Advanced Multiprocessor Programming

List-based Set

Daniel Zainzinger, Matr. Nr. 11777778
Noah Bruns, Matr. Nr. 11777778
e11777778@student.tuwien.ac.at
e01630029@student.tuwien.ac.at

Vienna, 22. Juni 2020

Inhaltsverzeichnis

1 Einleitung

Für das Programmieren mit mehreren Prozessoren ist es oft erforderlich, eine gemeinsame Datenstruktur zu verwenden. Eine Möglichkeit ist das List based Set. Dabei handelt es sich um eine Liste, bei welcher jedes Element einen Key besitzt. Dieser Key ist eindeutig und die gesamte Liste ist anhand dieses Keys sortiert. Um das Testen und Benchmarken der Datenstrukturen einfach zu halten, werden nur positive Integer Werte in die Datenstruktur eingefügt (siehe 4.1). Es gibt mehrere Optionen für die Implementatierung einer solchen Liste. In Folgendem Kapitel werden 5 verschiedene Implementirungen vergestellt die verschidenene Techniken verwenden um die Synchronisierung und den sicheren Zugriff der gemeinsamen Datenstruktur zu gewährleisten. Eine Anleitung zum kompilieren und ausführen kann aus dem "README.TXT" entnommen werden.

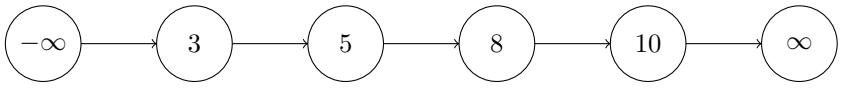

Abbildung 1: Aufbau der Liste

In der Abbildung 1 wird dargestellt wie eine solche Liste aussieht. Am Anfang und am Ende befinden sich 2 Nodes $-\infty$ und ∞ die den Anfang und das Ende der Liste markieren. Dazwischen befinden sich in sortierter Reihenfolge die eingefügten Keys.

Jede der Implementierungen enthält 3 Methoden die mit denen die Liste bearbeitet werden kann:

add Fügt ein Elemet in die Liste ein. Falls dieses schon enthalten ist gibt die Methode *false* zurück sonst *true*.

contains Wenn das Element in der Liste gefunden wurde gibt diese Methode *true* oder sonst *false* zurück.

remove Diese Methode entfernt einen Key aus der Liste und gibt *true* zurück. Falls der Key nicht in der Liste vorhanden ist wird *false* zurückgegeben.

2 Implementierung

2.1 List-based set with coarse-grained locks

Bei dieser Implementierung wir vor jedem Methodenaufruf die gesamte Liste gesperrt. Dadurch kann aber die Parallelisierung nicht wirklich ausgenutzt werden, da zu einem Zeitpunkt maximal ein Prozess auf die Liste zugreifen kann. Diese Implementierung ist daher sehr ähnlich zu einer sequentiellen Implementierung.

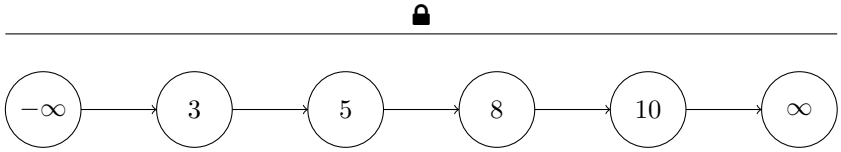

Abbildung 2: coarse-grained locks

Linearisation Point	Die Sperrung
add	Nachdem die Liste gesperrt wurde wird sie durchlaufen bis zu dem Punkt an dem der Key der betrachteten Node größer ist als der Einzufügende. Vor dieser wird die Neue Node eingefügt.
contains	Die Liste wird gesperrt und dann durchlaufen. Wenn dabei der gesuchten Wert gefunden wird, wird *true* sont *false* zurückgegeben.
remove	Nachdem die gesamte Liste gesperrt wurde wird das Element in der Liste gesucht und entfernt.

2.2 List-based set with fine-grained locks

Im Unterschied zu den coarse-grained locks wird bei dieser Implementierung immer nur die derzeitige Node gesprerrt. Beim Iterieren der Liste wird während dem Übergang zur nächsten Node die derzeitige und die darauffolgende Node gesperrt. Eine Node kann zu einem Zeitpunkt nur von einem Prozess gesperrt sein. Der Vorteil gegenüber den coarse-grained locks besteht darin, dass nun wirkliche Parallelisierung möglich ist und verschiedene Threads an unterschiedlichen Punkten an der Liste arbeiten können. Es hat jedoch den Nachteil, dass die Iterationsgeschwiendigkeit im Worst Case durch den langsamsten Thread definiert wird, da ein Thread in der Liste nicht "Überholt" werden kann.

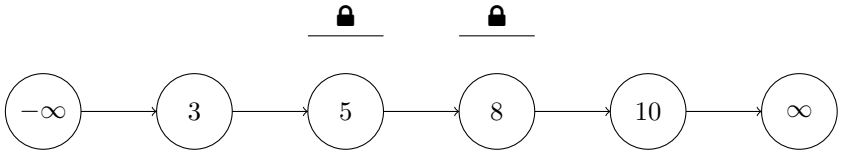

Abbildung 3: fine-grained locks

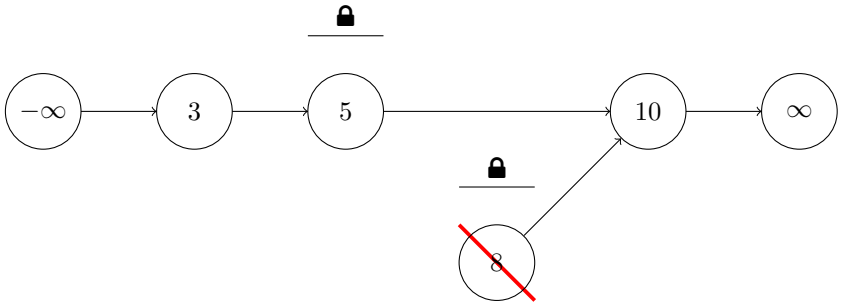

Abbildung 4: fine-grained locks remove

Linearisation Point Wenn der Key in der Liste ist dann der Punkt an dem die zu suchende Node gesperrt wurde. Sonst wenn die nächst höhere Node gesperrt wurde.

add Die Liste wird iteriert (mit Sperrungen) und an dem Punkt an dem die Node eingefügt werden soll wird sowohl der Vorgänger als auch der Nachfolger gesperrt. Dadurch kann die neue Node sicher eingefügt werden.

contains Die Liste wird analog zur coarse-grained locks iteriert dabei aber immer die derzeitige Node bzw. beim Übergang Vorgänger und Nachfolger gesperrt. Wenn die entwprechende Node gefunden wurde wird *true* sonst *false* zurückgegeben.

remove Die Liste wird wie bei contains iteriert und wenn die zu entfernende Node gefunden wurde wird der Zeiger der Vorgänger Node auf den darauffolgenden verwiesen und somit die zu entfernende Node übersprungen. Siehe in Abbildung 5

Diese Implementierung setzt Memory Management vorraus. Dies ist in Kapitel 3 genauer beschrieben.

