatur

[1] Thomas E. Anderson. „The performance of spin lock alternatives for shared-money multiprocessors“. In: *IEEE transactions on parallel and distributed systems*. 1. Ser. (1 Jan. 1990), S. 6–16. ISSN: 1045-9219.

[2] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams und Katherine A. Yelick. *The Landscape of Parallel Computing Research: A View from Berkeley*. Techn. Ber. Electrical Engineering und Computer Sciences University of California at Berkeley, 2006. URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf.

[3] *Atomic operations library*. URL: https://en.cppreference.com/w/c/atomic (besucht am 01. 09. 2021).

[4] *C11 (C standard revision)*. Wikimedia Foundation Inc., 14. Aug. 2021. URL: https://en.wikipedia.org/wiki/C11_(C_standard_revision) (besucht am 07. 09. 2021).

[5] Ralf Hartmut Gütung und Stefan Dieker. *Datenstrukturen und Algorithmen*. Wiesbaden: Springer, 2018. ISBN: 978-3-6580-4676-7.

[6] Ralf Hoffmann, Matthias Korch und Thomas Rauber. „Performance Evaluation of Task Pools Based on Hardware Synchronization“. In: *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*. IEEE, 2004. ISBN: 0-7695-2153-3. DOI: https://doi.org/10.1109/SC.2004.38.

[7] Ralf Hoffmann, Matthias Korch und Thomas Rauber. „Using Hardware Operations to Reduce the Synchronization Overhead of Task Pools“. In: *Proceedings 2004 International Conference on Parallel Processing ICPP*. IEEE, 2004, S. 241–249. ISBN: 0-7695-2197-5. DOI: https://doi.org/10.1109/ICPP.2004.1327927.

[8] Ralf Hoffmann und Thomas Rauber. „Adaptive Task Pools: Efficiently Balancing Large Number of Tasks on Shared-address Spaces“. In: *International journal of parallel programming* 39 (5 2011), S. 553–581. ISSN: 0885-7458.

[9] Michael Klemm und Jim Cownie. *High performance parallel runtimes. Design and implementation*. Berlin: De Gruyter, Boston, 2021. ISBN: 978-3-1106-3268-2.

[10] *Man Page von pthread_mutex_lock()*. die.net. URL: https://linux.die.net/man/3/pthread_mutex_lock (besucht am 31. 08. 2021).

[11] *Man Page von pthread_spin_lock()*. man7.org, März 2021. URL: https://man7.org/linux/man-pages/man3/pthread_spin_lock.3.html (besucht am 31. 08. 2021).

[12] *Man Page von pthread_yield()*. URL: https://man7.org/linux/man-pages/man3/pthread_yield.3.html (besucht am 15. 09. 2021).

[13] John Mellor-Crummey und Michael Scott. „Algorithms for scalable synchronization on shared-memory multiprocessors“. In: *ACM transactions on computer systems* 9 (1 1991), S. 21–65. ISSN: 0734-2071.

[14] *Portable Operating System Interface*. Wikimedia Foundation Inc., Feb. 2020. URL: https://de.wikipedia.org/wiki/Portable_Operating_System_Interface (besucht am 26. 08. 2021).

[15] Andreas Prell. „Embracing Explicit Communication in Work-Stealing Runtime Systems“. Diss. Universität Bayreuth, 2016.

[16] Thomas Rauber und Gudula Rünger. *Parallele Programmierung*. 3. Aufl. Berlin: Springer, 2012. ISBN: 978-3-6421-3604-7.

[17] Gadi Taubenfeld. *Synchronization algorithms and concurrent programming*. Harlow: Pearson/Prentice Hall, 2006. ISBN: 978-0-1319-7259-9.

[18] *Webside der Programmiersprache Go*. URL: https://golang.org/ (besucht am 16. 09. 2021).

[19] *Webside der Programmiersprache Rust*. URL: https://www.rust-lang.org/ (besucht am 16. 09. 2021).

Abbildungsverzeichnis

Listings

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate deutlich kenntlich gemacht zu haben.
Ich erkläre weiterhin, dass die vorliegende Arbeit in gleicher oder ähnlicher Form noch nicht im Rahmen eines anderen Prüfungsverfahrens oder zum Erlangen eines akademischen Grades eingereicht wurde.

Bayreuth, den 20. September 2021 Henrik Laubert