Spawn & Merge

Eine Programmierabstraktion zur deterministischen Synchronisation Verteilter Systeme

Von der Fakultät für Ingenieurwissenschaften,
Abteilung Informatik und Angewandte Kognitionswissenschaft
der Universität Duisburg-Essen

zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften

genehmigte Dissertation

von
Christopher Boelmann
aus
Geldern

1. Gutachter: Prof. Dr.-Ing. Torben Weis
2. Gutachter: Prof. Dr. Arno Wacker

Tag der mündlichen Prüfung: 26.02.2018

Abstract

Existing approaches for deterministic execution run all code sections in a fully deterministic manner, resulting in high performance costs and a loss of scalability. In this thesis we develop a programming model for a scalable deterministic execution of distributed applications, which introduces less performance costs than fully deterministic systems.

We introduce the concept of *Application-level Determinism*, which, in contrast to full determinism, limits the deterministic execution to code sections that potentially influence the deterministic result of the application when executed concurrently. Based on this concept, the *Spawn & Merge* programming model automates the decision whether the execution of two code segments must be kept in order to maintain a deterministic result. The evaluation of a prototype of Spawn & Merge for distributed systems shows that applications with a high share of parallelizable code can scale efficiently (achieve up to 100% of maximum speedup possible) and guarantee the deterministic and reproducible execution of the application logic.

The performance gain competes with the costs for the determinism-enforcing mechanisms used by Spawn & Merge: Operational Transformation (OT) and waiting conditions introduced. The majority of potential waiting conditions is automatically dealt with by internal dynamic scheduling of the parallel parts of the application. The remaining waiting conditions are further reduced by introducing a modified OT system that allows for an *efficient deterministic merge in any given order*. The costs for OT depend on the application and can take up most of the execution time (up to a worst case of 97,5% in the performed measurements) when many modifications of shared data structures are performed and when there is a high amount of synchronization between the parts of the application that are executed in parallel. This is due to the computational complexity of $O(n^2)$ for the OT systems used. However, these costs for OT are constant for an application for a given input. Thus, the share of OT on the overall application runtime reduces with rising parallelism. Therefore, the feasibility of Spawn & Merge for an application depends on the parallelizable share of the application, the amount of performed modifications of shared data structures, and the amount of internal synchronizations.

Zusammenfassung

Bestehende Ansätze zur deterministischen Ausführung führen alle Codebereiche einer Anwendung voll-deterministisch aus, was mit hohen Performancekosten und einem Verlust der Skalierbarkeit einhergeht. In dieser Arbeit wird ein Programmiermodell entwickelt, das eine skalierbare deterministische Ausführung einer verteilten Anwendung mit geringeren Performancekosten (gegenüber vollem Determinismus) ermöglicht.

Dazu wird das Konzept des *Determinismus auf Applikationsebene* eingeführt, das im Gegensatz zu vollem Determinismus die deterministische Ausführung auf diejenigen Codebereiche beschränkt, deren nebenläufige Ausführung einen Einfluss auf das deterministische Ergebnis hat. Das darauf aufbauende *Spawn & Merge Programmiermodell* ermöglicht die automatisierte Entscheidung, ob die Ausführungsreihenfolge zweier Codebereiche für ein deterministisches Ergebnis beibehalten werden muss. Die Evaluation eines Prototyps für Spawn & Merge in Verteilten Systemen zeigt, dass verteilte Spawn & Merge Anwendungen, die einen hohen parallelisierbaren Anteil haben, effizient skalieren können (bis zu 100% der maximal erreichbaren Beschleunigung), während eine deterministische Ausführung der Anwendungslogik garantiert wird.

Dem Performancegewinn stehen die Kosten für die Mechanismen gegenüber, die den Determinismus der Anwendung ermöglichen und sich aus den Kosten für die intern verwendete Operational Transformation (OT) und den eingeführten Wartebedingungen zusammensetzen. Der Großteil der potenziellen Wartebedingungen wird durch ein internes dynamisches Scheduling der parallel ausgeführten Anteile der Anwendung verhindert. Die verbleibenden Wartebedingungen wurden durch ein angepasstes OT-System, das eine effiziente *deterministische Zusammenführung in beliebiger Reihenfolge* ermöglicht, weiter reduziert. Die Höhe der OT-Kosten ergibt sich aus der Anwendung und kann einen Großteil der Ausführungszeit einnehmen (im Worst Case bis zu 97,5% in den durchgeführten Messungen), wenn viele Modifikationen an geteilten Datenstrukturen durchgeführt und häufig zu parallel ausgeführten Anteilen der Anwendung synchronisiert werden. Das liegt an der Berechnungskomplexität $O(n^2)$ der verwendeten OT-Systeme. Die OT-Kosten sind allerdings für eine Anwendung konstant (für feste Eingabedaten). Somit sinkt der Anteil der OT-Kosten an der Gesamtlaufzeit bei steigender Parallelität. Die Eignung von Spawn & Merge für eine Anwendung ist somit abhängig vom parallelisierbaren Anteil, der Anzahl durchgeführter Modifikationen an geteilten Datenstrukturen und der Häufigkeit von Synchronisationen innerhalb der Anwendung.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Wissenschaftlicher Beitrag . . . 3
1.2 Struktur der Arbeit . . . 5

2 Stand der Forschung **7**
2.1 Synchronisation nebenläufiger Systeme . . . 7
2.1.1 Sperren geteilter Ressourcen . . . 8
2.1.2 Synchronisation von Systemen mit geteiltem Arbeitsspeicher . . . 12
2.1.3 Synchronisation Verteilter Systeme . . . 15
2.2 Determinismus und Reproduzierbarkeit . . . 17
2.2.1 Determinismusarten . . . 19
2.2.2 Determinismus in Systemen mit geteiltem Arbeitsspeicher . . . 23
2.2.3 Determinismus in Verteilten Systemen . . . 28
2.2.4 Zusammenfassung . . . 30

3 Determinismus auf Applikationsebene **33**
3.1 Synchronisation nebenläufiger Prozesse . . . 34
3.1.1 Events und Happened-before Relationen . . . 34
3.1.2 Synchronisationspunkte . . . 35
3.2 Voll-deterministische Systeme . . . 37
3.2.1 Reduzierung des Performanceverlustes durch vollen Determinismus 38
3.3 Determinismus auf Applikationsebene . . . 40
3.3.1 Anforderungen an ein Programmiermodell . . . 42

4 Deterministische Synchronisation mit Spawn & Merge **43**
4.1 Anforderungen und Lösungsansatz . . . 43
4.1.1 Standardmäßiger Determinismus . . . 44
4.1.2 Reduktion von Synchronisationspunkten . . . 46
4.1.3 Entwicklerfreundlichkeit . . . 47
4.2 Spawn & Merge Programmiermodell . . . 47

4.2.1 Art der Parallelisierung 48
4.2.2 Use-Case Beschreibung 50
4.2.3 Tasks 54
4.2.4 Initialisierung des Frameworks 56
4.2.5 Zusammenführbare Datenstrukturen 57
4.2.6 Spawn 60
4.2.7 Merge 63
4.2.8 Bedingter Merge 66
4.2.9 Sync 69
4.2.10 SpawnSibling 74
4.2.11 Abort 92
4.3 Beispielhafte Einbettung in C++11 94
4.4 Abbildung typischer Synchronisationsbeispiele 101
4.5 Äquivalenz zu Semaphoren 108
4.6 Deadlockfreiheit 112

5 Mechanismen für Spawn & Merge 115
5.1 Operational Transformation 115
5.1.1 Operationen 116
5.1.2 Funktionsweise 118
5.1.3 Klassifikation 120
5.1.4 Alternativen zu Operational Transformation 122
5.2 Operational Transformation in Spawn & Merge 122
5.2.1 Operational Transformation im `Merge`-Vorgang 123
5.2.2 Datenstrukturen und versionsbasierter Merge 124
5.2.3 Alternative Mechanismen in Spawn & Merge 128
5.2.4 Kritik an Operational Transformation 129
5.3 Deterministischer Merge in beliebiger Reihenfolge 130
5.3.1 Herausforderungen bei beliebiger Merge-Reihenfolge 130
5.3.2 Analyse Spawn & Merge spezifischer Eigenschaften 132
5.3.3 OT-Algorithmus für beliebige Merge-Reihenfolge 133
5.3.4 Komplexität des neuen Algorithmus 149
5.4 Zusammenführbare Map 150
5.5 Verschachtlung zusammenführbarer Datenstrukturen 151

6 Spawn & Merge für Verteilte Systeme 157
6.1 Systemmodell 157
6.1.1 Architektur 157
6.1.2 Nachrichtenfluss 159

6.1.3 Fehlermodell 160
6.1.4 Serialisierbarkeit von Datenstrukturen 162
6.2 Umsetzung der Programmierabstraktion 163
6.2.1 Einordnung der neuen Mechanismen 163
6.2.2 NodeMaster 166
6.2.3 SchedulingMaster 170
6.2.4 Nachrichtenprotokoll 177
6.2.5 Parameterübertragung 179
6.2.6 Spezialfall: Lokale Ausführung 182
6.3 Einordnung in den Stand der Forschung 184

7 Evaluation 187
7.1 Versuchsaufbau 187
7.2 Szenarien 188
7.2.1 Normale Verteilung 189
7.2.2 Extreme Verteilung 189
7.2.3 Damenproblem 190
7.2.4 Game of Life 192
7.2.5 Operational Transformation Szenario 193
7.3 Skalierbarkeit 194
7.3.1 Normale Verteilung 195
7.3.2 Extreme Verteilung 197
7.3.3 Damenproblem 198
7.3.4 Game of Life 200
7.3.5 Skalierbarkeit von Spawn & Merge 201
7.4 Lokale Optimierungen 204
7.5 Determinismuskosten 207
7.5.1 Operational Transformation Kosten 207
7.5.2 Wartezeiten durch die deterministische Reihenfolge 210
7.5.3 Verteilungskosten 212
7.6 Zusammenfassung der Ergebnisse 213

8 Fazit 215
8.1 Weiterer Forschungsbedarf 217

Liste der Abkürzungen 219

Glossar 221

A Tabellen **235**
A.1 Task-Handle Schnittstelle . . . 235
A.2 Zusammenführbare Datenstrukturen . . . 236
A.3 Informationen in der Datenstruktur-Historie Φ . . . 237
A.4 Vollständige Liste valider `OpCodes` . . . 237

B Code-Beispiele und Syntax **239**
B.1 `MergeAll`-Syntax . . . 239
B.2 `MergeAllByHandle`-Syntax . . . 239
B.3 Anwendung der Synchronisationsprimitive . . . 240
B.4 Dining Philosophers mit Spawn & Merge . . . 241

C Operational Transformation Beispiele **243**
C.1 Vollständige Transformationen für Abbildung 5.9 . . . 243
C.2 Vollständige Transformationen für Abbildung 5.9 (FCFS) . . . 244

Literaturverzeichnis **247**

Liste der Publikationen **257**

Abbildungsverzeichnis

2.1 Beispiel für einen Deadlock (Verklemmung). 10

3.1 Beispielanwendung mit drei kommunizierenden Prozessen. 34
3.2 Beispiel für einen Eventgraphen. 35
3.3 Beispiel für einen Synchronisationsgraphen mit einem Prozessor. 36
3.4 Beispiel für einen Synchronisationsgraphen mit zwei Prozessoren. 36
3.5 Verzögerung von Events bei vollem Determinismus. 38
3.6 Skalierbarkeit der deterministischen Anwendung. 39
3.7 Aufteilung der Determinismusgarantien. 41

4.1 Aufbau einer datenparallelen Anwendung. 49
4.2 Aufbau einer taskparallelen Anwendung. 50
4.3 Agentenbasierte Aufteilung. 51
4.4 Aufteilung nach Straßensegmenten. 52
4.5 Task Hierarchie. 55
4.6 Task-Hierarchie vs. Datenstruktur-Hierarchien. 59
4.7 Child-Task-Einordnung in die Task-Hierarchie. 75
4.8 *SpawnSibling* Task-Einordnung. 76
4.9 Sibling-Task-Einordnung bei *SpawnSibling*. 81
4.10 Datenstrukturkopien bei *SpawnSibling*. 83
4.11 Datenstruktur-Hierarchie bei *SpawnSibling* mit Sibling-Relation. 84
4.12 Anpassung von Φ_{D_1} bei *SpawnSibling* und *Merge*. 86
4.13 Ergebnis der Anpassung der Datenstruktur-Historie. 90
4.14 Anpassung der Datenstruktur-Historie Φ_{D_1} bei *Abort*. 95
4.15 Deadlock zwischen Parent-Task und Child-Task. 113

5.1 Ablauf der Konfliktauflösung mit Operational Transformation. 119
5.2 Operational Transformation Beispiel. 121
5.3 Ignorieren bereits bekannter Operationen. 126
5.4 Rückübertragung der transformierten Parent-Operationen. 127

5.5 Einfluss der einzuhaltenden `Merge`-Reihenfolge. 130
5.6 Operationen die bei beliebiger `Merge`-Reihenfolge zu Problemen führen (1). 131
5.7 Operationen die bei beliebiger `Merge`-Reihenfolge zu Problemen führen (2). 134
5.8 Einordnung des von Task T_2 per *SpawnSibling* gestarteten Tasks T_4. 136
5.9 Beispielszenario für die Anwendung der Kontrollfunktion. 143
5.10 Zusammenführungsgraphen für T_1 mit $T_2 - T_5$. 144
5.11 Beispielszenario für verschachtelte Operationen. 154

6.1 Beispielhafte Verteilung gespawnter Tasks auf MPI-Knoten. 159
6.2 Kommunikation zwischen den Komponenten des Prototyps. 160
6.3 Aufbau einer über MPI versendeten Nachricht. 160
6.4 NodeMaster im Kontext eines MPI-Knotens. 166
6.5 Berechnungskomplexität der Verkehrssimulation. 171
6.6 Einordnung des SchedulingMasters in das Systemmodell. 173
6.7 Workstealing bei ungleichmäßig verteilter Berechnungskomplexität. 175
6.8 Nachrichtenfluss für *Spawn* und *Merge*. 178
6.9 Nachrichtenfluss für *Sync*. 179
6.10 Nachrichtenfluss für *SpawnSibling*. 180
6.11 Serialisierung eines `Spawn`-Aufrufs in eine `SendWork`-Nachricht. 180

7.1 Das Damenproblem für $n = 4$. 191
7.2 Das *Game of Life* Zustandsgitter. 192
7.3 Speedup-Graph der NV_N Messung. 196
7.4 Speedup-Graph der NV_H Messung. 196
7.5 Speedup-Graph der EV_N Messung. 198
7.6 Speedup-Graph der EV_H Messung. 198
7.7 Speedup-Graph der DP_{14} Messung. 199
7.8 Speedup-Graph der GoL_N Messung. 200
7.9 Speedup-Graph der GoL_H Messung. 201
7.10 Speedup-Graph der GoL_X Messung. 202
7.11 Einfluss der Optimierungen für lokale Ausführung. 205
7.12 Mögliche eingesparte Laufzeit bei FCFS Reihenfolge. 211

Tabellenverzeichnis

2.1 Eigenschaften der verteilten deterministischen Systeme. 31

4.1 In einem Task gespeicherte Informationen (Task-Informationen). 55
4.2 In der Datenstruktur-Historie Φ gespeicherte Informationen. 58
4.3 Erweiterung der in der Datenstruktur-Historie Φ gespeicherten Informationen. 84
4.4 Blockierverhalten der Spawn & Merge Primitive. 113

6.1 Einordnung von Spawn & Merge in die deterministischen Verteilten Systeme.186

7.1 Abstand der gemessenen Beschleunigung zum Optimum (in Prozent). . . . 202
7.2 Veränderung der Standardabweichungen abhängig vom Scheduling. 203
7.3 Laufzeiten der OT-Szenarien (mit Angabe des OT-Anteils). 208
7.4 Einfluss der Zusammenführung in FCFS Reihenfolge. 211
7.5 Einfluss der Zusammenführung in FCFS Reihenfolge (Szenario EV_X). 211

A.1 Task-Handle Schnittstelle. 235
A.2 Task-Handle Schnittstelle. 236
A.3 Datenstruktur-Historie Φ (vollständig). 237
A.4 Nachrichten `OpCodes` und ihre Bedeutung. 237

Algorithmen und Code-Beispiele

4.1 Verkehrssimulation mit Spawn & Merge. 53
4.2 Definition der *InitSpawnMerge* Schnittstelle. 56
4.3 Initialisierung des Spawn & Merge Frameworks. 57
4.4 Benutzung der *Spawn*-Primitive. 60
4.5 Definition der *Spawn* Schnittstelle. 62
4.6 Randbedingung der *Spawn*-Primitive. 63
4.7 Definition der *MergeAll* Schnittstelle. 63
4.8 Definition der *MergeAllByHandle* Schnittstelle. 64
4.9 Randbedingung der *MergeAllByHandle*-Primitive. 65
4.10 Definition der *MergeAny* Schnittstelle. 65
4.11 Definition der *MergeAnyByHandle* Schnittstelle. 66
4.12 Benutzung der *Merge*-Varianten. 67
4.13 Definition der Schnittstelle der *Bedingungsfunktion*. 68
4.14 Benutzung der Bedingungsfunktion. 69
4.15 Benutzung der *Sync*-Primitive. 71
4.16 Definition der *Sync* Schnittstelle. 72
4.17 Starten eines Sibling-Tasks mit der *SpawnSibling*-Primitive. 77
4.18 Definition der *SpawnSibling* Schnittstelle. 77
4.19 Starten eines Sibling-Tasks ohne *SpawnSibling*. 79
4.20 Beispiel für Einordnung der Sibling-Tasks. 81
4.21 Algorithmus zur Aktualisierung einer Datenstruktur-Historie Φ. 87
4.22 Algorithmus zur rekursiven Aktualisierung der Sibling-Nachkommen. . . . 89
4.23 Algorithmus zur rekursiven Aktualisierung der Sibling-Vorfahren. 90
4.24 Definition der *Abort* Schnittstelle. 92
4.25 *Abort* Beispiel (von innen). 93
4.26 *Abort* Beispiel (von außen). 94
4.27 Funktionssignaturen Beispiele. 96
4.28 Template Definition. 96
4.29 Expandiertes Template. 97
4.30 Variadisches rekursives Template. 98

4.31 Variadisches `Spawn` Template. 99
4.32 Metaprogrammierung von Bedingungen. 99
4.33 Bedingte Templates für `CopyOne` (vereinfacht). 100
4.34 Producer-Consumer (nach Tanenbaum [92, Kapitel 2.3.8]). 103
4.35 Producer-Consumer mit Spawn & Merge. 105
4.36 Dining Philosophers (nach Tanenbaum [92, Kapitel 2.5.1]). 107
4.37 Unterschiedliche Aufgaben auf Listen mit Spawn & Merge. 108
4.38 Simulation einer *Semaphore* mit Spawn & Merge. 110

5.1 Transformationsfunktion *TRANSFORM* nach [58]. 120
5.2 Notwendigkeit der Merge-Epochen. 138
5.3 Angepasste Transformationsfunktion *FCFS_TRANSFORM*. 139
5.4 Kontrollfunktion für eine Zusammenführung in beliebiger Reihenfolge. . . 141
5.5 Transformationsfunktion *MAP_TRANSFORM*. 150
5.6 Erweiterte Transformationsfunktion *MAP_TRANSFORM*. 153
5.7 Um `change` erweiterte Transformationsfunktion *FCFS_TRANSFORM*. . . . 154

6.1 Definition der *BlockingCallStart* Schnittstelle. 169
6.2 Definition der *BlockingCallEnd* Schnittstelle. 169
6.3 Hinweis auf einen blockierenden Funktionsaufruf. 169
6.4 Deserialisierung eines `Spawn`-Aufrufs und *Currying*. 182
6.5 Aufruf der parameterlosen Funktion. 183

B.1 Vollständige Definition der *MergeAll* Schnittstelle. 239
B.2 Definition der *MergeAllByHandle* Schnittstelle. 239
B.3 Anwendung der Synchronisationsprimitive. 240
B.4 Dining Philosophers mit Spawn & Merge (trivial). 241

Kapitel 1

Einleitung

„It's hard to fix something that doesn't even fail reliably." — Bowen Alpern et al. [3]

Die deterministische Ausführung einer Anwendung ist ein nützliches Werkzeug für das Testen und die Fehlersuche in nebenläufigen Systemen. Sie stellt sicher, dass eine Berechnung bei wiederholter Ausführung dasselbe Ergebnis liefert. Diese Reproduzierbarkeit ist unter anderem im Bereich des wissenschaftlichen Rechnens und bei der Durchführung von Simulationen essenziell. So werden Simulationen beispielsweise für die Verifikation im Rahmen der Entwicklung von Fahrerassistenzsystem (FAS) genutzt, da die erforderlichen Testkilometer nur schwer allein durch Testfahrten in der realen Welt erbracht werden können. Tritt im Rahmen einer solchen Simulation ein unerwartetes Verhalten des simulierten FAS auf, z.B. dass das gesteuerte Fahrzeug einen Fußgänger anfährt, so müssen die Entwickler des FAS die Ursache dieses Fehlverhaltens finden und beheben können. Dazu müssen die Umstände, die zu dem Fehlverhalten des FAS geführt haben, reproduziert werden. Beinhaltet die Simulation nichtdeterministisches Verhalten, dann ist es möglich, dass das Fehlverhalten durch eine Race-Condition verursacht wurde (d.h. eine Situation, in der nebenläufige Prozesse in einer nicht geplanten Reihenfolge auf einen kritischen Codebereich zugegriffen haben). Solch eine Fehlerursache, deren Auftreten sich aus nichtdeterministischen Timings der Interaktion nebenläufiger Prozesse ergibt, ist schwer aufzufinden.

Die deterministische Ausführung einer Anwendung kann erreicht werden, wenn sich der Entwickler darauf beschränkt, nur einen Thread zu verwenden. Dies ist allerdings keine Option, wenn die Skalierbarkeit der Anwendung eine Anforderung ist. Um die deterministische Ausführung einer Anwendung mit mehreren nebenläufigen Threads zu ermöglichen, bieten die etablierten Programmiersprachen Mechanismen an, die eine Synchronisation zwischen den Threads ermöglichen. Mithilfe dieser Synchronisationsmechanismen muss die nebenläufige Ausführung der Threads (z.B. der Zugriff auf gemeinsam verwendete Ressourcen) durch den Entwickler soweit eingeschränkt werden, dass das Pro-

gramm für jede mögliche Ausführungsreihenfolge der Thread-Befehle (d.h. der Statements innerhalb eines Threads) ein korrektes Ergebnis liefert. Allerdings sind die typischerweise genutzten Synchronisationsmechanismen (z.B. Semaphoren, Mutexe oder Nachrichtenaustausch) nicht inhärent deterministisch, da das Synchronisationsergebnis von den Synchronisationszeitpunkten abhängig ist (Timings). Dieser Umstand führt häufig dazu, dass durch Programmierfehler weiterhin Deadlocks (Verklemmungen) oder Race-Conditions in einem Programm enthalten sein können, die trotz der korrekten Verwendung der Synchronisationsmechanismen zu unerwünschtem nichtdeterministischen Programmverhalten führen können.

Zur Überprüfung der Korrektheit nebenläufiger Anwendungen (d.h. der korrekten Verwendung der Synchronisationsmechanismen und der Abwesenheit von Deadlocks und Race-Conditions), müsste jede mögliche Verschachtlung nebenläufiger Thread-Befehle (Ausführungspfade) validiert werden, um eine vollständige Testabdeckung zu erreichen. Die Anzahl möglicher Ausführungspfade kann allerdings exponentiell mit der Verwendung von Synchronisationsmechanismen ansteigen, sodass eine vollständige Testabdeckung häufig nicht erreichbar ist. Des Weiteren ist das Erzwingen einer spezifischen Verschachtlung von Thread-Befehlen technisch anspruchsvoll für die Tester einer Anwendung. Ohne eine vollständige Testabdeckung ist es nur möglich durch ausführliches Testen der Anwendung (*Fuzzy Testing*) die Aussage zu treffen, dass die Anwendung auf Computern, die vor allem in Bezug auf die Anzahl und die Taktung der Prozessorkerne dem Test-Computer entsprechen, mit *hoher Wahrscheinlichkeit* korrekt ausgeführt werden wird. Ändert sich allerdings die Anzahl oder die Taktung der Prozessorkerne, so können neue Verschachtlungen der Thread-Befehle ermöglicht werden, die wiederum zu neuen Deadlocks und Race-Conditions führen können. Das Risiko, dass Deadlocks oder Race-Conditions zur Laufzeit auftreten, die beim Testen nicht auffindbar waren, verbleibt somit. Die Synchronisationsmechanismen der etablierten Programmiersprachen eignen sich daher nur schlecht dafür, Entwicklern die Programmierung deterministischer komplexer nebenläufiger Anwendungen zu ermöglichen.

Eine Alternative zu traditionellen Synchronisationsmechanismen stellt die Verwendung von Frameworks oder Programmiermodellen für deterministische Anwendungen dar. Diese Ansätze garantieren die deterministische Ausführung einer Anwendung oder deren deterministische Wiederholung und erlauben es somit, einen Programmablauf zu reproduzieren. Allerdings gehen die Garantien mit hohen Performancekosten und dem Wegfall der Skalierbarkeit der Anwendungen einher. Diese ergeben sich aus den jeweiligen verwendeten Ansätzen, nämlich der voll-deterministischen Ausführung oder der deterministischen Wiederholung.

Diese Arbeit befasst sich mit der Fragestellung, wie die Erstellung skalierender und effizienter deterministischer Anwendungen für Entwickler erleichtert werden kann. Dazu

wird insbesondere untersucht, inwiefern die Performancekosten für die Einführung einer deterministischen Programmausführung reduziert oder vermieden werden können.

1.1 Wissenschaftlicher Beitrag

Die in dieser Arbeit untersuchte Hypothese lautet: eine deterministische und somit reproduzierbare Ausführung einer verteilten Anwendung kann derart realisiert werden, dass geringere Performancekosten entstehen als bei einer voll-deterministischen Ausführung und dass die entsprechende Anwendung mit den verfügbaren Ressourcen skalieren kann. Die Hypothese ist darin begründet, dass in voll-deterministischen Systemen und in Systemen zur deterministischen Wiederholung alle Codebereiche deterministisch ausgeführt werden. Das betrifft insbesondere auch Codebereiche, deren nebenläufige Ausführung keinen Einfluss auf das deterministische Ergebnis haben. Der Grund dafür ist, dass die entsprechenden Laufzeitumgebungen und Frameworks nicht entscheiden können, in welchen Fällen die Reihenfolge zweier nebenläufiger Ausführungen für eine deterministische Ausführung eingehalten werden muss und auf welche Einhaltung zugunsten höherer Performance verzichtet werden kann. An diesem Punkt setzt diese Arbeit an.

Zur Untersuchung der Hypothese wird das Programmiermodell Spawn & Merge entwickelt, das die automatische Entscheidung, ob die Reihenfolge einer nebenläufigen Ausführung zweier Codebereiche eingehalten werden muss oder nicht, ermöglicht. Die Hypothese wird anschließend anhand eines Prototyps des Spawn & Merge Programmiermodells für Verteilte Systeme überprüft.

Als Grundlage für das Konzept des Programmiermodells wird das Konzept des *Determinismus auf Applikationsebene* eingeführt, das die Determinismusgarantien einer Anwendung in zwei voneinander getrennte Ebenen aufteilt. Die Applikationsebene wird dabei deterministisch ausgeführt unabhängig davon, ob die Anwendung auf einem Multiprozessorsystem oder einem Verteilten System ausgeführt wird. In der Applikationsebene liegt die Anwendungslogik, die somit auch bei wiederholten Ausführungen der Anwendung deterministisch ausgeführt wird. So wird dem Entwickler die Fehlersuche im Rahmen der Entwicklung erleichtert, da das Anwendungsverhalten reproduziert werden kann. Ermöglicht wird dies durch die unterliegende Frameworkebene, die intern nichtdeterministische Mechanismen nutzen kann, um die nebenläufigen Anteile der Anwendung zu koordinieren und beispielsweise eine hohe Performance der Anwendung in Abhängigkeit von sich ändernden Ausführungsumgebungen zu erreichen.

Das auf diesem Konzept aufbauende *Spawn & Merge Programmiermodell* vereinfacht die Entwicklung von Anwendungen, deren Anwendungslogik garantiert deterministisch ausgeführt wird. Für die Programmierung der Applikationsebene werden neue Synchronisationsprimitive eingeführt, die eine deterministische Synchronisation nebenläufig aus-

geführter Tasks einer Task-Hierarchie garantieren. Diese Synchronisationsprimitive bilden dabei die Schnittstelle der Anwendungslogik zu dem unterliegenden nichtdeterministischen Verhalten in der Frameworkebene (z.B. die Ausführungsreihenfolge nebenläufiger Tasks).

Um einen hohen Grad an Parallelität für Spawn & Merge Anwendungen zu ermöglichen, wird auf die Nutzung von geteiltem Arbeitsspeicher und auf das Sperren geteilter Ressourcen verzichtet. Stattdessen wird *Operational Transformation* zur automatischen und deterministischen Konfliktauflösung zwischen nebenläufig modifizierten Datenstrukturkopien verwendet. Um ein deterministisches Ergebnis zu erreichen, muss bei der Verwendung herkömmlicher Operational Transformation Systeme entweder eine deterministische Reihenfolge für die Zusammenführung modifizierter Datenstrukturen eingehalten oder auf komplexe Operational Transformation Systeme zurückgegriffen werden, die mit einem hohen Berechnungsmehraufwand sowie hohem Arbeitsspeicherverbrauch einhergehen. Die Einhaltung einer deterministischen Reihenfolge für die Zusammenführung führt dabei zu Wartezeiten, die wiederum zu einer Verringerung der Performance der Anwendung führen. Zur Kompensation dieses Umstandes wird ein modifiziertes Operational Transformation System entwickelt, das eine *Zusammenführung nebenläufiger Datenstrukturen in beliebiger Reihenfolge (First-Come-First-Serve)* ermöglicht, dabei allerdings auf die teuren Mechanismen komplexer Operational Transformation Systeme verzichtet. Dazu wird das Spawn & Merge Programmiermodell analysiert, um Eigenschaften zu identifizieren, die einem Operational Transformation System als zusätzliche Eingaben übergeben werden können. Diese zusätzlichen Informationen ermöglichen zusammen mit einer Anpassung eines einfachen Operational Transformation Systems eine deterministische Konfliktauflösung in FCFS Reihenfolge. So können die Wartezeiten reduziert werden, die durch die Mechanismen zur deterministischen Ausführung der Anwendung eingeführt wurden, was wiederum eine Verbesserung der Performance der Anwendung zur Folge hat.

Um die Untersuchung der Hypothese zu ermöglichen, wird das Spawn & Merge Programmiermodell für die Entwicklung von skalierenden verteilten Anwendungen, deren Applikationsebene deterministisch ausgeführt wird, konzipiert und prototypisch implementiert. Die Evaluation des Prototyps für verteiltes Spawn & Merge zeigt, dass der Prototyp die Erstellung skalierender deterministischer verteilter Anwendungen ermöglicht. Im Rahmen der Evaluation wird des Weiteren untersucht, unter welchen Umständen sich das Spawn & Merge Programmiermodell für die Umsetzung einer Anwendung eignet und unter welchen Umständen die internen Mechanismen des Spawn & Merge Programmiermodells im Verhältnis zur erreichten Performance einen zu großen Mehraufwand bedeuten. Dazu wird untersucht, wie hoch die Kosten der Mechanismen, die eine deterministische Ausführung der Applikationsebene ermöglichen, in Abhängigkeit von unterschiedlichen Eigenschaften einer Anwendung sind, um Entwicklern eine Entscheidungsgrundlage für

den Einsatz des Spawn & Merge Programmiermodells zu geben. Die Ergebnisse der Evaluation stützen dabei die Hypothese, da aufgezeigt wird, dass die Entwicklung skalierender deterministischer Verteilter Systeme möglich ist.

1.2 Struktur der Arbeit

Die Arbeit ist in acht Kapitel unterteilt. In Kapitel 2 wird als Grundlage für die weitere Arbeit der aktuelle Stand der Forschung beschrieben. Dabei wird insbesondere auf unterschiedliche Determinismusarten und verwandte Arbeiten aus dem Forschungsbereich der deterministischen Synchronisation nebenläufiger Systeme eingegangen. Anschließend wird in Kapitel 3 der Begriff des *Determinismus auf Applikationsebene* eingeführt und beschrieben, wie sich dieser von den etablierten Determinismusarten unterscheidet. In Kapitel 4 wird das Spawn & Merge Programmiermodell beschrieben, das speziell für die Entwicklung von Anwendungen konzipiert wurde, die auf der Applikationsebene deterministisch ausgeführt werden sollen. Das Spawn & Merge Programmiermodell setzt dabei auf die Verwendung von Operational Transformation Mechanismen, die in Kapitel 5 beschrieben werden, um eine deterministische Zusammenführung nebenläufig veränderter Datenstrukturkopien zu ermöglichen. Um die durch das Spawn & Merge Programmiermodell eingeführten Performancekosten für die deterministische Ausführung der Anwendungslogik zu reduzieren, wird in Kapitel 5 des Weiteren beschrieben, wie Operational Transformation Systeme angepasst werden können, um eine deterministische Zusammenführung unabhängig von der Zusammenführungsreihenfolge zu ermöglichen. In Kapitel 6 wird die Umsetzung des Spawn & Merge Programmiermodells für Verteilte Systeme beschrieben, wodurch die Entwicklung skalierender deterministischer Verteilter Systeme erleichtert wird. Zur Verifikation der Aussage, dass Spawn & Merge skalierende deterministische Verteilte Systeme ermöglicht wird anschließend in Kapitel 7 eine prototypische Implementierung des in Kapitel 6 konzipierten Systems evaluiert. Dabei wird zum einen anhand von Beispielszenarien die Skalierbarkeit von Anwendungen untersucht, die auf dem Spawn & Merge Framework aufbauen. Zum anderen wird untersucht, unter welchen Umständen und für welche Arten von Anwendungen sich das Spawn & Merge Framework eignet und unter welchen Umständen die Kosten für die deterministische Ausführung so hoch sind, dass auf alternative Programmiermodelle zurückgegriffen werden sollte. In Kapitel 8 werden die Ergebnisse der Arbeit zusammengefasst und ein Ausblick auf weitere mögliche Forschungsarbeiten im Kontext des Spawn & Merge Programmiermodells gegeben.

Kapitel 2

Stand der Forschung

In diesem Kapitel wird der aktuelle Stand der Forschung auf dem Gebiet der deterministischen Synchronisation Verteilter Systeme vorgestellt. Dabei werden zuerst Forschungsarbeiten beschrieben, die sich mit der Synchronisation nebenläufiger Systeme auseinandersetzen. Anschließend wird das Gebiet weiter eingegrenzt auf Forschungsarbeiten, die sich mit einer deterministischen Ausführung lokaler und verteilter Anwendungen beschäftigt haben.

2.1 Synchronisation nebenläufiger Systeme

Der Wechsel von Anwendungen, die in einem einzelnen Prozess ausgeführt werden, hin zu Anwendungen, die aus mehreren parallel ausgeführten Prozessen[1] bestehen, bringt viele Herausforderungen mit sich. Die meisten Herausforderungen beziehen sich auf die *Synchronisation* dieser Prozesse. Eine Synchronisation bezeichnet hierbei das Aushandeln und die Einigung auf eine Reihenfolge für die Durchführung bestimmter Aktionen (z.B. der Zugriff auf geteilte Ressourcen oder auf einen *kritischen Bereich* [33] des Programmcodes) zwischen mindestens zwei Prozessen [61].

Werden Prozesse ohne eine feste Reihenfolge ausgeführt, so bezeichnet man diese als nebenläufig [93]. Eine *nebenläufige Ausführung* zweier Prozesse unterscheidet sich dabei von einer *parallelen Ausführung* der Prozesse dadurch, dass die nebenläufigen Prozesse sowohl gleichzeitig, als auch zeitversetzt ausgeführt werden können. Parallel ausgeführte Prozesse laufen hingegen zum selben Zeitpunkt, beispielsweise auf unterschiedlichen Prozessorkernen. Die nebenläufige Ausführung ist dementsprechend eine allgemeinere Klassifikation für Ausführungen, die auch die parallele Ausführung beinhaltet.

Im weiteren Verlauf dieses Kapitels wird beschrieben, welche Mechanismen zur Synchronisation nebenläufiger Systeme es gibt und wie diese funktionieren. Dabei wird zuerst

[1] Prozesse stehen hierbei auch stellvertretend für weitere Möglichkeiten zur Parallelisierung (z.B. Threads).

das Sperren geteilter Ressourcen (Locking) als verbreiteter Ansatz beschrieben. Anschließend werden weitere komplexere Mechanismen für die Synchronisation von lokalen und verteilten Anwendungen beschrieben.

2.1.1 Sperren geteilter Ressourcen

Ein Standardansatz zur Synchronisation zwischen nebenläufigen Prozessen ist die Nutzung von *Locking-Mechanismen*, mit denen ein Entwickler verhindern kann, dass Prozesse gleichzeitig einen kritischen Codebereich ausführen (*mutual exclusion*). Dabei können Mechanismen dahingehend unterschieden werden, ob sie bei Anwendungen mit gemeinsam genutztem Speicher angewendet werden können, oder ob sie auch in einem Verteilten System ohne gemeinsam genutzten Speicher funktionieren.

Locking bei gemeinsam genutztem Arbeitsspeicher

Zu den Locking-Mechanismen für Anwendungen mit geteiltem Arbeitsspeicher gehören unter anderem Mutexe, Semaphoren und Monitore [92]. Ein *Mutex* [92] (kurz für „mutual exclusion“) ist eine Variable, die immer von genau einem Prozess gleichzeitig gelockt werden kann. Wird einem Prozess der angeforderte Lock über den Mutex zugesprochen, so steht dies für die Garantie, dass beispielsweise der Zugriff auf einen kritischen Codebereich nur diesem Prozess gestattet ist. Falls ein anderer Prozess einen bereits gelockten Mutex anfordert, so blockiert der Prozess solange, bis der Mutex wieder freigegeben wurde. Eine *Semaphore* [92] kann genutzt werden, um die Anzahl der gleichzeitig auf einen kritischen Codebereich zugreifenden Prozesse zu beschränken. Dazu erhält die Semaphore einen initialen Wert (welcher der maximalen Anzahl an Prozessen in der kritischen Coderegion entspricht) und verringert diesen Wert für jeden Prozess, dem Zugriff auf den Codebereich gewährt wird, um 1 (*down*). Ist der Wert der Semaphore bei der Anfrage eines Prozesses gleich 0, dann muss der anfragende Prozess darauf warten, dass ein Prozess innerhalb der kritischen Coderegion diese wieder verlässt und der Wert der Semaphore wieder um 1 erhöht wird (*up*). Ein *Monitor* [92] bezeichnet die Kapselung zusammengehöriger kritischer Bereiche in ein zusammenhängendes Konstrukt, das Funktionen und private Objekte enthalten kann. Bei einem Funktionsaufruf einer der Funktionen des Monitors wird sichergestellt, dass sich aktuell kein weiterer Prozess innerhalb des Monitors aufhält. Befindet sich aktuell ein anderer Prozess innerhalb des Monitors, so muss der neu hinzugekommene Prozess warten. Monitore verwenden intern Mutexe oder binäre Semaphoren[2] zur Sicherstellung, dass sich nur ein Prozess innerhalb des Monitors aufhält. Diese Mechanismen zur Synchronisation nebenläufiger Prozesse sind in vielen General Purpose Languages (GPLs) bereits standardmäßig implementiert (z.B. in Java [43], C# [47] und C++[54]).

[2]Eine *binäre Semaphore* ist eine Semaphore, die nur einen Prozess in einem kritischen Bereich zulässt.

Locking in Verteilten Systemen

Um den Zugriff auf geteilte Ressourcen in einem Verteilten System sperren zu können, können verteilte Locking-Services wie beispielsweise *Terracotta* [95] genutzt werden. Terracotta versteht sich als „Clustering Service“ und setzt sich als Vermittler zwischen Java-Applikationen und die diese Applikationen ausführenden Java Virtual Machines (JVM) und führt beispielsweise Threads auf unterschiedlichen Computern aus. Terracotta übernimmt dabei die Kommunikation über die Grenzen einzelner Computer hinweg, sodass die Java-Applikationen selbst nur einen Computer mit gemeinsam genutztem Arbeitsspeicher sehen. Dazu wird beispielsweise der Heap zwischen den einzelnen JVMs geteilt. Java-Applikationen, die mit Terracotta genutzt werden sollen, brauchen nur minimal angepasst werden. Zur Synchronisation des Zugriffs auf gemeinsam genutzte Ressourcen, können die von Java nativ bereitgestellten Mechanismen (wie z.B. Locking) genutzt werden, die von der Terracotta-Zwischenschicht zentral verwaltet werden.

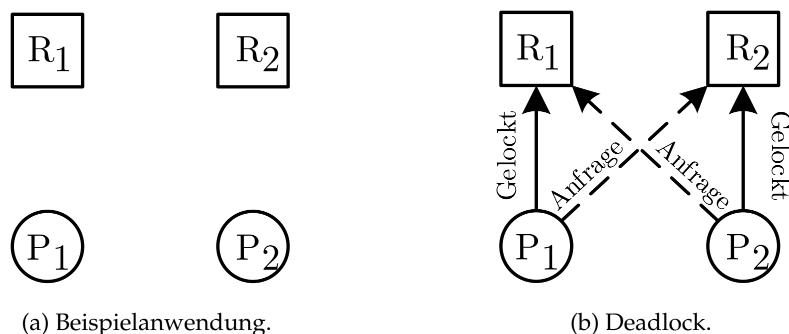
Race-Conditions

Bei Locking-Mechanismen wird demjenigen Prozess der Zugriff auf die geteilte Ressource (den kritischen Codebereich) gewährt, der zuerst den Lock anfordert. Diese Zeitpunkte, an denen die Prozesse Locks anfordern, sowie die sich daraus ergebende Reihenfolge für Zugriffe auf geteilte Ressourcen sind abhängig von Prozessortimings, Netzwerklatenzen und dem Scheduling von Threads durch das Betriebssystem. Auch eine Änderung der Ausführungsumgebung kann beispielsweise durch eine geänderte Anzahl an Prozessorkernen und des damit ermöglichten höheren Grades an Parallelität zu einer geänderten Lock-Reihenfolge führen. Durch diese sich ändernde Reihenfolge der Lock-Zugriffe kann sich das Programmverhalten zwischen zwei Ausführungen unterscheiden, sodass auch bei mehrfachem Testen einer Anwendung unklar sein kann, ob es noch eine weitere (selten vorkommende) Reihenfolge gibt, die zu einem Fehler in der Anwendung führt. Dieses Verhalten wird als *Race-Condition* bezeichnet.

Um eine korrekte Ausführung der Anwendung gewährleisten zu können, muss der Programmierer sicherstellen, dass jede mögliche Reihenfolge von Lock-Zugriffen zum gewünschten Ergebnis führt. Dieses Vorgehen wird dadurch erschwert, dass Locking-Mechanismen nur beschreiben können welcher Zustand verboten ist (z.B. zwei Prozesse befinden sich im gleichen kritischen Codebereich), nicht aber welche Zustände erlaubt sind. Das erschwert die Entwicklung komplexer Anwendungen für den Entwickler und sorgt dafür, dass der Entwicklungsprozess mit diesen Locking-Mechanismen aufwändig und fehleranfällig ist.

Deadlocks

Ein weiteres Problem, das sich bei der Nutzung dieser Locking-Mechanismen ergeben kann, ist ein sogenannter *Deadlock* (Verklemmung) [24]. Ein Deadlock beschreibt eine Situation, in der die Anwendung auf die Erfüllung von nicht mehr erfüllbaren Wartebedingungen wartet, und somit nicht mehr fertiggestellt werden kann. Ein Beispiel ist in Abbildung 2.1 dargestellt. Hier arbeiten zwei nebenläufige Prozesse P_1 und P_2, die beide wiederum auf die geteilten Ressourcen R_1 und R_2 zugreifen können, sobald sie einen Lock für die entsprechende Ressource zugesprochen bekommen haben (Abbildung 2.1a). Sollte nun Prozess P_1 einen Lock für die Ressourcen R_1 und R_2 nacheinander anfragen, während Prozess P_2 zur gleichen einen Lock für R_2 und R_1 nacheinander anfragt, so kann es passieren, dass Prozess P_1 einen Lock für R_1 zugesprochen bekommt, während Prozess P_2 einen Lock für R_2 bekommen hat. In diesem Falle würde Prozess P_1 fortan darauf warten, dass die Ressource R_2 freigegeben wird, damit er weiterarbeiten kann, während Prozess P_2 die Ressource R_2 festhält und darauf wartet, dass Prozess P_1 wiederum R_1 freigibt (siehe Abbildung 2.1b). Die Anwendung würde somit blockieren (verklemmen) und könnte nicht weiter ausgeführt werden.