2.3 List-based set with optimistic synchronization

In dieser Implementierung wird beim Suchen eines Elementes zuerst keine Node gesperrt sonder einfach nur eine Iteration durchgeführt. Sobald das gesuchte gefunden wurde wird dieses gesperrt und sichergestellt, dass diese Node immernoch erreichbar ist. Falls dies nicht der Fall sein sollte, wird startet der Algorithmus von vorne. Dies hat den Vorteil, dass beim Suchen eines Keys in der Liste keine Sperrungen benötigt werden. Erst wenn das Element gefunden wurde wird es gesperrt.

Linearisation Point ???

add	Beim Hinzufügen wird die Liste durchlaufen ohne Nodes zu sperren bis die entsprechende Punkt zum Einfügen gefunden wurde. Die vorgänger und nachfolger Nodes werden gesperrt und mittels erneutem durchlaufen der Liste wird sichergestellt, dass diese Nodes immernoch Teil der Liste sind. Falls dies scheitert wird der Prozess von vorne gestartet. Sind jedoch die gesperrten Nodes immernoch erreichbar so wir der neue Key zwischen den gesperrten eingefügt und die Sperre wird aufgehoben.
contains	Hier wird ebenfalls die Liste ohne Sperrungen durchsucht und nach Erfolgreichen Sperren plus der Validierung der Erreichbarkeit der Nodes wird das Ergebnis zurückgegeben.
remove	Gleich wie bei den anderen Methoden wird der Key gesucht, gesperrt und validiert. Danach wird das Element aus der Liste entfernt und alle Sperrungen werden aufgehoben.

Diese Implementierung setzt Memory Management vorraus. Dies ist in Kapitel 3 genauer beschrieben.

2.4 List-based set with lazy synchronization

Lazy Synchronization verwendet eine Markierung auf den Nodes die bekannt gibt ob diese Node noch aktiv ist oder nicht. Bei einem Lösch-Vorgang wird bei der zu löschenden Node diese Markierung gesetzt. Dadurch wird diese Node bei anderen Methoden, wie zum Beispiel Contains oder Add, ignoriert. Dies hat den Vorteil, dass für den Aufruf von der *contains* Methode keine Sperrungen mehr notwendig sind.

In späterer folge wird durch das Memory management, ähnlich wie bei optimistic synchronization, die Node gelöscht sobald sichergestellt werden kann, dass kein Process mehr auf die Node zugreift.

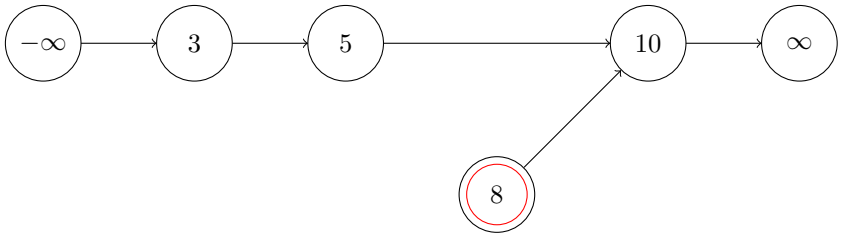

Abbildung 5: fine-grained locks remove

Linearisation Point ???

add	Gleich wie bei den fine-grained locks
contains	Die liste wird durchlaufen und bei einem erfolgreichem Fund wird sichergestellt, dass dieses Element noch nicht markiert wurde.
remove	Der Key wird gesucht und mitsamt seinem Vorgänger gesperrt. Danach wird Validiert ob dieses Element immernoch erreichbar ist. Wenn dies der Fall ist wird das Element markiert und einseitig aus der Liste entfernt, das bedeutet dass diese Node weiterhin auf den Nachfolger zeigt aber die vorherige Node wird ebenfalls auf den Nachfolger verwiesen. Dies ist in Abbildung 5 verdeutlicht.

Diese Implementierung setzt Memory Management vorraus. Dies ist in Kapitel 3 genauer beschrieben.

2.5 Lock-free list-based set

Das Ziel dieser Implementierung ist es eine Sperrungsfreie Liste zu erstellen. Dazu wird ähnlich wie bei der *lazy synchronization* eine Markierung verwendet. Diese Markierung wird aber in den next-Pointer der Node integriert damit beide *atomic* gelesen und geschrieben werden können. Um dies zu ermöglichen kann einfach ein Bit des Pointers genommen werden. Da bei einem 64-Bit Computer nicht alle Bits für die Adressierung des Speichers verwendet werden, kann das höchste Bit als Markierung umfunktioniert werden.

Linearisation Point ???

add	Bei add wird zuerst die der Punkt zum Einfügen gesucht und dann wird eine neue Node erzeugt die auf die nachfolger Node zeigt. Danach wird mittels *compare_exchange* der Vorgänger auf die neue Node verwiesen und somit die neue Node in die Liste eingefügt. Scheitert *compare_exchange* wird der Prozess von Neuem gestartet.
contains	Gleich wie bei *lazy synchronization*
remove	Bei remove wird die zu entfernende Node zuerst gesucht und dann mittels *compare_exchange* markiert. Falls *compare_exchange* scheitert startet der Prozess wieder von vorne oder sonst wird wieder mittels *compare_exchange* versucht die Node noch aus der Liste zu entfernen indem in der Vorgänger Node der next-Pointer auf den Nachfolger gelegt wird. Falls dies jedoch scheitert werde keine werteren Versuche zum Entfernen der Node aus der Liste vorgenommen.

stimmt das?

