

INSTITUT FÜR INFORMATIK
DER LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

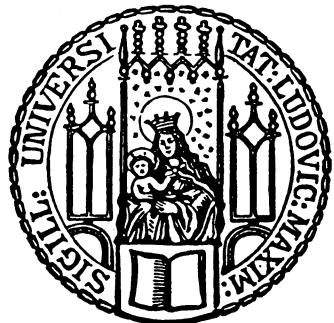


Masterarbeit

Entwicklung leistungsfähiger
RMA-Locks durch Portierung und
Optimierung von NUMA-Algorithmen

Adrian Uffmann

INSTITUT FÜR INFORMATIK
DER LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN



Masterarbeit

Entwicklung leistungsfähiger
RMA-Locks durch Portierung und
Optimierung von NUMA-Algorithmen

Adrian Uffmann

Aufgabensteller: Prof. Dr. Dieter Kranzlmüller

Betreuer: Dang Diep
Dr. Karl Fürlinger

Abgabetermin: 31. Januar 2022

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 31. Januar 2022

.....
(Unterschrift des Kandidaten)

Abstract

Locks sind einer der wichtigsten Grundbausteine für die parallele Programmierung. Trotzdem gibt es kaum Lock-Implementierungen für verteilten Speicher, dafür aber eine Vielzahl an Lock-Algorithmen für gemeinsamen Speicher, die *Non Uniform Memory Access* (NUMA) berücksichtigen. Genau wie bei verteiltem Speicher sind bei NUMA Zugriffe auf entfernten Speicher deutlich langsamer und müssen vermieden werden. Daher wird in dieser Arbeit eine Auswahl solcher Algorithmen auf verteilten Speicher portiert und dort optimiert. Für die Evaluation wird eine Benchmarksuite entwickelt, mit der die Geschwindigkeit, die praktische Fairness und andere algorithmusspezifische Kennzahlen von beliebigen Lock-Implementierungen auf verteiltem Speicher in verschiedenen Szenarien gemessen werden kann. Die Evaluation zeigt, dass durch Portierung und Optimierung eines Cohort-Locks mit globalem MCS-Lock und lokalem Hemlock eine Implementierung entwickelt wurde, die bis zu vier Mal so schnell ist wie der bisher beste Lock auf verteiltem Speicher, der RMA-MCS-Lock. Dabei hat die neue Implementierung einen geringeren Speicherverbrauch und bietet vergleichbare Fairness.

Inhaltsverzeichnis

1 Einleitung	1
1.1 Motivation	1
1.2 Übersicht	1
2 Hinführung zum Thema	3
2.1 Synchronisierung in parallelen Programmen	3
2.1.1 Spin-Locks	5
2.1.2 Fairness	6
2.1.3 MCS-Lock	7
2.2 Programmierung verteilter Systeme	8
2.3 NUMA	10
2.4 Ziel der Arbeit	10
3 Locks in verteilten Systemen	11
3.1 MPI Locks	11
3.2 RMA-MCS	12
3.3 Fortschrittsgarantien von MPI	13
3.4 Nicht-atomare Übergabe von Locks	15
3.5 Punkt-zu-Punkt Übergabe von Locks	17
3.6 Nutzung von gemeinsamem Speicher innerhalb eines Rechenknotens	18
4 Evaluation und Optimierung bestehender Locks	19
4.1 Vergleichskriterien	20
4.1.1 Geschwindigkeit	20
4.1.2 Konkurrenz	20
4.1.3 Fairness	21
4.2 Evaluation bestehender Locks	21
4.2.1 Performance freier Locks (UPB)	22
4.2.2 Leerer kritischer Abschnitt (ECSB)	24
4.2.3 Änderung des kritischen Arbeitsanteils (CCWB)	25
4.2.4 Vor dem Akquirieren warten (WBAB)	29
4.3 Optimierung bestehender Locks	34
4.3.1 Optimierung von D-MCS	34
4.3.2 Optimierung von dash::Mutex	37
5 Portierung von NUMA-Locks auf verteilten Speicher	41
5.1 Delegierende Locks	41
5.2 RH-Lock	42
5.2.1 Portierung und Optimierung des RH-Locks	43
5.2.2 Evaluation des RH-Locks	43

Inhaltsverzeichnis

5.3	HCLH-Lock	48
5.3.1	CLH-Lock	48
5.3.2	CLH-Lock Variante für NUMA	49
5.3.3	Evaluation des CLH-Locks	51
5.4	Cohort-Lock und HMCS-Lock	54
5.4.1	Portierung und Optimierung des Cohort-Locks	55
5.4.2	Evaluation der Optimierungen des Cohort-Locks	59
5.4.3	Evaluation der verschiedenen Cohort-Locks	61
5.4.4	Evaluation von lokalen Warteschlangen-Locks	67
5.5	AHMCS-Lock	70
5.6	CST-Lock	72
5.7	SHFL-Lock	72
5.7.1	Portierung und Optimierung des SHFL-Locks	74
5.7.2	Evaluation des SHFL-Locks	75
6	Fazit	79
	Abbildungsverzeichnis	81
	Benchmarkverzeichnis	82
	Glossar	83
	Akronyme	84
	Literaturverzeichnis	85

1 Einleitung

1.1 Motivation

Locks sind einer der grundlegendsten und verbreitetsten Mechanismen für die Synchronisierung von Speicherzugriffen in parallelen Programmen und es gibt eine Vielzahl von Algorithmen, die auf Locks basieren. Es ist daher nicht verwunderlich, dass immer wieder neue Lock-Algorithmen entwickelt werden, die weitere Verbesserungen in Geschwindigkeit, Speicherverbrauch oder anderen Bereichen erzielen. Die meisten dieser Algorithmen werden für Systeme mit gemeinsamem Speicher entwickelt. Locks können aber auch für die Programmierung in verteilten Systemen, z. B. im Bereich *High Performance Computing* (HPC), nützlich sein [SBH16].

Da in verteilten Systemen die Zugriffe auf entfernten Speicher deutlich langsamer sind als auf lokalen Speicher, müssen entfernte Zugriffe vermieden werden, um eine gute Performance zu erzielen. Locks, die nicht mit diesem Ziel entworfen wurden, eignen sich daher weniger gut für den Einsatz in verteilten Systemen.

Diese Anforderung, Zugriffe auf entfernten Speicher zu vermeiden, gibt es aber nicht nur bei verteilten Systemen, sondern auch bei Systemen mit gemeinsamem Speicher: Manche Prozessoren nutzen Architekturen mit *Non Uniform Memory Access* (NUMA). Hierbei sind Zugriffe auf den Speicher anderer Prozessoren langsamer als Zugriffe auf den eigenen Speicher [Lam13] und müssen entsprechend vermieden werden, um eine gute Performance zu erzielen. Algorithmen, die für NUMA-Architekturen entworfen wurden, könnten sich daher auch für verteilte Systeme eignen.

Im Rahmen dieser Arbeit soll daher untersucht werden, ob durch die Portierung von Locks für NUMA-Architekturen auf verteilten Speicher eine bessere Performance erreicht werden kann als durch Nutzung bestehender Implementierungen für verteilte Systeme.

1.2 Übersicht

In Kapitel 2 werden die notwendigen Grundlagen zu Locks, den Unterschieden zwischen gemeinsamem und verteiltem Speicher und NUMA etabliert. Darauf aufbauend wird das Ziel der Arbeit hergeleitet.

Kapitel 3 zeigt dann, welche Locks bereits auf verteiltem Speicher zur Verfügung stehen und mit welchen Optimierungen diese verbessert werden können. Neben einem Lock basierend auf `MPI_Win_lock` werden dabei drei weitere Locks betrachtet: `dash::Mutex` aus [ZMI⁺14], sowie D-MCS und RMA-MCS aus [SBH16].

Da es bisher kaum Locks für verteilte Systeme gibt, gibt es auch keine Benchmarksuites, mit denen neue Implementierungen detailliert mit Bestehenden verglichen werden könnten. Daher werden in Kapitel 4 die relevanten Kriterien für einen Vergleich zusammengefasst und eine neue Benchmarksuite entwickelt. Mit dieser können die Geschwindigkeit, die praktische Fairness und andere algorithmuspezifische Kennzahlen von beliebigen Lock-Implementierungen

1 Einleitung

auf verteiltem Speicher in verschiedenen Szenarien gemessen werden.

Anschließend werden in Kapitel 5 die sieben Lock-Algorithmen für NUMA aus Tabelle 1.1 analysiert, um zu entscheiden, welche der Algorithmen sich für eine Portierung auf verteilten Speicher eignen. Die geeigneten Algorithmen werden dann portiert, anhand der Erkenntnisse aus Kapitel 3 optimiert und mit der Benchmarksuite aus Kapitel 4 evaluiert.

Algorithmus	Jahr	Quelle	Analyse
RH-Lock	2002	[RH02]	Abschnitt 5.2
HCLH-Lock	2006	[LNS06]	Abschnitt 5.3
Cohort-Lock	2012	[DMS12]	Abschnitt 5.4
HMCS-Lock	2015	[CFMC15]	Abschnitt 5.4
AHMCS-Lock	2016	[CMC16]	Abschnitt 5.5
CST-Lock	2017	[KMK17]	Abschnitt 5.6
SHFL-Lock	2019	[KCC ⁺ 19]	Abschnitt 5.7

Tabelle 1.1: Untersuchte NUMA-Locks

Schließlich werden in Kapitel 6 die Ergebnisse zusammengefasst.

2 Hinführung zum Thema

In diesem Kapitel werden einige Grundlagen erklärt, die für diese Arbeit relevant sind. In Abschnitt 2.1 wird erklärt, wieso es Locks gibt und wie sie auf Systemen mit gemeinsamem Speicher funktionieren. Abschnitt 2.2 erläutert, worin sich verteilte Systeme von Systemen mit gemeinsamem Speicher unterscheiden und stellt die daraus resultierenden Konzepte vor, die für eine Programmierung verteilter Systeme notwendig sind. Anschließend behandelt Abschnitt 2.3 NUMA, ein Konzept für gemeinsamen Speicher, bei dem, ähnlich zu verteilten Systemen, nicht auf jeden Teil des Speichers gleich schnell zugegriffen werden kann. Schließlich wird in Abschnitt 2.4 das Ziel der Arbeit definiert.

2.1 Synchronisierung in parallelen Programmen

Synchronisierung ist in parallelen Programmen von großer Bedeutung, denn ohne sie können sich mehrere Prozesse (oder Threads) bei Speicherzugriffen in die Quere kommen. Das einfachste Beispiel hierfür ist wohl das Inkrementieren eines Zählers (siehe Abbildung 2.1).

```
1 int counter = 1;
2
3 void increment() {
4     int c = counter;
5     c = c + 1;
6     counter = c;
7     // Equivalente Kurzschrifweis:
8     // counter += 1;
9 }
```

Abbildung 2.1: Inkrementieren eines Zählers

Wird dieser Code von zwei Prozessen gleichzeitig ausgeführt, kann es passieren, dass beide Prozesse den Anfangswert 1 lesen, ihn auf 2 erhöhen und dieses Ergebnis speichern. Das Endergebnis ist dann fälschlicherweise 2 und nicht, wie erhofft, 3. Es ist also eine Inkrementierung verloren gegangen. So einen Fall bezeichnet man als *race condition*, da die beiden Prozesse sozusagen an einem Wettrennen teilnehmen, bei dem das Ergebnis des Programms davon abhängt, wie schnell die beiden Prozesse waren.

Davon abgesehen gibt es hier noch ein weiteres Problem: Wenn zwei Prozesse gleichzeitig normale Zugriffe auf denselben Speicherbereich ausführen, wobei mindestens einer der beiden Zugriffe schreibend ist, ist das Ergebnis in C++ und vielen anderen Sprachen undefiniert. Das Ergebnis kann also nicht nur die Zahl 2, sondern auch eine beliebige andere Zahl (z. B. durch Durchmischung der Bits), oder sogar ein Programmabsturz sein. So einen Fall nennt man bei gemeinsamem Speicher *data race* [EA03] [NAW06] [PFH06] [BCM10] [CLL⁺02] [SBN⁺97] [YRC05] und bei verteiltem Speicher *memory consistency error* [CDT⁺14] [DPFT19].

Um *data races* zu vermeiden, gibt es atomare Operationen. Anders als bei normalen Speicherzugriffen, ist bei atomaren Operationen der gleichzeitige Zugriff auf denselben Speicherbereich erlaubt. Neben atomarem Lesen und Schreiben bieten Prozessoren auch komplexere

2 Hinführung zum Thema

atomare Operationen an, wie z. B. *Test-and-Set* (TAS) und *Compare-and-Swap* (CAS), die in einem unteilbaren Schritt ausgeführt werden, selbst wenn zwei Prozesse gleichzeitig auf denselben Speicherbereich zugreifen.

Diese Vorteile haben aber auch ihren Preis. Atomare Operationen sind um ein Vielfaches langsamer als normale Speicherzugriffe, da der Prozessor bei jeder Operation dafür sorgen muss, dass die Speicheradresse kohärent bleibt. D. h. er muss dafür sorgen, dass alle Prozesse konsistente Werte beobachten. Außerdem verhindert die Nutzung von atomaren Operationen allein nicht unbedingt *race conditions*, da sie sich immer nur auf eine Speicheradresse beziehen, aber viele Programme mehrere Speicherbereiche gemeinsam bearbeiten möchten.

Eine Möglichkeit, *race conditions* (und auch *data races*) zu vermeiden, ist wechselseitiger Ausschluss (*mutual exclusion*). Die Idee dabei ist es, nur einem Prozess gleichzeitig zu erlauben, einen bestimmten Abschnitt im Code auszuführen. Denn wenn nur ein Prozess den Code ausführt, der den gemeinsamen Speicherbereich modifiziert, können in diesem sogenannten kritischen Abschnitt weder *race conditions*, noch *data races* auftreten. Wechselseitiger Ausschluss wird von verschiedenen Synchronisierungskonzepten, wie z. B. Monitoren, gewährleistet. Das zugrunde liegende Konzept ist aber meist das eines Locks und stammt bereits aus dem Jahr 1962 [Dij65].

Bei einem Lock beginnt der kritische Abschnitt, indem ein Prozess den Lock akquiriert und endet, indem der Prozess den Lock wieder frei gibt. Da der Lock nur von einem Prozess gleichzeitig akquiriert werden kann, warten weitere Prozesse beim Akquirieren so lange, bis der Lock wieder freigegeben wurde. Eine korrekte Implementierung der Inkrementierung eines Zählers mit einem Lock ist in Abbildung 2.2 zu sehen.

```
1 int counter = 1;
2 Lock lock;
3
4 void increment() {
5     lock.acquire(); // Warte auf Lock
6     int c = counter;
7     c = c + 1;
8     counter = c;
9     lock.release(); // Gebe Lock frei
10 }
```

Abbildung 2.2: Inkrementieren eines Zählers mit einem Lock

Neben diesen exklusiven Locks gibt es auch andere Varianten: die am weitesten Verbreitete ist wohl der *Reader-Writer-Lock* (kurz RW-Lock, wird auch geteilter Lock genannt). Die Idee hierbei ist, zwischen lesenden und schreibenden Zugriffen zu unterscheiden. Während schreibende Zugriffe, wie oben gezeigt, exklusiven Zugriff benötigen, können mehrere Prozesse problemlos gleichzeitig lesend auf dieselbe Speicheradresse zugreifen. Wechselseitiger Ausschluss für lesende Zugriffe würde daher das Programm unnötig verlangsamen. Ein RW-Lock bietet dementsprechend zwei verschiedene Möglichkeiten der Akquisition: eine Exklusive und eine Geteilte. Ein RW-Lock kann nur von einem Prozess gleichzeitig exklusiv akquiriert sein, hierdurch wird wechselseitiger Ausschluss für schreibende Zugriffe ermöglicht. Wenn ein RW-Lock hingegen nicht exklusiv akquiriert ist, können beliebig viele Prozesse ihn geteilt akquirieren, um lesende Zugriffe auszuführen. So können lesende Zugriffe parallel zu anderen lesenden Zugriffen sein, aber nicht parallel zu schreibenden Zugriffen. Diese Arbeit beschäftigt sich hauptsächlich mit normalen Locks, RW-Locks spielen bei der Programmierung verteilter Systeme aber eine wichtige Rolle (siehe Abschnitt 3.1).

2.1.1 Spin-Locks

Ein einfacher Algorithmus für die Implementierung eines Locks ist der TAS-Lock. Dieser Lock nutzt nur eine zentrale Speicheradresse an der steht, ob der Lock verfügbar ist. Da diese Speicheradresse nur die Werte 0 und 1 annehmen kann, wird sie häufig als *Flag* bezeichnet.

Um den Lock zu akquirieren, muss ein Prozess das *Flag* atomar von 0 auf 1 ändern. Dies wird mit einer TAS-Operation gemacht. Diese Operation setzt den Wert einer Speicheradresse atomar auf 1 und gibt den vorher enthaltenen Wert zurück. Mit dem Rückgabewert der TAS-Operation kann der Prozess prüfen, ob der Lock bereits akquiriert ist. Das ist der Fall, wenn das *Flag* bereits den Wert 1 hatte. Dann hat die TAS-Operation den Wert von 1 auf 1 gesetzt (also nichts geändert) und der Prozess muss es in einer Schleife erneut versuchen. Nur wenn der Lock frei war, ist der Rückgabewert eine 0, die atomar auf eine 1 geändert wurde. Dann kann der Prozess sicher den kritischen Abschnitt betreten, während alle anderen Prozesse durch den Wert 1 sehen, dass der Lock akquiriert ist.

Um den Lock wieder freizugeben setzt der Prozess das *Flag* atomar auf den Wert 0 (in Abbildung 2.3 mit der Funktion `clear`). Da Prozesse beim Akquirieren in einer Schleife immer wieder das *Flag* prüfen müssen, bezeichnet man den TAS-Lock und dessen Varianten als *Spin-Locks*. Abbildung 2.3a zeigt eine C++-Implementierung eines TAS-Locks, wobei `test_and_set` mit `tas` abgekürzt wurde.

```

1 std :: atomic_flag flag;
2
3 void acquire() {
4     int b = B_MIN;
5
6     std :: atomic_flag flag;
7     void acquire() {
8         while(flag.tas()) {
9             while(flag.test());
10            do {
11                wait(b);
12                b = min(b * 2, B_MAX);
13            } while(flag.test());
14        }
15    }
16
17 void release() {
18     flag.clear();
19 }
20
21 }
```

(a) TAS-Lock

(b) TTS-Lock

(c) TTS-Lock mit exp. Backoff

Abbildung 2.3: Einige Spin-Lock-Algorithmen

Ein großes Problem des TAS-Locks ist, dass wartende Prozesse durchgehend immer wieder die atomare TAS-Operation ausführen. Wie zuvor erwähnt, sind atomare Operationen deutlich langsamer als normale Speicherzugriffe. Das ist besonders ausgeprägt, wenn es sich, wie bei TAS, um schreibende Zugriffe handelt und es betrifft nicht nur den Prozess, der die Operation ausführt.

Um Speicherzugriffe schneller zu machen, verwenden Prozessoren einen schnellen Zwischenspeicher (engl. *cache*) (meist sogar mehrere). Die Werte von Speicheradressen werden beim ersten Zugriff in den Zwischenspeicher geladen, wodurch erneute Zugriffe nicht mehr auf den langsamen Hauptspeicher zugreifen müssen. Um den Hauptspeicher und die Zwischenspeicher aller Prozessoren bei Änderungen kohärent zu halten, wird ein sogenannter Zwischenspeicher-Kohärenz-Mechanismus eingesetzt [CF78]. Bei einer schreibenden atomaren Operation, wie TAS, muss dieser alle Kopien der Speicheradresse in den Zwischenspeichern der anderen Prozessoren invalidieren. Wenn mehrere Prozesse auf verschiedenen Prozessoren laufen und dabei

2 Hinführung zum Thema

atomar auf dieselbe Speicheradresse zugreifen, führt das dazu, dass die Prozesse ständig gegenseitig ihre Zwischenspeicher invalidieren, was bei allen beteiligten Prozessen erheblichen Overhead verursacht.

Um dieses Problem zu vermeiden, gibt es eine Variante des TAS-Locks: den *Test-and-Test-and-Set* (TTS)-Lock [RS84] (siehe Abbildung 2.3b). Wenn ein Prozess daran scheitert, mit der TAS-Operation den Lock zu akquirieren, versucht er es beim TTS-Lock nicht direkt erneut, sondern wartet zuvor in einer Schleife mit einer atomaren Lese-Operation (hier als *test* bezeichnet), bis der Lock frei ist. Dadurch wird die Anzahl von langsamem schreibenden atomaren Operationen stark reduziert, wodurch dieser Lock deutlich schneller ist.

Auch beim TAS-Lock kann es kurzzeitig zu erheblichem Overhead durch den Zwischenspeicher-Kohärenz-Mechanismus kommen, wenn der Lock freigegeben wird und daraufhin mehrere Prozesse versuchen, ihn mit einer TAS-Operation zu akquirieren. Um die Wahrscheinlichkeit für diesen Fall zu senken, wird in [And90] vorgeschlagen, im Falle eines solchen Konflikts eine kurze Zeit zu warten und diese Wartezeit mit jedem weiteren Konflikt, bis zu einer Obergrenze, zu verdoppeln. Diese Wartezeit steigt damit exponentiell. Um auch die Anzahl der Lese-Operationen zu senken, wird typischerweise auch nach Lese-Operationen gewartet und die Wartezeit erhöht, wenn der Lock nicht verfügbar ist. Diese Strategie wird exponentieller Backoff genannt und stammt ursprünglich aus [AC89], wo sie auf andere Synchronisierungskonzepte angewandt wurde. Ein TTS-Lock mit exponentiellem Backoff ist in Abbildung 2.3c zu sehen. Die Implementierung basiert auf [SS01].

Auch wenn der Einsatz von exponentiellem Backoff die Performance verbessert, bringt er einen neuen Nachteil mit sich. Die optimalen Werte für die initiale und maximale Wartezeit hängen von vielen Faktoren ab. Unter anderem von der Anzahl der beteiligten Prozesse, der Länge des kritischen Abschnitts und der Dauer einer einzelnen Operation. Daher muss der Lock für eine optimale Performance auf das spezifische System und Programm angepasst werden. Das bedeutet zusätzlichen Aufwand bei der Entwicklung.

2.1.2 Fairness

Neben den bereits genannten Problemen haben einfache *Spin-Locks*, wie die aus Unterabschnitt 2.1.1, noch einen weiteren Nachteil: Sie sind unfair. Fairness bezeichnet bei Locks, wie viele andere Prozesse einen Prozess maximal überholen dürfen, der als Erstes versucht, den kritischen Abschnitt zu betreten [Fra86].

Wenn ein *Spin-Lock* freigegeben wird, haben alle wartenden Prozesse die Möglichkeit, den Lock zu akquirieren. Es ist also Zufall, welcher Prozess als Nächstes den kritischen Abschnitt betreten darf. Ein *Spin-Lock* ist somit maximal unfair, da beliebig viele Prozesse einen wartenden Prozess überholen können.

Bei einem Lock der so unfair ist, kann es sogar passieren, dass ein Prozess für immer auf den Lock wartet, weil er immer wieder von einem anderen Prozess überholt wird. In so einem Fall spricht man von verhungern. Ein Lock, der im Gegensatz dazu zumindest eine minimale Fairnessgarantie bietet, also nur eine begrenzte Anzahl von Prozessen überholen lässt, wird entsprechend als hungerfrei (engl. *starvation-free*) bezeichnet. Ein maximal fairer Lock lässt gar keinen anderen Prozess überholen, wenn ein Prozess begonnen hat, auf den Lock zu warten. So ein Lock garantiert demnach eine *first in first out* (FIFO) Reihenfolge der wartenden Prozesse.

2.1.3 MCS-Lock

Der MCS-Lock ist ein sehr verbreiteter Lock von Mellor-Crummey und Scott aus [MCS91b], auf dem viele andere Lock-Algorithmen aufbauen. Dieser Lock verwaltet eine Warteschlange der akquirierenden Prozesse, um FIFO, also maximale Fairness zu garantieren. Darüber hinaus bietet er noch einen weiteren Vorteil gegenüber den einfachen *Spin-Locks* aus Unterabschnitt 2.1.1: Beim MCS-Lock greift jeder Prozess in der Warteschleife nur auf einen eigenen Speicherbereich zu. Dadurch wird vollständig vermieden, dass wartende Prozesse die Zwischenspeicher anderer Prozesse invalidieren. Umso mehr Prozesse beteiligt sind, umso größer ist der Gewinn hierdurch.

```

1  struct node {
2      atomic<bool> locked;
3      atomic<node*> next;
4  };
5  node mynode; // Pro Prozess
6  atomic<node*> tail = NULL; // Pro Lock

```

(a) Felder


```

1 mynode.next.store(NULL);           1 if (mynode.next.load() == NULL) {
2 node* pred = tail.exchange(&mynode); 2     node* expected = &mynode;
3 if (pred != NULL) {               3     if (tail.cas(expected, NULL))
4     mynode.locked.store(true);    4         return;
5     pred->next.store(&mynode);   5     while (mynode.next.load() == NULL);
6     while (mynode.locked.load());  6 }
7 }                                7     mynode.next.load() ->locked.store(false);

```

(b) Akquirieren des Locks (`acquire`)
(c) Freigeben des Locks (`release`)

Abbildung 2.4: MCS-Lock

Abbildung 2.4 zeigt eine C++-Implementierung des MCS-Locks. Die Funktion `compare_exchange_strong` wurde in Abbildung 2.4c mit `cas` abgekürzt, da es sich hierbei um eine *Compare-and-Swap* (CAS)-Operation handelt. Wie bereit erwähnt, verwaltet der MCS-Lock eine Warteschlange der akquirierenden Prozesse. Jeder Prozess hat einen Warteschlangenknoten (engl. *node*) in seinem Feld `mynode`. Dieser Knoten besteht aus zwei Feldern, auf die atomar von mehreren Prozessen zugegriffen werden kann: `locked` und `next` (vgl. Abbildung 2.4a). Das Feld `locked` ist ein *Flag*, welches dem Prozess signalisiert, ob er auf einen Vorgänger warten muss. Dies ist sehr ähnlich zu dem *Flag* eines *Spin-Locks* (vgl. Unterabschnitt 2.1.1), allerdings hat beim MCS-Lock jeder Prozess ein eigenes `locked-Flag`. Das Feld `next` ist ein Zeiger auf den Knoten des nächsten Prozesses in der Warteschlange. Über dieses Feld weiß ein Prozess, an wen er den Lock übergeben muss, wenn er fertig ist. Die Warteschlange ist demnach eine einfache verlinkte Liste. Zusätzlich gibt es einmal pro Lock einen Zeiger namens `tail` (deutsch Ende), der auf den letzten Knoten der Warteschlange zeigt. Dieser Zeiger wird von Prozessen genutzt, um sich am Ende der Warteschlange einzuröhren. Initial ist dieser Zeiger `NULL`, er zeigt also ins Leere, da noch kein Prozess wartet.

Will ein Prozess den MCS-Lock akquirieren, setzt er zunächst den `next`-Zeiger seines Knotens auf `NULL`, falls er zuvor schon einmal einen Nachfolger hatte. Dann reiht er sich in die Warteschlange ein, indem er mit einer atomaren `exchange`-Operation (Synonym für `swap`) den `tail`-Zeiger durch einen Zeiger auf seinen eigenen Knoten ersetzt (Abbildung 2.4b, Zeile 2). Diese Operation liefert als Rückgabewert den alten Wert von `tail`. Diesen alten Zeiger speichert er in der lokalen Variable `pred` (kurz für engl. *Predecessor*, deutsch Vorgänger).

2 Hinführung zum Thema

Die Variable `pred` enthält nun also entweder den Zeiger `NULL` oder einen Zeiger auf den Knoten des letzten Prozesses, der sich in die Warteschlange eingereiht hat. Im ersten Fall gibt es keinen Vorgänger, d. h. der Prozess kann den kritischen Abschnitt direkt betreten. Im zweiten Fall muss er darauf warten, dass sein Vorgänger ihm Bescheid gibt. Hierfür setzt der Prozess das *locked-Flag* seines Knotens auf `true` und lässt den `next`-Zeiger des Vorgängerknotens auf seinen eigenen Knoten zeigen (Abbildung 2.4b, Zeile 4 und 5). Dann wartet er in Zeile 6 in einer Schleife so lange, bis sein Vorgänger das *locked-Flag* wieder auf `false` setzt.

Um den Lock wieder freizugeben, prüft ein Prozess zunächst anhand des `next`-Feldes seines Knotens, ob er einen Nachfolger hat (Abbildung 2.4c, Zeile 1). Wenn das der Fall ist, setzt er in Zeile 7 das *locked-Flag* seines Nachfolgers auf `false` und ist fertig. Gibt es hingegen keinen Nachfolger, muss der `tail`-Zeiger zurück auf `NULL` gesetzt werden, damit der nächste Prozess nicht fälschlicherweise denkt, er hätte einen Vorgänger. Beim Zurücksetzen muss aber berücksichtigt werden, dass sich jederzeit parallel ein neuer Nachfolger einreihen könnte. Im MCS-Lock wird dafür eine CAS-Operation verwendet.

Die CAS-Operation ist sehr mächtig, da sie in einem atomaren Schritt eine Prüfung und einen schreibenden Zugriff ausführt. Zunächst prüft sie, ob die Speicheradresse einen erwarteten Wert enthält. Wenn das der Fall ist, schreibt sie einen neuen Wert an die Adresse, ansonsten ändert sie nichts. Damit ein Prozess weiß, ob die CAS-Operation den Wert geändert hat, liefert sie `true` zurück, wenn sie erfolgreich war und sonst `false`.

Der Prozess versucht also den `tail`-Zeiger mit einer CAS-Operation zurückzusetzen (Abbildung 2.4c, Zeile 3). Der zu schreibende Wert ist `NULL` und der erwartete Wert ist die Speicheradresse seines eigenen Knotens, denn wenn sich noch kein Nachfolger in die Warteschlange eingereiht hat, war der Prozess selbst der Letzte, sodass `tail` immer noch auf diesen Knoten zeigen sollte. Wenn die CAS-Operation erfolgreich ist, ist der Lock wieder in seinem Ausgangszustand. Der Prozess ist damit fertig und beendet mit `return` vorzeitig die `release`-Funktion.

Ansonsten hat sich bereits ein Nachfolger in die Warteschlange eingereiht. Dieser hat sich aber evtl. noch nicht bei seinem Vorgänger, also dem Prozess der gerade den Lock freigeben möchte registriert. Zur Erinnerung, ein akquirierender Prozess tauscht zuerst den `tail`-Zeiger (Abbildung 2.4b, Zeile 2) und registriert sich erst später bei seinem Vorgänger (Abbildung 2.4b, Zeile 5). Daher muss der Prozess nun warten, bis sein Nachfolger den `next`-Zeiger ändert. Genau wie beim Warten auf den Vorgänger (Abbildung 2.4b, Zeile 6), wird beim Warten auf den Nachfolger (Abbildung 2.4c, Zeile 5) in der Warteschleife nur auf den lokalen Speicher des eigenen Warteschlangenknotens zugegriffen. Sobald der Nachfolger sich registriert hat, kann der Vorgängerprozess wieder in Zeile 7 dessen *locked-Flag* auf `false` setzen und ist fertig.

2.2 Programmierung verteilter Systeme

Systeme mit gemeinsamem Speicher und verteilte Systeme unterscheiden sich stark bei der Entwicklung von parallelen Programmen. Bei Ersteren handelt es sich um einzelne Computer, bei denen Prozessoren und Speicher in derselben Maschine verbaut sind. Durch diese Nähe können alle Prozessoren relativ schnell auf alle Speicherbereiche zugreifen. Im Gegensatz dazu sind bei verteilten Systemen mehrere Computer (auch als Knoten, engl. *node*, bezeichnet) über Netzwerkkabel miteinander verbunden. Ein Zugriff auf den Speicher eines anderen

Computers im Netzwerk ist deutlich langsamer.

Der größte Vorteil von verteilten Systemen ist eine höhere Leistungsfähigkeit. Während es bei gemeinsamem Speicher ab einer gewissen Anzahl an Prozessoren nicht mehr möglich ist, weitere hinzuzufügen und einzelne Prozessoren auch nicht beliebig schnell sind, lassen sich verteilte Systeme erweitern, indem mehr Computer angeschlossen werden. Aus diesem Grund handelt es sich bei allen Supercomputern um verteilte Systeme.

Ein Nachteil ist dafür eine erhöhte Komplexität bei der Programmierung. Zusätzlich zu allen Herausforderungen, die die Entwicklung von parallelen Programmen in Systemen mit gemeinsamem Speicher mitbringt, muss bei verteilten Systemen die Datenlokalität berücksichtigt werden. Bevor mit Daten gerechnet werden kann, muss sichergestellt werden, dass diese Daten auf dem richtigen Knoten zur Verfügung stehen, um langsame Zugriffe auf entfernten Speicher zu vermeiden. Für die Kommunikation zwischen den Knoten eines verteilten Systems gibt es verschiedene Konzepte. Zwei der verbreitetsten Konzepte sind *Partitioned Global Address Space* (PGAS) [Alm11] und *Message Passing* [MPI15].

Bei PGAS macht jeder Prozess einen Teil seines Speichers für alle Prozesse zugänglich. Diese öffentlichen Speicherbereiche aller Computer werden zu einem großen Speicherbereich zusammengefasst. Dabei bleibt aber weiterhin klar, welche Speicheradressen lokalen und welche entfernten Speicher repräsentieren. So kann programmiert werden, wie in einem System mit gemeinsamem Speicher, aber langsame Zugriffe auf entfernten Speicher können bewusst vermieden werden. Es gibt zahlreiche Sprachen und Bibliotheken, die PGAS implementieren, z. B. Co-Array-Fortran (CAF) [NR98], DASH [FFK16] und UPC [UPC13].

Bei *Message Passing* werden direkt die Nachrichten, die zwischen den Computern ausgetauscht werden, modelliert. Dieses Konzept ist als *Message Passing Interface* (MPI) [MPI15] standardisiert. MPI ist sehr weit verbreitet und es gibt zahlreiche Implementierungen des Standards, die zum Teil für spezifische Hardware entwickelt und optimiert wurden [GBH18]. Einige PGAS-Bibliotheken nutzen daher intern MPI.

Bei MPI gibt es drei Ausprägungen der Kommunikation zwischen Prozessen: kollektive, Punkt-zu-Punkt- und einseitige Kommunikation. Bei kollektiver Kommunikation sind viele Prozesse an einer Operation beteiligt, ein einfaches Beispiel ist ein Broadcast, bei dem ein Prozess einen Wert an alle anderen Prozesse verteilt. Bei Punkt-zu-Punkt-Kommunikation tauschen zwei Prozesse einzelne Nachrichten aus. Einseitige Kommunikation erlaubt es einem Prozess auf einen zuvor veröffentlichten Speicherbereich eines anderen Prozesses zuzugreifen, ähnlich wie bei PGAS.

Um einen Speicherbereich für einseitige Kommunikation zu veröffentlichen und so anderen Prozessen zur Verfügung zu stellen, wird ein gemeinsames sogenanntes Fenster (engl. *window*) von allen an der Kommunikation beteiligten Prozessen erzeugt. Jeder Prozess kann dabei einen beliebigen Speicherbereich freigeben. Möchte ein Prozess über dieses Fenster nur auf den Speicher der anderen Prozesse zugreifen, aber selbst keinen Speicher freigeben, kann er auch einen Speicherbereich der Länge 0 freigeben. Über dieses Fenster kann dann durch Angabe des Zielprozesses der freigegebene Speicher referenziert werden, außerdem werden alle Speicherzugriffe über dieses Fenster synchronisiert.

Bei einseitiger Kommunikation mit MPI spricht man auch von *Remote Memory Access* (RMA), da über das Fenster auf einen Teil des Speichers des anderen Prozesses zugegriffen werden kann. Manche Hardware erlaubt solche Speicherzugriffe über das Netzwerk sogar ohne Beteiligung des Betriebssystems auf dem Zielrechner, typischerweise indem die Netzwerkkarte selbst Zugriff auf den Hauptspeicher bekommt [ARK10] [Inf07] [AAC⁺10] [BK11]. In so einem Fall spricht man von *Remote Direct Memory Access* (RDMA).

2.3 NUMA

Auch wenn bei Systemen mit gemeinsamem Speicher die Prozessoren und der Hauptspeicher im selben Computer sind, was Speicherzugriffe deutlich schneller macht als bei verteilten Systemen, gibt es Architekturen, bei denen jeder Prozessor auf einen bestimmten Speicherbereich schneller zugreifen kann als auf alle anderen. Bei solchen Architekturen spricht man von *Non Uniform Memory Access* (NUMA) [Lam13]. Während bei *Uniform Memory Access* (UMA)-Systemen der Hauptspeicher über einen Bus angebunden ist, der allen Prozessoren direkten Zugriff auf alle Speicherbereiche gewährt (vgl. Abbildung 2.5), hat bei NUMA-Systemen jeder einzelne Prozessor einen lokalen Speicherbereich, auf den er besonders schnell zugreifen kann (vgl. Abbildung 2.6). Ein Prozessor bildet mit seinem lokalen Speicherbereich einen so genannten NUMA-Knoten (engl. *NUMA node*). Zugriffe auf die Speicherbereiche der anderen Prozessoren sind weiterhin möglich, aber dazu ist Kommunikation mit dem zuständigen Prozessor notwendig, was diese deutlich langsamer macht. Wenn Prozessoren vor allem ihren eigenen Speicherbereich nutzen, kann NUMA die Performance verbessern, da Prozessoren sich nicht wie bei UMA auf dem einzigen Bus gegenseitig blockieren.

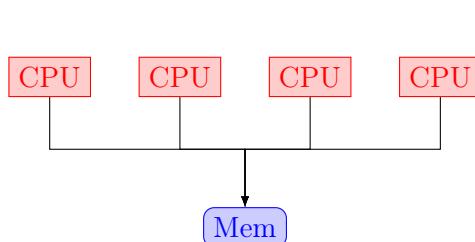


Abbildung 2.5: Uniform Memory Access

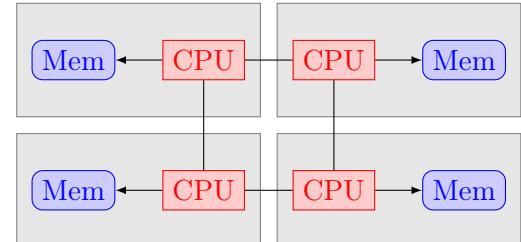


Abbildung 2.6: Non-Uniform Memory Access

2.4 Ziel der Arbeit

Obwohl Locks einer der Grundbausteine der Programmierung paralleler Programme sind und die Leistungsfähigkeit von Supercomputern nur mit parallelen Programmen voll ausgenutzt werden kann, gibt es kaum Forschung zu Lock-Algorithmen für verteilten Speicher. Dafür gibt es aber eine Vielzahl an Lock-Algorithmen für Systeme mit gemeinsamem Speicher, die auch NUMA berücksichtigen. Die Programmierung auf gemeinsamem Speicher ist sehr ähnlich zu der Programmierung von verteilten Systemen mit RMA. Zudem sind bei NUMA, genau wie bei verteiltem Speicher, Zugriffe auf lokalen Speicher schneller als Zugriffe auf entfernten Speicher.

In dieser Arbeit soll daher die folgende Forschungsfrage beantwortet werden: Können Lock-Algorithmen für NUMA-Systeme auf verteilten Speicher portiert werden, um eine bessere Performance als bestehende Implementierungen verteilter Locks zu erreichen?

3 Locks in verteilten Systemen

Genau wie in Systemen mit gemeinsamem Speicher ist auch in verteilten Systemen die Synchronisierung von Speicherzugriffen wichtig. Es gibt dort zwar nicht einen großen gemeinsamen Speicherbereich, aber mit Hilfe von RMA kann auf den Speicher anderer Knoten zugegriffen werden. Solche Zugriffe müssen geschützt werden, um *race conditions* und *memory consistency errors* zu vermeiden. Auch in verteilten Systemen eignen sich dafür Locks.

3.1 MPI Locks

Bei MPI ist es sogar zwingend erforderlich, RMA-Zugriffe zu synchronisieren. Es muss entweder *active target synchronization* oder *passive target synchronization* genutzt werden [MPI15, Kapitel 11.5, S. 436]. Bei *active target synchronization* sind beide Prozesse an der Synchronisierung beteiligt, was eine enorme Einschränkung bei der Programmierung sein kann, weil Prozesse nicht unabhängig voneinander agieren können. *passive target synchronization* hingegen erlaubt es, ähnlich wie bei gemeinsamem Speicher, ohne Beteiligung eines anderen Prozesses auf dessen freigegebenen Speicherbereich zuzugreifen. Zur Synchronisierung dienen hierbei RW-Locks (vgl. Abschnitt 2.1): Mit `MPI_Win_lock` wird zunächst ein exklusiver oder geteilter Lock für den freigegebenen Speicher eines Zielprozesses akquiriert (mit `MPI_LOCK_EXCLUSIVE` oder `MPI_LOCK_SHARED`). Damit beginnt eine sogenannte Zugriffsepoke (engl. *access epoch*), in der per RMA auf diesen Speicher zugegriffen werden kann. Schließlich muss die Epoche mit `MPI_Win_unlock` wieder beendet werden. Mehrere Prozesse können gleichzeitig geteilte Locks auf denselben Speicher haben, aber ein exklusiver Lock garantiert, dass kein anderer Prozess gleichzeitig einen Lock auf den Speicher hält. Er bietet also wechselseitigen Ausschluss, aber nicht ohne Einschränkung:

The call to `MPI_WIN_LOCK` may block until an exclusive lock on the window is acquired; or, the first [...] calls may not block, while `MPI_WIN_UNLOCK` blocks until a lock is acquired [MPI15, Kapitel 11.5.3, S. 448]

Da nicht garantiert ist, dass `MPI_Win_lock` direkt einen Lock akquiriert, muss eine Operation, z. B. ein lesender Zugriff, initiiert und mit `MPI_Win_flush` vollständig ausgeführt werden, um wechselseitigen Ausschluss zu gewährleisten.

Alternativ zu `MPI_Win_lock` kann mit `MPI_Win_lock_all` ein Lock für alle Prozesse, die ein Fenster nutzen, akquiriert werden. Dieser Lock ist allerdings immer geteilt. Auf diese Weise kann kein exklusiver Lock akquiriert werden. Ein geteilter Lock für alle Prozesse eignet sich gut, um auf verteiltem Speicher zu programmieren, als würde es sich um gemeinsamen Speicher handeln, da ungehindert auf alle Speicherbereiche des Fensters zugegriffen werden kann. So können Algorithmen, die für gemeinsamen Speicher entwickelt wurden auch auf verteiltem Speicher implementiert werden. Manche Bibliotheken wie BCL [BBY19] nutzen `MPI_Win_lock_all` sogar nur einmalig beim Programmstart und haben so über die gesamte Laufzeit eine einzige große Zugriffsepoke.

Es ist nicht möglich, über dasselbe Fenster gleichzeitig mehrere Locks für denselben Zielprozess zu halten:

Multiple RMA access epochs (with calls to MPI_WIN_LOCK) can occur simultaneously; however, each access epoch must target a different process. [MPI15, Kapitel 11.5.3, S. 446]

Daher kann in einer Zugriffsepoke, die mit `MPI_Win_lock_all` gestartet wurde, nicht auf demselben Fenster mit `MPI_Win_lock` ein exklusiver Lock akquiriert werden. Um trotzdem Locks mit dieser Methode nutzen zu können, muss in diesem Fall ein eigenes Fenster pro Lock erzeugt werden. Bei vielen Locks kann das je nach MPI-Implementierung ein ziemlicher Overhead sein, da ein Fenster im schlimmsten Fall $\Omega(p)$ Speicher auf jedem der p Prozesse benötigt [GBH18]. Mit einem einzigen Fenster können zwar theoretisch so viele unabhängige Locks implementiert werden, wie es beteiligte Prozesse gibt, aber die Handhabung wird dann schwierig, da die Anzahl der Prozesse meist erst zur Laufzeit bekannt ist.