(a) Beispielanwendung. (b) Deadlock.

Abbildung 2.1: Beispiel für einen Deadlock (Verklemmung).

Stresstests und Deadlockauflösung

Es gibt mehrere Forschungsarbeiten, die sich mit einer Lösung dieser Probleme beschäftigen.

Zum einen gibt es Werkzeuge wie *Racey* [49], die durch einen Stresstest versuchen so viele unterschiedliche Thread-Schedules für die Ausführung einer Anwendung wie möglich zu testen, um Race-Conditions aufzudecken. Racey kann dabei nur für multithreaded Anwendungen, nicht aber für eine verteilte Anwendung eingesetzt werden. Da es keine Möglichkeit gibt festzustellen, ob alle möglichen Reihenfolgen an Lock-Zuweisungen geprüft wurden, kann ein Stresstest keine Garantie darüber geben, ob es noch unentdeckte

Race-Conditions in der Anwendung gibt [49, 62]. Eine weitere Methode zur Bewältigung von Race-Conditions ist der *Schedule Specialization* Ansatz für multithreaded Anwendungen, der die hohe Anzahl möglicher Schedulings (unter anderem von Lock-Zugriffen) auf eine begrenzte Anzahl Schedules beschränkt, die wiederum in vertretbarer Zeit bezüglich ihrer Korrektheit untersucht werden können. Andere Arbeiten (die im späteren Verlaufe dieses Kapitels unter dem Begriff „deterministische Wiederholung" vorgestellt werden) konzentrieren sich hingegen auf die Möglichkeit eine aufgetretene Race-Condition deterministisch reproduzieren zu können, um das Beheben der Race-Condition zu erleichtern [22].

Zum anderen gibt es Arbeiten, die sich mit der Verhinderung, oder der Detektion und der Auflösung von Deadlocks in einem laufenden System beschäftigen. Ein möglicher Ansatz für die Verhinderung von Deadlocks ist dabei die Sicherstellung, dass es durch die Einhaltung bestimmter Regeln nicht zu einem Deadlock kommen kann [24]. Ansätze für die Detektion und Auflösung von Deadlocks setzen zum Beispiel da an, dass für einen Deadlock unter anderem eine zirkuläre Wartebedingung notwendig ist[3] [24]. Um eine zirkuläre Wartebedingung zu entdecken, kann ein gerichteter Graph aller angefragten Ressourcen nachgehalten werden, der gleichzeitig auf zirkuläre Bezüge geprüft wird [45]. Dies ist besonders in Verteilten Systemen sehr aufwändig, da es notwendig ist, dass ein Knoten den gesamten Graphen kennen muss. Des Weiteren muss der Graph bei jeder Lock-Anfrage aktualisiert werden. Als andere Möglichkeit, zirkuläre Wartebedingungen zu entdecken, wird das Versenden sogenannter *Probe*-Nachrichten entlang aktueller Wartebedingungen genutzt (zu dem Prozess, auf dessen Ressource aktuell gewartet wird) [45]. Diese können dann wiederum entlang ihrer Wartebedingungen diese Nachricht weiterleiten. Falls der initiale Prozess seine Probe-Nachricht wieder erhält, dann gibt es eine zirkuläre Wartebedingung. Um diesen Deadlock zu beheben, können entweder einzelne (an dem Deadlock beteiligte) Prozesse gestoppt und neu gestartet werden, oder einzelne Prozesse werden angewiesen bereits gelockte Ressourcen wieder freizugeben, bis dass der Deadlock aufgelöst ist. Diese Ansätze sind allerdings aufgrund ihres Mehraufwands für die normale Ausführung der Anwendung besonders im Hinblick auf Skalierbarkeit und Performance in den meisten Fällen nicht praktikabel. Somit bleiben die grundlegenden Probleme bestehen, sodass Locking-Mechanismen allein keine vollständige Lösung für die Synchronisationsprobleme komplexer Anwendungen darstellen. Daher werden im Folgenden Mechanismen zur Synchronisation nebenläufiger Systeme beschrieben, die versuchen die Probleme der primitiven Locking-Mechanismen zu lösen.

[3] Die Voraussetzungen für das Auftreten eines Deadlocks werden in Kapitel 4.6 genauer betrachtet.

2.1.2 Synchronisation von Systemen mit geteiltem Arbeitsspeicher

In diesem Kapitel werden verwandte Arbeiten beschrieben, die sich auf die Synchronisation von Systemen mit *gemeinsam genutztem Arbeitsspeicher* (*shared memory*) beziehen, und über die bereits in vielen Programmiersprachen integrierten Mechanismen hinausgehen. In diesen Systemen kann jeder nebenläufig ausgeführte Prozess auf den gemeinsam genutzten Arbeitsspeicher der Anwendung zugreifen. Die Prozesse können dabei von unterschiedlichen Kernen eines Prozessors oder von unterschiedlichen Prozessoren ausgeführt werden. In den folgenden Abschnitten werden die verwandten Arbeiten geordnet nach der Art des Ansatzes vorgestellt.

Frameworks für die Entwicklung paralleler Anwendungen

Frameworks und APIs für die Entwicklung paralleler Anwendungen sind Erweiterungen zu GPLs. Sie bieten Entwicklern Programmierschnittstellen und vorgegebene Anwendungsarchitekturen, auf deren Grundlage sich parallele Anwendungen einfacher umsetzen lassen. Dabei werden unterschiedliche Ansätze verfolgt.

Die *Open Multi-Processing (OpenMP)* API [75] bietet Entwicklern Compiler-Anweisungen, mit denen spezifiziert werden kann, welche Codebereiche parallel ausgeführt werden können. Der Entwickler kann durch Quelltext-Annotationen dem Compiler mitteilen, dass sich beispielsweise eine Schleife für eine parallele Ausführung der einzelnen Schleifendurchläufe eignet. Die Laufzeitumgebung kann dann bei der Ausführung der Anwendung die einzelnen Schleifendurchläufe in unterschiedlichen Threads ausführen. OpenMP kann dabei allerdings nicht überprüfen, ob die vom Entwickler spezifizierten Codebereiche tatsächlich nebenläufig oder parallel ausgeführt werden können, oder ob der Entwickler Seiteneffekte übersehen hat.

Die *Open Computing Language (OpenCL)* [59] ist ein Framework für die Entwicklung von Anwendungen, deren parallele Prozesse die unterschiedlichen Komponenten einer heterogenen Plattform nutzen können sollen. Dazu stellt OpenCL eine API bereit, die die Erstellung von Anwendungen ermöglicht, die sich sowohl auf Central Processing Units (CPUs) mit geteiltem Arbeitsspeicher, als auch auf Graphics Processing Units (GPUs) mit komplexen Arbeitsspeicherhierarchien ausführen lassen. *CUDA* [73] ist eine weitere Schnittstelle für die Entwicklung paralleler Anwendungen für die Ausführung auf GPUs des Herstellers *Nvidia*. CUDA ermöglicht dem Entwickler dazu den direkten Zugriff auf den Befehlssatz der GPU.

OpenHMPP (Hybrid Multicore Parallel Programming) [35] ist ein Standard für parallele Anwendungen, die auf heterogenen Plattformen ausgeführt werden können. Das Ziel von OpenHMPP ist es, die Nutzung von Hardwarebeschleunigung zu ermöglichen, ohne die Komplexität, die mit der Nutzung von GPUs einherkommt. OpenHMPP setzt da-

bei auf Compiler-Anweisungen (Quelltext-Annotationen) mit denen Funktionen für eine parallele Ausführung in Form von sogenannten *Codelets* markiert werden können. Codelet-Funktionen werden als Remote Procedure Call (RPC) ausgeführt und unterliegen dabei Einschränkungen, die eingehalten werden müssen. So darf die Funktion beispielsweise keinen Rückgabewert besitzen, keine `static` oder `volatile` Variablen beinhalten und selbst keine weiteren Codelets starten.

Die *Threading Buidling Blocks (TBB)* [83] sind ein Framework (in Form einer C++ Template Bibliothek), das von *Intel* für die Programmierung von parallelen Anwendungen für Mehrkernprozessoren entwickelt wurde. Eine auf TBB basierende Anwendung wird dabei in einzelne datenparallele Tasks aufgeteilt, die parallel ausgeführt werden können. Die Koordination der einzelnen Tasks (Erstellung, Synchronisation, Zerstörung) wird dabei automatisch durchgeführt, was es dem Entwickler ermöglicht, die Parallelität einer Anwendung auf eine einfache Art und Weise zu spezifizieren.

Laufzeitsysteme für parallele Anwendungen

Cilk [13] ist ein von *Intel* entwickeltes Laufzeitsystem (*runtime system*) für die Entwicklung von parallelen multithreaded Anwendungen. Bei Cilk kann ein Entwickler parallel ausführbare Codebereiche durch die Verwendung des Keywords *spawn* in Form von *Cilk Prozeduren* nebenläufig ausführen. Im Anschluss an die nebenläufige Ausführung geben die Prozeduren ihre berechneten Ergebnisse wieder an die Prozedur zurück, von der sie gestartet wurden. Das *sync* Keyword führt des Weiteren eine Barriere ein, die aussagt, dass die Ausführung erst dann fortgesetzt werden darf, wenn alle gestarteten Cilk Prozeduren fertiggestellt wurden. Die Cilk Laufzeitumgebung geht dabei davon aus, dass gestartete Prozeduren konfligierende Zugriffe auf geteilte Ressourcen vermeiden. Dafür gibt es allerdings keine Garantie, da dies vom Inhalt der gestarteten Prozeduren abhängig ist.

Nachrichtenaustausch

Die Idee, Prozesse durch das Austauschen von Nachrichten zu synchronisieren, geht auf das Paper *„Communicating Sequential Processes“* von Hoare zurück [51]. Aus dieser Idee haben sich mehrere, darauf aufbauende Ansätze entwickelt.

Ein aktuelles Beispiel ist die Programmiersprache *Go* (oder auch *golang*) [42] die von *Google* entwickelt wurde. In Go können Funktionen in Form von *Go-Routinen (goroutines)* nebenläufig ausgeführt werden. Go-Routinen sind dabei leichtgewichtige Prozesse und ermöglichen eine parallele Ausführung auf Computern mit geteiltem Arbeitsspeicher. Neben den Standardmechanismen für das Sperren geteilter Ressourcen unterstützt Go auch die Synchronisation von Go-Routinen durch den Austausch von Nachrichten über sogenannte *Kanäle (Channels)*. Dabei ist es einer Go-Routine sowohl möglich Nachrichten an mehrere

Channels zu senden, als auch Nachrichten von mehreren Channels zu empfangen. Go gibt dabei allerdings keine Garantien darüber, ob es in einer Anwendung zu Race-Conditions kommen kann, oder nicht.

Software Transactional Memory

Software Transactional Memory (STM) [57] ermöglicht es einem Entwickler, nebenläufige Zugriffe auf geteilte Ressourcen (in einem System mit geteiltem Arbeitsspeicher) in Transaktionen zusammenzufassen. Diese Transaktionen sind softwareseitig realisiert und prüfen nach ihrer Durchführung automatisch, ob alle initial (zu Beginn der Transaktion) gelesenen Werte unverändert geblieben sind. Falls sich Werte verändert haben, wird die Transaktion zurückgesetzt (Rollback) und neu ausgeführt. Dieses Vorgehen ermöglicht aufgrund seines optimistischen Verzichts auf ein Sperren der geteilten Ressourcen einen höheren Grad an Nebenläufigkeit und daraus folgend eine schnellere Ausführung der Anwendung.

Pfadausdrücke

Pfadausdrücke (path expressions) [20] nutzen eine Syntax, die der Idee von *Regulären Ausdrücken (regular expressions)* [96] ähnelt. Im Gegensatz zu Regulären Ausdrücken, die eine Menge aller erlaubten Zeichenketten definieren, beschreiben Pfadausdrücke eine Menge aller erlaubten parallelen Funktionsaufrufe für eine Menge von Funktionen. Sowohl Pfadausdrücke als auch Reguläre Ausdrücke haben trotz ihrer Mächtigkeit ein gemeinsames Problem. Die Ausdrücke sind sehr knapp und präzise gefasst, sodass sie bei komplexeren Problemen nur noch sehr schwer lesbar und verständlich sind. Ein weiteres Problem von Pfadausdrücken ist, dass sie zwar die nebenläufige Ausführung von Funktionen einschränken, aber für gewöhnlich dennoch eine große Menge von unterschiedlichen möglichen Ausführungspfaden erlauben. Diese unterschiedlichen Ausführungspfade können wiederum aufgrund von Race-Conditions zu unterschiedlichen Ergebnissen der Anwendung führen.

Coordination Languages

Coordination Languages [41] können genutzt werden, um Protokolle für die Kommunikation und Interaktion (Koordination) zwischen nebenläufigen Prozessen zu beschreiben. Ein Beispiel ist die Programmiersprache *Linda* [1]. Linda erlaubt die Erweiterung einer sequenziellen Programmiersprache um Befehle, die eine Kommunikation zwischen einzelnen Prozessen ermöglicht. Diese Kommunikation wird über *Tupel* realisiert, die in einem *Tuplespace* in einem speziell dafür vorbereiteten, gemeinsam genutzten Arbeitsspeicherbereich gespeichert werden. Dies erlaubt eine klare Trennung von der Prozesskoordination und den Prozessberechnungen. Der Tuplespace ermöglicht Linda dabei die Verbindung von

Prozessen über Datenstrukturen ohne das Austauschen von Nachrichten. Coordination Languages stellen allerdings keine Ausführung der Anwendungen ohne Race-Conditions sicher. Des Weiteren ist die über die Tupel realisierte Kommunikation zwischen den Prozessen langsamer als die Synchronisation über Nachrichtenaustausch (z.B. beim Message Passing Interface (MPI) [69]).

2.1.3 Synchronisation Verteilter Systeme

Ein Verteiltes System[4] zeichnet sich dadurch aus, dass das System aus einer Ansammlung von eigenständigen Computern (oder Rechenknoten) besteht, die für den Nutzer wie ein einzelnes zusammengehöriges System wirken [93]. In diesem Kapitel werden Arbeiten beschrieben, die sich mit der Synchronisation nebenläufiger Verteilter Systeme (vornehmlich Multi Computer Systeme) befassen und über verteilte Locking-Mechanismen (siehe Kapitel 2.1.1) hinausgehen.

Nachrichtenaustausch

Nachrichtenaustausch kann nicht nur für die Synchronisation zwischen nebenläufigen Prozessen auf einem Computer mit gemeinsam genutztem Arbeitsspeicher genutzt werden, sondern eignet sich insbesondere auch für die Synchronisation zwischen nebenläufigen Prozessen in einem Multi Computer System. Dabei liegt ebenfalls die Idee zugrunde, dass jeder Prozess Nachrichten an andere Prozesse im selben Verteilten System senden und Nachrichten von anderen Prozessen empfangen kann. Für Verteilte Systeme gibt es zum einen Ansätze, die dem Entwickler durch die Bereitstellung einer entsprechenden Schnittstelle erlauben Nachrichten zwischen einzelnen Prozessen im Verteilten System auszutauschen. Zum anderen gibt es Ansätze, die die gesamte Synchronisation der verteilten Prozesse übernehmen.

Ein Beispiel für die Bereitstellung einer Schnittstelle zum Austausch von Nachrichten zwischen Prozessen ist das *Message Passing Interface (MPI)* [69]. Die Prozesse können dabei auf unterschiedlichen Computern ausgeführt werden und werden über eindeutige IDs identifiziert und adressiert. Das Aufbauen der Verbindung der Rechenknoten untereinander, sowie die Sicherstellung der korrekten Übertragung von Nachrichten wird hierbei von MPI übernommen. Da das Senden von Nachrichten von der Ausführungsgeschwindigkeit der einzelnen Prozesse abhängt (und diese wiederum unter anderem von der CPU des entsprechenden Rechenkerns), können bei einer rein auf MPI basierenden Anwendung Race-Conditions nicht ausgeschlossen werden.

[4] Im Rahmen dieser Arbeit beschreiben *Verteilte Systeme* insbesondere Multi Computer Systeme. Synchronisationsmechanismen für Verteilte Systeme mit gemeinsam genutztem Arbeitsspeicher fallen unter die in Kapitel 2.1.2 vorgestellten Ansätze.

Joyce [46] ist ein Beispiel für eine Programmiersprache für nebenläufige Programme, die Nachrichtenaustausch zur Synchronisation parallel laufender Prozesse nutzt. Joyce basiert auf einer Teilmenge von *Pascal* [53] und bietet dem Entwickler die Möglichkeit in einer Anwendung Parallelität in Form von gestarteten *Agenten* zu realisieren. Agenten führen dabei eine Prozedur nebenläufig aus und können über Kanäle (Channels) Nachrichten zur Synchronisation untereinander austauschen. Agenten können dabei auch weitere (ihnen untergeordnete) Agenten starten (*Unteragenten*). Bevor die Ausführung eines Agenten als fertiggestellt gelten kann, müssen alle seine Unteragenten (und rekursiv deren Unteragenten) fertiggestellt worden sein.

Prozesskalküle

Prozesskalküle (Process Calculi) [80] können unter anderem dazu genutzt werden, nebenläufige (sowohl lokale als auch Verteilte) Systeme zu modellieren und anhand dieser Modelle Schlussfolgerungen über das System und dessen Eigenschaften herzuleiten. Sie bieten eine high-level Beschreibung von nebenläufigen Prozessen mit besonderem Fokus auf Kommunikation und Synchronisierung der Prozesse. Diese Beschreibung basiert auf einer begrenzten Anzahl an Befehlen und Operatoren. Des Weiteren können Systembeschreibungen, die auf Prozesskalkülen basieren, durch die Anwendung algebraischer Umformungen verändert und analysiert werden. Auf diesem Wege kann beispielsweise die Äquivalenz zwischen nebenläufigen Prozessen gezeigt und die Korrektheit einer nebenläufigen Anwendung verifiziert werden.

Ein bereits genanntes Beispiel für ein Prozesskalkül sind die *Communicating Sequential Processes (CSP)* [51], die als Grundlage für viele heutige Mechanismen dienen, die auf Nachrichtenaustausch basieren. Der *Calculus of Communicating Systems (CCS)* [67] modelliert die Kommunikation zwischen genau zwei Prozessen. Dabei kann CCS sowohl die erlaubte Parallelität der Prozesse, als auch erlaubte Aktionen oder Einschränkungen der erlaubten Aktionen beschreiben. Das *π-Kalkül (π-calculus)* [87] ist eine Weiterentwicklung des CCS und erlaubt die Modellierung der Kommunikation zwischen mehreren nebenläufigen Prozessen. Dabei ermöglicht die Nutzung benannter Kanäle auch dann eine Beschreibung des Systems, wenn sich beispielsweise die zugrundeliegende Netzwerkkonfiguration zur Laufzeit ändert.

Während sich Prozesskalküle gut für die Verifikation von Protokollen (z.B. kryptographische Protokolle) eignen, ist die Abbildung einer komplexen verteilten Anwendung (ähnlich den Pfadausdrücken) schwieriger zu realisieren. Hierbei kommt erschwerend hinzu, dass es für einen Entwickler nicht trivial möglich ist eine Implementierung zu erreichen, die in jeglicher Hinsicht äquivalent zum modellierten Prozesskalkül ist.

Programmiersprache für verteilte parallele Anwendungen

Julia [72] ist ein Beispiel für eine dynamisch typisierte Programmiersprache für verteilte parallele Anwendungen. Anwendungen, die in Julia entwickelt wurden, können für eine größere Flexibilität auch Bibliotheken einbinden und ansprechen, die in anderen Programmiersprachen entwickelt wurden (z.B. Python, C und C++). Zur Synchronisation zwischen parallel laufenden Prozessen bietet Julia *Remote References* und *Remote Calls* an. Remote Calls sind Funktionsaufrufe, die parallel ausgeführt werden. Beim aufrufenden Prozess geben Remote Calls sofort eine Remote Reference auf das Antwortobjekt zurück. Dabei handelt es sich um ein *Future*-Objekt, das erst bei Fertigstellung des Remote Calls das Ergebnis enthalten wird. Der aufrufende Prozess kann nun auf den Inhalt des Future Remote Objektes warten oder seine Ausführung fortsetzen. Die zweite Art von Remote References sind *RemoteChannels*, die zur Kommunikation zwischen Prozessen genutzt werden können. Die Verwendung von Julia gibt keine Garantie, dass keine Race-Conditions auftreten können.

Frameworks für die Entwicklung verteilter paralleler Anwendungen

Apache Spark [103] ist ein Beispiel für ein Framework für die Entwicklung von verteilten parallelen Anwendungen, die auf einem Cluster ausgeführt werden sollen. Spark ermöglicht dabei im Speziellen die Erstellung von datenparallelen Anwendungen. Die von Spark bereitgestellte Schnittstelle konzentriert sich auf das *resilient distributed dataset (RDD)*, eine nicht veränderbare Datenstruktur, die über ein Rechencluster verteilt ist. Diese RDDs dienen dabei der Anwendung als Eingabedaten, die parallel ausgeführte Operationen auf den RDDs starten können (wie beispielsweise *Map*, *Filter* oder *Reduce*), die wiederum neue RDDs erstellen. Zur Synchronisation zwischen parallel ausgeführten Prozessen können RDDs in Form von *broadcast variables* an andere Prozesse weitergegeben werden. Für die Ausführung einer Spark Anwendung wird ein Cluster Manager, sowie ein verteiltes Dateisystem benötigt. Das Spark Framework gibt dabei keine Garantie, dass keine Race-Conditions auftreten können.

2.2 Determinismus und Reproduzierbarkeit

Die bis hierhin vorgestellten Arbeiten haben sich mit der Synchronisation lokaler und Verteilter Systeme beschäftigt. Eine Eigenschaft, die diese Ansätze gemeinsam haben, ist, dass bei ihnen (standardmäßig oder bei einer fehlerhaften Nutzung) Race-Conditions auftreten können. Entwickler müssen dementsprechend alle möglichen Ausführungspfade in Betracht ziehen, um ein gleiches Verhalten der Anwendung in Anbetracht mehrerer Ausführung erreichen zu können. Dies wird dadurch zusätzlich erschwert, dass die Anzahl

möglicher Ausführungspfade in komplexen Anwendungen exponentiell steigen kann. Eine immer gleiche Ausführung von Anwendungen hat allerdings Vorteile, sowohl für den Entwickler einer Anwendung, der eine fehlerfreie Ausführung der Anwendung sicherstellen und aufgetretene Fehler reproduzieren kann, als auch für den Anwender, der bei jeder Ausführung dasselbe Ergebnis bekommt (unabhängig von nicht vorhersehbaren Prozessortimings und Nachrichtenlatenzen).

In diesem Kapitel werden Arbeiten vorgestellt, die eine deterministische Ausführung lokaler und verteilter nebenläufiger Anwendungen ermöglichen. Die Ausführung einer Anwendung wird hier als *deterministisch* verstanden, wenn sie für dieselben Eingaben bei jeder Ausführung dieselben Ausgaben generiert. Dabei gibt es unterschiedliche Determinismusarten und Granularitäten, die in der Forschung unterschieden werden. Diese werden in Kapitel 2.2.1 vorgestellt. Der Begriff der *Konsistenz* ist mit dem Begriff des Determinismus verwandt, beschreibt allerdings einen leicht anderen Sachverhalt. Im Gegensatz zu den identischen Ergebnissen einer deterministischen Ausführung gibt eine konsistentes Verhalten an, dass sich (beispielsweise bei einer verteilten Anwendung) alle Prozesse auf eine *gemeinsame Wahrheit* einigen. Diese kann sich allerdings von Ausführung zu Ausführung unterscheiden. Das Gegenstück zur deterministischen Ausführung ist die nichtdeterministische Ausführung einer Anwendung. Ist eine Anwendung *nichtdeterministisch*, so kann sich das Ergebnis der Anwendung auch bei derselben Eingabe bei unterschiedlichen Ausführungen unterscheiden. Im Folgenden wird des Weiteren der Begriff *Reproduzierbarkeit* verwendet. Reproduzierbarkeit beschreibt im Rahmen dieser Arbeit die Möglichkeit den Ausführungspfad einer Anwendungsausführung wiederholen zu können. Die reproduzierte Ausführung kann dementsprechend als deterministisch angesehen werden, selbst wenn die initiale Ausführung der Anwendung nichtdeterministisch war (*deterministische Wiederholung*).

Die Idee, dass nebenläufige Anwendungen standardmäßig deterministisch sein sollen, und dass nichtdeterministisches Verhalten nur explizit in das System eingeführt werden soll wenn es benötigt wird, ist nicht neu und wurde bereits von Brocchino et al. vorgeschlagen [14]. Als Vorteile einer deterministischen Ausführung werden dabei unter anderem die Abwesenheit von Race-Conditions, die Vereinfachung der Entwicklung und Fehlerbehebung sowie ein größeres Vertrauen in die Tests einer Software genannt. Auch Xiao et al. [100] weisen auf die Probleme hin, die sich durch nichtdeterministische Teilbereiche einer Anwendung (hier im speziellen bezogen auf *MapReduce* Anwendungen) ergeben können. Dazu beschreiben sie, wie die Nebenläufigkeit der *Map*-Prozesse zu einer unterschiedlichen Reihenfolge, in der die *Map*-Ergebnisse an die *Reducer*-Prozesse übergeben werden, führen können. Dies kann wiederum zu Folgefehlern (oder weiterem nichtdeterministischen Verhalten) führen, wenn die *Reducer*-Prozesse nicht kommutativ sind (d.h. keine Eingabe in unterschiedlicher Reihenfolge erlauben).

Es gibt allerdings auch Arbeiten, welche die Lösung der Probleme nebenläufiger Anwendungen nicht in einer deterministischen Ausführung sehen. Yang et al. [101, 102] vertreten, auf multithreaded Anwendungen bezogen, den Standpunkt, dass Determinismus weder ausreichend, noch notwendig für eine verlässliche Ausführung dieser Anwendungen sei. Sie begründen dies damit, dass Nichtdeterminismus innerhalb einer Anwendung nur ein kleiner Teil des Problems sei, das die Entwicklung verlässlicher Anwendungen erschwert, und Determinismus somit nur die Lösung für einen Teil des Problems sei. So könne eine deterministische Anwendung beispielsweise bei einer Änderung der Eingabe abstürzen und wäre somit zwar deterministisch, nicht aber verlässlich. Die Zielsetzung ihrer Arbeit unterscheidet sich dementsprechend von der Zielsetzung anderer Arbeiten, für die ein Fehler bei einer bestimmten Eingabe nebensächlich ist, solange dieser Fehler bei jeder Ausführung auftritt.

Konkret sehen Yang et al. das Problem in der Anzahl möglicher Schedules für die Zuordnung von Prozessen auf Prozessorkerne und die damit einhergehenden, möglichen Verzahnungen der Prozesskommunikationen. So kann eine deterministische Anwendung für unterschiedliche Eingaben auch unterschiedliche (dennoch korrekte) Schedules erzeugen, sofern dies nicht durch Synchronisationsmechanismen verhindert wird. Nach Ansicht von Yang et al. beeinträchtigt dies die Stabilität und die Robustheit der Anwendung, da eine Aussage über die Korrektheit der Anwendung für alle möglichen Eingaben und Schedules erschwert wird. In einer nichtdeterministischen Anwendung könne (nach Yang et al.) hingegen sichergestellt werden, dass alle Eingaben auf eine geringe Anzahl an Schedules abgebildet werden. Diese wenigen Schedules könnten anschließend durch ausführliches Testen auf ihre Verlässlichkeit (in Bezug auf die fehlerfreie Ausführung der Anwendung) überprüft werden (*Stable Multithreading*).

Der Nichtdeterminismus der Anwendung im von Yang et al. vorgeschlagenen Ansatz ergibt sich daraus, dass dieselbe Eingabe bei mehrfacher Ausführung der Anwendung auf unterschiedliche (der vorgegebenen) Schedules abgebildet werden kann. Dieser Ansatz schließt dabei eine deterministische Umsetzung der Zuordnung von Eingaben zu Schedules nicht aus. Insgesamt kann dieser Ansatz als eine abgeschwächte Determinismusart verstanden werden, da nicht jeder Aspekt der Anwendung deterministisch ausgeführt wird (durch die verschiedenen zugelassenen Schedules), aber am Ende ein deterministisches Ergebnis erreicht wird. Welche unterschiedlichen Determinismusarten in der Literatur unterschieden werden können, wird im Folgenden beschrieben.

2.2.1 Determinismusarten

In der Literatur werden unterschiedliche Determinismusarten und Granularitäten unterschieden. Mit Granularität ist dabei in diesem Kontext gemeint, wie strikt die deterministischen Ausführung für die einzelnen Komponenten einer nebenläufigen Anwendung

einzuhalten ist, oder ob sich der Determinismus nur auf Teilbereiche der Anwendung bezieht.

Voller Determinismus

Gilt für eine Anwendung, dass bei jeder Ausführung (mit gleicher Eingabe) der Ausführungspfad exakt gleich verläuft, so spricht man von einer *voll-deterministischen* Ausführung (manchmal auch *starker Determinismus (strong determinism)*). Die Zusicherung, dass jeder interne Zwischenstand der Anwendung während der Ausführung derselbe ist, ermöglicht dabei sowohl ein deterministisches Ergebnis, als auch die Reproduzierbarkeit des Ausführungspfades. Ein deterministischer Ausführungspfad bedeutet dabei, dass eine voll-deterministische nebenläufige Anwendung bei jeder Ausführung denselben statischen Schedule für die Prozesse nutzen muss. Dadurch geht ein großer Anteil der möglichen Parallelität der Anwendung verloren, was zu Performanceverlusten führt.

Deterministische Wiederholung

Ein anderer Ansatz ist die *deterministische Wiederholung (deterministic replay)* von Anwendungsausführungen [22, 65]. Hierbei ist die Zielsetzung, die Ausführung einer Anwendung voll-deterministisch reproduzieren zu können. Zu diesem Zweck werden zwei unterschiedliche Modi der Ausführung unterschieden: *Record* und *Replay*. Im Record-Modus wird die Anwendung normal ausgeführt. Dabei werden alle Aktionen innerhalb der Anwendung (sowohl ausgeführte Befehle innerhalb eines Prozesses als auch die Synchronisationen zwischen Prozessen) aufgezeichnet und gespeichert. Tritt ein Fehler auf, oder möchte sich ein Entwickler die letzte Ausführung der Anwendung aus einem anderen Grund noch einmal genauer anschauen, dann kann er die Anwendung im Replay-Modus starten. Dabei wird der Anwendung die Aufzeichnung einer früheren Ausführung übergeben. Diese wird nun voll-deterministisch reproduziert, sodass der Entwickler beispielsweise bei der Fehlersuche unterstützt wird. Zu beachten ist hierbei, dass die Ausführung im Record-Modus nicht zwingend deterministisch ist. So können bei der normalen Ausführung der Anwendung Race-Conditions oder Deadlocks auftreten (die dann allerdings voll-deterministisch nachvollzogen werden können). Ein weiterer zu beachtender Aspekt der deterministischen Wiederholung ist, dass sowohl die Aufzeichnung aller Aktionen aller Prozesse im Record-Modus, als auch das erneute Einspielen einer aufgezeichneten Aktionsreihenfolge, mit einem entsprechenden Geschwindigkeitsverlust der Anwendung einhergehen.

Sende-Determinismus

Die Klassifikation einer Anwendung als *sende-deterministisch (send determinism)* kommt ursprünglich aus dem *High-Performance-Computing (HPC)* Bereich und bezieht sich auf Anwendungen, die Nachrichtenaustausch (z.B. über MPI) zur Synchronisation einsetzen [21]. Die Arbeit ist dabei motiviert durch das Problem abstürzender Prozesse (die auf MPI-Knoten ausgeführt werden) und dem damit einhergehenden Mehraufwand für die Ausführung der gesamten Anwendung. So ist beispielsweise das Aufzeichnen aller Nachrichten des Systems (um einen abgestürzten Prozess neu starten und auf den letzten Stand vor seinem Absturz bringen zu können) ebenso teuer wie die Möglichkeit, die gesamte Anwendung im Fehlerfall neu starten zu müssen.

Aus diesem Grund wird die Entwicklung sende-deterministischer Anwendungen empfohlen. In einer sende-deterministischen Anwendung wird davon ausgegangen, dass jeder nebenläufig ausgeführte Prozess für sich genommen deterministisch ist, und von außen nur durch eingehende Nachrichten beeinflusst werden kann. Das bedeutet insbesondere, dass alle Nachrichten von diesem Prozess ebenfalls deterministisch versendet werden (Sende-Determinismus). Um die Anforderung zu erfüllen, dass ein Prozess selbst immer deterministisch ist, muss dieser auch bei Nachrichten, die in einer nichtdeterministischen Reihenfolge eingehen, ein deterministisches Verhalten sicherstellen (z.B. durch eine Sortierung (*Reordering*) der Nachrichten). Dies muss vom Entwickler sichergestellt werden, um eine deterministische Ausführung der gesamten Anwendung erreichen zu können.

Der Vorteil, der sich aus einer sende-deterministischen Anwendung für den Entwickler und den Nutzer ergibt, ist, dass abgestürzte Prozesse neu gestartet werden können, ohne dass entweder alle Nachrichten aufgezeichnet werden müssen, oder das gesamte System neu gestartet werden muss. Um dies zu erreichen, kann der abgestürzte Prozess zusammen mit den Prozessen, die ihm Nachrichten gesendet haben, neu gestartet werden. In diesem Fall werden die Nachrichten deterministisch neu erstellt und versendet, sodass der abgestürzte Prozess wieder auf den letzten Stand gebracht werden und seine Ausführung fortsetzen kann. Dabei ist zu beachten, dass dies zu einem „Domino-Effekt“ führen kann, da die Prozesse, welche die verlorenen Nachrichten neu generieren sollen, möglicherweise selbst wieder Abhängigkeiten zu anderen Prozessen haben, die ebenfalls neu gestartet werden müssten. Im Worst-Case kann es somit passieren, dass dennoch die gesamte Anwendung neu gestartet wird.

Schwacher Determinismus

Schwacher Determinismus (weak determinism) garantiert die deterministische Ausführung einer Anwendung unter der Voraussetzung, dass diese keine *Daten-Race-Conditions (data-race)* beinhaltet [88]. Eine Daten-Race-Condition ist dabei eine Race-Condition, die sich

auf Lese- und Schreib-Operationen auf geteilten Datenstrukturen (bei gemeinsam genutztem Arbeitsspeicher) bezieht. In diesem Falle reduziert sich die deterministische Ausführung der Anwendung auf die deterministische Durchführung der Synchronisation zwischen den nebenläufigen Prozessen. Hierbei ist anzumerken, dass es für einen Entwickler nicht trivial ist, eine sichere Aussage darüber zu treffen, ob seine Anwendung frei von Daten-Race-Conditions ist oder nicht.

„Don't Care" Nichtdeterminismus

Das von Pingali et al. eingeführte Konzept des *„Don't Care" Nichtdeterminismus (don't care non-determinism)* erlaubt einer Anwendung das Durchlaufen nichtdeterministischer (aber valider) Zwischenstände während der Ausführung, sofern das Ergebnis der Ausführung deterministisch bleibt [81]. Somit wird für die Anwendung ein höherer Grad an Parallelität ermöglicht. Diese gewonnene Parallelität kommt dabei allerdings auf Kosten der Reproduzierbarkeit des Ausführungspfades, der sich bei mehrfacher Ausführung der Anwendung unterscheiden kann. Dies erschwert wiederum die Fehlersuche und das Nachvollziehen eines auffälligen Verhaltens der Anwendung.

Externer und interner Determinismus

In manchen Fällen wird in den verwandten Arbeiten über „externen" und „internen" Determinismus geschrieben. Unter dem Begriff *externer Determinismus (external determinism)* wird die Eigenschaft einer Anwendung verstanden, dass diese für dieselbe Eingabe immer dieselbe Ausgabe liefert [88]. Eine Aussage über die internen Zustände, die die Anwendung während wiederholten Ausführungen durchläuft, wird hierbei nicht getroffen. Diese können sich dementsprechend zwischen mehreren Ausführungen unterscheiden. Der Begriff *interner Determinismus (interal determinism)* beschreibt hingegen die Eigenschaft einer Anwendung, dass sich bei einer wiederholten Ausführung auch die internen Zustände der Anwendung deterministisch wiederholen [88]. Beide Begriffe gehen dabei nicht weiter auf die feineren Determinismuseigenschaften der Anwendungen ein.

Es gibt viele Forschungsarbeiten, die sich mit der deterministischen Ausführung nebenläufiger Anwendungen befassen. In diesem Kapitel werden sowohl Ansätze für lokale Anwendungen mit geteiltem Arbeitsspeicher, als auch für Verteilte Systeme beschrieben, die eine deterministische Ausführung der Anwendungen, im Sinne der hier beschriebenen Determinismusarten, ermöglichen. Im Anschluss an die Vorstellung dieser Arbeiten werden in Kapitel 2.2.4 die wichtigsten Eigenschaften dieser Systeme zusammengefasst.

2.2.2 Determinismus in Systemen mit geteiltem Arbeitsspeicher

In diesem Kapitel werden Ansätze für die deterministische Ausführung lokaler Anwendungen beschrieben. Dabei wird auch darauf eingegangen, ob sich diese Ansätze theoretisch auf ein Verteiltes System übertragen lassen, oder ob eine Verteilung nicht trivial möglich ist[5].

Deterministische Ausführung beliebiger Programme

Die hier beschriebenen Arbeiten ermöglichen die deterministische Ausführung beliebiger Anwendungen, ohne dass eine Anpassung notwendig ist. Dazu wird in den meisten Fällen durch die Laufzeitumgebung sichergestellt, dass alle internen Vorgänge (sowie die Kommunikation zwischen nebenläufigen Prozessen) deterministisch durchgeführt werden.

CoreDet [10] ist ein Compiler in Verbindung mit einer Laufzeitumgebung, welche zusammen die deterministische Ausführung von beliebigen multithreaded C und C++ Anwendungen ermöglichen. In CoreDet werden alle Threads nebenläufig ausgeführt, solange diese nicht miteinander kommunizieren. Sobald Threads miteinander kommunizieren werden diese Threads mit Hilfe eines deterministischen Schedulers serialisiert in einer deterministischen Reihenfolge ausgeführt. So wird sichergestellt, dass die Kommunikation zwischen Threads immer in einer deterministischen Reihenfolge geschieht.

dThreads [64] ist eine Laufzeitumgebung, die es ermöglicht beliebige multithreaded Anwendungen deterministisch auszuführen. dThreads nutzt dabei Prozesse, um Threads zu realisieren. Jeder gestartete Prozess besitzt eine private und eine mit anderen Prozessen geteilte Sicht auf den gemeinsam genutzten Arbeitsspeicher. Daher kann jeder Prozess, unabhängig von den weiteren nebenläufig ausgeführten Prozessen, den privaten (und somit für ihn nutzbaren) Bereich des geteilten Arbeitsspeichers manipulieren. Änderungen am gemeinsam genutzten Arbeitsspeicher werden von der Laufzeitumgebung in eine deterministische Reihenfolge sortiert und an deterministischen Synchronisationspunkten angewendet.

Grace [12] ist eine Laufzeitumgebung, die Fork-Join-basierte Anwendungen deterministisch ausführen kann. Dazu löst Grace die Probleme, die sich durch die nebenläufige Ausführung ergeben (z.B. Deadlocks, Race-Conditions und das nichtdeterministische Scheduling von Threads). Um Deadlocks zu verhindern, werden alle Locking-Anfragen in `NOOPs` (no-operations) umgewandelt. Race-Conditions werden umgangen, indem alle Änderungen an Bereichen des gemeinsam genutzten Arbeitsspeichers in deterministischer Reihenfolge angewendet werden. Des Weiteren werden Threads in der Reihenfolge ausgeführt, in der sie im jeweiligen Funktionskörper definiert wurden, um einen deterministi-

[5] Hier soll in erster Linie eine Aussage darüber getroffen werden, ob eine Verteilung offensichtlich und ohne viel Aufwand möglich zu realisieren ist.

schen Schedule zu erreichen. Diese Serialisierung der Threads geht dabei mit einem großen Performanceverlust einher, da die Parallelität der Anwendung stark eingeschränkt wird.

Da in den ausgeführten Anwendungen potenziell Race-Conditions vorhanden sein können (da es sich um beliebige Anwendungen handelt), können diese auch bei der deterministischen Ausführung auftreten (abhängig vom gewählten durchgesetzten Schedule). Die Fehler, die durch die Race-Condition entstehen, können hier allerdings nachvollzogen und reproduziert werden, da sie nun aufgrund der statischen Schedules deterministisch auftreten würden. Die beschriebenen Verfahren setzen alle einen gemeinsam genutzten Arbeitsspeicher voraus und lassen sich somit nicht direkt auf ein Verteiltes System ohne geteilten Arbeitsspeicher abbilden.

Stable Multithreading

Parrot [26] ist eine Laufzeitumgebung, welche die Anzahl möglicher Schedulings für nebenläufige Threads durch ein stabiles Scheduling reduziert. Das Ziel ist dabei allerdings nicht die deterministische Ausführung der Anwendung, sondern die Möglichkeit, die Ausführung der Anwendung besser zu verstehen. Parrot setzt dabei auf den Ansatz des *stabilen Multithreadings (Stable Multithreading)*, der bereits zu Beginn dieses Kapitels als Gegenentwurf zur deterministischen Ausführung einer Anwendung genannt wurde, um eine *verlässliche* Ausführung zu erreichen. Als verlässlich gilt hierbei eine Anwendung, bei der eine geänderte Eingabe nicht zu einem Fehlverhalten der Anwendung führen kann. Dazu werden die Threads der Anwendung, unabhängig von der Eingabe, auf eine limitierte Anzahl von möglichen Schedules abgebildet, die von der Laufzeitumgebung für eine verlässliche Ausführung als „valide“ angesehen werden. Im Falle von Parrot geschieht das Scheduling dabei nach einem *Round-Robin* Verfahren [92]. Um eine bessere Performance zu erreichen, kann der Entwickler in der Anwendung Hinweise für den Scheduler geben, um das Scheduling zu beeinflussen. *Peregrine* [27] und *Tern* [28] sind weitere ähnliche Ansätze, die auf stabilem Multithreading aufbauen, und sich in erster Linie durch die Art des Schedulings von Parrot unterscheiden. Da diese Ansätze von Haus aus nichtdeterministisch sein können, eignen sie sich nicht für die Entwicklung deterministischer Verteilter Systeme und werden dahingehend nicht weiter betrachtet.

Frameworks für deterministische Anwendungen

Frameworks für deterministische Anwendungen stellen dem Programmierer Schnittstellen bereit, die die Entwicklung entsprechender Anwendungen erleichtern. Ein Beispiel ist das Software-Framework *Kendo* [74], das die Entwicklung nebenläufiger lokaler Anwendungen ermöglicht, die deterministisch ausgeführt werden. Dabei wird das Konzept des schwachen Determinismus (*weak determinism*) angewandt, sodass sich die Determinis-

musgarantie nur auf diejenigen Anwendungen bezieht, die keine Daten-Race-Conditions beinhalten. Für alle anderen Anwendungen, die auf Kendo basieren, ergibt sich die deterministische Ausführung durch eine deterministische Reihenfolge der Synchronisationsoperationen, die vom Framework sichergestellt wird (deterministisches Scheduling). Zur Verteilung von Kendo wäre es notwendig, dass die Anwendung (unabhängig vom Scheduling) keine Daten-Race-Conditions beinhaltet, was in einem Verteilten System nicht trivial erreichbar ist. Des Weiteren müsste das Framework eine deterministische Reihenfolge der Synchronisationsoperationen über die das gesamte Verteilte System hinweg sicherstellen, was ein neues Problem darstellt, das vom Framework (in einer effizienten Art und Weise) gelöst werden müsste. Eine Portierung von Kendo auf ein Verteiltes System ist somit nicht ohne weiteres möglich.

Deterministische Wiederholung

Die hier beschriebenen Ansätze ermöglichen die deterministische Wiederholung (das Reproduzieren) einer Anwendungsausführung (deterministic replay, siehe auch Kapitel 2.2.1). Dies soll im Fehlerfall das Nachvollziehen des Fehlers ermöglichen, da auch aufgetretene Race-Conditions deterministisch wiederholt werden.