2.5.1 Verbesserungen

In der *find* Funktion wird für das Auslinken eines Nodes wird ein Compare and Swap(CAS) verwendet. Falls dies fehlschlägt, weil z.b. der Node zum beispiel bereits ausgelinkt wurde, wird wieder beim Listenkopf begonnen. Um dieses Verhalten zu verbessern, wurde in *Lock_free_impr.cpp* folgendes implementiert:
Es wird immer der alte *predecessor* gespeichert. Sollte CAS fehlschlagen, wird überprüft ob der alte *predecessor* zum auslinken markiert wurde. Ist dies der Fall, werden *predecessor, current* und *successor* auf den jeweiligen Vorgänger gesetzt und an dieser Stelle weitergemacht.

3 Memmory Managment

Bei *List-based set with optimistic synchronization*, *List-based set with lazy synchronization* und *Lock-free list-based set* gibt es ein Problem mit den Memmory Managment. Bei diesen Datenstrukturen ist das lesen eines Nodes auch ohne einen Lock möglich. Dadurch kann von einen Thread nicht sichergestellt werden, dass nach dem unlinken eines Nodes dieser nicht noch von einen anderen Thread gelesen wird.
Beispiel:
Angenommen Thread A liest einen Node und legt sich schlafen. Thread B will diesen Thread löschen. Dazu linkt er den Node aus der Datenstruktur aus . Anschließend wird dieser von Thread A gelöscht. Wenn nun Thread A wieder aufwacht, und auf den Node zugreiffein will, ist der Speicher breits freigegeben und es tritt ein Segfault auf.

3.1 Makierung beim Lesen

Als erster Ansatz wurde versucht, mithilfe eines Counters einen Node zu makieren, dass dieser gerade gelesen wird. Dieser Ansatz ist jedoch nicht Zielführend und ist nur zum verdeutlichen der Schwierigkeit dieses Problems erwähnt.
Bevor ein Thread einen Node liest, erhöht er einen Counter in diesen Node. Nachdem ein Node verlassen wurde, wird dieser Counter wieder verringert. Ein Thread, der diesen Node löschen will, darf dies erst tun, sobald der Node ausgelinkt und der Counter den Wert 0 hat.
Damit ein Thread den nächsten Node markieren kann, muss er zuerst den “next“ Pointer aus den aktuellen Node auslesen und dann den Counter erhöhen (next-¿counter++). Es ist jedoch nicht möglich, das Auslesen des next Pointers und das erhöhen des Counters atomar auszuführen. Auch ein “compare and swap“ ist hierbei nicht hilfreich, da das vergleichen mit einen freigegebenen Speicherbereich zu einen Segmentation Fault führt.

3.2 Delete Queue mit Zeitstempel

Um zu verbeiden, dass ein Node gelöscht wird, wärend er von einen anderen Thread noch gelesen wird, muss sichergestellt werden, dass dieser “frei“ ist. Dazu wurden Zeitstempel eingeführt. Jeder

Thread besitzt einen globalen Integer, wo sein Zeitstempel gespeichert wird. Sobald eine Operation wie Beispielsweise “add“ abgeschlossen ist, wird dieser Zeitstempel erhöht. Will nun ein Thread einen Node löschen, wird dieser zuerst ausgelinkt. Anschließend werden alle Zeitstempel ausgelesen und gemeinsam mit einen Pointer auf den zu löschenden Node in einer Queue hinzugefügt. Diese Queue wird lokal im Thread gespeichert und wurde als C++ Vektor realisiert. Nach den Abschluss einer Operation wei Beispielsweise “add“, wird die Queue mit zu löschenden Kandidaten durchlaufen. Für jeden Node werden nun die gespeicherten Zeitstempel mit den aktuellen Zeitstempel verglichen. Wenn sich alle Zeitstempel eines Nodes von den aktuellen Zeitstempel unterscheiden, ist sichergestellt, dass dieser Node von keinen Thread mehr gelesen wird. Nun kann der Speicher des Nodes wieder Freigegeben werden und aus der Queue entfernt werden.
Bevor ein Thread geschlossen wird, muss sichergestellt werden dass die Queue gelehrt wird und somit der Speicher aller Kandidaten gelöscht wird. Da hierbei die Zeitstempel nicht mehr überprüft werden, kann die Liste nach dem Schließen eines Threads nicht mehr verwendet werden.

4 Benchmark

Beim Ausführen von ./main wird ein Benchmark test durchgeführt. Dieser gibt die jeweils benötigte Zeit aus und erstellt ein .csv für jede datenstruktur in den “result“ folder.

4.1 Aufbau

Es werden zwei files für einen Test benötigt: pre[0-9]+.csv und main[0-9]+.csv. Mithilfe der Einträge im pre-file wird die Datenstruktur befüllt. Sobald dies abgeschlossen ist, werden alle negativen Werte im main-file aus der Datenstruktur gelöscht und alle positiven Einträge hinzugefügt. Somit sind in der Liste nur positive Werte zulässig. Die implementierten Datenstrukturen würden grundsätzlich aber auch mit negativen Werten und anderen Datentypen funktionieren. Um jedoch einen einfachen Benchmark zu realisieren sind nur Einträge zwischen 1 und 2.147.483.646 möglich. Nach jeder Entfernung ung Hinzufügung eines Wertes wirt mithilfe von *contain* überprüft, ob das element hinzugefügt bzw. entfernt wurde. Als dritter und letzter Schritt wird mithilfe von *contain* überprüft, ob sich alle positiven Werte aus dem main-File in der Datenstruktur befinden und ob alle negativen Werte aus der Datenstruktur entfernt wurden.
pre[0-9]+.csv und main[0-9]+.csv sind so aufgebaut, dass im main-File alle Werte aus dem pre-file mit negativen Vorzeichen vorhanden sind und die positiven Werte nicht im pre-File vorkommen. Desweiteren werden alle Einträge in den beiden Files zufällig permutiert.

4.2 Testcases erstellen

Die Testcases werden mithilfe eines Python Skriptes erstellt. Dabei ist die größe der Testcases mithilfe der Konstanten RAW_BASE und COOLUMS_BASE einzustellen. Mit FILE_AMOUNT kann eingestellt werden, wie viele Testcases erstellt werden. Dabei werden bei jeder Interation RAW_BASE und COOLLUM_BASE mit der aktuellen Iteration-Variable multipliziert. Dies entspricht einen quadratischen Anstieg der Einträge in den Testfiles.