3.2 RMA-MCS

Um von der oben genannten Limitierung der Locks des MPI-Standards unabhängig zu sein, bietet es sich an, eigene Locks auf Basis anderer Operationen von MPI zu implementieren. Dies ermöglicht auch die Nutzung von Lock-Algorithmen, die besser für die eigenen Anforderungen geeignet sind, z. B. weil sie eine bessere Performance bei besonders hoher oder geringer Anzahl an konkurrierenden Lock-Akquisitionen (engl. *contention*) haben oder sie Vorrang für exklusive oder Vorrang für geteilte Lock-Akquisitionen bieten.

Es gibt zwar einige Lock-Algorithmen für verteilte Systeme, diese unterscheiden jedoch in der Regel nicht zwischen Prozessen, die auf demselben Knoten laufen und Prozessen auf anderen Knoten. D. h. sie berücksichtigen keine Geschwindigkeitsunterschiede zwischen Zugriffen auf lokalen und entfernten Speicher, wodurch keine optimale Performance erreicht werden kann. Bei Locks für NUMA-Systeme kann man höhere Geschwindigkeiten erreichen, wenn man die Fairness einschränkt, indem Prozesse, die auf demselben Knoten laufen, Vorrang bei der Lock-Akquisition erhalten. Denn wenn der Lock seltener an Prozesse in anderen Knoten übergeben wird, sind weniger langsame Zugriffe auf entfernten Speicher notwendig.

Die einzige bekannte Arbeit, die diese Technik für verteilte Systeme anwendet, ist [SBH16]. In ihr stellen Schmid et. al. ihren Lock RMA-RW vor, welcher ein verteilter RW-Lock auf Basis eines hierarchischen MCS-Locks (HMCS-Lock, siehe Abschnitt 5.4) ist, den sie RMA-MCS nennen. Für ihren Performancevergleich nutzen sie einen MCS-Lock, den sie als D-MCS-Lock (D für engl. *distributed*) bezeichnen, der auf einer Implementierung aus [GHTL14][Kapitel 4.7, S. 130] beruht, sowie die Locks, die von ihnen genutzten MPI-Implementierung *fast one-sided MPI* (foMPI).

foMPI implementiert nur den Teil der MPI-Spezifikation für einseitige Kommunikation und nutzt dafür *Distributed Memory Application* (DMAPP) [BR10], eine RDMA-Bibliothek für HPC-Systeme der Firma Cray [GBH18]. Da im Rahmen dieser Masterarbeit kein Cray-System zur Verfügung steht und die Performance von zwei Algorithmen nur schwer verglichen werden kann, wenn nicht beide auf derselben Hardware laufen, muss für einen Performancevergleich die MPI-Implementierung in RMA-MCS ausgetauscht werden.

Normalerweise muss bei einem Wechsel der MPI-Implementierung der Anwendungscode nicht angepasst werden, da eine einheitliche Schnittstelle verwendet wird. Da foMPI allerdings

nicht die ganze MPI-Spezifikation implementiert, haben die Autoren sich dazu entschieden, bei allen Funktionen und Konstanten den Präfix „MPI“ durch „foMPI“ zu ersetzen. Diese Ersetzung muss rückgängig gemacht werden, damit RMA-MCS die standardisierte MPI-Schnittstelle nutzt.

3.3 Fortschrittsgarantien von MPI

Leider ist es mit dieser einfachen Ersetzung nicht getan. Die so modifizierte Version von RMA-MCS läuft bei der Verwendung von Intel-MPI¹ mit mindestens zwei Rechenknoten in einen Deadlock. Der Grund dafür liegt in der Übergabe des Locks von einem Prozess an seinen Nachfolger.

Bei einem MCS-Lock reihen sich Prozesse in eine Warteschlange ein und warten in einer Schleife darauf, dass ein lokaler Speicherbereich von ihrem Vorgänger geändert wird (vgl. Unterabschnitt 2.1.3). Dies hat den großen Vorteil, dass die vielen Speicherzugriffe beim Warten nicht über das Netzwerk gehen müssen. Der D-MCS-Lock aus [GHTL14] und auch RMA-MCS und RMA-RW aus [SBH16] nutzen dafür eine Implementierung ähnlich zu der in Abbildung 3.1 gezeigten.

```

1 do {
2     MPI_Win_sync(window);
3 } while (window_mem[blocked] == 1);

(a) Auf Vorgänger warten
(b) Lock an Nachfolger übergeben

```

Abbildung 3.1: Lockübergabe, die in Intel-MPI zu einem Deadlock führt

Ein direkter Zugriff auf den lokalen Speicher des Fensters ist deutlich schneller als ein Zugriff über `MPI_Get` oder `MPI_Get_accumulate`. Durch `MPI_Win_sync` wird sichergestellt, dass auch bei dem Speichermodell `MPI_WIN_SEPARATE`² die Änderungen in der öffentlichen Kopie des Fensters für die private Kopie (also den lokalen Speicher) sichtbar werden. Bei Intel-MPI kann allerdings das `MPI_Win_flush` in Abbildung 3.1b nicht ausgeführt werden, während der Zielprozess mit der Schleife in Abbildung 3.1a wartet. Das liegt vermutlich an der folgenden Einschränkung der MPI-Spezifikation:

MPI implementations must guarantee that a process makes progress on all enabled communications it participates in, while blocked on an MPI call [...]. A similar issue is whether such progress must occur while a process is busy computing, or blocked in a non-MPI call [...]. Then MPI does not specify whether deadlock is avoided. [MPI15, Kapitel 11.7, S. 456]

Anscheinend zählt der Aufruf von `MPI_Win_sync` in der Schleife für Intel-MPI nicht als blockierender MPI-Aufruf, sodass `MPI_Win_flush` hängen bleibt.

¹Intel(R) MPI Library for Linux* OS, Version 2019 Update 10 Build 20210120 (id: d7908565b)

²MPI definiert zwei Speichermodelle: MPI_WIN_SEPARATE und MPI_WIN_UNIFIED. Bei dem ersten gibt es eine öffentliche und eine private Kopie des Fensterspeichers, die manuell synchron gehalten werden müssen. Dies repräsentiert Systeme mit nicht-kohärentem Speicher. Bei dem zweiten gibt es nur eine Version des Fensterspeichers, da der unterliegende Speicher selbst kohärent ist. Siehe [MPI15, Kapitel 11.4, S. 435]

3 Locks in verteilten Systemen

Eine mögliche Erklärung dafür, dass MPI keine Garantien in so einem Fall gibt, ist, dass bei Systemen, die kein RDMA unterstützen, das Betriebssystem des Zielprozesses an dem Datentransfer beteiligt sein muss. Wenn der Zielprozess selbst aber beschäftigt ist, müsste in so einem Fall ein anderer Prozess oder Thread diese Aufgabe übernehmen. Zu erkennen, dass der Zielprozess längerfristig beschäftigt sein wird, ist nicht wirklich möglich und alle RMA-Kommunikation durch einen nur dafür zuständigen Hintergrundprozess oder -thread zu leiten, könnte erheblichen Overhead mit sich bringen. Damit bleibt ohne RDMA nur die Option zu warten, bis der Zielprozess wieder eine MPI-Funktion aufruft. Wenn das aber nie passiert, gibt es einen Deadlock.

Bei Open-MPI³ ist dieses Problem noch ausgeprägter. Hier kann selbst bei ausschließlicher Nutzung von atomaren RMA-Funktionen (ohne manuelle Zugriffe auf den Speicher des Fensters) kein Lock übergeben werden, wenn der Zielprozess mit einer Schleife wie in Abbildung 3.2a wartet (die Lockfreigabe in Abbildung 3.2b ist unverändert). Dies ist mit Intel-MPI hingegen möglich, da dort `MPI_Win_flush` als blockierender Aufruf gilt.

```
1 int flag;
2 do {
3     int dummy;
4     MPI_Fetch_and_op(
5         &dummy, &flag, MPI_INT,
6         my_rank, blocked,
7         MPI_NO_OP, window);
8     MPI_Win_flush(my_rank, window);
9 } while (flag == 1);
```

(a) Auf Vorgänger warten

```
1 constexpr int ZERO = 0;
2 MPI_Accumulate(
3     &ZERO, 1, MPI_INT,
4     successor, blocked, 1, MPI_INT,
5     MPI_REPLACE, window);
6 MPI_Win_flush(successor, window);
```

(b) Lock an Nachfolger übergeben

Abbildung 3.2: Lockübergabe, die in Open-MPI zu einem Deadlock führt

Die Entwickler der PGAS-Bibliothek DASH haben eine Lösung für dieses Problem gefunden⁴. Sie verwenden die Funktion `MPI_Iprobe` um den Deadlock zu vermeiden. `MPI_Iprobe` dient eigentlich dazu, bei Punkt-zu-Punkt-Kommunikation herauszufinden, ob ein anderer Prozess versucht, eine Nachricht zu senden, die der eigene Prozess empfangen könnte, ohne auf die Nachricht zu warten. Die MPI-Spezifikation garantiert darüber hinaus aber auch Programmfortschritt:

The MPI implementation of [...] `MPI_IPROBE` needs to guarantee progress:
[...] if a process busy waits with `MPI_IPROBE` and a matching message has been issued, then the call to `MPI_IPROBE` will eventually return `flag = true` unless the message is received by another concurrent receive operation or matched by a concurrent matched probe. [MPI15, Kapitel 3.8.1, S. 66]

Damit eignet sich `MPI_Iprobe` perfekt, um den Deadlock in Open-MPI und Intel-MPI zu beheben (siehe Abbildung 3.3a, die Lockfreigabe in Abbildung 3.3b ist weiterhin unverändert).

³Open MPI 4.0.4; kompiliert mit: `icc (ICC) 19.0.5.281 20190815`

⁴<https://github.com/dash-project/dash/issues/372>

```

1  do {
2      int flag;
3      MPI_Iprobe(
4          MPI_ANY_SOURCE, MPI_ANY_TAG,
5          communicator,
6          &flag, MPI_STATUS_IGNORE);
7      MPI_Win_sync(window);
8  } while (window_mem[blocked] == 1);

```

(a) Auf Vorgänger warten

```

1  constexpr int ZERO = 0;
2  MPI_Accumulate(
3      &ZERO, 1, MPI_INT,
4      successor, blocked, 1, MPI_INT,
5      MPI_REPLACE, window);
6  MPI_Win_flush(successor, window);

```

(b) Lock an Nachfolger übergeben

Abbildung 3.3: Lockübergabe mit MPI_Iprobe

3.4 Nicht-atomare Übergabe von Locks

Für die Übergabe eines MCS-Locks an den Nachfolger wird in der Literatur (wie in Abschnitt 3.3 gezeigt) die atomare RMA-Operation **MPI_Accumulate** genutzt und auf Seite des Nachfolgers mit direkten nicht-atomaren Zugriffen auf den lokalen Speicher des Fensters gewartet. Dies ist in MPI-2.2 nicht erlaubt:

A correct program must obey the following rules.

1. A location in a window must not be accessed locally once an update to that location has started, until the update becomes visible in the private window copy in process memory. [...]

A program is erroneous if it violates these rules. [MPI09, Kapitel 11.7, S. 365]

In MPI-3.1 wurde diese Regel aufgeweicht:

Accessing a location in the window that is also the target of a remote update is valid (not erroneous) but the precise result will depend on the behavior of the implementation. Updates from a remote process will appear in the memory of the target, but there are no atomicity or ordering guarantees if more than one byte is updated. Updates are stable in the sense that once data appears in memory of the target, the data remains until replaced by another update. This permits polling on a location for a change from zero to non-zero or for a particular value, but not polling and comparing the relative magnitude of values. [MPI15, Kapitel 11.7, S. 455]

Dies gilt allerdings nur, wenn das Speichermodell des Fensters **MPI_WIN_UNIFIED** ist. Beim Speichermodell **MPI_WIN_SEPARATE** ist dies weiterhin nicht erlaubt, führt aber nun zu undefiniertem Verhalten statt zu einem Fehler, was es Implementierungen ermöglicht, diesen Fall zu unterstützen. Alle für diese Arbeit verwendeten Systeme verwenden das Speichermodell **MPI_WIN_UNIFIED**, daher ist es schwer zu sagen, ob die hier gezeigten Algorithmen auf **MPI_WIN_SEPARATE**-Systemen korrekt funktionieren.

Beide Versionen der MPI-Spezifikation unterscheiden an dieser Stelle nicht zwischen atomaren Operationen wie **MPI_Accumulate** und normalen Operationen wie **MPI_Put**. Und da der wartende Prozess sowieso nicht-atomare Operationen nutzt, erhält man durch die Nutzung einer atomaren RMA-Operation keine zusätzlichen Garantien. Aus diesem Grund wird im weiteren Verlauf dieser Arbeit für solche Fälle die nicht-atomare Operation **MPI_Put** verwendet. Diese ist etwas schneller (siehe Unterabschnitt 4.3.1) und hat auf den zur Verfügung

3 Locks in verteilten Systemen

stehenden Systemen denselben Effekt wie `MPI_Accumulate`. Sollte einer der hier gezeigten Locks auf einem System zum Einsatz kommen, bei dem in solchen Fällen eine atomare Operation erforderlich ist, kann dies leicht geändert werden.

Der Ausschnitt aus der MPI-3.1-Spezifikation legt außerdem nahe, nach Möglichkeit nur ein Byte mit einem RMA-Zugriff zu ändern, wenn gleichzeitig direkt auf die Speicheradresse zugegriffen wird. In der Literatur werden aber meist mehrere Bytes verwendet, beispielsweise vier Byte in [GHTL14] durch Nutzung des Datentyps `MPI_INT` und sogar acht Byte in [SBH16] durch Nutzung von `MPI_INT64_T`, obwohl für so eine Änderung von 1 zu 0 nur ein Byte nötig wäre (z. B. mit `MPI_CXX_BOOL`).

Der Grund hierfür ist vermutlich, dass der Speicher des Fensters meist als Array betrachtet wird, wodurch es für den Programmierer leichter ist, wenn alle Elemente denselben Datentyp haben. Dabei ist es problemlos möglich, auf den Speicher des Fensters über ein Struct zuzugreifen, wenn bei dem Erzeugen des Fensters eine *displacement unit* von einem Byte angegeben wurde. Um auf ein bestimmtes Feld des Structs zuzugreifen, kann dann bei einem RMA-Zugriff als *displacement* statt der Position im Array, der Byteoffset des Felds angegeben werden. Dieser wird mit dem Makro `offsetof`⁵ ermittelt.

Auf diese Weise ist es leicht, innerhalb eines Fensters verschiedene Datentypen zu verwenden und es wird möglich, mit dem Makro `alignas`⁶ die Felder auf verschiedenen Zeilen des Zwischenspeichers zu legen (siehe Abbildung 3.4).

```
1 struct memory_layout {
2     alignas(64) bool locked;
3     alignas(64) int next;
4     alignas(64) int tail;
5 };
6 static constexpr MPI_Aint locked_disp = offsetof(memory_layout, locked);
7 static constexpr MPI_Aint next_disp = offsetof(memory_layout, next);
8 static constexpr MPI_Aint tail_disp = offsetof(memory_layout, tail);
```

Abbildung 3.4: Nutzung eines Structs für MPI-Fenster

Insgesamt ergibt sich damit die optimierte Implementierung des D-MCS-Locks in Abbildung 3.5. Diese ähnelt sehr der C++-Implementierung in Abbildung 2.4, es gibt aber ein paar kleine Unterschiede:

1. Das *locked-Flag* wird zusammen mit dem `next`-Feld direkt am Anfang von `acquire` zurückgesetzt. Diese nicht-atomare lokale Schreib-Operation ist sehr schnell, es ist daher kein großes Problem, wenn sie auch ausgeführt wird, wenn es keinen Nachfolger gibt. Durch das Vorziehen dieser Operation ist kein zweites `MPI_Win_sync` notwendig, da beide Operationen gemeinsam synchronisiert werden können (Abbildung 3.5a, Zeile 1-3).
2. Da der Warteschlangenknoten eines Prozesses nicht in einem Feld gespeichert ist, sondern im Speicher des MPI-Fensters, handelt es sich bei `next` und `tail` nicht um Zeiger, sondern um `ints`. Diese enthalten die ID eines Prozesses (in MPI als Rang, engl. *rank*, bezeichnet).

⁵<https://en.cppreference.com/w/cpp/types-offsetof>

⁶<https://en.cppreference.com/w/cpp/language/alignas>

```

1  win_mem->locked = true;
2  win_mem->next = -1;
3  MPI_Win_sync(win);
4
5 // Finde Vorgänger
6 int pred;
7 MPI_Fetch_and_op(
8     &my_rank, &pred, MPI_INT,
9     main_rank, tail_disp,
10    MPI_REPLACE, win);
11 MPI_Win_flush(main_rank, win);
12
13 if (pred != -1) {
14     // Reihe nach Vorgänger ein
15     MPI_Put(&my_rank, 1, MPI_INT,
16             pred, next_disp, 1, MPI_INT,
17             win);
18     MPI_Win_flush(pred, win);
19
20 // Warte auf Vorgänger
21 while (win_mem->locked) {
22     int flag; // Garantiere Fortschritt
23     MPI_Iprobe(
24         MPI_ANY_SOURCE, MPI_ANY_TAG,
25         comm, &flag, MPI_STATUS_IGNORE);
26     MPI_Win_sync(win);
27 }
28 }

1  int succ = win_mem->next;
2  if (succ == -1) {
3      constexpr int NULL_RANK = -1;
4      int old_value;
5      MPI_Compare_and_swap(
6          &NULL_RANK, &my_rank, &old_value,
7          MPI_INT, main_rank, tail_disp, win);
8      MPI_Win_flush(main_rank, win);
9
10 // Die Warteschlange ist leer
11 if (old_value == my_rank) return;
12
13 // Warte auf Nachfolger
14 while ((succ = win_mem->next) == -1) {
15     int flag; // Garantiere Fortschritt
16     MPI_Iprobe(
17         MPI_ANY_SOURCE, MPI_ANY_TAG,
18         comm, &flag, MPI_STATUS_IGNORE);
19     MPI_Win_sync(win);
20 }
21
22 // Benachrichtige Nachfolger
23 constexpr bool FALSE = false;
24 MPI_Put(
25     &FALSE, 1, MPI_CXX_BOOL,
26     succ, locked_disp, 1, MPI_CXX_BOOL,
27     win);
28 MPI_Win_flush(succ, win);

```

(a) Akquirieren des Locks (acquire)

(b) Freigeben des Locks (release)

Abbildung 3.5: MCS-Lock in MPI (optimierter D-MCS)

3.5 Punkt-zu-Punkt Übergabe von Locks

Es gibt noch eine weitere Möglichkeit, die in Abschnitt 3.3 beschriebenen Probleme zu vermeiden: Statt für die Übergabe des Locks mit einer Schleife auf eine RMA-Operation zu warten, können der Vorgänger- und Nachfolgerprozess auch mit Punkt-zu-Punkt-Kommunikation mit `MPI_Send` und `MPI_Recv` eine Nachricht austauschen. Diese Technik kommt auch in der PGAS-Bibliothek DASH zum Einsatz [ZMI¹⁴]. Da dieser Nachrichtenaustausch nur der Synchronisierung dient, kann die eigentliche Nachricht leer sein (vgl. Abbildung 3.6).

```

1  MPI_Recv(NULL, 0, MPI_UINT8_T,
2           predecessor, 0,
3           communicator, MPI_STATUS_IGNORE);
1  MPI_Send(NULL, 0, MPI_UINT8_T,
2           successor, 0,
3           communicator);

```

(a) Auf Vorgänger warten

(b) Lock an Nachfolger übergeben

Abbildung 3.6: Lockübergabe mit Punkt-zu-Punkt-Kommunikation

Die in Abschnitt 3.3 gezeigte Technik unter Verwendung von `MPI_Iprobe` ist auch bei Nutzung von Punkt-zu-Punkt-Kommunikation wichtig, da der MCS-Lock noch eine zweite Schleife hat, in der auf eine Änderung des lokalen Speichers durch einen anderen Prozess gewartet wird. Bei der Freigabe des Locks (Abbildung 3.5b, ab Zeile 13) muss ein Prozess unter Umständen auf seinen Nachfolger warten. Dieser Fall kommt nur selten vor (wenn ein Prozess kurz davor ist, auf seinen Vorgänger zu warten, während sein Vorgänger dabei ist, den Lock freizugeben), aber auch hier kann es ohne `MPI_Iprobe` zu einem Deadlock kommen.

3.6 Nutzung von gemeinsamem Speicher innerhalb eines Rechenknotens

Direkte Zugriffe auf Speicher sind deutlich schneller als Zugriffe auf denselben Speicher mittels RMA über MPI. Das ist auch der Grund, weshalb D-MCS und RMA-MCS [SBH16] direkte Speicherzugriffe benutzen, wenn sie in einer Schleife auf ihren Vorgänger bzw. Nachfolger warten (siehe Abschnitt 3.3). Bei diesen beiden Lock-Implementierungen werden direkte Speicherzugriffe nur für den Speicher des eigenen Prozesses genutzt, da der Speicher der anderen Prozesse potenziell auf anderen Rechenknoten liegt.

NUMA-Locks hingegen unterscheiden Prozesse danach, ob sie auf demselben NUMA-Knoten laufen. Somit ist auch bekannt, ob die Prozesse über gemeinsamen Speicher verfügen. Das erlaubt es in vielen Fällen, auch bei der Kommunikation mit anderen Prozessen, direkte Speicherzugriffe zu verwenden. MPI ermöglicht es mit der Funktion `MPI_Win_allocate_shared` ein Fenster auf gemeinsamen Speicher zu erstellen. Dafür müssen zunächst die Prozesse in Gruppen aufgeteilt werden, sodass alle Prozesse in einer Gruppe einen gemeinsamen Speicher besitzen (typischerweise weil sie auf demselben Rechenknoten laufen). Das geht mit der Funktion `MPI_Comm_split_type` und dem Typ `MPI_COMM_TYPE_SHARED`. Ist so ein Fenster auf gemeinsamen Speicher erstellt, kann die Speicheradresse für jeden Prozess per Zeigerauthmetik ausgerechnet oder mit `MPI_Win_shared_query` abgefragt werden. So kann z. B. der lokale Lock eines Cohort-Locks (siehe Abschnitt 5.4) komplett ohne RMA auskommen und stattdessen mit C++ `std::atomic`⁷ implementiert werden.

⁷<https://en.cppreference.com/w/cpp/atomic/atomic>

4 Evaluation und Optimierung bestehender Locks

Das wohl wichtigste Kriterium für den Vergleich von Locks ist die Geschwindigkeit der Implementierung. Ein schnellerer Lock ist natürlich besser, aber die Geschwindigkeit hängt nicht nur vom Algorithmus und der Hardware ab, sondern auch davon, in welchem Kontext der Lock aufgerufen wird. Es kann zum Beispiel einen Unterschied machen, wie viele Prozesse gleichzeitig versuchen, den Lock zu akquirieren (Konkurrenz (engl. *contention*)) oder ob der Prozess, der den Lock als Letztes hatte, auf demselben Rechenknoten lief. Es gibt daher zahlreiche verschiedene Szenarien in der Literatur, in denen Benchmarks durchgeführt werden. Hierbei kann man zwischen einfachen synthetischen Benchmarks und der Integration in eine bestehende Anwendung unterscheiden.

Eine Integration in eine bestehende Anwendung hat den Vorteil, dass das Szenario realistischer ist und die Ergebnisse somit besser den realen Nutzen zeigen. Leider steht im Rahmen dieser Arbeit keine Anwendung zur Verfügung, in die die Locks integriert werden könnten.

Synthetische Benchmarks sind sehr fein granular und können daher gezielt bestimmte Aspekte einer Implementierung analysieren. Beispielsweise kann man nur mit einem synthetischen Benchmark messen, wie lange es dauert, einen freien Lock zu akquirieren. Außerdem laufen sie schneller, da nur genau das ausgeführt wird, was für den Benchmark notwendig ist. Eine Sammlung solcher Benchmarks in einer Programmabibliothek wird im Folgenden als Benchmarksuite bezeichnet.

In der Literatur [RH02] [LNS06] [DMS11] [LDT⁺12] [ZLW⁺16] [SBH16] wird für synthetische Benchmarks meist eine festgelegte Anzahl an Iterationen definiert. In einer Schleife akquiriert ein Prozess den Lock und führt je nach Benchmark noch andere Operationen aus. Dabei wird die Zeit gemessen, bis alle Prozesse fertig sind. Im Gegensatz dazu wird in dieser Arbeit (analog zu [DMS12], [DK19] und [KCC⁺19]) eine Zeit vorgegeben und nach jeder Iteration geprüft, ob diese Zeit schon abgelaufen ist. Das hat den Nachteil, dass jede Iteration zusätzlichen Overhead durch eine Abfrage der Systemzeit erfährt. Dieser Overhead hat allerdings nur eine Größenordnung von einigen Nanosekunden (ns), während die Dauer von RMA-Zugriffen auf entfernten Speicher in der Größenordnung von Mikrosekunden (μ s) liegt. Der Vorteil dieser Variante ist, dass dadurch eine empirische Analyse der Fairness (siehe Unterabschnitt 4.1.3) möglich wird. In dieser Arbeit wird für solche Benchmarks eine Laufzeit von einer Sekunde pro Lock und Prozessanzahl verwendet, wobei die ersten 10 % der Zeit nicht in die Messung eingehen, sondern zum Aufwärmen dienen.

Jeder Benchmark wird acht Mal ausgeführt. Gezeigt wird immer der Median dieser Ausführungen mit 95 % Konfidenzintervall. In dieser Arbeit wird der Median verwendet, da dieser im Gegensatz zum arithmetischen Mittel nicht von Ausreißern beeinflusst wird. Solche Ausreißer können auf einem verteilten System mit vielen Nutzern leicht auftreten und würden sonst das Messergebnis verfälschen. Durch die Angabe der Konfidenzintervalle ist außerdem ersichtlich, wie groß die Abweichung des Medians zwischen den einzelnen Läufen war. Mit dieser Information konnten bei der Erstellung dieser Arbeit einige Benchmarkergebnisse

identifiziert werden, die wegen hoher Auslastung des Rechenzentrums enorme Schwankungen aufwiesen. Diese Benchmarks wurden dann zu einem späteren Zeitpunkt wiederholt.

Alle Benchmarks wurden auf dem CoolMUC-2 Linux Cluster des Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften (LRZ) ausgeführt¹. Dieses Cluster nutzt Intels Haswell Prozessoren mit einer Taktfrequenz von 2,6 GHz und hat 28 Kerne pro Rechenknoten mit je 2 Hyperthreads. Die Knoten sind mit FDR14 Infiniband verbunden. Die Verbindung hat eine Latenz von 2,3 μ s und eine Bandbreite von 13,64 GB/s pro Knoten. Als MPI-Implementierung wurde Intel-MPI² verwendet und die Benchmarks wurden mit ICPC³ kompliiert.

Um die Effekte von Hyperthreading zu vermeiden, wurden in den Benchmarks maximal 28 Prozesse auf einem exklusiv reservierten Rechenknoten ausgeführt. Jeder Knoten erhielt dabei stets die gleiche Anzahl an Prozessen.

4.1 Vergleichskriterien

Um die verschiedenen Algorithmen und Optimierungen objektiv miteinander vergleichen zu können, müssen zunächst die Vergleichskriterien definiert werden. Der Fokus liegt in dieser Arbeit auf dem Vergleich der Geschwindigkeit, aber es wird auch die Fairness empirisch untersucht.

4.1.1 Geschwindigkeit

Die Geschwindigkeit der Implementierungen ist das wichtigste Vergleichskriterium. Im Rahmen dieser Arbeit wird manchmal die durchschnittliche Dauer einer Iteration betrachtet und manchmal invers dazu der Durchsatz, also wie viele kritische Abschnitte pro Sekunde ausgeführt werden. Je nach Szenario ist die eine oder die andere Betrachtungsweise geeigneter, um die Unterschiede zwischen zwei Implementierungen herauszustellen. Beide Darstellungen zeigen aber dieselben Daten und können leicht durch Berechnung von $y \rightarrow \frac{1}{y}$ ineinander umgewandelt werden.

4.1.2 Konkurrenz

Die Konkurrenz (engl. *contention*) ist kein direktes Vergleichskriterium, hat aber einen großen Einfluss auf das Verhalten eines Locks und hilft daher bei der Interpretation der Geschwindigkeit. Bei großer Konkurrenz muss ein Prozess z. B. länger warten, bis der Lock freigegeben wird. Außerdem muss ein Prozess bei vielen Algorithmen andere Codepfade durchlaufen, wenn eine geringe Konkurrenz vorliegt. Z. B. muss bei der Freigabe eines MCS-Locks der `tail`-Zeiger nur dann zurückgesetzt werden, wenn noch kein Nachfolger wartet (vgl. Abbildung 2.4c).

Die verschiedenen Benchmarks setzen die Locks verschiedenen Konkurrenzsituationen aus. Um die tatsächliche Konkurrenz quantifizieren zu können, wird sie definiert als: der Anteil der Akquisitionen, bei denen ein Prozess den Lock nicht direkt akquirieren konnte, sondern auf einen Vorgänger warten musste. Die Konkurrenz geht also von 0 % bis 100 % und es wird

¹<https://doku.lrz.de/display/PUBLIC/CoolMUC-2>

²Intel(R) MPI Library for Linux* OS, Version 2019 Update 10 Build 20210120 (id: d7908565b)

³icpc (ICC) 19.1.2.254 20200623

nicht unterscheiden, wie viele Prozesse bei einer Konkurrenz von 100 % versuchen, den Lock zu akquirieren.

Diese Messung muss in die Lock-Implementierung eingebaut werden, da sonst nicht ersichtlich ist, ob auf einen Vorgänger gewartet wurde. Daher ist leider keine Messung der Konkurrenz bei Locks aus Bibliotheken wie z. B. DASH möglich.

4.1.3 Fairness

Neben der Geschwindigkeit ist auch die Fairness ein wichtiges Vergleichskriterium. Fairness wird häufig nur auf theoretischer Ebene betrachtet (vgl. Unterabschnitt 2.1.2). In dieser Arbeit geht es aber nicht um die theoretische Fairness der Lock-Algorithmen, sondern um die praktische Fairness der Lock-Implementierungen.

Um diese zu messen wird wie in [DMS12] gezählt, wie viele Iterationen des Benchmarks jeder Prozess in der gegebenen Zeit ausführt. Damit erhält man den Programmfortschritt, also wie weit jeder einzelne Prozess ein Programm ausführen würde. Um nun die Fairness zu bewerten wird der empirische Variationskoeffizient (engl. *Coefficient of Variation* (CV)) der einzelnen Programmfortschritte bestimmt. Dieser berechnet sich aus der Standardabweichung σ und dem Durchschnitt μ mit $\frac{\sigma}{\mu}$. Da es sich um eine Stichprobe handelt wird Bessels Korrektur [So08] bei der Berechnung der Varianz (und damit der Standardabweichung) verwendet. Ein geringerer Variationskoeffizient ist somit besser, da er eine geringere Abweichung des Programmfortschritts und damit einen faireren Lock bedeutet.

Im Gegensatz zur Standardabweichung hat der Variationskoeffizient keine Einheit und ist somit unabhängig von der gemessenen Größe. Er eignet sich daher auch gut, um die Fairness von zwei Locks zu vergleichen, die eine sehr unterschiedliche Performance und damit auch sehr unterschiedliche Standardabweichungen des Programmfortschritts haben.

4.2 Evaluation bestehender Locks

Zunächst werden die aktuell auf verteiltem Speicher zur Verfügung stehenden Locks evaluiert, da diese die Basis darstellen, gegen die alle portierten Locks verglichen werden müssen:

1. „MPI“: Der Lock der MPI-Implementierung: `MPI_Win_lock` mit einem `MPI_Get` und `MPI_Win_flush` (siehe Abschnitt 3.1).
2. „DASH“: `dash::Mutex` aus [ZMI⁺14]. Hierbei handelt es sich um eine Variante eines MCS-Locks mit Punkt-zu-Punkt-Kommunikation für die Übergabe des Locks an den Nachfolger (siehe Abschnitt 3.5).
3. „D-MCS“: Die Implementierung eines MCS-Locks aus [SBH16], erweitert um `MPI_Iprobe` (siehe Abschnitt 3.3).
4. „RMA-MCS“: Die Implementierung eines HMCS-Locks [CFMC15] aus [SBH16] erweitert um `MPI_Iprobe` (siehe Abschnitt 3.3).

Da es bisher kaum Forschung zu Locks in verteiltem Speicher gibt, gibt es hierfür auch keine umfangreiche Benchmarksuite. Daher wurden Szenarien aus [SBH16] und einigen Arbeiten über NUMA-Locks auf verteilten Speicher übertragen und angepasst. Der Quelltext der Benchmarks und Locks aus dieser Arbeit ist auf GitHub veröffentlicht:

<https://github.com/Adrodoc/distributed-locks>

4.2.1 Performance freier Locks (UPB)

Für Anwendungen, in denen nur selten mehrere Prozesse gleichzeitig einen kritischen Abschnitt betreten möchten, ist es besonders wichtig, dass ein freier Lock sehr schnell akquiriert und wieder freigegeben werden kann. Dies wird mit dem *Uncontested Performance Benchmark* (UPB) aus [RH02] gemessen.

Die meisten Locks haben einen Hauptprozess, auf dessen Speicher beim Akquirieren immer zugegriffen wird. Das ist z. B. beim MCS-Lock der Prozess, in dessen Speicher der Zeiger liegt, der auf das Ende der Warteschlange zeigt. Daher macht es einen Unterschied, ob der freie Lock von dem Hauptprozess akquiriert wird, von einem anderen Prozess auf demselben Knoten (dem Hauptknoten) oder von einem Prozess auf einem anderen Knoten. Im letzten Fall ist für das Akquirieren sogar ein Zugriff auf entfernten Speicher nötig. Beim RH-Lock (siehe Abschnitt 5.2) macht es außerdem einen Unterschied, welcher Prozess zuletzt den Lock akquiriert hatte. Damit ergeben sich insgesamt die neun Szenarien in Tabelle 4.1.

Vorgänger	Hauptprozess	Selber Prozess	Selber Knoten	Anderer Knoten
Selber Prozess	1a	1b	1c	
Selber Knoten	2a	2b	2c	
Anderer Knoten	3a	3b	3c	

Tabelle 4.1: UPB-Szenarien

Anders als die anderen Benchmarks in dieser Arbeit nutzt der UPB nicht einen Lock, der in einer Schleife immer wieder verwendet wird, sondern viele Locks, die für eine Messung jeweils nur einmal verwendet werden. So wird sichergestellt, dass die Locks frei sind und je nach Szenario den entsprechenden Vorgänger hatten. In dieser Arbeit werden hierfür 1000 Locks verwendet. Um alle Szenarien durchführen zu können, werden 4 Prozesse benötigt, von denen jeweils 2 auf demselben Knoten laufen müssen. Da die Locks im UPB immer frei sind, ist die Konkurrenz immer 0 und es kann keine Fairness gemessen werden. Im UPB wird daher nur die Geschwindigkeit untersucht.

```

1      // Aufwärmen
2      acquire_and_release_all_locks(locks);
3      MPI_Barrier(comm);
4
5      // Mache Prozess 4 zum Vorgänger
6      if (rank == 4)
7          acquire_and_release_all_locks(locks);
8          MPI_Barrier(comm);
9
10     // Szenario 3a:
11     if (rank == 0)
12         dauer_3a = acquire_and_release_all_locks(locks);
13         MPI_Barrier(comm);
14
15     // Szenario 1a: ...

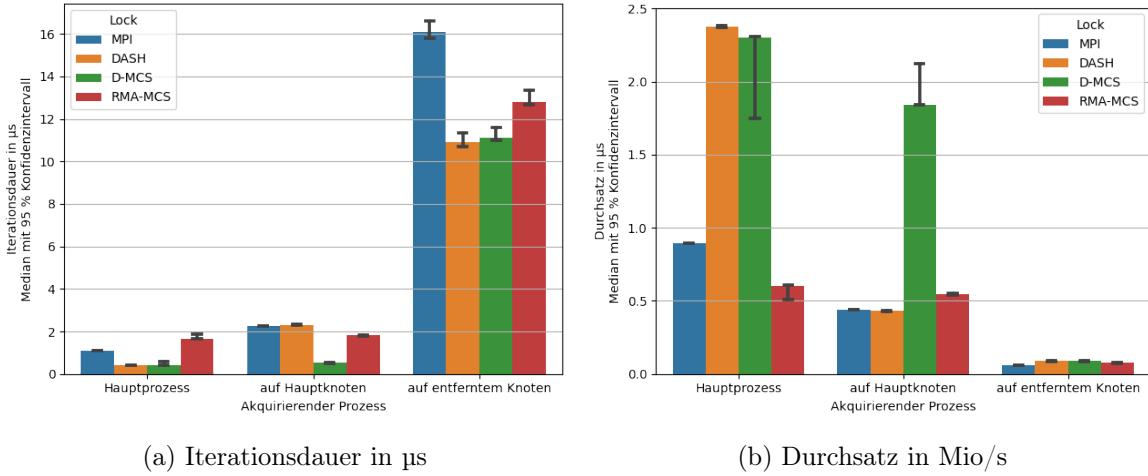
```

Abbildung 4.1: Anfang vom Quelltext des UPBs (Vollständiger Quelltext)

Abbildung 4.1 zeigt den Anfang vom Quelltext des UPBs. In diesem Beispiel hat der Hauptprozess aller Locks $rank = 0$ und läuft zusammen mit dem Prozess mit $rank = 1$ auf dem Hauptknoten. Zum Aufwärmen werden zunächst alle Locks von allen Prozessen einmal

4.2 Evaluation bestehender Locks

akquiriert und wieder freigegeben. Anschließend akquiriert der Prozess mit $rank = 4$ alle Locks, damit alle Locks einen definierten Vorgänger haben. Schließlich werden die Szenarien in folgender Reihenfolge ausgeführt: 3a, 1a, 2b, 1b, 2a, 3c, 1c, 2c, 3b. Die Reihenfolge ist so gewählt, dass in jedem Szenario die Locks den korrekten Vorgänger hatten.



Benchmark 4.1: UPB der Basislocks

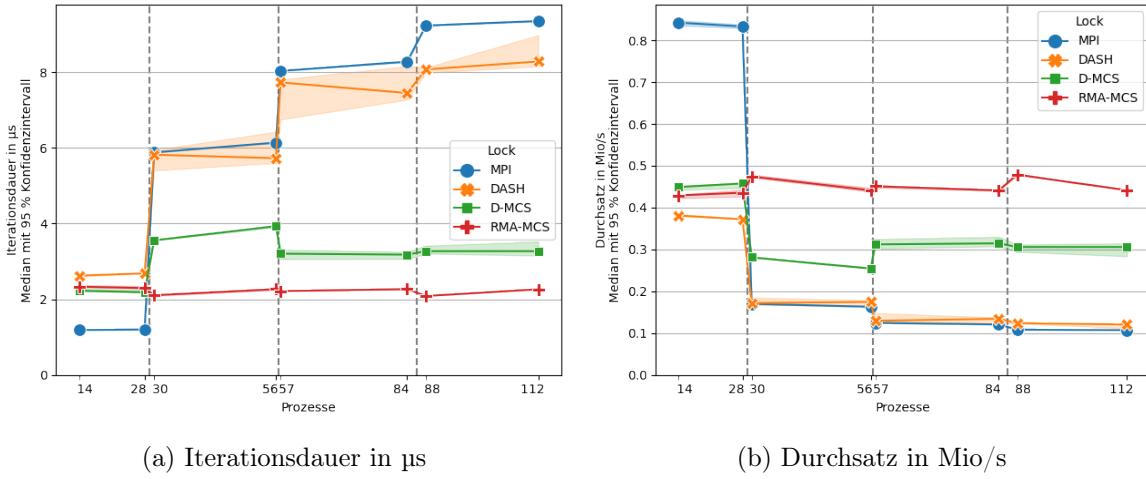
Benchmark 4.1a zeigt, wie lange es durchschnittlich dauert, einen freien Lock zu akquirieren. Benchmark 4.1b hingegen zeigt umgerechnet, wie viele Millionen freie Locks pro Sekunde akquiriert werden können. Da es für diese Locks keinen Unterschied macht, welcher Prozess der Vorgänger war, sind in den beiden Abbildungen die Messungen unabhängig vom Vorgänger zusammengefasst. Die Konfidenzintervalle beziehen sich hier somit auf 24 Messungen, statt wie sonst auf 8.

In den Abbildungen sieht man deutlich den Einfluss von entfernten Speicherzugriffen auf die Performance. Wenn der Hauptprozess des Locks auf einem anderen Knoten läuft als der Prozess, der den Lock akquiriert und freigibt, sind alle Locks deutlich langsamer. Beim Intel-MPI-Lock und `dash::Mutex` macht es darüber hinaus einen großen Unterschied, ob der Prozess selbst der Hauptprozess oder ein anderer Prozess auf dem Hauptknoten ist. Der Grund hierfür ist leider nicht bekannt. Zumindest bei `dash::Mutex` liegt es jedenfalls nicht an dem verwendeten Algorithmus, denn sowohl `dash::Mutex` als auch D-MCS müssen in diesem Benchmark nur zwei MPI-Operationen ausführen: ein `MPI_Fetch_and_op` zum Akquirieren und ein `MPI_Compare_and_swap` zum Freigeben (jeweils mit `MPI_Win_flush`). In Abschnitt 4.3 wird gezeigt, dass eine Reimplementierung des Algorithmus aus `dash::Mutex` dieses Verhalten nicht aufweist.

Außerdem zeigt der UPB, dass ein freier RMA-MCS-Lock immer etwas langsamer ist als ein freier D-MCS-Lock. Da RMA-MCS eine hierarchische Version des D-MCS mit zwei Hierarchieebenen ist, muss bei einem freien Lock sowohl der lokale Lock als auch der globale Lock akquiriert werden, während D-MCS nur aus einem globalen Lock besteht. Benchmark 4.1a zeigt allerdings, dass der Overhead von RMA-MCS gegenüber D-MCS nur bei etwa 1,5 μs liegt und zwar unabhängig davon, auf welchem Knoten der Prozess läuft.

4.2.2 Leerer kritischer Abschnitt (ECSB)

Der Benchmark mit leerem kritischen Abschnitt (engl. *Empty Critical Section Benchmark* (ECSB)) ist sehr verbreitet [RH02] [LNS06] [DMS11] [CFMC15], vermutlich, weil er sehr einfach ist. In einer Schleife akquiriert jeder Prozess den Lock und gibt ihn direkt wieder frei. Das ist zwar nicht sehr realistisch, dafür beeinflusst aber nur der Lock selbst die Messung.



(a) Iterationsdauer in μs

(b) Durchsatz in Mio/s

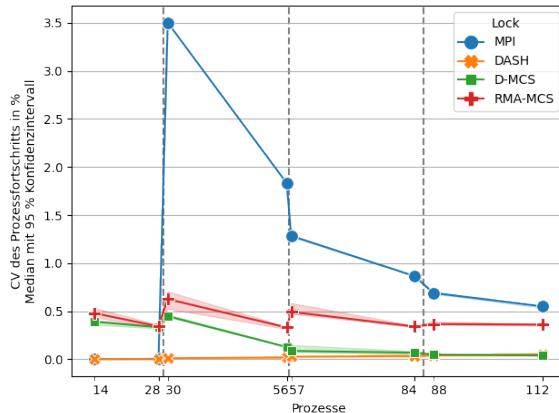
Benchmark 4.2: ECSB der Basislocks (Vollständiger Quelltext)

Benchmark 4.2 zeigt die Performance der Locks im ECSB. Man sieht, dass in diesem Benchmark der Lock aus Intel-MPI bei Weitem am schnellsten ist, solange nur ein Rechenknoten beteiligt ist. Hier besteht offenbar großes Optimierungspotenzial bei den anderen Locks. Jedes Mal, wenn ein weiterer Rechenknoten hinzukommt (bei 30, 57 und 88 Prozessen) nimmt die Geschwindigkeit des MPI-Locks jedoch stark ab. `dash::Mutex` zeigt ein sehr ähnliches Verhalten. Hier ist allerdings auffällig, dass dieser Lock hohe Schwankungen aufweist. D-MCS und RMA-MCS sind bei mehreren Knoten deutlich schneller, da sie direkte Speicherzugriffe statt RMA-Operationen in den Warteschleifen verwenden (siehe Abschnitt 3.3).

Es wird auch deutlich, dass Zugriffe auf entfernten Speicher vermieden werden sollten. D-MCS unterscheidet nicht zwischen Prozessen auf demselben und anderen Knoten. Die Performance nimmt daher bei 30 Prozessen deutlich ab, da dann mehr als ein Rechenknoten beteiligt ist, bleibt aber bei weiteren Knoten relativ konstant. RMA-MCS verhält sich als einzige Implementierung stabil, unabhängig von der Anzahl der Prozesse und Rechenknoten.