Das deterministische Betriebssystem *dOS* [11] erlaubt sowohl die deterministische Wiederholung einer Anwendungsausführung, als auch die voll-deterministische Ausführung beliebiger Anwendungen. Dazu werden Prozesse und Threads in sogenannten *Deterministic Process Groups (DPGs)* organisiert, die als einzelne deterministische Einheiten ausgeführt werden. Internes nichtdeterministisches Verhalten der einzelnen Prozesse, wie das Scheduling der Threads und Timings, werden durch die Nutzung eines deterministischen Schedulers verhindert. Dieser führt die DPGs in einer deterministischen Reihenfolge aus, während externes nichtdeterministisches Verhalten aufgezeichnet wird, um eine deterministische Wiederholung zu ermöglichen. Für dieselbe Eingabe in die Anwendung (dasselbe externe Verhalten) wird die Anwendung dabei immer deterministisch ausgeführt. Die deterministische Wiederholung ermöglicht dabei auch die Wiederholung nichtdeterministischer Eingaben (wie beispielsweise empfange Netzwerkkommunikation). Durch den, mit dem statischen Scheduling einhergehenden, Verlust der möglichen Parallelität steigt dabei auch die Ausführungszeit für die getesteten Anwendungen um bis zu eine Größenordnung (zehnfache Ausführungszeit) [11]. dOS kann für die Umsetzung deterministischer Verteilter Systeme genutzt werden, wie im folgenden Kapitel am Beispiel von *DDOS* [52] gezeigt wird.

Das *Deterministic Java Utility (DejaVu)* [3, 23] ist ein Werkzeug für die deterministische Wiederholung der Ausführung einer multithreaded Java-Applikation, die auf nur *einem* Prozessorkern ausgeführt wurde. DejaVu zielt dabei explizit auf Java-Applikationen, die für die Ausführung *Jalapeño* [2] nutzen, eine angepasste *Java Virtual Machine (JVM)*, die ei-

ne *Cross-Optimization* der Applikation zusammen mit der Laufzeitumgebung erlaubt, was wiederum die deterministische Ausführung einer Anwendung erschwert. DejaVu verfolgt zu diesem Zweck den internen Zustand der Anwendung durch *Reflection* [37] mit besonderem Fokus auf Wechsel des aktuell ausgeführten Threads. Hier wird die Anzahl vorher ausgeführter Aktionen gespeichert, sodass bei einer Wiederholung der Ausführung der Threadwechsel zum selben Zeitpunkt durchgeführt werden kann. Durch die Einschränkung der Ausführung auf nur einen Prozessorkern, wird ein deterministischer Schedule für die Threads erreicht. Des Weiteren werden die von der Applikation durchgeführten Operationen danach unterteilt, ob es sich um deterministische oder nichtdeterministische Funktionen handelt. Beim Aufruf nichtdeterministischer Funktionen wird das Ergebnis gespeichert, sodass dieses bei der Wiederholung der Ausführung wieder eingespielt werden kann. Der von DejaVu verfolgte Ansatz lässt sich, aufgrund der Einschränkung der Anwendbarkeit auf Systeme mit nur einem Prozessorkern, nicht sinnvoll auf ein Verteiltes System übertragen.

Samsara [84] ist ein weiteres System für die deterministische Wiederholung einer multithreaded Programmausführung. Im Gegensatz zu dem Ansatz, den Memory Zugriff innerhalb der Anwendung in eine deterministische Reihenfolge zu bringen, wird hier ein Ansatz verfolgt, der auf der Nutzung von hardwaregestützter Virtualisierung (*hardware-assisted virtualization*) basiert. Dazu werden während der Ausführung der Anwendung die *accessed*- und die *dirty*-Flags der Page-Tables ausgelesen, um nachzuhalten, auf welche Bereiche des Arbeitsspeichers die Anwendung zugegriffen hat, um eine deterministische Wiederholung der Anwendung zu ermöglichen. Samsara betrachtet hierbei nur den Zugriff auf den gemeinsam genutzten Arbeitsspeicher für die deterministische Ausführung. Eine Übertragbarkeit des Konzeptes auf ein System ohne geteilten Arbeitsspeicher ist dementsprechend nicht trivial zu realisieren.

Deterministische Programmiermodelle für parallele Anwendungen

Deterministic Parallel Java (DPJ) [15, 97] ist eine Erweiterung für die Programmiersprache Java, die die Entwicklung deterministischer Applikationen erleichtern soll. DPJ nutzt dazu ein auf Quelltext-Annotationen basierendes Type- und Effekt-System, das den Heap in hierarchische Regionen unterteilt und den Lese- und Schreibzugriff auf diese Regionen für jeden Prozess verwaltet. Die Regionen werden dabei genutzt, um den Zugriff auf Objekte oder auf Teile desselben Objektes erkennen zu können. Die Quelltext-Annotationen müssen vom Entwickler dazu genutzt werden, für jeden Prozess anzugeben, auf welche Regionen er zugreifen kann. Aufbauend auf diesen Informationen prüft DPJ, für welche Prozesse es überlappende Zugriffe auf den Arbeitsspeicher geben kann, die zu nichtdeterministischem Verhalten führen können und weist auf diese hin. Sind alle Quelltext-Annotationen korrekt, so kann kein nichtdeterministisches Verhalten versehentlich in die Anwendung eingeführt

werden. Aufgrund der Konzentration dieses Ansatzes auf die Zugriffe auf den gemeinsam genutzten Arbeitsspeicher ist eine Verteilung nicht ohne weiteres möglich.

MELD [32] integriert DPJ als deterministische Programmiersprache mit einem System zur deterministischen Ausführung von Anwendungen, das auf CoreDet (das bereits im Verlaufe dieses Kapitels beschrieben wurde) aufbaut, um die Stärken beider Ansätze zu verbinden. So kann DPJ für sich genommen keine beliebigen Anwendungen deterministisch ausführen, während CoreDet zwar beliebige Anwendungen deterministisch ausführen kann, die Verwendung allerdings mit einem hohen Performanceverlust einhergeht. MELD erreicht durch diese Kombination eine deterministische Ausführbarkeit für beliebige Anwendungen, die mit weniger Performanceverlusten als eine reine Nutzung von CoreDet und ohne die Notwendigkeit für die Anpassung der Anwendung (in Form von Quelltext-Annotationen) einhergeht. MELD setzt dazu (ebenso wie DPJ und CoreDet) einen gemeinsam genutzten Arbeitsspeicher voraus und eignet sich nicht für die Verteilung des Ansatzes.

Cilk++ ist eine auf C++basierende Umsetzung des bereits beschriebenen Cilk-Ansatzes. In [39] wird Cilk++ um sogenannte *Hyperobjekte (Hyperobjects)* erweitert, um eine Ausführung einer Cilk++-basierten Anwendung zu ermöglichen, die frei von Daten-Race-Conditions ist. Hyperobjekte stellen eine private, lokale Sicht auf nicht lokale Variablen für unterschiedliche Prozesse einer multithreaded Anwendung dar. Diese ermöglichen es Prozessen ohne eine Synchronisation mit anderen Prozessen auf diesen Hyperobjekten zu arbeiten, da es sich um eine private Ressource handelt. Erst bei der Synchronisation mit einem anderen Prozess werden die Veränderungen an den Hyperobjekten der beiden Prozesse wieder deterministisch zusammengeführt. Die Art der Zusammenführung muss dabei vom Entwickler spezifiziert werden. Da die Einführung von Hyperobjekten nur dafür sorgt, dass es keine Daten-Race-Conditions mehr geben kann, können sich weiterhin Race-Conditions aufgrund des Schedulings der Prozesse und deren Kommunikation ergeben. Dementsprechend kann dieser Ansatz nicht als deterministisch angesehen werden.

Coordination Languages

Concurrent Collections (CnC) [19] sind ein weiteres Programmiermodell für deterministische parallele multithreaded Anwendungen. CnC erlaubt es dabei, dass die Anwendungslogik und die Parallelisierung getrennt voneinander betrachtet werden können. Zur Entwicklung einer Anwendung, die auf CnC basiert, muss der Entwickler die Anwendungslogik in Form eines Graphen spezifizieren, der aus miteinander kommunizierenden Kernels besteht (die den Quellcode enthalten). Für die Parallelisierung der Anwendung kann anschließend ein für die Applikationslogik und die Ausführungsumgebung passender Schedule angegeben werden. Um eine deterministische Ausführung sicherzustellen, können Ressourcen nicht von mehreren Kernels gemeinsam genutzt werden. Stattdessen müssen die Ressourcen

in Form einer Kopie (einer neuen sogenannten *Data Collection*) an die einzelnen Kernels übergeben werden. Die Kernels konsumieren diese Data Collections und können als Ausgabe neue Data Collections erstellen, die als Eingabe für weitere Kernels dienen können. So ist auch bei einer nebenläufigen Ausführung der Kernels ein deterministisches Verhalten möglich. Die deterministische Zusammenführung der divergenten Data Collections wird dabei dem Entwickler der Kernels überlassen. Da die einzelnen Kernels nebenläufig ausgeführt werden können sobald die erforderliche Eingabe in Form der Data Collections vorliegt, ließe sich der Ansatz auch auf ein Verteiltes System übertragen. Dazu könnten die Kernels bestimmten Rechenknoten zugewiesen werden und die Eingabe Data Collections an die entsprechenden Knoten gesendet werden.

Funktionale Programmierung

Coutts [25] schlägt vor, die funktionale Programmiersprache *Haskell* für deterministische parallele Anwendungen zu nutzen. Die Nutzung einer funktionalen Programmiersprache hat dabei den Vorteil, dass diese von Haus aus keine Seiteneffekte zwischen einzelnen Funktionen aufweisen können, die somit ohne weiteres parallel ausgeführt werden können. Das deterministische Ergebnis der Anwendung ist dabei abhängig davon, ob die Eingabe in die Funktionen deterministisch ist. Für eine deterministische verteilte Haskell Anwendung müsste dementsprechend ein Mechanismus integriert werden, der die Netzwerkkommunikation (die Teil der Eingabe in eine Funktion ist) in eine deterministische Reihenfolge bringt. Eine funktionale Haskell Anwendung kann aus diesem Grund nicht ohne weitere Mechanismen trivial verteilt ausgeführt werden.

2.2.3 Determinismus in Verteilten Systemen

Die bis hierhin beschriebenen Ansätze beziehen sich allesamt auf die deterministische Ausführung von multithreaded Anwendungen mit geteiltem Arbeitsspeicher und können nur vereinzelt trivial auf ein Verteiltes System (Multi Computer System) abgebildet werden. Im Folgenden werden hingegen Ansätze beschrieben, die eine deterministische Ausführung nebenläufiger Verteilter Systeme ermöglichen.

Deterministische Wiederholung

Das *Distributed Deterministic Operating System (DDOS)* [52] ist ein deterministisches Verteiltes System, das auf dem bereits beschriebenen *dOS* aufsetzt. DDOS kann dabei sowohl für die deterministische Wiederholung einer Ausführung der Anwendung, als auch für die deterministische Ausführung beliebiger Anwendungen verwendet werden. In DDOS wird ein Verteiltes System als Menge von *Deterministic Process Groups (DPGs)* beschrieben, die auf verschiedene Rechenknoten verteilt sein können. Auf jedem Rechenknoten werden

die DPGs durch die Verwendung von dOS lokal deterministisch ausgeführt. DDOS stellt zusätzlich sicher, dass die Kommunikation zwischen den DPGs (innerhalb eines Rechenknotens und zwischen Rechenknoten) in einer deterministischen Art und Weise geschieht. Im Falle der deterministischen Wiederholung der Anwendung wird die aufgezeichnete Sende- und Empfangsreihenfolge der ausgetauschten Nachrichten reproduziert. Im Falle der deterministischen Ausführung einer beliebigen Anwendung werden Nachrichten bis zu festgelegten periodischen Synchronisationspunkten zwischengespeichert und anschließend deterministisch zugestellt. DDOS setzt dabei auf das Konzept des bereits in Kapitel 2.2.2 beschriebenen Sende-Determinismus, der aussagt, dass ein deterministischer Nachrichtenaustausch für die deterministische Ausführung einer Anwendung ausreichend ist, solange die einzelnen Prozesse deterministisch sind. Diese Voraussetzung wird hier durch die DPGs und dOS erfüllt. Die voll-deterministische Ausführung des Verteilten Systems sorgt dabei für eine etwa verzehnfachte Ausführungszeit (im Vergleich zur nichtdeterministischen Ausführung) [52].

Deterministische Ausführung

Determinator [6] ist eine Kombination aus einem Betriebssystem und einer Laufzeitumgebung für deterministische lokale und verteilte Anwendungen, wobei die Nutzbarkeit für verteilte Anwendungen aufgrund schlechter Performance auf *embarrassingly parallel* Anwendungen relativiert wird. Determinator verhindert Daten-Race-Conditions innerhalb einer Anwendung dadurch, dass jeder Prozess nur einen privaten Workspace, nicht aber einen geteilten Arbeitsspeicher hat. Konflikte bei der Zusammenführung der lokalen Kopien müssen vom Entwickler behoben werden. Die Kommunikation zwischen Prozessen kann außerdem nur über eine deterministische Schnittstelle stattfinden, wodurch eine deterministische Synchronisation der Prozesse sichergestellt wird. Ein Prozess kann dabei nur mit seinem Elternprozess, der ihn gestartet hat, und seinen Kinderprozessen kommunizieren. Um weitere Quellen für nichtdeterministisches Verhalten innerhalb der Prozesse auszuschließen, verhindert Determinator zusätzlich den Zugriff auf jegliche Hardwareressourcen.

Coordination Language

Orc [68] ist eine Spezifikationssprache zur Definition von deterministischen Internet Systemen, die genutzt werden können, um Aussagen über Services und Abläufe im Internet zu treffen. Die Spezifikationssprache ist sehr ähnlich zum bereits beschriebenen Ansatz von *CnC* und setzt auch eine abstrakte Definition der Applikationslogik voraus. Orc bietet für die Spezifikation des Verteilten Systems drei Komponenten an: Datenmanagement, arithmetische und logische Operationen, sowie Kommunikationskanäle. Wie bei CnC ergibt

sich das Verhalten des Programmes aus dem Aufruf der für eine Anwendung spezifizierten Komponenten zusammen mit deren Eingaben, wodurch neue Werte generiert werden, die im nachfolgenden Schritt wiederum als Eingabedaten genutzt werden können. Orc organisiert dabei die Ausführung und Übertragung der Ergebnisse zwischen den einzelnen Komponenten.

2.2.4 Zusammenfassung

Tabelle 2.1 fasst noch einmal die Eigenschaften der Systeme für verteile (und verteilbare) deterministische Anwendungen zusammen. Für die Systeme wird dabei angegeben, ob diese verteilt sind, welche Determinismusart sie umsetzen, und ob sie reproduzierbare Programmabläufe erreichen. Anschließend wird eine Aussage darüber getroffen, ob die Systeme mit zusätzlichen Ressourcen skalieren und ob sie sich für alle Probleme eignen, oder nur für bestimmte Probleme (Generalität). Des Weiteren werden die Wiederverwendbarkeit von entwickelten Komponenten in dem entsprechenden Ansatz, sowie Einschränkungen, die sich bei der Verwendung des Ansatzes ergeben, angegeben. Als letzte Eigenschaften werden abschließend die mit dem Ansatz einhergehenden, zu erwartenden Performanceverluste und die Nutzbarkeit der Ansätze angegeben. Diese Aufstellung der Eigenschaften wird in Kapitel 6.3 noch einmal aufgegriffen, um den in dieser Arbeit entwickelten Prototypen in den Kontext der verwandten Arbeiten einordnen zu können.

Hervorzuheben ist hierbei, dass sowohl *DDOS* als auch *Determinator* die deterministische Ausführung beliebiger Anwendungen ermöglichen, dabei aber aufgrund der verwendeten Mechanismen nicht skalierbar sind. *Orc* und *CnC* bieten eine deterministische Ausführung aller Funktionsaufrufe und skalieren dabei mit hinzugefügten Ressourcen. Allerdings ist für die Verwendung beider Ansätze eine Definition der Anwendungslogik in einer Spezifikationssprache erforderlich, was die Verwendung für die Entwicklung beliebiger Anwendungen erschwert.

Eigenschaft	DDOS	Determinator	Orc (CnC)
Verteilt	Ja	Ja	Ja / (Möglich)
Determinismus	Voll/Replay	Voll/Replay	Funktionen
Reproduzierbarkeit	Alles	Alles	Funktionsaufrufe
Skalierbarkeit	Eingeschränkt durch Barrieren	Embarrassingly parallel	Skaliert
Generalität	Alles	Alles	Speziell
Wiederverwendbarkeit	Ja	Ja	Nein
Einschränkungen	Skalierbarkeit	Skalierbarkeit	Domäne / Keine Zwischenergebnisse
Verlangsamung	10x	bis zu 10x	–
Einfache Verwendung	Ja	Ja	Graphsprache

Tabelle 2.1: Eigenschaften der verteilten deterministischen Systeme.

Kapitel 3

Determinismus auf Applikationsebene

Die deterministische Ausführung einer Anwendung bietet viele Vorteile, sowohl während der Entwicklung der Anwendung, als auch während ihrer Ausführung. Bei der Entwicklung einer Anwendung, die garantiert deterministisch ausgeführt wird, hat der Entwickler die Zusicherung, dass die Anwendung bei gleicher Eingabe auf dieselbe Weise ausgeführt wird. Dies bedeutet insbesondere, dass eine fehlerfreie Ausführung auf dem Entwicklungssystem gleichzeitig bedeutet, dass sich die Anwendung bei derselben Eingabe auf anderen Systemen gleich verhält. Außerdem können Fehler bei der Entwicklung schneller zurückverfolgt werden, da sie bei gleicher Eingabe deterministisch auftreten. Falls beispielsweise bei der Ausführung der Anwendung beim Nutzer ein Fehler oder ein unerwartetes Verhalten der Anwendung auftritt, dann kann dies auf einem anderen System (z.B. dem Entwicklungssystem) nachvollzogen werden, indem die Eingabe wiederholt wird.

Ein Problem, das viele der in Kapitel 2.2 beschriebenen Ansätze für deterministische Anwendungen[1] gemeinsam haben, ist, dass die Verwendung dieser Mechanismen mit hohen Performanceverlusten verbunden ist und nicht mit einer Erhöhung der Rechenkapazitäten der Ausführungsumgebung skaliert, oder die Ansätze auf spezielle Domänen oder funktionale Programmierung beschränkt sind.

In diesem Kapitel wird darauf eingegangen, wodurch Geschwindigkeitsverluste bei der deterministischen Ausführung einer Anwendung verursacht werden und was die Skalierbarkeit deterministischer Anwendungen erschwert. Dazu wird untersucht, inwiefern die deterministische Ausführung aller Bereiche innerhalb einer Anwendung notwendig ist, um eine reproduzierbare Anwendungsausführung sicherzustellen, und ob es Bereiche gibt, in denen eine nichtdeterministische Ausführung das Ergebnis nicht beeinflussen würde.

[1] Gemeint sind Anwendungen mit mehr als einem Prozess/Thread. Anwendungen mit nur einem Prozess/Thread sind (unter der Annahme, dass keine von Haus aus nichtdeterministischen Funktionsaufrufe getätigt werden) durch die serielle Ausführung generell deterministisch.

Aufbauend auf dieser Untersuchung wird das Konzept des Determinismus auf Applikationsebene eingeführt, das eine skalierende aber dennoch deterministische Ausführung der Anwendungslogik ermöglicht.

3.1 Synchronisation nebenläufiger Prozesse

Im Folgenden wird beispielhaft eine Anwendung betrachtet, die aus mehreren nebenläufigen, miteinander kommunizierenden Prozessen besteht (siehe Abbildung 3.1). Jeder der Prozesse A (blau), B (orange) und C (grün) führt eine Menge an Aktionen aus und kann mit anderen Prozessen kommunizieren (Pfeile). Diese Kommunikation ist notwendig, um beispielsweise den Zugriff auf geteilte Ressourcen zu synchronisieren (z.B. durch Nachrichtenaustausch oder das Locking von Datenstrukturen).

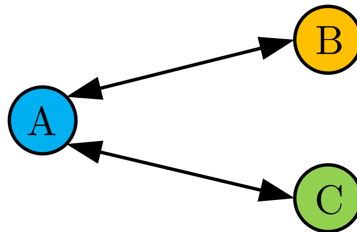


Abbildung 3.1: Beispielanwendung mit drei kommunizierenden Prozessen.

3.1.1 Events und Happened-before Relationen

Bei der Ausführung einer Anwendung generiert jeder Prozess eine Sequenz von Ereignissen (*Events*) [61]. Ein Event kann dabei lokal sein, sodass es nur den eigenen Prozess betrifft (z.B. ein lokaler Funktionsaufruf), oder ein Synchronisationsevent, das zwei oder mehr Prozesse betrifft (z.B. ein Nachrichtenaustausch zur Synchronisation). Abbildung 3.2 zeigt einen beispielhaften *Eventgraphen* zur Veranschaulichung der Events der drei nebenläufigen Prozesse aus dem vorangegangenen Beispiel. Der Eventgraph stellt dabei die Zusammenhänge zwischen den einzelnen nebenläufigen Prozessen unabhängig von der Ausführungsumgebung dar. Lokale Events sind in dem Eventgraphen grau dargestellt (z.B. die Events a und d), während die Kommunikation zwischen Prozessen durch Pfeile in der Farbe des Prozesses dargestellt werden (z.B. das Sendeevent b und Empfangsevent c zwischen den Prozessen A und B).

Events können zueinander in einer sogenannten *Happened-Before Relation* stehen [61]. Die Happened-Before Relation (auch beschrieben als „$\rightarrow$“) trifft eine Aussage darüber, ob zwei Events in einer Anwendung in einer definierten Reihenfolge geschehen, oder ob die

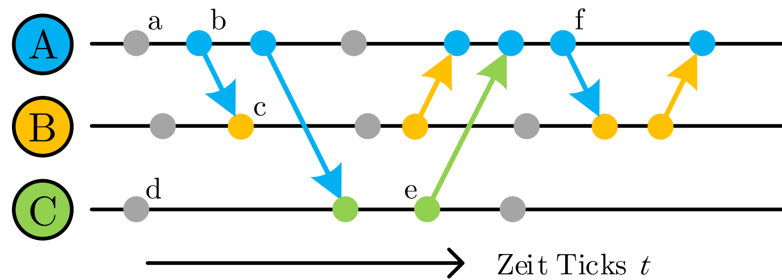


Abbildung 3.2: Beispiel für einen Eventgraphen.

Events nebenläufig sind. Nach Lamport muss die „→“-Relation dabei drei Bedingungen erfüllen [61]:

1. Wenn die Events a und b Events desselben Prozesses sind, und a vor b geschieht, dann gilt $a \rightarrow b$.
2. Wenn a ein Sendeevent ist und b das zugehörige Empfangsevent, dann gilt $a \rightarrow b$.
3. Wenn $a \rightarrow b$ und $b \rightarrow c$ gelten, dann gilt auch $a \rightarrow c$.

Des Weiteren gelten zwei Events a und b als *nebenläufig*, wenn sowohl $a \nrightarrow b$ als auch $b \nrightarrow a$ gilt. Die Happened-Before Relation kann somit aussagen, welche Events sich gegenseitig kausal beeinflussen können [61]. So bedeutet $a \rightarrow b$, dass es für das Event a möglich ist, das Event b zu beeinflussen. Durch die Kommunikation zwischen Prozessen können sich diese Abhängigkeiten zwischen Events über Prozessgrenzen hinaus ausbreiten.

Bezogen auf das Beispiel in Abbildung 3.2 bedeutet dies für die Events a bis d folgendes: Es gilt $a \rightarrow b$, da a und b beides Events in demselben Prozess sind und a vor b stattfindet. Des Weiteren gilt $b \rightarrow c$, da b und c Events derselben Nachrichtenübertragung sind. Abschließend gilt $a \rightarrow c$, da $a \rightarrow b$ und $b \rightarrow c$ gelten (Transitivität). Das Event d ist dabei nebenläufig zu den Events a bis c, da es keine Happened-Before Relation zwischen diesen Events gibt.

3.1.2 Synchronisationspunkte

Events, die bei der Kommunikation zwischen den Prozessen entstehen, werden *Synchronisationspunkte* (oder auch *Synchronisationsevents*) genannt [22]. Ein Synchronisationspunkt stellt einen Informationsaustausch oder eine zeitliche Synchronisation zwischen zwei oder mehreren Prozessen dar. Im Verlaufe der Ausführung der Anwendung ergibt sich somit eine (für diese Ausführung) feste Reihenfolge in der die Synchronisationspunkte aufgetreten sind (im Folgenden *Synchronisationsgraph* genannt). Ein Synchronisationsgraph stellt im Gegensatz zu einem Eventgraphen die Reihenfolge der Synchronisationspunkte im

Zusammenhang mit den Prozessorkernen, auf denen die Prozesse ausgeführt wurden, dar.

Ein Beispiel für einen Synchronisationsgraphen, der sich aus dem Eventgraphen in Abbildung 3.2 bei der Verwendung eines einzelnen Prozessorkerns ergeben kann, ist in Abbildung 3.3 zu sehen. Hier werden die Prozesse A, B und C auf demselben Prozessorkern ausgeführt, sodass die Synchronisationspunkte für das Senden und Empfangen von ausgetauschten Nachrichten hintereinander im Synchronisationsgraphen zu sehen sind. Das Betriebssystem weist dabei jedem Prozess einen Teil der zur Verfügung stehenden Rechenzeit zu, die dieser für die Durchführung seiner Events nutzt.

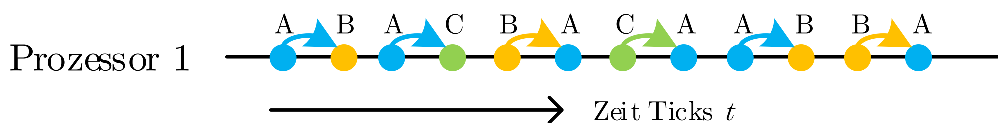


Abbildung 3.3: Beispiel für einen Synchronisationsgraphen mit einem Prozessor.

Die Reihenfolge, in der die Synchronisationspunkte auftreten, kann sich zwischen mehreren Ausführungen der Anwendungen ändern. Der Grund dafür sind nichtdeterministische Timings der Prozessoren, das Scheduling des Betriebssystems, sowie mögliche Veränderungen der zur Verfügung stehenden Ressourcen bei einem Wechsel der Ausführungsumgebung (durch die ein höherer oder geringerer Grad der Parallelität erreicht werden kann). Abbildung 3.4 zeigt zur Veranschaulichung der Problematik einen Synchronisationsgraphen (ebenfalls für den Eventgraphen in Abbildung 3.2), der sich bei einem zusätzlich verfügbaren Prozessorkern ergeben kann.

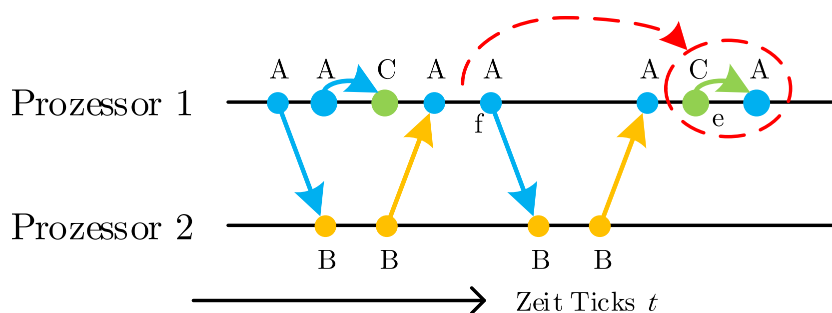


Abbildung 3.4: Beispiel für einen Synchronisationsgraphen mit zwei Prozessoren.

In diesem Beispiel wird die Ausführung von Prozess B auf den hinzugekommenen Prozessorkern ausgelagert. Die geänderten Timings innerhalb der Anwendung können dabei

dafür sorgen, dass das Betriebssystem die Ausführung der zweiten Nachrichtenübertragung von A nach B, sowie das Empfangen der Antwort von Prozess B an Prozess A vor die Ausführung der Nachrichtenübertragung von C nach A zieht. Dabei ändern sich auch die Happened-Before Relationen zwischen den Events e und f von $e \rightarrow f$ zu $f \rightarrow e$. Dies bedeutet, dass sich beispielsweise die Reihenfolge einer Lock-Zuweisung geändert haben könnte, die das Ergebnis der Anwendung beeinflusst (Race-Condition). An diesem Punkt setzen voll-deterministische Systeme an.

3.2 Voll-deterministische Systeme

Um ein deterministisches Verhalten einer Anwendung zu erreichen, ist die Einhaltung der Reihenfolge (und somit der Happened-Before Relation) von denjenigen Synchronisationspunkten, die sich untereinander tatsächlich beeinflussen (z.B. wenn sie den Zugriff auf dieselbe Ressource betreffen), über mehrere Programmausführungen hinweg notwendig [88]. Eine sich ändernde Reihenfolge dieser Synchronisationspunkte kann dazu führen, dass es innerhalb der Anwendung zu nichtdeterministischem Verhalten aufgrund von Race-Conditions kommt. Voll-deterministische Systeme führen aus diesem Grund alle internen Aktionen (Events) einer Anwendung immer in derselben Reihenfolge aus (z.B. durch das Durchsetzen eines festgelegten Schedules, siehe auch Kapitel 2.2). So wird sichergestellt, dass ein deterministischer Eventgraph erreicht wird und somit auch die Reihenfolge der Synchronisationspunkte bei jeder Ausführung dieselbe ist. Während der Ausführung der Anwendung ergibt sich somit nicht nur ein deterministisches Endergebnis, sondern auch eine reproduzierbare Abfolge der internen Zwischenstände der Anwendung.

Der Nachteil dieses Ansatzes sind die Performanceeinbußen, die mit der Sicherstellung einer voll-deterministischen Ausführung einer Anwendung einhergehen (insbesondere bei deterministischen verteilten Anwendungen, siehe z.B. *dDOS* [52]). Diese Einbußen ergeben sich daraus, dass eine voll-deterministische Anwendung einen Großteil der flexibel erreichbaren Parallelität verliert. Die eingeschränkte Parallelität ergibt sich dabei daraus, dass die voll-deterministische Anwendung einen deterministischen Eventgraphen erzwingt. Dieser kann sich dabei weder auf die nichtdeterministischen Timings der CPUs einstellen, die zwischen den Ausführungen der Anwendung variieren können, noch können zusätzliche Ressourcen in die Ausführung der Anwendung mit einbezogen werden, um diese zu beschleunigen.

Bezogen auf den Eventgraphen aus Abbildung 3.2 bedeutet dies, dass der Synchronisationsgraph aus Abbildung 3.4 für eine voll-deterministische Anwendung nicht zulässig ist, da die Happened-Before Relation $e \rightarrow f$ im Synchronisationsgraph nicht mehr eingehalten wird. Ein voll-deterministisches System würde dementsprechend das Event f während der Ausführung soweit verzögern, bis die Happened-Before Relation $e \rightarrow f$ wieder gilt, wo-

durch sich für die einzelnen Prozessoren Leerlaufzeiten ergeben können (siehe Abbildung 3.5). Der Ansatz skaliert des Weiteren nicht mit hinzugefügten Ressourcen für die Ausführung der Anwendung (z.B. zusätzliche Prozessorkerne), da diese aufgrund des statischen Schedulings nicht genutzt werden können.

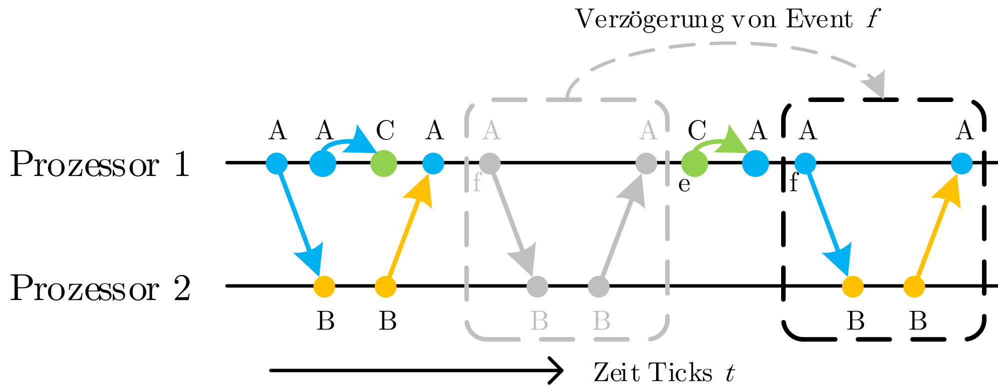


Abbildung 3.5: Verzögerung von Events bei vollem Determinismus.

3.2.1 Reduzierung des Performanceverlustes durch vollen Determinismus

Eine voll-deterministische Anwendung erzwingt, wie im vorangegangenen Kapitel beschrieben, die Einhaltung aller Happened-Before Relationen innerhalb der Anwendung, um einen reproduzierbaren Programmablauf und ein deterministisches Ergebnis zu erreichen. Dies führt dabei, aufgrund der Wartezeiten zur Erhaltung der Eventreihenfolge, zu hohen Performanceeinbußen, die mit der Anzahl einzuhaltender Happened-Before Relationen zwischen den nebenläufigen Prozessen ansteigen.

Allerdings ist nicht zwingend die Einhaltung *aller* Happened-Before Relationen notwendig, um ein deterministisches Ergebnis und einen reproduzierbaren Programmablauf zu erreichen, da sich nicht alle Events gegenseitig beeinflussen. Das liegt daran, dass eine Happened-Before Relation nur angibt, dass ein Event a (das vor einem anderen Event b geschieht) dieses Event b beeinflussen *kann*, aber nicht zwingend beeinflussen *muss* (siehe Kapitel 3.1.1). Eine Reduzierung der einzuhaltenden Happened-Before Relationen auf diejenigen, die für eine deterministische Ausführung erforderlich sind, könnte somit zur Realisierbarkeit effizienterer deterministischer Anwendungen führen.

Bezogen auf die bereits genannten Beispiele führt ein Verzicht auf die Einhaltung von (für die deterministische Ausführung unkritischen) Happened-Before Relationen dazu, dass in Abbildung 3.4 das Event f direkt eintreten darf. Dies darf natürlich nur dann zugelassen werden, wenn das Event f keinen Einfluss auf die Events ausüben kann, vor die es gezogen wurde. In diesen Fällen ist es möglich, Wartezeiten und somit auch die

Laufzeit der gesamten Anwendung zu reduzieren.

Ein weiterer positiver Effekt, der sich daraus ergibt, ist die Möglichkeit, zusätzliche Ressourcen der Ausführungsumgebung für einen höheren Grad an Parallelität nutzen zu können, sodass die Anwendung trotz der deterministischen Ausführung skalieren kann. Dazu ist sowohl ein Verzicht auf die Nutzung statischer Schedules für die Zuordnung von Prozessen zu Prozessorkernen notwendig, als auch das Wissen, welche Events einzelner Prozesse nebenläufig ausgeführt werden können, ohne dass das deterministische Verhalten der Anwendung beeinträchtigt wird. Noch einmal bezugnehmend auf das Beispiel aus Abbildung 3.2 ist es somit möglich, jeden Prozess auf einem eigenen Prozessorkern auszuführen. Dabei muss allerdings sichergestellt werden, dass keine kritische Happened-Before Relation verloren geht, die sich auf eines der nun nebenläufigen Events bezieht (siehe Abbildung 3.6).

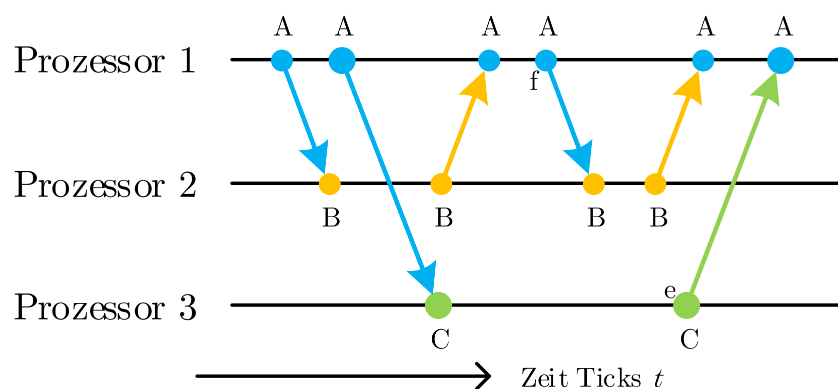


Abbildung 3.6: Skalierbarkeit der deterministischen Anwendung.

Die Reduktion der Happened-Before Relationen auf diejenigen, die für eine deterministische Ausführung der Anwendung notwendig sind, bewirkt gleichzeitig eine Reduktion der Performanceverluste, die mit einer deterministischen Ausführung einer Anwendung einhergehen. Allerdings ist es sowohl für die Laufzeitumgebung, als auch für die Anwendung selbst, sehr schwierig herauszufinden, an welchen Stellen der Anwendung die Einhaltung einer Happened-Before Relation notwendig ist, und an welchen Stellen auf die Einhaltung verzichtet werden kann. Im folgenden Kapitel wird ein Ansatz vorgestellt, der an die Erkenntnisse dieses Kapitels anknüpft und eine neue Art des Determinismus einführt, die eine Reduktion der einzuhaltenden Happened-Before Relationen ermöglicht, und somit die Erstellung effizienter deterministischer Anwendungen erlaubt.

3.3 Determinismus auf Applikationsebene

In diesem Kapitel wird ein Konzept für die reproduzierbare Ausführung einer Anwendung mit einem deterministischen Ergebnis vorgestellt. Das Konzept lockert die Anforderung von voll-deterministischen Anwendungen, dass alle Happened-Before Relationen zwischen Events der Anwendung eingehalten werden müssen. Dabei wird die Erkenntnis aus Kapitel 3.2.1 genutzt, dass für ein deterministisches Ergebnis der Anwendung nicht zwingend die Einhaltung aller internen Eventreihenfolgen notwendig ist.

Bei der Ausführung einer voll-deterministischen Anwendung gibt es innerhalb der Abwendung „guten“ und „schlechten“ Determinismus. *Guter Determinismus* beschreibt hier die Einhaltung von Happened-Before Relationen zwischen Events, die ein deterministisches Ergebnis der Anwendung sicherstellen. *Schlechter Determinismus* beschreibt die Einhaltung von Happened-Before Relationen zwischen Events, die für ein deterministisches Ergebnis *nicht* notwendig sind. Dieser schlechte Determinismus kann dafür sorgen, dass der mögliche erreichbare Grad an Parallelität in der Anwendung unnötigerweise eingeschränkt wird. An diesen Stellen wäre es für eine Anwendung möglich Nichtdeterminismus zu tolerieren, ohne dass das deterministische Ergebnis der Anwendung beeinflusst wird. Die Entscheidung, ob die Einhaltung von Eventreihenfolgen einen guten oder schlechten Determinismus darstellt, erfordert dabei anwendungsspezifisches Wissen, weshalb voll-deterministische Ansätze sicherheitshalber alle Happened-Before Relationen deterministisch einhalten (siehe Kapitel 2.2).

Um den Geschwindigkeitsverlust, der mit schlechtem Determinismus einhergeht, zu reduzieren, wird hier das Konzept des *Determinismus auf Applikationsebene* vorgestellt. Eine Anwendung wird dabei als „deterministisch auf Applikationsebene“ bezeichnet, wenn der für den Entwickler oder Nutzer der Anwendung beobachtbare Programmfluss der Anwendungslogik bei jeder Ausführung, unabhängig von der Ausführungsumgebung, dieselben Funktionsaufrufe mit identischen Berechnungen, Eingaben und Ausgaben beinhaltet. Währenddessen kann die Anwendung intern nichtdeterministisches Verhalten beinhalten, um schlechten Determinismus zu vermeiden, solange dies den Determinismus auf Applikationsebene nicht beeinflusst. Zu diesem Zweck wird die Sicht auf den Determinismus der Anwendung in zwei voneinander getrennt betrachtete Bereiche entkoppelt.

Die *Applikationsebene* beinhaltet die vom Entwickler geschriebene Anwendungslogik und wird garantiert deterministisch ausgeführt, unabhängig von Prozessortimings oder wechselnden Ausführungsumgebungen. Somit ist die sichtbare Anwendungsausführung reproduzierbar. Um die Determinismusgarantien auf der Applikationsebene zu erreichen, muss sichergestellt werden, dass immer dieselben Funktionen mit denselben Eingabedaten (Funktionsparametern) aufgerufen werden, und dass die Ergebnisse der Funktionen immer

identisch sind[2].

Die *Frameworkebene* enthält die internen Mechanismen des Frameworks, die unter anderem die Synchronisation zwischen nebenläufig ausgeführten Prozessen der Anwendung übernehmen und die Zuweisung von Prozessen auf Prozessorkerne steuern. Diese Mechanismen dürfen auch nichtdeterministisches Verhalten beinhalten. Dies erlaubt es einer Anwendung zu skalieren und den Grad der erreichbaren Parallelität in Abhängigkeit von der Ausführungsumgebung besser zu nutzen.

Die Frameworkebene und die Applikationsebene sind über eine Reihe von Mechanismen miteinander verbunden. Diese Mechanismen stellen sicher, dass die Anwendungslogik auf die Funktionalitäten der Frameworkebene zugreifen kann, ohne dass sich der darin enthaltene Nichtdeterminismus in die Anwendungslogik ausbreitet, indem nichtdeterministische Ereignisse wieder in eine deterministische Form für die Applikationsebene gebracht werden (siehe Abbildung 3.7). Insgesamt wird durch das Zulassen nichtdeterministischer Komponenten in der Frameworkebene (und dem damit einhergehenden höheren Grad an Parallelität für die gesamte Anwendung) der Geschwindigkeitsverlust für eine deterministische Ausführung reduziert. Das gilt dabei unter der Voraussetzung, dass die Kosten für die Mechanismen der Frameworkebene nicht den Geschwindigkeitsvorteil übersteigen[3].

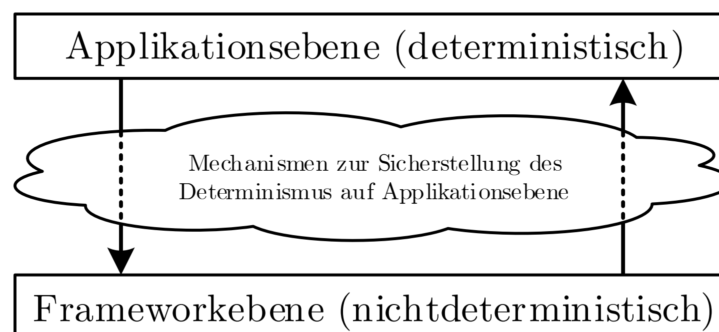


Abbildung 3.7: Aufteilung der Determinismusgarantien.

Das Konzept des Determinismus auf Applikationsebene unterscheidet sich von den in Kapitel 2.2.1 vorgestellten Determinismusarten. Im Gegensatz zu voll-deterministischen Anwendungen dürfen hier nichtdeterministische Mechanismen genutzt werden, um beispielsweise eine bessere Skalierbarkeit zu erreichen, solange die Ausführung der Applikationsebene deterministisch bleibt. Von der deterministischen Wiederholung unterscheidet sich das Konzept dadurch, dass sich jede Ausführung der Anwendung gleich verhält, und nicht nur eine nichtdeterministische Ausführung deterministisch wiederholt

[2]Daraus ergibt sich gleichzeitig, dass die Applikationsebene frei von Race-Conditions ist.

[3]Dies wird anhand einer Proof-of-Concept Implementierung des Spawn & Merge Programmiermodells, das das Konzept des Determinismus auf Applikationsebene umsetzt, in Kapitel 7 evaluiert.

werden kann. Der Determinismus auf Applikationsebene unterscheidet sich auch vom Sende-Determinismus, da eine nichtdeterministische Kommunikation zwischen Prozessen innerhalb der Frameworkebene explizit erlaubt ist. Der „Don't Care" Nichtdeterminismus erlaubt das Durchlaufen nichtdeterministischer Zwischenstände bei der Ausführung einer Anwendung, wodurch auf Kosten der Reproduzierbarkeit ein höherer Grad an Parallelität ermöglicht wird.

Das Konzept des Determinismus auf Applikationsebene ähnelt am ehesten der Funktionsweise von Coordination Languages oder verteilter funktionaler Programmiersprachen, da diese ebenfalls auf eine Trennung von Funktionsaufrufen und dem Scheduling dieser Aufrufe setzen. Diese Ansätze sind dabei allerdings entweder auf eine spezielle Domäne beschränkt, oder auf funktionale Programmierung. Diese Einschränkung gilt für den Determinismus auf Applikationsebene nicht, der in General Purpose Languages (GPLs) integriert werden kann.

3.3.1 Anforderungen an ein Programmiermodell

Um eine Anwendung nach dem Konzept des Determinismus auf Applikationsebene realisieren zu können, muss ein Framework entscheiden können, ob es sich bei einem Ereignis innerhalb der Anwendung um einen „guten Determinismus" handelt, der beibehalten werden muss, oder um einen „schlechten Determinismus", der auf der Frameworkebene aufgeweicht werden kann. Für Events, bei denen eine nichtdeterministische Ausführung in der Frameworkebene zugelassen wurde, muss das Framework gleichzeitig Mechanismen bereitstellen können, die diese wieder in eine deterministische Form für die Weiterverarbeitung in der Applikationsebene bringen.