4.3 Interpretation Benchmark Werte

Die Dauer jedes einzelnen Tests und ob der Test erfolgreich war wird in der Kommandozeile ausgegeben. Desweiteren wird im Ordner *results* für jede Datenstruktur ein CSV File angelegt. Die Beschreibung der einzelnen Features kann aus Tablelle 1 entnommen werden.

Algemeines	
Cores:	Anzahl der verwendeten Cores
TestSizePre	Anzahl der Werte, die im ersten Schritt in die Datenstruktur eingefügt wurden
TestSizeMain	Anzahl der Werte, die im zweiten Schritt hinzugefügt oder entfernt wurden

Ausschließliches einfügen in die Datenstruktur nur *add()*	
time write	Benötigte Zeit, um die Werte aus dem ersten Schritt in die Datenstruktur einzufügen
goToStart write	Anzahl der Vorfälle, die ein erneutes Durchsuchen der Datenstruktur vom Beginn an erfordern
lostTime write	Zeit die durch ein erneutes Durchsuchen Verloren geht.

Gemischter Zugriff *add()*, *remove()* und *contain()*	
time mixed	Benötigte Zeit, um die Werte aus dem zweiten Schritt in die Datenstruktur einzufügen bzw. zu entfernen und jeweils anschließend zu überprüfen
goToStart mixed	Anzahl der Vorfälle, die ein erneutes Durchsuchen der Datenstruktur vom Beginn an erfordern
lostTime mixed	Zeit die durch ein erneutes Durchsuchen Verloren geht.

Lesender Zugriff nur *contain()*	
time check	Benötigte Zeit, um zu überprüfen, ob alle Werte aus dem zweiten Schritt hinzugefügt bzw. entfernt wurden
goToStart check	Anzahl der Vorfälle, die ein erneutes Durchsuchen der Datenstruktur vom Beginn an erfordern
lostTime check	Zeit die durch ein erneutes Durchsuchen Verloren geht.

Tabelle 1: CSV Spalten

4.4 Wiederholungen

Um ein möglichst genaues Ergebnis zu erhalten, müssen die Benchmarks wiederholt werden. Dazu steht in *benchmark.hpp* die Konstante REPEAT_TESTS zur Verfügung. In den CSV-Files werden dann die Durchschnittswerte aller Wiederholungen eingetragen.

5 Resultate

Bei den Resultaten handelt es sich um die Datenstrukturen, welche in Kapitel 2 beschreiben sind. Bei Erweiterung "_impr" handelt es sich um eine Verbesserte Version vie in Kapitel 3 beschrieben. Bei Datenstrukturen mit der Erweiterung "_mem" ist ein Memorymanagement implementiert (siehe 3). Alle Benchmarks wurden auf den TU Nebula Maschine ausgeführt. Aufgrund der beschrankten Rechenzeit auf dieser Maschine, wurden die Test nur 10 Mal wiederholt. Die folgenden Abbildungen zeigen jeweils den Mittelwert aller Wiederholungen. Dabei ist Anzumerken, das die Anzahl der Threads und die Anzahl der verwendeten Cores bei allen Tests identisch ist.

5.1 Laufzeit mit 64 Cores

5.1.1 Schreibvorgang

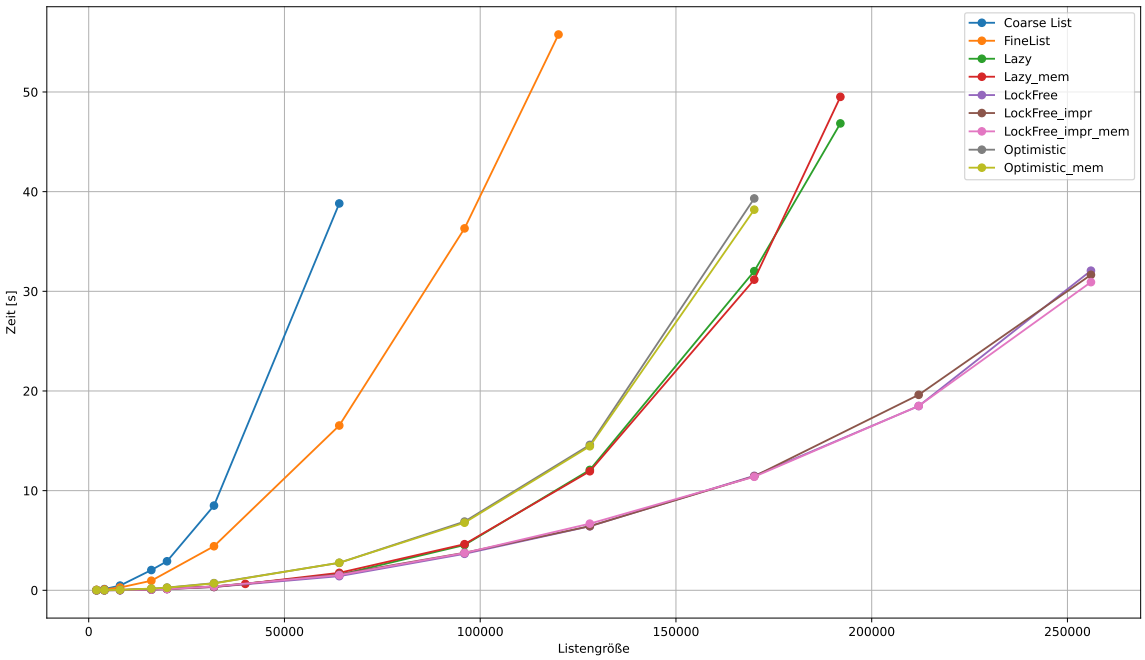

Abbildung 6: Laufzeit des Schreibvorganges

Die Abbildung 6 zeigt die benötigte Laufzeit in Sekunden für das Einfügen von Elementen in eine leere Liste. Dabei ist auf der X-Achse die Anzahl der eingefügten Elemente zu sehen. Das *List-based set with coarse-grained locks* ist dabei am Langsamsten, da nur jeweils ein Node gleichzeitig Zugriff auf die Liste hat. Die schnellste Datenstruktur ist die *lock-free improved with memorymanagement* Liste.