Da beim ECSB außerhalb des kritischen Abschnitts nichts gemacht wird, versuchen ständig Prozesse den kritischen Abschnitt zu betreten und der Lock ist nie frei. Die Konkurrenz (nicht gezeigt) ist daher bei allen Locks unabhängig von der Anzahl an Prozessen bei 100 %.

Benchmark 4.3 zeigt die Fairness der Locks im ECSB. Obwohl `dash::Mutex` und D-MCS beides Varianten des MCS-Locks sind und theoretisch FIFO garantieren, weist D-MCS eine etwas schlechtere Fairness auf, die bei mehr und mehr Prozessen aber zu `dash::Mutex` aufholt. RMA-MCS hat theoretisch eine beschränkte Unfairness von 50, d. h. es können maximal 50 andere Prozesse den ersten Prozess in der Warteschlange überholen. Diese schlechtere theoretische Fairness könnte auch der Grund für die etwas schlechtere praktische Fairness sein. Der Intel-MPI-Lock ist bei mehreren Knoten mit Abstand am unfairsten, alle Locks sind mit einem Variationskoeffizienten von deutlich unter 5 % allerdings ziemlich fair.



Benchmark 4.3: Fairness der Basislocks im ECSB: CV des Fortschritts in %

4.2.3 Änderung des kritischen Arbeitsanteils (CCWB)

Einen deutlich realistischeren Ansatz hat der *Changing Critical Work Benchmark* (CCWB). Auch dieser wird in verschiedenen Varianten häufig in der Literatur verwendet [RH02] [LNS06] [DMS11] [DMS12] [LDT⁺12] [CFMC15] [ZLW⁺16]. Die Idee bei diesem Benchmark ist es, die Konkurrenz zwischen den Prozessen zu erhöhen, indem ein immer größerer Anteil der Arbeit im kritischen Abschnitt ausgeführt wird, bis diese den Hauptteil der Rechenzeit beansprucht. Zunächst ist der kritische Abschnitt leer. Anders als beim ECSB führen Prozesse aber vor dem Akquirieren des Locks einige Speicherzugriffe aus. Diese Zugriffe repräsentieren den parallelisierbaren Anteil eines Programms und werden im Folgenden als unkritische Arbeit u bezeichnet. Dann werden mehr und mehr Operationen im kritischen Abschnitt ausgeführt, wodurch ein immer größerer Anteil des Programms serialisiert werden muss und immer mehr Prozesse gleichzeitig den kritischen Abschnitt ausführen wollen. Diese Operationen werden als kritische Arbeit k bezeichnet. Abbildung 4.2 zeigt den Quelltext einer CCWB-Iteration.

```

1 void ccwb(Lock &lock) {
2     lock.acquire();
3     for (int i = 0; i < k; i++)
4         increment_non_atomically(sizeof(int) * i);
5     lock.release();
6     int u = distribution(generator);
7     for (int i = 0; i < u; i++)
8         increment_non_atomically(sizeof(int) * (k + i));
9 }
10
11 void increment_non_atomically(MPI_Aint disp) {
12     // Der „rank“ eines Prozesses, der wahrscheinlich auf einem anderen Knoten läuft.
13     int remote_rank = (my_rank + p / 2) % p;
14     int value;
15     MPI_Get(&value, 1, MPI_INT, remote_rank, disp, 1, MPI_INT, win);
16     MPI_Win_flush(remote_rank, win);
17     value += 1;
18     MPI_Put(&value, 1, MPI_INT, remote_rank, disp, 1, MPI_INT, win);
19     MPI_Win_flush(remote_rank, win);
20 }
```

Abbildung 4.2: Quelltext einer CCWB Iteration (Vollständiger Quelltext)

4 Evaluation und Optimierung bestehender Locks

Ein spannender Punkt ist erreicht, wenn kritische und unkritische Arbeit gleichermaßen an der Laufzeit beteiligt sind. Das ist der Fall, wenn die unkritische Arbeit $p - 1$ Mal so groß ist wie die kritische Arbeit, wobei p die Anzahl der Prozesse ist. Damit stellt die kritische Arbeit einen Anteil von $\frac{1}{p}$ der gesamten Arbeit eines Prozesses dar. Dies ist der letzte Punkt, an dem das Programm theoretisch perfekt parallelisierbar ist und wird im Folgenden als Equilibrium bezeichnet.

Man kann sich vorstellen, dass im Optimalfall immer ein Prozess den kritischen Abschnitt ausführt, während die anderen $p - 1$ Prozesse unkritische Arbeit ausführen. In der Realität ist bereits etwas früher keine perfekte Parallelisierung mehr möglich, da die Prozesse nicht immer zum perfekten Zeitpunkt den kritischen Abschnitt betreten möchten und das Akquirieren und Freigeben des Locks zusätzlich Zeit benötigt.

Echte Anwendungen brauchen nicht zwischen jedem kritischen Abschnitt die gleiche Anzahl an Operationen. Um das zu simulieren wird die tatsächliche Menge der unkritischen Arbeit zufällig bestimmt. Ausgehend von einer minimalen Anzahl an Operationen a bestimmt jeder Prozess in jeder Iteration seine tatsächliche Arbeit \tilde{a} indem er über eine Gleichverteilung eine Zufallszahl aus dem geschlossenen Intervall $[a; 2a]$ bestimmt. Dies geschieht vor dem kritischen Abschnitt und nutzt `std::mt19937`⁴ und `std::uniform_int_distribution`⁵ aus C++. Die unkritische Arbeit u bestimmt sich dann mit $u = \tilde{a} - k$.

Nun soll a so gewählt werden, dass das Equilibrium e im Durchschnitt bei einer kritischen Arbeit von $k = 3$ erreicht ist. So kann bei $k \in [0; 2]$ beobachtet werden, wie der Lock sich verhält, wenn die unkritische Arbeit dominiert und bei $k \in [4; 5]$, wie er sich verhält, wenn die kritische Arbeit dominiert. Da \tilde{a} im Durchschnitt $1,5a$ ist, berechnet sich a mit: $a = p \cdot e / 1,5$, in diesem Fall mit $e = 3$. Für 112 Prozesse wird daher im Folgenden eine minimale Arbeit von $a = 224$ verwendet, wodurch die tatsächliche Arbeit \tilde{a} im Durchschnitt 336 ist.

Da die gesamte Arbeit a gleich bleibt, hätte eine theoretisch perfekte Lock-Implementierung eine durchschnittliche Iterationsdauer von $\max(\frac{1,5a}{p}, k)$, multipliziert mit der Dauer einer Operation. Solange das Equilibrium noch nicht erreicht ist, ist das Programm perfekt parallelisierbar, hat also eine Iterationsdauer von $\frac{1,5a}{p}$ ($\frac{\text{Durchschnittliche Arbeit}}{\text{Anzahl Prozesse}}$). Sobald das Equilibrium erreicht ist, gilt $\frac{k}{1,5a} = \frac{1}{p} \iff \frac{1,5a}{p} = k$. Die kritische und unkritische Arbeit tragen dann gleichermaßen zur Laufzeit bei. Nach dem Equilibrium hängt die Iterationsdauer linear von k ab.

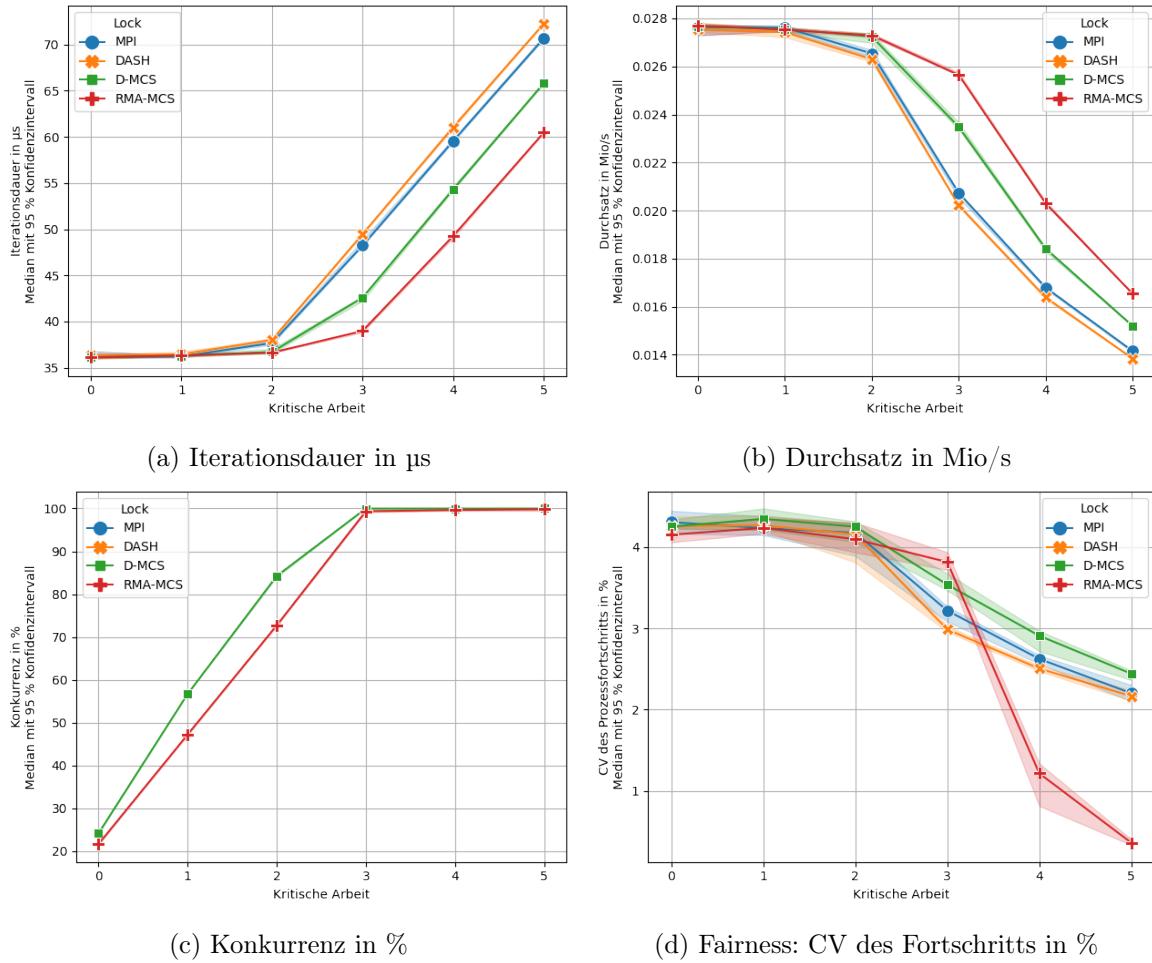
Als Operation wird sowohl im kritischen Abschnitt als auch außerhalb mit `MPI_Get` auf einen Speicherbereich eines Prozesses auf einem anderen Knoten zugegriffen, der Wert inkrementiert und mit `MPI_Put` zurückgeschrieben. In einem normalen Programm würde man hierfür `MPI_Accumulate` verwenden, um mit nur einem entfernten Zugriff den Wert atomar zu erhöhen, aber da ein Lock normalerweise nicht-atomare Operationen schützt, werden für diesen Benchmark zwei nicht-atomare Operationen verwendet. So werden nacheinander die Werte an \tilde{a} verschiedenen entfernten Speicheradressen inkrementiert.

Wichtig ist dabei, dass nicht alle Prozesse auf dem Speicher desselben Prozesses arbeiten. Ohne RDMA ist der Zielprozess an einer RMA-Operation beteiligt. Wenn dann alle Prozesse ständig auf den Speicher desselben Prozesses zugreifen, ist keine echte Parallelisierung möglich, da dieser Prozess die Anfragen aller Prozesse beantworten muss. Jeder Prozess hat daher einen anderen „Partner“, auf dessen Speicher er arbeitet.

⁴https://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine

⁵https://en.cppreference.com/w/cpp/numeric/random/uniform_int_distribution

4.2 Evaluation bestehender Locks



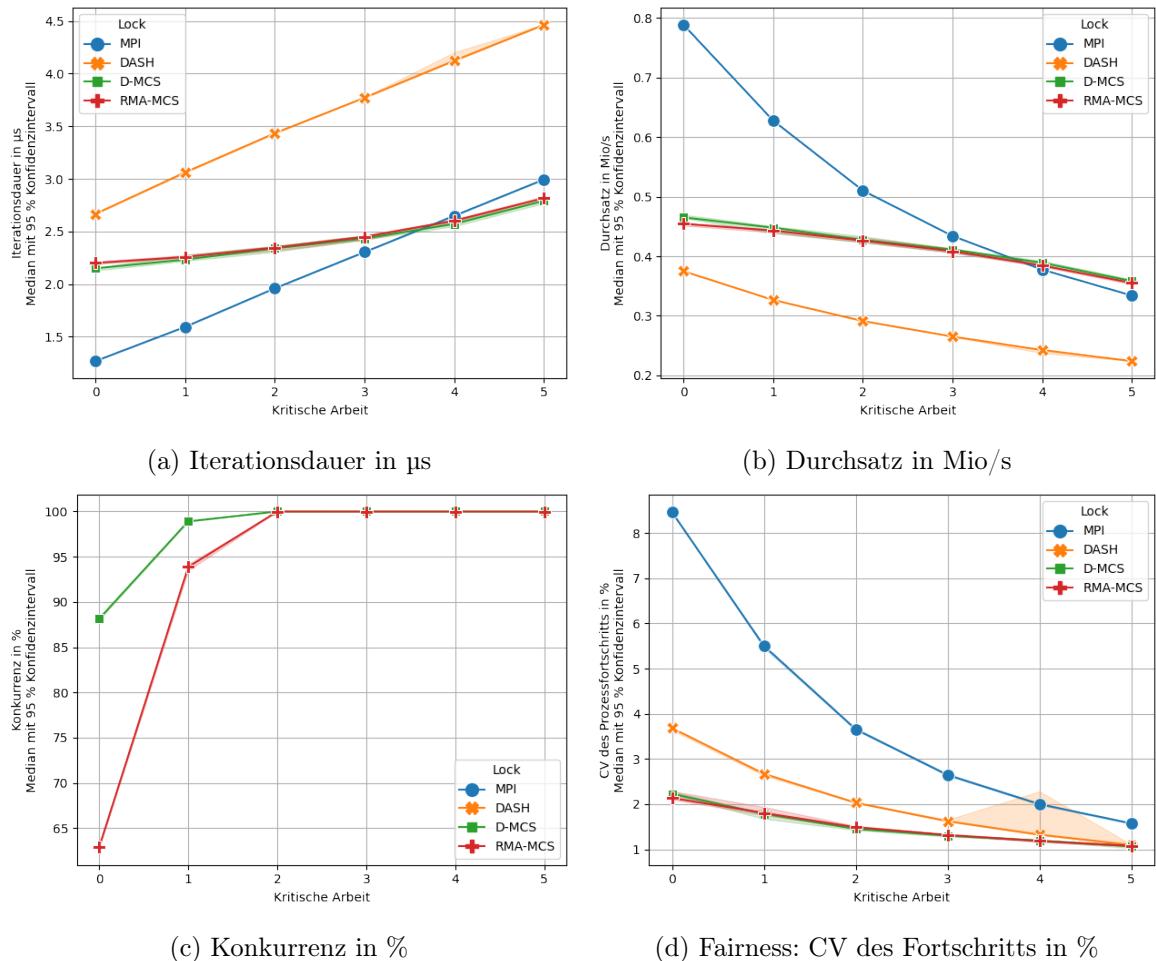
Benchmark 4.4: CCWB der Basislocks mit 112 Prozessen

Benchmark 4.4a und Benchmark 4.4b zeigen die Geschwindigkeit der Basislocks im CCWB bei 112 Prozessen, also vier voll ausgelasteten Knoten. Ähnlich wie beim ECSB liegen der MPI-Lock und `dash::Mutex` nah beieinander, während D-MCS und RMA-MCS deutlich schneller sind. Allerdings ist in diesem Benchmark `dash::Mutex` etwas langsamer als der MPI-Lock. Es ist auch ersichtlich, dass D-MCS und RMA-MCS einen deutlich geringeren Overhead haben als die anderen beiden Locks, da das Equilibrium bei ihnen noch nicht bei einer kritischen Arbeit von 2 einsetzt, sondern erst später. Dies geht auch aus Benchmark 4.4c hervor, auch wenn die Konkurrenz für die Locks aus den Bibliotheken MPI und DASH nicht gemessen werden kann.

Die Fairness der Locks in Benchmark 4.4d ist deutlich schlechter und hat deutlich größere Konfidenzintervalle als beim ECSB in Benchmark 4.3. Der Grund hierfür ist wahrscheinlich, dass durch die geringere Konkurrenz die Locks immer wieder frei werden. Bei freien Locks macht es, wie der UPB in Benchmark 4.1 gezeigt hat, einen großen Unterschied, ob der akquirierende Prozess auf dem Hauptknoten läuft. Solche Prozesse können freie Locks schneller akquirieren, da keine entfernten Speicherzugriffe benötigt werden und sind damit im Durchschnitt etwas schneller, was zu einer höheren Unfairness führt. Das erklärt auch, warum die Fairness deutlich besser wird, sobald das Equilibrium erreicht ist.

4 Evaluation und Optimierung bestehender Locks

Darüber hinaus hat der CCWB einen deutlich größeren Zufallsanteil, da die Arbeit, die jeder Prozess ausführen muss, für jede Iteration aus einem recht großen Intervall bestimmt wird und vollständig aus entfernten Speicherzugriffen besteht. Auch der UPB hat bereits in Benchmark 4.1a gezeigt, dass die Geschwindigkeit von entfernten Speicherzugriffen deutlich höheren Schwankungen unterliegt als die von Lokalen. Das könnte die ungenauen Konfidenzintervalle in Benchmark 4.4d erklären. Trotz der schlechteren Fairness haben alle Locks auch vor dem Equilibrium immer noch einen Variationskoeffizienten von unter 5 % und sind damit relativ fair.



Benchmark 4.5: CCWB der Basislocks mit 28 Prozessen

Um das beim ECSB identifizierte Optimierungspotenzial bei der Nutzung von nur einem Knoten näher zu untersuchen, zeigt Benchmark 4.5 den CCWB mit 28 Prozessen. Da nur ein Knoten an dem Benchmark beteiligt ist, handelt es sich bei den Operationen vor und in dem kritischen Abschnitt nun um lokale Operationen. Da lokale Operationen deutlich schneller als entfernte Operationen sind, sieht man den Overhead der Locks stärker. Dabei macht es einen Unterschied, welcher Anteil dieses Overheads auf dem kritischen Pfad (also quasi im kritischen Abschnitt) und welcher außerhalb liegt.

Overhead außerhalb des kritischen Pfades hat denselben Effekt wie eine höhere unkritische

Arbeit: Die Iterationsdauer wird unabhängig von der Menge der kritischen Arbeit um eine Konstante erhöht. Overhead auf dem kritischen Pfad führt zusätzlich dazu, dass das Equilibrium früher erreicht ist, wodurch die Iterationsdauer früher linear mit der kritischen Arbeit steigt. Selbst wenn der Anteil der kritischen Arbeit klein ist, ist Overhead auf dem kritischen Pfad schlechter, weil es trotzdem passieren kann, dass mehrere Prozesse zufällig gleichzeitig den kritischen Abschnitt betreten möchten, wodurch sich der Overhead auf dem kritischen Pfad auf mehr als einen Prozess auswirken kann. Umso höher der Anteil der kritischen Arbeit ist, umso wahrscheinlicher ist es, dass so ein Fall auftritt und ab dem Equilibrium betrifft der Overhead auf dem kritischen Pfad immer alle Prozesse. Umso mehr Prozesse beteiligt sind, umso höher ist tendenziell der Anteil der kritischen Arbeit, da sich die unkritische Arbeit auf mehr Prozesse verteilen kann (siehe auch *Amdahl's Law* aus [Amd67]). Für eine gute Skalierbarkeit, besonders im HPC-Bereich mit Tausenden von Prozessen, ist daher ein möglichst geringer Overhead vor allem auf dem kritischen Pfad wichtig.

Benchmark 4.5a zeigt deutlich, dass der MPI-Lock und `dash::Mutex` bei nur einem Knoten einen so hohen kritischen Overhead aufweisen, dass dieser auch ohne kritische Arbeit die unkritische Arbeit von $\text{avg}(\tilde{a}) = 84 \text{ MPI_Put- und MPI_Get-Operationen}$ (jeweils mit `MPI_Win_flush`) dominiert. D-MCS und RMA-MCS hingegen erreichen erst bei einer kritischen Arbeit von 1 bzw. 2 das Equilibrium, wie Benchmark 4.5c zeigt. Dafür weisen diese Locks einen deutlich höheren unkritischen Overhead als der MPI-Lock auf, sodass sie bei geringer kritischer Arbeit trotzdem langsamer sind. Überraschenderweise ist das Equilibrium bei D-MCS und RMA-MCS nicht in Benchmark 4.5a zu sehen. Zu erwarten wäre bei den beiden Locks ein plötzlicher starker linearer Anstieg der Iterationsdauer ab einer kritischen Arbeit von 1 bzw. 2, mit derselben Steigung wie bei den anderen beiden Locks. Der kritische Overhead der beiden Locks nimmt anscheinend ab, wenn der kritische Abschnitt größer wird. Dies wird bei der Optimierung des D-MCS in Abschnitt 4.3 näher untersucht.

Bei der Fairness schneidet der MPI-Lock mit einem Variationskoeffizienten von über 8 % deutlich schlechter ab als die anderen Locks, welche weiterhin unter 5 % bleiben, wie Benchmark 4.5d zeigt.

4.2.4 Vor dem Akquirieren warten (WBAB)

Der ECSB hat gezeigt, dass die Basislocks auf verteiltem Speicher so viel kritischen Overhead haben, dass selbst bei leerem kritischen Abschnitt 100 % Konkurrenz herrscht. Daher lässt sich die Konkurrenz auch variieren, indem mehr und mehr Arbeit außerhalb eines leeren kritischen Abschnitts ausgeführt wird. Das hat gegenüber dem CCWB den Vorteil, dass weniger Operationen ausgeführt werden, die zur Ungenauigkeit der Messergebnisse beitragen und dass der Overhead des Locks einen größeren Teil der Laufzeit ausmacht, wodurch er besser zu sehen ist. Der Nachteil ist, dass dieser Benchmark nicht so realistisch ist wie der CCWB, da ein leerer kritischer Abschnitt in der Praxis keinen Sinn macht.

Beim *Wait Before Acquire Benchmark* (WBAB) warten Prozesse vor dem Akquirieren des Locks für eine zufällig bestimmte Zeit in einer Schleife (Vollständiger Quelltext). Diese Zeit wird wie beim CCWB aus dem geschlossenen Intervall $[w; 2w]$ bestimmt. Dabei ist w (die minimale Wartezeit in Nanosekunden) ein Parameter des Benchmarks. Die tatsächliche Wartezeit \tilde{w} ist damit im Durchschnitt $1,5w$. Der kritische Abschnitt ist wie beim ECSB leer. Anders als beim CCWB werden in der Wartezeit vor dem kritischen Abschnitt keine MPI-Operationen ausgeführt. Das ermöglicht zum einen eine beliebige Präzision, da die Wartezeit auch in Schritten erhöht werden kann, die kleiner sind als eine MPI-Operation dauert und

4 Evaluation und Optimierung bestehender Locks

zum anderen eine genaue Bestimmung des Overheads bei Messung der Iterationsdauer, da die Wartezeit in einer Zeiteinheit angegeben ist, statt in einer Anzahl von Operationen.

Der WBAB in dieser Form kommt nicht in der Literatur über NUMA-Locks vor. Der Grund dafür ist wahrscheinlich, dass auf gemeinsamem Speicher der Overhead der Locks geringer ist, wodurch die Konkurrenz bei einem leeren kritischen Abschnitt schon bei geringer Wartezeit stark abnimmt. Bei einer Änderung der Wartezeit im Bereich von wenigen Nanosekunden würde dann bereits die Ermittlung der Systemzeit das Messergebnis zu sehr verfälschen. Außerdem spielt bei gemeinsamem Speicher häufig der Zwischenspeicher-Kohärenz-Mechanismus eine große Rolle und es ist wichtig zu messen, ob der Speicher, auf den im kritischen Abschnitt zugegriffen wird, bereits im Zwischenspeicher liegt. Das ist besonders bei delegierenden Locks ein wichtiger Faktor (siehe Abschnitt 5.1). In [DMS11], [DMS12] und [CFMC15] werden ähnliche Benchmarks mit einer festen Anzahl an Operationen im kritischen Abschnitt verwendet. Nur [SBH16] behandelt verteilten Speicher und nutzt einen sehr ähnlichen Benchmark mit leerem kritischen Abschnitt und Wartezeit nach dem Freigeben des Locks. All diese Benchmarks variieren aber die Anzahl der Prozesse und nicht die Wartezeit. Sie nutzen eine willkürlich gewählte Wartezeit, die unabhängig von der Anzahl der Prozesse ist.

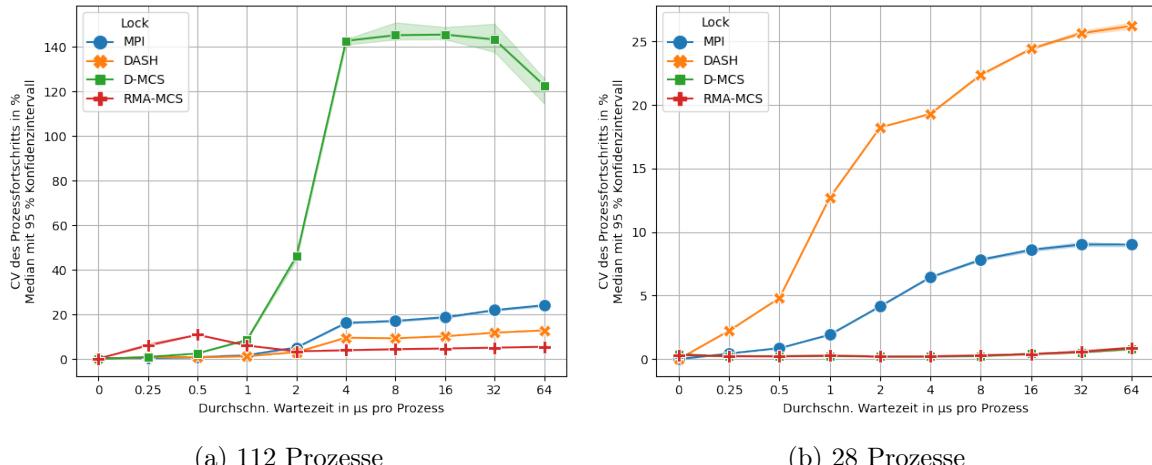
Da bei verteiltem Speicher deutlich mehr Prozessoren zur Verfügung stehen als bei gemeinsamem Speicher, sollte der Benchmark auch mit beliebig vielen Prozessen ausgeführt werden können. Daher wird beim WBAB, wie beim CCWB, die Wartezeit abhängig von der Anzahl von Prozessen bestimmt. Zunächst wird eine minimale Wartezeit von $w = 0$ verwendet, um bei maximaler Konkurrenz zu beginnen. Dieses Szenario ist identisch zum ECSB. Anschließend wird w so gewählt, dass der Durchschnitt der tatsächlichen Wartezeit \tilde{w} , ausgehend von $0,25 \mu\text{s} \cdot p$ in jedem Schritt verdoppelt wird. p ist hierbei wieder die Anzahl der Prozesse. Durch das exponentielle Wachstum von \tilde{w} kann fast das gesamte Spektrum der Konkurrenz untersucht werden. Die Konkurrenz nimmt nämlich bei zunehmender Wartezeit immer langsamer ab, was durch das schnelle Wachstum ausgeglichen wird.

Wenn die Prozesse in der Wartezeit einfach nur in einer Schleife die Zeit prüfen, garantiert MPI in dieser Zeit keinen Fortschritt für andere Prozesse, die auf den Speicher dieses Prozesses zugreifen wollen, da ohne RDMA das Betriebssystem des Zielrechners an der Kommunikation beteiligt sein muss (vgl. Abschnitt 3.3). Das ist vor allem ein Problem, wenn der wartende Prozess der Hauptprozess eines Locks ist, da die meisten Locks bei jeder Akquisition des Locks auf den Speicher dieses Prozesses zugreifen.

Wie Benchmark 4.6a zeigt, führt das zu extremer Unfairness. Dabei sind die Locks verschieden schwer betroffen: D-MCS erreicht bei 112 Prozessen einen Variationskoeffizienten von mehr als 140 %. Das bedeutet die Standardabweichung ist über 40 % höher als der Durchschnitt an ausgeführten Iterationen. Die minimale Anzahl und der Median an ausgeführten Iterationen bei einer durchschnittlichen Wartezeit von 4 μs waren bei D-MCS im Durchschnitt der acht Wiederholungen gerade einmal 99.25 und 102.75, während der schnellste Prozess im Durchschnitt 1966 Iterationen schaffte. Das liegt daran, dass sich in der Wartezeit nur der Hauptprozess (evtl. auch die anderen Prozesse auf dem Hauptknoten) in die Warteschlange einreihen konnte. Der Hauptprozess muss nicht nur nicht warten, um sich in die Warteschlange einzureihen, sondern hat im Durchschnitt auch weniger Vorgänger in der Warteschlange, da sich in seiner Wartezeit kein anderer Prozess einreihen konnte.

`dash::Mutex` ist hiervon weniger stark betroffen, obwohl dieser auch eine Variante des MCS-Locks ist. Hier wird beim Freigeben des Locks immer eine CAS-Operation ausgeführt. Auch diese kann nicht ausgeführt werden, während der Hauptprozess wartet, sodass in dieser

4.2 Evaluation bestehender Locks



Benchmark 4.6: Fairness im WBAB ohne MPI_Iprobe: CV des Fortschritts in %

Zeit gar kein Prozess kritischen Fortschritt machen kann. Das bedeutet aber auch, dass sich die Warteschlange in dieser Zeit nicht abbaut, wodurch auch der Hauptprozess im Durchschnitt mehr Vorgänger hat. Dadurch, dass der Hauptprozess sich länger in der Warteschlange befindet, gibt es auch ein größeres Zeitintervall, in dem Prozesse sich in die Warteschlange einreihen können. `dash::Mutex` wird also nicht so unfair wie D-MCS, leidet aber auch unter massiven Einbußen bei der Geschwindigkeit.

RMA-MCS kommt von allen Locks am besten mit der Situation klar. Da dieser Lock NUMA berücksichtigt, ist nicht bei jeder Akquisition des Locks ein Zugriff auf den Hauptprozess notwendig. Dadurch hat RMA-MCS weniger Einbußen, sowohl bei Geschwindigkeit als auch Fairness. Nur bei einer durchschnittlichen Wartezeit von 0,5 μ s steigt der Variationskoeffizient bei RMA-MCS knapp über 10 %.

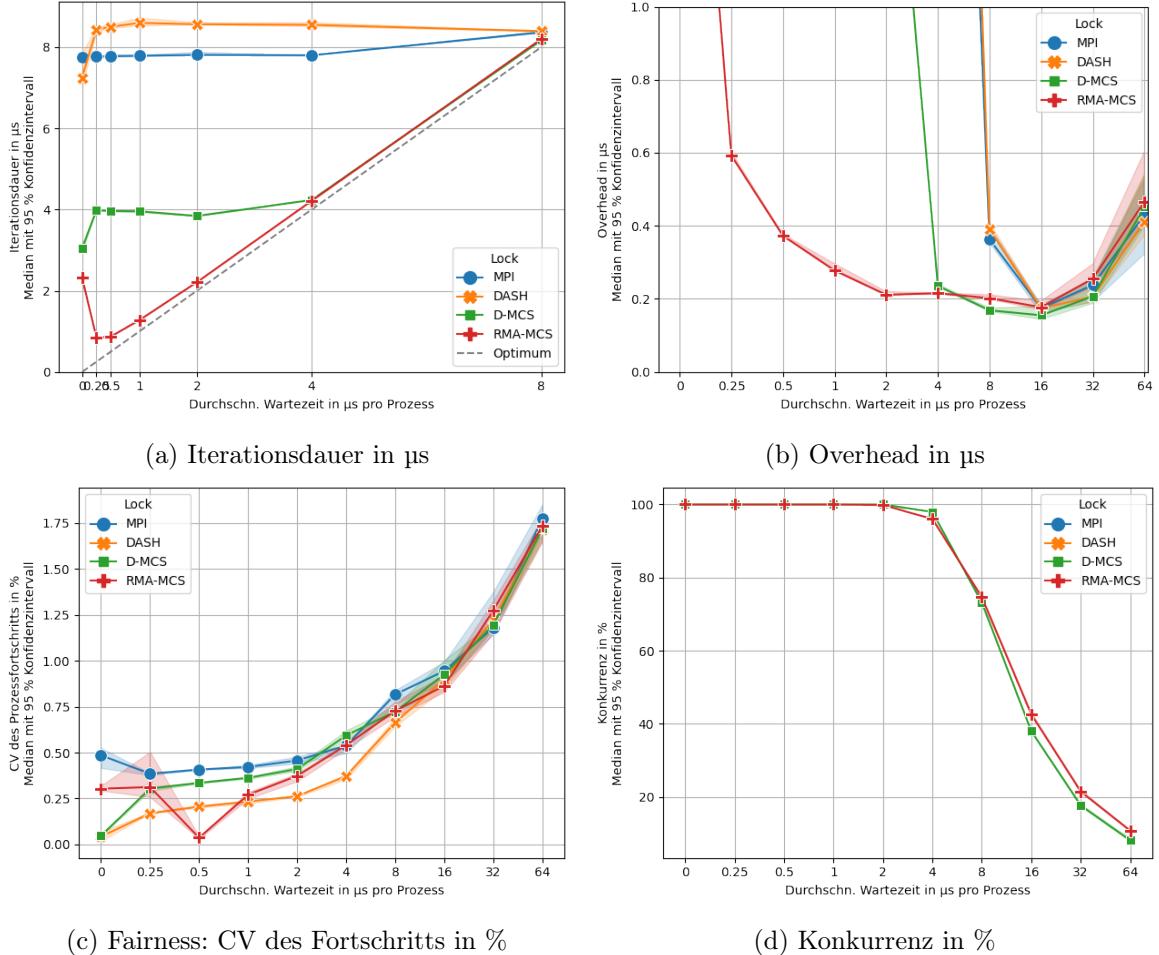
Bei 28 Prozessen, also auf nur einem Rechenknoten ist das Bild deutlich anders (vgl. Benchmark 4.6b). Da es keine Zugriffe auf entfernten Speicher gibt, gibt es auch keine Probleme mit Fortschrittsgarantien. Trotzdem haben `dash::Mutex` und der Intel-MPI-Lock eine schlechte Fairness. Der Grund hierfür ist unbekannt, allerdings verschwindet dieser Effekt bei einer Reimplementierung des Algorithmus aus `dash::Mutex` in Abschnitt 4.3. Das erinnert an die höhere Latenz beim Akquirieren eines freien Locks durch einen nicht-Hauptprozess auf dem Hauptknoten im UPB. Auch dort gab es bei `dash::Mutex` und dem MPI-Lock einen Unterschied zwischen dem Hauptprozess und anderen Prozessen auf demselben Knoten, nicht aber bei D-MCS und RMA-MCS. Und auch dieses Verhalten verschwindet bei einer Reimplementierung von `dash::Mutex`.

Das Szenario ist insgesamt durch den leeren kritischen Abschnitt und der vollständigen Abwesenheit von entfernten Speicherzugriffen außerhalb des kritischen Abschnitts nicht sehr realistisch. Trotzdem ist es in echten Anwendungen durchaus möglich, dass Prozesse außerhalb von kritischen Abschnitten vor allem lokale Berechnungen durchführen, die ohne MPI-Operationen auskommen. In solchen Fällen wäre, genau wie in diesem Benchmark, kein Programmfortschritt möglich. Daher sollten bei der Verwendung von RMA in MPI-Anwendungen lange Abschnitte ohne MPI-Operationen vermieden werden. Wie in Abschnitt 3.3 gezeigt, garantiert nicht jede MPI-Operation Fortschritt. Die Garantie hängt von der MPI-Implementierung ab: Weder in Intel-MPI, noch in Open-MPI löst `MPI_Win_sync`

4 Evaluation und Optimierung bestehender Locks

das Problem, in Open-MPI noch nicht einmal `MPI_Win_flush`. Locks, die NUMA berücksichtigen, sind hierdurch gegenüber anderen Lock-Algorithmen gleich doppelt im Vorteil. Sie vermeiden neben langsam entfernten Speicherzugriffen bei hinreichender Konkurrenz auch solche Fortschrittsprobleme.

Um die Probleme mit dem Programmfortschritt in WBAB zu lösen, führen alle Prozesse in der Warteschleife `MPI_Iprobe` mit `MPI_ANY_SOURCE` und `MPI_ANY_TAG` aus. Es würde zwar reichen, wenn nur der Hauptprozess das tun würde, aber im Benchmark sollen alle Prozesse fair behandelt werden. Die Ergebnisse dieser beiden Varianten unterscheiden sich tatsächlich kaum.



Benchmark 4.7: WBAB der Basislocks mit 112 Prozessen

Benchmark 4.7 zeigt die Performance der Locks im korrigierten WBAB. Dabei sieht man in Benchmark 4.7c, dass die Fairnessprobleme durch `MPI_Iprobe` behoben wurden: Alle Prozesse sind fair, mit Variationskoeffizienten von deutlich unter 2 %. Mit steigender Wartezeit steigt auch die Unfairness, da Prozesse auf dem Hauptknoten etwas schneller sind. Wenn die Konkurrenz sinkt, gibt es seltener eine Warteschlange, die Fairnessgarantien durchsetzt.

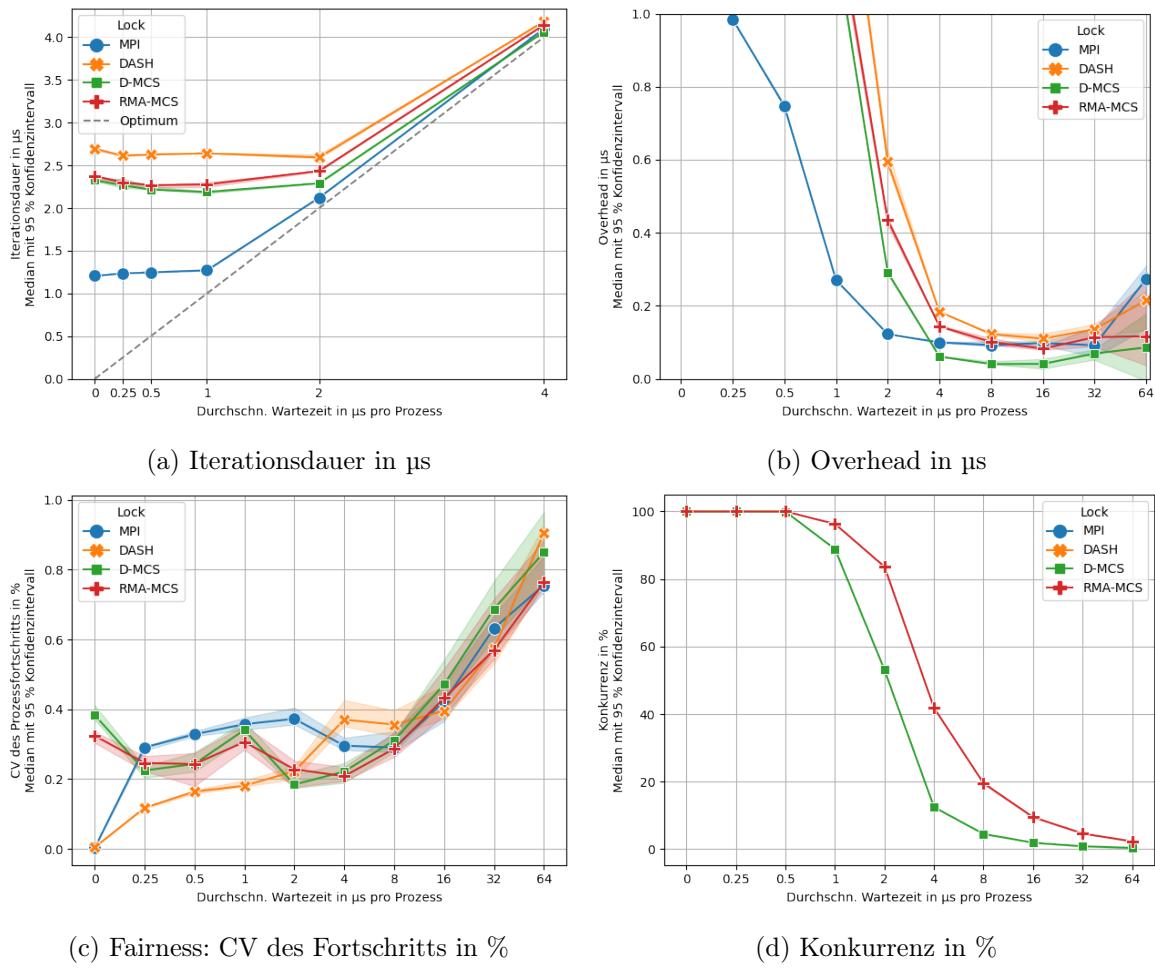
Da in Benchmark 4.7a sowohl die x-, als auch die y-Achse Mikrosekunden zeigen, kann der kritische Overhead im WBAB quantifiziert werden. Er entspricht der Iterationsdauer vor

4.2 Evaluation bestehender Locks

Eintreten des Equilibrium. Das liegt daran, dass vor Eintreten des Equilibrium die kritische Arbeit dominiert und diese besteht im WBAB nur aus dem kritischen Overhead des Locks, da der kritische Abschnitt leer ist. Da der kritische Overhead unabhängig von der Wartezeit ist, bleibt auch die Iterationsdauer vor dem Einsetzen des Equilibrium relativ konstant. Nur bei einer durchschnittlichen Wartezeit von 0 μ s gibt es größere Schwankungen.

Genau wie der CCWB, zeigt der WBAB, dass der kritische Overhead vom MPI-Lock und `dash::Mutex` deutlich höher ist als der von D-MCS und RMA-MCS. Erstere haben einen kritischen Overhead von knapp unter, bzw. etwas über 8 μ s. Der kritische Overhead von D-MCS und RMA-MCS ist deutlich niedriger: bei etwa 4 μ s bzw. knapp unter 1 μ s.

Zusätzlich enthält Benchmark 4.7a das theoretische Optimum, welches der durchschnittlichen Wartezeit entspricht. Da bei WBAB das theoretische Optimum bekannt ist, kann der reine Overhead der Locks gezeigt werden, indem das Optimum von der Iterationsdauer abgezogen wird. Dieser Overhead wird in Benchmark 4.7b gezeigt. Im Gegensatz zu Benchmark 4.7a, geht hier die durchschnittliche Wartezeit bis 64 μ s pro Prozess, wodurch die Konkurrenz auf etwa 10 % fällt (vgl. Benchmark 4.7d). Eine noch höhere Wartezeit würde zwar die Konkurrenz noch weiter senken, aber eine Wartezeit von $64 \mu\text{s} \cdot 112 \approx 7 \text{ ms}$ ist



Benchmark 4.8: WBAB der Basislocks mit 28 Prozessen

4 Evaluation und Optimierung bestehender Locks

bereits so groß, dass das Messergebnis ungenau wird, wie die Konfidenzintervalle zeigen. Bei einer Laufzeit von einer Sekunde, wobei 10 % auch noch zum Aufwärmen verwendet werden, schafft jeder Prozess bei dieser Wartezeit nur noch etwa 125 Iterationen. Hinzu kommt, dass bei sehr geringer Konkurrenz immer mehr Zugriffe auf entfernten Speicher zum Overhead beitragen. Das liegt daran, dass immer mehr freie Locks akquiriert werden müssen und die meisten Prozesse auf einem anderen Knoten laufen als der Hauptprozess. Entfernte Zugriffe haben größere Schwankungen, was die Ungenauigkeit weiter erhöht. Da sie auch langsam sind, nimmt der Overhead ab einer Konkurrenz von 40 % wieder zu.