Diese Anforderungen können durch ein Programmiermodell erfüllt werden, das speziell für die Entwicklung von Anwendungen gedacht ist, die das Konzept des Determinismus auf Applikationsebene umsetzen. Anhand der Bestandteile des Programmiermodells könnten dabei die Entscheidungen getroffen werden, welche Aktionen vom Framework nichtdeterministisch durchgeführt werden dürfen und welche Aktionen direkt in der Applikationsebene durchgeführt werden müssen. Da die möglichen nichtdeterministischen Aktionen direkt durch das Programmiermodell vorgegeben werden, ist es ebenso möglich, dass das Programmiermodell für eben diese Aktionen Mechanismen für eine deterministische Zusammenführung bereitstellt. Somit wäre die Grundlage für die Entscheidung, was deterministisch ausgeführt werden muss und was nichtdeterministisch ausgeführt werden darf, implizit gegeben. Im nächsten Kapitel wird das Programmiermodell Spawn & Merge vorgestellt, das diesen Anforderungen genügt.

Kapitel 4

Deterministische Synchronisation mit Spawn & Merge

In diesem Kapitel wird das Konzept für das Programmiermodell Spawn & Merge beschrieben, das eine deterministische Synchronisation nebenläufig ausgeführter Programme ermöglicht. Das entwickelte Programmiermodell ist dabei unabhängig von der Ausführungsumgebung und kann sowohl für Anwendungen, die auf einem Mehrkernprozessor mit geteiltem Arbeitsspeicher laufen, als auch für verteilte Anwendungen ohne geteilten Arbeitsspeicher genutzt werden. Im Folgenden werden dazu die Anforderungen beschrieben, die das konzipierte Programmiermodell erfüllen soll. Anschließend werden die neu eingeführten Synchronisationsprimitive und deren Funktionsweise definiert. Abschließend werden beispielhaft die Einbettung von Spawn & Merge in C++, sowie die Anwendungsgebiete des Programmiermodells betrachtet.

4.1 Anforderungen und Lösungsansatz

Bei der Konzeption des Spawn & Merge Programmiermodells stehen drei Anforderungen im Vordergrund. Das Programmiermodell soll Entwicklern die Möglichkeit bieten nebenläufig ausgeführte Anwendungen zu entwickeln, die *standardmäßig deterministisch* sind. Dementsprechend muss das Auftreten von Race-Conditions innerhalb der Anwendungen verhindert werden. Um die Geschwindigkeitseinbußen zu umgehen, die voll-deterministische Anwendungen mit sich bringen (siehe Kapitel 3.2), wird bei Spawn & Merge das Konzept des Determinismus auf Applikationsebene angewendet, das in Kapitel 3.3 eingeführt wurde. Das bedeutet, dass innerhalb einer Spawn & Merge basierten Anwendung alle Berechnungen und Darstellungen deterministisch sind, obwohl Programmabläufe innerhalb des Spawn & Merge Frameworks nichtdeterministisch sein dürfen. Dadurch sollen *Geschwindigkeitseinbußen reduziert* werden, die sich durch Wartezeiten an Synchronisations-

punkten ergeben würden, die für die deterministische Ausführung auf Applikationsebene nicht notwendig sind. Als dritte Anforderung soll das Programmiermodell bestehende General Purpose Languages (GPLs) um nur wenige Synchronisationsprimitive erweitern, sodass nur *wenig zusätzliche Komplexität* für Entwickler eingeführt wird. Im Folgenden werden diese Anforderungen sowie die Lösungsansätze, die das Programmiermodell verfolgt, genauer betrachtet.

4.1.1 Standardmäßiger Determinismus

Wenn eine nebenläufige Anwendung garantiert deterministisch ausgeführt wird, dann ergeben sich daraus mehrere Vorteile sowohl für den Entwicklungsprozess als auch für die Nutzung.

Zum einen bedeutet die garantiert deterministische Ausführung, dass die gleiche Eingabe in die Anwendung immer zur gleichen Ausgabe führt. Dieser Determinismus soll dabei unabhängig von der Ausführungsumgebung sein. Das heißt, es sollen keine neuen Race-Conditions (und somit nichtdeterministisches Verhalten) in die Anwendung eingeführt werden, falls sich die Anzahl oder Taktung der Prozessoren oder die Anzahl der verfügbaren Knoten in einem Rechencluster ändert (siehe Kapitel 3.1.1). Für den Entwickler wird somit automatisch sichergestellt, dass eine fehlerfreie Ausführung der Anwendung auf einem Testsystem gleichzeitig bedeutet, dass die Anwendung auch auf anderen Ausführungsumgebungen fehlerfrei und deterministisch ausgeführt wird. Das erleichtert die Entwicklung von Anwendungen, die später auf einem sich von der Entwicklungsumgebung stark unterscheidenden System ausgeführt werden sollen (z.B. auf einem anderen Mehrkernprozessor, einem Verteilten System oder einem Supercomputer (Massively Parallel Processing)).

Zum anderen bedeutet eine deterministische Ausführung, dass sich Fehler und das Verhalten[1] der Anwendung auch im Nachhinein deterministisch reproduzieren lassen. Tritt dementsprechend bei der Entwicklung der Anwendung ein Fehler auf, so kann es sich nicht um eine Race-Condition (die in der Regel aufwändig zu reproduzieren und zu beheben sind) handeln. Der Fehler wird bei jeder Ausführung der Anwendung auftreten und lässt sich somit zurückverfolgen. Falls ein Fehler bei der Ausführung auf einer anderen Ausführungsumgebung aufgetreten ist, dann kann der Fehler vom Entwickler bei gleicher Eingabe in die Anwendung auf seinem lokalen System nachvollzogen werden. Dadurch wird die Fehleranalyse und Fehlerbehebung von komplexen nebenläufigen Anwendungen erleichtert.

Wie in Kapitel 2.1.1 beschrieben, ist es nicht trivial, deterministische nebenläufige An-

[1] Im Folgenden wird vornehmlich über die Reproduktion von Fehlern gesprochen. Ein auffälliges oder fehlerhaftes Verhalten der Anwendung (z.B. im Rahmen einer Simulation) kann analog reproduziert und nachvollzogen werden.

wendungen mit herkömmlichen Synchronisationsmechanismen zu entwickeln. Viele Programmiermodelle, die auf deterministische Anwendungen spezialisiert sind, beschränken sich entweder auf Anwendungen mit gemeinsam genutztem Arbeitsspeicher oder im verteilten Fall auf die exakte Reproduktion eines Programmablaufs (voll-deterministisch oder die deterministische Wiederholung), womit ein hoher Performanceverlust bei einem Wechsel der Ausführungsumgebung einhergeht (siehe Kapitel 2.2.2 und 2.2.3).

Um skalierende, standardmäßig deterministische Anwendungen unabhängig von der Ausführungsumgebung zu ermöglichen, werden bei Spawn & Merge neue Synchronisationsprimitive eingeführt. Diese erlauben es dem Entwickler, nebenläufig ausgeführte Funktionen zu starten, die isoliert ausgeführt werden. Dazu werden die an die Funktionen übergebenen Argumente vom Framework automatisch kopiert. Das Kopieren der Argumente ermöglicht neben der Isolation der Funktionen auch die Übertragung des Programmiermodells auf Verteilte Systeme ohne einen gemeinsam genutzten Arbeitsspeicher. Nach erfolgreicher Ausführung der Funktion übernimmt das Framework die deterministische Zusammenführung der veränderten Kopien der Argumente. Die Synchronisation zwischen Prozessen wird explizit über die neuen Synchronisationsprimitive angestoßen. Somit hat der Entwickler die Kontrolle darüber, zu welchem logischen Zeitpunkt eine Synchronisation (und somit auch eine Beeinflussung geteilter Datenstrukturen) stattfindet. Dieser Zeitpunkt ist durch den Quellcode der Anwendung spezifiziert und somit unabhängig vom Grad der Parallelität, der durch die Anzahl der Prozessorkerne der Ausführungsumgebung bestimmt wird.

Das Programmiermodell garantiert, dass der auf Applikationsebene sichtbare Programmablauf deterministisch ist. Der sichtbare Programmablauf umfasst dabei die Menge aller aufgerufenen Funktionen zusammen mit deren Eingaben und Ausgaben. Die Determinismusgarantie gilt allerdings nur unter den folgenden Annahmen:

1. Die Synchronisation zwischen nebenläufig ausgeführten Funktionen geschieht ausschließlich über die von Spawn & Merge bereitgestellten Synchronisationsprimitive.

2. Die nebenläufig ausgeführten Funktionen beinhalten keine von Haus aus nichtdeterministischen Aufrufe (z.B. Abhängigkeiten von der Systemzeit).

3. Die Isolation nebenläufig ausgeführter Funktionen wird nicht umgangen (z.B. durch Call by Reference Parameter).

Neben der Anforderung, dass mit Spawn & Merge entwickelte Anwendungen standardmäßig deterministisch sein sollen, gibt es auch Szenarien, in denen nichtdeterministisches Verhalten für eine Anwendung sinnvoll ist. Das ist zum Beispiel immer dann der Fall, wenn die Anwendung auf nichtdeterministische Eingaben von außerhalb der Anwendung reagieren soll (z.B. auf Netzwerkkommunikation oder Nutzerinteraktionen). Spawn

& Merge bietet zu diesem Zweck zusätzliche, nichtdeterministische Synchronisationsprimitive an, die es dem Entwickler erlauben explizit zwischen einem deterministischen und einem nichtdeterministischen Programmablauf zu wählen. Dieses Vorgehen ist das exakte Gegenteil zu den etablierten Synchronisationsmechanismen, bei denen die Anwendungen standardmäßig nichtdeterministisch sind und der Entwickler zusätzlichen Implementierungsaufwand betreiben muss, um eine deterministische Ausführung zu gewährleisten (siehe Kapitel 2.2).

4.1.2 Reduktion von Synchronisationspunkten

Ein Punkt, der die voll-deterministische Ausführung einer Anwendung verlangsamt, ist die erzwungene Einhaltung und Reproduktion jedes einzelnen Synchronisationspunktes zwischen nebenläufigen Prozessen der Anwendung (siehe Kapitel 3.2). Bei der Konzeption von Spawn & Merge sollten die Geschwindigkeitsverluste, die sich durch die Einführung des Determinismus in die Anwendung ergeben, möglichst geringgehalten werden.

Um dies zu erreichen, nutzt Spawn & Merge das Konzept des Determinismus auf Applikationsebene, das in Kapitel 3.3 eingeführt wurde. Spawn & Merge baut dabei auf der Erkenntnis auf, dass es für ein deterministisches Ergebnis der Anwendung nicht zwingend erforderlich ist, dass jeder Teilbereich der Anwendung voll-deterministisch ausgeführt wird (siehe Kapitel 3.2.1). Die neu eingeführten Synchronisationsprimitive bilden hierbei die klar definierte Schnittstelle zwischen dem deterministisch ablaufenden Programmfluss, den der Entwickler vorgibt (Applikationsebene), und dem nichtdeterministischen Programmfluss innerhalb des Spawn & Merge Frameworks (Frameworkebene). Durch diese Aufteilung der Anwendung sollen Synchronisationspunkte, die für das deterministische Ergebnis nicht zwingend erforderlich sind, eingespart werden. Durch das Kopieren der Argumente für die nebenläufig ausgeführten Prozesse werden des Weiteren die Prozesse voneinander entkoppelt, wodurch weitere Synchronisationspunkte eingespart werden. Somit werden Wartezeiten zwischen den nebenläufig ausgeführten Prozessen eingespart.

Der Reduktion von Synchronisationspunkten stehen die Kosten für das Kopieren der Datenstrukturen, sowie die Berechnungskomplexität für die genutzten Mechanismen zur deterministischen Zusammenführung von geteilten Datenstrukturen gegenüber. Diese Mechanismen können sehr aufwändig zu berechnen sein. In Kapitel 7 wird anhand unterschiedlicher Szenarien mit Hilfe einer prototypischen Implementierung des Spawn & Merge Frameworks der Einfluss der Kosten für das Kopieren und die Zusammenführung der geteilten Datenstrukturen auf die Laufzeit untersucht. An dieser Stelle soll noch einmal herausgestellt werden, dass das oberste Optimierungsziel für Spawn & Merge die Erstellung *korrekter* nebenläufiger Anwendungen ist und nicht die vollständige Ausschöpfung der möglichen parallelen Performance der Anwendungsumgebung.

4.1.3 Entwicklerfreundlichkeit

Die Entwicklerfreundlichkeit ist ein wichtiger Aspekt für die Einführung eines neuen Programmiermodells. Wenn ein neues Programmiermodell zu viel zusätzliche Komplexität mit sich bringt, dann kann es passieren, dass das ursprüngliche Problem nicht gelöst, sondern durch ein neues komplexes Problem ersetzt wird. Einige der in Kapitel 2.2 beschriebenen Ansätze zur Entwicklung deterministischer, nebenläufiger Anwendungen setzen beispielsweise Quelltext-Annotationen oder die Umsetzung anderer komplexer Mechanismen durch den Entwickler voraus. Im Falle von *Deterministic Parallel Java (DPJ)* [15] bedeutet das zum Beispiel, dass für nebenläufig ausführbare Codebereiche korrekt markiert werden muss, auf welche anderen Codebereiche zugegriffen wird, damit eine deterministische Ausführung erreicht werden kann.

Um die Programmierung nebenläufiger Anwendungen für den Entwickler zu vereinfachen, wird bei Spawn & Merge ein dazu inverser Ansatz verfolgt (siehe Kapitel 4.1.1). Bei der Verwendung der von Spawn & Merge bereitgestellten (deterministischen) Synchronisationsprimitive wird dem Entwickler die Garantie[2] gegeben, dass der Programmablauf auf Applikationsebene deterministisch sein wird. Falls der Entwickler explizit an einer Stelle ein nichtdeterministisches Verhalten benötigt, kann dieses durch die explizite Verwendung einer der nichtdeterministischen Synchronisationsprimitive erreicht werden.

Um die durch das Programmiermodell eingeführte Komplexität gering zu halten, werden nur wenige neue Synchronisationsprimitive eingeführt. Auch bei Spawn & Merge gibt es Komponenten, die mit einer hohen Berechnungskomplexität aufwarten. Insbesondere sind das die Datenstrukturen, die modifiziert werden müssen, damit diese bei der Verwendung der Synchronisationsprimitive automatisch kopiert und anschließend wieder automatisch zusammengeführt werden können. Abhängig von der Datenstruktur können entsprechende Mechanismen aufwändig in der Umsetzung sein. Diese Komplexität wird allerdings dadurch relativiert, dass einmal implementierte Datenstrukturen in anderen Anwendungen, die auf Spawn & Merge basieren, wiederverwendet werden können.

4.2 Spawn & Merge Programmiermodell

Um eine deterministische Ausführung einer nebenläufigen Anwendung zu erreichen, führt das hier konzipierte Programmiermodell zwei neue Synchronisationsprimitive *Spawn* und *Merge* ein. `Spawn` ist vergleichbar mit dem UNIX Systemaufruf `fork` [92], d.h. `Spawn` startet (im Folgenden auch *„spawnt“*) eine Funktion als einen neuen sogenannten *Task*. Tasks werden nebenläufig[3] ausgeführt und arbeiten auf ihrer eigenen lokalen Kopie der Eingabedaten. Der neu gespawnte Task wird als *Child-Task* des aufrufenden *(Parent-)Tasks*

[2] Unter Einhaltung der drei in Kapitel 4.1.1 genannten Annahmen.
[3] Nebenläufig zu allen anderen Tasks der Anwendung.

bezeichnet.

Ab einem gewissen Zeitpunkt wartet der Parent-Task darauf, dass die von ihm gestarteten Child-Tasks mit ihren Berechnungen fertig werden. Anschließend führt er die Ergebnisse seiner Child-Tasks (d.h. deren Veränderungen an den kopierten Datenstrukturen) mit seinen eigenen Ergebnissen, unter Verwendung der `Merge`-Primitive, deterministisch zusammen[4]. In UNIX gibt es kein Gegenstück zu `fork`, das mit der `Merge`-Primitive vergleichbar ist. Das liegt daran, dass ein Betriebssystem zwar Speicherbereiche für einen nebenläufig ausgeführten Prozess kopieren, nicht aber die durch die Child-Tasks veränderten Kopien wieder mit dem eigenen Speicherbereich zusammenführen (im Folgenden auch „mergen") kann. Für das Zusammenführen der divergenten Kopien ist anwendungsspezifisches Wissen notwendig, das dem Betriebssystem fehlt.

Die `Merge`-Primitive ist dabei der Schlüssel zur Einführung des Determinismus auf Applikationsebene. Hier werden die nebenläufigen Veränderungen in einer deterministischen Reihenfolge zusammengeführt. Die Reihenfolge ergibt sich durch die Position der `Merge`-Primitive im Quelltext und ist somit unabhängig von der Parallelität der Anwendung und der Ausführungsgeschwindigkeit. Wie genau die Primitive `Spawn` und `Merge` arbeiten, welche Art von Parallelisierung sich mit dem Spawn & Merge Programmiermodell umsetzen lässt und welche weiteren Primitive das Programmiermodell einführt, wird in den folgende Kapiteln beschrieben.

4.2.1 Art der Parallelisierung

Es gibt zwei verbreitete Arten für die Parallelisierung von Anwendungen[5] (auf Threadebene) die unterschieden werden: *Datenparallelität* und *Taskparallelität*[6] [7].

Datenparallelität

Der Aufbau einer datenparallelen Anwendung am Beispiel eines Arrays mit n Elementen ist in Abbildung 4.1 dargestellt. Bei einer datenparallelen Anwendung [50] werden die Eingabedaten, die der Berechnung zugrunde liegen, zu gleichen Teilen auf die einzelnen Rechenknoten aufgeteilt. Jeder Knoten berechnet dieselbe Funktion (hier „Task 1") parallel auf dem ihm zugewiesenen Segment der Daten. Anschließend werden die Ergebnisse der einzelnen Berechnungen wieder zusammengeführt. Eine datenparallele Anwendung hat dadurch, dass alle Knoten die selbe Berechnung auf einem Datensegment derselben Größe durchführen, einen besonderen Vorteil: Die Berechnungen der unterschiedlichen Knoten

[4]Eine detaillierte Beschreibung der Mechanismen für die Zusammenführung von geteilten Datenstrukturen folgt in Kapitel 5.

[5]Im Bereich des Machine-Learning/Deep-Learning gibt es noch den Ansatz der *Modellparallelität* [31]. Dabei wird ein zu lernendes Modell auf verschiedene Ressourcen verteilt, damit nicht jede Ressource das gesamte Modell vorhalten muss. Auf diese Art der Parallelisierung wird hier nicht eingegangen.

[6]In der Literatur auch *Kontrollparallelität* oder *Funktionsparallelität [7].*

dauern alle gleich lange (hier Δt). Dies bedeutet, dass automatisch (für eine entsprechende Ausführungsumgebung) ein optimales Scheduling erreicht wird.

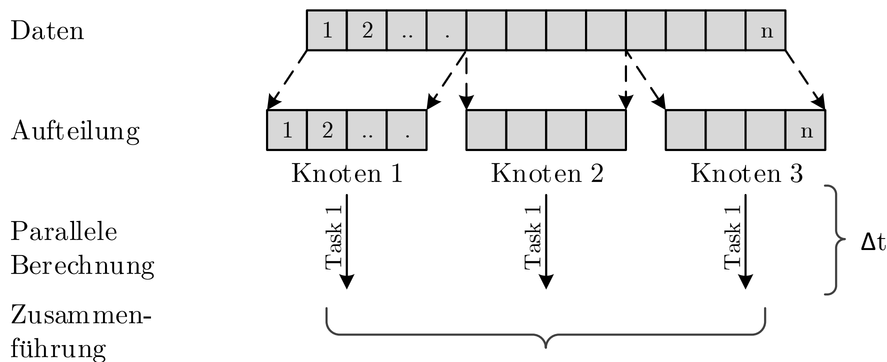


Abbildung 4.1: Aufbau einer datenparallelen Anwendung.

Taskparallelität

Der Aufbau einer taskparallelen Anwendung ist in Abbildung 4.2 dargestellt. Im Gegensatz zu einer datenparallelen Anwendung gibt es bei einer taskparallelen Anwendung [8] keine fest vorgegebene Aufteilung der Eingabedaten. Auch gibt es nicht nur eine Funktion, die von den Rechenknoten berechnet wird. Stattdessen können unterschiedliche Funktionen („Tasks“) nebenläufig ausgeführt werden, die auf (möglicherweise gemeinsam genutzten) Teilen der Eingabedaten arbeiten können. Die Anzahl der Tasks, die gestartet werden können, ist dabei nicht auf die Anzahl der verfügbaren Rechenknoten beschränkt. Das bedeutet, dass die Tasks zur Laufzeit auf verfügbare Rechenknoten verteilt werden müssen (Scheduling). Tasks können außerdem weitere (Sub-)Tasks starten, um ihre eigenen durchzuführenden Berechnungen weiter zu parallelisieren. Nachdem alle Tasks berechnet wurden, werden auch hier die Ergebnisse wieder zusammengeführt. Da die Tasks unterschiedliche Funktionen berechnen, unterscheidet sich deren Laufzeit. Das führt dazu, dass die Ausführungszeit der Anwendung abhängig vom Scheduling der Tasks auf die zur Verfügung stehenden Knoten ist.

Grundsätzlich lässt sich auch eine datenparallele Anwendungen mit einem taskparallelen Modell umsetzen. So können bei einer taskparallelen Anwendung beliebige Funktionen (d.h. auch mehrfach dieselbe) auf beliebigen Eingabedaten (d.h. auch auf einem exklusiven Segment) ausgeführt werden. Hier können allerdings nicht zwingend die Vorteile einer datenparallelen Anwendung genutzt werden (z.B. optimale Nutzung der Ressourcen der Ausführungsumgebung).

Das Spawn & Merge Programmiermodell dient der Entwicklung von taskparallelen Anwendungen. Die bereitgestellten Synchronisationsprimitive erlauben es dem Program-

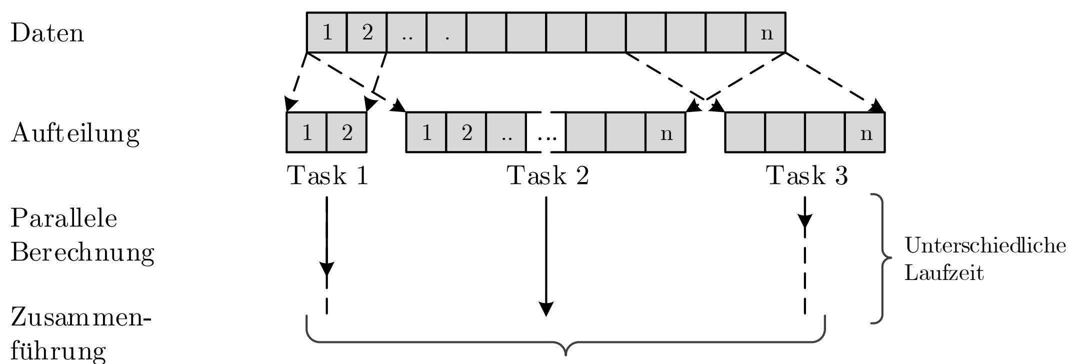


Abbildung 4.2: Aufbau einer taskparallelen Anwendung.

mierer unterschiedliche Funktionen mit gemeinsam genutzten (oder exklusiv genutzten) Daten nebenläufig auszuführen. Anschließend können die Ergebnisse wieder deterministisch zusammengeführt werden.

4.2.2 Use-Case Beschreibung

Das Spawn & Merge Programmiermodell soll die Entwicklung deterministischer nebenläufiger Anwendungen vereinfachen, die auf geteilten Datenstrukturen arbeiten. Hierbei soll neben der reinen Parallelisierung der Anwendung auch die Synchronisation zwischen nebenläufigen Funktionen (d.h. die Übergabe und das deterministische Zusammenführen geteilter Datenstrukturen) für den Entwickler vereinfacht werden. In diesem Kapitel wird anhand eines Beispielszenarios beschrieben, wie sich dieses mit dem Spawn & Merge Programmiermodell umsetzen lässt. Anschließend werden die einzelnen Komponenten des Programmiermodells im Detail vorgestellt.

Straßenverkehrssimulation

Als Anwendungsbeispiel sei eine Straßenverkehrssimulation gegeben, mit deren Hilfe ein neues Fahrerassistenzsystem (FAS) getestet werden soll. Damit ein FAS intensiv getestet und die ordnungsgemäße Funktion sichergestellt werden kann, ist es notwendig, mehrere Millionen Testkilometer mit dem FAS zu fahren [9]. Die Nutzung von Simulationen für das Absolvieren der Testkilometer sind eine Möglichkeit, die Entwicklung und Validierung eines FAS zu vereinfachen [9]. Simulationen mit vielen Komponenten, die miteinander interagieren, sind komplex und aufwändig zu berechnen. Um trotz der Berechnungskomplexität eine annehmbare Ausführungsgeschwindigkeit zu erreichen, soll das Beispielszenario parallelisiert werden. Dazu wird das Gesamtproblem in kleinere Teilprobleme aufgeteilt, die anschließend parallel gelöst werden können.

Agentenbasierte Simulation

Es gibt unterschiedliche Möglichkeiten, eine entsprechende Verkehrssimulation zu parallelisieren. Eine Möglichkeit ist ein agentenbasierter Ansatz zur Zerlegung der Simulation in kleinere Teile [86]. Dieser Ansatz wird in Abbildung 4.3 beispielhaft dargestellt. Dabei wird jeder einzelne Agent in der Simulation (hier ein einzelnes Fahrzeug und dessen FAS im Straßenverkehr) für sich selbst gesehen simuliert. Jeder Agent benötigt zur Simulation seiner Entscheidungen Informationen über andere Agenten und die Straßen in seiner näheren Umgebung. Aus diesem Grund sind viele Synchronisationen (hier durch Zugriffe auf geteilte Speicherbereiche) zwischen den einzelnen Agenten notwendig, wodurch die erreichbare Parallelität limitiert wird (siehe Kapitel 3.2).

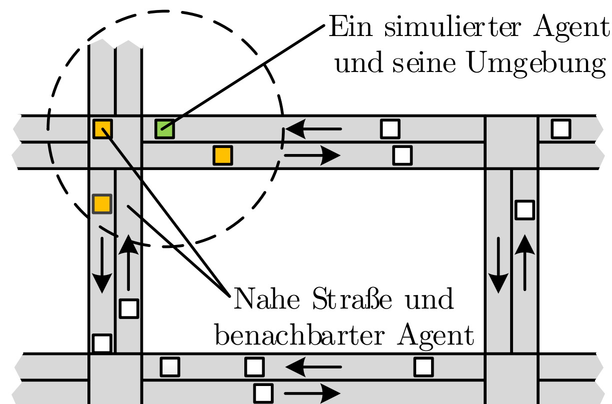


Abbildung 4.3: Agentenbasierte Aufteilung.

Simulation einzelner Straßensegmente

Eine andere Möglichkeit, die Verkehrssimulation in kleinere Teile zu zerlegen, ist die Aufteilung des Straßennetzes in Straßensegmente, die parallel simuliert werden können [70]. Abbildung 4.4 zeigt beispielhaft eine Zerlegung des Straßennetzes anhand von Straßenkreuzungen und Verkehrsampeln. Die Simulation kann nebenläufig für jedes Straßensegment und die dort befindlichen Fahrzeuge und deren FAS durchgeführt werden. Die Synchronisationspunkte beschränken sich dabei auf zwei Punkte. Erstens auf den Informationsaustausch mit den direkten Nachbarsegmenten des simulierten Straßensegmentes, wenn ein simuliertes Fahrzeug auf ein benachbartes Segment oder von einem benachbarten Segment auf das eigene Segment wechselt. Zweitens ist eine zeitliche Synchronisation notwendig, um sicherzustellen, dass alle Teilsimulationen wissen, zu welchem Zeitpunkt sie welche Informationen kennen.

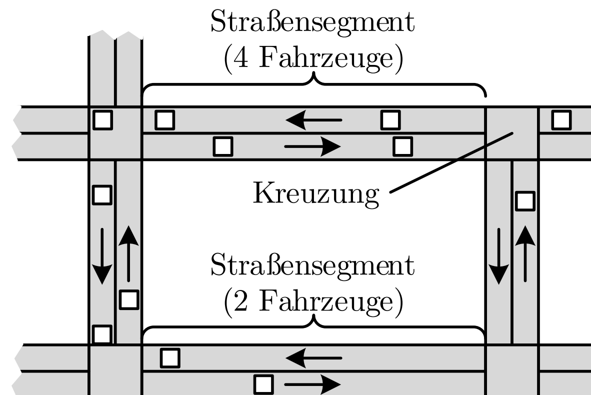


Abbildung 4.4: Aufteilung nach Straßensegmenten.

Simulation der Straßensegmente mit Spawn & Merge

Das Spawn & Merge Programmiermodell erleichtert die Entwicklung dieser Art von Anwendungen. Mit Spawn & Merge kann die Simulation der einzelnen Straßensegmente in nebenläufige Tasks ausgegliedert werden. Da jeder Task auf einer Kopie der Datenstrukturen arbeitet, ist eine Synchronisation während eines Simulationsschrittes auch dann nicht notwendig, wenn auf die Daten eines benachbarten Segmentes zugegriffen wird. So werden die Synchronisationspunkte zwischen den Simulationen der Segmente reduziert und unnötige Wartezeiten vermieden. Ein weiterer Vorteil, der sich aus dem Einsatz von Spawn & Merge ergibt, ist der deterministische und reproduzierbare Programmablauf der Simulation. Diese Eigenschaft ist im beschriebenen Szenario besonders wichtig. Ohne ein deterministisches und reproduzierbares Systemverhalten wäre es den Entwicklern des FAS nicht möglich, bei einem ungewöhnlichen Verhalten eines Fahrzeuges (oder dessen FAS) im Verlaufe der Simulation nachträglich die Ursache zu untersuchen. Das Listing 4.1 zeigt anhand eines vereinfachten Beispiels[7], wie eine Umsetzung des Beispielszenarios mit Spawn & Merge aussehen könnte.

Nachdem das Modell des Straßennetzes in einzelne Straßensegmente aufgeteilt wurde (Zeile 4), werden zu jedem Segment die mit ihm direkt verbundenen Segmente gesucht (Zeile 5). Mit jedem Durchlauf der `For`-Schleife in Zeile 7 wird ein Simulationsschritt durchgeführt. Für jedes in `segments` enthaltene Straßensegment wird ein Task zur Simulation dieses Segmentes gespawnt. Dazu wird in Zeile 10 die `Spawn`-Primitive zusammen mit der nebenläufig auszuführenden Simulationsfunktion (`SimulateSegment`), dem Straßenseg-

[7]Die Beispiele werden in einem vereinfachten C++-ähnlichen Pseudocode beschrieben. Zur Vereinfachung werden beispielsweise die Template-Parameter von Datenstrukturen, sowie die Initialisierung des Spawn & Merge Frameworks ausgelassen.

Listing 4.1 Verkehrssimulation mit Spawn & Merge.

```
1:  void SimulateSegment(Segment* self,Segment* neighbors[]);
2:
3:  void mainTask(){
4:    Segment* segments[] = InitSegments();
5:    Segment* neighbors[][] = InitAdjacentSegments();
6:
7:    for (int simStp = 0; simStp < simEnd; ++simStp){
8:      for (int i = 0; i < segments.size(); ++i){
9:        // Spawn one Task per segment
10:       Spawn(SimulateSegment, segments[i], neighbors[i]);
11:     }
12:     // Wait for completion of all Child-Tasks
13:     Merge();
14:   }
15: }
```

ment (`segment[i]`) und den mit diesem Straßensegment verbunden Nachbarsegmenten (`neighbors[i]`) aufgerufen. Bei der Erstellung des Tasks werden die übergebenen Parameter für die lokale Nutzung innerhalb des Tasks automatisch kopiert. Um dies zu ermöglichen, müssen `Segment` Datenstrukturen für die Nutzung in Spawn & Merge angepasst sein. Das bedeutet, dass sie die notwendige Programmlogik beinhalten, um automatisch kopiert und wieder deterministisch zusammengeführt (gemerged) zu werden. Die gespawnten `SimulateSegment` Tasks werden nebenläufig ausgeführt.

Tasks, die ihre Berechnungen abgeschlossen haben, geben ihre Ergebnisse zurück an ihren Parent-Task (hier der Task der die Funktion `mainTask` in Zeile 3 ausführt). Nach dem parallel ausgeführten Simulationsschritt folgt ein Synchronisationsschritt, in dem der Wechsel von Fahrzeugen zwischen Straßensegmenten behandelt wird. Sobald im Parent-Task die `Merge`-Primitive (Zeile 13) aufgerufen wird, blockiert diese und startet den Synchronisationsschritt. Nun wartet der Parent-Task darauf, dass alle gestarteten Child-Tasks ihre Berechnungen fertigstellen und die Ergebnisse zurückgeben. Innerhalb der `Merge`-Primitive werden dabei die veränderten Kopien der Straßensegmente deterministisch mit den Straßensegmenten des Parent-Tasks zusammengeführt. Das Zusammenführen (Mergen) muss dabei ausdrücklich durch den Parent-Task angestoßen werden. Der explizite Aufruf ist essenziell, um die deterministische Ausführung auf Applikationsebene zu ermöglichen und wird in Kapitel 4.2.7 genauer beschrieben. Sobald alle Child-Tasks in den Parent-Task gemerged wurden hört die `Merge`-Primitive auf zu blockieren und die Schleife startet in Zeile 8 neu.

Die resultierende Anwendung ist deterministisch und unabhängig von der Ausführungsumgebung. Außerdem gibt es nur wenige Synchronisationspunkte zwischen den

nebenläufig ausgeführten Tasks, was einen höheren Grad an Parallelität erlaubt. Dem gegenüber stehen die Kosten für die deterministisch Zusammenführung der divergenten Kopien der Datenstrukturen. In Kapitel 7 wird der Kompromiss zwischen einem höheren Grad an Parallelität und den Kosten für die Zusammenführung von Datenstrukturen diskutiert. In den nachfolgenden Kapiteln werden die einzelnen Komponenten des Spawn & Merge Programmiermodells im Detail vorgestellt.

4.2.3 Tasks

Im Spawn & Merge Programmiermodell wird eine Funktion, die durch die `Spawn()`-Primitive zur nebenläufigen Ausführung gestartet wurde, als *Task* bezeichnet. Es wird der Begriff „Task" anstelle von „Thread" verwendet, da es nicht zwingenderweise eine Eins-zu-eins-Zuordnung zwischen Threads und Tasks gibt. Das ist darin begründet, dass Tasks beispielsweise auch (abhängig von der Implementierung des Spawn & Merge Frameworks) auf einem Thread-Pool ausgeführt werden könnten. Wird ein neuer Task gespawnt, so erhält er eine Kopie der übergebenen Parameter. Das erlaubt eine *seiteneffektfreie*, nebenläufige Ausführung der Tasks, da jeder Task auf seiner eigenen lokalen Datenkopie arbeiten kann, ohne dass eine Synchronisation mit anderen Tasks notwendig ist.

Task-Informationen und Task-Hierarchie

Jeder Task beinhaltet Spawn & Merge spezifische Informationen, die für das Erreichen des deterministischen Programmablaufes notwendig sind. Um die Beziehung eines Tasks zu anderen Tasks rekonstruieren zu können, speichert ein Task sowohl eine Verknüpfung[8] zu seinem Parent-Task (gilt nicht für den Main-Task, der keinen Parent-Task besitzt) als auch eine Verknüpfung zu den von ihm gespawnten Child-Tasks. Für die Child-Tasks wird zusätzlich die Spawn-Reihenfolge festgehalten, da diese für die Funktionalität der `Merge`-Primitive notwendig ist. Ebenso werden in einer Liste namens `finishedChildren` alle Child-Tasks zwischengespeichert, die bereits ihre Berechnungen durchgeführt und ihre Ergebnisse zurückgegeben haben, aber noch nicht gemerged wurden. Abschließend wird gespeichert, mit welchen zusammenführbaren Datenstrukturen der Task gestartet wurde. Die Kenntnis dieser sogenannten `startingStructures` ist notwendig, damit der Task bei seiner Fertigstellung prüfen kann, welche Datenstrukturen an seinen Parent-Task zurückgesendet werden müssen. Tabelle 4.1 fasst die gespeicherten *„Task-Informationen"* und deren Zweck zusammen.

Betrachtet man die Struktur, die sich aus den gespeicherten Parent-Task zu Child-Tasks Relationen ergeben, dann ergibt sich eine baumartige *Task-Hierarchie* wie sie in Abbildung

[8] Abhängig von der Implementierung des Frameworks kann es sich bei dieser Verknüpfung beispielsweise um eine Referenz auf das Objekt oder eine eindeutige ID handeln.

Zugehörigkeit	Gespeicherte Information
Task-Hierarchie	Parent-Task Referenz Child-Tasks Referenzen `startingStructures`
Zeitliche Entwicklung	Spawn-Reihenfolge `finishedChildren`

Tabelle 4.1: In einem Task gespeicherte Informationen (Task-Informationen).

4.5 beispielhaft dargestellt ist. Die baumartige Struktur ergibt sich daraus, dass jeder Task genau einen Parent-Task hat (die Ausnahme bildet hierbei der Main-Task). Ein Task wiederum kann beliebig viele (Child-)Tasks spawnen, um seine eigenen Berechnungen weiter zu parallelisieren. Die neu gespawnten Tasks werden dabei wieder als Child-Tasks unter ihm in der Hierarchie eingeordnet. Somit ist die Tiefe der Task-Hierarchie (konzeptionell) nicht begrenzt.

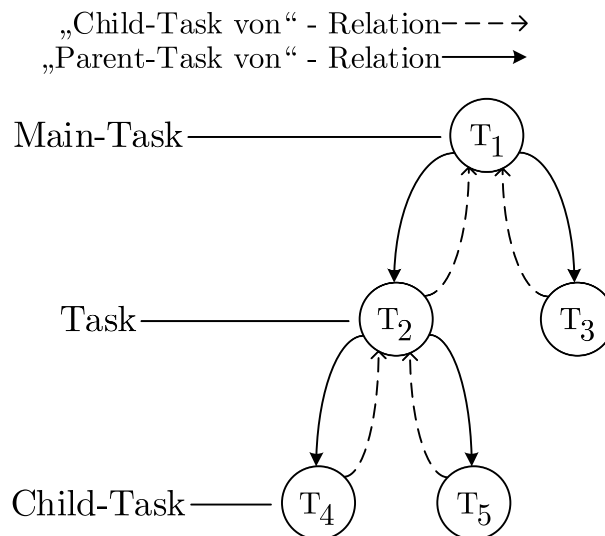


Abbildung 4.5: Task Hierarchie.

Task-Terminierung

Ein Task gilt erst dann als fertig berechnet, wenn alle seine Child-Tasks fertig berechnet sind und deren Ergebnisse mit seinen lokalen Ergebnissen zusammengeführt wurden. Um dies zu gewährleisten, wartet jeder Task nach der Berechnung der Funktion, mit der er gespawnt wurde, noch einmal final auf alle Child-Tasks, die bisher noch nicht

gemerged wurden[9]. Erst wenn alle eigenen Child-Tasks gemerged wurden, gilt der Task als fertig berechnet und übergibt seine Ergebnisse seinem Parent-Task (d.h. er trägt sich selbst in dessen `finishedChildren` Liste ein). Dadurch wird ebenfalls sichergestellt, dass der Main-Task erst dann abgeschlossen ist, wenn alle gespawnten Tasks gemerged wurden.

Task-Handles

Ein Task ist ein komplexes Objekt, das direkt an Framework-Interna gekoppelt ist. Des Weiteren kann nicht garantiert werden, dass sich ein Task Objekt nach dem Spawnen noch im Scope des spawnenden Tasks befinden wird. Das liegt daran, dass, abhängig von der Implementierung des Frameworks sowie der Ausführungsumgebung, ein Task auf einem anderen Computer ausgeführt werden kann. Daher wird ein Task im Code durch ein *Task-Handle* repräsentiert, das dem Entwickler eine wohldefinierte Schnittstelle für den Zugriff auf das Task Objekt bietet. Dieses Task-Handle kapselt dabei den Zugriff auf einen Task, unabhängig davon, ob der Task auf demselben Rechenknoten ausgeführt wird wie der eigene Task, oder nicht[10]. Über ein Task-Handle kann der Entwickler den Status eines Tasks überprüfen (d.h. prüfen ob der Task noch läuft, ob er abgebrochen wurde oder ob er fertiggestellt wurde). Falls die Ergebnisse eines Tasks nicht mehr benötigt werden, bietet das Task-Handle zusätzlich die Möglichkeit den Task abzubrechen (näheres zur `Abort` Funktionalität folgt in Kapitel 4.2.11).

4.2.4 Initialisierung des Frameworks

Jede Spawn & Merge basierte Anwendung beginnt mit der Initialisierung des Frameworks durch `InitSpawnMerge`. Bei der Initialisierung wird der erste Task, der sogenannte *„Main-Task“* gestartet. Die abstrakte Schnittstellendefinition für `InitSpawnMerge` ist in Listing 4.2 beschrieben[11].

Listing 4.2 Definition der *InitSpawnMerge* Schnittstelle.

```
void InitSpawnMerge(function)
```

`InitSpawnMerge` erwartet als *`function`* Argument eine parameterlose Funktion ohne Rückgabewert (`(void) function(void)`), die als `main`-Methode der Spawn & Merge basierten Anwendung angesehen werden kann. In Listing 4.3 wird gezeigt, wie unter Verwendung von `InitSpawnMerge` ein Main-Task gestartet wird, der die Funktion `mainTask` ausführt. Dieser Main-Task ist ein vollwertiger Task, der Child-Tasks spawnen

[9] Dies geschieht durch den Aufruf der `MergeAll`-Primitive (siehe Kapitel 4.2.7) mit dem Parameter `TILL_ALL_FINISHED` (siehe Kapitel 4.2.9) die im Folgenden erst noch vorgestellt werden.

[10] Eine vollständige Auflistung der Task-Handle Funktionalität kann in Anhang A.1 eingesehen werden.

[11] In Kapitel 4.3 wird für C++ beschrieben, wie die Synchronisationsprimitive konkret umgesetzt werden können.

und die Ergebnisse mit seinen eigenen Daten wieder zusammenführen kann. Innerhalb des Main-Tasks kann der Entwickler somit die im Folgenden vorgestellten Synchronisationsprimitive nutzen, um eine deterministische Anwendungen zu entwickeln. Der Aufruf von `InitSpawnMerge` blockiert dabei solange, bis der Main-Task abgeschlossen ist, d.h. bis die gesamte Task-Hierarchie abgearbeitet und gemerged wurde.

Listing 4.3 Initialisierung des Spawn & Merge Frameworks.

```
1: void mainTask(){
2:   // Start of application-level deterministic execution
3: }
4:
5: void main(int argc, const char* argv[]){
6:   // Initialization of Spawn & Merge Framework
7:   InitSpawnMerge(mainTask);
8: }
```

4.2.5 Zusammenführbare Datenstrukturen

Im Spawn & Merge Programmiermodell werden Datenstrukturen, die von Tasks genutzt werden, kopiert. Damit wird verhindert, dass die nebenläufigen Tasks untereinander kommunizieren müssen und somit eine große Anzahl an Synchronisationspunkten erzeugen. Dieser Ansatz orientiert sich dabei an der Beobachtung, dass viele Probleme, die sich bei der Entwicklung paralleler Anwendungen ergeben, verschwinden, wenn jeder nebenläufig ausgeführte Task seine eigenen Datenstrukturen besitzt (siehe z.B. durch `fork` erstellte nebenläufige Prozesse). Um zu erreichen, dass Datenstrukturen vom Spawn & Merge Framework kopiert und anschließend wieder deterministisch zusammengeführt werden können, müssen die Datenstrukturen um Spawn & Merge spezifische Funktionalität erweitert werden, die im Folgenden beschrieben werden. Datenstrukturen, die diese Anforderungen erfüllen, werden auch als *„zusammenführbare Datenstrukturen"* bezeichnet. In Code-Beispielen werden diese auch als Spezialisierungen des Typs `Mergeable` abgebildet.