5.1.2 Gemischter Zugriff

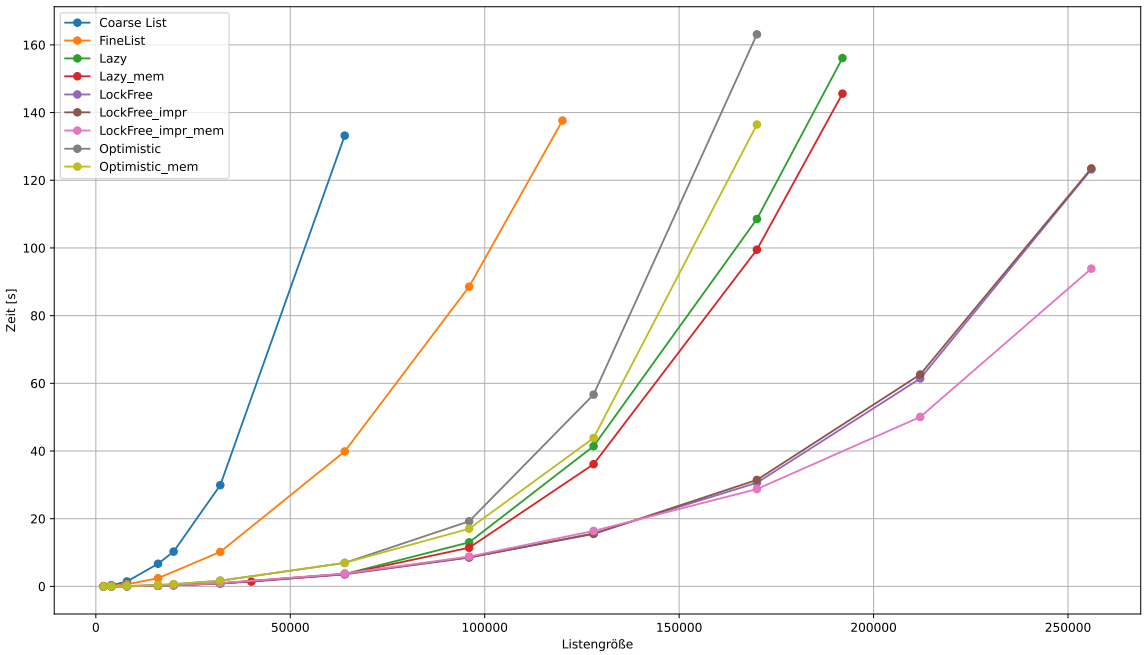

Abbildung 7: Laufzeit bei gemischten Zugriff

Die Abbildung 6 zeigt die benötigte Laufzeit bei gemischten Zugriff. Dabei sind die Zugriffe folgendermaßen aufgeteilt: 50% *contain()*, 25% *add()* und 25% *remove()*. Dabei ist auf der X-Achse die größe der Liste zu sehen, was auch jeweils der Anzahl der hinzugefügten und entfernten Daten entspricht. Auffallend ist hier, dass Datenstrukturen mit einem implementierten Memorymanagement bei großen Listen schneller sind, obwohl das Memorymanagement zusätzlichen aufwand bedeutet. Eine mögliche Begründung liegt darin, dass dem Betriebssystem immer wieder Speicher zurückgegeben wird und somit das auslagern von Daten aus dem Cache minimiert wird.

5.1.3 Lesender Zugriff

Die Abbildung 8 zeigt die benötigte Laufzeit, wenn auf die Datenstruktur nur mit *contain()* zugegriffen wird. Dabei wurden alle Listenelemente abgefragt. Desweiteren wurde die selbe Anzahl an Daten abgefragt, welche sich nicht in der Datenstruktur befinden. Somit wurden doppelt so viele *contain()* Anfragen ausgeführt, wie Listenelemente vorhanden sind. Auffallend ist hier, dass beispielsweise die Lazy-Liste bei großen Listen erheblich länger benötigt, als die Lock-free list, obwohl die *contain()* Funktionen fast identisch sind. Eine mögliche Ursache ist die unterschiedliche größe der Nodes. Die Lazy-Liste beinhaltet in ihren Nodes zusätzlich eine Variable für Mutex und einen Boolean-Variable zum markieren der gelöschten Nodes. Dieser Größenunterschied könnte für das Memorymanagement des Betriebssystem einen erhöhten Aufwand beim laden der Nodes beim durchlaufen bedeuten. Dies ist jedoch eine Vermutung und konnte nicht bestätigt werden.