Die Betrachtung des Overheads auch bei höheren Wartezeiten zeigt, dass auch nach Erreichen des Equilibrium der Overhead zunächst weiter abnimmt und später wieder steigt. Dabei sieht man in Benchmark 4.7b, dass bei geringer Konkurrenz der D-MCS-Lock etwas besser ist als der RMA-MCS-Lock. Genau wie beim UPB sieht man hier den Nachteil der zwei Hierarchieebenen des RMA-MCS-Locks, durch die ein Prozess bei geringer Konkurrenz oft beide MCS-Locks nacheinander akquirieren muss, was natürlich langsamer ist als nur einen MCS-Lock zu akquirieren, so wie es bei D-MCS der Fall ist. Da einer der Locks aber nur lokal ist, ist dieser Nachteil sehr klein.

Der Vollständigkeit halber zeigt Benchmark 4.8 den WBAB mit 28 Prozessen. Wie auch in ECSB und CCWB ist der MPI-Lock bei nur einem Rechenknoten deutlich am schnellsten. D-MCS und RMA-MCS sind nah beieinander auf Platz zwei und drei und `dash::Mutex` am langsamsten. Die Fairness ist wie bei 112 Prozessen mit einem Variationskoeffizienten von deutlich unter 1 % sehr gut.

4.3 Optimierung bestehender Locks

Bevor NUMA-Algorithmen auf verteilten Speicher portiert werden, werden die in Kapitel 3 identifizierten Optimierungen auf die bestehenden Locks angewandt. So können diese bei der Portierung direkt berücksichtigt werden.

Die bestehenden Locks enthalten zwei Varianten, einen Lock weiterzugeben: D-MCS und RMA-MCS nutzen hierfür RMA, während `dash::Mutex` hierfür Punkt-zu-Punkt-Kommunikation nutzt. Um beide Varianten fair evaluieren zu können, werden D-MCS in Unterabschnitt 4.3.1 und `dash::Mutex` in Unterabschnitt 4.3.2 optimiert.

In diesem Abschnitt wird der WBAB mit linear statt exponentiell steigender Wartezeit ausgeführt, da für die Vergleiche vor allem die Iterationsdauer interessant ist und so eine höhere Auflösung bei geringen Werten vorliegt.

4.3.1 Optimierung von D-MCS

Der D-MCS-Lock ist bereits ein sehr optimierter Lock. Dennoch wurden in dieser Arbeit einige Verbesserungen identifiziert, die bereits in Abbildung 3.5 gezeigt wurden.

Erstens nutzt D-MCS nach Aufrufen von `MPI_Accumulate` kein `MPI_Win_flush` und auch keine sonstige Synchronisierungsfunktion. Das betrifft zum einen das Benachrichtigen des Vorgängers beim Einreihen in die Warteschlange und zum anderen die letzte Operation beim Weitergeben des Locks an einen Nachfolger. Ohne Synchronisierungsfunktion ist aber nicht garantiert, dass diese Operation überhaupt ausgeführt wird:

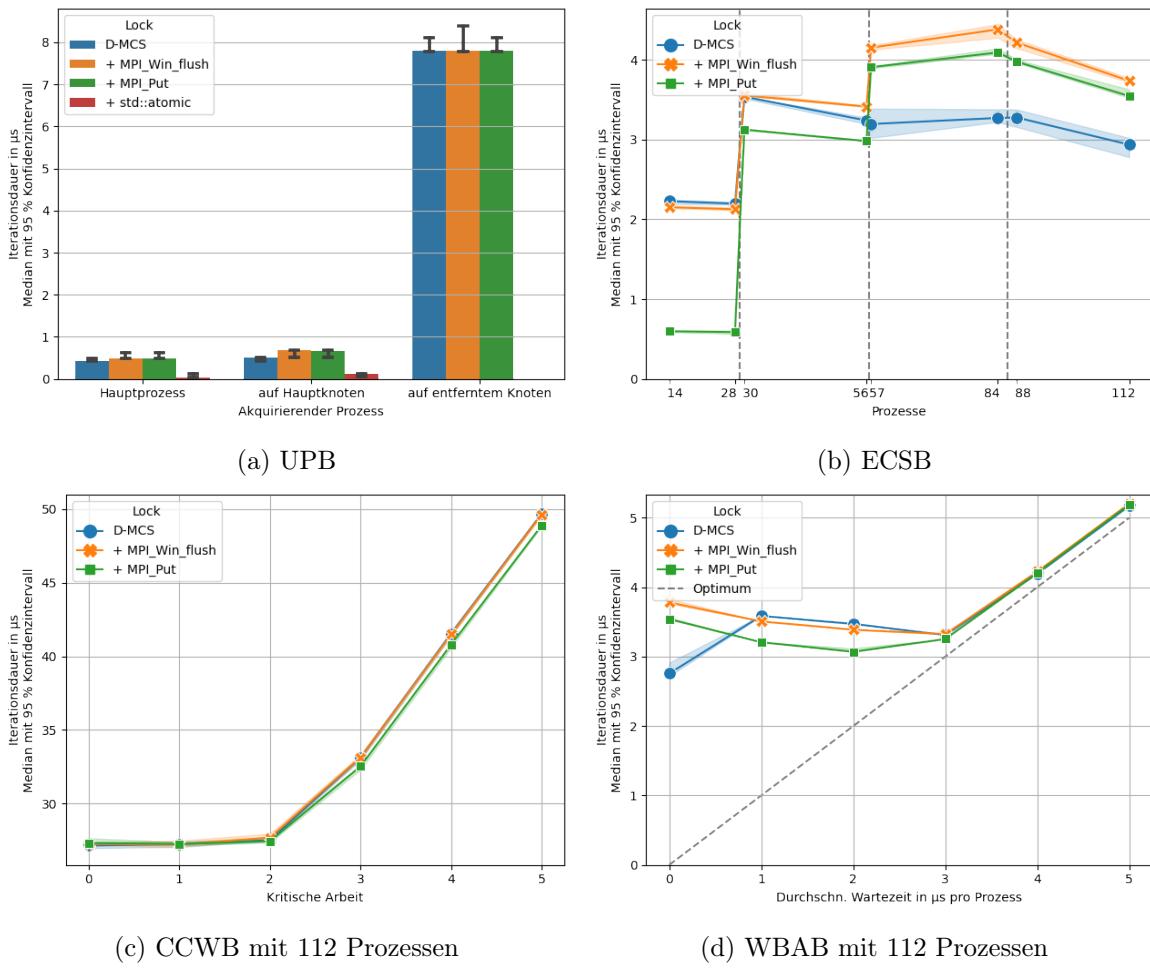
4.3 Optimierung bestehender Locks

[A]n RMA communication may become enabled as soon as the corresponding put, get or accumulate call has executed, or as late as when the ensuing synchronization call is issued. [MPI15, Kapitel 11.7.3, S. 462]

Außerdem verletzt das MPI_Accumulate, welches den Lock an den Nachfolger übergibt, die MPI-Spezifikation:

[RMA] operations are nonblocking: the call initiates the transfer, but the transfer may continue after the call returns. The transfer is completed, at the origin or both the origin and the target, when a subsequent synchronization call is issued by the caller on the involved window object [...]. The local communication buffer of an RMA call should not be updated [...] after the RMA call until the operation completes at the origin. [MPI15, Kapitel 11.3, S. 417]

Da es die letzte Operation in der release-Funktion ist, wird nach dieser Operation der Speicher auf dem Stack freigegeben, der an MPI_Accumulate übergeben wurde. Je nachdem, wann die MPI-Implementierung auf diesen Speicher zugreift, ist der Speicher also invalide.



Benchmark 4.9: Iterationsdauer der D-MCS-Optimierungen in μ s

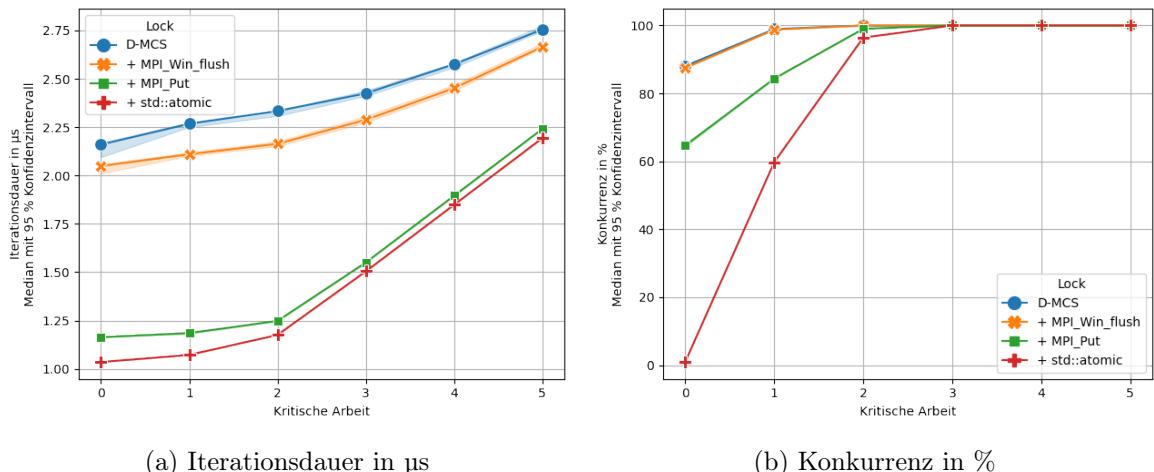
4 Evaluation und Optimierung bestehender Locks

Als Korrektur und erste Optimierung werden daher beide `MPI_Accumulate`-Operationen direkt im Anschluss mit `MPI_Win_flush` vollständig ausgeführt. Zwar würde eine lokale Ausführung mit `MPI_Win_flush_local` reichen, aber da sich die Operationen jeweils auf dem kritischen Pfad befinden, ist eine direkte vollständige Ausführung wünschenswert.

Benchmark 4.9 zeigt, dass diese Änderung einen interessanten Effekt hat: Wenn keine Arbeit ausgeführt wird, wie es im ECSB (vgl. Benchmark 4.9b) und WBAB (vgl. Benchmark 4.9d) ohne Wartezeit der Fall ist, wird der Lock langsamer. Ansonsten wird er schneller. Dies ist in Benchmark 4.9c allerdings kaum zu erkennen, da die Verbesserung nur klein ist.

Die Verbesserung ist von der unkritischen Arbeit abhängig, da ohne unkritische Arbeit der Lock direkt wieder akquiriert wird, wobei auch `MPI_Win_flush` zum Einsatz kommt. In diesem Fall verursacht die Änderung also nur unnötigen Overhead. Wenn aber zunächst unkritische Arbeit ausgeführt wird, kann sich dadurch die Ausführung von `MPI_Accumulate` verzögern und damit der kritische Pfad verlängern.

Die zweite Optimierung ist die nicht-atomare Übergabe von Locks aus Abschnitt 3.4. Statt der beiden `MPI_Accumulate`-Operationen wird `MPI_Put` verwendet. Dadurch muss die MPI-Implementierung nicht mehr prüfen, ob mehrere parallele RMA-Operationen auf diesen Speicherbereich zugreifen. Parallelere Zugriffe sind an diesen Stellen sowieso durch den Algorithmus ausgeschlossen. Die Optimierung realisiert auch das in Unterabschnitt 4.2.2 identifizierte Optimierungspotenzial auf einem einzelnen Rechenknoten. Sowohl in Benchmark 4.9b als auch in Benchmark 4.10 wird das deutlich. Der so optimierte D-MCS schlägt sogar den Intel-MPI-Lock, welcher auf einem Rechenknoten zuvor die beste Performance hatte (siehe Benchmark 4.12a).



(a) Iterationsdauer in µs

(b) Konkurrenz in %

Benchmark 4.10: CCWB der D-MCS-Optimierungen mit 28 Prozessen

Die letzte Optimierung ist das Ersetzen der MPI-Operationen durch C++ `std::atomic`. Damit ist zwar nur lokale, also auf den Knoten beschränkte Kommunikation möglich, aber wie in Abschnitt 3.6 erwähnt, kann so ein Lock als Baustein für NUMA-Locks wie den Cohort-Lock [DMS12] dienen. Außerdem wird hierdurch noch einmal verdeutlicht, welchen Overhead die lokale Nutzung von RMA-Operationen mit sich bringt. Das ist besonders in Benchmark 4.9a ersichtlich.

4.3.2 Optimierung von dash::Mutex

Damit der Algorithmus von `dash::Mutex` leichter optimiert werden kann, wird er zunächst wie die anderen Locks mit einem eigenen MPI-Fenster reimplementiert. Der Quelltext der Reimplementierung ist in Abbildung 4.3 zu sehen. Dabei ändern sich einige Implementierungsdetails und da der Lock nun nicht mehr in die Bibliothek DASH eingebunden ist, entfallen einige lokale Operationen, wie etwa das Laden von Metadaten des DASH-Teams.

```

1  win_mem->next = -1;
2  MPI_Win_sync(win);
3
4  // Finde Vorgänger
5  int pred;
6  MPI_Fetch_and_op(
7      &my_rank, &pred, MPI_INT,
8      main_rank, tail_disp,
9      MPI_REPLACE, win);
10 MPI_Win_flush(main_rank, win);
11
12 if (pred != -1) {
13     // Reihe nach Vorgänger ein
14     int result;
15     MPI_Fetch_and_op(
16         &my_rank, &result, MPI_INT,
17         pred, next_disp,
18         MPI_REPLACE, win);
19     MPI_Win_flush(pred, win);
20
21     // Warte auf Vorgänger
22     MPI_Recv(NULL, 0, MPI_UINT8_T,
23             predecessor, 0, comm,
24             MPI_STATUS_IGNORE);
25 }

```

(a) Akquirieren des Locks (`acquire`)

```

1  constexpr int NULL_RANK = -1;
2  int old_value;
3  MPI_Compare_and_swap(
4      &NULL_RANK, &my_rank, &old_value,
5      MPI_INT, main_rank, tail_disp, win);
6  MPI_Win_flush(main_rank, win);
7
8  // Die Warteschlange ist leer
9  if (old_value == my_rank) return;
10 // Warte auf Nachfolger
11 int succ;
12 do {
13     int flag; // Garantiere Fortschritt
14     MPI_Iprobe(
15         MPI_ANY_SOURCE, MPI_ANY_TAG,
16         comm, &flag, MPI_STATUS_IGNORE);
17
18     MPI_Fetch_and_op(
19         NULL, &successor, MPI_INT,
20         my_rank, next_disp, MPI_NO_OP, win);
21     MPI_Win_flush(my_rank, win);
22 } while(succ == -1);
23 // Benachrichtige Nachfolger
24 MPI_Send(NULL, 0, MPI_UINT8_T,
25         successor, 0, comm);

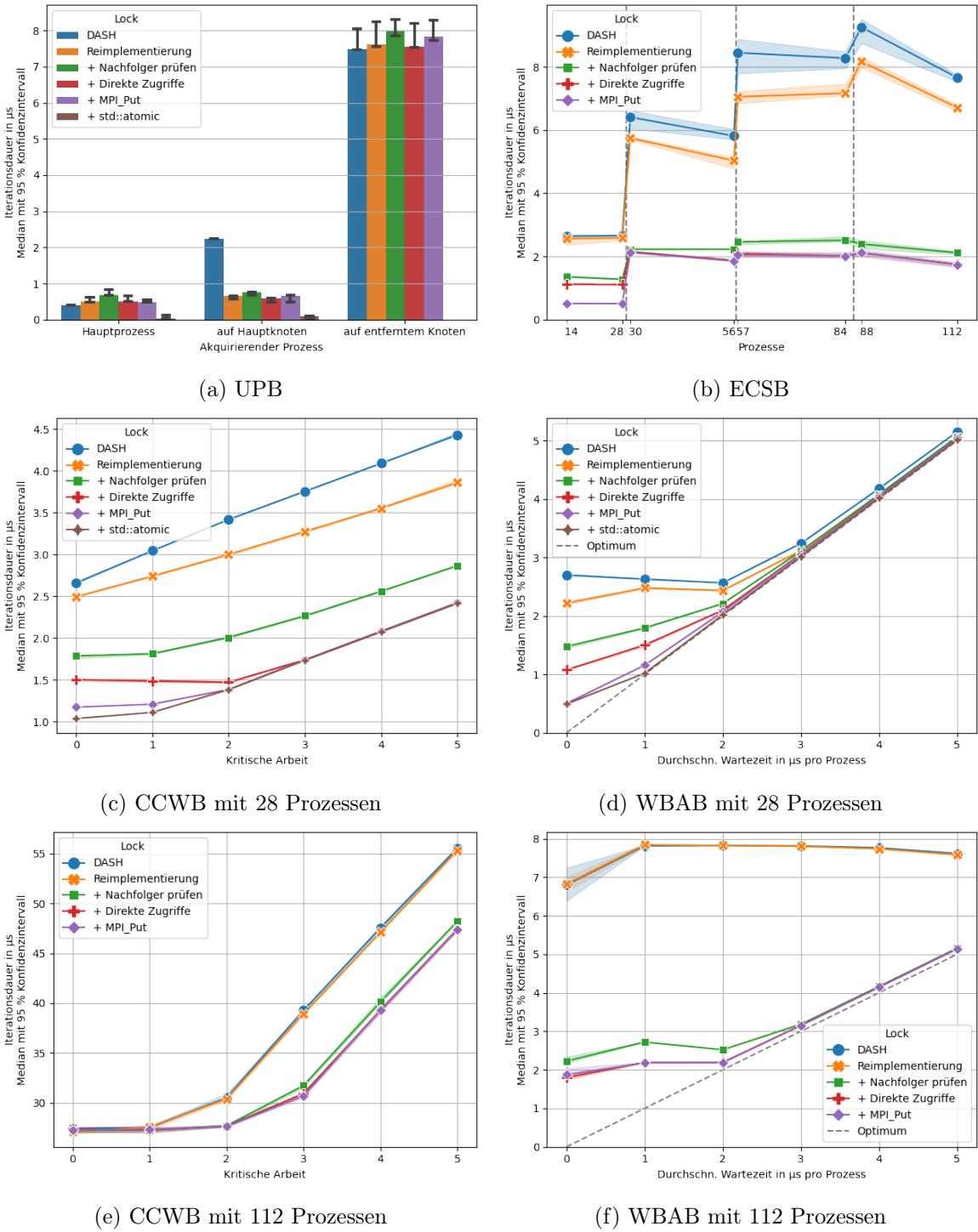
```

(b) Freigeben des Locks (`release`)Abbildung 4.3: Implementierung von `dash::Mutex`

Durch das Entfallen dieser Operationen nimmt vor allem der unkritische Overhead ab. Das sieht man besonders im CCWB und WBAB auf einem Rechenknoten (in Benchmark 4.11c und Benchmark 4.11d): Beide Implementierungen erreichen bei einer kritischen Arbeit von 0, bzw. einer Wartezeit von 2 µs das Equilibrium, haben also vergleichbaren kritischen Overhead, die Reimplementierung ist aber immer etwas schneller. Durch diese Reimplementierung verschwindet auch das in Unterabschnitt 4.2.1 festgestellte Verhalten, dass das Akquirieren eines freien Locks länger dauert, wenn es sich nicht um den Hauptprozess handelt (siehe Benchmark 4.11a).

Das Hauptproblem an dem Algorithmus hinter `dash::Mutex` ist, dass in `release` nicht, wie bei einem MCS-Lock üblich, mit einer lokalen Operation geprüft wird, ob sich bereits ein Nachfolger registriert hat. Stattdessen wird direkt versucht, mit einer CAS-Operation auf den entfernten Speicher des Hauptprozesses den Lock freizugeben. Erst wenn diese Operation fehlschlägt, wird mit `MPI_Fetch_and_op` und `MPI_NO_OP` auf einen lokalen Nachfolger gewartet (Abbildung 4.3b, Zeile 10-22). Die erste Optimierung besteht daher darin, in `release` bereits vor dem CAS mit `MPI_Fetch_and_op` und `MPI_NO_OP` atomar zu prüfen, ob lokal ein Nachfolger registriert ist. Hier wird eine atomare MPI-Funktion benötigt, um einen *memory consistency error* zu vermeiden, wenn der Nachfolger genau im selben Moment Zeile 13-19 in Abbildung 4.3a ausführt, um sich zu registrieren.

4 Evaluation und Optimierung bestehender Locks



Benchmark 4.11: Iterationsdauer der DASH-Optimierungen in µs

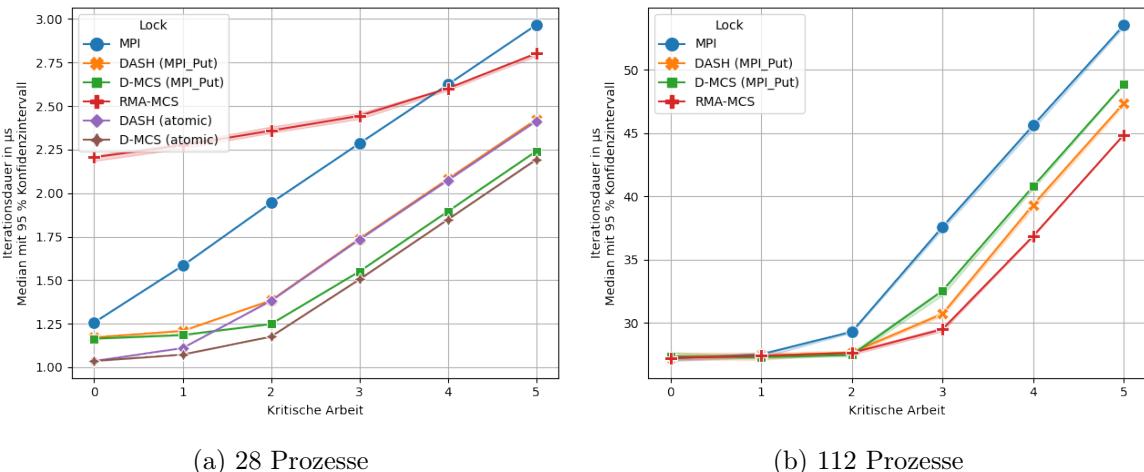
Der UPB zeigt in Benchmark 4.11a, dass diese Änderung bei einem freien Lock etwas langsamer ist, da eine zusätzliche Operation ausgeführt wird, die unnötig ist, wenn es keinen Nachfolger gibt. Da es sich aber nur um eine Lese-Operation handelt, ist dieser zusätzliche

Overhead sehr gering, besonders wenn man ihn mit dem Overhead der Einbettung in DASH vergleicht. Dafür verbessert diese Änderung die Performance in allen anderen Benchmarks drastisch: Wenn es einen Nachfolger gibt, wird eine CAS-Operation vermieden, die deutlich langsamer ist als die atomare Lese-Operation, die in diesem Fall sowieso im Anschluss ausgeführt würde. Wenn mehrere Knoten beteiligt sind, ist dieser Effekt sogar noch stärker, da die CAS-Operation auf den Speicher des Hauptprozesses und damit potenziell auf entfernten Speicher zugreift.

Als zweite Optimierung werden, wie im D-MCS-Lock, für Zugriffe auf lokalen Speicher direkte Speicherzugriffe verwendet (vgl. Abbildung 3.5b, Zeile 13-20). Das betrifft nur das in der ersten Optimierung hinzugefügte `MPI_Fetch_and_op` und das Warten auf den Nachfolger (Abbildung 4.3b, Zeile 18-21), da `dash::Mutex` Punkt-zu-Punkt-Kommunikation für die Lockübergabe nutzt. Diese Optimierung gleicht die kleine Verschlechterung im UPB wieder aus, da ein direkter lesender Zugriff auf lokalen Speicher sehr schnell ist, und auch in allen anderen Benchmarks verbessert sich dadurch die Geschwindigkeit.

Als drittes wird wie in Unterabschnitt 4.3.1 `MPI_Put` verwendet, um sich beim Vorgänger zu registrieren (Abbildung 4.3a, Zeile 15-18). Anders als bei D-MCS ersetzt `MPI_Put` hier ein `MPI_Fetch_and_op` und nicht ein `MPI_Accumulate`. Warum `dash::Mutex` an dieser Stelle `MPI_Fetch_and_op` verwendet ist nicht klar. Diese Funktion liefert im Gegensatz zu `MPI_Accumulate` den Wert zurück, der vor der Operation an der Speicheradresse stand. Dieser alte Wert (die lokale Variable `result`) wird aber gar nicht verwendet. Diese Optimierung bringt anders als bei D-MCS nur eine relativ kleine Verbesserung bei mehreren Rechenknoten, da die Übergabe des Locks an den Nachfolger Punkt-zu-Punkt-Kommunikation nutzt und somit nicht betroffen ist. Nach dem Registrieren beim Vorgänger müssen Prozesse typischerweise sowieso auf ihren Vorgänger warten. Daher bringt es nichts, wenn das Registrieren schneller ist. Trotzdem verbessert sich die Performance bei nur einem Rechenknoten deutlich. Zu prüfen, ob mehrere parallele RMA-Operationen auf denselben Speicherbereich zugreifen, verursacht demnach erheblichen Overhead bei der MPI-Implementierung.

Als letzte Optimierung werden wieder alle MPI-Operationen durch C++ `std::atomic` ersetzt, wodurch dieser Lock wieder nur als Baustein für NUMA-Locks wie den Cohort-Lock [DMS12] dienen kann.



Benchmark 4.12: CCWB der Basislocks-Optimierungen: Iterationsdauer in μs

4 Evaluation und Optimierung bestehender Locks

Benchmark 4.12 zeigt schließlich die optimierten Varianten von D-MCS und `dash::Mutex` noch einmal im Vergleich miteinander und zusätzlich im Vergleich zu dem Intel-MPI-Lock und RMA-MCS auf einem und vier Rechenknoten. In Benchmark 4.12a sieht man, dass bei einem Knoten der optimierte D-MCS, also RMA, schneller ist. Benchmark 4.12b zeigt, dass bei mehreren Knoten hingegen der optimierte `dash::Mutex`, also Punkt-zu-Punkt-Kommunikation, schneller ist. Beide optimierte Varianten schlagen den Intel-MPI-Lock, welcher zuvor auf einem Rechenknoten am schnellsten war, aber trotz aller Optimierungen erreicht bei mehreren Rechenknoten weiterhin keiner der Locks die Geschwindigkeit von RMA-MCS. Das zeigt deutlich das enorme Potenzial der Portierung von NUMA-Locks auf verteilten Speicher.

5 Portierung von NUMA-Locks auf verteilten Speicher

Es gibt einige Locks für Systeme mit gemeinsamem Speicher, die die Unterschiede zwischen Zugriffen auf lokalen und entfernten Speicher bei NUMA berücksichtigen und Fairness opfern, um höhere Geschwindigkeiten zu erreichen. Obwohl die Geschwindigkeitsunterschiede von Speicherzugriffen in verteilten Systemen noch deutlich ausgeprägter sind, wird diese Technik dort kaum genutzt. Dabei ähneln sich Speicherzugriffe auf gemeinsamem Speicher und RMA auf verteilten Systemen stark. In diesem Kapitel werden daher einige NUMA-Locks untersucht und wenn möglich auf verteilten Speicher portiert.

5.1 Delegierende Locks

Neben klassischen Locks, die in Abschnitt 2.1 beschrieben wurden, gibt es bei Systemen mit gemeinsamem Speicher das Konzept von delegierenden Locks, welche zum Teil auch NUMA berücksichtigen [OTY99] [HIST10] [FK11] [FK12] [LDT⁺12] [PRS15] [ZLW⁺16] [YY20]. Die Idee dabei ist, dass ein oder mehrere Prozesse als Server agieren und kritische Abschnitte für alle anderen Client-Prozesse ausführen. Client-Prozesse schicken dafür Anfragen an den Server, in denen der auszuführende kritische Abschnitt, sowie die notwendigen Parameter enthalten sind und warten auf eine Antwort des Servers, die zeigt, dass der kritische Abschnitt ausgeführt wurde.

Unterschieden wird dabei zwischen statischen und dynamischen Locks, je nachdem, ob ein oder mehrere fest bestimmte Prozesse als Server dienen, oder jeder Prozess dynamisch für eine gewisse Zeit zu einem Server werden kann.

Delegierende Locks haben in Systemen mit gemeinsamem Speicher den großen Vorteil, dass die Daten, die durch den Lock geschützt werden und auf die im kritischen Abschnitt zugegriffen wird, immer von demselben Prozess (dem Server-Prozess) verwendet werden. Damit liegen sie dort typischerweise in lokalen Zwischenspeichern vor, wodurch besonders schnell auf sie zugegriffen werden kann.

Ein Nachteil von delegierenden Locks ist allerdings, dass sie nicht dieselbe Programmierschnittstelle nutzen können wie klassische Locks, weil sie Zugriff auf alle Parameter des kritischen Abschnitts benötigen und der kritische Abschnitt selbst eine eigene Funktion sein muss, damit der Server-Prozess diese direkt aufrufen kann. Ein Programm von klassischen Locks auf delegierende Locks umzustellen, kann daher sehr aufwendig sein.

Bei verteiltem Speicher besteht darüber hinaus das Problem, dass Prozesse im kritischen Abschnitt auch auf private Daten zugreifen können, die nicht über ein Fenster veröffentlicht wurden. In so einem Fall ist es für einen anderen Prozess gar nicht möglich, den kritischen Abschnitt auszuführen. Delegierende Locks eignen sich daher nicht für eine Portierung auf verteilten Speicher. Außerdem kommt der Hauptvorteil von delegierenden Locks, dass die Daten des kritischen Abschnitts meist schon im Zwischenspeicher des Server-Prozesses liegen,

bei verteiltem Speicher nicht zum Tragen, da in MPI die Daten von entfernen Prozessen nicht automatisch zwischengespeichert werden.

Das Konzept der Delegierung an einen Server-Prozess passt allerdings sehr gut zu *Message Passing* und die hier beschriebenen Probleme lassen sich leicht vermeiden, wenn ein Programm speziell dafür entwickelt wird. Daher eignet sich dieses Konzept möglicherweise, um parallele Datenstrukturen, z. B. Warteschlangen (engl. *Queues*), für verteilte Systeme zu entwickeln.

5.2 RH-Lock

Der RH-Lock [RH02] ist der erste bekannte Lock der NUMA berücksichtigt, um eine bessere Performance zu erzielen. Er besteht aus einem TTS-Lock [RS84] mit exponentiellem Backoff [AC89] (siehe Unterabschnitt 2.1.1) pro NUMA-Knoten. Da der RH-Lock nur zwei NUMA-Knoten unterstützt, sind es also immer zwei TTS-Locks.

Statt einer *Test-and-Set*-Operation, nutzen die TTS-Locks für lokale Akquisitionen keine TAS-, sondern eine *swap*-Operation, da der Lock-Status nicht nur die Werte „FREE“ und „ACQUIRED“ annehmen kann. Wenn der Lock nicht verfügbar ist, enthält der Lock-Status stattdessen die eindeutige ID des Prozesses, der zuletzt mit einer *swap*-Operation versucht hat, den Lock zu akquirieren, oder den Status „REMOTE“, wenn zuletzt ein Prozess des anderen NUMA-Knotens den Lock erfolgreich akquiriert hat.

Durch die IDs hat ein Prozess die Möglichkeit, den Lock mit einer CAS-Operation nur dann global freizugeben, wenn der Status immer noch seine ID enthält. Wenn diese CAS-Operation fehlschlägt, muss es innerhalb seines NUMA-Knotens einen anderen Prozess geben, der auf den Lock wartet. In diesem Fall wird der Lock mit dem Status „L_FREE“ nur lokal, also für Prozesse im selben NUMA-Knoten freigeben.

Liest ein Prozess den Status „REMOTE“, weiß er, dass er die Speicheradresse des TTS-Locks des jeweils anderen NUMA-Knotens für die Akquisition nutzen muss. Durch die *swap*-Operation sieht nur der erste Prozess eines NUMA-Knotens den „REMOTE“ Status. Alle weiteren Prozesse sehen die ID ihres Vorgängers und warten auf den lokalen TTS-Lock.

Um einen entfernten TTS-Lock zu akquirieren, prüft ein Prozess, anders als bei seinem lokalen TTS-Lock, nicht zuerst mit einer Lese-Operation, ob der Lock frei ist. Der Lock funktioniert in diesem Fall also eher wie ein TAS-Lock (vgl. Abbildung 2.3a) mit Backoff. Dadurch, dass entfernte Zugriffe langsamer sind, hätte ein entfernter Prozess sonst kaum eine Chance, den Lock zu akquirieren, da er deutlich länger braucht, um darauf zu reagieren, dass der Lock frei ist. Stattdessen versucht ein Prozess immer direkt mit einer CAS-Operation, den entfernten Lock zu akquirieren. Hierbei wird CAS, statt *swap* genutzt, da der entfernte Lock nur akquiriert werden soll, wenn er den Status „FREE“, nicht aber z. B. den Status „L_FREE“ hat.

Um Verhungern (engl. *Starvation*) zu vermeiden, setzt der RH-Lock einen Zufallszahlen-generator ein und gibt mit konfigurierbarer Wahrscheinlichkeit den Lock global frei, selbst wenn es einen lokalen wartenden Prozess gibt, sodass alle Prozesse die Möglichkeit haben, den Lock zu akquirieren.

5.2.1 Portierung und Optimierung des RH-Locks

Das größte Problem des RH-Locks für eine Portierung auf verteilten Speicher ist, dass der Lock nur zwei NUMA-Knoten unterstützt. Es ist aber eher selten, dass ein verteiltes System aus nur zwei Knoten besteht. Dieses Problem lässt sich z. B. beheben, indem man den „REMOTE“ Status erweitert und dort zusätzlich die ID des entfernten Prozesses codiert, in dessen Speicher der TTS-Lock ist. Für die Portierung werden hierfür negative Zahlen verwendet ($-id - 1$ um die ID 0 zu unterstützen), da Prozess IDs typischerweise positiv sind. Somit signalisiert eine positive Zahl, dass der Prozess lokal warten muss und eine negative Zahl, dass der Prozess auf den TTS-Lock im Speicher des entfernten Prozesses mit der zugehörigen positiven ID warten muss.

Bei mehr als zwei Knoten kann es nun allerdings passieren, dass ein Prozess auf Knoten a einen „REMOTE“ Wert liest, der ihn auf Knoten b verweist, obwohl Knoten b den Lock gar nicht mehr hat, da inzwischen ein Prozess von Knoten c den Lock akquiriert hat. Daher kann ein Prozess nicht mehr einfach darauf warten, dass der TTS-Lock des anderen Knotens frei wird, sondern muss auch dort prüfen, ob es sich um einen „REMOTE“ Wert handelt, der ihn weiter verweist. Bei einem freien RH-Lock muss ein Prozess so im schlimmsten Fall jeden anderen TTS-Lock fragen, bis er den richtigen findet. Dieser Overhead steigt demnach linear mit der Anzahl der beteiligten Knoten.

Beim RH-Lock gehen die meisten Speicherzugriffe auf den lokalen TTS-Lock des NUMA-Knotens, was gut ist, da lokale Zugriffe schneller als entfernte sind. Dabei handelt es sich vor allem um die Lese-Operation, mit der ermittelt wird, ob der Lock frei ist. Da es sich um einen lokalen Zugriff handelt, kann genau diese Operation ohne RMA mit einem schnelleren direkten Speicherzugriff implementiert werden, wenn für die lokale Kommunikation ein Fenster auf den gemeinsamen Speicher erstellt wurde (vgl. Abschnitt 3.6). Dabei ist zu beachten, dass in der Warteschleife `MPI_Iprobe` aufgerufen wird, um Programmfortschritt zu garantieren (vgl. Abschnitt 3.3).

5.2.2 Evaluation des RH-Locks

Da die Warteschleife einen Aufruf von `MPI_Iprobe` enthalten muss, um Programmfortschritt zu garantieren, ist sie deutlich langsamer als eine leere Warteschleife, wie sie in [RH02] genutzt wird. Außerdem ist die Latenz von Zugriffen auf entfernten Speicher in einem verteilten System deutlich höher als in einem System mit gemeinsamem NUMA-Speicher, da der Zugriff über das Netzwerk gehen muss. Allein aus diesem Grund ist die Frequenz, mit der versucht wird, einen entfernten TTS-Lock zu akquirieren, deutlich geringer. Aus diesen beiden Gründen können nicht einfach dieselben Werte für Backoff (vgl. Unterabschnitt 2.1.1) verwendet werden wie in [RH02].

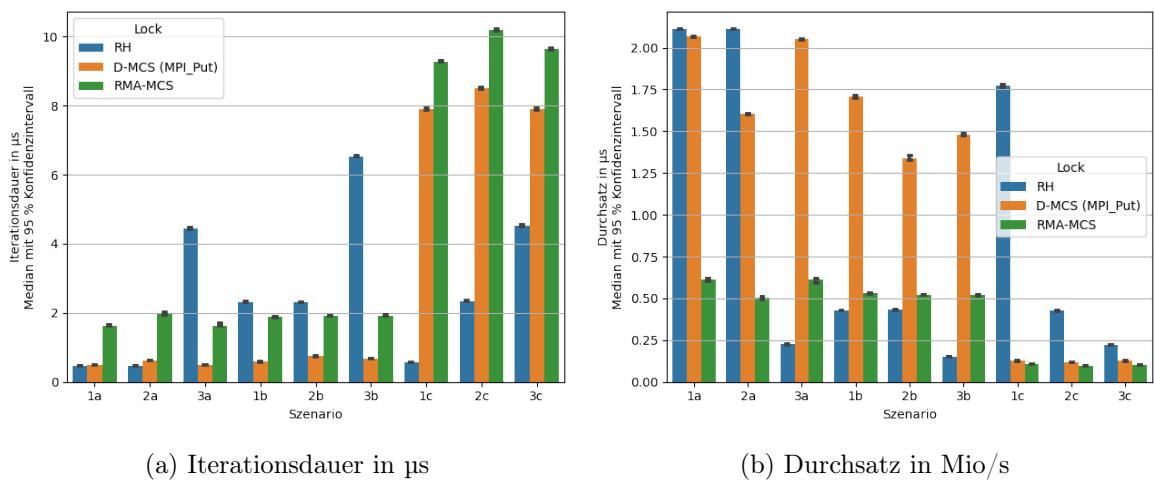
Durch die hohe Latenz von entfernten Speicherzugriffen, ist die Frequenz dieser Zugriffe während des Wartens auf einen entfernten TTS-Lock bereits so niedrig, dass zusätzlicher Backoff keine Verbesserung bringt, sondern entfernte Prozesse nur unfair benachteiligt. Der initiale und maximale Backoff für das entfernte Akquirieren werden daher in der Evaluation auf 0 gesetzt, wodurch der Backoff deaktiviert wird.

Für Zugriffe auf den lokalen TTS-Lock hingegen ist Backoff weiterhin sinnvoll. Durch die Verwendung von `MPI_Iprobe` sind die optimalen Werte aber deutlich geringer als in [RH02]. Als maximaler Backoff werden lokal 16, 32 und 64 Iterationen evaluiert, da ungefähr bei 32 Backoff-Iterationen ein lokales Optimum bei einer durchschnittlichen Wartezeit von

1 μ s im WBAB liegt. Da diese Maximalwerte so klein sind, wird als Minimalwert immer 1 verwendet. So erreicht ein Prozess spätestens nach dem 7. Zugriff auf den TTS-Lock die maximale Wartezeit von $2^6 = 64$ Iterationen.

Wie bereits in Unterabschnitt 2.1.1 erläutert, hängt die optimale Backoff-Konfiguration von der Geschwindigkeit des verwendeten Systems, aber auch von dem konkreten Szenario (Länge des kritischen Abschnitts, Anzahl der Prozesse) ab. Es ist also möglich, dass in manchen Szenarien eine andere Konfiguration besser wäre. Die Optimierung auf eine durchschnittliche Wartezeit von 1 μ s im WBAB ist zwar willkürlich, liefert aber zumindest deutlich bessere Ergebnisse als eine einfache Wiederverwendung der Konfiguration aus [RH02].

Neben dem Backoff ist beim RH-Lock auch die Fairness konfigurierbar. Hier muss eingestellt werden, mit welcher Wahrscheinlichkeit trotz lokalen Nachfolgers der Lock global freigegeben wird. In dieser Arbeit werden für einen maximalen lokalen Backoff von 32, wie in [RH02] die Fairnessfaktoren 1, 2 & 100 evaluiert. D. h. der Lock wird bei einem lokalen Nachfolger trotzdem mit einer Wahrscheinlichkeit von 1, $\frac{1}{2}$ oder $\frac{1}{100}$ global freigegeben. Bei einem maximalem lokalem Backoff von 16 und 64 ist der RH-Lock in dieser Arbeit immer maximal fair (also mit einem Fairnessfaktor von 1) eingestellt.

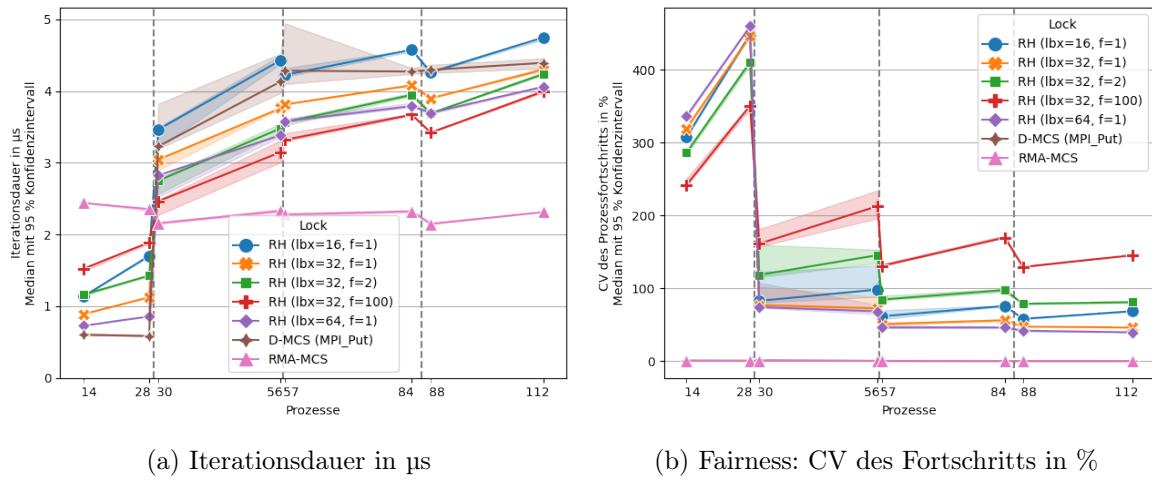


Benchmark 5.1: UPB des RH-Locks

Benchmark 5.1a zeigt, wie lange es dauert, einen freien RH-Lock zu akquirieren, im Vergleich zum optimierten D-MCS-Lock aus Unterabschnitt 4.3.1 und RMA-MCS. Die Backoff- und Fairness-Konfiguration spielt für diesen Benchmark keine Rolle, da alle Locks frei sind, somit wird hier nur ein RH-Lock gezeigt. Anders als bei den anderen UPB-Evaluationen, sind hier alle neun Szenarien (vgl. Tabelle 4.1) separat aufgeführt. Beim RH-Lock macht es nämlich einen Unterschied, welcher Prozess den Lock als Letztes besaß.

Während es bei den optimierten D-MCS- und RMA-MCS-Locks vor allem entscheidend ist, wie weit der akquirierende Prozess vom Hauptprozess entfernt ist, also ob es ein a-, b- oder c-Szenario ist, ist es beim RH-Lock vor allem wichtig, ob der Vorgängerprozess auf demselben Knoten lief, also ob es ein 1er, 2er oder 3er-Szenario ist. Das liegt daran, dass es beim RH-Lock keinen globalen Hauptprozess gibt. Alle Rechenknoten sind beim RH-Lock äquivalent. Es gibt allerdings auf jedem Rechenknoten einen lokalen Hauptprozess, in dessen Speicher sich der TTS-Lock befindet.

Überraschenderweise lässt sich ein RH-Lock schneller durch einen lokalen Hauptprozess akquirieren. Benchmark 5.1 zeigt deutlich, dass der RH-Lock in den b-Szenarien (Prozess ist kein Hauptprozess) deutlich langsamer ist als in den a-Szenarien (Prozess ist ein Hauptprozess). In den c-Szenarien ist das Bild gemischt. Die c-Szenarien zeigen die Performance, wenn der akquirierende Prozess nicht auf dem Hauptknoten läuft. D-MCS und RMA-MCS sind hier deutlich langsamer als in den anderen Szenarien, da sie entfernte Zugriffe auf den Hauptknoten benötigen. Beim RH-Lock gibt es aber keinen Hauptknoten, sodass er sich in c-Szenarien genauso verhält wie in den anderen. Das gemischte Bild entsteht dadurch, dass der akquirierende Prozess in Szenario 1c und 3c ein lokaler Hauptprozess ist, in 2c aber nicht. Daher ist die Performance des RH-Locks in Szenario 1c identisch zu der in 1a, in 2c identisch zu der in 2b und in 3c identisch zu der in 3a.