Die Erweiterungen zusammenführbarer Datenstrukturen dienen zwei Zielen: Die Datenstruktur soll durch das Framework automatisch (d.h. ohne das Zutun des Entwicklers) *kopierbar* sein. Der Prozess des Kopierens beschränkt sich dabei nicht nur auf ein reines Abbild der Datenstruktur, da zusätzliche Informationen gespeichert werden müssen, die für die spätere deterministische Zusammenführung notwendig sind. Das zweite Ziel ist das Ermöglichen der *deterministischen Zusammenführung* von Datenstrukturkopien. Das bedeutet insbesondere, dass eine zusammenführbare Datenstruktur einen Mechanismus zur Zusammenführung (und insbesondere zur deterministischen Konfliktauflösung) bereitstellen muss. Eine tabellarische Auflistung aller notwendigen Funktionalitäten einer

zusammenführbaren Datenstruktur[12] kann in Anhang A.2 eingesehen werden.

Datenstruktur-Historie (Φ)

Beim Kopieren einer Datenstruktur in Spawn & Merge werden Informationen über die Relationen zwischen der kopierten Datenstruktur sowie der erstellten Kopie gespeichert. Diese Informationen umfassen dabei zum einen den Zustand beider Datenstrukturen zum Kopierzeitpunkt in Form einer Versionsnummer. Diese ermöglicht es im Nachhinein festzustellen, welche Veränderungen auf den Daten seit dem Kopieren durchgeführt wurden. Außerdem wird festgehalten, welche der beiden Datenstrukturen beim spawnenden Parent-Task und welche beim gespawnten Child-Task liegt. Diese Informationen sind notwendig für die spätere Zusammenführung der divergenten Datenstrukturkopien und werden im Folgenden *„Datenstruktur-Historie“* genannt (abgekürzt: Φ), da sie die Relationen zwischen einzelnen Kopien einer Datenstruktur und die durchgeführten Veränderungen verfolgen. Jede Datenstruktur D speichert dabei ihre eigene Datenstruktur-Historie Φ_D. Tabelle 4.2 zeigt die in der Datenstruktur-Historie Φ enthaltenen Informationen, die durch die Einführung neuer Synchronisationsprimitive im Folgenden noch erweitert werden[13].

Zugehörigkeit	Gespeicherte Information
Datenstruktur-Hierarchie	Parent-Task Datenstruktur Referenz Kopien der eigenen Datenstruktur
Änderungsverfolgung	Versionsnummer beim Kopieren Eigene aktuelle Versionsnummer Durchgeführte Veränderungen

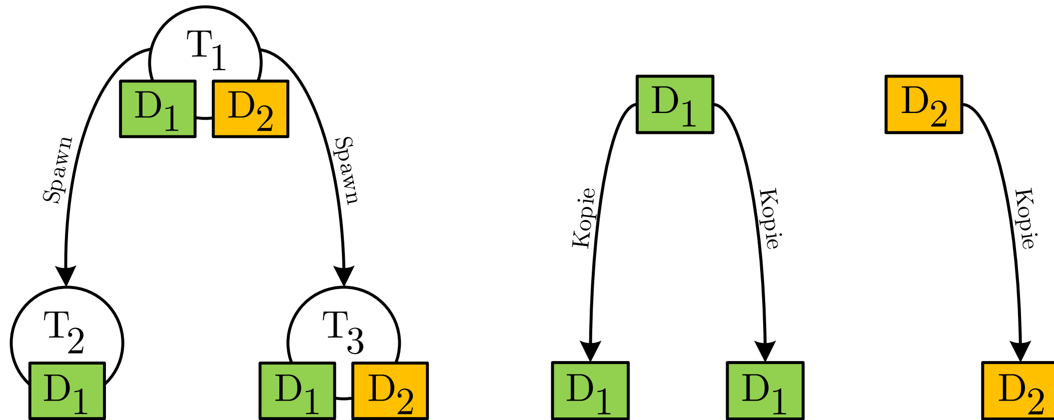
Tabelle 4.2: In der Datenstruktur-Historie Φ gespeicherte Informationen.

Durch das Kopieren der Datenstrukturen beim Spawnen neuer Tasks ergibt sich für die Abhängigkeiten zwischen einzelnen Kopien eine baumartige *Datenstruktur-Hierarchie*, die in Φ in der Referenz auf die Datenstruktur des Parent-Tasks und die Referenzen auf die Kopien der eigenen Datenstruktur abgebildet sind. Diese Baumstruktur ist nicht notwendigerweise deckungsgleich mit der in Kapitel 4.2.3 beschriebenen Task-Hierarchie. Stattdessen bildet die Datenstruktur-Hierarchie für jede Datenstruktur einen Teilbaum der Task-Hierarchie. In Abbildung 4.6a ist zur Verdeutlichung ein Task T_1 dargestellt, der zwei zusammenführbare Datenstrukturen D_1 und D_2 beinhaltet. T_1 spawnt nun zwei Child-Tasks, zum einen T_2 mit der Datenstruktur D_1 und zum anderen T_3 mit beiden Datenstrukturen (D_1 und D_2). Abbildung 4.6b zeigt die Datenstruktur-Hierarchien, die

[12]Die tabellarische Auflistung im Anhang beinhaltet auch die Anpassungen für die im Folgenden noch vorgestellten Synchronisationsprimitive `Sync` und `SpawnSibling`.

[13]Die vollständige Tabelle kann im Anhang A.3 eingesehen werden.

sich dadurch für D_1 und D_2 ergeben. Beide Datenstruktur-Hierarchien stellen dabei einen Teilbaum der Task-Hierarchie dar.



(a) Task-Hierarchie. (b) Datenstruktur-Hierarchien.

Abbildung 4.6: Task-Hierarchie vs. Datenstruktur-Hierarchien.

Schnittstelle zu internen Mechanismen

Der Mechanismus für die deterministische Zusammenführung ist abhängig von der Datenstruktur selbst und kann somit nicht als interner Teil des Spawn & Merge Frameworks angeboten werden. Da der Mechanismus unabhängig von der entwickelten Anwendung ist, kann eine einmal implementierte, zusammenführbare Datenstruktur (z.B. eine Liste) für unterschiedliche Anwendungen wiederverwendet werden. Für die Bereitstellung des Mechanismus muss die Datenstruktur um eine Schnittstelle erweitert werden, die es dem Framework ermöglicht den Zusammenführungsmechanismus aufzurufen. Die Zusammenführung geschieht dabei immer zwischen einer Datenstruktur des Parent-Tasks und der entsprechenden Kopie der Datenstruktur eines Child-Tasks. Eine Zusammenführung zwischen zwei Geschwistern (d.h. Child-Tasks des selben Parent-Tasks) kann es nicht geben, da es keine Synchronisationsprimitive gibt, die zwei Child-Tasks zusammenführt. Als Schnittstelle dient hierbei die Funktion `MergeWithCopy` der Datenstruktur des Parent-Tasks, der die Datenstrukturkopie des Child-Tasks übergeben wird. Nach der Durchführung der Funktion wurden die Ergebnisse der Datenstrukturkopie des Child-Tasks deterministisch in die Datenstruktur des Parent-Tasks übernommen. Die tatsächliche Umsetzung der Schnittstelle ist dabei implementierungsabhängig. Mechanismen für die deterministische Zusammenführung werden in Kapitel 5 vorgestellt.

4.2.6 Spawn

`Spawn` erstellt einen neuen nebenläufig ausgeführten Child-Task, der die übergebene Funktion ausführt. Die übergebenen Parameter werden dabei für den neuen Task kopiert. Somit kann eine gespawnte Funktion so betrachtet werden, als seien alle Parameter als Wertparameter übergeben worden (Call by Value), auch wenn die wirkliche Umsetzung mehr als nur das reine Kopieren beinhaltet (siehe Kapitel 4.2.5). Die gespawnten Funktionen werden anschließend nebenläufig ausgeführt. Der Aufruf von `Spawn` blockiert nicht und gibt ein Task-Handle (siehe Kapitel 4.2.3) zurück, das den neu gestarteten Task repräsentiert.

Nutzungsbeispiel

Listing 4.4 zeigt beispielhaft wie `Spawn` genutzt werden kann, um zwei Child-Tasks zu starten[14]. Beide Child-Tasks führen die Funktion `ModifyList` (Zeile 2) aus und bekommen jeweils eine Kopie derselben zusammenführbaren Liste `list` sowie unterschiedliche Integer `i` übergeben (Zeile 11 und Zeile 12). Die drei Kopien der Liste (eine im Parent-Task und eine in jedem Child-Task) werden nun nebenläufig modifiziert, d.h. der Parent-Task hängt eine 3 an (Zeile 15), während die beiden Child-Tasks jeweils den ihnen übergebenen Integer `i` anhängen (also 4 und 5 in Zeile 4).

Listing 4.4 Benutzung der *Spawn*-Primitive.

```
1:  // Function executed in parallel
2:  void ModifyList(MergeableList* mList, int i){
3:    // Modification of the list-copy within the Child-Task
4:    mList->append(i);
5:  }
6:
7:  // Create new list with elements 1 and 2
8:  MergeableList* list = new MergeableList(1,2);
9:
10: // Spawn two Child-Tasks
11: TaskHandle task1 = Spawn(ModifyList, list, 4);
12: TaskHandle task2 = Spawn(ModifyList, list, 5);
13:
14: // Modification of the list within the Parent-Task
15: list->append(3)
16:
17: [...]
```

Am Ende des Beispielcodes, wenn jeder Child-Task abgearbeitet ist, liegen drei Listen

[14] Das Beispiel wird im Laufe diese Kapitels erweitert, um die Nutzung der weiteren Synchronisationsprimitive zu demonstrieren. Das vollständige Beispiel befindet sich im Anhang B.3.

mit unterschiedlichem Inhalt vor ($[1,2,3]$, $[1,2,4]$ und $[1,2,5]$). Bei der Betrachtung des Ergebnisses das entstanden wäre, wenn die drei Prozesse direkt auf einer geteilten Liste gearbeitet hätten, fällt auf, dass in diesem Falle das Ergebnis nicht so eindeutig beschrieben werden könnte. Nach der Durchführung würde zwar immer eine Liste mit 5 Elementen vorliegen, die Reihenfolge der angehängten Elemente $3, 4$ und 5 wäre aber abhängig davon in welcher Reihenfolge die nebenläufigen Prozesse auf die Liste zugegriffen haben. Dies wiederum ist abhängig von der Anzahl der Prozessorkerne der Ausführungsumgebung und den entsprechenden Timings, die sich von Ausführung zu Ausführung unterscheiden können (Race-Condition). Das Ergebnis würde dementsprechend zwischen sechs möglichen Ergebnissen variieren[15].

Aktualisierung der Task-Informationen und der Datenstruktur-Historie Φ

Um in Kapitel 4.2.7 für die `Merge`-Primitive beschreiben zu können, wie die drei Ergebnisse wieder deterministisch zusammengeführt werden, ist es wichtig, bei `Spawn` noch einmal zu betrachten, welche Informationen intern in den Tasks und in den Datenstrukturen gespeichert werden und wie diese Informationen beim Spawnen aktualisiert werden. Der Parent-Task speichert die in Tabelle 4.1 beschriebenen Task-Informationen. Diese beinhalten insbesondere die Angabe, welche Child-Tasks er selbst gespawnt hat und in welcher Reihenfolge diese Child-Tasks gespawnt wurden (Spawn-Reihenfolge). Wird ein neuer Child-Task gespawnt, dann wird der neue Task in den Task-Informationen hinten an die Spawn-Reihenfolge angehängt. Außerdem wird gespeichert, welche zusammenführbaren Datenstrukturen diesem neuen Child-Task übergeben wurden (`startingStructures`). Datenstrukturen, die einem neu gespawnten Task übergeben und somit kopiert wurden, speichern des Weiteren in ihrer Datenstruktur-Historie Φ (siehe Tabelle 4.2) eine Referenz auf die kopierten Datenstrukturen sowie den Zustand (in Form einer Versionsnummer), in dem die Kopien erstellt wurden. Diese Informationen ergeben sich allesamt aus der Position der `Spawn`-Primitive im Quelltext und sind somit bei jeder Ausführung der Anwendung gleich.

Spawn-Primitive

Die abstrakte Schnittstellendefinition der `Spawn`-Primitive ist in Listing 4.5 zu sehen. `Spawn` gibt immer ein Task-Handle (*`handle`*) zurück, das für den Entwickler als Repräsentation des gestarteten Tasks dient. Als ersten Parameter erwartet `Spawn` die Übergabe der Funktion (*`function`*), die nebenläufig ausgeführt werden soll. Diese Funktion muss, wie in Kapitel 4.1.1 beschrieben, für sich genommen deterministisch sein, um die Determinismusgarantie zu erhalten, d.h. sie darf keine von Haus aus nichtdeterministischen Aufrufe

[15]Möglichkeiten: $[1,2,3,4,5]$, $[1,2,3,5,4]$, $[1,2,4,3,5]$, $[1,2,4,5,3]$, $[1,2,5,3,4]$ und $[1,2,5,4,3]$.

beinhalten. Der Rückgabetyp der übergebenen Funktion sollte vom Typ `void` sein, da der Austausch von Ergebnissen zwischen Tasks ausschließlich über die zusammenführbaren Datenstrukturen laufen darf. Von der Funktion über den Rückgabetypen zurückgegebene Daten werden vom Framework verworfen. Die weiteren optionalen Parameter von `Spawn` (*functionParameters*) sind die Argumente, die an die gespawnte Funktion übergeben werden. Damit übergebene Argumente von der `Spawn`-Primitive akzeptiert werden, müssen diese den folgenden Anforderungen genügen. Zuerst einmal müssen die `functionParameters` der Signatur der gespawnten Funktion entsprechen. Erlaubt sind als Parameter für gespawnte Funktionen zum einen Werteparameter (Call-by-Value), die für die Verwendung in der gespawnten Funktion ohne weiteres kopiert werden können. Veränderungen an diesen Daten werden allerdings (wie bei Werteparametern üblich) nicht an den spawnenden Task zurück übergeben. Referenzparameter (Call-by-Reference) sind nur dann erlaubt, wenn es sich bei dem referenzierten Objekt um eine zusammenführbare Datenstruktur handelt. Die zusammenführbare Datenstruktur darf dabei entweder vom aktuellen Task selbst erstellt worden oder von dessen Parent-Task übergeben worden sein. Um die Isolation zwischen den Tasks zu erhalten, wird hier durch das Framework eine Kopie der Datenstruktur erstellt und eine Referenz auf diese Kopie an die gespawnte Funktion übergeben. Die Nutzung von Referenzparametern zur Übergabe nicht zusammenführbarer Objekte ist nicht erlaubt und sollte durch das Framework unterbunden werden. Dies darf auch nicht vom Entwickler (beispielsweise durch die Übergabe einer Referenz in Form eines Pointers als Integer) umgangen werden, da dies die dritte der in Kapitel 4.1.1 beschriebenen Anforderungen verletzt und eine deterministische Ausführung der Anwendung somit nicht mehr garantiert werden kann.

Listing 4.5 Definition der *Spawn* Schnittstelle.

```
handle Spawn(function[,functionParameters...])
```

Randbedingungen

Bei der Verwendung von `Spawn` kann es zu einer Konstellation kommen, in der sich das Verhalten nicht eindeutig aus der bisherigen Spezifikation ergibt. Diese Randbedingung wird im Folgenden betrachtet und der Umgang mit dieser beschrieben. Konkret geht es um den Fall, dass der Entwickler der `Spawn`-Primitive mehrfach dieselbe zusammenführbare Datenstruktur übergibt. In Listing 4.6 wird beispielsweise die Liste `list` zweimal an die Funktion `fct` übergeben (Zeile 4). Da es sich bei den Parametern um Referenzparameter handelt, erwartet der Entwickler hier, dass innerhalb der Funktion `fct` die Datenstrukturen `list1` und `list2` dasselbe Objekt darstellen. Das bedeutet, dass der Parameter nur einmal kopiert und später auch nur einmal zusammengeführt werden darf. Dies muss im

Framework sichergestellt werden.

Listing 4.6 Randbedingung der *Spawn*-Primitive.

```
1: void fct(MergeableList* list1, MergeableList* list2);
2:
3: MergeableList* list = new MergeableList();
4: Spawn(fct, list, list);
5:
6: [...]
```

4.2.7 Merge

Die *Merge*-Primitive übernimmt in Spawn & Merge das deterministische Zusammenführen eines Tasks mit seinen gespawnten Child-Tasks. Um den Determinismus zu erreichen, geschieht die Zusammenführung mit den Child-Tasks in einer deterministischen Reihenfolge. Diese Merge-Reihenfolge wird vom Entwickler im Quellcode im Parent-Task durch die Nutzung der `Merge`-Primitive festgelegt.

Deterministische Varianten - MergeAll und MergeAllByHandle

Es gibt zwei deterministische Varianten der `Merge`-Primitive, die vom Entwickler genutzt werden können: *MergeAll* und *MergeAllByHandle*. `MergeAll` (für die abstrakte Schnittstellendefinition siehe Listing 4.7) blockiert die weitere Ausführung des aufrufenden Tasks und wartet auf die Fertigstellung *aller* Child-Tasks. Die Child-Tasks werden nun (d.h. sobald sie fertiggestellt sind) in der deterministischen Spawn-Reihenfolge aus der `finishedChildren` Liste herausgenommen und mit dem Parent-Task zusammengeführt. Die Zusammenführung zweier Tasks bedeutet dabei, dass zum einen die Kopien aller an den entsprechenden Child-Task übergebenen Datenstrukturen mit ihren jeweiligen Originalen zusammengeführt werden. Zum anderen werden die beendeten Child-Tasks aus der Liste der gestarteten Tasks (Spawn-Reihenfolge) entfernt. Durch diese Aktualisierung der gespeicherten Informationen kann der Parent-Task nachhalten, welche Child-Tasks aktuell noch laufen und noch gemerged werden müssen.

Listing 4.7 Definition der *MergeAll* Schnittstelle.

```
void MergeAll()
```

`MergeAllByHandle` erlaubt dem Entwickler das deterministische Mergen einer Teilmenge aller gestarteten Child-Tasks in einer von ihm selbst festgelegten Reihenfolge. Dazu kann der Entwickler beim Aufruf von `MergeAllByHandle` eine beliebige Anzahl an Task-Hand-

les (`taskHandles`) übergeben (siehe Schnittstellendefinition in Listing 4.8). Die entsprechenden Child-Tasks, die durch die für den variadischen Parameter `childTaskHandles` übergebenen Task-Handles repräsentiert werden, werden anschließend mit dem eigenen Task zusammengeführt. Dies geschieht dabei in der Reihenfolge, in der die Parameter an `MergeAllByHandle` übergeben wurden. Dem Entwickler wird somit eine Möglichkeit geboten, selektiv einzelne oder mehrere Tasks zu mergen, ohne auf alle Child-Tasks warten zu müssen. `MergeAll` ist folglich ein Spezialfall von `MergeAllByHandle`, bei dem alle Child-Tasks in der Reihenfolge übergeben wurden, in der sie erstellt worden sind.

Listing 4.8 Definition der *MergeAllByHandle* Schnittstelle.

```
void MergeAllByHandle([taskHandles...])
```

Die deterministische Reihenfolge der Zusammenführung der Tasks bei der Nutzung der deterministischen `Merge`-Primitive ist entscheidend für den Determinismus der Anwendung. Diese Reihenfolge, die sich rein aus dem Quelltext ergibt und unabhängig von der Ausführungsumgebung ist, sorgt dafür, dass die datenstrukturinternen Mechanismen[16] zur deterministischen Zusammenführung zweier Datenstrukturkopien ebenfalls immer gleich aufgerufen werden. Somit wird unabhängig davon, dass nicht festgelegt ist in welcher Reihenfolge die einzelnen Tasks ausgeführt werden (Frameworkebene), wieder eine Ordnung in den Programmablauf gebracht, der für den Entwickler sichtbar ist (Applikationsebene).

Bei einem Aufruf von `MergeAllByHandle` wartet der Task auf die Fertigstellung aller übergebenen Child-Tasks. Ist einer der Child-Tasks bereits fertiggestellt und wurde mit dem eigenen Task gemerged, so wird ein Fehler geworfen, um einem Deadlock durch die nicht erfüllbare Wartebedingung (Warten, dass der Task noch einmal fertiggestellt wird) vorzubeugen (siehe Listing 4.9 Zeile 6). Um diesen Umstand zu erkennen, prüft `MergeAllByHandle`, ob alle übergebenen Task-Handles zu noch laufenden und noch nicht gemergeten Tasks gehören. Die Alternative, fertiggestellte Tasks im Kontext von `MergeAllByHandle` zu ignorieren, wird nicht verwendet, da ansonsten Fehler unbeabsichtigt maskiert werden könnten. Gibt ein Entwickler beispielsweise unwissentlich explizit einen Task zum Mergen an, der bereits gemerged wurde, so wird der Task ignoriert und die Fehlersuche für den Entwickler erschwert. Falls kein Task-Handle übergeben wurde, blockiert `MergeAllByHandle` nicht und setzt die Ausführung der Anwendung fort, da es keinen Task gibt, auf den gewartet werden könnte.

[16]Eine detaillierte Beschreibung der genutzten Mechanismen zur Zusammenführung von Datenstrukturkopien folgt in Kapitel 5.

Listing 4.9 Randbedingung der *MergeAllByHandle*-Primitive.

```
1: [...]
2:
3: TaskHandle task1 = Spawn(f);
4:
5: MergeAllByHandle(task1);
6: MergeAllByHandle(task1); // Potential deadlock
```

Nichtdeterministische Varianten - MergeAny und MergeAnyByHandle

Eine nichtdeterministische Ausführung der Anwendung kann in manchen Fällen wünschenswert sein. Dies ist beispielsweise der Fall, wenn eine Anwendung mit unvorhersehbaren externen Eingaben (wie Nutzerinteraktionen oder mit Netzwerkkommunikation) umgehen soll. Ein anderer Fall ist die Entwicklung einer Anwendung, bei der das deterministische Ergebnis nicht von der Reihenfolge, in der die Child-Tasks fertig werden, abhängt. Hier würde die erzwungene Einhaltung der Reihenfolge zu unnötigen Wartezeiten führen[17]. In Kapitel 4.1.1 wurde dementsprechend die Anforderung formuliert, dass das Programmiermodell auch ein nichtdeterministisches Verhalten ermöglichen soll, wenn dieses vom Entwickler explizit gefordert wird. Zu diesem Zweck gibt es zwei weitere Varianten der `Merge`-Primitive, die explizit ein nichtdeterministisches Verhalten ermöglichen: *`MergeAny`* und *`MergeAnyByHandle`*.

`MergeAny` blockiert die Ausführung des aufrufenden Tasks und wartet auf *den ersten* Child-Task, der fertiggestellt wird[18]. Dieser Child-Task wird gemerged und anschließend in Form seines Task-Handles als Rückgabewert (*`handle`*) von `MergeAny` zurückgegeben (siehe Schnittstellendefinition in Listing 4.10). Dies gibt dem Entwickler die Möglichkeit, abhängig vom gemergeten Child-Task, in der Anwendung zu reagieren. Um potenzielle Deadlocks zu verhindern, prüft `MergeAny`, dass mindestens ein Child-Task noch nicht fertiggestellt ist und wirft einen Fehler, sollte diese Bedingung nicht erfüllt sein.

Listing 4.10 Definition der *MergeAny* Schnittstelle.

```
handle MergeAny()
```

`MergeAnyByHandle` ermöglicht das Warten auf den als ersten fertiggestellten Task aus einer Teilmenge aller gestarteten Child-Tasks. Dazu kann `MergeAnyByHandle` eine beliebige Anzahl an Task-Handles übergeben werden (siehe Schnittstellendefinition in Listing 4.11). `MergeAnyByHandle` blockiert den aufrufenden Task nun solange, bis einer der angegebenen

[17]Hier ist anzumerken, dass in diesem Falle zwar ein deterministisches Ergebnis, nicht aber ein deterministischer Programmablauf auf Applikationsebene entsteht.

[18]Falls schon fertiggestellte Child-Tasks in der Liste `finishedChildren` vorliegen, dann wird der erste Child-Task daraus ausgewählt.

Child-Tasks fertiggestellt wird. Anschließend wird dieser Child-Task gemerged und zurückgegeben (*handle*). Analog zu `MergeAny` wird auch bei `MergeAnyByHandle` geprüft, dass mindestens ein übergebener Child-Task noch nicht fertiggestellt wurde, um Deadlocks zu verhindern.

Listing 4.11 Definition der *MergeAnyByHandle* Schnittstelle.

```
handle MergeAnyByHandle([taskHandles...])
```

Dadurch, dass die Reihenfolge, in der die einzelnen Child-Tasks fertiggestellt werden, von verschiedenen Faktoren abhängt, führt das Warten auf den ersten Child-Task, der fertiggestellt wird, bewusst ein nichtdeterministisches Verhalten ein.

Nutzungsbeispiel

In Listing 4.12 wird das Code-Beispiel aus Kapitel 4.2.6 aufgegriffen und fortgeführt. Es zeigt, wie die unterschiedlichen `Merge`-Varianten genutzt werden können, um deterministisch oder nichtdeterministisch auf die Fertigstellung der Child-Tasks `task1`, `task2` und `task3` zu warten[19]. In Zeile 11 wartet die Anwendung darauf, dass alle Child-Tasks fertiggestellt werden und merged diese in der Spawn-Reihenfolge (d.h. erst `task1`, dann `task2` gefolgt von `task3`). In Zeile 14 wird explizit auf die Fertigstellung der Child-Tasks `task1` und `task2` gewartet. Der Child-Task `task3` wird nicht gemerged. Ein nichtdeterministisches Verhalten wird in den Zeilen 17 und 20 erlaubt, indem auf den nächsten fertiggestellten Child-Task gewartet wird. Dies geschieht einmal ohne eine weitere Spezifizierung (Zeile 17), und einmal mit der Angabe, dass es entweder `task1` oder `task2` sein muss (Zeile 20). Bei Verwendung der nichtdeterministischen Primitive kann der Entwickler durch die Rückgabe im Nachhinein testen, welcher der Tasks gemerged wurde (siehe Zeile 22).

4.2.8 Bedingter Merge

Es gibt Fälle, in denen es Sinn macht das Ergebnis eines Child-Tasks erst auf die Erfüllung bestimmter Bedingungen zu überprüfen, bevor es mit dem Parent-Task zusammengeführt wird. Ein Beispiel hierfür ist ein parallel ausgeführter Suchalgorithmus, bei dem jeder Child-Task auf seinem eigenen Datensegment nach dem größten Element sucht. Dessen Suchergebnis soll nur dann übernommen werden, wenn das gefundene Ergebnis auch größer als das aktuell bereits vorliegende Ergebnis ist.

Zu diesem Zweck bietet Spawn & Merge die optionale Nutzung einer *Bedingungsfunktion (Conditional-Function)* an. Die Bedingungsfunktion kann an ein Task-Handle angehängt

[19]Hierbei ist zu beachten, dass nur zu Demonstrationszwecken alle vier Varianten im Beispiel dargestellt sind. Eigentlich sind `task1`, `task2` und `task3` bereits nach dem ersten Aufruf von `MergeAll` fertiggestellt und die weiteren `Merge`-Primitive somit überflüssig.

Listing 4.12 Benutzung der *Merge*-Varianten.

```
1: [...]
2:
3: // Spawn three Child-Tasks
4: TaskHandle task1 = Spawn(ModifyList, list, 4);
5: TaskHandle task2 = Spawn(ModifyList, list, 5);
6: TaskHandle task3 = Spawn(ModifyList, list, 6);
7:
8: [...]
9:
10: // Variant 1: MergeAll
11: MergeAll();
12:
13: // Variant 2: MergeAllByHandle
14: MergeAllByHandle(task1, task2);
15:
16: // Variant 3: MergeAny
17: TaskHandle mergedTask = MergeAny();
18:
19: // Variant 4: MergeAnyByHandle
20: TaskHandle mergedTask = MergeAnyByHandle(task1, task2);
21:
22: if (mergedTask == task1){
23:   print("Task1␣merged!");
24: } else {
25:   print("Task2␣merged!");
26: }
```

werden, um *Nachbedingungen (Post-Conditions)* [66] zu überprüfen. Bedingungsfunktionen müssen für einzelne Tasks definiert werden. Eine allgemeingültige Bedingungsfunktion, die beispielsweise einer `Merge`-Primitive übergeben wird, ist nicht möglich. Das liegt daran, dass beim Aufruf der `Merge`-Primitive unklar sein kann (z.B. bei `MergeAny`), welcher Task gemerged wird und somit auch unklar ist, welche Datenstrukturen die Bedingungsfunktion erwarten muss. Die Kenntnis über die zu erwartenden Datenstrukturen ist allerdings erforderlich, da die Bedingungsfunktion auf ebendiesen eine Entscheidung darüber fällen soll, ob der `Merge` durchgeführt werden soll. Diese Überprüfung wird vorgenommen, bevor die Ergebnisse zusammengeführt werden. Sollte die Nachbedingung nicht erfüllt werden, dann wird das Ergebnis des Child-Tasks verworfen. Das Verhalten kann mit einem Rollback, der von der Laufzeitumgebung realisiert wird, verglichen werden.

Das Konzept für die Bedingungsfunktion ähnelt dem Konzept von *Transactional Memory* [48]. Bei der Nutzung von Transactional Memory können Bereiche des Quelltextes in Transaktionen gruppiert werden. Diese Transaktionen werden entweder vollständig

ausgeführt oder gar nicht. Ein Unterschied zu der Nutzung einer Bedingungsfunktion in Spawn & Merge ist das Verhalten bei einem konfligierenden Schreibzugriff auf dieselben Daten aus zwei nebenläufigen Prozessen. Wenn zwei Threads, die Transactional Memory benutzen, schreibend auf dieselben Daten zugreifen, dann wird mindestens eine der beiden Transaktionen nicht ausgeführt (Rollback). Bei Spawn & Merge hingegen werden konfligierende Schreibzugriffe auf die Kopien der Datenstrukturen in einem Task im Nachhinein deterministisch aufgelöst (Details dazu folgen in Kapitel 5). Ein Rollback der Ergebnisse eines Tasks geschieht nur dann, wenn die Bedingungen für die Zusammenführung nicht erfüllt sind. Das Auftreten von Rollbacks ist somit nicht abhängig von der Parallelität der Anwendung, sondern von den (deterministischen) Ergebnissen der Task-Berechnungen. Das bedeutet wiederum, dass die Einführung der Bedingungsfunktion keinen Einfluss auf die deterministische Ausführung der Anwendung hat.

Schnittstelle der Bedingungsfunktion

Die abstrakte Schnittstelle einer Bedingungsfunktion ist wie in Listing 4.13 dargestellt aufgebaut. Als Parameter bekommt sie sowohl die zusammenführbaren Datenstrukturen des Parent-Tasks (*`parentStructures`*), als auch die veränderten Kopien dieser Datenstrukturen, die wiederum die Ergebnisse des Child-Tasks darstellen (*`childStructures`*). Die Datenstrukturen werden jeweils durch eine Liste der nicht veränderbaren Referenzen auf die zusammenführbaren Datenstrukturen (`const Mergeable*`) repräsentiert. Die Bedingungsfunktion kann nun für alle Datenstrukturpaare sicherstellen, dass alle Nachbedingungen erfüllt sind. Das Ergebnis dieser Überprüfung wird über die Rückgabe (*`success`*) der Funktion zurückgegeben (z.B. `true` wenn alle Bedingungen erfüllt sind und `false` wenn nicht).

Listing 4.13 Definition der Schnittstelle der *Bedingungsfunktion*.

```
success ConditionalFct(parentStructures, childStructures)
```

Nutzungsbeispiel

Listing 4.14 zeigt ein Beispiel dafür, wie eine Bedingungsfunktion an ein Task-Handle angehängt und benutzt werden kann. Zuerst wird die Bedingungsfunktion in Zeile 1 definiert. In diesem Beispiel ist die Nachbedingung, dass die erste zusammenführbare Datenstruktur des Child-Tasks (hier ein `MergeableValue`, der gecastet werden muss) einen größeren Wert enthält als der aktuell im Parent-Task gespeicherte Wert. In Zeile 13 wird dem Task `task` diese Bedingungsfunktion durch den Aufruft von `TaskHandle.setCondFct(function)` zugewiesen. Die Ergebnisse des Child-Tasks `task` werden dementsprechend in Zeile 15 nur

dann mit den Datenstrukturen des Parent-Tasks zusammengeführt, wenn diese Bedingung zutrifft.

Listing 4.14 Benutzung der Bedingungsfunktion.

```
1:  bool Condition(list<const Mergeable*> parentStr,
2:                 list<const Mergeable*> childStr){
3:    int cVal = (const MergeableValue*)childStr[0]->Val();
4:    int pVal = (const MergeableValue*)parentStr[0]->Val();
5:    return (cVal > pVal);
6:  }
7:
8:  [...] // Definition (Function f and MergeableValue* mVal)
9:
10: TaskHandle task = Spawn(f, mVal);
11:
12: // Set conditional function
13: task.setCondFct(Condition);
14:
15: MergeAll();
```

Negative Evaluation

Ein durch die negative Evaluation der Bedingungsfunktion abgebrochener Merge-Vorgang zählt für die `Merge`-Primitive als abgeschlossener Merge. Das heißt, dass beispielsweise `MergeAny` auch bei einem Rollback des gemergeten Child-Tasks aufhört zu blockieren und die Programmausführung fortgesetzt wird. So wird verhindert, dass es zu Deadlocks kommt weil die `Merge`-Primitive weiterhin blockiert, obwohl kein weiterer Child-Task mehr vorhanden ist, der noch gemerged werden muss.

4.2.9 Sync

Eine Einschränkung des Spawn & Merge Programmiermodells, wie es bis zu diesem Punkt beschrieben wurde, ist, dass Tasks nur nach ihrer Fertigstellung zusammengeführt (gemerged) werden können. Dementsprechend ist besonders die Erstellung langlebiger Tasks problematisch, die die Datenstrukturen ihres Parent-Tasks beeinflussen sollen (z.B. um Zwischenergebnisse zu übergeben). Ebendies ist allerdings oft ein gewünschtes Verhalten.

Als Beispiel seien Child-Tasks gegeben, die TCP-Verbindungen [94] handhaben. Diese sollten veränderte Daten jedes Mal mit ihrem Parent-Task zusammenführen, wenn sie eine Anfrage verarbeitet haben, und nicht erst dann, wenn die TCP-Verbindung geschlossen wurde. Eine umständliche Lösung für dieses Problem wäre es, wenn ein Child-Task sich nach jeder Aktion (z.B. dem Behandeln einer TCP-Anfrage) selbst beenden und seine Daten

mit dem Parent-Task zusammenführen würde. Der Parent-Task müsste dann überprüfen, ob der Child-Task noch weitere Aufgaben zu vollziehen hat (z.B. weil die TCP-Verbindung noch offen ist) und einen neuen Child-Task spawnen, der diese Aufgaben übernimmt. Diese Lösung hat, abgesehen davon, dass sie umständlich zu implementieren ist, einige Nachteile. Zum einen ergibt sich ein nicht notwendiger Performance-Verlust dadurch, dass ein Task beendet wird, nur um danach wieder neu gestartet zu werden. Dabei müssen auch geteilte Datenstrukturen neu kopiert werden. Zum anderen ist es für diese Umsetzung erforderlich, dass auch alle von diesem Child-Task gespawnten weiteren Child-Tasks fertiggestellt sein müssen, damit der Task mit dem Parent-Task zusammengeführt werden kann.

Um die Entwicklung solcher Szenarien zu erleichtern, wird in diesem Kapitel die *Sync*-Primitive eingeführt. Die `Sync`-Primitive ermöglicht den Austausch von Zwischenergebnissen zwischen einem Child-Task und seinem Parent-Task. Daraus ergibt sich der Vorteil, dass der Entwickler explizit Änderungen an geteilten Datenstrukturen zwischen zwei Tasks austauschen kann, ohne dass einer der Tasks beendet und neu gestartet werden muss. Das sorgt auch für einen kürzeren und übersichtlicheren Quelltext einer Anwendung. Ein weiterer Vorteil, der sich daraus ergibt, ist, dass geteilte Datenstrukturen nicht erneut kopiert werden müssen. Als Datenaustausch zwischen den Tasks ist es ausreichend, wenn diese die Veränderungen an Daten dem jeweils anderen Task melden (Details siehe Kapitel 5). Der dritte Vorteil ist, dass ein `Sync`-Aufruf unabhängig davon durchgeführt werden kann, ob es noch laufende eigene Child-Tasks gibt, oder nicht.

Nutzungsbeispiel

In Listing 4.15 wird ein Beispiel dargelegt, in dem die `Sync`-Primitive zur Synchronisation zwischen Tasks genutzt wird. Zuerst werden in den Zeilen 13 und 14 die Tasks `task1` und `task2` gespawnt, die beide eine Kopie der Liste `list` erhalten. Beide Tasks fügen nun nebenläufig in Zeile 2 einen Wert (d.h. `task1` den Wert 1 und `task2` den Wert 2) in die Liste ein. Anschließend wird die `Sync`-Primitive aufgerufen, die dafür sorgt, dass der aufrufende Task blockiert und darauf wartet, dass der eigene Task vom Parent-Task gemerged wird.

Es werden zwei Argumente an `Sync` übergeben. Das erste Argument gibt den *Sync-Modus* für die Durchführung des `Sync` an. Der hier gewählte Sync-Modus `RETURN_DIRECTLY` bedeutet, dass der Parent-Task die synchronisierte Datenstruktur sofort nach der Zusammenführung und Aktualisierung an den Child-Task zurückgeben soll. Auf das Beispiel bezogen hat dies folgende Auswirkung: Der Parent-Task wartet in Zeile 16 auf seine Child-Tasks in der Spawn-Reihenfolge. Nachdem der erste Child-Task (hier `task1`) gemerged wurde, prüft der Parent-Task den Sync-Modus. Im Falle von `RETURN_DIRECTLY` gibt er die synchronisierte Datenstruktur sofort zurück an den Child-Task, der seine Ausführung fortsetzen kann. Erst im Anschluss daran setzt der Parent-Task das Mergen mit Task `task2` fort. Somit würde sich für `task1` die Print-Ausgabe „`[1]`“ erge-

Listing 4.15 Benutzung der *Sync*-Primitive.

```
1: void fct(MergeableList* mList, int i){
2:   mList->append(i);
3:
4:   // Synchronize with Parent-Task
5:   Sync(RETURN_DIRECTLY, mList);
6:
7:   // Print list
8:   print(mList->ToString());
9: }
10:
11: MergeableList* list = new MergeableList();
12:
13: TaskHandle task1 = Spawn(fct, list, 1);
14: TaskHandle task2 = Spawn(fct, list, 2);
15:
16: MergeAll();
```

ben (Zeile 8), da die Aktualisierungen von `task2` noch nicht vom Parent-Task gemerged wurden. Der zweite mögliche Sync-Modus, der für `Sync` zur Verfügung steht, ist `RETURN_AFTER_MERGE_CYCLE_COMPLETED`. Dieser Modus sagt aus, dass die Datenstrukturen erst dann aktualisiert und an den Child-Task zurückgegeben werden sollen, wenn der Parent-Task den aktuellen `Merge`-Aufruf vollständig abgearbeitet hat[20]. Das hat den Effekt, dass die Child-Tasks die Aktualisierungen aller im aktuellen `Merge`-Aufruf gemergeten Tasks erhalten und nicht nur die Aktualisierungen der Tasks, die vor ihnen gemerged wurden. In diesem Beispiel würde sich damit die Print-Ausgabe für `task1` zu „`[1,2]`“ ändern, da er auch die Aktualisierungen von `task2` erhalten hätte.

Ab dem zweiten Argument folgen die Datenstrukturen, die mit ihrem Gegenstück beim Parent-Task synchronisiert werden sollen, in diesem Falle `mList`. Die übergebenen Datenstrukturen werden an den Parent-Task übertragen und von diesem mit den eigenen Datenstrukturen zusammengeführt. Das Ergebnis wird anschließend wieder an den Child-Task zurückgegeben, der seine Ausführung fortsetzt.

Sync-Primitive

Die abstrakte Schnittstelle der `Sync`-Primitive ist in Listing 4.16 beschrieben. `Sync` erwartet als ersten Parameter (*syncMode*) die Spezifikation des Sync-Modus, der verwendet werden soll. Hier kann der Programmierer, wie bereits beschrieben, zwischen `RETURN_DIRECTLY`

[20] Ein `MergeAll(ByHandle)`-Aufruf gilt als vollständig abgearbeitet, wenn alle (übergebenen) Tasks einmal gemerged wurden. Ein `MergeAny(ByHandle)`-Aufruf gilt als vollständig abgearbeitet, sobald ein Task gemerged wurde und die Primitive nicht mehr blockiert.

und `RETURN_AFTER_MERGE_CYCLE_COMPLETED` wählen. Anschließend können beim Parameter `structures` beliebig viele zusammenführbare Datenstrukturen, die initial vom Parent-Task stammen, angegeben werden, damit diese mit dem Parent-Task synchronisiert werden. Die Sync-Primitive darf dabei nicht im Main-Task aufgerufen werden, da es dort keinen Parent-Task gibt, mit dem sich der Task synchronisieren könnte. Ein Aufruf innerhalb des Main-Tasks führt daher zu einem Fehler.

Listing 4.16 Definition der *Sync* Schnittstelle.

```
merged Sync(syncMode[,structures...])
```

Der Entwickler ist nicht verpflichtet Datenstrukturen zu übergeben und kann die Sync-Primitive auch zur logischen Synchronisation zwischen Parent-Task und Child-Tasks nutzen. Das ermöglicht beispielsweise das Nachverfolgen des Fortschritts eines Child-Tasks, wenn der Parent-Task weiß, dass der Child-Task bei einem Sync-Aufruf bestimmte Fortschritte erreicht hat. Die Rückgabe (*merged*) gibt an, wie die Bedingungsfunktion[21] beim Parent-Task ausgewertet wurde und somit, ob der Parent-Task die Veränderungen mit seinen eigenen zusammengeführt hat oder nicht. Das ermöglicht es dem Entwickler nach einem abgewiesenen Sync-Aufruf zu entscheiden, ob die Ausführung des Tasks oder der Anwendung fortgesetzt oder beendet werden soll. Diese Entscheidung kann davon abhängig sein, ob die fehlgeschlagene Zusammenführung eine Aussage über zukünftige Zusammenführungen fällt. Handelt es sich bei der Zusammenführung durch den Sync-Aufruf um eine notwendige Bedingung für den Programmablauf, dann kann auch kein zukünftiger Stand gemerged werden. Das heißt, es gibt eine Verletzung der *Lebendigkeits-Eigenschaft (liveness)* [78], die angibt, ob noch einmal ein bestimmter Zustand (hier ein späterer erfolgreicher Merge-Aufruf) erreicht werden kann. Andernfalls ist es möglich, dass sich durch die weitere Ausführung des Tasks zu einem späteren Zeitpunkt noch ein Zustand ergeben kann, der wieder gemerged werden kann. Dies kann allerdings nur mit anwendungsspezifischem Wissen entschieden werden.

Datenstrukturen, die an Sync übergeben werden, müssen Sync unterstützen. Das bedeutet, dass es die darin implementierten Methoden für die deterministische Zusammenführung erlauben müssen, nicht nur die Veränderungen des Child-Tasks zu übergeben, sondern auch die Veränderungen des Parent-Tasks für den Child-Task aufzubereiten und zu übergeben. Diese Anforderung wird in Kapitel 5.2.2 im Detail beschrieben.

[21]Eine für Sync genutzte Bedingungsfunktion muss vom Entwickler so entworfen werden, dass sie mit den an die Sync-Primitive übergebenen Datenstrukturen umgehen kann. Bei Nichtbeachtung kann sich zur Laufzeit das Problem ergeben, dass in der Bedingungsfunktion alle Datenstrukturkopien erwartet werden, der Entwickler beim Aufruf der Sync-Primitive allerdings nur einzelne Datenstrukturen zur Übertragung von Zwischenergebnissen übergibt.

Auswirkungen der Einführung von Sync

Durch die Einführung der `Sync`-Primitive ergibt sich keine Einschränkung bezüglich der Determinismus Eigenschaften des Programmiermodells. Dadurch, dass der `Sync`-Aufruf eines Child-Tasks aktiv auf der Seite des Parent-Tasks durch einen `Merge`-Aufruf entgegen genommen werden muss, geschieht diese Zusammenführung von Zwischenergebnissen nur an deterministisch durch den Quelltext vorgegebenen Zeitpunkten. Auch die Auswertung der Bedingungsfunktion und das damit verbundene Verhalten der Anwendung ergibt sich deterministisch aus dem Zustand der zusammenführbaren Datenstrukturen. Es wird somit kein Nichtdeterminismus in das Programmiermodell eingeführt.