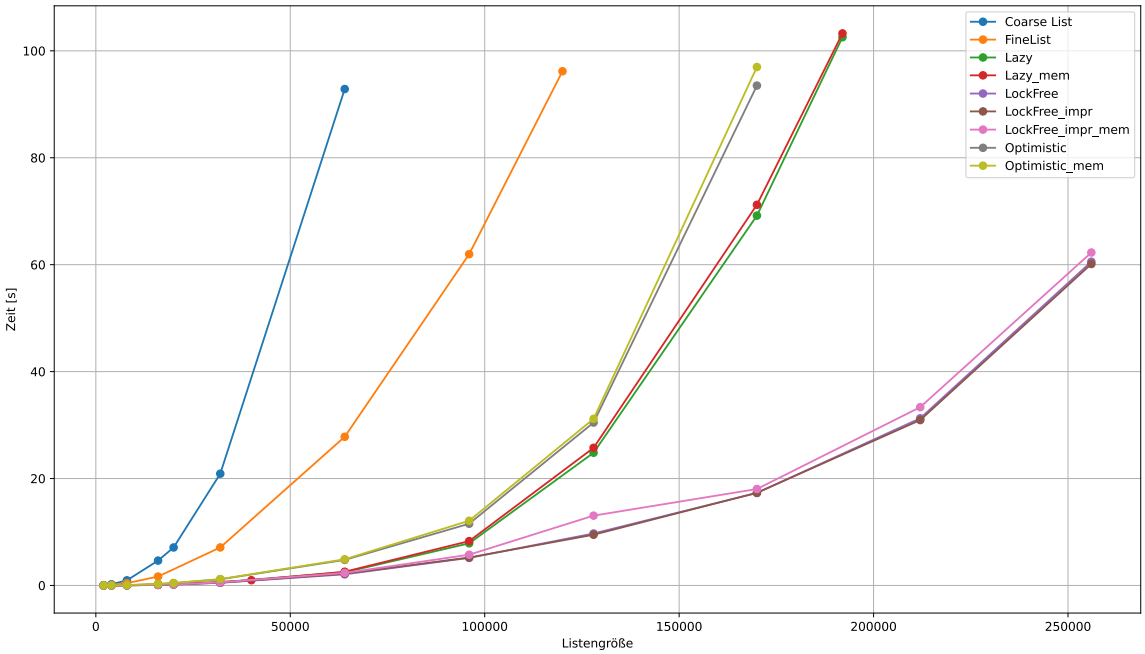

Abbildung 8: Laufzeit für Lesevorgang

5.2 Neustarts von Zugriffen

Wie bereits in Kapitel 2.5.1 beschrieben, kann es vorkommen, dass Datenstrukturen aufgrund eines konfliktes mit einem anderen Thread einen laufenden Task neu beginnen müssen. In Abbildung 10 ist zu sehen, wie oft eine laufende Funktion abgebrochen werden musste und wieder beim Listenkopf beginnen musste. In Abbildung 9 ist ersichtlich, wie viel Zeit für diese Neustarts aufgewendet werden musste.

Bei einer Listengröße von 256 000 mit 64 Cores und 512 000 Zugriffen startet die Lock-free Liste 3163 mal neu. Dies bedeutet einen zusätzlichen Zeitaufwand von 21 Sekunden. Somit spendet jeder Core 0,33 Sekunden mit Neustarts. Bei einer Laufzeit von 123204 Sekunden entspricht das 0,00028% der Laufzeit. Da wie in Abbildung 10 ersichtlich, steigt die zusätzliche Zeit durch Neustart exponentiell. Somit wird dies bei noch größeren Listen relevant. Im zuge dieses Projektes wurden jedoch keine größeren Listen getestet.

5.3 Laufzeit mit unterschiedlichen Cores

5.3.1 Vergleich 20 000 Listeneinträgen

Die Abbildung 11 zeigt die Laufzeit mit einer Listengröße von 20000 Einträgen bei *contain()*, 1000 *add()* und 100 *remove()* zugriffen. Auffallend ist hier, dass das *fine-grained locks* mit zwei Cores fast drei mal so lange benötigt als mit einem Core, da sich die einzelnen Threads gegenseitig behindern. Weiters ist auch ersichtlich, dass bei der *coarse-grained locks* Liste die Laufzeit erhöht wird, je höher die Anzahl der verwendeten Cores. Warum??

Vergleicht man die jeweils schnellsten Datenstrukturen, kommt man auf folgende Ergebnisse: Ein Core mit der *coarse-grained locks* Liste benötigt 4636 Sekunden und 64 Cores mit der *lock-free*