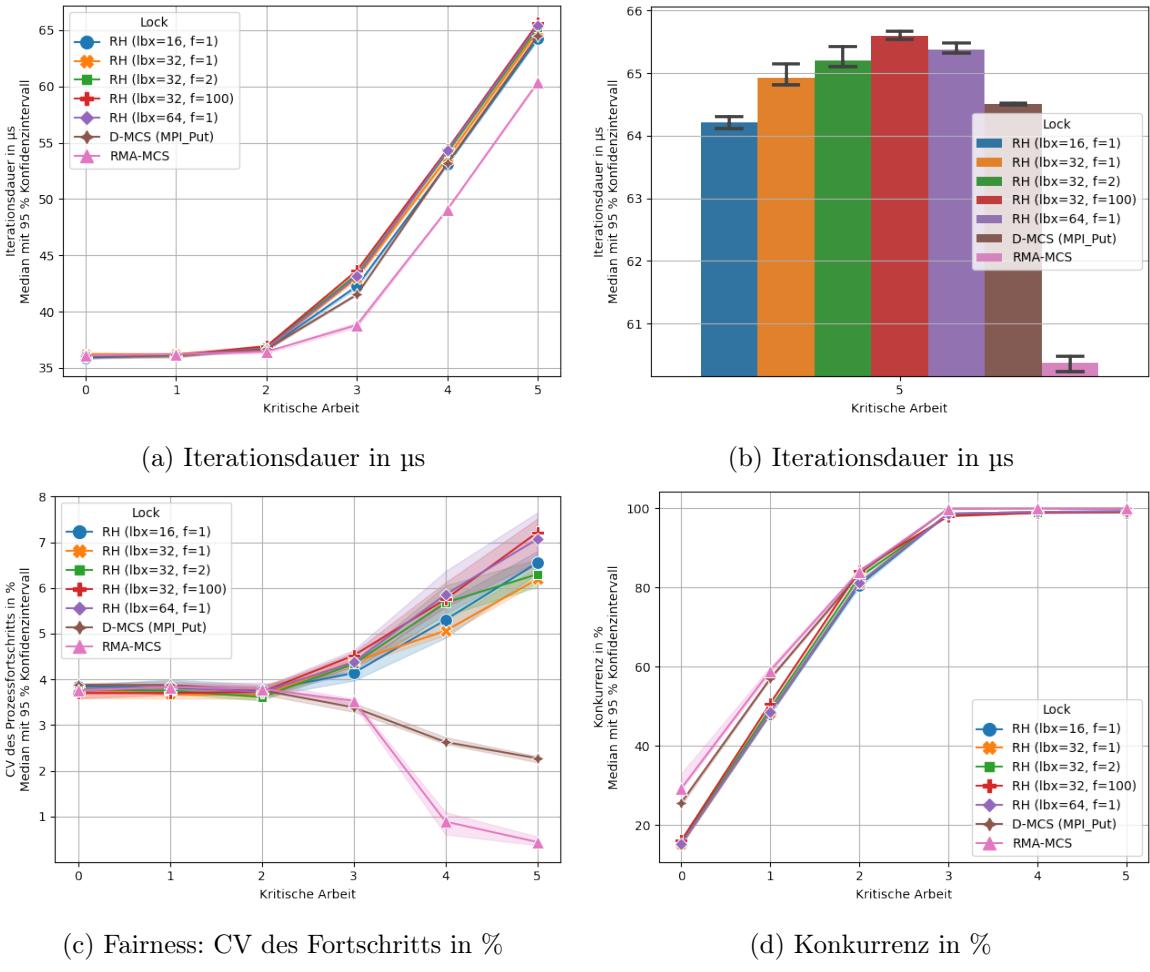


Benchmark 5.2: ECSB des RH-Locks

Benchmark 5.2 zeigt den ECSB für die verschiedenen Konfigurationen des RH-Locks, den D-MCS und RMA-MCS. Dabei steht „lbx“ für „lokales Backoff-Maximum“ und „f“ für „Fairnessfaktor“. In Benchmark 5.2a sieht man, dass bei mehreren Rechenknoten alle RH-Locks (außer mit lbx=16) schneller sind als der D-MCS-Lock. Dieser Vorteil wird aber mit erheblichen Fairnessproblemen erkauft, wie Benchmark 5.2b zeigt. Während D-MCS und RMA-MCS sehr fair sind (ohne erkennbare Abweichung von 0 %), haben alle RH-Locks extrem schlechte Fairnesswerte, auf einem Rechenknoten sogar einen Variationskoeffizienten von bis zu 460 %. Bei mehr Rechenknoten wird es zwar etwas besser, der erste RH-Lock fällt bei 112 Prozessen auf einen Variationskoeffizienten von knapp unter 40 %, aber sie bleiben ziemlich unfair. Da im ECSB Prozesse außerhalb des kritischen Abschnitts nichts tun müssen, kann derselbe Prozess den Lock immer wieder akquirieren und so anderen Prozessen zuvorkommen. Daher ist die Fairness in diesem Benchmark besonders schlecht.

Benchmark 5.2b zeigt auch, dass die Fairness auf mehreren Rechenknoten noch weiter abnimmt, wenn andere Werte für den Fairnessfaktor verwendet werden (2 und 100). Die Geschwindigkeit in Benchmark 5.2a nimmt dafür weiter zu. Man hat also hier die Möglichkeit, noch mehr Fairness für Geschwindigkeit zu opfern. Trotzdem kommt der RH-Lock mit keiner der Konfigurationen an die Geschwindigkeit von RMA-MCS heran, welcher obendrein auch noch sehr fair ist. Dieser hat allerdings in diesem Vergleich die mit Abstand schlechteste Performance bei nur einem Rechenknoten (auf 14 und 28 Prozessen).

5 Portierung von NUMA-Locks auf verteilten Speicher



Benchmark 5.3: CCWB des RH-Locks mit 112 Prozessen

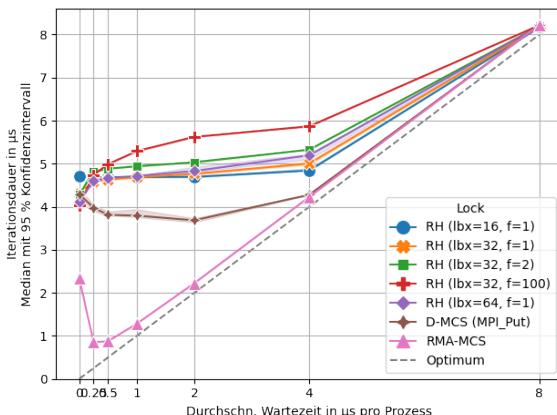
Auch beim CCWB sieht man in Benchmark 5.3c, dass der RH-Lock deutlich unfairer ist als D-MCS und RMA-MCS. Vor dem Equilibrium sind alle Locks gleich fair. Während bei D-MCS und RMA-MCS die Fairness ab dem Equilibrium besser wird, da Prozesse sich bei einer Konkurrenz von 100 % in eine Warteschlange mit beschränkter Fairness einreihen müssen, wird die Fairness des RH-Locks in allen Konfigurationen immer schlechter. Die Fairness ist aber um ein Vielfaches besser als im ECSB, da der CCWB ein deutlich realistischeres Szenario implementiert, bei dem Prozesse auch außerhalb des kritischen Abschnitts Arbeit ausführen.

Die bessere Fairness bringt aber leider auch eine schlechtere Geschwindigkeit mit sich. Im Gegensatz zum ECSB ist der RH-Lock hier nur mit einem „lbx“ von 16 schneller als der D-MCS-Lock. Und das auch nur bei einer kritischen Arbeit von 4 und 5. Beim ECSB war es genau anders herum: Dort war der RH-Lock nur mit einem „lbx“ von 16 langsamer als der D-MCS-Lock. Da die Locks in Benchmark 5.3a so dicht beieinanderliegen, dass sie schwer zu unterscheiden sind, zeigt Benchmark 5.3b eine Nahaufnahme der Performance bei einer kritischen Arbeit von 5. Diese Nahaufnahme zeigt überraschenderweise, dass ein höherer Fairnessfaktor im CCWB zu einem langsameren Lock führt. Auch das war im ECSB genau umgekehrt. Die Idee des Fairnessfaktors ist es eigentlich, Fairness zu opfern, um die

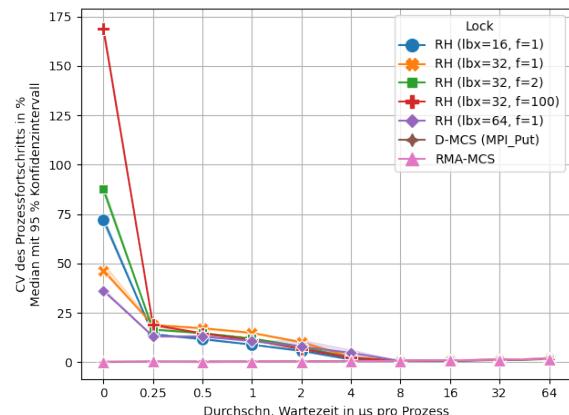
Geschwindigkeit zu verbessern. Der CCWB zeigt aber, dass in einem realistischen Szenario durch einen höheren Fairnessfaktor sowohl Fairness als auch Geschwindigkeit geopfert werden.

Der Grund hierfür ist wahrscheinlich, dass die Lock-Freigabe aus einer zusätzlichen CAS-Operation besteht, wenn sich der Prozess unfair verhält, also den Lock nur lokal freigibt. Der Status darf nur auf „L_FREE“ gesetzt werden, wenn es einen lokalen Nachfolger gibt. Sonst könnte es zu einem Deadlock kommen. Daher wird bei der unfairen Freigabe der Status erst gesetzt, wenn die CAS-Operation fehlschlägt, die den Lock global freigeben würde. Die zusätzliche CAS-Operation liegt auf dem kritischen Pfad und schlägt bei einer kritischen Arbeit von 5 immer fehl, da die Konkurrenz bei 100 % liegt (vgl. Benchmark 5.3d). Die faire Freigabe hingegen besteht nur aus einer Operation, die den Status immer auf „FREE“ setzt.

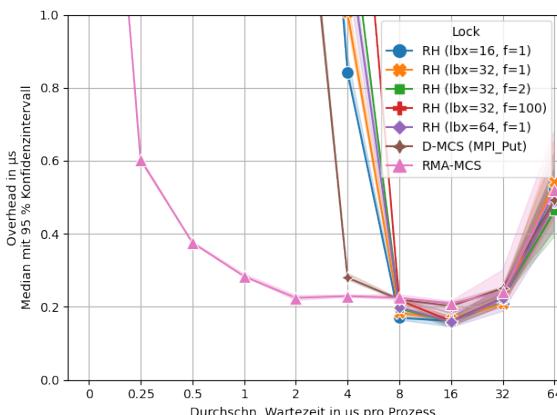
Obwohl die zusätzliche CAS-Operation auf lokalen Speicher zugreift, kann sie nicht mit C++ `std::atomic` implementiert werden, da auf dieselbe Speicheradresse auch entfernt zugegriffen wird. Es muss also ein langsamer RMA-Zugriff verwendet werden. Auf gemeinsamem Speicher wird die zusätzliche CAS-Operation dadurch ausgeglichen, dass durch eine lokale Lockübergabe die Daten, auf die der Nachfolgerprozess im kritischen Abschnitt zugreift, bereits im Zwischenspeicher vorliegen. Bei verteiltem Speicher gibt es allerdings keine Zwischenspeicher für entfernten Speicher, sodass dieser Vorteil hier nicht zum tragen kommt.



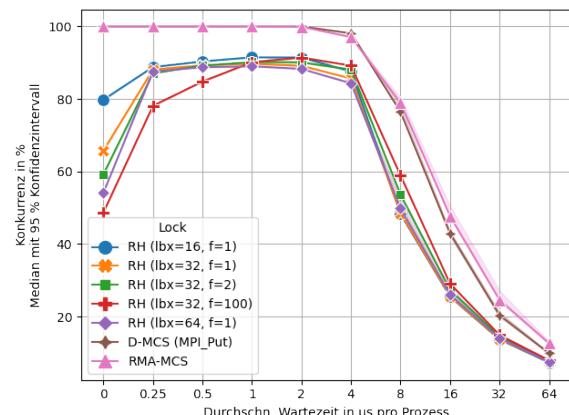
(a) Iterationsdauer in µs



(b) Fairness: CV des Fortschritts in %



(c) Overhead in µs



(d) Konkurrenz in %

Benchmark 5.4: WBAB des RH-Locks mit 112 Prozessen

Benchmark 5.4 zeigt die Ergebnisse des RH-Locks im WBAB. Genau wie im CCWB führt auch beim WBAB ein höherer Fairnessfaktor zu einem langsameren Lock. Bei der Geschwindigkeit in Benchmark 5.4a erreicht keiner der RH-Locks mehr die Performance von D-MCS, sobald Prozesse außerhalb des kritischen Abschnitts warten. Dafür zeigt Benchmark 5.4c aber, dass bei geringer Konkurrenz (Wartezeit von 8 µs bis 32 µs) der RH-Lock in allen Konfigurationen schneller ist als D-MCS und sogar RMA-MCS. Das passt auch zu den Ergebnissen aus Benchmark 5.1, nach denen ein freier RH-Lock schneller als D-MCS und RMA-MCS akquiriert werden kann, wenn der Prozess nicht auf dem Hauptknoten läuft. Der RH-Lock hat hier den Vorteil, dass keine MCS-Warteschlangen initialisiert werden müssen.

In Benchmark 5.4b sieht man noch einmal die extreme Unfairness des RH-Locks, wenn Prozesse keine Wartezeit außerhalb des kritischen Abschnitts haben, die auch der ECSB gezeigt hat. Diese wird bereits bei geringer Wartezeit deutlich besser, bleibt aber suboptimal, bis die Konkurrenz bei einer Wartezeit von 8 µs stark abfällt (vgl. Benchmark 5.4d). Ab diesem Punkt muss kein Prozess mehr lange auf den Lock warten, wodurch der RH-Lock auch nicht mehr unfair sein kann.

Bei der Konkurrenz in Benchmark 5.4d ist sehr auffällig, dass bei keinem RH-Lock je eine Konkurrenz von deutlich über 90 % erreicht wird. Ohne Wartezeit ist sie sogar noch geringer. Das stützt die Hypothese, dass ohne unkritische Arbeit häufig derselbe Prozess den Lock immer wieder akquiriert und so anderen Prozessen zuvorkommt. Dass auch mit unkritischer Arbeit etwa jeder zehnte Prozess einen freien Lock beobachtet, ist so allerdings nicht zu erklären.

Zusammenfassend lässt sich sagen, dass der RH-Lock zwar bei geringer Konkurrenz eine gute Performance hat, sobald die Konkurrenz jedoch steigt, hat er enorme Fairnessprobleme und ist kaum schneller als ein optimierter MCS-Lock. Außerdem muss der RH-Lock je nach System und erwarteter Konkurrenzsituation anders konfiguriert werden, um eine gute Performance zu erreichen, was zusätzlichen Aufwand für den Einsatz bedeutet.

Es könnte aber Fälle geben, in denen die Eigenschaft, dass es keinen festen Hauptprozess gibt, sondern der Lock immer auf dem Rechenknoten lokal ist, auf dem er zuletzt akquiriert wurde, große Vorteile bringt. Dadurch muss nicht im Voraus ein Hauptprozess definiert werden und wenn besonders Prozesse, die auf demselben Knoten laufen, häufig gleichzeitig den kritischen Abschnitt betreten möchten, kann so mit Sicherheit eine gute Performance erreicht werden. So ein Szenario ist in den Benchmarks dieser Arbeit zwar nicht enthalten, aber durchaus denkbar. Möglicherweise lässt sich der Ansatz eines wechselnden Hauptprozesses in einem zukünftigen Lock mit einer besseren Fairness realisieren.

5.3 HCLH-Lock

Der HCLH-Lock [LNS06] basiert auf dem CLH-Lock, welcher unabhängig von Craig [Cra93] und Landin und Hagersten [MLH94] entwickelt wurde und dem MCS-Lock sehr ähnelt.

5.3.1 CLH-Lock

Abbildung 5.1 zeigt eine C++-Implementierung des CLH-Locks. Die Felder und Initialisierung des Locks in Abbildung 5.1a sind dabei leicht vereinfacht. Die Warteschlangenknoten (engl. *node*) enthalten im Gegensatz zum MCS-Lock nur ein *locked-Flag* und keinen Zeiger auf den Nachfolger. Jeder Prozess hat in seinem privaten Speicher zwei Zeiger: *myreq* steht für „Meine Anfrage“ (engl. *my request*) und zeigt auf den Knoten, den der Prozess in die

```

1 struct node { atomic<bool> locked; };
2 node* myreq; // Pro Prozess
3 node* watch; // Pro Prozess
4 atomic<node*> tail; // Pro Lock
5
6 void initialize() {
7     myreq = new node();
8     if (hauptprozess)
9         tail.store(new node());
10 }

```

(a) Felder und Initialisierung

```

1 void acquire() {
2     myreq->locked.store(true);
3     watch = tail.exchange(myreq);
4     while (watch->locked.load());
5 }
6
7 void release() {
8     myreq->locked.store(false);
9     myreq = watch;
10 }

```

(b) Akquirieren und Freigeben des Locks

Abbildung 5.1: CLH-Lock

Warteschlange einreihen wird. Der Zeiger `watch` (deutsch beobachten) zeigt später auf den Knoten des Vorgängers, auf den gewartet wird, initial zeigt er jedoch ins Leere.

Zusätzlich gibt es einmal pro Lock einen Zeiger namens `tail` (deutsch Ende), der genau wie beim MCS-Lock auf den letzten Knoten der Warteschlange zeigt. Auf diesen Zeiger greifen alle Prozesse atomar zu, er befindet sich daher in öffentlichem Speicher. Genau wie der `myreq`-Zeiger jedes Prozesses, zeigt der `tail`-Zeiger initial auf einen eigenen Knoten. Im CLH-Lock gibt es daher immer einen Knoten mehr, als Prozesse beteiligt sind. Das `locked-Flag` aller Knoten hat am Anfang den Wert `false`.

In der `acquire`-Funktion setzt ein Prozess zunächst das `locked-Flag` seines Knotens auf `true`. Dann tauscht er mit einer `exchange`-Operation (anderer Name für `swap`) den Endknoten der Warteschlange durch seinen eigenen Knoten aus. Den bisherigen Endknoten merkt er sich mit seinem `watch`-Zeiger. Wenn der Prozess der Erste war, der `acquire` ausgeführt hat, dann zeigt `watch` nun auf den Knoten, der initial keinem Prozess gehört hat. Das `locked-Flag` des Knotens hat in diesem Fall immer noch den Initialwert `false`. Ansonsten zeigt `watch` nun auf den Knoten eines Vorgängers, welcher sein `locked-Flag` vor dem Einreihen in die Warteschlange auf `true` gesetzt hat. Nun muss der Prozess also nur noch warten, bis das `locked-Flag` seines Vorgängers `false` wird. Dann kann er den kritischen Abschnitt betreten.

In `release` setzt ein Prozess einfach das `locked-Flag` seines Knotens auf `false` und signalisiert so seinem Nachfolger, dass er den kritischen Abschnitt betreten kann. Von nun an kann der Prozess seinen Knoten nicht mehr verwenden, um sich erneut in die Warteschlange einzureihen, schließlich weiß er nicht, wann sein Nachfolger mitbekommt, dass der Lock frei ist. Stattdessen nutzt ein Prozess beim nächsten Mal einfach den Knoten, den er von seinem Vorgänger erhalten hat. Das erreicht er, indem er den Zeiger `myreq` auf denselben Knoten zeigen lässt wie `watch`. Die Knoten werden im CLH-Lock also rotiert, jeder Prozess erhält jedes Mal den Knoten seines Vorgängers für die nächste Runde.

5.3.2 CLH-Lock Variante für NUMA

Der Hauptunterschied zum MCS-Lock besteht darin, dass Prozesse im CLH-Lock beim Warten auf ihren Vorgänger einen Speicherbereich beobachten, der nicht unbedingt in ihrem lokalen Speicher liegt. Stattdessen beobachten sie bei jeder Akquisition einen anderen Speicherbereich, den sie von ihrem Vorgänger erhalten. Bei einem Rechnersystem mit Zwischenspeicher ist das kein Problem, da der Nachfolgerprozess den Speicherbereich des Vorgängers beim ersten Zugriff automatisch in seinen Zwischenspeicher lädt, wodurch weitere Zugriffe sehr schnell sind. MPI bietet allerdings keinen solchen Mechanismus, wodurch der CLH-Lock (und damit

5 Portierung von NUMA-Locks auf verteilten Speicher

auch der HCLH-Lock) bei einer einfachen Portierung in der Warteschleife meist auf entfernten Speicher zugreifen würde, was sehr ineffizient wäre. Dies wurde auch von Craig erkannt:

Without a cache coherence mechanism, each processor must spin on a location in its local shared memory. With our scheme [...] however, the physical location of the Request record that a process watches is unrelated to which processor is doing the watching. [Cra93, S. 12].

Daher schlägt Craig in [Cra93] eine Variante des CLH-Locks vor. Eine C++-Implementierung dieser Variante ist in Abbildung 5.2 zu sehen.

```

1 static constexpr int PENDING = -1;
2 static constexpr int GRANTED = -2;
3 struct node { atomic<int> status = GRANTED; };
4 atomic<bool> locked; // Pro Prozess

```

(a) Neue Felder

```

1 myreq->status.store(PENDING);
2 locked.store(true);
3 watch = tail.exchange(myreq);
4 int status = watch->status
5 .exchange(&locked);
6 if (status == PENDING)
7   while (locked.load());

```

(b) Akquirieren des Locks (`acquire`)

```

1 int status = myreq->status
2 .exchange(GRANTED);
3 if (status != PENDING) {
4   auto succ = (atomic<bool*>) status;
5   succ->store(false);
6 }
7 myreq = watch;

```

(c) Freigeben des Locks (`release`)

Abbildung 5.2: CLH-Lock für NUMA

Im Gegensatz zum normalen CLH-Lock liegt das `locked-Flag` nun im lokalen Speicher jedes Prozesses – zusammen mit den Zeigern `myreq` und `watch`. Der Warteschlangenknoten enthält stattdessen ein `status`-Feld.

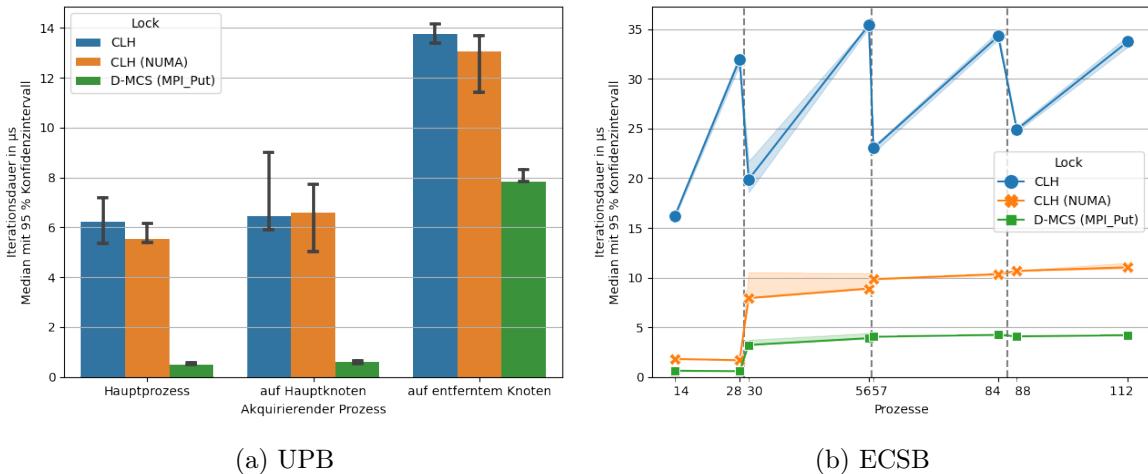
Wenn ein Prozess in `acquire` den Knoten seines Vorgängers erhält und in diesen `watch` speichert, enthält dessen `status`-Feld entweder den Wert `PENDING` oder `GRANTED`, je nachdem, ob der Vorgänger den Lock bereits freigegeben hat. Um in der Warteschlange nicht auf diesen potenziell entfernten Speicher zugreifen zu müssen, tauscht der akquirierende Prozess den Status atomar durch die Speicheradresse seines lokalen `locked-Flags` aus (Abbildung 5.2b, Zeile 4-5). Wenn der Status noch `PENDING` war, hatte der Vorgänger den Lock noch nicht freigegeben und der Prozess wartet darauf, dass sein lokales `locked-Flag` von seinem Vorgänger auf `false` gesetzt wird. Ansonsten kann er den kritischen Abschnitt direkt betreten.

Entsprechend tauscht ein Prozess in `release` den Wert des `status`-Feldes mit dem Wert `GRANTED`. Wenn der Status noch `PENDING` war, hat noch kein Nachfolger versucht, den Lock zu akquirieren. Der Prozess kann dann, wie im normalen CLH-Lock, den Knoten seines Vorgängers in `myreq` speichern und ist fertig. Ansonsten hat bereits ein Nachfolger die Speicheradresse seines `locked-Flags` mitgeteilt und wartet auf dieses `Flag`. In diesem Fall setzt der Prozess dieses `Flag` auf `false`, um den wartenden Nachfolger zu befreien (Abbildung 5.2c, Zeile 4-5).

Bei der gezeigten Implementierung ist es wichtig, dass die Speicheradressen der `locked-Flags` nicht die Werte -1 und -2 haben dürfen, damit sie nicht mit den Statuswerten `PENDING` und `GRANTED` verwechselt werden. Die ist eine kleine Abweichung zu [Cra93]: Dort wurde vorausgesetzt, dass die Speicheradressen gerade sind, sodass `PENDING` und `GRANTED` in das letzte Bit codiert werden konnten. Für den Algorithmus macht das keinen großen Unterschied, die Codierung mit negativen Zahlen ist aber mit MPI einfacher umzusetzen.

5.3.3 Evaluation des CLH-Locks

Vor einer Portierung des HCLH-Locks auf verteilten Speicher wurden zunächst die beiden Varianten des CLH-Locks portiert und mit dem optimierten D-MCS-Lock aus Unterabschnitt 4.3.1 verglichen, um abschätzen zu können, ob eine Portierung des HCLH-Locks überhaupt Sinn macht, wenn es keine Zwischenspeicher für entfernte Zugriffe gibt. Da im „CLH (NUMA)“-Lock Prozesse auf einen lokalen Speicherbereich warten, werden in der Warteschleife direkte Zugriffe und MPI_Iprobe genutzt (vgl. Abschnitt 3.3).



Benchmark 5.5: Iterationsdauer der CLH-Locks in μs

Benchmark 5.5 zeigt die Geschwindigkeit der CLH- und D-MCS-Locks bei Abwesenheit von Konkurrenz im UPB und maximaler Konkurrenz im ECSB.

Gerade in Benchmark 5.5a wird wieder einmal der Geschwindigkeitsunterschied zwischen lokalen und entfernten Speicherzugriffen deutlich. Wenn der akquirierende Prozess nicht auf dem Hauptknoten läuft, sind beim D-MCS-Lock nur zwei entfernte Speicherzugriffe notwendig: ein Zugriff auf den Speicher des Hauptprozesses, um den Lock zu akquirieren und einer um ihn wieder freizugeben.

Bei den beiden CLH-Locks sind hingegen meistens noch zwei weitere entfernte Zugriffe notwendig. Alle drei Locks müssen zu Beginn der Akquisition ihren Warteschlangenknoten initialisieren (Abbildung 2.4b, Zeile 1, Abbildung 5.1b, Zeile 2, bzw. Abbildung 5.2b, Zeile 1). Bei den beiden CLH-Varianten befindet sich dieser aber nicht unbedingt in lokalem Speicher, oder gar auf demselben Rechenknoten, sodass es sich hierbei häufig um einen entfernten Zugriff handelt. Zusätzlich wird in beiden CLH-Varianten, selbst bei einem freien Lock, über den `watch`-Zeiger einmal auf den Warteschlangenknoten des Vorgängers zugegriffen (Abbildung 5.1b, Zeile 4, bzw. Abbildung 5.2b, Zeile 4-5). Auch hier handelt es sich meist (aber nicht immer) um einen entfernten Zugriff. Diese beiden Zugriffe, die manchmal auf lokalen und manchmal auf entfernten Speicher gehen, erklären auch die ungenauen Konfidenzintervalle in Benchmark 5.5a: je nachdem, wie oft der Lock bereits akquiriert wurde, ist er manchmal schneller und manchmal langsamer.

Die beiden CLH-Locks führen im UPB also fast doppelt so viele entfernte Zugriffe aus wie der D-MCS-Lock, wenn der Prozess nicht auf dem Hauptknoten läuft. Dementsprechend sind sie auch fast doppelt so langsam. Wenn der akquirierende Prozess auf dem Hauptknoten

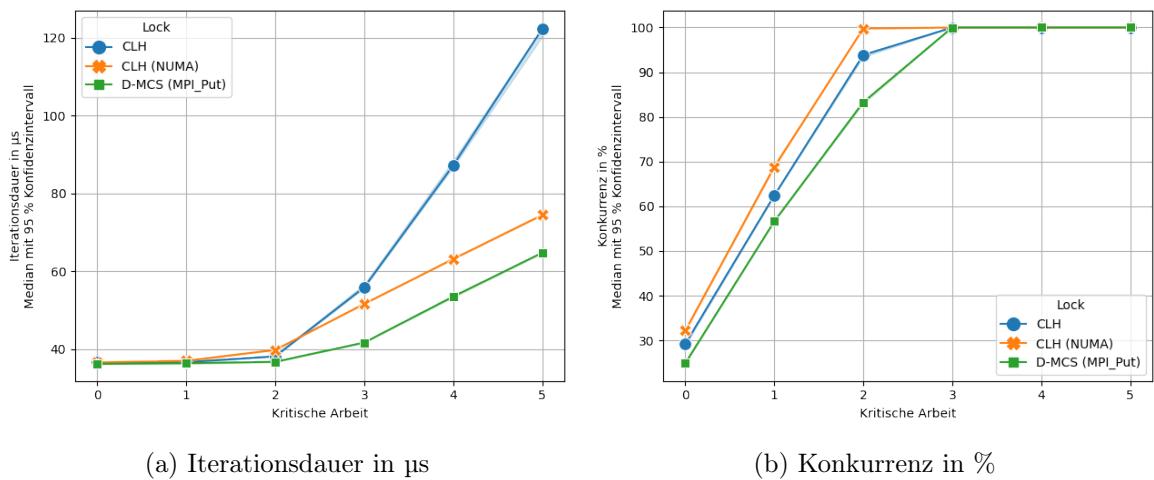
5 Portierung von NUMA-Locks auf verteilten Speicher

läuft, ist der Unterschied noch stärker: Nun sind bei allen drei Locks die Zugriffe auf den Speicher des Hauptprozesses lokal. Der D-MCS-Lock benötigt daher gar keine entfernten Zugriffe mehr, die beiden CLH-Locks hingegen meist schon.

Auch im ECSB (Benchmark 5.5b) sind die beiden CLH-Locks um ein Vielfaches langsamer als der D-MCS-Lock. Bei 112 Prozessen bleibt dessen Iterationsdauer bei etwa 4 µs, während der CLH-Lock für NUMA etwa 11 µs und der normale CLH-Lock sogar etwa 34 µs pro Iteration braucht. Die Nutzung von lokalen Zugriffen in der Warteschleife des CLH-Locks für NUMA bringen also eine enorme Verbesserung auf verteiltem Speicher. Trotzdem kommt auch diese Variante des CLH-Locks auf verteiltem Speicher nicht an den D-MCS-Lock heran.

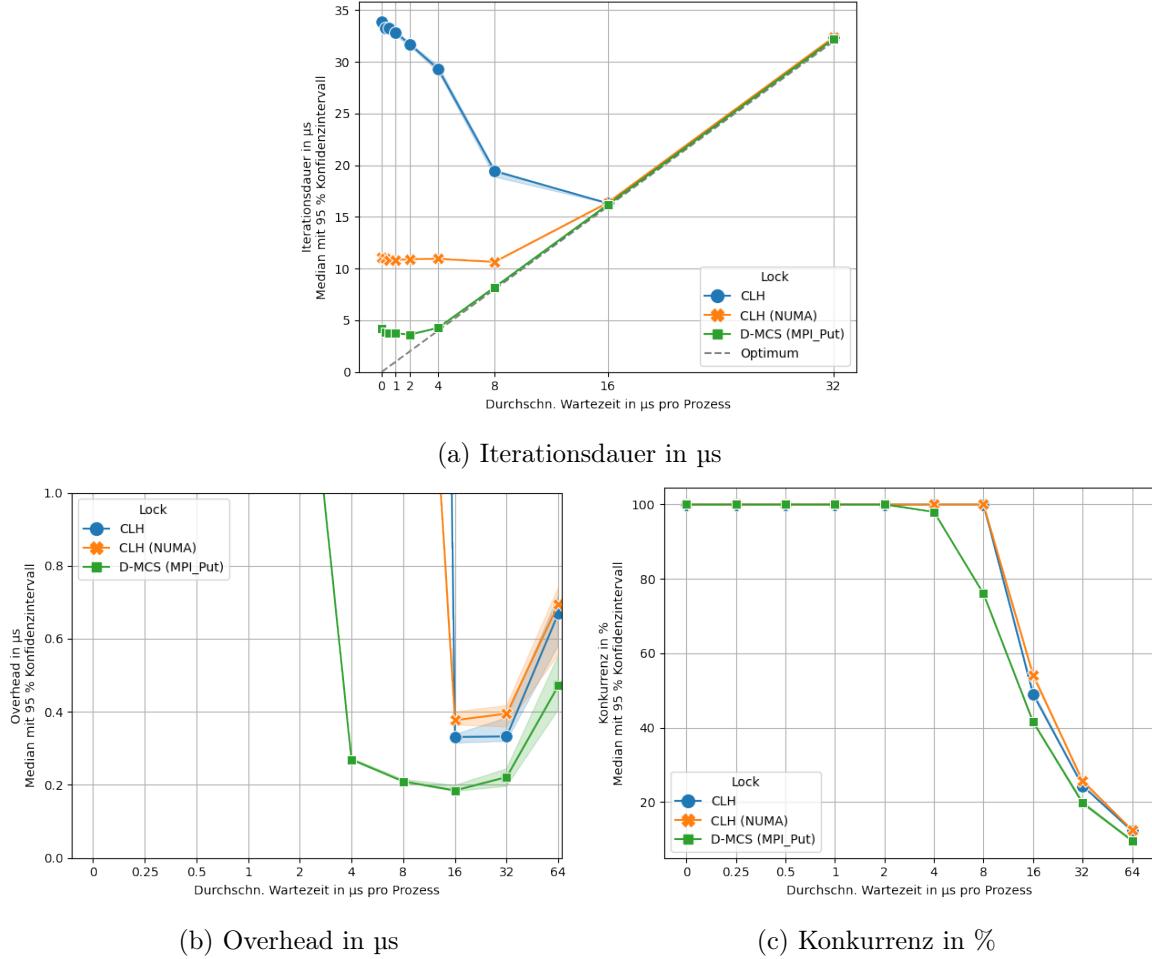
Der Grund ist hier vermutlich ein anderer als im UPB, da die Zugriffe, die dort zu schlechter Performance geführt haben, nicht auf dem kritischen Pfad liegen, sondern unkritischen Overhead verursachen. Stattdessen ist das Problem, dass der CLH-Lock für NUMA bei hoher Konkurrenz meist zwei entfernte Zugriffe benötigt, um den Nachfolger zu informieren. Zuerst wird in Abbildung 5.2c in Zeile 1-2 der Status des Warteschlangenknotens auf GRANTED geändert (dies ist nicht immer, aber meistens ein entfernter Zugriff) und dann wird in Zeile 5 der `locked`-Status des Nachfolgers mit einem entfernten Zugriff auf `false` gesetzt. Diese Zugriffe liegen beide auf dem kritischen Pfad und sind daher bei hoher Konkurrenz besonders problematisch. Der D-MCS-Lock kommt hingegen mit nur einem entfernten Zugriff auf den Speicher des Nachfolgers aus.

Die CLH-Varianten sind also sowohl bei minimaler als auch bei maximaler Konkurrenz deutlich langsamer als der D-MCS-Lock.



Benchmark 5.6: CCWB der CLH-Locks mit 112 Prozessen

Benchmark 5.6 zeigt die drei Locks im deutlich realistischeren CCWB. Auch dieser Benchmark bestätigt die bisherigen Beobachtungen. Der höhere kritische Overhead der NUMA-Variante des CLH-Locks führt dazu, dass diese Variante früher als die anderen Locks (bereits bei einer kritischen Arbeit von 2) das Equilibrium erreicht (vgl. Benchmark 5.6b). Dadurch hat er an diesem Punkt auch die schlechteste Iterationsdauer (vgl. Benchmark 5.6a). Sobald auch der normale CLH-Lock sein Equilibrium erreicht, wird dieser allerdings deutlich schlechter als der „CLH (NUMA)“-Lock, da Prozesse dann immer länger in der Warteschleife verweilen müssen, was zu sehr vielen entfernten Speicherzugriffen führt.



Benchmark 5.7: WBAB des CLH-Locks mit 112 Prozessen

Zuletzt zeigt Benchmark 5.7 die Performance der drei Locks im WBAB. In Benchmark 5.7a lässt sich erkennen, dass sich die Performance des D-MCS-Locks und des CLH-Locks für NUMA kaum ändert, wenn ein wenig Wartezeit hinzukommt. Der normale CLH-Lock wird hingegen mit zunehmender Wartezeit immer schneller, da die Prozesse (genau wie bei abnehmender kritischer Arbeit im CCWB) immer weniger Zeit mit entfernten Speicherzugriffen in der Warteschleife des Locks verbringen müssen.

Auch die Konkurrenz der Locks (Benchmark 5.7c) ist ähnlich zum CCWB: der D-MCS-Lock hat den geringsten kritischen Overhead und damit auch die geringste Konkurrenz, gefolgt vom normalen CLH-Lock. Der CLH-Lock für NUMA hat den höchsten kritischen Overhead und damit auch die höchste Konkurrenz.

Benchmark 5.7b bestätigt die Ergebnisse des UPB: Auch bei geringer Konkurrenz (durchschn. Wartezeit von 16 μs -32 μs) ist der Overhead des D-MCS-Locks nur etwa halb so groß wie der der beiden CLH-Locks. Die normale Variante ist hierbei etwas schneller als die NUMA-Variante, da zweitens eine langsamere Lockübergabe hat, wenn es einen Nachfolger gibt.

Zusammenfassend hat der CLH-Lock auf verteiltem Speicher eine sehr schlechte Performance bei hoher Konkurrenz und eine schlechte Performance bei niedriger Konkurrenz, weil er deutlich mehr entfernte Speicherzugriffe benötigt als der D-MCS-Lock. Die Variante des

CLH-Locks für NUMA aus [Cra93] verbessert die Performance bei hoher Konkurrenz zwar deutlich, ist aber trotzdem sowohl bei hoher als auch bei geringer Konkurrenz nur etwa halb so schnell wie der D-MCS-Lock.

Der HCLH-Lock basiert demnach auf einem Lock, der sich nicht performant auf verteilten Speicher portieren lässt. Selbst wenn man abweichend zu [LNS06] als Basis die NUMA-Variante des CLH-Locks verwenden würde, ist keine gute Performance zu erwarten. Aus diesem Grund wird der HCLH-Lock im Rahmen dieser Arbeit nicht auf verteilten Speicher portiert.

5.4 Cohort-Lock und HMCS-Lock

Die Idee von Cohort-Locks aus [DMS12] ist es, zwei Lock-Algorithmen zu kombinieren, um eine kleine Hierarchie zu erzeugen. Es gibt einen globalen Lock und pro NUMA-Knoten einen lokalen Lock. Um den kritischen Abschnitt auszuführen, muss ein Prozess sowohl den lokalen Lock seines NUMA-Knotens als auch den globalen Lock besitzen. Ist der Cohort-Lock frei, muss ein Prozess erst den lokalen und anschließend den globalen Lock akquirieren.

Wenn ein Prozess den Cohort-Lock freigibt, prüft er zunächst, ob ein anderer Prozess versucht, denselben lokalen Lock zu akquirieren. Diese Fähigkeit des lokalen Locks zu erkennen, ob ein lokaler Nachfolger existiert, wird Kohortenerkennung (engl. *cohort detection*) genannt. Ein Lock, der lokal eingesetzt werden soll, muss daher eine Funktion anbieten, die diese Prüfung durchführt. In [DMS12] wird hierfür eine Funktion mit dem Namen *alone?* (deutsch alleine) verwendet, die den booleschen Wert `false` zurückgibt, wenn ein lokaler Nachfolger existiert. Existiert ein lokaler Nachfolger, gibt der Vorgänger nur den lokalen, nicht aber den globalen Lock frei und signalisiert dem Nachfolger (z. B. über ein Feld *top-granted* im lokalen Lock), dass dieser den globalen Lock nicht akquirieren muss. Wenn kein lokaler Nachfolger existiert, wird zunächst der globale und dann der lokale Lock freigegeben, sodass der Cohort-Lock wieder vollkommen frei ist.

Beim Cohort-Lock passiert es häufig, dass ein Prozess den globalen Lock akquiriert, aber ein anderer Prozess (aus demselben NUMA-Knoten) ihn wieder freigibt. Ein Lock, der diese Nutzungsweise unterstützt, wird als prozessunabhängig (engl. *thread oblivious*) bezeichnet. Nicht jeder Lock-Algorithmus ist prozessunabhängig, denn normalerweise werden Locks von demselben Prozess akquiriert und wieder freigegeben. Für Locks, die global in einem Cohort-Lock eingesetzt werden sollen, ist das aber erforderlich.

Da Prozesse desselben NUMA-Knotens immer bevorzugt werden, könnte das dazu führen, dass andere Prozesse verhungern. Um das zu verhindern, wird in einer Variable im Cohort-Lock gezählt, wie oft in Folge der Lock direkt an einen lokalen Nachfolger übergeben wurde. Wenn dieser Zähler einen bestimmten Schwellwert erreicht, wird der Cohort-Lock komplett freigegeben, selbst wenn es einen lokalen Nachfolger gibt.

Mit leichten Modifikationen sind sehr viele Locks prozessunabhängig oder unterstützen Kohortenerkennung und können somit als globale bzw. lokale Locks verwendet werden. Und da globale und lokale Locks beliebig kombiniert werden können, gibt es sehr viele Cohort-Locks. In [DMS12] werden als globale und lokale Locks ein TTS-Lock mit Backoff [And90], ein Ticket-Lock [MCS91a] (kurz TKT-Lock) und MCS-Lock [MCS91b] verwendet.

Die notwendigen Modifikationen für eine korrekte Implementierung sind dabei meist klein. Bei einem lokalen MCS-Lock muss ein Prozess für die Kohortenerkennung lediglich in der Funktion `alone` prüfen, ob sich lokal bereits ein Nachfolger registriert hat. Dafür wird nur

ein lokaler Lesezugriff benötigt, analog zu der ersten Operation in der `release`-Funktion (Abbildung 3.5b, Zeile 1). Darüber hinaus gibt es aber noch einige Optimierungen, die von den Autoren angewandt werden. So wird z. B. bei einem lokalen MCS-Lock keine Variable im lokalen Lock benötigt, um dem Nachfolger zu signalisieren, dass der globale Lock nicht akquiriert werden muss. Stattdessen lässt sich das Statusfeld erweitern, auf dessen Änderung ein Nachfolger wartet. Bei einem normalen MCS-Lock kann dieses Feld nur zwei Werte annehmen: „akquiriert“ und „freigegeben“. Wenn man allerdings einen dritten Wert „lokal freigegeben“ hinzufügt, kann der Nachfolger direkt wissen, ob er den globalen Lock akquirieren muss oder nicht.