Durch die Erweiterung um die `Sync`-Primitive und die damit einhergehende Steigerung der Ausdrucksfähigkeit des Spawn & Merge Programmiermodells, ergeben sich neue Anforderungen an das Framework, die im Folgenden beschrieben werden. Als erstes muss das Framework bei der Zusammenführung zweier Tasks nicht nur, wie bisher beschrieben, die Datenstrukturen des Parent-Tasks anpassen sondern gleichzeitig auch die Veränderungen des Parent-Tasks in die Datenstrukturen des Child-Tasks einpflegen, sofern es sich um einen `Sync`-Aufruf handelt. Dies ist notwendig, um den Child-Task über Veränderungen der Daten durch den Parent-Task oder Sibling-Tasks[22] zu informieren. Nach dem `Sync` sollen somit die Datenstrukturen des Parent-Tasks und des Child-Tasks auf demselben Stand sein.

Als zweites müssen die `MergeAll`- und die `MergeAllByHandle`-Primitive angepasst werden. Die beiden `MergeAll`-Primitive geben an, dass alle (übergebenen) Child-Tasks mit dem Parent-Task zusammengeführt werden sollen. Dies kann allerdings durch die Einführung der `Sync`-Primitive zwei unterschiedliche semantische Bedeutungen haben. Einerseits kann gemeint sein, dass jeder Child-Task genau einmal mit dem Parent-Task zusammengeführt werden soll. Durch die `Sync`-Primitive kann es dabei vorkommen, dass nach der Durchführung des Merges noch Child-Tasks aktiv sind. Andererseits kann bei den `MergeAll`-Primitiven gemeint sein, dass alle (übergebenen) Child-Tasks solange gemerged werden sollen, bis diese Child-Tasks fertiggestellt sind. Dies ist zum Beispiel dann das gewünschte Veralten, wenn `MergeAll` nach der Durchführung der Task-Funktion genutzt wird, um final auf die Fertigstellung aller Child-Tasks zu warten (siehe Kapitel 4.2.3).

Für diese Unterscheidung des Verhaltens der beiden `MergeAll`-Primitive wird ein *Merge-Modus* eingeführt, der als erster Parameter übergeben wird (siehe Anhang B.1 und B.2 für die vollständige Schnittstellendefinition). Der Merge-Modus kann dabei entweder den Wert `ALL_TASKS_ONCE` oder den Wert `TILL_ALL_FINISHED` annehmen. Bei dem Wert `ALL_TASKS_ONCE` wird jeder zu mergende Task in der durch die Primitive festgelegte Reihenfolge genau einmal gemerged. Anschließend wird die Ausführung der Anwendung

[22] Weitere Child-Tasks eines gemeinsamen Parent-Tasks werden auch als *Sibling-Tasks (Geschwister-Tasks)* bezeichnet.

hinter der `MergeAll`-Primitive fortgesetzt. Bei `TILL_ALL_FINISHED` werden die zu mergenden Tasks solange wiederholt gemerged, bis alle fertiggestellt sind. Dazu wird solange wiederholt die entsprechende `MergeAll`-Primitive mit `ALL_TASKS_ONCE` und allen noch aktiven zu mergenden Child-Tasks aufgerufen, bis alle fertiggestellt wurden. So wird erreicht, dass jeder Child-Task mit einem `Sync`-Aufruf auch die Veränderungen durch seine Sibling-Tasks bekommt.

Aktualisierung der Datenstruktur-Historie Φ

Wird ein `Sync` durchgeführt, so müssen auch die gespeicherten Datenstruktur-Historien Φ_{D_X} für die synchronisierten Datenstrukturen aktualisiert werden (siehe Kapitel 4.2.5). Zuerst muss die gespeicherte Versionsnummer geändert werden, die angibt, zu welchem Zeitpunkt die Datenstruktur für den Child-Task kopiert wurde. Durch den `Sync` und die damit einhergehende Zusammenführung der Datenstrukturen, ist der Ausgangspunkt für die nächste Zusammenführung der aktualisierte Stand der Datenstruktur. Ebenso ändert sich durch die Zusammenführung der Datenstrukturen die Versionsnummer der Datenstruktur des Parent-Tasks, die somit angepasst werden muss.

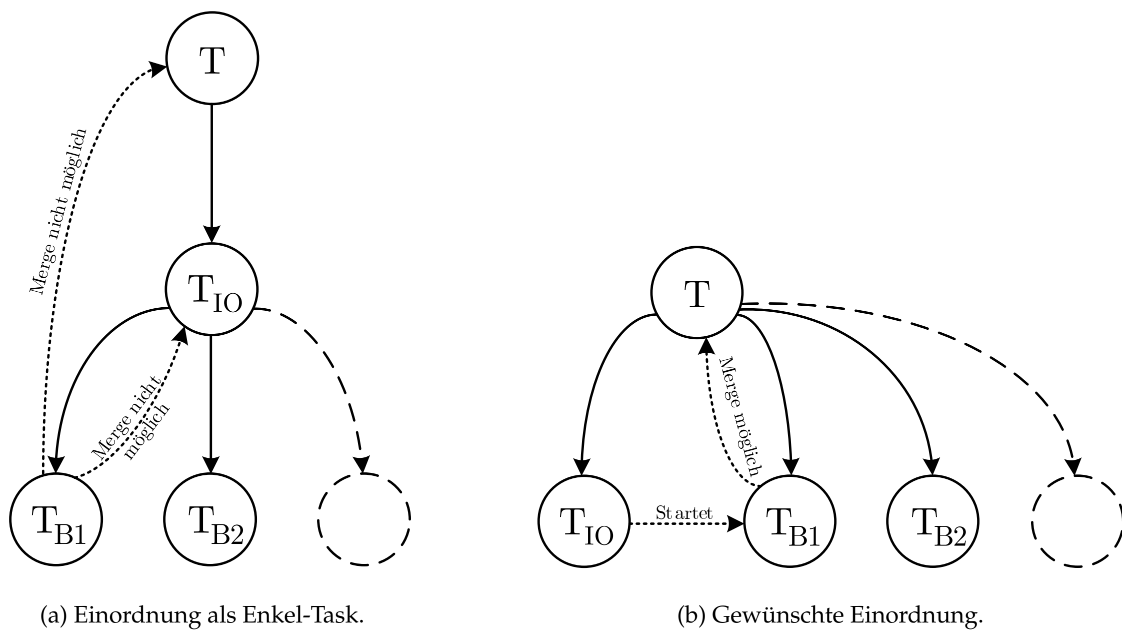
4.2.10 SpawnSibling

`Spawn` bietet dem Entwickler die Möglichkeit, einen Child-Task unter dem aktuellen Task zu erstellen. Dieser Child-Task kann (nach seiner Durchführung) seine Änderungen an den übergebenen Datenstrukturen mit den Datenstrukturen des Parent-Tasks zusammenführen. Mit den bisher beschriebenen Synchronisationsprimitiven gibt es keine direkte Möglichkeit für einen Task, einen *Sibling-Task* (d.h. einen Task mit dem er einen gemeinsamen Parent-Task besitzt) zu erstellen. Diese Funktionalität wäre notwendig, wenn ein Task einen weiteren Task erstellen möchte, der beispielsweise eine andere Funktion ausführt, aber auch mit dem Parent-Task zusammengeführt werden kann.

Diese Funktionalität ist unter anderem in Anwendungen notwendig, in denen ein Task zwei blockierende Funktionen aufrufen möchte. Dies kann unter anderem dann der Fall sein, wenn der Task (T) gleichzeitig auf eine Nutzereingabe und auf fertiggestellte Child-Tasks warten möchte, die bereits getätigte Nutzereingaben verarbeiten. Hier wäre der Aufruf zweier blockierender Funktionen notwendig: `getline(std::cin, str)` und der Aufruf einer `Merge`-Primitive. Für jede getätigte Nutzereingabe soll ein neuer Child-Task gestartet werden, der diese Eingabe verarbeitet und das Ergebnis an den Parent-Task T zurückgibt.

Die einzige Möglichkeit einen zweiten blockierenden Aufruf neben dem Aufruf einer `Merge`-Primitive zu ermöglichen, ist es, einen Child-Task zu spawnen (T_{IO}), der sich um die Ausführung der zweiten blockierenden Funktion (hier `getline(std::cin, str)`) küm-

mert. Das bedeutet allerdings wiederum, dass der Task T_{IO} selbst keine `Merge`-Primitive aufrufen kann, um die Ergebnisse gespawnter Child-Tasks zu verarbeiten. Der Parent-Task T selbst kann nun zwar durch den Aufruf einer `Merge`-Primitive auf die Fertigstellung seiner Child-Tasks warten, nicht aber auf die Tasks T_B, die von T_{IO} zur Verarbeitung der Nutzereingaben mit `Spawn` gestartet wurden (siehe Abbildung 4.7a). Das Ziel ist es jedoch, dass die von Task T_{IO} neu erstellten Tasks (T_B) mit dem Task T zusammengeführt werden können (siehe Abbildung 4.7b). Mit den bisher beschriebenen Synchronisationsprimitiven kann dies nur umständlich abgebildet werden (siehe Kapitel 4.2.10).



(a) Einordnung als Enkel-Task.

(b) Gewünschte Einordnung.

Abbildung 4.7: Child-Task-Einordnung in die Task-Hierarchie.

Da es sich hierbei um ein sich wiederholendes Muster handelt, das bei der Nutzung von blockierenden Aufrufen (z.B. Eingaben in das System) angewendet werden muss, wurde die Synchronisationsprimitive `SpawnSibling` eingeführt. `SpawnSibling` erlaubt es dem Entwickler innerhalb eines laufenden Tasks einen Sibling-Task zu spawnen, der mit dem gemeinsamen Parent-Task zusammengeführt werden kann (siehe Abbildung 4.8).

Nutzungsbeispiel

Listing 4.17 zeigt, wie das oben beschriebene Beispiel mit der `SpawnSibling`-Primitive umgesetzt werden kann. In Zeile 16 wird der Task T_{IO} mit der Funktion `readInput` gestartet und bekommt die initiale Version der geteilten Datenstruktur `sharedData` übergeben. Dabei wird für Task T_{IO} eine Kopie der Datenstruktur erstellt, die mit der Datenstruktur beim Parent-Task zusammengeführt werden kann. Der Task T_{IO} wartet in Zeile 6 auf eine

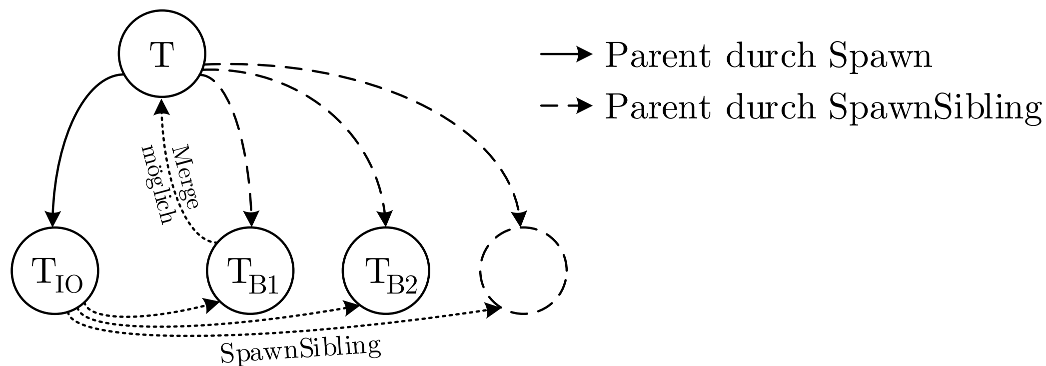


Abbildung 4.8: *SpawnSibling* Task-Einordnung.

Nutzereingabe. Wurde eine Eingabe registriert, so startet er einen Sibling-Task durch den Aufruf von `SpawnSibling` (Zeile 10). Hierbei übergibt er sowohl die Funktion `compute`, die der neue Task berechnen soll, als auch die registrierte Eingabe `readStr` sowie eine Kopie der geteilten Datenstruktur `data`, die Task T_{IO} selbst vom Parent-Task erhalten hat. Das Spawn & Merge Framework informiert nun den Parent-Task über seinen neuen Child-Task und startet die Ausführung des neuen Tasks.

SpawnSibling-Primitive

Die Definition der abstrakten Schnittstelle von `SpawnSibling` ist in Listing 4.18 beschrieben. Die Funktion, die der Sibling-Task ausführen soll, wird über den Parameter *`function`* angegeben. Für die übergebenen Funktionen gelten dabei dieselben Annahmen, wie für den *`function`*-Parameter der `Spawn`-Primitive (siehe Kapitel 4.2.6). Das bedeutet vor allem, dass die Funktion für sich genommen deterministisch sein muss, um die Determinismusgarantie zu erhalten. Die übergebenen Datenstrukturen für den gespawnten Sibling-Task werden im Parameter *`functionParameters`* angegeben.

Wie bei `Spawn` sind jegliche Werteparameter (Call-by-Value) erlaubt, die für die Verwendung innerhalb des Sibling-Tasks kopiert werden können. Für zusammenführbare Datenstrukturen (Mergeables) ist es notwendig, dass zwei Bedingungen eingehalten werden. Zum einen müssen die übergebenen Datenstrukturen ursprünglich vom Parent-Task stammen (d.h. die Datenstrukturen müssen in den `startingStructures` des Tasks (siehe Kapitel 4.2.3) enthalten sein). Dies ist erforderlich, da der Sibling-Task seine Veränderungen an den Datenstrukturen abschließend wieder mit seinem Parent-Task zusammenführen muss. Würde hier eine Datenstruktur übergeben werden, die der `SpawnSibling` aufrufende Task selbst erstellt hat, so würde die Zusammenführung des Sibling-Tasks mit dem Parent-Task fehlschlagen oder zu undefiniertem Verhalten der Anwendung führen. Zum anderen muss die zusammenführbare Datenstruktur für die Unterstützung der `SpawnSibling`-Primitive

Listing 4.17 Starten eines Sibling-Tasks mit der *SpawnSibling*-Primitive.

```
1:  void compute(int value, Mergeable* data);
2:
3:  void readInput(Mergeable* data){
4:    [...] // Initialize variables
5:    while (true){
6:      getline(std::cin, readStr);
7:      if (readStr.ToInt() == -1){
8:        break; // Break condition
9:      }
10:     SpawnSibling(compute, readStr.ToInt(), data);
11:   }
12: }
13:
14: Mergeable* sharedData = new Mergeable();
15:
16: TaskHandle T_IO = Spawn(readInput, sharedData);
17:
18: while (true){
19:   if (MergeAny() == T_IO){
20:     break; // Break condition
21:   }
22: }
```

Listing 4.18 Definition der *SpawnSibling* Schnittstelle.

```
void SpawnSibling(function[,functionParameters...])
```

angepasst worden sein (die genaue Art der Anpassung wird im Verlaufe dieses Kapitels beschrieben). Die Einhaltung dieser Bedingungen muss vom Framework geprüft werden. Falls eine dieser Bedingungen verletzt wird, so muss dies zu einem Fehler führen. Wenn beide Bedingungen eingehalten werden, dann wird der aktuelle Stand der zusammenführbaren Datenstruktur, wie er gerade im Task vorliegt, an den Sibling-Task übergeben. Ein weiterer Unterschied im Vergleich mit der `Spawn`-Primitive ist, dass es bei einem Aufruf von `SpawnSibling` keine Rückgabe eines Task-Handles gibt. Das liegt daran, dass der Task, der `SpawnSibling` aufgerufen hat, nicht mit dem neuen Sibling-Task zusammengeführt werden kann und die Hoheit über den neuen Task beim Parent-Task liegt.

Ebenso wie die `Sync`-Primitive darf `SpawnSibling` nicht im Main-Task aufgerufen werden. Das liegt daran, dass der Main-Task keinen Parent-Task hat und es somit keinen Sibling-Task mit „demselben Parent-Task“ geben kann. Der Aufruf von `SpawnSibling` im Main-Task muss somit zu einem Fehler führen.

Alternative Lösungsmöglichkeit durch Sync

Eine alternative Lösungsmöglichkeit dieser Problemstellung, ohne eine Verwendung der `SpawnSibling`-Primitive, ist in Listing 4.19 zu sehen. Hier synchronisiert sich der Task T_{IO}, der in der Funktion `getline` auf eine Nutzereingabe wartet, nach jeder Eingabe mit dem Parent-Task über die `Sync`-Primitive. Damit der Task T_{IO} den Parent-Task über neue Eingaben informieren kann, wird hier eine Liste der gelesenen Werte `readValues` als Variable für den Informationsaustausch genutzt. An diese Liste hängt der Task T_{IO} den gelesenen (und in einen Integer konvertierten) eingegebenen Wert an (Zeile 7) und synchronisiert sich mit dem Parent-Task (Zeile 11). Der Parent-Task wiederum, der in Zeile 21 darauf wartet, dass ein Child-Task mergen möchte, empfängt die von Task T_{IO} übergebenen Änderungen an der Liste und spawnt einen neuen Child-Task. Dieser Child-Task wird in Zeile 28 mit der Funktion `compute` (Zeile 1), dem eingelesenen Wert aus der `readValues`-Liste und den geteilten Daten (`sharedData`), die er später wieder mit dem Parent-Task zusammenführen soll, gespawnt. Anschließend entfernt der Parent-Task den eingelesenen Wert aus `readValues`, damit nach dem nächsten `Sync` von Task T_{IO} wieder nur ein Wert in der Liste enthalten ist.

Im Gegensatz zur Verwendung der `SpawnSibling`-Primitive ergibt sich aus dem hier beschriebenen Lösungsansatz eine zusätzliche Entwicklungskomplexität, die sich an dem Beispielcode in Listing 4.19 nachvollziehen lässt. Zum einen muss bei der Verwendung von `Sync` für das Starten von Sibling-Tasks eine zusammenführbare Datenstruktur (`readValues`) zur Übertragung von Werten zwischen dem Task T_{IO} und den Parent-Task eingeführt werden. Diese Umleitung der Daten, die für den Parent-Task selbst nicht relevant sind, ist notwendig, damit der Parent-Task die entsprechenden, gelesenen Daten an die neu zu startenden Tasks T_B übergeben kann. Hierzu muss der Entwickler ein eigenes „Protokoll" implementieren, das die korrekte Übertragung und das korrekte Verhalten der Anwendung in Anbetracht der übertragenen Daten sicherstellt. Im Beispiel ist mit diesem Protokoll unter anderem die Übertragung von immer genau einem Wert, sowie das Entfernen des Wertes, sobald der entsprechende Child-Task gespawnt wurde (Zeile 29), gemeint. Zusätzlich muss das Protokoll hier nach einem `MergeAny` entscheiden können, ob der Task T_{IO} einen eingelesenen Wert übergeben hat und somit einen weiteren Sibling-Task starten möchte, oder ob der Task seine Arbeit beendet hat und abschließend gemerged werden möchte. Im Beispiel gibt der Task T_{IO} dies durch das Übertragen des Wertes -1 an (Zeile 7–9 und Zeile 25–26).

Der Umweg über einen `Sync`-Aufruf führt zusätzliche Wartezeiten beim Parent-Task und beim Child-Task ein, die bei der Verwendung von `SpawnSibling` nicht vorliegen. Jedes Mal, wenn ein Sibling-Task gespawnt werden soll, wird beim Child-Task ein `Sync` aufgerufen und beim Parent-Task ein `Merge` durchgeführt. Der `Sync`-Aufruf bedeutet dabei, dass der Child-Task darauf warten muss, dass der Parent-Task dazu bereit ist den

Listing 4.19 Starten eines Sibling-Tasks ohne *SpawnSibling*.

```
1:  void compute(int value, Mergeable* data);
2:
3:  void readInput(MergeableList* values){
4:    [...] // Initialize variables
5:    while (true){
6:      getline(std::cin, readStr);
7:      values->append(readStr.ToInt());
8:      if (readStr.ToInt() == -1){
9:        break; // Break condition
10:     }
11:     Sync(RETURN_DIRECTLY, values);
12:   }
13: }
14:
15: Mergeable* sharedData = new Mergeable();
16: MergeableList* readValues = new MergeableList();
17:
18: TaskHandle T_IO = Spawn(readInput, readValues);
19:
20: while (true){
21:   MergeAny();
22:   if (readValues.size() == 0){
23:     continue; // Compute-Task merged
24:   }
25:   if (readValues[0] == -1){
26:     break; // Break condition
27:   }
28:   Spawn(compute, readValues[0], sharedData);
29:   readValues.remove(0);
30: }
```

`Merge` durchzuführen (was zu Wartezeiten führen kann). Zusätzlich müssen für die zusammenführbaren Datenstrukturen, die für die Parameterübertragung angelegt wurden, bei jedem `Sync` die Mechanismen zur deterministischen Zusammenführung angewendet werden. Auch dies führt zu einer zusätzlichen Berechnungskomplexität.

Des Weiteren lassen sich mit diesem Lösungsansatz nicht alle Szenarien abbilden, die sich mit der `SpawnSibling`-Primitive realisieren lassen. Angenommen, der Entwickler möchte, dass jeder neue Child-Task, der die Funktion `compute` ausführen soll, mit derselben initialen leeren Liste gestartet wird. Diese Annahme ist nicht unwahrscheinlich, spiegelt sie doch den Aufbau einer Anwendung wieder, bei der alle Child-Tasks zu Beginn der Ausführung mit demselben Stand der Datenstrukturen (z.B. einer leeren Liste) ge-

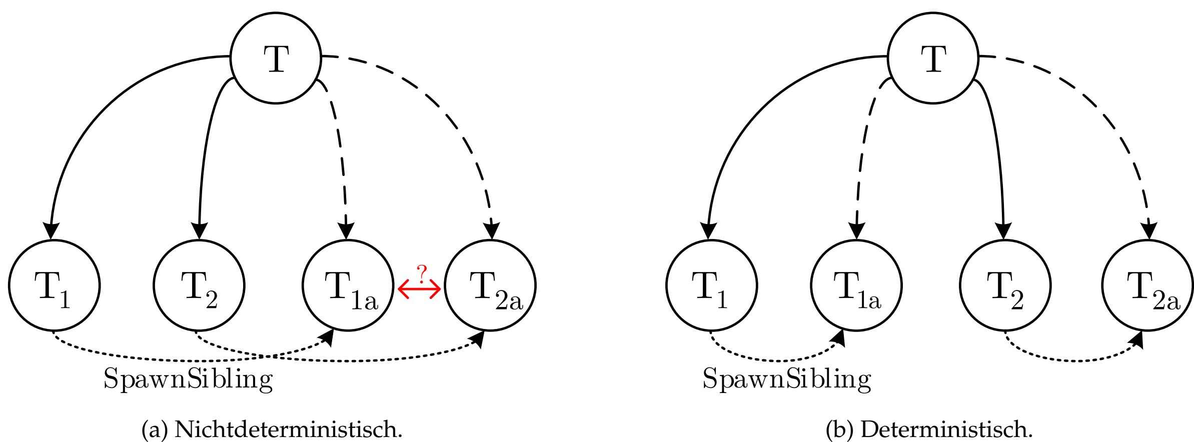
startet werden. Dadurch, dass ein neuer Child-Task immer über den Parent-Task gestartet werden muss, kann der Entwickler nicht verhindern, dass bereits vorliegende Ergebnisse anderer Tasks an den neuen Child-Task übergeben werden. Im Beispiel bedeutet dies, dass beim Aufruf von `Spawn` in Zeile 28 bereits Ergebnisse anderer Child-Tasks in `sharedData` enthalten sein können. Hierbei handelt es sich nicht zwingend um ein Problem für die Anwendungslogik, dennoch wird hier dem Entwickler die Möglichkeit genommen, selbst zu entscheiden welcher Stand der `sharedData` Datenstruktur an den Child-Task übergeben werden soll.

Deterministische Einordnung neuer Sibling-Tasks

Wenn ein neuer Child-Task in die Liste der gestarteten Child-Tasks eingefügt wird, dann muss dies in deterministischer Reihenfolge geschehen. Ohne diese deterministische Einordnung könnte bei `MergeAll` keine Determinismusgarantie gegeben werden, da diese auf der deterministischen Reihenfolge innerhalb der Liste gestarteter Child-Tasks beruht. Bei `Spawn` ist diese deterministische Reihenfolge dadurch gewährleistet, dass `Spawn` immer in der selben Reihenfolge im Quellcode der Anwendung aufgerufen wird. Bei `SpawnSibling` ist diese Garantie nicht gegeben. Eine Veranschaulichung des Problems bietet das Beispiel in Listing 4.20. Hier werden zwei Child-Tasks gespawnt (Zeile 7 und Zeile 8), die anschließend jeweils `SpawnSibling` aufrufen (Zeile 4). Da nicht beeinflusst werden kann, welcher der beiden Child-Tasks bei nebenläufiger Ausführung zuerst die `SpawnSibling`-Primitive aufrufen wird, kann nicht beeinflusst werden welcher der neu gestarteten Child-Tasks zuerst in die Liste aufgenommen wird. Folgt man dem intuitiven Ansatz, dass die Child-Tasks hinten an die Liste angehängt werden, so wäre das Ergebnis nichtdeterministisch (siehe Abbildung 4.9a). Um eine deterministische Reihenfolge zu erreichen, wird daher ein neuer Child-Task T_{1a} direkt hinter dem Child-Task T_1 angehängt, der `SpawnSibling` aufgerufen hat. Zu beachten ist hierbei, dass auch ein zweiter gespawnter Sibling-Task T_{1b} direkt hinter dem aufrufenden Child-Task T_1 eingeordnet werden muss (d.h. noch vor dem vorher gestarteten Sibling-Task T_{1a}). Ohne diese Annahme wäre es möglich, dass die Reihenfolge der Child-Tasks wieder nicht deterministisch ist, falls der erste Sibling-Task T_{1a} selbst auch `SpawnSibling` aufruft und der Child-Task T_1 erneut `SpawnSibling` zur gleichen Zeit aufruft. In diesem Falle würden beide Tasks versuchen den neu gestarteten Child-Task an dieselbe Stelle (direkt hinter T_{1a}) einzuordnen und die Reihenfolge würde vom nicht vorhersehbaren Timing abhängen. Bei Einhaltung der beschriebenen Regeln ergibt sich für das Beispiel in Listing 4.20 die Task-Hierarchie, die in Abbildung 4.9b dargestellt ist.

Listing 4.20 Beispiel für Einordnung der Sibling-Tasks.

```
1:  void fct();
2:
3:  void fctChild(){
4:    SpawnSibling(fct);
5:  }
6:
7:  Spawn(fctChild);
8:  Spawn(fctChild);
9:
10: MergeAll();
```



(a) Nichtdeterministisch.

(b) Deterministisch.

Abbildung 4.9: Sibling-Task-Einordnung bei *SpawnSibling*.

Notwendigkeit der Erweiterung der Datenstruktur-Historie Φ

Durch die Einführung von `SpawnSibling` gibt es einen neuen Kontext, aus dem heraus zusammenführbare Datenstrukturen kopiert werden können. In diesem Kontext muss die Kopie der Datenstruktur so erstellt werden, dass diese nicht mit dem aufrufenden Task, sondern mit dessen Parent-Task zusammengeführt werden kann. Dabei gibt es mehrere Herausforderungen, die im Folgenden betrachtet werden.

1. Die Datenstruktur-Hierarchien müssen um eine Sibling-Relation erweitert werden.

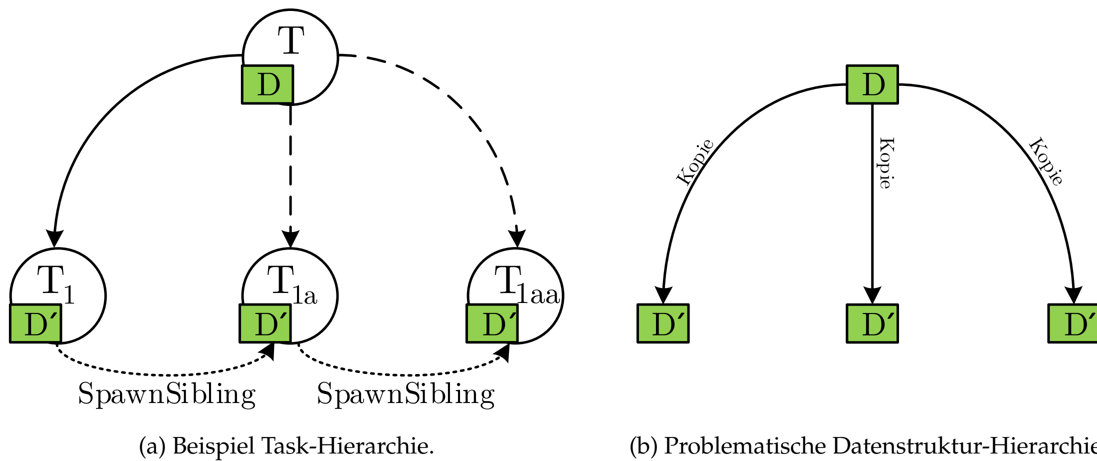
2. Die Änderungen an Datenstrukturkopien können auf unterschiedlichen Wegen beim Parent-Task ankommen.

3. Die Datenstruktur-Historien Φ müssen diese Anforderungen abbilden und aktuell gehalten werden.

Ohne die `SpawnSibling`-Primitive konnten neue Datenstrukturkopien nur von einem Task für seine Child-Tasks erstellt werden. Damit wusste der Task immer genau, welchen Stand der Datenstruktur er zuletzt gesehen hat und konnte damit feststellen, welche Modifikationen der Datenstruktur neu hinzugekommen sind. Durch `SpawnSibling` können nun allerdings Datenstrukturkopien von einem Child-Task für einen neuen Sibling-Task angelegt werden. Die neue Datenstrukturkopie kann dabei, im Gegensatz zu dem erstellten Sibling-Task, nicht ohne weiteres in der Datenstruktur-Hierarchie unter die Datenstruktur des Parent-Tasks gehängt werden. Das liegt daran, dass die Information, von welcher Datenstrukturkopie (und in welcher Version) die neue Kopie erstellt wurde, für die spätere Zusammenführung benötigt wird. Der Grund dafür ist, dass Änderungen an den Datenstrukturen durch `SpawnSibling` auf unterschiedlichen Wegen zum Parent-Task zurückgeführt werden können. Abbildung 4.10a zeigt ein Beispiel, in dem ein Parent-Task T einen Child-Task T_1 mit der Datenstruktur D spawnt. Dieser Child-Task T_1 startet anschließend einen Sibling-Task T_{1a}, der wiederum einen Sibling-Task T_{1aa} startet (beide durch Nutzung der `SpawnSibling`-Primitive). Allen drei Child-Tasks wird die Datenstruktur D übergeben. Angenommen der Child-Task T_1 hat eine Veränderung an D vorgenommen, bevor er den Sibling-Task T_{1a} gespawnt hat (wodurch D zu D' wurde). Diese Veränderung ist dementsprechend den Tasks T_{1a} und T_{1aa} bekannt, nicht jedoch dem Parent-Task T. Würden neue Datenstrukturkopien, die durch `SpawnSibling` erstellt wurden, wie bei `Spawn` direkt unter die Ursprungs-Datenstruktur eingehängt werden, dann würde sich aus Φ eine Datenstruktur-Hierarchie wie in Abbildung 4.10b ergeben. Das Problem dabei ist, dass es keine Repräsentation der Abhängigkeiten zwischen den Datenstrukturkopien gibt. Wird beispielsweise der Child-Task T_{1aa} zuerst vom Parent-Task T gemerged, so bekommt der Parent-Task damit auch die Änderungen, die T_1 an D vorgenommen hat. Der Parent-Task bekommt dabei alle Datenveränderungen, die transitiv von allen Datenstrukturkopien vorgenommen wurden, von denen die zusammengeführte Kopie abstammt. Allerdings hat der Parent-Task keine Möglichkeit dies herauszufinden und würde die Child-Tasks T_1 und T_{1a} weiterhin so mergen, als hätten diese dieselbe Veränderung unabhängig von T_{1aa} durchgeführt. Das kann dazu führen, dass die Veränderungen mehrfach durchgeführt werden.

Erweiterung der Datenstruktur-Historie Φ um die Sibling-Relation

Ohne die Informationen bezüglich des Ursprungs der Datenstrukturkopien kann der Parent-Task, wie im vorangegangenen Abschnitt beschrieben, nicht unterscheiden, welche Veränderungen er schon gesehen hat und welche Veränderungen neu sind. Daher wird die Datenstruktur-Historie Φ, die für jede Datenstruktur und jede Datenstrukturkopie gepflegt wird (siehe Kapitel 4.2.5), wie in Tabelle 4.3 kursiv hervorgehoben zu sehen, erweitert. Zur verbesserten Abbildung der Entstehung der Datenstruktur-Hierarchie werden zwei



(a) Beispiel Task-Hierarchie.

(b) Problematische Datenstruktur-Hierarchie.

Abbildung 4.10: Datenstrukturkopien bei *SpawnSibling*.

neue Felder in Φ hinzugefügt. Zum einen wird eine Referenz auf die Datenstrukturkopie gespeichert, von der beim Aufruf von `SpawnSibling` eine Kopie angelegt wurde. Zum anderen wird für jede Datenstrukturkopie gespeichert, welche weiteren Kopien sie mit `SpawnSibling` erstellt hat (*Sibling-Relation*). So ergibt sich für die Beziehungen zwischen den mit `SpawnSibling` erstellten Datenstrukturkopien ein doppelt verketteter Baum, der in Abbildung 4.11 in Form der blauen Sibling-Relationen zu erkennen ist. Dieser Baum kann sowohl in Richtung der erstellten Kopien, als auch in Richtung des Ursprungs der Kopien begangen werden. Um eine Zusammenführung der Parent-Datenstruktur mit Datenstrukturkopien zu ermöglichen, die durch `SpawnSibling` erstellt wurden, werden drei zusätzliche Werte gespeichert. Zum einen wird die Versionsnummer der Datenstruktur zu dem Zeitpunkt, als sie an `SpawnSibling` übergeben wurde, gespeichert. Zusätzlich wird die Versionsnummer gespeichert, bis zu der die übergebene Datenstrukturkopie die Datenstruktur des Parent-Tasks kennt. Außerdem ist es erforderlich, dass für eine Datenstrukturkopie nachgehalten wird, welche Veränderungen dem Parent-Task bereits bekannt sind, sodass diese nicht ein zweites Mal mit dem Parent-Task zusammengeführt werden. Diese Liste der zu ignorierenden Veränderungen ist initial leer, und ergibt sich erst, wenn Datenstrukturen mit redundanten Veränderungen gemerged werden und Φ aktualisiert wird.

Abbildung 4.11 zeigt eine Visualisierung des Beispiels aus Abbildung 4.10a, das um die Sibling-Relation (in Blau) erweitert wurde (die weiteren Informationen von Φ wurden hier zugunsten der Übersicht nicht mit aufgenommen). Durch die zusätzlichen Informationen ist der Parent-Task in der Lage nachzuvollziehen, von welchem Versionsstand welcher anderen Datenstrukturkopie die Datenstruktur abstammt, die aktuell gemerged wird. Somit kann für jede Datenstrukturkopie bei der Zusammenführung festgestellt werden, welche

Zugehörigkeit	Gespeicherte Information
Datenstruktur-Hierarchie	Parent-Task Datenstruktur Referenz (Spawn)
	Kopien der eigenen Datenstruktur (Spawn)
	Referenz auf den Ursprung der Kopie (Sibling)
	Selbst erstellte Kopien (Sibling)
Änderungsverfolgung	Versionsnummer beim Kopieren (Spawn)
	Eigene aktuelle Versionsnummer (Spawn)
	Durchgeführte Veränderungen (Spawn)
	Versionsnummer beim Kopieren (Sibling)
	Dem Ursprung bekannte Parent-Version (Sibling)
	Zu ignorierende Veränderungen (Sibling)

Tabelle 4.3: Erweiterung der in der Datenstruktur-Historie Φ gespeicherten Informationen.

Modifikationen bereits übernommen wurden und somit beim Zusammenführen mit der Parent-Datenstruktur ignoriert werden müssen. Diese werden dann in dem entsprechenden Feld von Φ gespeichert, damit sie bei der Zusammenführung berücksichtigt werden können.

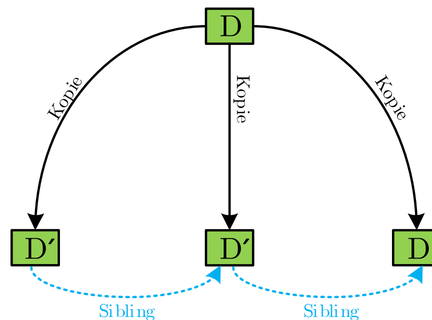


Abbildung 4.11: Datenstruktur-Hierarchie bei *SpawnSibling* mit Sibling-Relation.

Aktualisierung der Datenstruktur-Historie Φ

Mit der erhöhten Komplexität der Datenstruktur-Historie Φ steigt auch die Komplexität für die Aktualisierung von Φ bei der Durchführung eines `Merge`-Aufrufs. Das reine Entfernen eines Child-Tasks aus der Liste der gestarteten Tasks (und somit aus Φ), wie es bisher nach der Durchführung eines `Merge` der Fall war, ist nicht mehr ausreichend. Durch die Sibling-Relationen gibt es Verbindungen zwischen den einzelnen Datenstrukturkopien, die unterbrochen werden können, falls Knoten entfernt werden ohne diese Verbindungen zu betrachten. Würde man (bezogen auf Abbildung 4.10a) nach dem `Merge` von Child-Task T_{1a}

einfach den entsprechenden Knoten von D' aus der Datenstruktur-Hierarchie entfernen, dann würde man die transitive Verbindung zwischen den Datenstrukturen von Child-Task T_1 und Child-Task T_{1aa} verlieren. Es wird also ein Regelwerk benötigt, das für die einzelnen Synchronisationsprimitive angibt, wie Φ anzupassen ist. Dieses Regelwerk wird im Folgenden für die einzelnen Synchronisationsprimitive beschrieben.

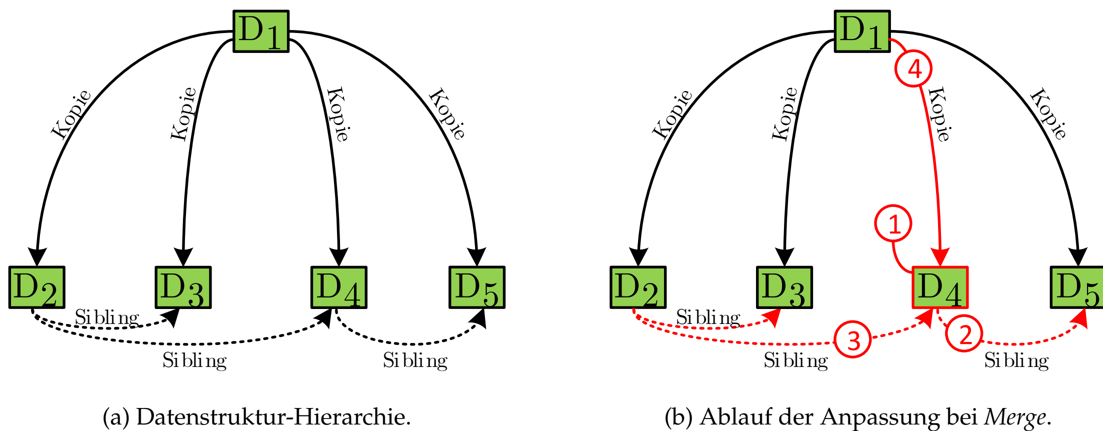
Werden bei `Spawn` oder `SpawnSibling` zusammenführbare Datenstrukturen übergeben, so müssen für diese neue Knoten in Φ angelegt werden. Abhängig von der Primitive unterscheidet sich dabei, wie ein neuer Knoten mit den vorhandenen Knoten verbunden ist. Bei `Spawn` wird der neue Knoten über eine Kopie-Relation mit der Datenstruktur des Parent-Tasks verbunden (siehe Datenstruktur D' zum mittleren Child-Task in Abbildung 4.11). In Φ_D der Datenstruktur im Parent-Task wird die neue Datenstrukturkopie in die Liste der Kopien der eigenen Datenstruktur aufgenommen. Für die Zusammenführung der Daten wird außerdem für die neue Datenstrukturkopie gespeichert, welche Versionsnummer die Datenstruktur des Parent-Tasks beim Kopieren hatte. Die eigene Versionsnummer der neuen Datenstrukturkopie wird initial auf dieselbe Versionsnummer gesetzt.

Bei `SpawnSibling` wird der neue Knoten auch über eine Kopie-Relation mit der Datenstruktur des Parent-Tasks verbunden. Zusätzlich wird der Knoten mit einer Sibling-Relation mit der ursprünglichen Datenstrukturkopie verbunden, die bei `SpawnSibling` übergeben wurde. Für die doppelte Verkettung wird auch hier der Knoten der neuen Datenstrukturkopie bei der Datenstruktur des Parent-Tasks und beim `SpawnSibling` aufrufenden Task eingetragen; bei ersterem in die Liste der Kopien der eigenen Datenstruktur, bei letzterem in die Liste der selbst erstellten Kopien. Als Beispiel dient hier die Datenstruktur D' des mittleren Child-Tasks in Abbildung 4.11. Für die Zusammenführung der Daten wird bei `SpawnSibling` im neu erstellten Knoten gespeichert, welche Versionsnummer die ursprüngliche Datenstruktur beim Kopieren hatte, die an `SpawnSibling` übergeben wurde. Zusätzlich wird gespeichert, bis zu welcher Versionsnummer die Veränderungen der ursprünglichen Datenstruktur bereits dem Parent-Task bekannt sind. Die Liste der zu ignorierenden Veränderungen wird von der ursprünglichen Datenstruktur übernommen. Somit werden Veränderungen, die die ursprüngliche Datenstrukturkopie beim nächsten `Merge` ignorieren soll auch von der neu erstellten Kopie beim `Merge` ignoriert.

Wenn der Child-Task gemerged wird, dann wird die Datenstruktur-Hierarchie der entsprechenden Datenstrukturen so angepasst, dass es für alle direkt verbundenen Datenstrukturen so aussieht, als wäre die Erstellung der Tasks durch ein `Spawn` geschehen. So wird verhindert, dass durch das eventuell notwendige Entfernen eines Knotens transitive Verbindungen zwischen Sibling-Datenstrukturen verloren gehen. Im Folgenden wird ein Algorithmus vorgestellt, mit dem Datenstruktur-Historien Φ (und somit der enthaltenen Datenstruktur-Hierarchien) aktualisiert werden können, sodass im Anschluss der einzelne Knoten ohne Seiteneffekte entfernt werden kann. Die Aktualisierung von Φ ist notwendig,

da sich durch den `Merge` die, dem Parent-Task bekannten, Versionsnummern der unterschiedlichen Datenstrukturkopien ändern können. Für eine bessere Lesbarkeit des Textes wird im Folgenden „Datenstruktur X soll gemerged werden“ anstelle von „der Child-Task, der die Datenstruktur X beinhaltet, soll gemerged werden“ geschrieben.

Gegeben sei als Beispiel eine Anwendung, aus der sich die in Abbildung 4.12a dargestellte Datenstruktur-Hierarchie ergibt[23]. Die Datenstruktur D_4 wird gemerged. Die hierbei aktualisierten Informationen liegen alle in Φ_{D_1} (siehe Kapitel 4.2.5), sodass keine Kommunikation mit anderen Tasks für die Durchführung der Aktualisierung notwendig ist. Es gibt zwei Datenstrukturkopien, die über eine Sibling-Relation mit D_4 verbunden sind: D_2 als ursprüngliche Datenstruktur und D_5 als eine von D_4 erstellte Sibling-Kopie.



(a) Datenstruktur-Hierarchie.

(b) Ablauf der Anpassung bei *Merge*.

Abbildung 4.12: Anpassung von Φ_{D_1} bei *SpawnSibling* und *Merge*.

Algorithmus zur Aktualisierung der Datenstruktur-Historie

Der Algorithmus zur Aktualisierung der Datenstruktur-Historie Φ (im Beispiel Φ_{D_1}) läuft in vier Schritten ab, die in Abbildung 4.12b zur Veranschaulichung eingezeichnet sind:

1. Aktualisierung der zur eigenen Datenstrukturkopie gespeicherten Informationen.

2. Aktualisierung der selbst erstellten Sibling-Datenstrukturkopien.

3. Aktualisierung der (transitiven) ursprünglichen Datenstrukturkopien.

4. Abschluss und Behandlung eines `Sync`-Aufrufs.

[23]Für den Algorithmus zur Aktualisierung von Φ_D sind nur Sibling-Kopien interessant, die gemeinsamen unter der Datenstruktur D liegen. Child-Kopien, die von diesen Sibling-Kopien erstellt wurden, beeinflussen den Algorithmus nicht, da die Veränderungen dieser Child-Kopien wieder in die einzelnen Sibling-Kopien gemerged werden. Ab diesem Zeitpunkt können die Sibling-Kopien so betrachtet werden, als hätten sie diese Veränderungen selbst durchgeführt.