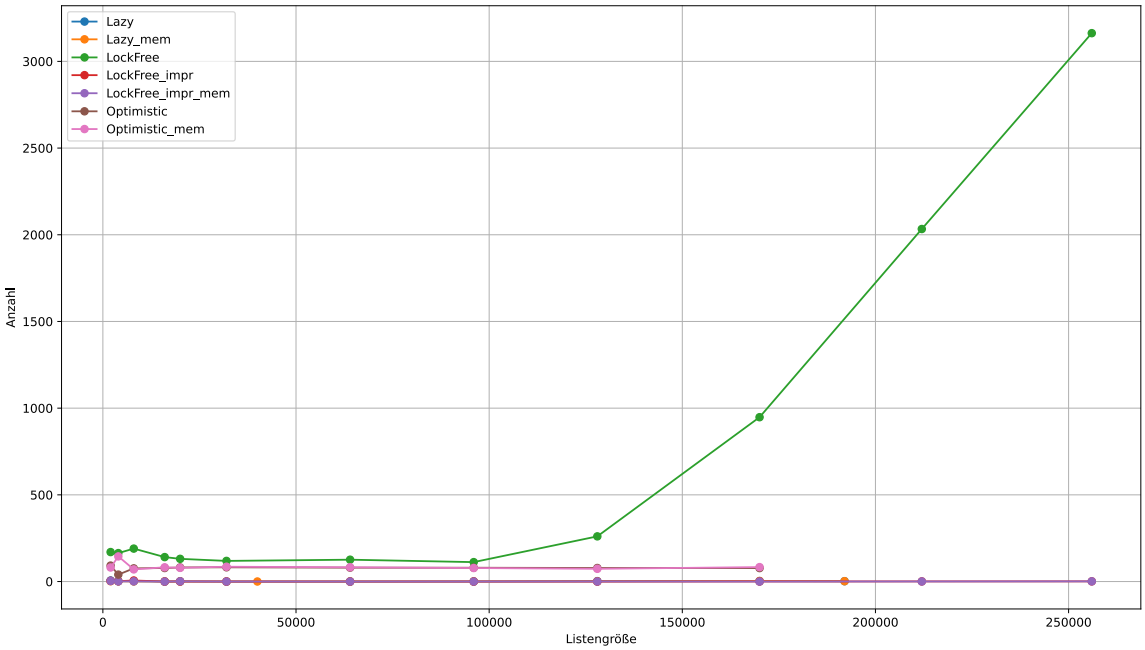

Abbildung 9: Neustarts

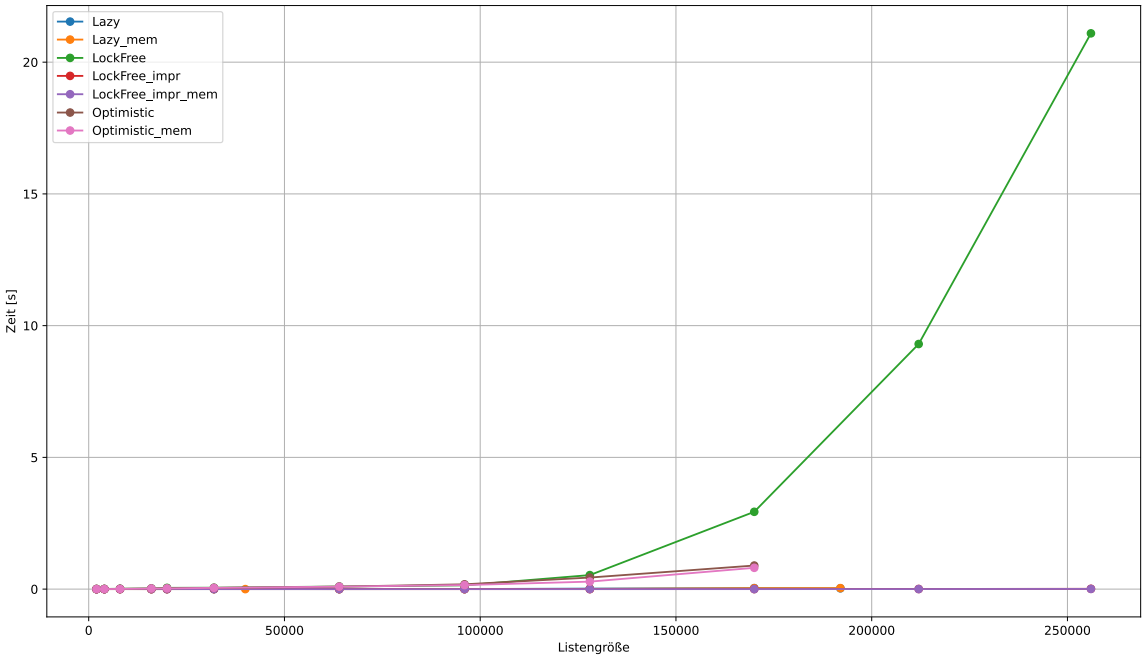

Abbildung 10: Zusätzliche Zeit durch Neustarts

improved with memorymanagement Liste benötigen 311 Sekunden. Das entspricht einen Speedup von 14,9.

Vergleicht man nun den Speedup der jeweils Schnellsten Datenstrukturen bei 20000 Listeneinträgen zwischen einem und zwei Cores mit den Speedup von 32 und 64 Cores, dann erhält man folgende Ergebnisse:

Seedup T1/T2=1,3

Seedup T2/T4=1,94
Seedup T8/T16=1,64
Seedup T16/T32=1,31
Seedup T32/T64=1,43

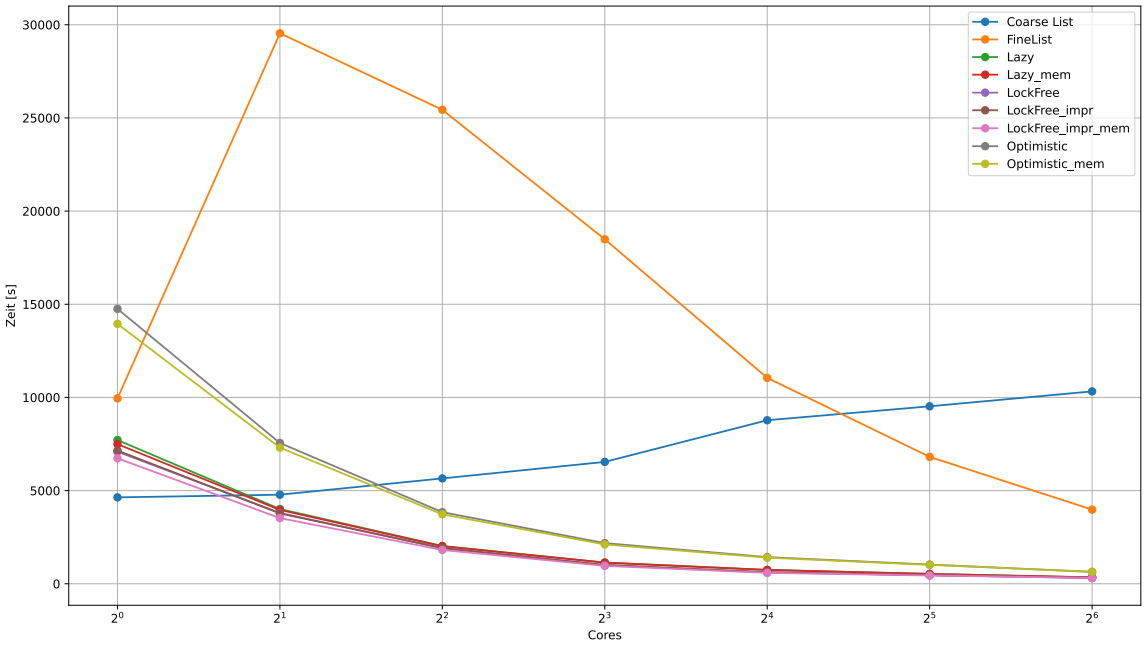

Abbildung 11: Laufzeit mit 20 000 Listeneinträgen

5.3.2 Vergleich 40 000 Listeneinträgen

Um die schnelleren Datenstrukturen besser zu vergleichen zu können, zeigt Abbildung 12 die Laufzeit mit verschiedenen Cores mit der doppelten Anzahl an Listeneinträgen und zugegriffen. Vergleicht man nun die Speedups der schnellsten Datenstruktur (*lock-free improved with memorymanagement*) mit 40 000 Einträgen, kommt man auf folgende Werte:
Seedup T16/T32=1,37
Seedup T32/T64=1,65