Cohort-Locks haben in [DMS12] immer zwei Hierarchiestufen. Es gibt allerdings in modernen Computern häufig eine tiefere Hierarchie von Ebenen mit schnellerer Kommunikation, nach der Prozessoren gruppiert werden können. Zwei Prozessoren können einen gemeinsamen L1-, L2- oder L3-Zwischenspeicher haben, in demselben NUMA-Knoten liegen, einen gemeinsamen Hauptspeicher besitzen, auf derselben Insel im Netzwerk eines Rechenzentrums sein oder noch weiter voneinander entfernt sein. Für jede dieser Hierarchiestufen könnte man einen eigenen Lock nutzen, um die Lokalität perfekt auszunutzen. Daher wird in [CFMC15] der Cohort-Lock mit globalem und lokalem MCS-Lock generalisiert, sodass beliebig viele MCS-Locks in einer Baumhierarchie kombiniert werden können. Dieser generalisierte Cohort-Lock wird Hierarchischer-MCS-Lock, kurz HMCS-Lock genannt.

5.4.1 Portierung und Optimierung des Cohort-Locks

Cohort-Locks eignen sich sehr gut für die Portierung auf verteilten Speicher, da die Aufteilung in zwei Ebenen (global und lokal) sehr gut zu MPI passt. Mit der Funktion `MPI_Comm_split_type` und dem Typ `MPI_COMM_TYPE_SHARED`, können Prozesse leicht danach gruppiert werden, ob sie gemeinsamen Speicher besitzen, also ob sie auf demselben Knoten laufen und es ist sogar möglich, wie in Abschnitt 3.6 beschrieben, den lokalen Lock komplett ohne RMA zu implementieren und stattdessen C++ `std::atomic` zu verwenden.

Eine Aufteilung in mehr als zwei Ebenen, wie beim HMCS-Lock, ist mit dem MPI-Standard alleine allerdings nicht möglich. Open-MPI bietet beispielsweise neben dem standardisierten `MPI_COMM_TYPE_SHARED` weitere Gruppierungen an, unter anderem `OMPI_COMM_TYPE_L1CACHE` für Prozesse mit gemeinsamem L1-Zwischenspeicher und `OMPI_COMM_TYPE_NUMA` für Prozesse auf demselben NUMA-Knoten. Intel-MPI hingegen bietet keine Möglichkeit solch einer Aufteilung, sodass hardwarespezifische Bibliotheken herangezogen werden müssten. Damit die Lock-Implementierungen dieser Arbeit allgemeingültig bleiben und nicht wie in [SBH16] nur auf bestimmter Hardware mit einer bestimmten MPI-Implementierung funktionieren, beschränkt sich die Portierung hier auf Cohort-Locks mit zwei Ebenen.

Optimierung 1: Lokaler Zähler

Ein großes Problem der Cohort-Locks aus [DMS12] ist die Verwendung eines globalen Zählers, um das Verhungern von Prozessen zu vermeiden. Bei der Portierung auf verteilten Speicher müssen für solch einen Zähler langsame entfernte Zugriffe getätigt werden. Tatsächlich ist es gar nicht nötig, dass alle Prozesse denselben Zähler verwenden. Wie bereits erläutert, zählt dieser Zähler, wie oft der Cohort-Lock an einen lokalen Nachfolger übergeben wurde, ohne den globalen Lock freizugeben. Erreicht der Zähler einen zuvor bestimmten Wert, wird er

zurückgesetzt und der globale Lock wird freigegeben. Auch wenn der globale Lock schon vorher freigegeben wird, weil kein lokaler Nachfolger existiert, muss der Zähler zurückgesetzt werden, damit der Lock auf dem nächsten Rechenknoten wieder möglichst oft lokal weitergegeben werden kann. Das bedeutet, dass alle Zugriffe auf den Zähler, bis er zurückgesetzt wird, durch Prozesse in derselben lokalen Gruppe ausgeführt werden. Es kann also auch ein Zähler pro Rechenknoten verwendet werden, um die entfernten Zugriffe zu vermeiden.

Optimierung 2: Direkte Zugriffe auf lokalen Zähler

Dadurch, dass es nun einen Zähler pro Rechenknoten gibt, können direkte Speicherzugriffe verwendet werden, statt lokale RMA-Operationen mit MPI. Hierbei werden auch keine atomaren Zugriffe benötigt, da der Zähler nur von dem Prozess benutzt wird, der gerade den Cohort-Lock hält. Die Zugriffe auf den Zähler sind daher bereits durch wechselseitigen Ausschluss geschützt.

Optimierung 3: Inline-Zähler

In [CFMC15] geht die Optimierung dieses Zählers für den HMCS-Lock noch einen Schritt weiter. Neben den Statuswerten „akquiriert“ und „freigegeben“, wird eine lokale Freigabe der MCS-Locks durch die Zahl der aufeinander folgenden Freigaben signalisiert. Für die Portierung auf verteilten Speicher wird ein Byte im Format `uint8_t` verwendet. Das Statusfeld kann also die Werte 0 bis 255 annehmen. Der Status (global) „freigegeben“ wird durch den Wert 0 und „akquiriert“ durch den Wert 255 repräsentiert. Die Zahlen 1 bis 254 signalisieren demnach eine lokale Freigabe und erlauben es gleichzeitig festzustellen, wie oft der Lock lokal weitergegeben wurde, ohne dass ein separater Zähler benötigt wird. Ein Prozess erhält den Status seines Vorgängers, wenn er den lokalen Lock akquiriert hat und kann damit prüfen, ob er den globalen Lock akquirieren muss und ob er den Lock lokal weitergeben darf. Falls er den Lock in `release` lokal weitergibt, schickt er seinem Nachfolger `status + 1` (siehe Abbildung 5.3).

```

1 constexpr uint8_t MAX_LOCAL_PASSES = 50;
2 uint8_t status; // Pro Prozess
3
4 void acquire() {
5     status = local_lock.acquire();
6     if (status == 0) {
7         global_lock.acquire();
8     }
9 }
```

(a) Felder und `acquire`

```

1 bool alone = local_lock.alone();
2 bool may_pass_local =
3     status < MAX_LOCAL_PASSES;
4 if (!alone && may_pass_local) {
5     local_lock.release(status + 1);
6 } else {
7     global_lock.release();
8     local_lock.release(0);
9 }
```

(b) Freigeben des Locks (`release`)

Abbildung 5.3: Cohort-Lock mit Inline-Zähler

Diese Optimierung ist auch in RMA-MCS aus [SBH16] enthalten, welcher eine Implementierung eines HMCS-Locks mit zwei Ebenen ist. Sie lässt sich aber auch auf andere Cohort-Locks übertragen. TAS-, TTS- und CLH-Locks nutzen genau wie der MCS-Lock ein zweiwertiges Statusfeld, welches analog erweitert werden kann. Und da diese Optimierung nur den lokalen Lock betrifft, kann weiterhin ein beliebiger globaler Lock eingesetzt werden.

Lokale Locks, die kein zweiwertiges Statusfeld für die Lockübergabe nutzen, können nicht so einfach von dieser Optimierung profitieren. Ein Beispiel hierfür ist der TKT-Lock [MCS91a]. Dieser nutzt zwei globale Zähler: `next_ticket` und `now_serving`. Beide sind initial 0.

```
1 int my_ticket = next_ticket.fetch_add(1);
2 while (my_ticket != now_serving.load());
```

(a) Akquirieren des Locks (`acquire`)

```
1 now_serving.fetch_add(1);
```

(b) Freigeben des Locks (`release`)

Abbildung 5.4: TKT-Lock

Zum Akquirieren des Locks (siehe Abbildung 5.4a) inkrementiert ein Prozess mit einer atomaren Operation den Zähler `next_ticket`. Dabei erhält er den vorherigen Wert. Diese eindeutige Zahl merkt er sich als sein Ticket (im Zähler `my_ticket`). Dann wartet er in einer Schleife so lange, bis der Zähler `now_serving` denselben Wert hat wie sein Ticket.

Zum Freigeben muss ein Prozess nur den Zähler `now_serving` inkrementieren (siehe Abbildung 5.4b). Dies kann entweder mit einer atomaren *Fetch-And-Increment* (FAI) Operation, oder mit einer Lese- und einer atomaren Schreib-Operation gemacht werden, da kein anderer Prozess gleichzeitig schreibend auf `now_serving` zugreifen kann.

Würde man einen Inline-Zähler in den TKT-Lock integrieren, müsste ein Prozess mit dem Zugriff auf `now_serving` gleichzeitig das aktuelle Ticket und die Anzahl von lokalen Lockübergaben erhalten. Dies ist aber nicht möglich, ohne einen größeren Datentyp zu verwenden, der für beide Informationen Platz hat. Nicht bei jedem Lock-Algorithmus kann ein größerer Datentyp verwendet werden, denn atomare Operationen können nicht mit beliebig großen Datentypen umgehen.

In dieser Arbeit wird daher darauf verzichtet diese Optimierung auf den TKT-Lock anzuwenden. Stattdessen wird hier der Zähler in das Feld *top-granted* integriert. Dieses *Flag* wird in [DMS12] den lokalen TKT-Locks eines Cohort-Locks hinzugefügt, damit ein Prozess seinem Nachfolger signalisieren kann, ob es sich um eine lokale Freigabe handelt. Dieses zweiwertige Feld wird wieder durch ein Byte im Format `uint8_t` ersetzt, wobei der Wert 0 eine globale Freigabe und die anderen Werte eine lokale Freigabe signalisieren. Genau wie beim Anwenden der Optimierung auf TAS-, TTS-, CLH- und MCS-Locks ist der Zähler so Teil des lokalen Locks. Diese Anpassung ist bei jedem lokalen Lock möglich, der von Optimierung 3 nicht profitieren kann und daher auf ein *top-granted-Flag* angewiesen ist. So wird zwar keine Verbesserung der Geschwindigkeit erzielt, aber es kann eine einheitliche Schnittstelle für die lokalen Locks und damit immer die Cohort-Lock-Implementierung aus Abbildung 5.3 verwendet werden.

Optimierung 4: Direkte Zugriffe auf globale MCS-Warteschlangenknoten

Bei der Portierung der globalen und lokalen Locks auf verteilten Speicher, müssen die Eigenschaften der Prozessunabhängigkeit bzw. Kohortenerkennung erhalten bleiben. Allerdings ist die Definition der Prozessunabhängigkeit in [DMS12] ungenau:

A lock x is thread-oblivious, if [...] for a lock method call of x by a given thread, it allows the matching unlock method call [...] to be executed by a different thread.

In dieser Definition wird nicht spezifiziert, von welchem anderen Thread/Prozess der Lock freigegeben werden darf. Damit ein Lock als globaler Lock eines Cohort-Locks verwendet

werden kann, reicht es, wenn unterstützt wird, dass ein Prozess den Lock akquiriert und ein anderer Prozess in derselben lokalen Gruppe den Lock freigibt. Es ist nicht erforderlich, dass ein Prozess in einer anderen lokalen Gruppe (z. B. auf einem anderen NUMA-Knoten) den Lock freigeben kann.

Darüber hinaus lässt sich beobachten, dass in jeder lokalen Gruppe immer höchstens ein Prozess gleichzeitig versucht, den globalen Lock zu akquirieren, da zunächst der lokale Lock akquiriert werden muss. Der globale Lock wird außerdem immer vor dem lokalen Lock freigegeben und damit garantiert, bevor ein Prozess der lokalen Gruppe erneut versuchen kann, ihn zu akquirieren. Jede lokale Gruppe von Prozessen agiert aus Sicht des globalen Locks demnach wie ein einziger Prozess. Daher können sich bei einem globalen MCS-Lock alle Prozesse einer lokalen Gruppe einen Warteschlangenknoten teilen. Auf diese Weise wird im HMCS-Lock aus [CFMC15] und damit auch im RMA-MCS ein globaler MCS-Lock implementiert. Dies ist eine deutliche Vereinfachung gegenüber [DMS12]. Dort wurde ein deutlich komplizierter Algorithmus genutzt, durch den der globale MCS-Lock vollständig prozessunabhängig ist und es kein Problem wäre, wenn mehrere Prozesse einer lokalen Gruppe gleichzeitig versuchen würden, den globalen Lock zu akquirieren. Das ist aber für einen Cohort-Lock nicht notwendig.

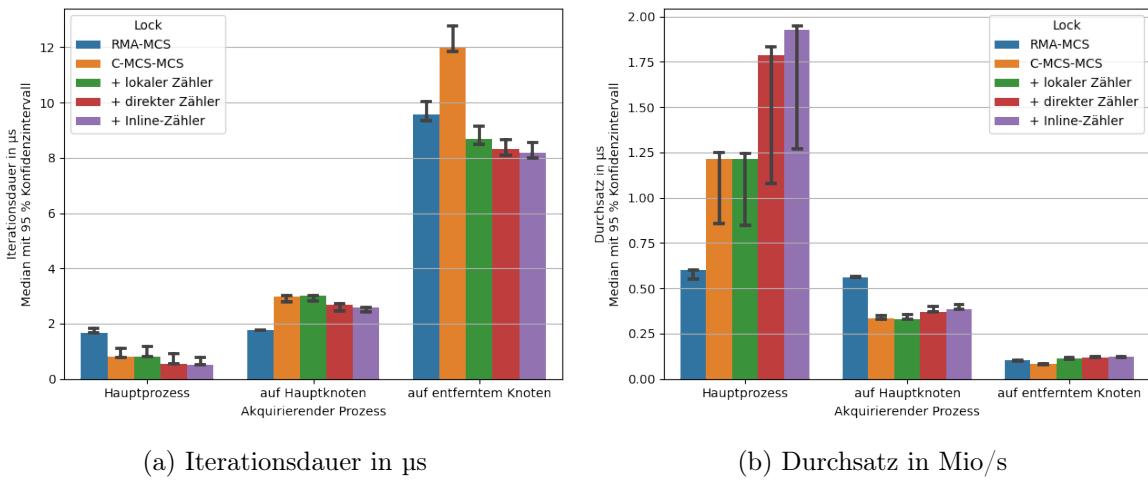
Da sich alle Prozesse einer lokalen Gruppe einen Warteschlangenknoten teilen, wird der hierfür benötigte Speicher nur auf einem dieser Prozesse allokiert, dem lokalen Hauptprozess. Ein Cohort-Lock mit globalem MCS-Lock hat daher einen globalen Hauptprozess, in dessen Speicher sich der Zeiger auf das Ende der Warteschlange befindet und einen lokalen Hauptprozess pro Rechenknoten, der den Warteschlangenknoten beinhaltet. Für das Einreihen in die Warteschlange nutzt ein Prozess nicht mehr seine eigene ID, sondern stellvertretend die seines lokalen Hauptprozesses, damit entfernte RMA-Zugriffe den richtigen Speicher modifizieren. Weil sich der Warteschlangenknoten und damit auch das Statusfeld nicht unbedingt im Speicher des Prozesses befinden, der versucht, den globalen MCS-Lock zu akquirieren, sondern in dessen lokalen Hauptprozess, wird in RMA-MCS aus [SBH16] mittels RMA darauf zugegriffen.

Die Zugriffe auf den Warteschlangenknoten sind in RMA-MCS zwar lokal, aber wie in Unterabschnitt 4.3.2 gezeigt wurde, sind direkte Speicherzugriffe deutlich schneller als lokale RMA-Operationen. Da die Prozesse einer lokalen Gruppe auf demselben Rechenknoten laufen, verfügen sie über gemeinsamen Speicher und auf diesen kann auch direkt zugegriffen werden, indem ein Fenster mit `MPI_Win_allocate_shared` erzeugt wird, wie in Abschnitt 3.6 beschrieben. So können viele RMA-Zugriffe vermieden werden, wodurch die Implementierung dieser Arbeit effizienter als RMA-MCS ist.

Die Nutzung von Punkt-zu-Punkt-Kommunikation zur Übergabe des Locks an einen Nachfolger ist auch mit dieser Optimierung weiterhin möglich, obwohl ein Prozess nicht die ID seines Vorgängers, sondern die dessen lokalen Hauptprozesses erhält. Auf eine Nachricht von dieser ID kann nicht gewartet werden, da der Hauptprozess nicht der ist, der den Lock freigeben wird. Stattdessen muss mit `MPI_Recv` auf eine Nachricht von `MPI_ANY_SOURCE` gewartet werden. Der Vorgänger benötigt allerdings die konkrete ID seines Nachfolgers, um eine Nachricht mit `MPI_Send` senden zu können, weshalb ein Nachfolger hier seine echte ID an den Vorgänger melden muss (Abbildung 4.3a, Zeile 16-19).

5.4.2 Evaluation der Optimierungen des Cohort-Locks

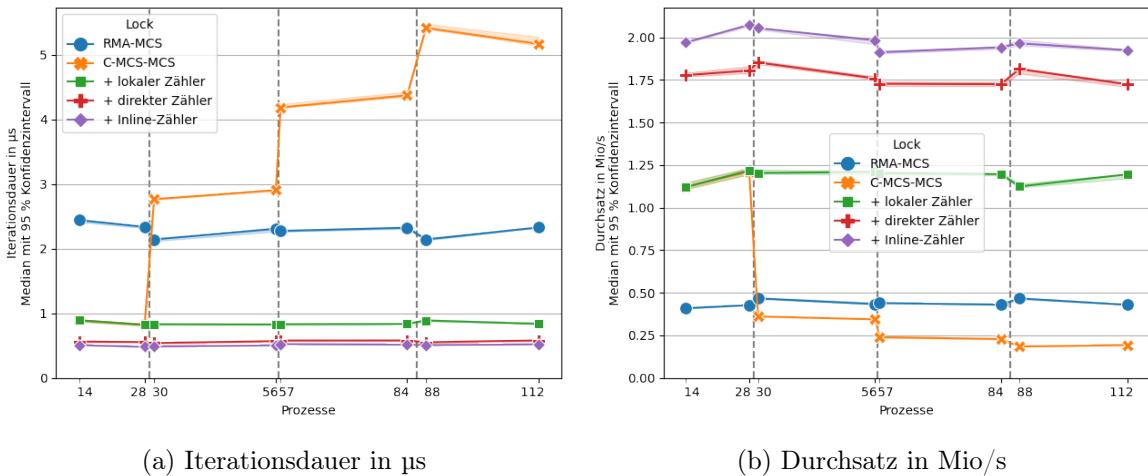
Für die Evaluation der vier vorgeschlagenen Optimierungen wird ein Cohort-Lock mit globalem und lokalem MCS-Lock (kurz C-MCS-MCS) verwendet. Das ermöglicht einen aussagekräftigen Vergleich mit RMA-MCS, welches ebenfalls ein Cohort-Lock ist, der aus zwei MCS-Locks besteht. Außerdem ist die vierte Optimierung nur auf Cohort-Locks mit globalem MCS-Lock anwendbar. Als globaler Lock wird der optimierte MCS-Lock mit `MPI_Put` basierend auf `dash::Mutex` aus Unterabschnitt 4.3.2 verwendet, da diese Variante mit Punkt-zu-Punkt-Kommunikation bei der Beteiligung von mehreren Rechenknoten am schnellsten ist (vgl. Benchmark 4.12b). Der lokale Lock hingegen ist der optimierte MCS-Lock mit `C++ std::atomic` basierend auf D-MCS aus Unterabschnitt 4.3.1, da dieser am schnellsten bei nur einem beteiligten Rechenknoten ist (vgl. Benchmark 4.12a). Alle Locks erlauben maximal 50 lokale Lockübergaben bevor der globale Lock freigegeben werden muss.



Benchmark 5.8: UPB der Cohort-Optimierungen

Benchmark 5.8 zeigt Latenz und Durchsatz freier Locks. Überraschenderweise ist die Performance nur bei RMA-MCS unabhängig davon, ob der akquirierende Prozess selbst der Hauptprozess ist. Es ist leider nicht klar, warum das bei den anderen Cohort-Locks einen Unterschied macht. Die separaten Evaluationen des globalen und lokalen Locks in Unterabschnitt 4.3.2 und Unterabschnitt 4.3.1 zeigen kein derartiges Verhalten und der Code des Cohort-Locks macht keinen Unterschied zwischen Prozessen auf demselben Knoten (vgl. Abbildung 5.3). Der einzige andere Unterschied zwischen RMA-MCS und C-MCS-MCS mit Inline-Zähler ist Optimierung 4: Bei RMA-MCS greifen die Prozesse mit lokalen RMA-Operationen auf ihre Warteschlangenknoten des globalen MCS-Locks zu, während bei C-MCS-MCS direkte Speicherzugriffe verwendet werden. Dass lokale RMA-Zugriffe auf den Speicher eines anderen Prozesses schneller sind als direkte Speicherzugriffe, ist aber nicht plausibel, da direkte Zugriffe auf den Zähler des C-MCS-MCS-Locks durch Optimierung 2 (direkter Zähler) auch dann zu einer besseren Performance führen, wenn der akquirierende Prozess nicht der Hauptprozess ist, sondern nur auf dem Hauptknoten läuft (vgl. Benchmark 5.8a). Die hier vorgeschlagenen optimierten C-MCS-MCS-Locks sind in dem zweiten UPB-Szenario zwar langsamer als RMA-MCS, dafür sind sie in den anderen beiden Szenarien aber schneller.

Darüber hinaus zeigt Benchmark 5.8a, dass alle Optimierungen des Zählers (Optimierung 1, 2 und 3) gewinnbringend sind. Wenn der akquirierende Prozess nicht auf dem Hauptknoten läuft, ist besonders die Nutzung eines lokalen Zählers pro Rechenknoten (Optimierung 1) von Vorteil, da entfernte Zugriffe vermieden werden. Wenn sich der Hauptprozess und damit der Zähler auf demselben Knoten befinden, macht diese Optimierung hingegen keinen Unterschied, da sowieso alle Zugriffe lokal sind. Optimierung 2 und 3 führen in allen drei Szenarien zu kleinen Verbesserungen der Geschwindigkeit. Diese Optimierungen des lokalen Zählers bewirken unabhängig vom Szenario eine konstante Verringerung der Iterationsdauer und wirken sich somit besonders bei geringer Iterationsdauer stark auf den Durchsatz aus (vgl. Benchmark 5.8b).

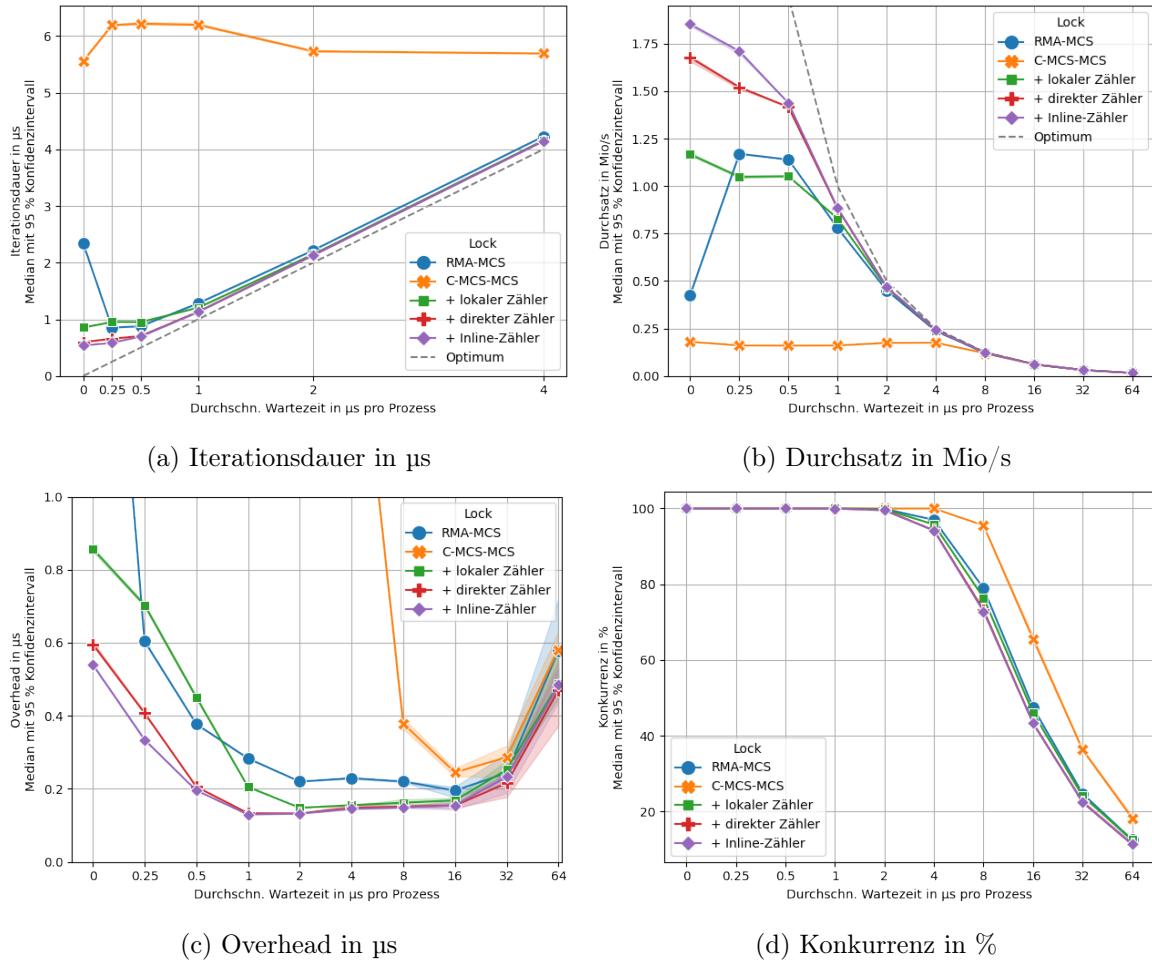


Benchmark 5.9: ECSB der Cohort-Optimierungen

In Benchmark 5.9 zeigt der ECSB, dass auch bei maximaler Konkurrenz jede der Zähler-Optimierungen den Lock verbessert. Das ist auch zu erwarten, da alle Zugriffe auf den Zähler im kritischen Abschnitt des globalen und lokalen Locks liegen. Auch bei diesem Benchmark wird deutlich, dass Optimierung 1 keinen Unterschied macht, wenn nur ein Rechenknoten beteiligt ist.

Obwohl RMA-MCS eine Implementierung eines HMCS-Locks ist und damit einen Inline-Zähler verwendet, ist der C-MCS-MCS-Lock mit Inline-Zähler etwa vier Mal so schnell. Beides sind Implementierungen des gleichen Algorithmus, aber die Optimierungen des lokalen und globalen Locks aus Abschnitt 4.3, zusammen mit Optimierung 4 sind bei hoher Konkurrenz offenbar sehr wirkungsvoll.

Der CCWB wird hier nicht gezeigt, da durch den Overhead der kritischen und unkritischen Arbeit, der Unterschied zwischen den Locks dort kaum zu sehen ist. Nur der C-MCS-MCS-Lock mit globalem Zähler sticht dort negativ hervor, das zeigen alle anderen Benchmarks aber bereits klar.



Benchmark 5.10: WBAB der Cohort-Optimierungen mit 112 Prozessen

In Benchmark 5.10 zeigt auch der WBAB, dass jede der Zähler-Optimierungen zu einer Verbesserung führt. In Benchmark 5.10a und Benchmark 5.10b sieht man, dass die Geschwindigkeit von RMA-MCS mit etwas Wartezeit außerhalb des kritischen Abschnitts stark zunimmt. Bei einer Wartezeit von 0,25 µs und 0,5 µs überholt er sogar den C-MCS-MCS-Lock mit lokalem Zähler und der C-MCS-MCS-Lock mit Inline-Zähler ist nur noch fast doppelt so schnell wie RMA-MCS. Das ist ein gutes Beispiel dafür, dass der ECSB unrealistische Ergebnisse liefern kann. In der Realität würde immer etwas Zeit zwischen kritischen Abschnitten vergehen, wodurch der RMA-MCS-Lock besser ist, als er im ECSB aussieht. Dennoch ist der Overhead des optimierten C-MCS-MCS-Locks auch im WBAB bei jedem Konkurrenzlevel deutlich geringer (vgl. Benchmark 5.10c und Benchmark 5.10d), was auch zu einem höheren Durchsatz führt (vgl. Benchmark 5.10b).

5.4.3 Evaluation der verschiedenen Cohort-Locks

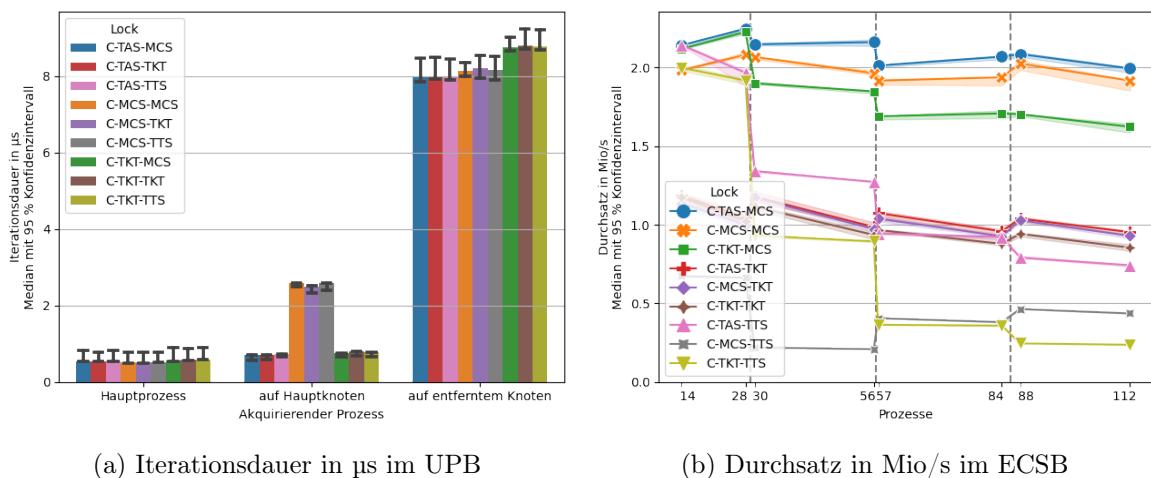
Nun werden Cohort-Locks bestehend aus verschiedenen Locks evaluiert. In [DMS12] wurden global und lokal drei verschiedene Locks verwendet: (1) Ein TTS-Lock mit Backoff (siehe Unterabschnitt 2.1.1), (2) ein TKT-Lock aus [MCS91a] (3) und ein MCS-Lock. Damit lassen

sich durch Kombination insgesamt $3 \cdot 3 = 9$ Cohort-Locks konstruieren. Von diesen wurden in [DMS12] nur fünf untersucht. Im Folgenden werden hingegen alle neun Varianten betrachtet.

Anders als in [DMS12] wird in dieser Arbeit statt eines globalen TTS-Locks ein TAS-Lock verwendet. Da entfernte Zugriffe deutlich langsamer sind, ist der Overhead der zusätzlichen Lese-Operationen im TTS-Lock größer als der Nutzen durch eine geringere Anzahl an atomaren Operationen. Außerdem ist ein globaler TAS-Lock etwas fairer als ein TTS-Lock: Prozesse, die nicht auf dem Hauptknoten laufen, haben im Falle eines Konflikts beim TAS-Lock eine etwas höhere Chance, den Lock zu akquirieren, da sie nur einen langsamen entfernten Zugriff benötigen. Beim TTS-Lock benötigen sie hingegen zwei: einen Test, ob der Lock frei ist und eine TAS-Operation, um den Lock zu akquirieren.

Für die lokalen TTS-Locks wird ein minimaler Backoff von 2^{10} ns und ein maximaler Backoff von 2^{14} ns verwendet. Für die globalen TAS-Locks hingegen ein Backoff von 1 ns bis 2^8 ns. Obwohl die RMA-Operationen des globalen TAS-Locks nicht zwischengespeichert werden, verbessert der Backoff die Geschwindigkeit und Fairness. Für die TKT-Locks wird kein Backoff verwendet. Da diese in der Warteschleife nur Lese-Operationen ausführen, hat Backoff in den Benchmarks dieser Arbeit zu keiner Verbesserung geführt.

Alle Cohort-Locks nutzen die Optimierungen aus Unterabschnitt 5.4.1. Optimierung 4 ist dabei nur auf Cohort-Locks mit globalem MCS-Lock anwendbar. Die Implementierungen werden als „C-Global-Lokal“ bezeichnet, wobei „Global“ der Name des globalen Locks und „Lokal“ der Name des lokalen Locks ist. Z. B. bezeichnet C-TAS-MCS einen Cohort-Lock mit globalem TAS- und lokalem MCS-Lock.



Benchmark 5.11: UPB & ECSB verschiedener Cohort-Locks

Benchmark 5.11a zeigt die Geschwindigkeit der neun Cohort-Locks im UPB. Die Locks sind in diesem Benchmark nach globalem und anschließend nach lokalem Lock sortiert, da die Geschwindigkeit in diesem Benchmark vor allem vom globalen Lock abhängt. Da im UPB die Locks immer frei sind, muss in jeder Iteration sowohl der lokale als auch der globale Lock akquiriert werden. Da nur der globale Lock langsame entfernte Zugriffe benötigt, ist er der entscheidende Faktor für die Geschwindigkeit.

Wenn der akquirierende Prozess nicht auf dem Hauptknoten läuft, sind Cohort-Locks mit globalem TAS-Lock am schnellsten. Darauf folgen die Cohort-Locks mit globalem MCS-Lock und schließlich die mit globalem TKT-Lock. Genau wie in Benchmark 5.8a sieht man bei

globalen MCS-Locks den Effekt, dass andere Prozesse auf dem Hauptknoten langsamer sind als der Hauptprozess. Ansonsten gibt es keinen großen Unterschied zwischen den Locks.

In den weiteren Benchmarks sind die Locks zuerst nach lokalem und anschließend nach globalem Lock geordnet. Es werden aber dieselben Farben verwendet. Im ECSB (siehe Benchmark 5.11b) hängt die Geschwindigkeit vor allem vom lokalen Lock ab. Da in diesem Benchmark maximale Konkurrenz vorliegt, werden die Cohort-Locks immer 50 Mal lokal weitergegeben, ohne dass der globale Lock akquiriert oder freigegeben werden muss. Der lokale Lock wird im ECSB also deutlich öfter verwendet als der globale Lock und hat daher einen größeren Einfluss als im UPB. Am schnellsten sind die Cohort-Locks mit lokalem MCS-Lock (C-*‐MCS) mit einem Durchsatz von knapp 2 Millionen kritischen Abschnitten pro Sekunde. Bei der Nutzung von vier Rechenknoten (88 bis 112 Prozesse) folgen dann die Cohort-Locks mit lokalem TKT-Lock (C-*‐TKT). Diese sind etwa halb so schnell. Cohort-Locks mit einem lokalen TTS-Lock (C-*‐TTS) sind auf vier Rechenknoten am langsamsten. Während der Durchsatz von C-*‐MCS und C-*‐TKT relativ unabhängig von der Anzahl der Rechenknoten ist, verschlechtert sich die Performance von C-TAS-TTS und C-TKT-TTS mit zunehmender Anzahl von Rechenknoten deutlich. Bei C-MCS-TTS nimmt sie hingegen leicht zu.

Betrachtet man nur die Cohort-Locks mit lokalem MCS-Lock (C-*‐MCS), so hängt die Geschwindigkeit vom globalen Lock ab. Dabei ist ein globaler TAS-Lock am schnellsten, gefolgt vom MCS- und schließlich dem TKT-Lock. Dies entspricht genau der beobachteten Reihenfolge im UPB. Genau dasselbe lässt sich auch bei lokalen TKT-Locks und auf vier Rechenknoten auch bei lokalen TTS-Locks beobachten. Bei hinreichend vielen Rechenknoten gibt es im ECSB also eine strikte Ordnung der globalen Locks und eine strikte Ordnung der lokalen Locks: Global ist ein TAS-Lock immer am besten (gefolgt von einem MCS- und TKT-Lock), lokal ein MCS-Lock (gefolgt von einem TKT- und TTS-Lock).

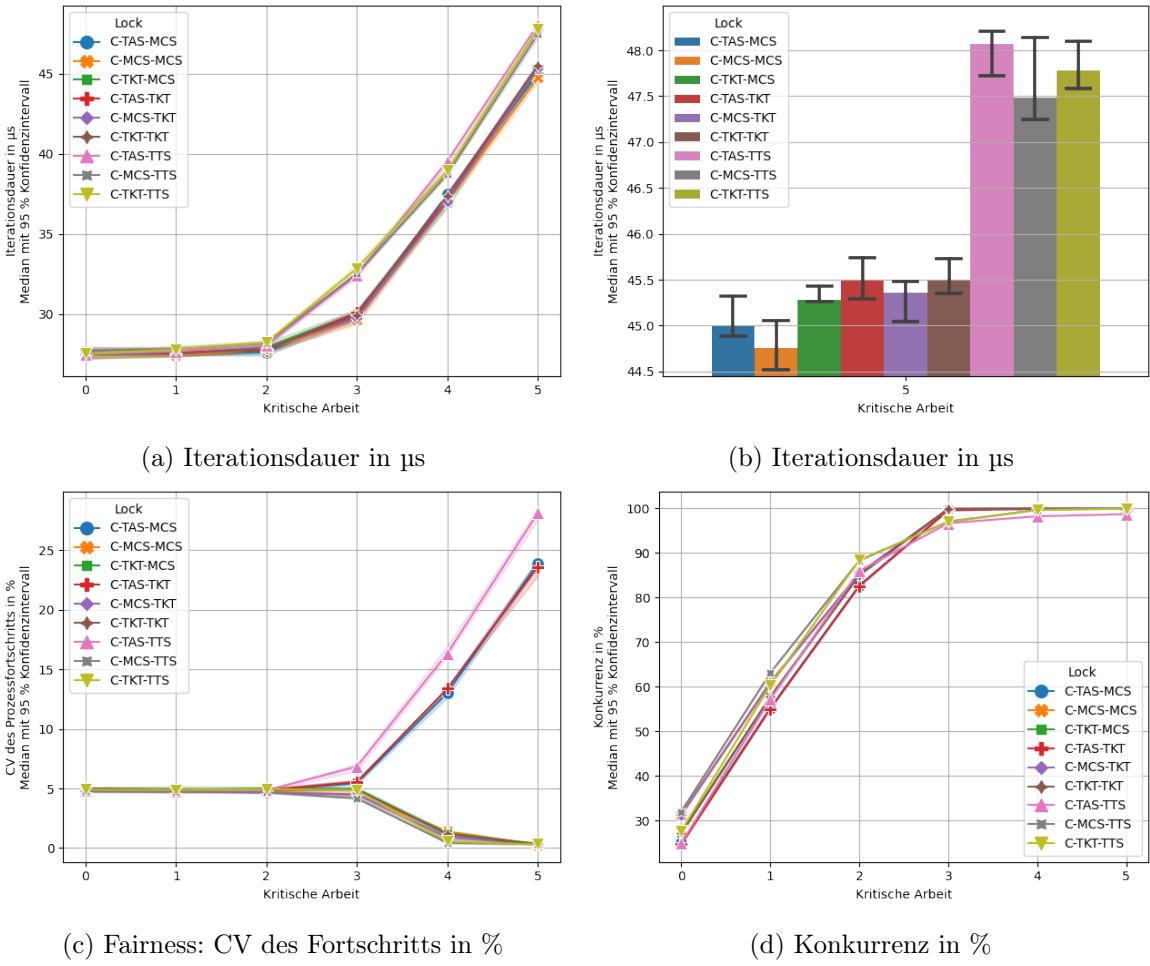
Vergleicht man die Ergebnisse aus Benchmark 5.11b mit denen aus [DMS12], findet man eine große Übereinstimmung: Die Geschwindigkeitsreihenfolge der fünf untersuchten Locks vom schnellsten zum langsamsten war dort: C-TTS-MCS, C-TKT-MCS, C-MCS-MCS, C-TKT-TKT und schließlich C-TTS-TTS. Dabei hatten C-TKT-MCS und C-MCS-MCS in [DMS12] eine nahezu identische Performance. In dieser Arbeit ist C-MCS-MCS schneller als C-TKT-MCS, ansonsten stimmen die Ergebnisse aber überein.

In Benchmark 5.12 sind die Ergebnisse des CCWB zu sehen. Da die Unterschiede zwischen den Locks in Benchmark 5.12a schwer zu erkennen sind, zeigt Benchmark 5.12b die Geschwindigkeit bei einer kritischen Arbeit von 5.

Im Gegensatz zum ECSB sind im realistischeren CCWB Cohort-Locks mit globalen TAS-Locks etwas langsamer als die entsprechenden Locks mit globalen MCS-Locks, der C-TAS-TTS-Lock ist sogar langsamer als der C-TKT-TTS-Lock. Ansonsten sind die Ergebnisse aber analog. Ein Blick auf die Fairness in Benchmark 5.12c liefert eine Erklärung für die schlechtere Performance der C-TAS-*‐Locks: Sobald die Locks bei einer kritischen Arbeit von 3 das Equilibrium erreichen, wodurch sie einer Konkurrenz von 100 % ausgesetzt sind (siehe Benchmark 5.12d), beginnen sie immer unfairer zu werden. Da Prozesse auf dem Hauptknoten die globalen Locks schneller akquirieren können, haben sie einen unfairen Vorteil. MCS- und TKT-Locks garantieren bei hinreichender Konkurrenz Fairness, sodass C-MCS-*‐ und C-TKT-*‐Locks ab dem Equilibrium fairer werden. Ein globaler TAS-Lock hingegen verhindert nicht, dass Prozesse auf dem Hauptknoten immer wieder anderen Prozessen zuvorkommen.

Im ECSB wirkt sich die Unfairness nicht so negativ auf die Geschwindigkeit aus (vgl. Benchmark 5.11b), da es keine unkritische Arbeit und damit keinen parallelisierbaren Programmteil gibt. Die maximale Geschwindigkeit kann also theoretisch erreicht werden, indem

5 Portierung von NUMA-Locks auf verteilten Speicher

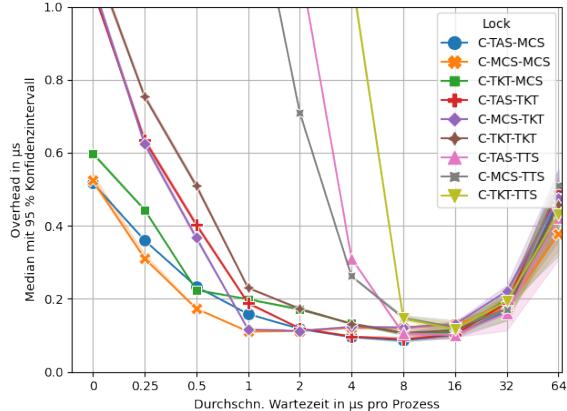


Benchmark 5.12: CCWB verschiedener Cohort-Locks mit 112 Prozessen

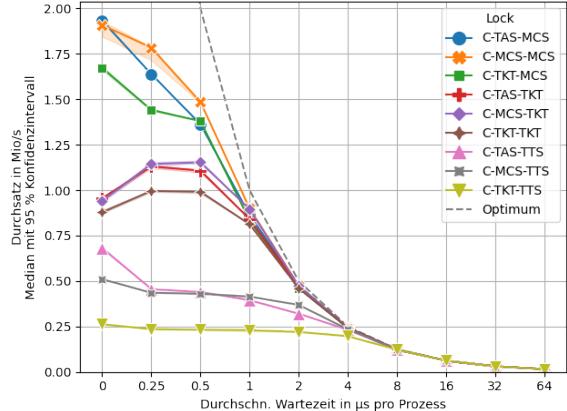
ausschließlich derselbe Prozess immer wieder den leeren kritischen Abschnitt ausführt. Im CCWB ist das anders: Würde hier nur ein Prozess voran schreiten, dann würde auch die unkritische Arbeit sequentiell ausgeführt werden, was zu einer sehr schlechten Performance führen würde. Da die Unfairness in Benchmark 5.12c nur bis etwa 25 % steigt, ist klar, dass so ein extremer Fall hier nicht vorliegt. Aber auch in weniger extremen Fällen kann Unfairness zu einer schlechteren Verteilung der unkritischen Arbeit und damit zu einer schlechteren Parallelisierung führen, was sich negativ auf die Geschwindigkeit auswirkt.