Listing 4.21 zeigt den Algorithmus `UpdateDatastructureHistory` zur Aktualisierung einer Datenstruktur-Historie Φ. Dieser bekommt als Eingabe den Parent-Task (*`parentTask`*) und den gerade zusammengeführten Child-Task (*`currentlyMergedTask`*). Intern verwendet der Algorithmus des Weiteren die Variable *`datastructureCopies`*, die als Proxy für den Zugriff auf die in Φ_{D_1} gespeicherten Informationen zu den Datenstrukturkopien der Datenstruktur des Parent-Tasks dient. Die weiteren Variablen beinhalten einmal die Parent-Datenstruktur (*`parentStructure`*) und einmal die Datenstruktur, die aktuell zusammengeführt wurde (*`currentStructure`*).

Listing 4.21 Algorithmus zur Aktualisierung einer Datenstruktur-Historie Φ.

```
 1: function UpdateDatastructureHistory(parentTask,currentlyMergedTask)
 2:     var
 3:         datastructureCopies                        ▷ Copies of own datastructure
 4:
 5:         parentStructure                             ▷ The original datastructure
 6:         currentStructure                  ▷ The currently merged datastructure
 7:     end var
 8:                                           ▷ Step 1: Update current datastructure
 9:     parentStructure ← getStructureByTask(datastructureCopies, parentTask)
10:     currentStructure ← getStructureByTask(datastructureCopies, currentlyMergedTask)
11:     SetLastSeenVersion(currentStructure, parentStructure.GetVersion())
12:     SetOwnVersion(currentStructure, parentStructure.GetVersion())
13:     ClearIgnoreOps(currentStructure)
14:
15:                                          ▷ Step 2: Update descendant siblings
16:     RecursivelyUpdateDescendantSiblings(currentStructure)
17:
18:                                            ▷ Step 3: Update ancestor siblings
19:     RecursivelyUpdateAncestorSiblings(currentStructure, null)
20:
21:                                        ▷ Step 4: Finalize and Sync handling
22:     RemoveAncestorSiblingRelation(currentStructure)
23:     if !currentlyMergedTask.IsRunning() then ▷ Check if currently merged task finished
24:         RemoveStructureFromHierarchy(datastructureCopies, currentStructure)
25:     end if
26: end function
```

Als erster Schritt werden die zu D_4 gespeicherten Informationen aktualisiert (die genannten Felder beziehen sich dabei auf Tabelle 4.3). Dazu wird in Zeile 10 zu dem gerade zusammengeführten Task (*`currentlyMergedTask`*) die entsprechende Datenstrukturkopie gesucht[24]. Da die Parent-Struktur D_1 gerade den `Merge`-Vorgang abgeschlossen hat, wird die *Versionsnummer beim Kopieren (Spawn)* von D_4 auf die aktuelle Versionsnummer von

[24]Zur Vereinfachung wird hier von „Datenstrukturkopie“ gesprochen, obwohl es sich genauer um die „zu einer Datenstrukturkopie in Φ gespeicherten Metainformationen“ handelt.

D_1 gesetzt (Zeile 11). Dies bedeutet, dass die Parent-Datenstruktur zuletzt die angegebene Version der Datenstruktur D_4 gesehen hat. Auch die *eigene Aktuelle Versionsnummer (Spawn)* wird angepasst (Zeile 12). Abschließend wird die Liste der *zu ignorierenden Veränderungen (Sibling)* für D_4 geleert (Zeile 13). Die zu ignorierenden Veränderungen halten nach, welche eigenen Veränderungen der Parent-Datenstruktur bereits aus anderer Quelle (d.h. durch den `Merge` einer der ursprünglichen Datenstrukturen) bekannt sind. Nach dem erfolgreichen Merge sind der Parent-Datenstruktur alle eigenen Veränderungen bekannt, sodass keine mehr beim nächsten `Merge` ignoriert werden müssen.

Im zweiten Schritt werden in Zeile 16 alle Sibling-Kopien aktualisiert, die transitive Sibling-Nachkommen von D_4 sind (im Beispiel betrifft dies die Datenstruktur D_5). Die dabei aufgerufene Funktion `RecursivelyUpdateDescendantSiblings` ist in Listing 4.22 beschrieben und bekommt initial die zusammengeführte Datenstruktur (hier D_4) übergeben. In Zeile 5 werden die direkt abhängigen Sibling-Nachkommen ausgelesen und in der Variable *`descendants`* gespeichert. Für jede dieser Sibling-Kopien in *`descendants`* (Zeile 6) wird geprüft, ob der Parent-Kopie durch den durchgeführten `Merge` nun bereits Veränderungen in der Sibling-Kopie bekannt sind (Zeile 7). Falls ja, dann werden diese in die Liste der zu ignorierenden Veränderungen der Sibling-Kopie (*`structure`*) übernommen. So wird verhindert, dass die Veränderungen doppelt angewendet werden. Anschließend werden rekursiv (ausgehend von diesen Sibling-Kopien) alle weiteren Sibling-Kopien aktualisiert (Zeile 8). Dabei wird wiederum jeweils die Liste der zu ignorierenden Veränderungen aktualisiert (Zeile 7). Das ist notwendig, weil auch die Kopien von D_5 auf den Veränderungen von D_4 basieren. Für jede hier betrachtete Sibling-Kopie wird zusätzlich geprüft, ob ab diesem Zeitpunkt alle Veränderungen, die sie von der Datenstrukturkopie von der sie abstammt mitbekommen hat, der Parent-Datenstruktur bekannt sind (Zeile 9). Falls ja, dann wird die Sibling-Relation *zum Ursprung der Kopie* entfernt und die Datenstrukturkopie kann so behandelt werden, als wäre sie mit `Spawn` und nicht mit `SpawnSibling` gestartet worden (Zeile 10). Im Beispiel bedeutet dies, dass die Sibling-Relation von D_4 zu D_5 entfernt wird, weil alle Veränderungen an D_4, die an D_5 übergeben wurden, der Parent-Datenstruktur nun bekannt sind und D_5 durch die vollständige Liste der zu ignorierenden Veränderungen in der Lage ist, normal gemerged zu werden.

Im dritten Schritt werden in Zeile 19 in Listing 4.21 alle Sibling-Kopien aktualisiert, von denen die aktuell gemergete Datenstruktur abhängt. Die dabei aufgerufene Funktion `RecursivelyUpdateAncestorSiblings` ist in Listing 4.23 beschrieben und erwartet neben der aktuell zusammengeführten Datenstruktur die Angabe des Siblings für den nächsten Rekursionsschritt, der beim ersten Aufruf der Funktion noch nicht vorliegt. Als Ausgangspunkt für die Rekursion sucht der Algorithmus (Zeile 7 – 10) von der aktuell gemergeten Datenstruktur-Kopie *`currentStructure`* ausgehend entlang der *Referenzen auf den Ursprung der Kopie (Sibling)* nach derjenigen Kopie, die ursprünglich über `Spawn` ge-

Listing 4.22 Algorithmus zur rekursiven Aktualisierung der Sibling-Nachkommen.

```
 1: function RecursivelyUpdateDescendantSiblings(currentStructure)
 2:     var
 3:         descendants                                  ▷ Direct sibling descendants
 4:     end var
 5:     descendants ← GetDirectDescendants(currentStructure)
 6:     for structure ∈ descendants do                   ▷ Update every descendant
 7:         CheckAndUpdateIgnoreOps(structure, structure.GetMergedChanges())
 8:         RecursivelyUpdateDescendantSiblings(structure)
 9:         if AllAncestorChangesKnownToStructure(structure) then
10:             RemoveAncestorSiblingRelation(structure)
11:         end if
12:     end for
13: end function
```

startet wurde. Diese Kopie kann daran erkannt werden, dass es keinen Vorgänger-Sibling gibt, und dass sie somit für den Baum, den die Sibling-Relationen aufspannen, den Wurzelknoten darstellt (im Beispiel D_2). Von dieser Wurzel, die in der Variable `rootSibling` gespeichert wird, ausgehend wird nun wiederum rekursiv jeder Sibling-Nachkomme aktualisiert[25] (Zeile 11). Für jede dieser Sibling-Kopien wird wieder geprüft, welche ihrer Veränderungen durch den `Merge` der Datenstrukturkopie `currentStructure` der Parent-Datenstruktur bekannt gemacht wurden (Zeile 15). Abschließend wird auch hier nach dem Aufruf der Rekursion in Zeile 17 untersucht, ob der Parent-Datenstruktur ab diesem Zeitpunkt alle Veränderungen bekannt sind, die die Datenstrukturkopie `structure` von ihrem Sibling-Vorfahren `nextSibling` mitbekommen hat (Zeile 18). Falls ja, dann wird auch hier die Sibling-Relation zum Ursprung der Kopie entfernt (Zeile 19).

Abschließend entfernt die aktuell gemergete Kopie in Schritt 4 ihre Sibling-Relation zu ihrem Ursprung und kann ab diesem Zeitpunkt so behandelt werden, als wäre sie mit `Spawn` gestartet worden (Zeile 22 in Listing 4.21). Falls der gemergte Child-Task `Sync` aufgerufen hat, werden die aktualisierten Datenstrukturen an den Child-Task zurück übergeben und dieser kann seine Ausführung fortsetzen. Falls der Child-Task fertiggestellt ist (Zeile 23), wird die Datenstruktur aus den *Kopien der eigenen Datenstruktur (Spawn)* der Parent-Datenstruktur entfernt, und verschwindet somit aus Φ_{D_1} (Zeile 24). Die Datenstruktur-Historie Φ_{D_1} nach der Durchführung des Algorithmus für die Datenstrukturkopie D_4 ist in Abbildung 4.13 zu sehen. Die Datenstrukturen D_2, D_4 (falls `Sync` genutzt wurde) und D_5 sind nun unabhängig voneinander. Lediglich für die Sibling-Relation von D_2 zu D_3 ist nicht eindeutig bestimmbar, ob es die Relation noch gibt oder nicht, da das davon abhängt, welche Veränderungen D_3 von D_2 beim Aufruf von `SpawnSibling` übergeben bekommen hat.

[25] Beim Erreichen der aktuell gemergeten Datenstrukturkopie `currentStructure` wird die Rekursion in diesem Teilbaum abgebrochen.

Listing 4.23 Algorithmus zur rekursiven Aktualisierung der Sibling-Vorfahren.

```
1:  function RecursivelyUpdateAncestorSiblings(currentStructure,nextSibling)
2:      var
3:          rootSibling                                   ▷ Root ancestor sibling
4:          descendants                     ▷ Descendant siblings of nextSibling
5:      end var
6:      if rootSibling = null then      ▷ Find root ancestor sibling in initial call
7:          rootSibling ← currentStructure ▷ Initialize root sibling with current structure
8:          while rootSibling.HasAncestorSibling() do    ▷ Find root ancestor sibling
9:              rootSibling ← rootSibling.GetAncestorSibling()
10:         end while
11:         RecursivelyUpdateAncestorSiblings(currentStructure, rootSibling)
12:     else                                                      ▷ Do recursion
13:         descendants ← GetDirectDescendants(nextSibling)
14:         for structure ∈ descendants do
15:             CheckAndUpdateIgnoreOps(structure, structure.GetMergedChanges())
16:             if structure ≠ currentStructure then
17:                 RecursivelyUpdateAncestorSiblings(currentStructure, structure)
18:                 if AllAncestorChangesKnownToStructure(structure) then
19:                     RemoveAncestorSiblingRelation(structure)
20:                 end if
21:             end if
22:         end for
23:     end if
24: end function
```

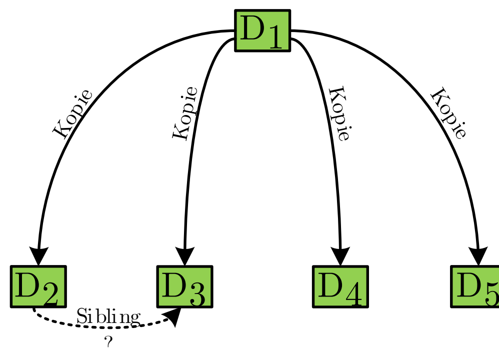


Abbildung 4.13: Ergebnis der Anpassung der Datenstruktur-Historie.

Die hier beschriebene Aktualisierung von Φ kann komplett innerhalb des Frameworks geschehen und ist damit unabhängig von den zusammenführbaren Datenstrukturen. Allerdings müssen zusammenführbare Datenstrukturen dennoch angepasst werden, damit

sie mit `SpawnSibling` genutzt werden können. Das liegt daran, dass es für die Nutzung mit `SpawnSibling` notwendig ist, dass der Mechanismus zur deterministischen Zusammenführung, der in einer Datenstruktur implementiert ist, die Benennung von zu ignorierenden Veränderungen erlaubt. Nur so kann gewährleistet werden, dass Modifikationen, die durch die Nutzung von `SpawnSibling` mehrfach beim Parent-Task ankommen, nicht die Ergebnisse verfälschen.

Task-Handles für Sibling-Tasks

Da die `SpawnSibling`-Primitive innerhalb eines Child-Tasks aufgerufen wird, bekommt der Parent-Task im Gegensatz zur Verwendung der `Spawn`-Primitive nicht automatisch ein Task-Handle, um seinen neuen Child-Task zu verwalten. Das Task-Handle ist nicht zwingend erforderlich, solange im Parent-Task nur `MergeAll` oder `MergeAny` benutzt werden. Für den Fall, dass der Entwickler jedoch explizit mit einem der Child-Tasks mergen möchte, die über `SpawnSibling` gespawnt wurden, muss es eine Möglichkeit geben, die Task-Handles neuer (über `SpawnSibling` gestarteter) Child-Tasks abzufragen.

Dazu wird die Funktionalität des Merge-Vorgangs und der Task-Handles erweitert. Wird ein Child-Task gemerged, so werden nicht nur die Veränderungen der geteilten Datenstrukturen übernommen, sondern automatisch auch eine Liste der neuen Sibling-Tasks, welche mit `SpawnSibling` durch den aktuell zusammengeführten Child-Task gestartet wurden, für den Parent-Task bereitgestellt[26]. Damit der Parent-Task diese Liste neuer Child-Tasks auslesen kann, wird das Task-Handle um die Funktion `GetSpawnedSiblings()` erweitert. Somit kann der Parent-Task für einen bekannten Child-Task prüfen, ob dieser weitere Sibling-Tasks gestartet hat. Die Liste beinhaltet dabei alle Child-Tasks, die vom Parent-Task noch nicht ausgelesen wurden. Da die Übertragung der Liste in die `Merge`-Primitive integriert ist, geschieht die Übertragung immer zu einem fest vorgegebenen Zeitpunkt (`Merge`-Aufruf) und in einem deterministischen Zustand (mit allen Sibling-Tasks, die bis zur Fertigstellung oder zum `Sync`-Aufruf des Child-Tasks gespawnt wurden). Daher kann der Entwickler über ein Task-Handle jederzeit deterministisch prüfen, ob von diesem Child-Task neue Sibling-Tasks gespawnt wurden (vor dem letzten durchgeführten `Merge`). So soll die Benutzung von `SpawnSibling` für den Entwickler erleichtert werden, ohne dass zusätzliche Synchronisationsprimitive für das Auslesen neuer Child-Tasks eingeführt werden müssen.

[26] Bei der Fertigstellung eines Child-Tasks (oder analog bei einem `Sync`-Aufruf) überträgt dieser alle neuen Sibling-Tasks, damit der Parent-Task diese abfragen kann.

4.2.11 Abort

Zusätzlich zu den bereits vorgestellten Synchronisationsprimitiven, die dazu dienen Informationen zwischen den Tasks auszutauschen, soll einem Task auch die Möglichkeit gegeben werden, seine eigene Berechnung zu beenden und die bisherigen Ergebnisse zu verwerfen. So kann ein Entwickler innerhalb eines Tasks entscheiden, ob die durchgeführten Berechnungen und deren Ergebnisse noch für die gesamte Anwendung benötigt werden, oder ob die Arbeit des Tasks eingestellt werden soll.

Abbruch von innen

Das Spawn & Merge Programmiermodell bietet dafür die `Abort`-Primitive an (die abstrakte Schnittstelle ist in Listing 4.24 beschrieben), die innerhalb eines Tasks aufgerufen werden kann, um diesen Task zu beenden und seine Änderungen zu verwerfen. Der beendete Task setzt bei sich selbst eine Markierung (*Abort-Flag*), dass er abgebrochen wurde. Dieses Abort-Flag wird beim Parent-Task beim `Merge` geprüft[27]. Ist es gesetzt, dann werden die Änderungen an den kopierten Datenstrukturen nicht mit den Datenstrukturen des Parent-Tasks zusammengeführt. Das betrifft auch alle Veränderungen, die durch die Child-Tasks des abgebrochenen Tasks durchgeführt wurden. Wenn ein Task `Abort` aufruft, dann beendet er seine Berechnungen und wartet darauf, dass seine Child-Tasks fertiggestellt werden. Das ist notwendig, da es bei einem `Sync`-Aufruf in einem dieser Child-Tasks noch eine Gegenstelle geben muss, die die Wartebedingung aufhebt. Sollte ein Child-Task mit einem `Sync`-Aufruf zu einem abgebrochenen Task kommen, dann informiert dieser den Child-Task in der Antwort auf das `Sync` darüber, dass dessen Parent-Task abgebrochen wurde, und der Child-Task seine Berechnungen ebenfalls einstellen kann. So wird rekursiv sichergestellt, dass der gesamte Teilbaum der Task-Hierarchie, der unter dem abgebrochenen Task liegt, über den Abbruch informiert wird. Somit gibt es durch das Abbrechen keine Tasks, die in einer nicht erfüllbaren Wartebedingung gefangen sind.

Listing 4.24 Definition der *Abort* Schnittstelle.

```
void Abort()
```

Ein abgebrochener Task kann nur die Veränderungen verwerfen, die nicht vorher schon mit dem Parent-Task zusammengeführt wurden. Gegeben sei als Beispiel die Anwendung in Listing 4.25. Der Child-Task `task1` wird gespawnt und hängt in Zeile 2 eine 1 an die kopierte Liste `list` an. Anschließend ruft er `Sync` auf (Zeile 3) und synchronisiert dabei die Liste `list` mit der originalen Liste `mList` des Parent-Tasks. Wenn der Parent-Task nun `task1` durch den Aufruf von `MergeAnyByHandle` in Zeile 11 gemerged hat, dann enthält seine Liste

[27] Ein Task mit Abort-Flag wird beim `Merge` als erfolgreich gemergeter Task angesehen. Andernfalls würde beispielsweise `MergeAnyByHandle(task1)` dauerhaft blockieren, wenn `task1` abgebrochen wurde.

`mList` die Veränderungen durch den Child-Task (d.h. die eingefügte 1). Nach dem `Sync` hängt der Child-Task eine 2 an die Liste an. Anschließend ruft er `Abort` auf (Zeile 5) und signalisiert seinem Parent-Task damit, dass seine Ergebnisse verworfen werden sollen. Der Parent-Task erkennt während der Durchführung von `MergeAnyByHandle` in Zeile 12, dass bei `task1` das Abort-Flag gesetzt wurde. Dementsprechend werden die Ergebnisse von `task1` verworfen und der Inhalt der Liste `mList` bleibt unverändert (`[1]`).

Listing 4.25 *Abort* Beispiel (von innen).

```
 1: void fct(MergeableList* list){
 2:   list->append(1);
 3:   Sync(RETURN_DIRECTLY, list);
 4:   list->append(2);
 5:   Abort();
 6: }
 7:
 8: MergeableList* mList = new MergeableList();
 9: TaskHandle task1 = Spawn(fct, mList);
10:
11: MergeAnyByHandle(task1); // mList is [1]
12: MergeAnyByHandle(task1); // mList remains [1]
```

Abbruch von außen

Zusätzlich zu der `Abort`-Primitive wird dem Entwickler noch eine Möglichkeit gegeben, einen Child-Task von außen abzubrechen. Diese Möglichkeit ist notwendig, da bei Spawn & Merge ein Task immer mit allen seinen Child-Tasks gemerged werden muss. Es kann allerdings vorkommen, dass ein Task die Ergebnisse eines Child-Tasks nicht mehr benötigt. Um dies zu ermöglichen, kann im Parent-Task von außen das Abort-Flag bei einem Child-Task gesetzt werden, wie in Listing 4.26 zu sehen. Dafür wird die Schnittstelle des Task-Handles um die Funktion `Abort` erweitert (Zeile 5). Wie schon bei den Child-Tasks eines Tasks, der sich selbst über die `Abort`-Primitive beendet hat, wird auch hier der abgebrochene Task nicht sofort beendet, um nicht erfüllbare Wartebedingungen in Child-Tasks zu vermeiden. Stattdessen wartet der Parent-Task darauf, dass der mit dem Abort-Flag markierte Child-Task gemerged wird und verwirft dabei die Ergebnisse des Child-Tasks. Außerdem wird der Child-Task im Falle eines `Sync`-Aufrufs darüber informiert, dass er abgebrochen wurde. Dies wird, wie auch bei den Child-Tasks eines von innen abgebrochen Tasks, mit der Zeit durch den gesamten abgebrochenen Teilbaum propagiert. Dass die Tasks in dem abgebrochenen Teilbaum der Task-Hierarchie nicht direkt beendet werden, bedeutet im Umkehrschluss, dass Rechenzeit in der Anwendung für Ergebnisse aufgewendet wird, die im Nachhinein verworfen werden. Wie bereits beschrieben ist dies jedoch notwendig, um

Deadlocks durch Sync-Aufrufe von Child-Tasks zu verhindern, die nicht mehr von einem Parent-Task entgegen genommen werden können, weil dieser sofort beendet wurde. Des Weiteren würde ein direktes Beenden von Child-Tasks potenziell nichtdeterministisches Verhalten einführen, da der Zeitpunkt der Terminierung nicht deterministisch ist. So könnte es beispielsweise dazu kommen, dass ein Child-Task abhängig vom Abort-Zeitpunkt *vor* oder *nach* dem Aufruf einer SpawnSibling-Primitive abgebrochen wird, wodurch der weitere Ablauf der Anwendung beeinflusst wird.

Listing 4.26 *Abort* Beispiel (von außen).

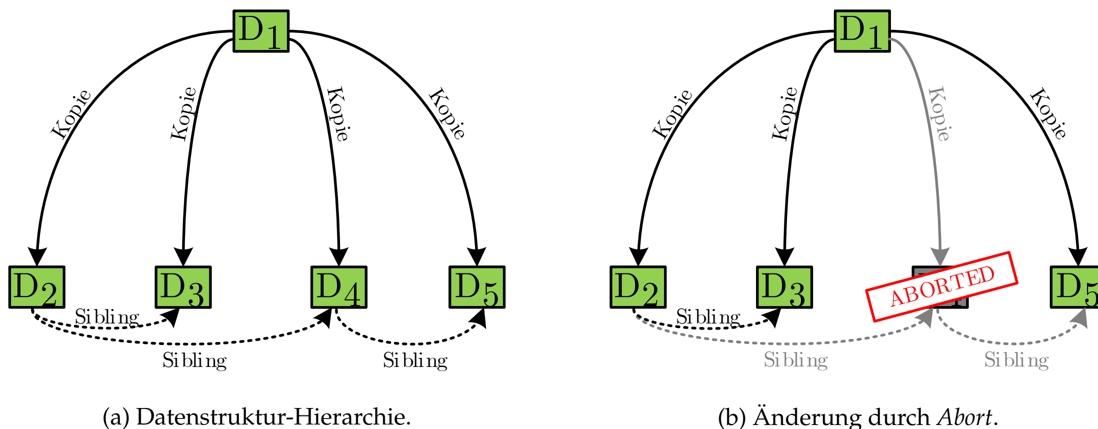
```
1: void fct();
2: TaskHandle task1 = Spawn(fct);
3:
4: // Abort Task-Handle
5: task1.Abort();
6:
7: MergeAll();
```

Aktualisierung der Datenstruktur-Historie Φ

Wird ein Child-Task abgebrochen, so muss auch die Datenstruktur-Historie Φ_D der zugehörigen Datenstruktur D angepasst werden, die die Abhängigkeiten der Datenstrukturkopien untereinander beschreibt. In Abbildung 4.14a ist erneut das Beispiel dargestellt, das bereits in Kapitel 4.2.10 genutzt wurden. Angenommen der Child-Task, zu dem die Datenstruktur D_4 gehört, wird abgebrochen. Um die transitive Abhängigkeit zwischen D_3 und D_5 nicht zu verlieren, wird die Datenstrukturkopie D_4 als „abgebrochen“ markiert und verbleibt nur noch als Informationsspeicher für die vorhandenen Sibling-Relationen, damit spätere Merge-Aufrufe weiterhin auf die entsprechenden Informationen zugreifen können. Der Informationsspeicher zu D_4 wird aus Φ_{D_1} entfernt, sobald er keine Sibling-Relationen zu anderen Datenstrukturkopien mehr besitzt (durch die Anpassung von Φ_{D_1} im Rahmen der Merge-Primitive). Somit ergibt sich die angepasste Datenstruktur-Hierarchie Φ_{D_1} in Abbildung 4.14b.

4.3 Beispielhafte Einbettung in C++11

Das Spawn & Merge Programmiermodell kann, ohne Änderungen an der Sprache selbst oder dem Typ-System, in General Purpose Languages (GPLs) integriert werden. In diesem Kapitel wird beispielhaft gezeigt, wie die einzelnen Komponenten von Spawn & Merge in der Programmiersprache C++11 abgebildet werden können. Zuerst wird dabei beschrieben, wie Tasks und zusammenführbare Datenstrukturen umgesetzt werden können. An-



(a) Datenstruktur-Hierarchie.

(b) Änderung durch *Abort*.

Abbildung 4.14: Anpassung der Datenstruktur-Historie Φ_{D_1} bei *Abort*.

schließend werden die Synchronisationsprimitive mit *Metaprogrammierung* und *Templates* realisiert, die seit C++11 [54] fester Bestandteil der Sprache sind.

Umsetzung der Komponenten von Spawn & Merge in C++

Tasks und zusammenführbare Datenstrukturen können in C++ als Objekte realisiert werden, die die entsprechenden Funktionalitäten kapseln. Tasks verwenden für die Ausführung der gespawnten Funktion dabei intern einen eigenen Thread. Die zu startende Funktion wird Tasks in Form eines Funktionspointers übergeben, der von der `Spawn`- oder `SpawnSibling`-Primitive durchgereicht wird. Die Parameter für Ausführung der Funktion werden entweder als Werteparameter (bei kopierbaren Datenstrukturen) oder bei zusammenführbaren Datenstrukturen als Referenzen auf die Datenstrukturkopien übergeben. Zusammenführbare Datenstrukturen sind Objekte, die neben der Funktionalität der Datenstruktur zusätzlich ein vom Spawn & Merge Framework bereitgestelltes Interface implementieren. Dieses Interface dient als Schnittstelle des Spawn & Merge Frameworks zu Datenstrukturen, die von Entwicklern für die Nutzung in Spawn & Merge angepasst wurden. Es definiert dementsprechend die Funktionalitäten, die eine zusammenführbare Datenstruktur für die Verwendung in Spawn & Merge bereitstellen muss, z.B. die Mechanismen für die Zusammenführung zweier Datenstrukturkopien oder das Erstellen einer Datenstrukturkopie. Das vollständige Interface für zusammenführbare Datenstrukturen ist im Anhang in Tabelle A.2 beschrieben.

Die Synchronisationsprimitive können mit einer unbegrenzten Anzahl unterschiedlicher Parameter aufgerufen werden, die jeweils unterschiedliche Funktionssignaturen voraussetzen. Sollen beispielsweise die Funktionen `Calculate` und `Verify` aus Listing 4.27 mit der `Spawn`-Primitive gestartet werden, dann ergeben sich die Funktionssignaturen

`(void) Calculate(int,int)` und `(void) Verify(list)`. Um diese Flexibilität, die für die Synchronisationsprimitive des Spawn & Merge Frameworks benötigt wird, zu erreichen werden Metaprogrammierung und Templates verwendet, die im Folgenden beschrieben werden.

Listing 4.27 Funktionssignaturen Beispiele.

```
1: void Calculate(int a, int b){
2:   // Calculate something
3: }
4:
5: void Verify(list l){
6:   // Verify the list
7: }
```

Metaprogrammierung und Templates

Metaprogrammierung ermöglicht es dem Entwickler im Quellcode einer Anwendung Vorlagen (Templates) für Klassen oder Funktionen zu definieren, die erst zur Compile-Zeit vom Compiler instanziiert werden. Dazu werden Platzhalter (z.B. für eine Funktion) definiert, die der Compiler im Rahmen des Compile-Vorgangs durch Typnamen oder bestimmte Werte ersetzen kann, um das Template zu vollwertigem C++-Code zu expandieren (der im folgenden Compile-Schritt auch von den Compiler-Optimierungen profitiert). In Listing 4.28 wird als Beispiel ein Funktions-Template für eine Summenfunktion `Sum` definiert, die zwei Parameter `a` und `b` erwartet. Für beide Parameter wird derselbe Typ `T` erwartet. Welchen Typ `T` einnehmen wird, ist durch das Template allerdings noch nicht festgelegt und entscheidet sich erst, wenn das Template durch den Compiler expandiert wird.

Listing 4.28 Template Definition.

```
1: template<typename T>
2: T Sum (T a, T b){
3:   return a + b;
4: }
5:
6: int A = 1;
7: int B = 2;
8:
9: Sum(A,B);
```

Zur Compile-Zeit prüft der Compiler für alle Funktionsaufrufe, ob diese einer bereits definierten Funktion zugeordnet werden können. Ist dies nicht der Fall (wie hier zum Bei-

spiel beim Aufruf in Zeile 9), dann prüft der Compiler, ob es ein Funktions-Template gibt, das so expandiert werden kann, dass es zu dem entsprechenden Funktionsaufruf passt. Falls es mehrere Templates gibt, die zu dem aktuellen Funktionsaufruf passen könnten, dann wird das Template ausgewählt, das die genauste Übereinstimmung mit dem Funktionsaufruf aufweist. Wurde ein passendes Funktions-Template gefunden und ausgewählt, dann generiert der Compiler anhand der Vorlage neuen Quellcode, indem die Typen für die Spezialisierung des Templates für die entsprechenden Platzhalter eingesetzt werden. Für den Aufruf der Funktion `Sum` in Zeile 9 in Listing 4.28 würde dementsprechend das Template wie in Listing 4.29 zu sehen expandiert werden, indem für `T` der Typname `int` eingesetzt wird. Im Gegensatz zu *Reflection* [37] wird hierbei keine Rechenzeit zur Laufzeit aufgewendet, da die Codegenerierung bereits zur Compile-Zeit durchgeführt wird.

Listing 4.29 Expandiertes Template.

```
1:  int Sum (int a, int b){
2:    return a + b;
3:  }
```

Variadische Templates

Templates können auch *variadisch* sein und somit eine beliebige Anzahl von Parametern eines Typs abbilden. Variadische Parameter werden durch drei aufeinanderfolgende Punkte „`...`“ definiert, die auf den Namen des Template-Parameters folgen. Listing 4.30 zeigt als Beispiel die Definition eines variadischen Summenfunktions-Templates. Der Summenfunktion `Sum` können hier beliebig viele Parameter des selben Typs übergeben werden. Zur Compile-Zeit expandiert der Compiler die Templates und ersetzt den variadischen Parameter durch die entsprechende notwendige Anzahl an Parametern. Für den Aufruf der Funktion `Sum` in Zeile 13 würde dementsprechend `T` durch `int` und `Rest` durch zwei weitere, aufeinander folgende `int`-Parameter ersetzt werden, die im Inneren der Funktion unter dem Parameter `rest` referenziert werden können.

Rekursive Expansion variadischer Templates

Damit ein variadisches Template mit einer beliebigen Anzahl an Parametern umgehen kann, werden variadische Templates häufig *rekursiv* definiert. Grundlegend bedeutet dies, dass ein variadisches Template sich solange selbst mit reduzierten Parametern aufruft, bis der variadische Parameter vollständig abgearbeitet wurde. Im Beispiel in Listing 4.30 ruft die Funktion `Sum` dazu wiederum sich selbst auf (Zeile 3). Als Parameter werden alle Parameter übergeben, die in `rest` zusammengefasst sind (also `B` und `C`). Der erste Parameter `a`, der in diesem Aufruf von `Sum` bereits addiert wurde, fällt dementsprechend weg und

Listing 4.30 Variadisches rekursives Template.

```
1:  template<typename T, typename... Rest>
2:  T Sum (T a, Rest... rest){
3:    return a + Sum(rest...);
4:  }
5:
6:  template<typename T>
7:  T Sum (T a){
8:    return a;
9:  }
10:
11: [...] // Initialize integer variables A, B and C
12:
13: Sum(A, B, C);
```

die Anzahl der Parameter wird um eins reduziert. Der Compiler expandiert daher das Template zu einer neuen Funktion `Sum`, die nur noch zwei `int`-Parameter beinhaltet. Beim letzten Aufruf der Funktion `Sum` in Zeile 3 ist nur noch ein Parameter (nämlich `C`) in `rest` enthalten. Nun hat der Compiler für den Aufruf der Funktion `Sum` mit nur einem Parameter zwei Funktions-Templates zur Auswahl. Zum einen das Template aus Zeile 1, wenn der variadische Parameter die Länge 0 hat, oder das Template aus Zeile 6. Hier greift der bereits beschriebene Mechanismus, dass der Compiler immer das Funktions-Template mit der größten Übereinstimmung auswählt. Im Beispiel ist dies das Funktions-Template in Zeile 6, weil dieses bereits nur einen Parameter erwartet. Die durch die Expansion dieses Templates entstehende Funktion `Sum` gibt nur den als Parameter übergebenen Wert zurück und bricht somit die Rekursion ab. Das Ergebnis der gesamten Rekursion ist dementsprechend die Summe aller Parameter.

Dieses Vorgehen wird zur Umsetzung der verschiedenen Synchronisationsprimitive genutzt, die eine flexible Anzahl an Parametern benötigen (d.h. `Spawn`, `MergeAllByHandle`, `MergeAnyByHandle`, `Sync` und `SpawnSibling`). In Listing 4.31 ist beispielhaft das Template für die `Spawn`-Primitive abgebildet. Der Typ `T` wird bei der Expansion des Templates durch den Typen des Funktionspointers der zu startenden Funktion ersetzt. Der variadische Typ `Arguments` expandiert zu den Typen der übergebenen Parameter für die Funktion. Die Parameter werden innerhalb der `Spawn`-Primitive rekursiv verarbeitet.

Bedingte Templates

Neben der Expansion von Templates kann Metaprogrammierung auch (durch die Verwendung *bedingter Templates*) dazu genutzt werden, den Compiler für bestimmte Typen Funktionen bereitstellen oder austauschen zu lassen.

Listing 4.31 Variadisches Spawn Template.

```
1: template<typename T, typename... Arguments>
2: TaskHandle Spawn(T fct, Arguments... arguments){
3:   // Spawn internals
4: }
```

Angenommen eine Template-Funktion soll nur für Referenzparameter bereitgestellt werden. In Listing 4.32 wird dazu das Funktions-Template `onlyReference` definiert. Die Metaprogrammierung verbirgt sich hierbei in der Definition des Rückgabewertes von `onlyReference` in Zeile 2. Anstelle des direkten Typnamens (`void`) wird hier eine Bedingung aufgestellt, die vom Compiler zur Compile-Zeit geprüft wird. Die Anweisung `enable_if<expr,type>` bedeutet für den Compiler, dass die Funktion nur dann erstellt werden darf, wenn die Bedingung `expr` erfüllt ist. Falls ja, dann bekommt die gesamte Anweisung als Typnamen den Typ `type` zugewiesen. Als Bedingung ist hier `is_reference<T>` angegeben, was für den Compiler bedeutet, dass er prüfen muss, ob es sich beim Typen von `T` um eine Referenz handelt. Ist dies der Fall, dann wird die Bedingung mit `true` evaluiert und der gesamte Ausdruck bekommt den Wert `void` und die Funktion wird expandiert. Ist es nicht der Fall, dann wird die Funktion nicht generiert und es tritt ein Compile-Zeit-Fehler auf der angibt, dass kein Template passend für den Funktionsaufruf expandiert werden kann. Um dem Programmierer eine bessere Fehlermeldung bieten zu können, kann man zusätzlich eine Funktion mit der inversen Bedingung erstellen (siehe Zeile 8), und in dieser dann beispielsweise mit `static_assert(false, msg)` dafür sorgen, dass der Compiler einen Fehler ausgibt und dabei die übergebene Nachricht `msg` anzeigt.

Listing 4.32 Metaprogrammierung von Bedingungen.

```
1: template<typename T>
2: enable_if<is_reference<T>::value,void>::type
3: onlyReference(T parameter){
4:   [...]
5: }
6:
7: template<typename T>
8: enable_if<!is_reference<T>::value,void>::type
9: onlyReference(T parameter){
10:   static_assert(false, "Only␣references␣allowed.");
11: }
```

Dieses, durch den Typen eines Parameters bedingte, Austauschen von Funktionen wird im Spawn & Merge Framework unter anderem für das Kopieren zusammenführbarer Da-

tenstrukturen innerhalb der `Spawn`- und `SpawnSibling`-Primitive genutzt. Listing 4.33 zeigt die vereinfachte[28] Definition der `CopyOne` Funktion, die von den `Spawn`-Primitiven für jeden Parameter aufgerufen werden und deren Implementierung sich je nach Typ des Parameters unterscheidet. `CopyOne` unterscheidet dabei, ob ein Parameter direkt kopiert werden kann (Werteparameter in Zeile 1–5) oder ob es sich um einen Referenzparameter handelt. In letzterem Falle muss zusätzlich (Zeile 8 – 9) geprüft werden, ob dieser Referenzparameter vom Typ `Mergeable` ist (d.h. eine zusammenführbarere Datenstruktur). Referenzparameter, die nicht zusammenführbare Datenstrukturen beinhalten, führen zu einem Fehler zur Compile-Zeit (Zeile 14 – 19).

Listing 4.33 Bedingte Templates für `CopyOne` (vereinfacht).

```
1:  template<typename T>
2:  typename enable_if<!is_pointer<T>::value,void>::type
3:  CopyOne(T parameter){
4:    // Call-by-value
5:  }
6:
7:  template<typename T>
8:  typename enable_if<is_pointer<T>::value &&
9:    is_base_of<Mergeable,T>::value,void>::type
10: CopyOne(T parameter){
11:   // Call-by-reference and is Mergeable
12: }
13:
14: template<typename T>
15: typename enable_if<is_pointer<T>::value &&
16:   !is_base_of<Mergeable,T>::value,void>::type
17: CopyOne(T parameter){
18:   // Call-by-reference, but is not Mergeable -> throw error
19: }
```

Compile-Zeit Fehler und Laufzeitfehler

Im Spawn & Merge Framework erlaubt die Metaprogrammierung des Weiteren die Überprüfung der korrekten Verwendung der Synchronisationsprimitive zur Compile-Zeit. Dies ist ein weiterer Vorteil, der sich aus der Nutzung von Templates ergibt. Im Folgenden wird daher beschrieben, welche Arten von Fehlern zur Compile-Zeit gefunden werden können und welche Fehler nur zur Laufzeit erkennbar sind.

[28]Für eine bessere Lesbarkeit wird hier auf das Entfernen der Referenzen und Pointer der Parameter verzichtet. Für die Überprüfung des Typs der Referenzparameter muss im tatsächlichen Quellcode zusätzlich der Pointer mit `remove_pointer<T>` entfernt werden und mit `is_referenz<T>` geprüft werden ob es sich um eine Referenz statt eines Pointers handelt.

Bei der Nutzung der `Spawn`- und `SpawnSibling`-Primitive kann zur Compile-Zeit sichergestellt werden, dass abgesehen von zusammenführbaren Datenstrukturen keine Referenzparameter übergeben werden dürfen (siehe vorangegangener Abschnitt). Wird doch ein Referenzparameter übergeben oder kann ein Template nicht expandiert werden (z.B. weil `Spawn` als erster Parameter kein Funktionspointer übergeben wurde), so wird dem Entwickler bereits zur Compile-Zeit ein entsprechender Fehler ausgegeben. Zusätzlich kann sichergestellt werden, dass `Sync` und `SpawnSibling` nur Datenstrukturen übergeben werden, die auch für die Verwendung mit der entsprechenden Primitive angepasst wurden. Dazu kann eine *Markierungsschnittstelle* verwendet werden, um eine Datenstruktur als „SpawnSibling-fähig“ oder „Sync-fähig“ zu kennzeichnen. So wird für die Aufrufe der Synchronisationsprimitive zur Compile-Zeit sichergestellt, dass sie mit der richtigen Art von Parametern aufgerufen werden.

Die Spawn & Merge Primitive dürfen (abgesehen von `InitSpawnMerge`) nur im Kontext eines laufenden Tasks aufgerufen werden. Ob ein Aufruf innerhalb eines Tasks geschieht kann nur zur Laufzeit entschieden werden. Falls dies nicht der Fall ist, dann sind die Primitive nicht funktionsfähig. Bei einem `Spawn`-Aufruf gäbe es dann, in diesem aktuellen Kontext, beispielsweise keinen Task, unter dem der neu gespawnte Child-Task eingeordnet werden kann. Auch bei einem `Merge`-Aufruf würden die Informationen fehlen, welcher Task nun auf seine Child-Tasks warten möchte. Für die Primitive `Sync` und `SpawnSibling` gilt des Weiteren die zusätzliche Bedingung, dass sie nicht im Kontext des Main-Tasks aufgerufen werden dürfen. Das liegt daran, da beide Primitive darauf angewiesen sind mit ihrem Parent-Task zu kommunizieren, den es beim Main-Task per Definition nicht gibt. Primitive, die in einem ungültigen Kontext aufgerufen werden führen daher zu einem Laufzeitfehler mit einer entsprechenden Fehlermeldung und zur Terminierung der Anwendung, da eine weitere fehlerfreie Ausführung nicht sichergestellt werden kann.

Diese Fehlerbetrachtung geschieht unter der Annahme, dass sowohl das Betriebssystem korrekt funktioniert, als auch dass der generelle vom Entwickler geschriebene C++-Code fehlerfrei ist. Nicht abgefangene Laufzeitfehler, die innerhalb eines Tasks auftreten (z.B. durch einen Zugriff auf eine nicht existierende Stelle eines Arrays), können vom Framework nicht behandelt werden und werden an den Main-Task propagiert. Dieser gibt den Fehler aus und terminiert die Anwendung, da keine deterministische Ausführung mehr gewährleistet werden kann.

4.4 Abbildung typischer Synchronisationsbeispiele

In diesem Kapitel wird für zwei typische Synchronisationsbeispiele gezeigt, wie die zugrundeliegenden Problemstellungen mit dem Spawn & Merge Programmiermodell abgebildet würden. Bei den Beispielen handelt es sich um das *Producer-Consumer Problem*

und das *Dining Philosophers Problem*. Im Folgenden wird dazu für jedes Problem zuerst beschrieben, wie es mit standardmäßigen Synchronisationsprimitven gelöst werden kann. Die Beschriebenen Lösungen (in Listing 4.34 und 4.36) sind dabei von A. S. Tanenbaum [92] übernommen, allerdings wurde der Quelltext an den hier genutzten Pseudocode angepasst, sowie die Kommentare übersetzt und gekürzt. Anschließend wird dargestellt, wie die entsprechende Problemstellung mit dem Spawn & Merge Programmiermodell gelöst werden kann und welche Besonderheiten bei der entsprechenden Lösung zu beachten sind. Abschließend wird noch ein weiteres von Tanenbaum angerissenes Synchronisationsproblem und dessen Abbildung mit Spawn & Merge betrachtet.

Producer-Consumer Problem

Beim Producer-Consumer Problem teilen sich zwei parallel ausgeführte Prozesse einen gemeinsamen Puffer (mit einer begrenzten Kapazität) für Waren (items). Einer der Prozesse produziert neue Waren und fügt diese in den Puffer ein, der andere Prozess entnimmt dem Puffer die Waren und verbraucht diese. Dabei darf der Producer-Prozess keine neuen Waren in den Puffer einfügen, wenn dieser bereits an seiner Kapazitätsgrenze ist. Ebenso darf der Consumer-Prozess keine nicht vorhandenen Waren aus dem Puffer entnehmen, wenn dieser aktuell leer ist. Die Prozesse müssen in diesen Fällen darauf warten, dass der andere Prozess entweder neue Waren produziert, oder vorhandene Waren verbraucht.