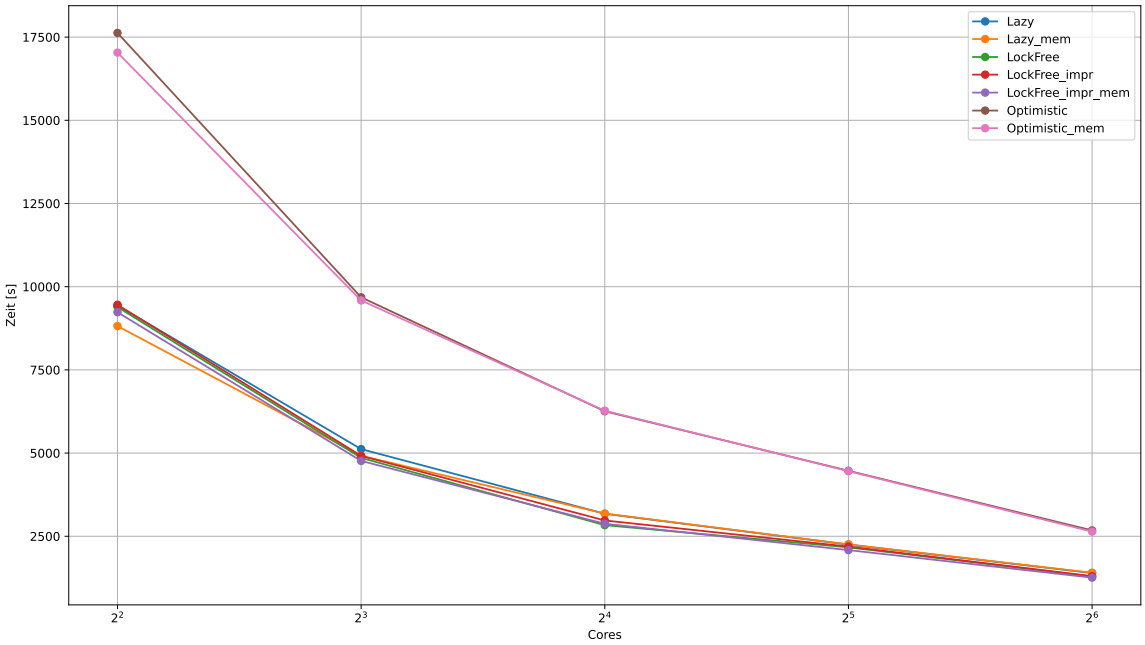

Abbildung 12: Laufzeit mit 40 000 Listeneinträgen

5.3.3 Vergleich 120 000 Listeneinträgen

Die Abbildung 13 zeigt die Laufzeit bei 120 000 Listeneinträgen mit 240 000 zugriffen mit der gleichen Aufteilung wie die beiden vorangegengenen Abbildungen. Vergleicht man nun die Speedups der schnellsten Datenstruktur (*lock-free improved with memorymanagement* Liste) mit 120000 Einträgen, kommt man auf folgende Werte:
Seedup T16/T32=1.42
Seedup T32/T64=1,7
Vergleicht man diese Werte mit den Speedups von 5.3.1 und 5.3.2, ist ersichtlich, dass das vergrößern der Liste keinen signifikanten Einfluss auf den Speedup hat.

6 Fazit

Es wurden Unterschiedliche List-baset Sets implementiert und die Performance evaluiert. Hierbei wurde zwei Kategorien untersucht. Zum einen die Performance bei steigender Listengröße und die Laufzeit mit unterschiedlicher Anzahl von Cores. Es war ersichtlich, dass bei der Verendung von nur einen Core die Liste mit einem globalen Lock am schnellsten ist. Schon ab der Verwendung von 2 Threads sind andere Listen besser. Es hat sich auch gezeigt, dass das Memmorymanagment, also das zurückgeben von nicht mehr verwendeten Speicher, positive auswirkungen auf die Performance. Ein weiteres interessantes Resultat war, dass sich die Laufzeit bei der Liste mit *fine-grained locks* mit zwei Threads im Vergleich zu einem Thread fast verdreifacht. Es war auch gezeigt, dass Listen bei einer Kollision mit anderen Threads abbrechen müssen und dadurch wieder beim Listenkopf beginnen müssen. Dieses Verhalten kann mit geringen Programmieraufwand erheblich verbessert werden. Hierbei gibt es jedoch noch weiteres Verbesserungspotential.
Ein weiteres nicht erwartetes Ergebnis war, dass die *lazy* Liste für eine *contain()* Abfrage erheblich

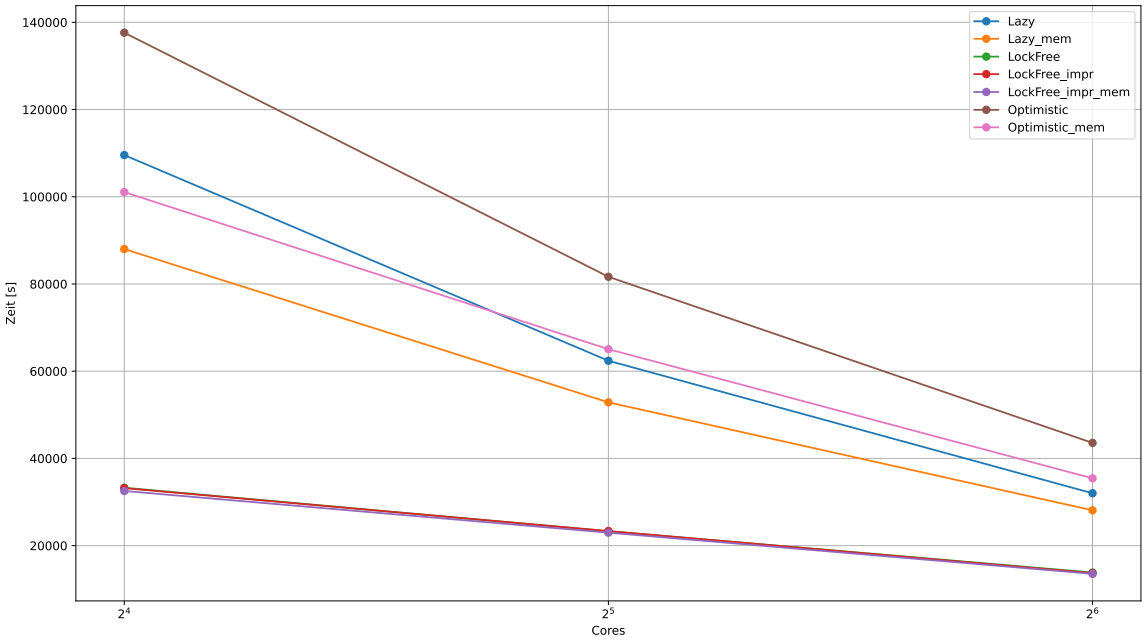

Abbildung 13: Laufzeit mit 120 000 Listeneinträgen

länger benötigt als die *lock-free* benötigt, obwohl die Funktionen fast identisch sind. In diesen Projekt sind alle Zugriffe in zufälliger Reihenfolge erfolgt. Eine weitere Möglichkeit, die Performance zu evaluieren, wäre der Zugriffen welche nicht gleichverteilt sind. Da die Dichte der Zugriffe auf einigen Stellen der Liste dadurch höher wird, ist davon auszugehen, dass sich die Performance dadurch verändert. Dies wurde hier jedoch nicht behandelt und wäre eine Möglichkeit für ein zukünftiges Projekt.