Die große Menge an unkritischer Arbeit im CCWB ist auch einer der Gründe dafür, dass C-MCS-TTS und C-TKT-TTS sehr fair sind, obwohl die lokalen Locks keine Fairness garantieren. Sie verhindert bei hoher Konkurrenz, dass derselbe Prozess mehrmals in Folge den Cohort-Lock akquirieren kann, und durch die fairen globalen Locks wird auch verhindert, dass Prozesse desselben Rechenknotens mehrmals in Folge den Cohort-Lock akquirieren können. Zusätzlich hat bei lokalen TTS-Locks der Hauptprozess einen weniger großen Vorteil beim Akquirieren. Dieser kann zwar möglicherweise etwas schneller auf seinen Speicher zugreifen, dieser Unterschied ist aber bei weitem nicht so ausgeprägt wie der Geschwindigkeitsunterschied zwischen lokalen und entfernten RMA-Operationen.

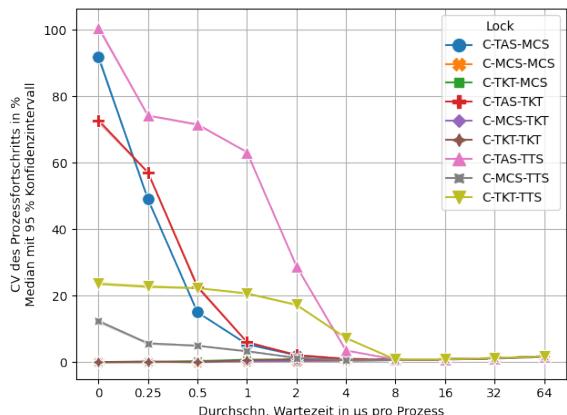
5.4 Cohort-Lock und HMCS-Lock



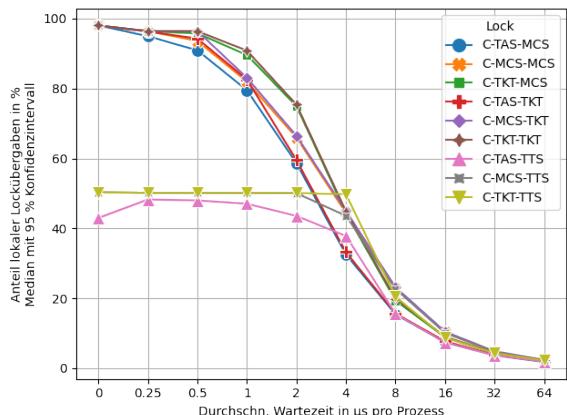
(a) Overhead in μ s



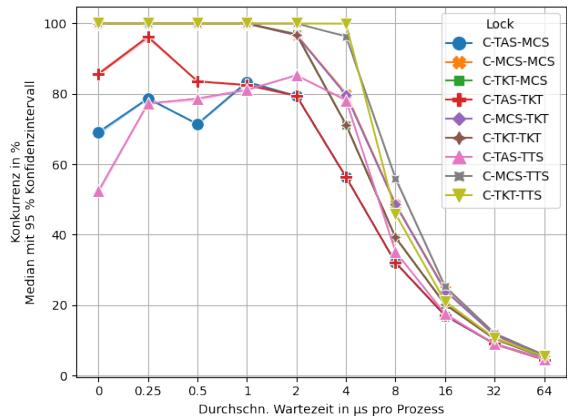
(b) Durchsatz in Mio/s



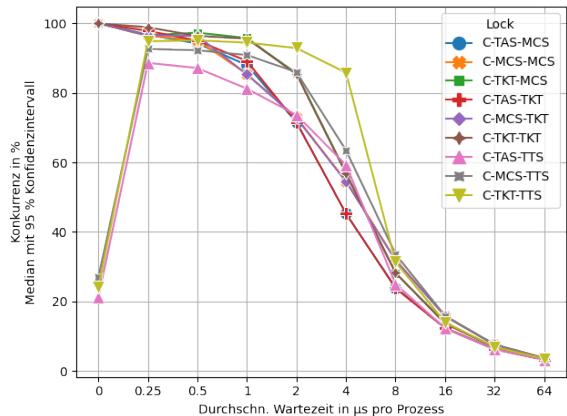
(c) Fairness: CV des Fortschritts in %



(d) Anteil lokaler Lockübergaben in %



(e) Globale Konkurrenz in %



(f) Lokale Konkurrenz in %

Benchmark 5.13: WBAB verschiedener Cohort-Locks mit 112 Prozessen

Benchmark 5.13 zeigt die Performance der Locks im WBAB. Benchmark 5.13b bestätigt die Beobachtungen aus dem CCWB: Ohne Wartezeit gleicht der WBAB dem ECSB, daher sind C-TAS-*-Locks am schnellsten, sobald aber unkritische Arbeit (in Form von Wartezeit) hinzukommt, werden sie von den entsprechenden C-MCS-*-Locks überholt.

Am Overhead in Benchmark 5.13a sieht man gut den wechselnden Einfluss der globalen und lokalen Locks. Bei geringer Wartezeit (0 µs bis 0,5 µs) ist wie beim ECSB der lokale Lock ausschlaggebend für die Performance: Den geringsten Overhead haben die C-*-MCS-Locks, gefolgt von den C-*-TKT-Locks. Bei höherer Wartezeit (2 µs bis 8 µs) hingegen ist wie beim UPB der globale Lock entscheidend: C-MCS-MCS und C-MCS-TKT haben den gleichen Overhead, ebenso wie C-TAS-MCS und C-TAS-TKT und auch C-TKT-MCS und C-TKT-TKT. Die Cohort-Locks mit lokalem TTS-Lock (C-*-TTS) sind so langsam, dass dieser Effekt bei ihnen nicht mehr beobachtet werden kann, bevor die Wartezeit so hoch ist, dass die Messung ungenau wird.

Benchmark 5.13c zeigt, dass auch im WBAB die Fairness von C-TAS-*-Locks extrem schlecht ist. Ohne Wartezeit steigt der Variationskoeffizient auf bis zu 100 %. Um das näher zu untersuchen wurde die Konkurrenz separat für die globalen und lokalen Locks in den Cohort-Locks gemessen.

Die globale Konkurrenz (vgl. Benchmark 5.13e) zeigt, dass bei geringer Wartezeit die globalen TAS-Locks eine sehr geringe Konkurrenz haben, während diese bei den anderen Locks maximal ist. Das ist ein klares Indiz dafür, dass der Lock wiederholt von Prozessen desselben Knotens, vermutlich des Hauptknotens, akquiriert wird. Das erklärt auch die extreme Unfairness.

Bei der lokalen Konkurrenz (vgl. Benchmark 5.13f) ist besonders auffällig, dass ohne Wartezeit die lokalen TTS-Locks eine sehr geringe Konkurrenz haben. Auch hier deutet dies darauf hin, dass häufig derselbe Prozess mehrmals hintereinander den lokalen Lock akquiriert. In diesem Fall wirkt sich das aber kaum auf die Fairness aus. C-MCS-TTS und C-TKT-TTS sind zwar bei einer Wartezeit von 0 unfair, aber nicht viel unfairer als bei einer etwas höheren Wartezeit.

Um zu erklären, warum die Unfairness der C-*-TTS-Locks erst bei einer Wartezeit von 8 µs verschwindet, zeigt Benchmark 5.13d den Anteil der lokalen Lockübergaben, also wie oft der Cohort-Lock direkt an einen lokalen Nachfolger übergeben wird, ohne den globalen Lock freizugeben. Hier wird das Problem dieser Locks deutlich: Bei einer Wartezeit von unter 8 µs überschreitet der Anteil der lokalen Lockübergaben niemals 50 %. Das liegt daran, dass die Kohortenerkennung im TTS-Lock bei kurzen kritischen Abschnitten nicht korrekt funktioniert, wodurch sich lokale und globale Freigaben immer abwechseln.

Um Kohortenerkennung im TTS-Lock zu ermöglichen, wurde dieser in [DMS12] um das *Flag successor-exists* erweitert. Bevor ein Prozess beginnt, den Lock zu akquirieren, setzt er dieses *Flag* auf **true**, um den Anderen zu signalisieren, dass er auf den Lock wartet. Sobald der Prozess den Lock akquiriert, setzt er dieses *Flag* zurück auf **false**. Wenn viele Prozesse auf den Lock warten, würde dieses *Flag* nun fälschlicherweise auf **false** stehen, obwohl weitere lokale Nachfolger vorhanden sind. Daher prüft jeder Prozess in der Warteschleife auch das *successor-exists-Flag* und setzt es auf **true**, wenn er beobachtet, dass es zurückgesetzt wurde.

Bei einem sehr kurzen kritischen Abschnitt haben die anderen Prozesse keine Zeit, das *successor-exists-Flag* zurück auf **true** zu setzen, bevor der akquirierende Prozess den Lock wieder freigibt. Das führt dann zu einer globalen Freigabe. Der Grund, weshalb jede zweite Freigabe trotzdem lokal ist, ist, dass nach einer globalen Freigabe erst der lokale und anschließend der globale Lock akquiriert wird. Nachdem ein Prozess den lokalen Lock akquiriert, muss er also erst auf den globalen Lock warten und in dieser Zeit können die anderen Prozesse das *successor-exists-Flag* zurück auf **true** setzen. Das erklärt auch, warum der C-TAS-TTS-Lock etwas weniger als 50 % lokale Lockübergaben macht. Da der globale TAS-Lock unfair ist, kann es passieren, dass ein Prozess ihn freigibt und der nächste Prozess desselben Knotens

ihn direkt wieder akquiriert. Dieser muss dann nicht auf den globalen Lock warten, sodass die anderen Prozesse des Knotens trotz der globalen Freigabe keine Zeit haben, das *successor-exists-Flag* zurück auf `true` zu setzen. In so einem Fall folgt auf eine globale Freigabe keine lokale, sondern eine weitere globale Freigabe.

Die große Menge an globalen Freigaben erklärt auch die extrem schlechte Geschwindigkeit der C-* TTS-Locks in Benchmark 5.13a und Benchmark 5.13b.

Insgesamt geht der C-MCS-MCS als Gewinner aus der Evaluation hervor. Unfaire globale Locks, wie der TAS-Lock, sollten auf verteiltem Speicher vermieden werden, da diese bei hoher Konkurrenz zu extremer Unfairness führen können, und lokale TTS-Locks führen zu extrem schlechter Performance bei kurzen kritischen Abschnitten und können auch sonst nicht mit den anderen Alternativen mithalten. Von den verbleibenden vier Cohort-Locks, bestehend aus MCS- und TKT-Locks, schneidet der C-MCS-MCS in allen Benchmarks am besten ab¹.

5.4.4 Evaluation von lokalen Warteschlangen-Locks

In Unterabschnitt 5.4.3 ist der C-MCS-MCS-Lock als klarer Gewinner hervor gegangen. So-wohl global als auch lokal verwendet dieser einen Lock mit Warteschlange, den MCS-Lock. Dabei ist der MCS-Lock nicht unbedingt der schnellste Warteschlangen-Lock: In [LNS06] wird der CLH-Lock als der effizienteste Warteschlangen-Lock für Rechner mit Zwischenspeicher-Kohärenz-Mechanismus bezeichnet und in den Evaluationen des MCS- und CLH-Locks in [RH02] und [DMS11] schneidet der CLH-Lock etwas besser ab als der MCS-Lock.

Außerdem gibt es einen sehr neuen Warteschlangen-Lock, den diese Arbeiten noch gar nicht berücksichtigen konnten: den Hemlock aus [DK22]. Dieser ähnelt sehr dem CLH-Lock, nutzt aber auch ein paar Techniken aus dem MCS-Lock und enthält einen ganz neuen Ansatz: Beim Hemlock gibt es nur einen Warteschlangenknoten pro Prozess für alle Lock-Instanzen. Dadurch benötigt der Hemlock deutlich weniger Speicher: Jede zusätzliche Instanz eines Hemlocks benötigt nur einen `tail`-Zeiger.

Genau wie beim CLH-Lock (vgl. Abbildung 5.1a) enthält ein Warteschlangenknoten des Hemlocks nur ein `status`-Feld, auf dessen Änderung ein Nachfolger wartet. Während beim CLH-Lock höchstens ein Nachfolger auf die Änderung eines `status`-Feldes wartet, teilen sich beim Hemlock alle Lock-Instanzen dieselben Warteschlangenknoten. Bei zwei Hemlock-Instanzen ist es daher möglich, dass zwei verschiedene Prozesse auf eine Änderung desselben `status`-Feldes warten, wenn der Vorgänger der beiden Prozesse beide Lock-Instanzen akquiriert hat. Aus diesem Grund reicht beim Hemlock kein boolesches *Flag*, wie es im CLH-Lock eingesetzt wird. Statt einem booleschen `false` schreibt ein Vorgänger bei der Freigabe eines Hemlocks die eindeutige Speicheradresse der Lock-Instanz in das `status`-Feld. So kann jeder Nachfolger prüfen, ob die Lockübergabe an ihn gerichtet ist. Dieser Nachfolger setzt das `status`-Feld dann zurück auf den speziellen Zeiger `NULL`, sodass es für die nächste Lockübergabe frei ist.

Wie Unterabschnitt 5.3.3 gezeigt hat, eignet sich der CLH-Lock nicht für verteilten Speicher, da die Prozesse auf eine Änderung im Speicher ihres Vorgängers warten. Folglich eignet sich der CLH-Lock nicht als globaler Lock im Cohort-Lock. Dasselbe gilt auch für den Hem-

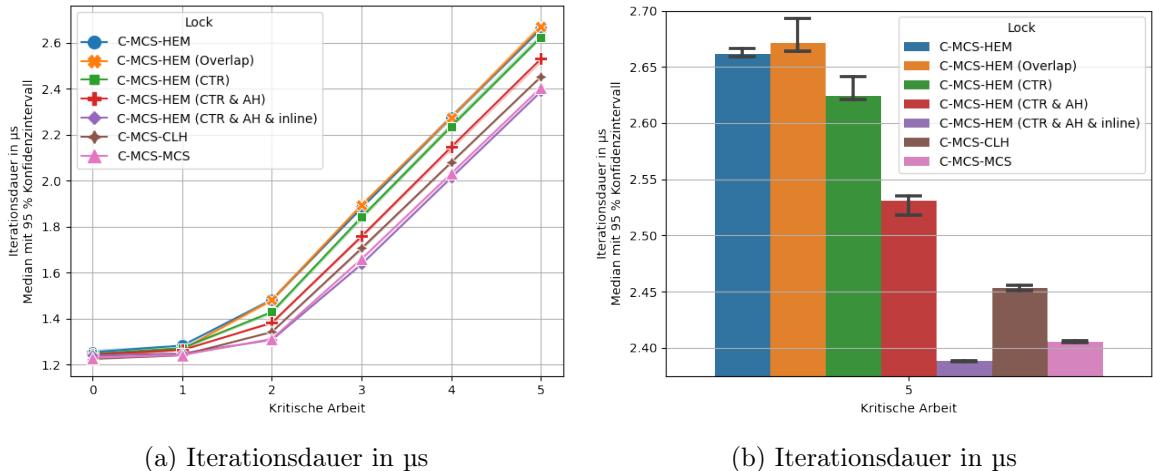
¹Obwohl die Geschwindigkeit des lokalen TKT-Locks durch Optimierung 3 (Inline Zähler) verbessert werden kann, ändert das nichts an den Ergebnissen der Evaluation: Ein lokaler MCS-Lock ist in allen Benchmarks trotzdem schneller (nicht gezeigt).

lock. Innerhalb eines Rechenknotens gibt es hingegen gemeinsamen Speicher und Zwischenspeicher, daher wird nun evaluiert, ob sich diese beiden Locks als lokale Locks eignen.

In [DK22] werden neben dem Basis-Algorithmus des Hemlocks drei Optimierungen vorgestellt: Overlap, *Coherence Traffic Reduction* (CTR) und *Aggressive Hand-over* (AH). Die Overlap-Optimierung wird von den Autoren zwar erläutert, aber nicht empfohlen, da sie keine große Verbesserung beobachten konnten. Der Pseudocode dieser Optimierung hatte in [DK22] einen Fehler, der zu Deadlocks führte und im Rahmen dieser Masterarbeit aufgefallen ist. Die Autoren von [DK22] wurden darüber informiert und haben daraufhin eine neue Version (v4) ihrer Arbeit veröffentlicht, in der der Fehler behoben ist.

Die CTR-Optimierung ist eine geschickte Verwendung von atomaren Operationen, die zu einer besseren Nutzung der Zwischenspeicher führt. Diese wurde speziell für Zwischenspeicher-Cohärenz-Mechanismen vom Typ MESI oder MESIF entwickelt. Dabei steht MESIF für *modified, exclusive, shared, invalid and forward* und bezeichnet die Status, die ein Speicherbereich im Zwischenspeicher haben kann. Zur Erinnerung: In dieser Arbeit werden die Benchmarks auf Intels Haswell Prozessoren ausgeführt. Diese nutzen einen auf MESIF basierenden Mechanismus [MHSN15]. Damit ist eine Betrachtung dieser Optimierung sinnvoll.

Durch die AH-Optimierung wird die Lockübergabe spekulativ ausgeführt, bevor geprüft wird, ob es überhaupt einen Nachfolger gibt. Dadurch wird der kritische Pfad bei Konkurrenz kürzer, was den Lock schneller macht. So eine spekulative Lockübergabe hat aber auch einen Nachteil: Abhängig von der Speicherverwaltung kann es durch *use-after-free*-Fehler zum Programmabsturz kommen [DK22]. Bei der Nutzung von MPI wird der gemeinsame Speicher über ein Fenster verwaltet. Da ein Fenster von allen beteiligten Prozessen freigegeben werden muss, bevor der Speicher selbst freigegeben wird, können solche *use-after-free*-Fehler hier nicht auftreten. In [DK22] werden noch zwei Varianten der AH-Optimierung vorgestellt, die solche Fehler vermeiden. Jedoch schreiben die Autoren, dass der Hemlock mit CTR und AH die beste Performance hat und daher zu bevorzugen ist, wenn die Speicherverwaltung es zulässt. Aus diesem Grund werden die beiden Varianten der AH-Optimierung in dieser Arbeit nicht untersucht.



Benchmark 5.14: CCWB von C-MCS-HEM mit 28 Prozessen

Benchmark 5.14a zeigt die Geschwindigkeit der Cohort-Locks mit den lokalen Hemlock-Varianten und einem lokalen CLH-Lock im Vergleich zum C-MCS-MCS-Lock. Für den Bench-

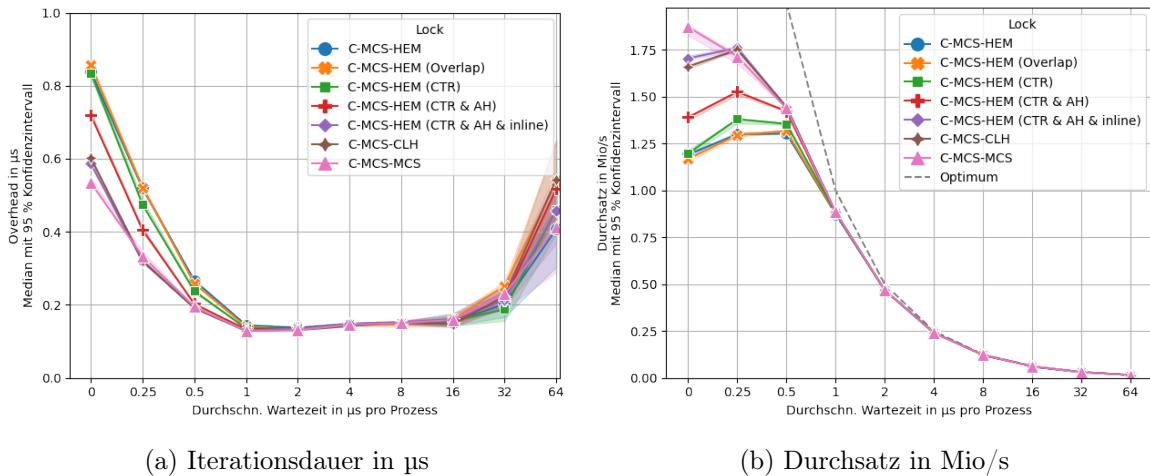
mark wurde nur ein Rechenknoten verwendet (daher 28 Prozesse), damit der Overhead des globalen Locks geringer ist und die Unterschiede zwischen den verschiedenen lokalen Locks stärker hervortreten. In Benchmark 5.14b ist eine Nachaufnahme der Geschwindigkeiten bei einer kritischen Arbeit von 5 zu sehen. In dieser sieht man die Unterschiede noch besser. Die Fairness wird nicht gezeigt. Da alle Locks FIFO-Warteschlangen nutzen, sind alle sehr fair.

In diesem Benchmark ist der lokale CLH-Lock langsamer als der lokale MCS-Lock. Die Overlap-Optimierung des Hemlocks führt, wie in [DK22] beschrieben, nicht zu einer Verbesserung. CTR, insbesondere in Kombination mit AH, hingegen schon. Trotzdem ist der C-MCS-HEM mit CTR und AH deutlich langsamer als die Cohort-Locks mit lokalem CLH- und MCS-Lock.

Der Grund dafür ist Optimierung 3 (Inline Zähler) des Cohort-Locks aus Abschnitt 5.4.1, die auf den lokalen CLH- und MCS-Lock, aber nicht auf den Hemlock angewandt wurde. Genau wie der TKT-Lock nutzt auch der Hemlock kein boolesches *Flag* für die Lockübergabe. Daher wurde in dieser Arbeit auch der lokale Hemlock um ein Feld erweitert, mit dem die Anzahl der lokalen Lockübergaben an den Nachfolger kommuniziert wird.

Um den Hemlock besser mit dem CLH- und MCS-Lock vergleichen zu können, wurde dieser angepasst, sodass ein Nachfolger nicht auf die eindeutige Speicheradresse des Locks, sondern, wie bei den beiden anderen Locks, auf einen booleschen Wert wartet. Durch diese Anpassung konnte auch auf den Hemlock Optimierung 3 angewandt werden, aber die Warteschlangenknoten können nicht mehr von mehreren Prozessen verwendet werden. Der Hemlock hat also keinen Vorteil mehr im Bezug auf Speicherverbrauch. Das könnte jedoch leicht behoben werden, indem der Zähler für lokale Lockübergaben und die eindeutige Speicheradresse des Locks (oder eine andere Zahl, die die Lock-Instanz eindeutig identifiziert) in einen gemeinsamen Speicherbereich kodiert werden (vgl. Diskussion über Optimierung 3 für TKT-Lock). Da der Speicherverbrauch in dieser Arbeit nicht untersucht wird, wurde auf so eine Kodierung verzichtet.

In Benchmark 5.14b sieht man deutlich die Verbesserung durch Optimierung 3. Der angepasste Hemlock überholt sogar den lokalen MCS-Lock, obwohl er wie der CLH-Lock NUMA nicht optimal nutzen kann. Mit diesen Optimierungen setzt sich der C-MCS-HEM knapp als schnellster Lock im CCWB durch.



Benchmark 5.15: WBAB von C-MCS-HEM mit 112 Prozessen

Um die Performance auf mehreren Rechenknoten zu evaluieren, zeigt Benchmark 5.15 die Ergebnisse des WBAB mit 112 Prozessen, also vier Rechenknoten. Diese Ergebnisse unterscheiden sich kaum von denen des WBABs mit nur einem Rechenknoten (nicht gezeigt). Der CCWB mit 112 Prozessen wird nicht gezeigt, da die Konfidenzintervalle durch die vielen entfernten Zugriffe des Benchmarks so ungenau sind, dass die Locks nicht aussagekräftig miteinander verglichen werden können.

Benchmark 5.15a und Benchmark 5.15b zeigen, dass auch im WBAB die Overlap-Optimierung keine Verbesserung bringt, während der Hemlock mit CTR etwas schneller und mit zusätzlich AH deutlich schneller wird. Wie bereits im CCWB mit 28 Prozessen (vgl. Benchmark 5.14) überholt der Hemlock aber erst mit Nutzung von Optimierung 3 (Inline Zähler) den CLH-Lock. Ohne Wartezeit bleibt in diesem Benchmark der C-MCS-MCS-Lock der schnellste Lock, bei einer Wartezeit von 0,25 μ s wird er aber knapp vom C-MCS-CLH-Lock und dem voll optimierten C-MCS-HEM-Lock überholt. Danach ist zwischen den drei Locks kein Unterschied mehr zu erkennen.

Zusammenfassend lässt sich sagen, dass ein Cohort-Lock mit lokalem Hemlock eine sehr gute Performance hat. In den meisten Fällen ist er genauso schnell wie ein Cohort-Lock mit lokalem MCS-Lock, in manchen Fällen etwas schneller und in manchen Fällen etwas langsamer. Dabei benötigt er aber deutlich weniger Speicherplatz, da mehrere Lock-Instanzen dieselben Warteschlangenketten verwenden können. Das ist ein klarer Vorteil.

5.5 AHMCS-Lock

Der adaptive HMCS-Lock (kurz AHMCS-Lock) aus [CMC16] basiert auf dem HMCS-Lock aus [CFMC15]. Wie bereits in Abschnitt 5.4 erklärt, ist ein HMCS-Lock eine optimierte Verallgemeinerung eines Cohort-Locks für beliebig viele Hierarchieebenen, wobei auf jeder Ebene ein MCS-Lock verwendet wird. Das hat zwar den Vorteil, dass bei hoher Konkurrenz die Lokalität auf jeder Ebene, z. B. durch L1-, L2- und L3-Zwischenspeicher, perfekt genutzt werden kann. Es führt jedoch bei geringer Konkurrenz zu mehr Overhead, da Prozesse dann mehrere Locks akquirieren müssen. Bei einem freien HMCS-Lock muss ein Prozess sogar die Locks aller Hierarchieebenen nacheinander akquirieren.

Der AHMCS-Lock vermeidet dieses Problem, indem er Prozessen erlaubt, direkt die Locks der höheren Ebenen zu akquirieren, ohne die Locks der unteren Ebenen zu besitzen. Die Einstiegsebene wird dabei abhängig von der Konkurrenz durch Hysterese bestimmt. D. h., jeder Prozess misst die Konkurrenz bei Akquirieren und Freigeben des Locks und entscheidet darauf basierend, auf welcher Ebene er beim nächsten Mal mit der Akquisition beginnen soll. Um die Konkurrenz zu messen, beobachtet der Prozess, ob er auf seiner Einstiegsebene einen Vorgänger und ob er einen Nachfolger hat. Wenn beides der Fall ist, wird eine hohe Konkurrenz angenommen und der Prozess steigt beim nächsten Mal eine Ebene niedriger ein. Wird weder ein Vorgänger, noch ein Nachfolger beobachtet, wird eine niedrige Konkurrenz angenommen und der Prozess steigt beim nächsten Mal eine Ebene höher ein. Ansonsten nutzt der Prozess auch beim nächsten Mal wieder dieselbe Einstiegsebene.

So passt sich der AHMCS-Lock dynamisch an die Konkurrenzsituation an. Wenn keine Konkurrenz vorliegt, nutzen Prozesse bei jedem Mal eine höhere Ebene, wodurch sich der HMCS-Lock nach ein paar Akquisitionen wie ein normaler MCS-Lock verhält, da alle Prozesse direkt den globalen MCS-Lock akquirieren. Liegt hingegen eine hohe Konkurrenz vor, wandern Prozesse in der Hierarchie immer weiter nach unten, wodurch der AHMCS-Lock

sich wie ein HMCS-Lock verhält und die Lokalität auf jeder Ebene nutzt.

Wenn nach einer gewissen Zeit von hoher Konkurrenz die Konkurrenz plötzlich stark abfällt, braucht die Anpassung der Einstiegsebenen durch Hysterese einige Akquisitionen. Um bei so einem Szenario schneller reagieren zu können, ist in den AHMCS-Lock eine Abkürzung eingebaut. Jeder Prozess prüft als Erstes, ob der Lock auf seiner Einstiegsebene frei ist und wenn ja, ob auch der globale MCS-Lock frei ist. Ist beides der Fall, akquiriert er den globalen Lock direkt und überspringt alle unteren Ebenen. Erst wenn wieder Konkurrenz vorliegt, wird die Einstiegsebene wieder durch Hysterese angepasst.

Wenn ein System *Hardware Transactional Memory* (HTM) unterstützt, kann diese Abkürzung noch effizienter implementiert werden. HTM erlaubt es, Transaktionen mit Hardwareunterstützung auszuführen. Dabei ist garantiert, dass der gesamte Code innerhalb einer Transaktion atomar ausgeführt wird. Wird ein Konflikt mit der Transaktion eines anderen Prozesses festgestellt, wird die Transaktion abgebrochen und alle Änderungen rückgängig gemacht. HTM wird in [CMC16] genutzt, um den kritischen Abschnitt spekulativ auszuführen, statt den globalen MCS-Lock zu akquirieren, wenn die Abkürzung verwendet wird. Bricht die Transaktion ab, fällt der Prozess auf den normalen Weg mit Hysterese zurück.

In dieser Arbeit werden, wie in Abschnitt 5.4 erläutert, nur Locks mit zwei Hierarchieebenen betrachtet. Eine Hysterese ist aber erst bei einer höheren Anzahl an Ebenen von Nutzen. Der UPB hat in Benchmark 4.1 bereits gezeigt, dass der Geschwindigkeitsunterschied zwischen dem D-MCS mit einer Ebene und dem RMA-MCS mit zwei Ebenen gering ist, wenn die Locks frei sind, und es viel wichtiger ist RMA-Zugriffe, besonders auf entfernten Speicher, zu vermeiden.

Auch die Abkürzung bringt bei nur zwei Hierarchieebenen keinen Vorteil, da immer der lokale und globale Lock überprüft werden, um Ebenen zu überspringen. Ein normaler Cohort-Lock hat aber keine weiteren Ebenen, neben dem lokalen und globalen Lock. Und auch HTM kann im Rahmen dieser Arbeit nicht genutzt werden, da das CoolMUC-2 Linux Cluster des LRZ HTM nicht unterstützt.

Aus diesen Gründen ist es nicht sinnvoll, den AHMCS-Lock auf verteilten Speicher zu portieren. Auf einem System mit HTM-Support, oder wenn der Lock z. B. nur mit Open-MPI funktionieren muss, sodass mehrere Hierarchieebenen unterstützt werden können, könnte sich dieser Lock im Rahmen einer anderen Arbeit aber als nützlich erweisen.

5.6 CST-Lock

Der CST-Lock [KMK17] basiert auf einem Cohort-Lock [DMS12] mit einem globalen MCS-Lock [MCS91b] und einem lokalen K42-Lock [AEK⁺02] (einer Variante des MCS-Locks) und hat im wesentlichen zwei große Verbesserungen:

1. Der benötigte Speicher wird möglichst spät dynamisch allokiert, sodass kein unnötiger Speicher für Prozesse verbraucht wird, die den Lock vielleicht gar nicht nutzen. Das ist in Systemen mit gemeinsamem Speicher sinnvoll, da vorher nicht klar ist, welche Prozesse den Lock nutzen.
2. Der Scheduler wird berücksichtigt, d. h., Prozesse warten nur so lange aktiv, bis die ihnen vom Scheduler zugeteilte Zeitscheibe abläuft und werden dann geparkt, bis sie von ihrem Vorgänger aufgeweckt werden. Das sorgt für eine bessere Performance bei überladenen Systemen, die mehr Prozesse ausführen, als Prozessoren zur Verfügung stehen, da wartende Prozesse nach Ablauf ihrer Zeitscheibe keinen Prozessor mehr blockieren.

Diese beiden Verbesserungen sind aber bei verteilten Systemen im Bereich HPC nicht so relevant, da erstens alle Prozesse, die den Lock nutzen möchten, ein gemeinsames Fenster erzeugen müssen, d. h. wenn bekannt ist, dass manche Prozesse den Lock nicht benötigen, können diese ausgeschlossen werden und zweitens im HPC-Bereich Systeme typischerweise nicht überladen sind, da für optimale Performance in der Regel genau so viele Prozessoren verwendet, wie Prozesse ausgeführt werden. Der CST-Lock ist daher kein guter Kandidat für eine Portierung auf verteilten Speicher.

5.7 SHFL-Lock

Der SHFL-Lock aus [KCC⁺19] kombiniert einen TTS-Lock (siehe Unterabschnitt 2.1.1) mit einem MCS-Lock (siehe Unterabschnitt 2.1.3): Ein Prozess versucht erst einmalig, den TTS-Lock zu akquirieren. Wenn er das nicht schafft, akquiriert er den MCS- und dann den TTS-Lock. Durch den MCS-Lock ist sichergestellt, dass bei hoher Konkurrenz nicht alle Prozesse ständig über den TTS-Lock auf denselben Speicherbereich zugreifen und durch den TTS-Lock können Prozesse bei geringer Konkurrenz den Overhead des MCS-Locks vermeiden.

Das Besondere beim SHFL-Lock ist, dass die Reihenfolge der Warteschlangenknoten des MCS-Locks während des Wartens geändert wird. Diese werden gruppiert, sodass Prozesse desselben NUMA-Knotens in der Schlange beieinanderliegen und aufeinanderfolgend den kritischen Abschnitt betreten. Die Sortierung der Warteschlange wird dabei zunächst von dem Prozess durchgeführt, der bereits den MCS-Lock akquiriert hat, aber noch auf den TTS-Lock wartet. Dieser Prozess iteriert durch die Schlange und bewegt die Knoten aller Prozesse, die auch auf seinem NUMA-Knoten laufen, an den Anfang der Schlange. Den letzten Prozess, dessen Knoten er bewegt hat, ernennt er dann zum neuen Sortierer.

Durch diese Strategie wird die Sortierung der Warteschlange von den Prozessen ausgeführt, die sonst sowieso nur warten würden. Da immer nur ein Prozess gleichzeitig sortiert, muss beim Sortierungsalgorithmus nicht auf *race conditions* geachtet werden. Lediglich der letzte Knoten der Warteschlange darf nicht bewegt werden, da sich hier nebenläufig weitere Prozesse einreihen können.

```

1  if (tts_lock.disabled_and_locked.load() == {0, 0})
2    if (tts_lock.disabled_and_locked.cas({0, 0}, {0, 1}))
3      return; // Lock geklaut
4
5  bool first = mcs_lock.acquire();
6  if (first) // Deaktiviere Klauen
7    tts_lock.disabled.store(1);
8
9  do {
10    if (mcs_node.batch == 0 || mcs_node.is_shuffler)
11      shuffle_waiters();
12    while (tts_lock.locked.load() == 1);
13  } while (!tts_lock.locked.cas(0, 1));
14
15 bool last = mcs_lock.release();
16 if (last) // Aktiviere Klauen
17   tts_lock.disabled.store(0);

```

Abbildung 5.5: Akquirieren eines SHFL-Locks (`acquire`)

Abbildung 5.5 zeigt eine vereinfachte C++-Implementierung der Akquisition. Wenn ein Prozess direkt in Zeile 1-3 den TTS-Lock akquiriert, ohne den MCS-Lock zu nutzen, spricht man davon, dass der Prozess den Lock klaut. Um zu verhindern, dass ein neuer Prozess alle wartenden Prozesse überholt, indem er den Lock klaut, obwohl bereits ein Prozess den MCS-Lock akquiriert hat und in Zeile 9-13 auf den TTS-Lock wartet, ist der TTS-Lock so modifiziert, dass das Klauen deaktiviert werden kann. Dafür nutzt er zwei Bytes: Das `locked`-Byte enthält die Information, ob er bereits akquiriert ist und das `disabled`-Byte enthält die Information, ob er deaktiviert ist. Um den SHFL-Lock zu klauen, darf der TTS-Lock weder deaktiviert, noch akquiriert sein. Dies wird sichergestellt, indem `disabled` und `locked` gemeinsam mit einer CAS-Operation atomar von `{0, 0}` auf `{0, 1}` geändert werden (siehe Abbildung 5.5, Zeile 2). Wenn nicht beide Bytes 0 sind, schlägt die CAS-Operation fehl und der Prozess fährt fort, indem er den MCS-Lock akquiriert.

Der MCS-Lock, den der SHFL-Lock nutzt, ist ebenfalls modifiziert: Die `acquire`-Funktion liefert `true` zurück, wenn der Prozess nicht auf einen Vorgänger warten musste, sondern den MCS-Lock direkt akquirieren konnte. In diesem Fall deaktiviert er den TTS-Lock in Zeile 7, sodass der Lock nicht mehr geklaut werden kann, bis die Warteschlange wieder leer ist.

Nachdem der Prozess den MCS-Lock akquiriert hat, prüft er in Zeile 10, ob er für die Sortierung verantwortlich ist und führt diese ggf. durch. Anschließend wartet er in Zeile 12 auf den TTS-Lock und versucht ihn in Zeile 13 mit einer CAS-Operation zu akquirieren. Wenn das fehlschlägt, sortiert er die Warteschlange ggf. erneut. Im Gegensatz zu Zeile 1 und 2 wird in Zeile 12 und 13 das `disabled`-Byte ignoriert. Der TTS-Lock kann hier also auch dann akquiriert werden, wenn das Klauen deaktiviert ist.

Für die Sortierung iteriert der Prozess über die Warteschlange und verschiebt jeden Knoten, dessen Prozess auf demselben NUMA-Knoten läuft, hinter den Knoten, den er zuletzt verschoben hat. Den ersten gefundenen Knoten verschiebt er hinter seinen eigenen Knoten. Dabei benutzt er das Feld `batch`, um alle verschobenen Knoten (inklusive seines eigenen Knotens) zu nummerieren. Die Nummerierung beginnt mit der Zahl 1. Nachdem der Sortierer einmal über die Warteschlange iteriert ist, sind alle Knoten desselben NUMA-Knotens an den Anfang der Warteschlange verschoben worden und haben aufsteigende Werte in ihrem `batch`-Zähler. Der Sortierer setzt dann das *Flag* `is_shuffler` des letzten verschobenen Knotens auf `true` und beendet seine Sortierung.

Für die Prüfung, ob der Prozess die Warteschlangen sortieren soll, gibt es daher zwei Kriterien: Wenn in Zeile 10 sein `batch`-Zähler noch den Initialwert 0 hat, ist er der erste Sortierer seines NUMA-Knotens. Ist hingegen sein `is_shuffler-Flag` `true`, so wurde er von einem Vorgänger zum Sortierer ernannt. In beiden Fällen soll er die Warteschlange sortieren.

Um zu verhindern, dass immer die Prozesse eines NUMA-Knotens an den Anfang der Warteschlange verschoben werden und andere Prozesse verhungern, wird der `batch`-Zähler zu Beginn jeder Sortierung überprüft. Wenn er einen Schwellwert überschreitet, wird die Sortierung abgebrochen, sodass nach einer Weile der nächste Knoten mit einem `batch`-Wert von 0 den MCS-Lock akquiriert und zum neuen Sortierer wird.

Da ein ganzer Sortierungslauf zu lange dauern könnte, prüft der Sortierer außerdem am Ende jeder Iteration, ob der TTS-Lock inzwischen frei ist, bevor er mit dem nächsten Knoten fortfährt. Ist das der Fall, ernennt er ebenfalls den Prozess des letzten verschobenen Knotens zum neuen Sortierer und beendet seine Sortierung vorzeitig.

Durch das `is_shuffler-Flag` kann es passieren, dass ein Prozess zum Sortierer ernannt wird, der gerade auf den MCS-Lock wartet. Daher ist der MCS-Lock zusätzlich so modifiziert, dass wartende Prozesse immer wieder ihr `is_shuffler-Flag` prüfen und ggf. auch die Funktion `shuffle_waiters` aufrufen. In diesem Fall prüft der Sortierer am Ende jeder Iteration allerdings, ob der MCS-Lock inzwischen frei ist, nicht der TTS-Lock.

Nachdem ein Prozess ggf. die Warteschlange sortiert und den TTS-Lock akquiriert hat, gibt er den MCS-Lock in Zeile 15 direkt wieder frei. Wenn er dabei der letzte Prozess in der Warteschlange war, aktiviert er in Zeile 17 wieder die Möglichkeit den SHFL-Lock zu klauen.

Der kritische Abschnitt wird eigentlich nur durch den TTS-Lock geschützt, der MCS-Lock dient lediglich dazu, eine hohe Konkurrenz auf dem TTS-Lock zu vermeiden und eine Warteschlange zu haben, die sortiert werden kann, um lokale Nachfolger zu bevorzugen. Um den SHFL-Lock freizugeben, muss daher nach dem Ausführen des kritischen Abschnitts nur noch der TTS-Lock freigegeben werden (siehe Abbildung 5.6).

```
1 tts_lock.locked.store(0);
```

Abbildung 5.6: Freigeben eines SHFL-Locks (`release`)

5.7.1 Portierung und Optimierung des SHFL-Locks

Bei der Portierung auf verteilten Speicher wurde eine kleine Optimierung in der nicht gezeigten Funktion `shuffle_waiters` durchgeführt. In [KCC⁺19] wird in jeder Iteration sowohl für den betrachteten Knoten, als auch für dessen Vorgänger geladen, zu welchem NUMA-Knoten er gehört. Das ist kein Problem auf Systemen mit gemeinsamem Speicher, da der Wert beim ersten Zugriff in den Zwischenspeicher geladen wird. Auf verteiltem Speicher führt solcher Code hingegen zu unnötig vielen entfernten Zugriffen. Die portierte Version speichert daher den Wert des aktuellen Knotens in einer lokalen Variable und verwendet diese in der nächsten Iteration wieder, statt den Wert neu zu laden.

Ein großes Problem des SHFL-Locks auf verteiltem Speicher ist die Verwendung eines TTS-Locks. Während TTS-Locks auf gemeinsamem Speicher schneller sind als TAS-Locks, ist das auf verteiltem Speicher genau umgekehrt, da ein TTS-Locks deutlich mehr entfernte Zugriffe benötigt. Daher wird auch eine Variante des SHFL-Locks evaluiert, in der der TTS-Lock durch einen TAS-Lock ersetzt ist. Dafür müssen in Abbildung 5.5 lediglich Zeile 1 und 12 entfernt werden.

5.7.2 Evaluation des SHFL-Locks

Für die Evaluation werden der portierte SHFL-Lock und die SHFL-Lock-Variante mit TAS-Lock mit dem optimierten MCS-Lock basierend auf D-MCS aus Unterabschnitt 4.3.1 verglichen. Um den Effekt der Sortierung zu untersuchen, wird zusätzlich noch eine Kombination aus TTS- und MCS-Lock evaluiert, die dem SHFL-Lock entspricht, aber die Reihenfolge der Warteschlange unverändert lässt (Abbildung 5.5 ohne Zeile 10 und 11), sowie eine entsprechende Kombination aus TAS- und MCS-Lock (Abbildung 5.5 ohne Zeile 1 und 10-12).

Benchmark 5.16a zeigt die Geschwindigkeit im UPB: Auf dem Hauptknoten sind alle Locks ziemlich schnell, da durch die Möglichkeit, den Lock zu klauen, nur ein freier TTS- oder TAS-Lock akquiriert werden muss. Man sieht allerdings, dass die Varianten mit TTS-Lock etwas langsamer sind. Wenn der akquirierende Prozess auf einem anderen Knoten läuft, gibt es denselben Unterschied. Dieser ist aber um ein Vielfaches ausgeprägter, da es sich nun um entfernte Speicherzugriffe handelt.

Bei geringer Konkurrenz ist ein TAS-Lock schneller als ein TTS-Lock, da letzterer zusätzliche unnötige lesende Zugriffe ausführt. Im SHFL-Lock wird die Konkurrenz um den TTS-Lock durch den MCS-Lock gering gehalten. Daher liegt es nahe, das auch auf gemeinsamem Speicher (hier sind TTS-Locks bei hoher Konkurrenz schneller), ein SHFL-Lock mit TAS-Lock in jeder Konkurrenzsituation schneller ist als ein SHFL-Lock mit TTS-Lock. Dies wird durch Benchmark 5.16c und Benchmark 5.16d bestätigt.

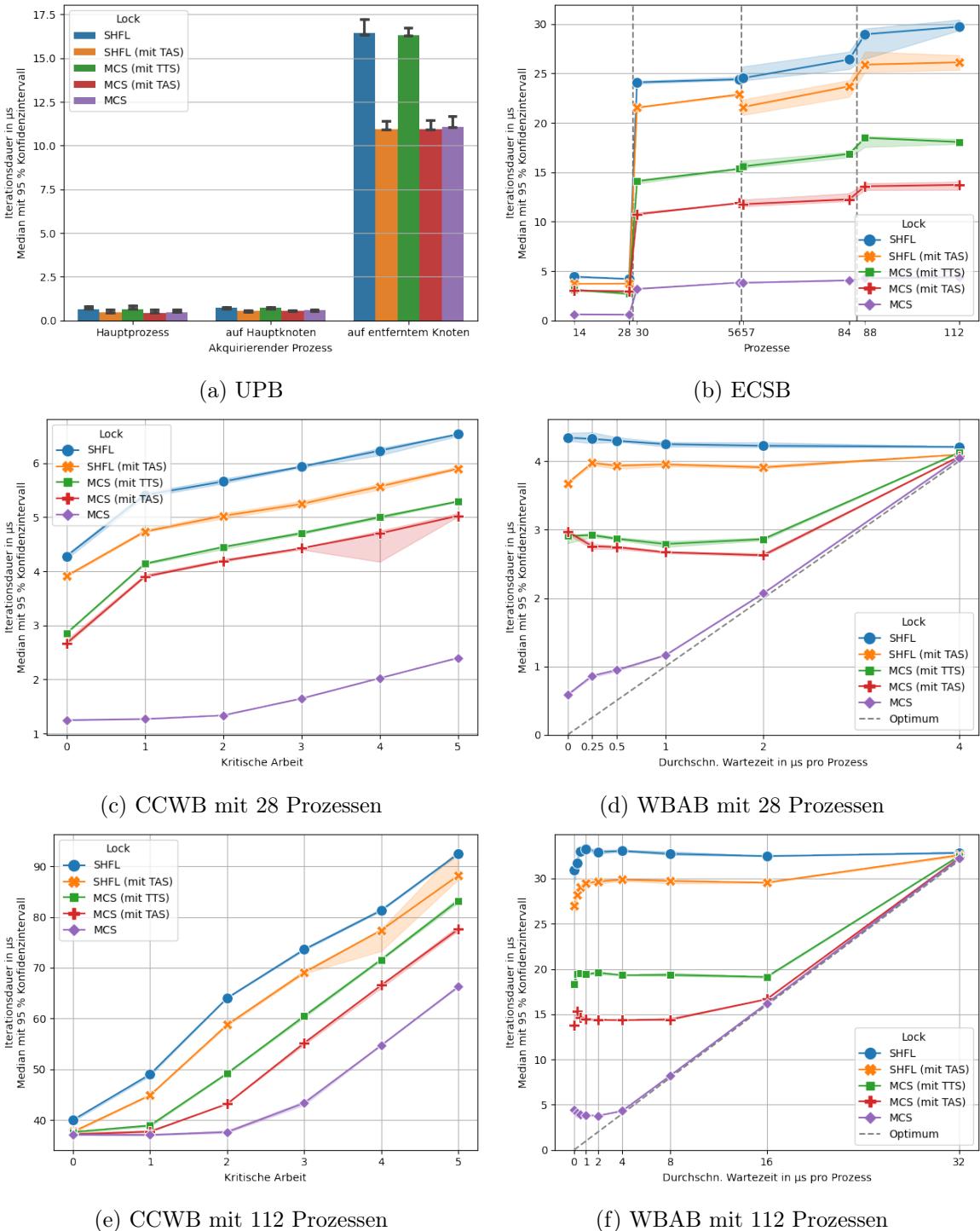
Im Gegensatz zum UPB zeigt der ECSB in Benchmark 5.16b Unterschiede zwischen allen Locks: Der SHFL-Lock ist hier extrem langsam mit einer Iterationsdauer von bis zu 30 µs, gefolgt von der Variante mit TAS-Lock mit bis zu 25 µs Iterationsdauer. Auch die Kombination von TTS- bzw. TAS- und MCS-Lock ist sehr langsam mit einer Iterationsdauer von bis zu 18 µs bzw. 13 µs. Der reine MCS-Lock hat zum Vergleich eine Iterationsdauer von unter 5 µs und ist damit in diesem Benchmark mehr als sechs Mal so schnell wie der SHFL-Lock. Der CCWB in Benchmark 5.16e und der WBAB in Benchmark 5.16f zeigen ein ähnliches Bild.

Die beiden Besonderheiten des SHFL-Locks, zum einen die Kombination von TTS- und MCS-Lock für die Möglichkeit, den Lock zu klauen und zum anderen die Sortierung der Warteschlange, führen also beide auf verteiltem Speicher zu einer Verschlechterung der Geschwindigkeit, selbst wenn statt einem TTS- ein TAS-Lock verwendet wird: In allen Benchmarks, außer dem UPB, ist die Kombination von TAS- und MCS-Lock deutlich langsamer als ein reiner MCS-Lock. Der SHFL-Lock, welcher zusätzlich die Warteschlange sortiert, ist noch einmal deutlich langsamer.

Während auf gemeinsamem Speicher *Spin*-Locks, wie der TTS- und TAS-Lock, sehr schnell sind – bei geringer Konkurrenz auch schneller als der MCS-Lock – ist das auf verteiltem Speicher nicht der Fall. Durch die Abwesenheit eines Zwischenspeichers sind hier viele langsame Zugriffe auf entfernten Speicher notwendig, wodurch ein wartender Prozess mit deutlich geringerer Frequenz den Status des Locks abfragen kann. Da im SHFL-Lock ein Prozess immer als letztes einen *Spin*-Lock akquirieren muss, bevor er den kritischen Abschnitt betreten darf, hängt die Performance des Shfl-Locks stark von diesem *Spin*-Lock ab. Das ist auf gemeinsamem Speicher gut und auf verteiltem Speicher schlecht.

Möglicherweise wäre es besser, zwei MCS-Locks zu kombinieren oder eine Variante zu finden, bei der ein Prozess im MCS-Lock effizient feststellen kann, dass er am Anfang der Warteschlange ist, sodass er die Sortierung starten kann. Beides erfordert allerdings eine umfangreiche Anpassung des SHFL-Locks und ist daher nicht mehr Teil dieser Arbeit.

5 Portierung von NUMA-Locks auf verteilten Speicher



Benchmark 5.16: Iterationsdauer in µs des Shfl-Locks in allen Benchmarks

Selbst wenn man eine Variante findet, zwei Locks effizient zu kombinieren, bleibt das Problem, dass eine Sortierung der Warteschlange die Geschwindigkeit verschlechtert, statt sie zu verbessern. Obwohl die Sortierung von Prozessen ausgeführt wird, die eigentlich nur warten

sollten, wirkt sie sich negativ auf die Geschwindigkeit aus. Durch die vielen entfernten Zugriffe, die beim Iterieren über die Warteschlange notwendig sind, ist eine Iteration auf verteiltem Speicher deutlich langsamer als auf gemeinsamem Speicher. Da auf jedem NUMA-Knoten zunächst der Prozess am Anfang der Warteschlange die Sortierung übernimmt, prüft dieser Prozess mit einer noch geringeren Frequenz den Status des Locks, wodurch er noch später den kritischen Abschnitt betritt. Erst wenn dieser Prozess über die komplette Warteschlange iteriert ist oder festgestellt hat, dass er den TTS- oder MCS-Lock akquiriert hat, bestimmt er einen neuen Sortierer. Dabei könnte er auch direkt den ersten Prozess, dessen Knoten er bewegt, zum neuen Sortierer ernennen, um dafür zu sorgen, dass Prozesse am Anfang der Warteschlange schnell bereit sind, den kritischen Abschnitt zu betreten. Da wartende Prozesse im MCS-Lock aktiv in einer Schleife warten und nicht vom Scheduler des Betriebssystems geparkt werden, sollte so eine frühere Ernennung des Sortieres keinen negativen Effekt haben.

Doch auch wenn man einen Weg findet, die Warteschlange besser zu sortieren, ist nicht zu erwarten, dass ein SHFL-Lock mit TAS- und MCS-Lock schneller wird als ein normaler MCS-Lock. Auf gemeinsamem Speicher gibt es zwei Faktoren, durch die eine Lockübergabe innerhalb eines NUMA-Knotens schneller ist:

1. Wenn ein Prozess für die Übergabe des Locks auf den Speicher seines Nachfolgers zugreifen muss, wie es z. B. beim MCS-Lock der Fall ist, ist dieser Zugriff innerhalb eines NUMA-Knotens schneller, da der Zugriff lokal ist. Somit erhält der Nachfolger den Lock schneller.
2. Wenn ein Lock innerhalb eines NUMA-Knotens weitergegeben wird, sind die Speicherbereiche, auf die der Vorgänger innerhalb des kritischen Abschnitts zugegriffen hat noch im Zwischenspeicher des NUMA-Knotens. Da alle Prozesse im kritischen Abschnitt typischerweise auf dieselben Speicherbereiche zugreifen, kann der Nachfolger von dem Zwischenspeicher profitieren.

Das gilt nicht nur für die Variablen auf die im kritischen Abschnitt zugegriffen wird, sondern auch für den Lock selbst. So ist mit Zwischenspeicher eine Lockübergabe innerhalb eines NUMA-Knotens auch dann schneller, wenn der Vorgänger für die Freigabe einen entfernen Zugriff tätigt. Auf gemeinsamem Speicher werden auch entfernte Zugriffe zwischengespeichert, sodass der Nachfolger z. B. das *Flag* eines TTS-Locks bei der Akquisition im Zwischenspeicher seines NUMA-Knoten vorfindet.

Der erste Faktor ist zwar auch auf verteiltem Speicher relevant, betrifft aber weder den TTS- noch den TAS-Lock, also genau die Locks, die ein Prozess beim SHFL-Lock akquirieren muss, unmittelbar bevor er den kritischen Abschnitt betreten darf. Der SHFL-Lock profitiert daher nicht von diesem Faktor. Da auf verteiltem Speicher entfernte Speicherzugriffe nicht zwischengespeichert werden, entfällt auch der zweite Faktor, sofern im kritischen Abschnitt hauptsächlich entfernte Zugriffe getätigt werden. Für den SHFL-Lock macht es auf verteiltem Speicher daher gar keinen Performanceunterschied, ob die Warteschlange sortiert wurde oder nicht.

6 Fazit

Diese Arbeit hat untersucht, ob sich Lock-Algorithmen, die für Systeme mit gemeinsamem Speicher mit *Non Uniform Memory Access* (NUMA) entwickelt wurden, auch für Systeme mit verteiltem Speicher eignen. Für die Implementierung der Algorithmen wurde MPI verwendet. So konnten die Zugriffe auf entfernten Speicher mit RMA realisiert werden.

Um einen Vergleich der Locks zu ermöglichen, wurde eine neue Benchmarksuite entwickelt, mit der die Geschwindigkeit, die praktische Fairness und andere algorithmusspezifische Kennzahlen von beliebigen Lock-Implementierungen auf verteiltem Speicher in verschiedenen Szenarien gemessen werden kann. Diese Suite beinhaltet vier Benchmarks, die einen Lock allen möglichen Konkurrenzsituationen aussetzen. Die ersten drei davon nutzen einen leeren kritischen Abschnitt, um den reinen Overhead der Locks zu messen:

1. Der UPB misst die Geschwindigkeit von freien Locks (0 % Konkurrenz).
2. Der ECSB misst die Geschwindigkeit von vollständig ausgelasteten Locks (100 % Konkurrenz).
3. Der WBAB variiert die Konkurrenz von maximal zu minimal durch eine Veränderung der Arbeit außerhalb des kritischen Abschnitts.
4. Der CCWB enthält Arbeit im kritischen Abschnitt, um ein möglichst realistisches Szenario zu erreichen. Er variiert die Konkurrenz von minimal zu maximal durch eine Veränderung dieser kritischen Arbeit.

Alle Benchmarks wurden so entworfen, dass sie mit einer beliebigen Anzahl von Prozessen funktionieren, sodass sie sich auch für Hochleistungsrechner mit einer großen Anzahl von Prozessoren und damit für den HPC-Bereich eignen.

Die Evaluation der aktuell auf verteiltem Speicher verfügbaren Lock-Implementierungen zeigte, dass bei der Verwendung von MPI nach Möglichkeit RMA-Zugriffe auf lokalen Speicher vermieden werden sollten, da diese deutlich langsamer sind als direkte Speicherzugriffe. Mit diesem Wissen und einigen Optimierungen konnte die Geschwindigkeit von D-MCS aus [SBH16] und von `dash::Mutex` aus [ZMI⁺14] verbessert werden: Der optimierte `dash::Mutex` ist in den Benchmarks bis zu vier Mal so schnell wie die unoptimierte Variante. Trotz der Optimierungen schneidet der RMA-MCS-Lock aus [SBH16] von allen verfügbaren Locks auf verteiltem Speicher am besten ab.

Für die Portierung von NUMA-Locks auf verteilten Speicher wurden sieben Algorithmen näher untersucht (siehe Tabelle 1.1). Die Analyse ergab, dass sich drei der Algorithmen (RH-Lock, Cohort-Lock und SHFL-Lock) für eine Portierung auf verteilten Speicher eignen. Diese drei wurden mit MPI implementiert und für verteilten Speicher optimiert. Die Implementierungen und die einzelnen Optimierungen wurden ebenfalls mit der Benchmarksuite evaluiert. Hierbei zeigte sich, dass auf verteiltem Speicher entfernte Zugriffe noch dringender vermieden werden müssen als auf gemeinsamem Speicher mit NUMA. Auf dem kritischen Pfad und in Warteschleifen wirken sich entfernte Zugriffe besonders negativ auf die Geschwindigkeit aus.

6 Fazit

Der portierte SHFL-Lock hatte eine sehr schlechte Performance. Bei Locks tragen auf gemeinsamem Speicher im Allgemeinen zwei Faktoren dazu bei, dass die Bevorzugung lokaler Prozesse bei der Lockübergabe vorteilhaft ist: Die Übergabe eines Locks an einen lokalen Nachfolger kann ohne Zugriffe auf entfernten Speicher realisiert werden und lokale Nachfolger können meist davon profitieren, dass die Daten, auf die sie im kritischen Abschnitt zugreifen, durch ihren Vorgänger bereits in einen Zwischenspeicher geladen wurden. Der SHFL-Lock benötigt allerdings auch für eine lokale Lockübergabe entfernte Zugriffe und in MPI gibt es keinen Zwischenspeicher für entfernte Zugriffe, der lokalen Nachfolgern einen Vorteil verschaffen würde. Diese veränderten Gegebenheiten wirken sich auch auf andere Algorithmen negativ aus, z. B. auf den HCLH-Lock und vermutlich auch auf den CNA-Lock aus [DK19] und Fissile-Locks aus [DK21].

Die Evaluation des RH-Locks zeigte, dass dieser bei geringer Konkurrenz die Geschwindigkeit von RMA-MCS übertreffen kann. Dafür ist dieser Lock bei hoher Konkurrenz extrem unfair, wodurch Prozesse verhungern können.

Am besten schneidet von den portierten Locks der Cohort-Lock ab. Dies ist auch der Algorithmus, der in RMA-MCS implementiert ist, wobei allerdings noch eine entscheidende Optimierung aus dem HMCS-Lock einfließt. Durch die zusätzliche Vermeidung von RMA-Zugriffen ließ sich RMA-MCS stark optimieren. Der daraus resultierende Cohort-Lock mit globalem und lokalem MCS-Lock ist bei hoher Konkurrenz etwa vier Mal so schnell und bei mittlerer Konkurrenz fast doppelt so schnell wie RMA-MCS.

Eine Untersuchung von elf Varianten des Cohort-Locks mit drei globalen und fünf lokalen Locks ergab schließlich, dass durch die Nutzung eines lokalen Hemlocks anstelle eines MCS-Locks eine vergleichbare Geschwindigkeit bei einem deutlich geringeren Speicherverbrauch erreicht werden kann.

Die Forschungsfrage kann also positiv beantwortet werden: Es wurde eine schnellere Implementierung mit einem geringeren Speicherverbrauch als der bisher beste Lock für verteilten Speicher (RMA-MCS) gefunden, ohne Verluste bei der Fairness in Kauf nehmen zu müssen.

Dieses Ergebnis zeigt, dass die Portierung von NUMA-Locks auf verteilten Speicher großes Potenzial birgt und es gibt noch weitere Algorithmen, die untersucht werden können. Beispiele hierfür sind der FC-MCS-Lock aus [DMS11], der CNA-Lock aus [DK19], Fissile-Locks aus [DK21] und das sehr aktuelle Framework CLoF aus [LCPB⁺21]. Eine Evaluation dieser Algorithmen kann mit der neuen Benchmarksuite aus Kapitel 4 durchgeführt werden.

Das Ergebnis zeigt auch, dass Bibliotheken, wie DASH [ZMI⁺14], auf neuere Lock-Algorithmen umsteigen sollten. Dann könnten neue und bestehende Anwendungen im HPC-Bereich leicht von den Erkenntnissen der Forschung profitieren.

Weiterer Forschungsbedarf besteht außerdem bei Lock-Varianten, wie RW-Locks oder abbrechbaren Locks. Auch solche Locks sind kaum für verteilten Speicher verfügbar und hier ist ebenfalls eine Portierung von Algorithmen für gemeinsamen Speicher auf verteilten Speicher denkbar. Ein besserer RW-Lock könnte für die Implementierung von `MPI_Win_lock` und `MPI_Win_lock_all` genutzt werden. Davon könnten alle Anwendungen profitieren, die RMA mit *passive target synchronization* nutzen. Neue RW-Locks sollten daher mit den Fenster-Locks mehrerer MPI-Implementierungen, wie Intel-MPI und Open-MPI, verglichen werden.

Eine Arbeit in diesem Forschungsgebiet könnte zudem untersuchen, ob sich die Optimierungen, die in dieser Arbeit gefunden wurden, auch auf RMA-RW aus [SBH16] anwenden lassen. Das sind insbesondere die nicht-atomare Lockübergabe (vgl. Abschnitt 3.4), direkte Zugriffe auf globale MCS-Warteschlangenknoten (vgl. Abschnitt 5.4.1) und die Verwendung eines lokalen Hemlocks (vgl. Unterabschnitt 5.4.4).

Abbildungsverzeichnis

2.1	Inkrementieren eines Zählers	3
2.2	Inkrementieren eines Zählers mit einem Lock	4
2.3	Einige Spin-Lock-Algorithmen	5
2.4	MCS-Lock	7
2.5	Uniform Memory Access	10
2.6	Non-Uniform Memory Access	10
3.1	Lockübergabe, die in Intel-MPI zu einem Deadlock führt	13
3.2	Lockübergabe, die in Open-MPI zu einem Deadlock führt	14
3.3	Lockübergabe mit <code>MPI_Iprobe</code>	15
3.4	Nutzung eines Structs für MPI-Fenster	16
3.5	MCS-Lock in MPI (optimierter D-MCS)	17
3.6	Lockübergabe mit Punkt-zu-Punkt-Kommunikation	17
4.1	Anfang vom Quelltext des UPBs (Vollständiger Quelltext)	22
4.2	Quelltext einer CCWB Iteration (Vollständiger Quelltext)	25
4.3	Implementierung von <code>dash::Mutex</code>	37
5.1	CLH-Lock	49
5.2	CLH-Lock für NUMA	50
5.3	Cohort-Lock mit Inline-Zähler	56
5.4	TKT-Lock	57
5.5	Akquirieren eines SHFL-Locks (<code>acquire</code>)	73
5.6	Freigeben eines SHFL-Locks (<code>release</code>)	74

Benchmarkverzeichnis

4.1	UPB der Basislocks	23
4.2	ECSB der Basislocks (Vollständiger Quelltext)	24
4.3	Fairness der Basislocks im ECSB: CV des Fortschritts in %	25
4.4	CCWB der Basislocks mit 112 Prozessen	27
4.5	CCWB der Basislocks mit 28 Prozessen	28
4.6	Fairness im WBAB ohne MPI_Iprobe: CV des Fortschritts in %	31
4.7	WBAB der Basislocks mit 112 Prozessen	32
4.8	WBAB der Basislocks mit 28 Prozessen	33
4.9	Iterationsdauer der D-MCS-Optimierungen in μ s	35
4.10	CCWB der D-MCS-Optimierungen mit 28 Prozessen	36
4.11	Iterationsdauer der DASH-Optimierungen in μ s	38
4.12	CCWB der Basislocks-Optimierungen: Iterationsdauer in μ s	39
5.1	UPB des RH-Locks	44
5.2	ECSB des RH-Locks	45
5.3	CCWB des RH-Locks mit 112 Prozessen	46
5.4	WBAB des RH-Locks mit 112 Prozessen	47
5.5	Iterationsdauer der CLH-Locks in μ s	51
5.6	CCWB der CLH-Locks mit 112 Prozessen	52
5.7	WBAB des CLH-Locks mit 112 Prozessen	53
5.8	UPB der Cohort-Optimierungen	59
5.9	ECSB der Cohort-Optimierungen	60
5.10	WBAB der Cohort-Optimierungen mit 112 Prozessen	61
5.11	UPB & ECSB verschiedener Cohort-Locks	62
5.12	CCWB verschiedener Cohort-Locks mit 112 Prozessen	64
5.13	WBAB verschiedener Cohort-Locks mit 112 Prozessen	65
5.14	CCWB von C-MCS-HEM mit 28 Prozessen	68
5.15	WBAB von C-MCS-HEM mit 112 Prozessen	69
5.16	Iterationsdauer in μ s des Shfl-Locks in allen Benchmarks	76

Glossar

Message Passing ist ein Konzept für die Programmierung von verteilten Systemen, bei dem die beteiligten Computer Nachrichten austauschen, um miteinander zu Kommunizieren. 9, 42

Fenster (engl. *window*) ist in MPI der Begriff für eine Sicht auf die Speicherbereiche mehrerer Prozesse, die diese veröffentlicht haben, um RMA-Zugriffe zu ermöglichen. Mit mehreren Fenstern kann ein Prozess mehrere Speicherbereiche für unterschiedliche Zwecke veröffentlichen. Die veröffentlichten Speicherbereiche mehrerer Fenster können sich auch überschneiden. Um bei einem RMA-Zugriff die Ziel-Speicheradresse eindeutig anzugeben, muss ein Prozess das Fenster, den eindeutigen Rang des Zielprozesses, und den Versatz der Ziel-Adresse innerhalb des im Fenster veröffentlichten Speichers angeben. 9, 11–16, 18, 37, 41, 43, 58, 68, 72, 80

hungerfrei (engl. *starvation-free*) bezeichnet Algorithmen, in denen es unmöglich ist, dass ein Prozess verhungert (siehe Verhungern). 6

Kohärenz ist eine Eigenschaft von Zwischenspeichern. Die Zwischenspeicher und der Hauptspeicher sind kohärent, wenn alle Kopien einer Speicheradresse denselben Wert enthalten. Damit wird verhindert, dass verschiedene Prozesse bei Zugriffen auf dieselbe Speicheradresse verschiedene Werte lesen. Eine Inkonsistenz der Zwischenspeicher muss demnach spätestens bei einem lesenden Zugriff behoben werden. 4–6, 13, 30, 67, 83

Konkurrenz (engl. *contention*) bezeichnet, wie viele Prozesse gleichzeitig versuchen, einen Lock zu akquirieren. Um sie in dieser Arbeit messen zu können, wird sie in Unterabschnitt 4.1.2 als „der Anteil der Akquisitionen, bei denen ein Prozess einen Lock nicht direkt akquirieren konnte, sondern auf einen Vorgänger warten musste“ definiert. Diese empirische Konkurrenz geht demnach von 0 bis 100 %. 19–22, 24, 25, 27, 29, 30, 32–34, 46–48, 51–54, 60, 70, 71

Verhungern (engl. *starvation*) bezeichnet eine Situation, in der ein Prozess für immer wartet, weil er wieder und wieder von anderen Prozessen überholt wird (siehe Unterabschnitt 2.1.2). 6, 42, 54, 55, 74, 80, 83

Zwischenspeicher (engl. *cache*) bezeichnet einen schnellen Speicher, der genutzt wird, um wiederholte Zugriffe auf dieselbe Speicheradresse zu beschleunigen. Die Daten werden beim ersten Zugriff aus dem langsamen Hauptspeicher in den schnellen Zwischenspeicher geladen und liegen so bei erneutem Zugriff bereits vor. Um Hauptspeicher und Zwischenspeicher kohärent zu halten, wird ein Zwischenspeicher-Kohärenz-Mechanismus benötigt [CF78]. 5–7, 16, 30, 41, 47, 49, 51, 55, 62, 67, 68, 70, 74, 75, 77, 80, 83

Akronyme

CAS *Compare-and-Swap.* 4, 7, 8, 30, 37, 39, 42, 47, 73

CCWB *Changing Critical Work Benchmark.* 25, 27–30, 33, 34, 37, 46–48, 52, 53, 60, 63–65, 69, 70, 75, 79

CV *Coefficient of Variation.* 21

ECSB *Empty Critical Section Benchmark.* 24, 25, 27–30, 34, 36, 45, 46, 48, 51, 52, 60, 61, 63, 65, 66, 75, 79

FIFO *first in first out.* 6, 7, 24, 69

HPC *High Performance Computing.* 1, 12, 29, 72, 79, 80

HTM *Hardware Transactional Memory.* 71

LRZ Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften. 20, 71

MESIF *modified, exclusive, shared, invalid and forward.* 68

MPI *Message Passing Interface.* 9, 11–16, 18, 20, 21, 23, 27, 29–31, 33, 35–37, 39, 42, 49, 50, 55, 56, 68, 79, 80, 83

NUMA *Non Uniform Memory Access.* vii, 1–3, 10, 12, 18, 21, 30–32, 34, 36, 39–43, 52–55, 58, 69, 72–74, 77, 79, 80

PGAS *Partitioned Global Address Space.* 9, 14, 17

RDMA *Remote Direct Memory Access.* 9, 12, 14, 26, 30

RMA *Remote Memory Access.* 9–11, 14–19, 24, 26, 31, 34, 36, 39–41, 43, 47, 55, 56, 58, 59, 62, 64, 71, 79, 80, 83

TAS *Test-and-Set.* 4–6, 42, 56, 57, 62, 63, 66, 67, 74, 75, 77

TTS *Test-and-Test-and-Set.* 5, 6, 42–44, 54, 56, 57, 61–64, 66, 67, 72–75, 77

UMA *Uniform Memory Access.* 10

UPB *Uncontested Performance Benchmark.* 22, 23, 27, 28, 31, 34, 38, 39, 44, 51–53, 59, 62, 63, 66, 71, 75, 79

WBAB *Wait Before Acquire Benchmark.* 29, 30, 32–34, 36, 37, 44, 48, 53, 61, 65, 66, 70, 75, 79

Literaturverzeichnis

- [AAC⁺10] ARIMILLI, Baba ; ARIMILLI, Ravi ; CHUNG, Vicente ; CLARK, Scott ; DENZEL, Wolfgang ; DRERUP, Ben ; HOEFLER, Torsten ; JOYNER, Jody ; LEWIS, Jerry ; LI, Jian ; NI, Nan ; RAJAMONY, Ram: The PERCS High-Performance Interconnect. In: *2010 18th IEEE Symposium on High Performance Interconnects*, 2010. – ISSN 2332–5569, S. 75–82
- [AC89] AGARWAL, A. ; CHERIAN, M.: Adaptive Backoff Synchronization Techniques. In: *SIGARCH Comput. Archit. News* 17 (1989), April, Nr. 3, 396–406. <http://dx.doi.org/10.1145/74926.74970>. – DOI 10.1145/74926.74970. – ISSN 0163–5964
- [AEK⁺02] AUSLANDER, Marc ; EDELSOHN, David ; KRIEGER, Orran ; ROSENBURG, Bryan ; WISNIEWSKI, Robert: *Enhancement to the MCS lock for increased functionality and improved programmability*. US20030200457A1. <https://patents.google.com/patent/US20030200457A1>. Version: 2002
- [Alm11] Kapitel PGAS (Partitioned Global Address Space) Languages. In: ALMASI, George: *Encyclopedia of Parallel Computing*. Boston, MA : Springer US, 2011. – ISBN 978–0–387–09766–4, 1539–1545
- [Amd67] AMDAHL, Gene M.: Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In: *Proceedings of the April 18–20, 1967, Spring Joint Computer Conference*. New York, NY, USA : Association for Computing Machinery, 1967 (AFIPS '67 (Spring)). – ISBN 9781450378956, 483–485
- [And90] ANDERSON, T. E.: The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. In: *IEEE Trans. Parallel Distrib. Syst.* 1 (1990), Januar, Nr. 1, 6–16. <http://dx.doi.org/10.1109/71.80120>. – DOI 10.1109/71.80120. – ISSN 1045–9219
- [ARK10] ALVERSON, Robert ; ROWETH, Duncan ; KAPLAN, Larry: The Gemini System Interconnect. In: *2010 18th IEEE Symposium on High Performance Interconnects*, 2010. – ISSN 2332–5569, S. 83–87
- [BBY19] BROCK, Benjamin ; BULUÇ, Aydin ; YELICK, Katherine: BCL: A Cross-Platform Distributed Data Structures Library. In: *Proceedings of the 48th International Conference on Parallel Processing*. New York, NY, USA : Association for Computing Machinery, 2019 (ICPP 2019). – ISBN 9781450362955
- [BCM10] BOND, Michael D. ; COONS, Katherine E. ; MCKINLEY, Kathryn S.: PACER: Proportional Detection of Data Races. In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. New

- York, NY, USA : Association for Computing Machinery, 2010 (PLDI '10). – ISBN 9781450300193, 255–268
- [BK11] BECK, Motti ; KAGAN, Michael: Performance Evaluation of the RDMA over Ethernet (RoCE) Standard in Enterprise Data Centers Infrastructure. In: *Proceedings of the 3rd Workshop on Data Center - Converged and Virtual Ethernet Switching*, International Teletraffic Congress, 2011 (DC-CaVES '11). – ISBN 9780983628323, S. 9–15
- [BR10] BRUGGENCATE, Monika ten ; ROWETH, Duncan: DMAAPP - An API for One-sided Program Models on Baker Systems. In: *Cray User Group Conference*, 2010
- [CDT⁺14] CHEN, Zhezhe ; DINAN, James ; TANG, Zhen ; BALAJI, Pavan ; ZHONG, Hua ; WEI, Jun ; HUANG, Tao ; QIN, Feng: MC-Checker: Detecting Memory Consistency Errors in MPI One-Sided Applications. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE Press, 2014 (SC '14). – ISBN 9781479955008, 499–510
- [CF78] CENSIER, L. M. ; FEAUTRIER, P.: A New Solution to Coherence Problems in Multicache Systems. In: *IEEE Trans. Comput.* 27 (1978), Dezember, Nr. 12, 1112–1118. <http://dx.doi.org/10.1109/TC.1978.1675013>. – DOI 10.1109/TC.1978.1675013. – ISSN 0018–9340
- [CFMC15] CHABBI, Milind ; FAGAN, Michael ; MELLOR-CRUMMEY, John: High Performance Locks for Multi-Level NUMA Systems. In: *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA : Association for Computing Machinery, 2015 (PPoPP 2015). – ISBN 9781450332057, 215–226
- [CLL⁺02] CHOI, Jong-Deok ; LEE, Keunwoo ; LOGINOV, Alexey ; O'CALLAHAN, Robert ; SARKAR, Vivek ; SRIDHARAN, Manu: Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. New York, NY, USA : Association for Computing Machinery, 2002 (PLDI '02). – ISBN 1581134630, 258–269
- [CMC16] CHABBI, Milind ; MELLOR-CRUMMEY, John: Contention-Conscious, Locality-Preserving Locks. In: *SIGPLAN Not.* 51 (2016), feb, Nr. 8. <http://dx.doi.org/10.1145/3016078.2851166>. – DOI 10.1145/3016078.2851166. – ISSN 0362–1340
- [Cra93] CRAIG, Travis: Building FIFO and priority queuing spin locks from atomic swap / University of Washington, Department of Computer Science and Engineering, FR-35. Seattle, WA 98195, Februar 1993 (93-02-02). – Forschungsbericht
- [Dij65] DIJKSTRA, E. W.: Solution of a Problem in Concurrent Programming Control. In: *Commun. ACM* 8 (1965), September, Nr. 9, 569. <http://dx.doi.org/10.1145/365559.365617>. – DOI 10.1145/365559.365617. – ISSN 0001–0782

- [DK19] DICE, Dave ; KOGAN, Alex: Compact NUMA-Aware Locks. In: *Proceedings of the Fourteenth EuroSys Conference 2019*. New York, NY, USA : Association for Computing Machinery, 2019 (EuroSys '19). – ISBN 9781450362818
- [DK21] DICE, Dave ; KOGAN, Alex: Fissile Locks. In: GEORGIOU, Chryssis (Hrsg.) ; MAJUMDAR, Rupak (Hrsg.): *Networked Systems*. Cham : Springer International Publishing, 2021. – ISBN 978-3-030-67087-0, 192–208
- [DK22] DICE, Dave ; KOGAN, Alex: Hemlock : Compact and Scalable Mutual Exclusion. In: *CoRR* abs/2102.03863v4 (2022), Januar. <https://arxiv.org/abs/2102.03863v4>
- [DMS11] DICE, Dave ; MARATHE, Virendra J. ; SHAVIT, Nir: Flat-Combining NUMA Locks. In: *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*. New York, NY, USA : Association for Computing Machinery, 2011 (SPAA '11). – ISBN 9781450307437, 65–74
- [DMS12] DICE, David ; MARATHE, Virendra J. ; SHAVIT, Nir: Lock Cohorting: A General Technique for Designing NUMA Locks. In: *SIGPLAN Not.* 47 (2012), Februar, Nr. 8, 247–256. <http://dx.doi.org/10.1145/2370036.2145848>. – DOI 10.1145/2370036.2145848. – ISSN 0362–1340
- [DPFT19] DIEP, Thanh-Dang ; PHAM, Kien T. ; FÜRLINGER, Karl ; THOAI, Nam: A time-stamping system to detect memory consistency errors in MPI one-sided applications. In: *Parallel Computing* 86 (2019), 36-44. <http://dx.doi.org/https://doi.org/10.1016/j.parco.2019.04.013>. – DOI <https://doi.org/10.1016/j.parco.2019.04.013>. – ISSN 0167–8191
- [EA03] ENGLER, Dawson ; ASHCRAFT, Ken: RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. New York, NY, USA : Association for Computing Machinery, 2003 (SOSP '03). – ISBN 1581137575, 237–252
- [FFK16] FUERLINGER, Karl ; FUCHS, Tobias ; KOWALEWSKI, Roger: DASH: A C++ PGAS Library for Distributed Data Structures and Parallel Algorithms. In: *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2016, S. 983–990
- [FK11] FATOUROU, Panagiota ; KALLIMANIS, Nikolaos D.: A Highly-Efficient Wait-Free Universal Construction. In: *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*. New York, NY, USA : Association for Computing Machinery, 2011 (SPAA '11). – ISBN 9781450307437, 325–334
- [FK12] FATOUROU, Panagiota ; KALLIMANIS, Nikolaos D.: Revisiting the Combining Synchronization Technique. In: *SIGPLAN Not.* 47 (2012), Februar, Nr. 8, 257–266. <http://dx.doi.org/10.1145/2370036.2145849>. – DOI 10.1145/2370036.2145849. – ISSN 0362–1340

- [Fra86] FRANCEZ, Nissim: *Fairness*. 1. Springer, New York, NY, 1986 (Texts and Monographs in Computer Science). <http://dx.doi.org/10.1007/978-1-4612-4886-6>. <http://dx.doi.org/10.1007/978-1-4612-4886-6>. – ISBN 978–1–4612–4886–6
- [GBH18] GERSTENBERGER, Robert ; BESTA, Maciej ; HOEFLER, Torsten: Enabling Highly Scalable Remote Memory Access Programming with MPI-3 One Sided. In: *Commun. ACM* 61 (2018), September, Nr. 10, 106–113. <http://dx.doi.org/10.1145/3264413>. – DOI 10.1145/3264413. – ISSN 0001-0782
- [GHTL14] GROPP, William ; HOEFLER, Torsten ; THAKUR, Rajeev ; LUSK, Ewing: *Using Advanced MPI: Modern Features of the Message-Passing Interface*. The MIT Press, 2014. – ISBN 0262527634
- [HIST10] HENDLER, Danny ; INCZE, Itai ; SHAVIT, Nir ; TZAFRIR, Moran: Flat Combining and the Synchronization-Parallelism Tradeoff. In: *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*. New York, NY, USA : Association for Computing Machinery, 2010 (SPAA '10). – ISBN 9781450300797, 355–364
- [Inf07] InfiniBandSM Trade Association: *InfiniBandTM Architecture Specification Volume 1, Release 1.2.1*. November 2007
- [KCC⁺19] KASHYAP, Sanidhya ; CALCIU, Irina ; CHENG, Xiaohe ; MIN, Changwoo ; KIM, Taesoo: Scalable and Practical Locking with Shuffling. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. New York, NY, USA : Association for Computing Machinery, 2019 (SOSP '19). – ISBN 9781450368735, 586–599
- [KMK17] KASHYAP, Sanidhya ; MIN, Changwoo ; KIM, Taesoo: Scalable NUMA-Aware Blocking Synchronization Primitives. In: *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*. USA : USENIX Association, 2017 (USENIX ATC '17). – ISBN 9781931971386, S. 603–615
- [Lam13] LAMETER, Christoph: NUMA (Non-Uniform Memory Access): An Overview: NUMA Becomes More Common Because Memory Controllers Get Close to Execution Units on Microprocessors. In: *Queue* 11 (2013), jul, Nr. 7, 40–51. <http://dx.doi.org/10.1145/2508834.2513149>. – DOI 10.1145/2508834.2513149. – ISSN 1542–7730
- [LCPB⁺21] LIMA CHEHAB, Rafael L. ; PAOLILLO, Antonio ; BEHRENS, Diogo ; FU, Ming ; HÄRTIG, Hermann ; CHEN, Haibo: CLoF: A Compositional Lock Framework for Multi-Level NUMA Systems. In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. New York, NY, USA : Association for Computing Machinery, 2021 (SOSP '21). – ISBN 9781450387095, 851–865
- [LDT⁺12] LOZI, Jean-Pierre ; DAVID, Florian ; THOMAS, Gaël ; LAWALL, Julia ; MULLER, Gilles: Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications. In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USA : USENIX Association, 2012 (USENIX ATC'12), S. 6

- [LNS06] LUCHANGCO, Victor ; NUSSBAUM, Dan ; SHAVIT, Nir: A Hierarchical CLH Queue Lock. In: *Proceedings of the 12th International Conference on Parallel Processing*. Berlin, Heidelberg : Springer-Verlag, 2006 (Euro-Par'06). – ISBN 3540377832, 801–810
- [MCS91a] MELLOR-CRUMMEY, John M. ; SCOTT, Michael L.: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. In: *ACM Trans. Comput. Syst.* 9 (1991), Februar, Nr. 1, 21–65. <http://dx.doi.org/10.1145/103727.103729>. – DOI 10.1145/103727.103729. – ISSN 0734–2071
- [MCS91b] MELLOR-CRUMMEY, John M. ; SCOTT, Michael L.: Synchronization without Contention. In: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA : Association for Computing Machinery, 1991 (ASILOPS IV). – ISBN 0897913809, 269–278
- [MHSN15] MOLKA, Daniel ; HACKENBERG, Daniel ; SCHONE, Robert ; NAGEL, Wolfgang E.: Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. In: *Proceedings of the 2015 44th International Conference on Parallel Processing (ICPP)*. USA : IEEE Computer Society, 2015 (ICPP '15). – ISBN 9781467375870, 739–748
- [MLH94] MAGNUSSON, Peter S. ; LANDIN, Anders ; HAGERSTEN, Erik: Queue Locks on Cache Coherent Multiprocessors. In: *Proceedings of the 8th International Symposium on Parallel Processing*. USA : IEEE Computer Society, 1994. – ISBN 0818656026, 165–171
- [MPI09] Message Passing Interface Forum: *MPI: A Message-Passing Interface Standard*. <https://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf>. Version: 2.2, September 2009
- [MPI15] Message Passing Interface Forum: *MPI: A Message-Passing Interface Standard*. <https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>. Version: 3.1, Juni 2015
- [NAW06] NAIK, Mayur ; AIKEN, Alex ; WHALEY, John: Effective Static Race Detection for Java. In: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA : Association for Computing Machinery, 2006 (PLDI '06). – ISBN 1595933204, 308–319
- [NR98] NUMRICH, Robert W. ; REID, John: Co-Array Fortran for Parallel Programming. In: *SIGPLAN Fortran Forum* 17 (1998), August, Nr. 2, 1–31. <http://dx.doi.org/10.1145/289918.289920>. – DOI 10.1145/289918.289920. – ISSN 1061–7264
- [OTY99] OYAMA, Yoshihiro ; TAURA, Kenjiro ; YONEZAWA, Akinori: Executing parallel programs with synchronization bottlenecks efficiently. In: *Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications* Bd. 16 Citeseer, 1999, S. 95

Literaturverzeichnis

- [PFH06] PRATIKAKIS, Polyvios ; FOSTER, Jeffrey S. ; HICKS, Michael: LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA : Association for Computing Machinery, 2006 (PLDI '06). – ISBN 1595933204, 320–331
- [PRS15] PETROVIĆ, Darko ; ROPARS, Thomas ; SCHIPER, André: On the Performance of Delegation over Cache-Coherent Shared Memory. In: *Proceedings of the 2015 International Conference on Distributed Computing and Networking*. New York, NY, USA : Association for Computing Machinery, 2015 (ICDCN '15). – ISBN 9781450329286
- [RH02] RADOVIĆ, Zoran ; HAGERSTEN, Erik: Efficient Synchronization for Nonuniform Communication Architectures. In: *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. Washington, DC, USA : IEEE Computer Society Press, Nov 2002 (SC '02). – ISBN 076951524X, 1–13
- [RS84] RUDOLPH, Larry ; SEGALL, Zary: Dynamic Decentralized Cache Schemes for Mimd Parallel Processors. In: *SIGARCH Comput. Archit. News* 12 (1984), Januar, Nr. 3, 340–347. <http://dx.doi.org/10.1145/773453.808203>. – DOI 10.1145/773453.808203. – ISSN 0163–5964
- [SBH16] SCHMID, Patrick ; BESTA, Maciej ; HOEFLER, Torsten: High-Performance Distributed RMA Locks. In: *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. New York, NY, USA : Association for Computing Machinery, 2016 (HPDC '16). – ISBN 9781450343145, 19–30
- [SBN⁺97] SAVAGE, Stefan ; BURROWS, Michael ; NELSON, Greg ; SOBALVARRO, Patrick ; ANDERSON, Thomas: Eraser: A Dynamic Data Race Detector for Multithreaded Programs. In: *ACM Trans. Comput. Syst.* 15 (1997), November, Nr. 4, 391–411. <http://dx.doi.org/10.1145/265924.265927>. – DOI 10.1145/265924.265927. – ISSN 0734–2071
- [So08] SO, Stephen: Why is the sample variance a biased estimator? (2008), September. – Signal Processing Laboratory, Griffith School of Engineering, Griffith University, Brisbane, QLD, Australia, 4111
- [SS01] SCOTT, Michael L. ; SCHERER, William N.: Scalable Queue-Based Spin Locks with Timeout. In: *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*. New York, NY, USA : Association for Computing Machinery, 2001 (PPoPP '01). – ISBN 1581133464, 44–52
- [UPC13] UPC Consortium: *UPC Language Specifications*. <https://upc.lbl.gov/docs/user/upc-lang-spec-1.3.pdf>. Version: 1.3, November 2013
- [YRC05] YU, Yuan ; RODEHEFFER, Tom ; CHEN, Wei: RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. New York, NY, USA

- : Association for Computing Machinery, 2005 (SOSP '05). – ISBN 1595930795, 221–234
- [YY20] YI, ZhengMing ; YAO, YiPing: A scalable lock on NUMA multicore. In: *Concurrency and Computation: Practice and Experience* 32 (2020), Nr. 24, e5964. <http://dx.doi.org/https://doi.org/10.1002/cpe.5964>. – DOI <https://doi.org/10.1002/cpe.5964>
- [ZLW⁺16] ZHANG, Mingzhe ; LAU, Francis C. M. ; WANG, Cho-Li ; CHENG, Luwei ; CHEN, Haibo: Scalable Adaptive NUMA-Aware Lock: Combining Local Locking and Remote Locking for Efficient Concurrency. In: *SIGPLAN Not.* 51 (2016), Februar, Nr. 8. <http://dx.doi.org/10.1145/3016078.2851176>. – DOI 10.1145/3016078.2851176. – ISSN 0362–1340
- [ZMI⁺14] ZHOU, Huan ; MHEDHEB, Yousri ; IDREES, Kamran ; GLASS, Colin W. ; GRA-CIA, José ; FÜRLINGER, Karl: DART-MPI: An MPI-Based Implementation of a PGAS Runtime System. In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. New York, NY, USA : Association for Computing Machinery, 2014 (PGAS '14). – ISBN 9781450332477