Producer-Consumer Problem - Message Passing

Tanenbaum beschreibt verschiedene Lösungsmöglichkeiten für das Producer-Consumer Problem, die mit unterschiedlichen Synchronisationsmechanismen umgesetzt wurden (Mutexe, Monitore, Semaphoren und Message Passing) [92]. In Listing 4.34 wird die auf Message Passing basierende Lösung dargestellt. Die Idee basiert darauf, dass es immer `N` Nachrichten gibt, die sich im Umlauf befinden (`N` ist in diesem Falle auf 100 initialisiert und gibt die Kapazität des Puffers an). Diese initialen leeren Nachrichten werden in Zeile 17–19 vom Consumer-Prozess generiert und an den Producer-Prozess gesendet. Beide Prozesse warten durch den Aufruf von `receive` in den Zeilen 8 und 21 darauf, dass sie eine Nachricht vom jeweils anderen Prozess erhalten. `receive` blockiert dabei solange, bis eine neue Nachricht vorliegt. Solange der Producer-Prozess immer nur dann eine neue Wareneinheit Produziert, wenn er eine leere Nachricht vorliegen hat, dann kann die Kapazität nicht überschritten werden. Solange der Consumer-Prozess nur dann eine Wareneinheit verbraucht, wenn er eine Nachricht, die eine Ware beinhaltet, vorliegen hat, dann kann ebenso die Menge der Waren nicht unter 0 fallen.

Listing 4.34 Producer-Consumer (nach Tanenbaum [92, Kapitel 2.3.8]).

```
1:  #define N 100                // Buffer capacity
2:
3:  void producer(void) {
4:    int item;
5:    message m;                 // Messagebuffer
6:    while (true) {
7:      item = produce_item();   // Item production
8:      receive(consumer, &m);   // Wait for empty-message
9:      build_message(&m, item);// Create message
10:     send(consumer, &m);      // Send items to consumer
11:   }
12: }
13:
14: void consumer(void) {
15:   int item, i;
16:   message m;
17:   for (i = 0; i < N; i++) {
18:     send(producer, &m);      // Send N empty-messages
19:   }
20:   while (true) {
21:     receive(producer, &m);   // Wait for sent item
22:     item = extract_item(&m);// Extract item
23:     send(producer, &m);      // Send empty-message
24:     consume_item(item);      // Consume item
25:   }
26: }
```

Producer-Consumer Problem - Spawn & Merge

Auch mit dem Spawn & Merge Programmiermodell gibt es mehrere Möglichkeiten die gegebene Problemstellung zu lösen. Ein Beispiel ist in Listing 4.35 dargestellt. Für den Producer und den Consumer wird jeweils ein Child-Task mit der Liste `itemList` gespawnt. Die Liste `itemList` repräsentiert den Puffer, über den die beiden Child-Tasks die Waren austauschen. Der Parent-Task wartet nun entweder in Zeile 32 auf einen der beiden Child-Tasks, oder explizit auf den Consumer (Zeile 34) oder den Producer (Zeile 36). Das ist abhängig von der Anzahl der Elemente in der Liste der Waren (`itemList`). Der Producer-Task erstellt immer genau eine Wareneinheit und hängt diese hinten an die Liste an sofern diese noch nicht voll ist (Zeile 7 und Zeile 8). Anschließend ruft er `Sync` auf, um die neue Wareneinheit an den Parent-Task zu melden. `Sync` blockiert dabei solange, bis der Parent-Task die Änderungen an der Liste übernommen hat. Danach setzt der Producer-Task seine Ausführung fort und beginnt durch die Endlosschleife (Zeile 5) erneut mit der Erzeugung einer Wa-

reneinheit. Der Consumer-Task prüft, ob in der Liste eine Wareneinheit zum Verbrauchen enthalten ist. Falls ja, dann wird die Ware aus der Liste entfernt (Zeile 18) und verbraucht (Zeile 19). Anschließend ruft er `Sync` auf, um zum einen den Parent-Task über die verbrauchte Ware zu informieren, und zum anderen um neue Waren in Empfang zu nehmen. Nach der Durchführung des `Sync` beginnt der Consumer-Task von vorne.

Die Verwendung der `MergeAny`-Primitive in Zeile 32 führt hierbei ein nichtdeterministisches Verhalten ein, wie es auch in der auf Nachrichtenaustausch basierenden Anwendung vorliegt. Um eine deterministische Ausführung zu erreichen, könnte in der Endlosschleife (Zeile 30) die `MergeAll`-Primitive verwendet werden. In dem Falle würde durch das Framework sichergestellt, dass beide Tasks immer abwechselnd mit dem Parent-Task synchronisiert werden und es könnte nie dazu kommen, dass mehr als zwei Wareneinheiten gleichzeitig existieren. Der genaue Ablauf, in dem der Parent-Task hier erstellte und verbrauchte Waren sieht, wäre somit bei jeder Programmausführung derselbe.

Dining Philosophers Problem

Beim Dining Philosophers Problem sitzt eine bestimmte Anzahl `N` an Philosophen um einen runden Tisch herum, in dessen Mitte eine Mahlzeit steht [34]. Jeder Philosoph wechselt immer zwischen dem Zustand *Denken (Thinking)* und *Essen (Eating)*. Auf dem Tisch liegen `N` Gabeln Jeder Philosoph benötigt zum Essen zwei Gabeln. Wenn ein Philosoph essen möchte, dann muss er schauen ob die beiden Gabeln neben seinem Teller aktuell ungenutzt sind. Ansonsten muss der Philosoph darauf warten, dass seine Nachbarn die Gabeln freigeben. Es können dementsprechend nicht alle Philosophen gleichzeitig essen.

Eine Besonderheit dieses Problems ist, dass es die Möglichkeit gibt, dass eine zirkuläre Wartebedingung erzeugt wird, die zu einem Deadlock führt. Angenommen jeder Philosoph nimmt zuerst die linke Gabel und hält diese fest, wenn er gerne etwas essen möchte. Wollen alle Philosophen gleichzeitig anfangen zu essen, dann nehmen alle gleichzeitig die linke Gabel und schauen, ob die rechte Gabel frei ist. Da die rechte Gabel allerdings blockiert ist (da der rechte Nachbar sie als seine linke an sich genommen hat), wartet nun jeder Philosoph darauf, dass sein rechter Nachbar die Gabel wieder freigibt, damit er selbst essen kann. Dies kann aber aufgrund der zirkulären Wartebedingung nie der Fall sein (Deadlock).

Dining Philosophers Problem - Semaphore

In Listing 4.36 ist eine deadlockfreie Umsetzung des Dining Philosophers Problems zu sehen, die auf Semaphoren basiert und so von Tanenbaum beschrieben wurde [92]. Die Umsetzung basiert auf der Idee, dass sich immer nur genau ein Philosoph in dem kritischen Bereich befinden darf, in dem er versucht Gabeln aufzunehmen. Falls er nicht beide Gabeln

Listing 4.35 Producer-Consumer mit Spawn & Merge.

```
1:  #define N 100                  // Buffer capacity
2:
3:  void producer(MergeableList* items){
4:    int item;
5:    while (true){
6:      if (items.size() < N){
7:        item = produce_item();
8:        items->append(item);
9:      }
10:     Sync(RETURN_DIRECTLY, items);
11:   }
12: }
13:
14: void consumer(MergeableList* items){
15:   int item;
16:   while (true){
17:     if (items->size() > 0){
18:       item = extract_first_item(items);
19:       consume_item(item);
20:     }
21:     Sync(RETURN_DIRECTLY, items);
22:   }
23: }
24:
25: MergeableList* itemList = new MergeableList();
26:
27: TaskHandle T_Prod = Spawn(producer, itemList);
28: TaskHandle T_Cons = Spawn(consumer, itemList);
29:
30: while (true){
31:   if (itemList.size() > 0 && itemList.size() < N){
32:     MergeAny();
33:   } else if (itemList.size() == N) {
34:     MergeAllByHandle(T_Cons);
35:   } else if (itemList.size() == 0) {
36:     MergeAllByHandle(T_Proc);
37:   }
38: }
```

gleichzeitig bekommen könnte, dann wartet er darauf, dass einer seiner Nachbarn mit Essen fertig wird (Zeile 14). Jeder Philosoph hält somit immer zwei Gabeln oder keine. Wird ein Philosoph mit Essen fertig, dann prüft er, ob seine Nachbarn links und rechts auf die Gabeln warten und nun essen könnten (Zeile 31 und 32). Er selbst geht wieder in den

denkenden Zustand über.

Dining Philosophers Problem - Spawn & Merge

Das Dining Philosophers Problem basiert auf dem Teilen von Ressourcen zwischen nebenläufigen Prozessen. Da das Spawn & Merge Programmiermodell das Teilen von Ressourcen nicht unterstützt, sondern Kopien der Ressourcen für jeden nebenläufigen Prozess bereitstellt, lässt sich das Dining Philosophers Problem nicht so umsetzen, dass die Aussage des Problems (der mögliche Deadlock) erhalten bleibt. Eine triviale Implementierung des Problems mit Spawn & Merge, in dem die Ressourcen (hier die Gabeln) für jeden Philosophen kopiert werden, ist der Vollständigkeit halber im Anhang B.4 zu sehen.

Unterschiedliche Aufgaben auf Listen

Ein weiteres Synchronisationsproblem, das von Tanenbaum erwähnt wird, um die Notwendigkeit von Synchronisationsmechanismen zu betonen [92, Kapitel 2.3.10], betrifft zwei Prozesse, die auf einer gemeinsamen Liste arbeiten. Der erste Prozess versucht die Liste zu sortieren, während der zweite Prozess versucht den Durchschnitt der Werte, die in der Liste gespeichert sind, zu errechnen. Ohne Synchronisationsmechanismen kann es passieren, dass der Durchschnitts-Prozess einzelne Werte mehrfach sieht, während er andere Werte gar nicht sieht. Das liegt daran, dass der Sortierungs-Prozess die Positionen der Werte innerhalb der Liste im Rahmen der Sortierung verändert, während der Durchschnitts-Prozess die Liste von vorne nach hinten durchläuft, um den Durchschnitt zu errechnen. Um ein valides Ergebnis zu erhalten, müssten beide Prozesse daher nacheinander versuchen einen exklusiven Zugriff auf die Datenstruktur zu erhalten.

Unterschiedliche Aufgaben auf Listen - Spawn & Merge

Hierbei handelt es sich um ein Beispiel, das sich sehr gut mit Spawn & Merge abbilden lässt (siehe Listing 4.37). Für jede der beiden Aufgaben (Berechnung des Durchschnitts und Sortierung der Liste) wird ein Child-Task gestartet. Jeder der Child-Tasks bekommt eine Kopie der Liste und kann somit den anderen Child-Task nicht beeinflussen. Daher können beide losgelöst voneinander ihre Aufgabe erfüllen. Der Durchschnitts-Task bekommt zusätzlich eine Variable übergeben, über die er seine Ergebnisse der Berechnung wieder an den Parent-Task übergeben kann. Beide Child-Tasks können außerdem weitere Child-Tasks erstellen, um ihre Aufgabe weiter zu parallelisieren. Abschließend wartet der Parent-Task darauf, dass beide Child-Tasks mit ihrer Arbeit fertig sind, und übernimmt deren Veränderungen an der Liste, sowie den errechneten Durchschnittswert, wieder in die eigenen Versionen der Datenstrukturen.

Listing 4.36 Dining Philosophers (nach Tanenbaum [92, Kapitel 2.5.1]).

```
1:  #define N 5                // Number of philosophers
2:  #define LEFT (i+N-1)%N     // ID of left neighbor
3:  #define RIGHT (i+1)%N      // ID of right neighbor
4:  [...]                      // Define THINKING, HUNGRY, EATING
5:
6:  typedef int semaphore;     // Definition of semaphore
7:  int state[N];              // State of philosophers
8:  semaphore mutex = 1;       // Mutex
9:  semaphore s[N];            // One semaphore per philosopher
10:
11: void philosopher (int i){
12:   while (true){            // Infinite-loop: the philosopher
13:     think();               // ..thinks
14:     take_forks(i);         // ..takes forks or blocks
15:     eat();                 // ..eats
16:     put_forks(i);          // ..puts forks down
17:   }
18: }
19:
20: void take_fork(int i){
21:   down(&mutex);            // Start of critical region
22:   state[i] = HUNGRY;       // Philosopher needs forks
23:   test(i);                 // Try to get two forks
24:   up(&mutex);              // End of critical region
25:   down(&s[i]);             // Block if no forks available
26: }
27:
28: void put_forks(int i){
29:   down(&mutex);            // Start of critical region
30:   state[i] = THINKING;     // Philosopher puts down forks
31:   test(LEFT);              // Left Phil. now able to eat?
32:   test(RIGHT);             // Right Phil. now able to eat?
33:   up(&mutex);              // End of critical region
34: }
35:
36: void test(int i){
37: if (state[i] == HUNGRY && state[LEFT] != EATING &&
38:     state[RIGHT] != EATING){
39:     state[i] = EATING;
40:     up(&s[i]);
41:   }
42: }
```

Listing 4.37 Unterschiedliche Aufgaben auf Listen mit Spawn & Merge.

```
1: void sort(MergeableList* list){
2:   // Sort list copy
3:   // Can be further parallelized
4:   [...]
5: }
6:
7: void average(MergeableList* list,MergeableValue* avgRes){
8:   // Calculate average of list-items
9:   // Can be further parallelized
10:  [...]
11:  avgRes = calculated_average;
12: }
13:
14: MergeableList* lst = new MergeableList();
15: MergeableValue* avgResult = new MergeableValue();
16:
17: // Initialize list
18: [...]
19:
20: Spawn(sort, lst);
21: Spawn(average, lst, avgResult);
22:
23: MergeAll();
```

4.5 Äquivalenz zu Semaphoren

Mit Spawn & Merge lassen sich alle Programme abbilden, die sich auch mit zu Semaphoren äquivalenten Primitiven umsetzen lassen. Um dies zu zeigen, wird in diesem Kapitel beschrieben, wie man eine *Semaphore* [33] (siehe auch Kapitel 2.1.2) nur durch Nutzung der von Spawn & Merge bereitgestellten Primitive umsetzen kann. Dies zeigt, dass dieselbe nebenläufige Ausführung, die eine Anwendung die auf Semaphoren basiert bietet, auch mit Spawn & Merge erreicht werden kann.

Für diesen Beweis wird angenommen, dass in einer Semaphore-basierten Anwendung kein nebenläufiger Zugriff auf denselben Speicherbereich erfolgt, solange dieser nicht durch eine Semaphore gelockt wurde. Diese Annahme ist realistisch, da ein anderweitiges Verhalten sehr wahrscheinlich ein Fehler in der Anwendung ist, der genau die Art von Nichtdeterminismus in die Anwendung einführt, die durch Spawn & Merge verhindert werden soll.

Umsetzung einer Semaphore mit Spawn & Merge

Der Beispielcode für die Umsetzung einer Semaphore mit Spawn & Merge ist in Listing 4.38 zu sehen. Die Semaphore wird hier durch eine Liste von Integern (`semaphoreList`) dargestellt. Das erste Element der `semaphoreList` stellt den aktuellen Wert der Semaphore dar (d.h. die Anzahl an Prozessen die noch auf den geteilten Speicherbereich zugreifen dürfen). Im Beispiel wird die Semaphore mit dem Wert 1 initialisiert (siehe Zeile 16). Die weiteren Elemente der `semaphoreList` sind die IDs der Tasks, die an der Semaphore darauf warten, Zugriff auf den geteilten Speicherbereich zu bekommen (initial warten keine Tasks).

In Anlehnung an die Semaphore-basierte Anwendung wird für jeden Thread im Semaphore-basierten System ein Child-Task gestartet. Jedem Child-Task wird eine Kopie der `semaphoreList` übergeben. Der Parent-Task erstellt für sich eine Liste aller Tasks (`waitForChildren` in Zeile 22), auf die er aktuell wartet. Zu Beginn der Ausführung beinhaltet diese Liste alle Child-Tasks. Anschließend wartet der Parent-Task durch den Aufruf von `MergeAnyByHandle` (Zeile 27) darauf, dass ein Child-Task aus dieser Liste gemerged werden möchte[29].

Anfordern des Locks

Wenn ein Child-Task (im Beispiel `task1` oder `task2`, die in Zeile 18 und 19 gespawnt wurden) über die Semaphore Zugriff auf den geteilten Speicherbereich erhalten möchte, dann hängt er seine eigene ID hinten an die `semaphoreList` an (Zeile 3). Anschließend wird zwei Mal `Sync` aufgerufen, und dabei die `semaphoreList` an den Parent-Task übergeben (Zeile 4 und 7). Der erste `Sync`-Aufruf sorgt dafür, dass der Parent-Task nicht weiter blockiert, falls er gerade unter anderem auf den entsprechenden Child-Task wartet (d.h. falls der Child-Task in `waitForChildren` enthalten ist). Wenn `MergeAnyByHandle` nicht mehr blockiert, setzt der Parent-Task seine Ausführung fort und prüft unter anderem, ob der Wert der Semaphore größer als 0 ist (Zeile 32).

Falls der Wert größer als 0 ist, kann die Semaphore einem wartenden Task Zugriff auf den geteilten Speicherbereich geben. Dazu prüft der Parent-Task, ob ein Task seine ID in die `semaphoreList` eingetragen hat (Zeile 34). Ist dies der Fall, dann wird das Task-Handle aus der `semaphoreList` herausgenommen (Zeile 35) und in Zeile 36 an die Liste `waitForChildren` angehängt (falls es nicht schon in der Liste enthalten ist). Anschließend wird der Wert der Semaphore in Zeile 37 um 1 reduziert. Beim nächsten `MergeAnyByHandle`-Aufruf des Parents ist der Child-Task, der Zugriff über die Semaphore erhalten hat, in der Liste `waitForChildren` enthalten und das zweite `Sync` (Zeile 7) kann

[29]Wird `MergeAnyByHandle` oder `MergeAllByHandle` mit einer Liste von Task-Handles aufgerufen, so wird diese Liste zu den einzelnen Task-Handle Parametern für die Primitive expandiert. Unabhängig von der Reihenfolge der Task-Handles in der Liste geschieht die Zusammenführung bei `MergeAnyByHandle` somit weiterhin nichtdeterministisch in der Reihenfolge, in der die Child-Tasks fertiggestellt werden.

Listing 4.38 Simulation einer *Semaphore* mit Spawn & Merge.

```
1:  void fct(MergeableList* semaphore){
2:    // Request access
3:    semaphore->append(this->TaskID);
4:    Sync(RETURN_DIRECTLY, semaphore);
5:
6:    // Wait for granted access
7:    Sync(RETURN_DIRECTLY, semaphore);
8:
9:    [...] // Critical section
10:
11:   // Release access lock
12:   semaphore->append(this->TaskID * -1);
13:   Sync(RETURN_DIRECTLY, semaphore);
14: }
15:
16: MergeableList* semaphoreList = new MergeableList(1);
17:
18: TaskHandle task1 = Spawn(fct, semaphoreList);
19: TaskHandle task2 = Spawn(fct, semaphoreList);
20:
21: // Semaphore management
22: List waitForChildren(task1, task2);
23: TaskHandle waitingHandle;
24:
25: while (!exitCondition){
26:   // Wait for Sync of a Child-Task
27:   MergeAnyByHandle(waitForChildren);
28:
29:   // Released access lock
30:   CheckForReleasedAccessAndCleanUp(semaphoreList);
31:
32:   if (semaphoreList[0] > 0){
33:     // Grant access
34:     if (CheckForWaitingTask(semaphoreList)){
35:       waitingHandle = ExtractWaitingTask(semaphoreList);
36:       AppendToList(waitForChildren, waitingHandle);
37:       DecreaseSemaphoreValue(semaphoreList);
38:     }
39:   } else {
40:     // Access denied
41:     RemoveWaitingTasks(semaphoreList, waitForChildren);
42:   }
43:
44:   [...] // Check break condition
45: }
```

durchgeführt werden. Der Child-Task hat nun Zugriff auf den geteilten Speicherbereich (Zeile 9).

Falls der Wert der Semaphore bei der Überprüfung gleich 0 ist, dann verbleiben die IDs wartender Tasks in der `semaphoreList`. Allerdings werden die entsprechende Task-Handles aus der Liste `waitForChildren` entfernt (Zeile 41). So wird sichergestellt, dass der zweite `Sync`-Aufruf (Zeile 7) dafür sorgt, dass die Child-Tasks solange blockieren, bis ihnen der Zugriff auf den geteilten Speicherbereich erlaubt wird (d.h. bis der Wert der Semaphore größer als 0 ist und wartende Tasks wieder in die Liste `waitForChildren` eingefügt werden).

Freigabe des Locks

Wenn ein Child-Task seinen Semaphore-Lock wieder abgeben möchte, dann hängt er seine negative ID hinten an die `semaphoreList` an und ruft `Sync` auf (Zeile 12 und 13). Der `Sync`-Aufruf weckt den Parent-Task auf, da `MergeAnyByHandle` aufhört zu blockieren. Der Parent-Task prüft nun, ob in der `semaphoreList` negative Task-IDs enthalten sind und entfernt diese (Zeile 30). Dabei wird für jede entfernte Task-ID der Wert der Semaphore wieder um 1 erhöht, da der entsprechende Task den Zugriff über die Semaphore abgegeben hat. Da das Task-Handle des Child-Tasks weiterhin in der Liste `waitForChildren` verbleibt, kann der Task auch im weiteren Verlauf der Anwendung erneut Zugriff über die Semaphore erbitten. Anschließend prüft der Parent-Task, ob es andere Child-Tasks gibt, denen nun Zugriff auf den geteilten Speicherbereich gewährt werden kann.

Anwendbarkeit

Da diese Umsetzung einer Semaphore ineffizient und umständlich ist, ist eine derartige Nachprogrammierung nicht erstrebenswert. Allerdings zeigt das Beispiel, dass mit dem Spawn & Merge Programmiermodell dieselbe nebenläufige Ausführung erreicht werden kann, die auch mit einem Semaphore-basierten System realisiert werden kann. Durch die Nutzung der `MergeAnyByHandle`-Primitive ist die resultierende Anwendung nichtdeterministisch. Dies ist notwendig, da ein Semaphore-basiertes System auch nichtdeterministisch ist. Generell sollte ein Entwickler bei der Entwicklung einer Spawn & Merge basierten Anwendung versuchen, standardmäßig die deterministischen `Merge`-Primitive zu nutzen, um die Vorteile einer deterministischen Anwendung zu erhalten. Nichtdeterministische `Merge`-Primitive sollten nur dann genutzt werden, wenn explizit nichtdeterministisches Verhalten in der Anwendung erforderlich ist.

4.6 Deadlockfreiheit

In diesem Kapitel wird das Zusammenspiel zwischen den vorgestellten Synchronisationsprimitiven analysiert um zu zeigen, dass Spawn & Merge basierte Anwendungen keine Deadlocks beinhalten, die sich aus diesen Primitiven ergeben. Ein Deadlock ist eine Situation, in der sich eine Anwendung intern selbst blockiert (verklemmt) und dadurch eine weitere Ausführung der Anwendung nicht mehr möglich ist.

Damit ein Deadlock beim Zugriff auf geteilte Ressourcen entstehen kann, müssen nach Coffman [24] vier Bedingungen gleichzeitig erfüllt werden:

1. Es darf immer nur ein Prozess gleichzeitig auf die geteilten Ressource zugreifen (*mutual exclusion*).

2. Ein Prozess hält bereits eine der geteilten Ressourcen und wartet auf Zugriff auf eine anderer geteilte Ressource, die von einem anderen Prozess gehalten wird (*hold & wait*).

3. Eine Ressource kann nur freiwillig von dem Prozess wieder freigegeben werden, der sie aktuell hält (*no preemption*).

4. Es gibt eine (möglicherweise transitive) zirkuläre Abhängigkeit, sodass mindestens zwei Prozesse aufeinander warten (*circular wait*).

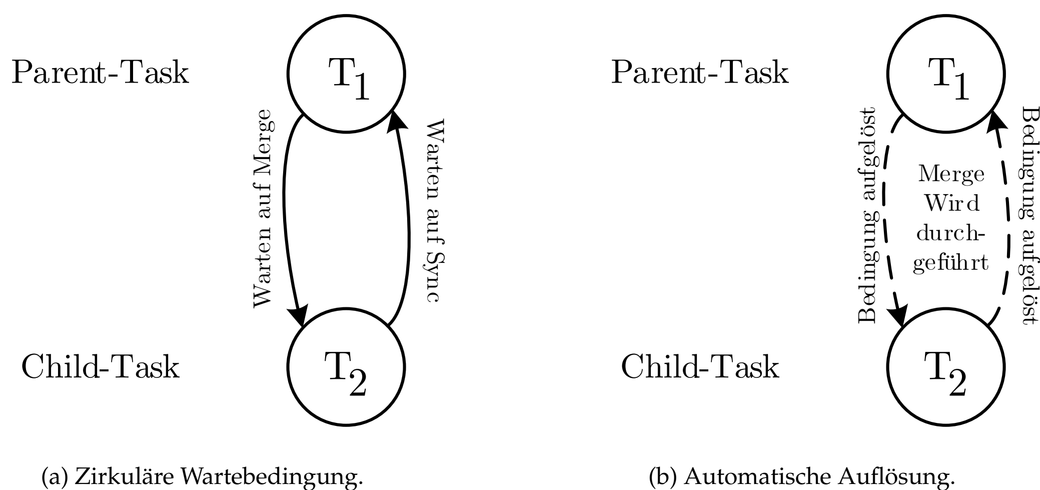
Nichterfüllbarkeit der Deadlock Bedingungen in Spawn & Merge

Unsere Analyse fokussiert sich auf die Anforderung, dass es eine zirkuläre Wartebedingung geben muss, damit ein Deadlock auftreten kann. Da es sich bei der Task-Hierarchie, die eine Spawn & Merge basierte Anwendung aufbaut, um eine Baumstruktur (d.h. einen azyklischen Graphen) handelt, kann eine zirkuläre Wartebedingung nur zwischen einem Parent-Task und einem seiner Child-Tasks auftreten. Eine zirkuläre Wartebedingung zwischen Parent-Task und Child-Tasks kann wiederum nur dann entstehen, wenn eine `Merge`-Primitive beim Parent-Task und die `Sync`-Primitive beim Child-Task aufgerufen wird (siehe Abbildung 4.15a). Abgesehen von diesen beiden Primitiven gibt es keine weiteren Primitive in Spawn & Merge, die eine Wartebedingung zwischen zwei Tasks zur Folge haben können (siehe Tabelle 4.4). Bei Betrachtung dieser möglichen zirkulären Wartebedingung fällt auf, dass der Parent-Task hier mit der `Merge`-Primitive auf die Fertigstellung des Child-Tasks wartet, während der Child-Task darauf wartet, dass der Parent-Task bereit ist ihn zu mergen. In diesem Falle würde der Parent-Task den Child-Task mergen und beide Tasks würden nicht weiter blockieren und ihre Ausführung fortsetzen (siehe Abbildung 4.15b). Somit wäre die zirkuläre Wartebedingung automatisch sofort aufgelöst. Das

Erreichen einer zirkulären Wartebedingung unter Benutzung der Spawn & Merge Synchronisationsprimitive ist somit nicht möglich, was wiederum bedeutet, dass es unmöglich ist einen Deadlock zu erreichen.

Primitive	Blockierverhalten
Spawn	blockiert nicht
MergeAll	wartet auf Child-Task
MergeAllByHandle	wartet auf Child-Task
MergeAny	wartet auf Child-Task
MergeAnyByHandle	wartet auf Child-Task
Sync	wartet auf Parent-Task
SpawnSibling	blockiert nicht
Abort	blockiert nicht

Tabelle 4.4: Blockierverhalten der Spawn & Merge Primitive.



(a) Zirkuläre Wartebedingung.

(b) Automatische Auflösung.

Abbildung 4.15: Deadlock zwischen Parent-Task und Child-Task.

Livelocks

Nachdem in Kapitel 4.5 gezeigt wurde, dass mit den Spawn & Merge Synchronisationsprimitiven eine Semaphore umgesetzt werden kann, ist es interessant zu betrachten, wie dieses Spawn & Merge basierte System ein Semaphore basiertes System simuliert, das sich in einem Deadlock befindet. Das beschriebene Semaphore basierte System befindet sich dann in einem Deadlock, wenn alle Child-Tasks auf den Zugriff auf die Semaphore warten, dieser aber nie gegeben wird (z.B. wenn die Semaphore im Beispiel mit 0 initiali-

siert worden wäre). Wenn alle Tasks auf den Zugriff warten und somit durch den zweiten `Sync`-Aufruf blockiert sind, dann ist die Liste `waitForChildren`, die beim Parent-Task gespeichert ist, leer. Das liegt daran, dass per Definition alle blockierten Child-Tasks aus der Liste `waitForChildren` entfernt werden. Daher wird der Parent-Task in einer Endlosschleife `MergeAnyByHandle` aufrufen, weil `waitForChildren` leer ist. Ein Aufruf von `MergeAnyByHandle` ohne Parameter sorgt dafür, dass die Anwendung ohne zu blockieren weiter ausgeführt wird (siehe Kapitel 4.2.7), weil es nichts gibt, auf das gewartet werden könnte.

Das zeigt, dass ein Spawn & Merge basiertes System immernoch einen *Livelock* [92] beinhalten kann. Allerdings ist dies für jegliche Programmiersprache der Fall, die die Erstellung einer Endlosschleife ermöglicht. Beispielsweise führt die Zeile „`while(true){}`“ bereits zu einem Livelock. Livelocks gehören somit nicht zu den Problemen, die durch ein Synchronisationssystem gelöst werden können.

Blockierende Aufrufe der Anwendung

Abgesehen von Livelocks gibt es noch weitere Probleme, die durch Synchronisationssysteme wie Spawn & Merge nicht gelöst werden können. So kann unter anderem nicht verhindert werden, dass der Entwickler blockierende Aufrufe (z.B. Systemaufrufe) mit Ziel außerhalb der Anwendung absetzt, deren Wartebedingung nie erfüllt wird. Ein Beispiel ist ein Aufruf, der auf einen Tastendruck auf der Tastatur wartet. Dennoch bleibt die Aussage, dass sich rein durch die Verwendung der Spawn & Merge Synchronisationsprimitiven kein Deadlock erreichen lässt und Spawn & Merge basierte Anwendungen somit deadlockfrei sind, bestehen.

Kapitel 5

Mechanismen für Spawn & Merge

In diesem Kapitel werden die in Spawn & Merge verwendeten Mechanismen beschrieben, die eine deterministische Zusammenführung von Datenstrukturkopien ermöglichen. Spawn & Merge setzt auf die Verwendung von *Operational Transformation (OT)* [36] als Mechanismus zur Konfliktauflösung. Im Folgenden wird beschrieben, wie OT funktioniert, und wie sie im Kontext von Spawn & Merge eingesetzt wird. Anschließend wird beschrieben, wie sich bestimmte Einschränkungen, die mit der Verwendung klassischer OT-Algorithmen einhergehen, durch die Verwendung eines speziell auf Spawn & Merge angepassten Algorithmus abmildern lassen.

5.1 Operational Transformation

Bei Spawn & Merge arbeitet jeder Task auf seiner eigenen Kopie der übergebenen Datenstrukturen (siehe Kapitel 4.2.3). Wenn ein Task seine Arbeit fertiggestellt hat, so hat er möglicherweise durch Veränderungen seine eigene Version der Datenstrukturkopie erstellt, die sich von den Kopien anderer Tasks unterscheidet. Um diese Konflikte der Datenstrukturveränderungen zwischen einem Parent-Task und seinen Child-Tasks in einer deterministischen Weise auflösen zu können, nutzt Spawn & Merge *Operational Transformation (OT)*, die erstmals von Ellis und Gibbs vorgestellt wurde [36]. OT ist dabei eine Technik, um eine Menge an nebenläufig durchgeführten Operationen in eine nicht-nebenläufige Reihenfolge von Operationen umzuwandeln (d.h. die Operationen zu serialisieren[1]).

Operational Transformation hat ihren Ursprung in der Forschung zu *computergestütztem kollaborativen Arbeiten (CSCW)* und zu *kollaborativem Editieren (collaborative editing)* [36]. Die Grundidee von OT ist, dass alle Teilnehmer (z.B. nebenläufig arbeitende Benutzer, die ein gemeinsames Dokument bearbeiten) jeweils auf einer eigenen lokalen Kopie der Daten

[1] Im Rahmen von Operational Transformation bedeutet das „Serialisieren“ von Operationen, dass diese in eine totale Ordnung gebracht werden. Diese Serialisierung unterscheidet sich somit von der Serialisierung im Rahmen der Datenstrukturen, die eine Datenstruktur auf einen Bytestream abbilden.

arbeiten und nur auf den Daten durchgeführte Operationen zwischen den Teilnehmern ausgetauscht werden. So wird verhindert, dass für jede Veränderung der (scheinbar) gemeinsam genutzten Daten diese gesperrt werden müssen (siehe Locking-Mechanismen in Kapitel 2.1.1). Systeme für das nebenläufige Editieren von Dokumenten sind dabei das hauptsächliche Anwendungsszenario für OT-Algorithmen. Ein Beispiel dafür ist *Google Wave*, das gezeigt hat, dass OT auch im großen Maßstab effizient arbeiten kann [99].

Im Gegensatz zu anderen Serialisierungstechniken, wie sie beispielsweise bei Datenbanktransaktionen angewendet werden, gibt es bei Operational Transformation keine Abbrüche und Rollbacks. Eine auf OT basierende Zusammenführung gelingt immer. Dieselbe Unterscheidung kann auch bei einem Vergleich mit *Transactional Memory* getroffen werden. Falls mehrere Threads dieselbe Zeile im Cache schreiben, dann wird mindestens einer der Threads zurückgesetzt (*rollback*). Bei OT hingegen gibt es keine Rollbacks.

Die Verwendung von Operational Transformation erfordert den Wechsel hin zu einer Betrachtung, die sich auf die durchgeführten Operationen konzentriert. Das bedeutet, dass jeder Task die Operationen aufzeichnen muss, die er auf seine Datenstrukturkopien angewendet hat. Im Rahmen der Zusammenführung (*Merge*) werden anschließend die nebenläufig durchgeführten Operationen der unterschiedlichen Tasks *transformiert* und in eine *deterministische*, nicht-nebenläufige Sequenz von Operationen gebracht.

5.1.1 Operationen

Datenstrukturen, deren nebenläufige Veränderungen unter Verwendung eines OT-Algorithmus deterministisch zusammengeführt werden sollen, müssen die *Operationen* (d.h. Veränderungen) aufzeichnen, die auf sie angewendet wurden. Zusätzlich muss die Datenstruktur einen OT-Algorithmus für die deterministische Konfliktauflösung bereitstellen. Solch eine Datenstruktur wird im Folgenden als *zusammenführbare Datenstruktur* bezeichnet[2] (siehe auch Kapitel 4.2.5).

Um den operationenzentrierten Ansatz zu motivieren, wird im Folgenden betrachtet, wie das Ergebnis einer nebenläufigen Ausführung formal beschrieben werden kann. Angenommen es gibt zwei Funktionen f und g, die eine Liste modifizieren. Dazu bekommen die beiden Funktionen eine Referenz auf die Liste als Argument übergeben und modifizieren diese. Des Weiteren führt die Funktion h die beiden Funktionen f und g parallel aus, wobei beiden Funktionen eine Referenz auf die Liste l übergeben wird. Somit können beide Funktionen nebenläufig die Liste l modifizieren. Außerdem bedeutet im Folgenden die Schreibweise $f(l) \rightarrow l_f$, dass die Funktion f eine Veränderung an Parameter l vorgenommen hat, wodurch sich l_f ergeben hat. Dies gilt allerdings nur dann, wenn keine andere Funktion l nebenläufig modifiziert hat.

[2] In Kapitel 5.1.4 wird diese Definition gelockert, da auch zusammenführbare Datenstrukturen denkbar sind, die andere Konfliktauflösungsmechanismen nutzen.

$$f(l) \rightarrow l_f \tag{5.1}$$

$$g(l) \rightarrow l_g \tag{5.2}$$

$$h(l) := f(l) \parallel g(l) \rightarrow l_h, \neg\exists x : x(l_f, l_g) \rightarrow l_h \tag{5.3}$$

In den Formeln 5.1 bis 5.3 sind die Listen l, l_f, l_g und l_h zu sehen. Die Schwierigkeit ist, das Ergebnis von $h(l)$ deterministisch ausdrücken zu können. Durch den nebenläufigen Zugriff auf die Liste l und den sich daraus ergebenden Nichtdeterminismus gibt es keine deterministische Funktion x, die das Ergebnis von $h(l)$ anhand der Ergebnisse von $f(l)$ und $g(l)$ ausdrücken könnte.

Um dieses Problem zu lösen, werden bei Operational Transformation nicht die Ergebnisse l_f und l_g, sondern die Operationen betrachtet, die von f und g erzeugt werden. Eine *Operation* ist dabei ein 4-Tupel bestehend aus der durchgeführten Operation (*OperationCode*), dem *Element* (falls es sich um eine einfügende Operation handelt), der *Position* an der die Operation angewendet werden soll und einem skalaren Versionszähler (*Version*), wie in Formel 5.4 beschrieben. Im Folgenden steht ops_x für eine Liste von Operationen, die auf den Parameter der Funktion x angewendet werden kann.

$$Operation := (OperationCode, Element, Position, Version) \tag{5.4}$$

$$f(l) \rightarrow ops_f, ops_f(l) \rightarrow l_f \tag{5.5}$$

$$g(l) \rightarrow ops_g, ops_g(l) \rightarrow l_g \tag{5.6}$$

$$h(l) := f(l) \parallel g(l) \rightarrow (ops_f, ops_g) \tag{5.7}$$

$$transform(ops_f, ops_g) \rightarrow ops_h \tag{5.8}$$

$$ops_h(l) \rightarrow l_h \tag{5.9}$$

In Formel 5.7 ist das Ergebnis deterministisch, da es unabhängig von der Ausführungsreihenfolge von f und g ist. Das Ergebnis ist ein einfaches Tupel, bestehend aus den Ergebnissen (durchgeführten Operationen) der Funktionen f und g. In Formel 5.8 kann anschließend eine *OT-Transformationsfunktion* $transform$ genutzt werden, um die Operationen ops_f und ops_g zu serialisieren. Das Ergebnis ist eine neue Sequenz von Operationen ops_h, die auf die Liste l angewendet werden kann, um das Ergebnis l_h für die Durchführung von $h(l)$ zu erhalten. Die Funktion $transform$ ist dabei eine komplexe Funktion, die Schlussfolgerungen über das Verhalten anderer Funktionen ziehen muss. Nach der Anwendung von $transform$ ergibt sich allerdings eine deterministische Beschreibung des Ergebnisses des parallelen Aufrufs von f und g. Dabei ist zu beachten, dass im Normalfall gilt, dass $transform(x, y) \neq transform(y, x)$ (d.h. $transform$ ist nicht kommutativ). Das liegt an der

Eigenschaft einiger OT-Algorithmen, dass diese ein *konsistentes* Ergebnis erreichen (z.B. jeder Teilnehmer einer Dokumentbearbeitung sieht am Ende dasselbe Ergebnis), dieses jedoch (aufgrund einer unterschiedlichen Reihenfolge der Transformationsparameter) bei Wiederholung derselben Eingaben nicht zwangsweise *deterministisch* ist. Die Reihenfolge, in der die Operationslisten zusammengeführt werden, ist dementsprechend entscheidend für ein deterministisches Ergebnis.

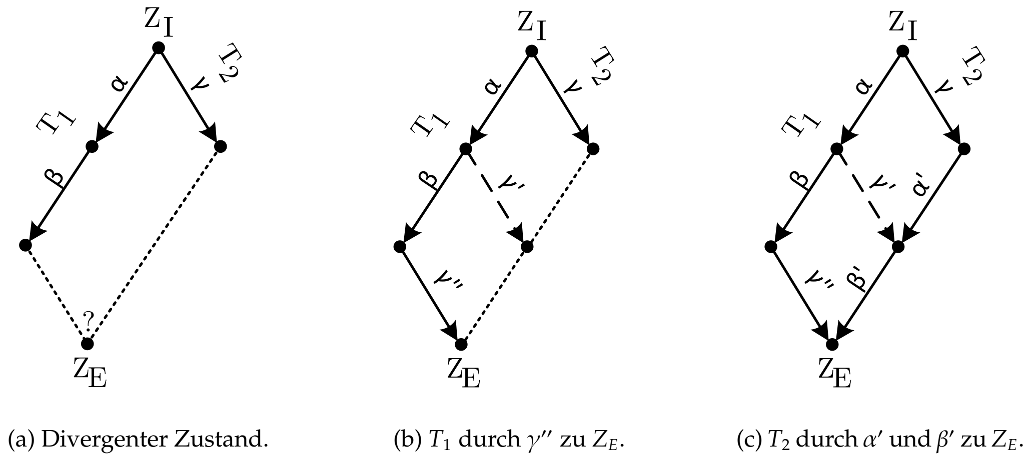
5.1.2 Funktionsweise

Ein OT-System beinhaltet zwei Algorithmen, die *Kontrollfunktion (Control Algorithm)* und die *Transformationsfunktion (Transformation Algorithm)* [36]. Die Kontrollfunktion trifft die Entscheidung, welche Transformationsfunktion auf welche Kombinationen der nebenläufigen Operationen angewendet werden muss. Die Transformationsfunktion wird genutzt, um die nebenläufigen Operationen gegeneinander zu transformieren. Dabei werden immer genau zwei Operationen gegeneinander transformiert. Wenn die Transformationsfunktion und die Kontrollfunktion korrekt sind, dann konvergieren die lokalen Daten bei jedem Teilnehmer des Systems zu einem gleichen, konsistenten Zustand, selbst wenn jeder Teilnehmer die Operationen in einer anderen Reihenfolge anwendet.

Abbildung 5.1 veranschaulicht an einem Beispiel die Funktionsweise der Kontrollfunktion, indem die durch angewendete Operationen entstehenden Zustandsübergänge dargestellt werden[3]. Zwei Teilnehmer T_1 und T_2 eines Systems, das OT nutzt, führen nebenläufig eine Reihe von Operationen auf ihrer lokalen Kopie eines gemeinsamen Dokumentes durch. Initial haben beide Kopien denselben Zustand Z_I. T_1 führt die Operationen α und β durch, während T_2 die Operation γ durchführt. In Abbildung 5.1a ist dargestellt, wie sich dadurch der Zustand der lokalen Kopien von T_1 und T_2 auseinanderentwickelt. Das OT-System hat nun die Zielsetzung es beiden Teilnehmern zu ermöglichen, denselben Endzustand Z_E zu erreichen, wenn sie die Operationen des jeweils anderen Teilnehmers bekommen. Erhält T_1 die Operationen, die T_2 auf den Daten durchgeführt hat, so entscheidet die Kontrollfunktion, wie diese Operationen gegen die eigenen Operationen transformiert werden (d.h. sie legt die Reihenfolge der Transformationsparameter fest). Die Kontrollfunktion ruft nun einzeln die Transformationsfunktion für jede eigene Operation (d.h. α und β) in Kombination mit allen empfangenen Operationen (hier nur γ) auf. Der Aufruf der Transformationsfunktion verändert dabei eine übergebene Operation x zur transformierten Operation x', die als Eingabe für den nächsten Transformationsfunktionsaufruf dient (siehe Abbildung 5.1b). Wendet T_1 nun die gegen α und β transformierte Operation γ'' an, so erreicht er den Zustand Z_E. Ebenso kann T_2 die empfangenen Operationen α und β jeweils gegen die lokale Operation γ transformieren und die daraus resultierenden transformierten Operationen α' und β' anwenden, um den Zustand Z_E zu erreichen

[3] Die Darstellungsweise ist dabei an die Zustandsraum-Graphen aus [71] angelehnt.

(siehe Abbildung 5.1c). Beide Teilnehmer befinden sich nun, trotz der unterschiedlichen Reihenfolge der Operationen, in einem konsistenten Zustand.



(a) Divergenter Zustand. (b) T_1 durch γ'' zu Z_E. (c) T_2 durch α' und β' zu Z_E.

Abbildung 5.1: Ablauf der Konfliktauflösung mit Operational Transformation.

Im Folgenden wird die Funktionsweise der Transformationsfunktion anhand eines Beispiels betrachtet. Als Beispiel für eine Transformationsfunktion wird dabei eine (für die Lesbarkeit leicht abgewandelte) Version der Funktion $XFORM_L$ aus [58] genommen, die in Listing 5.1 wiedergegeben ist. In Listing 5.1 ist zu sehen, dass es eine *TRANSFORM*-Funktion für jede mögliche Kombination von validen Operationen gibt. Dies sind hier alle möglichen Kombinationen von `insert`-, `delete`- und `no-op`-Operationen, wobei letztere nicht betrachtet werden, da sie andere Operationen nicht beeinflussen können. Innerhalb dieser *TRANSFORM*-Funktionen wird das weitere Vorgehen davon abhängig gemacht, wie die beiden Operationen zueinander positioniert sind, d.h. ob sie an derselben Stelle durchgeführt wurden, oder ob eine Operation (bezogen auf die Position) vor oder hinter einer anderen Operation liegt. Anschließend werden beide Operationen so transformiert, dass sie den Effekt der jeweils anderen Operation mit einbeziehen. Besonders hervorzuheben ist hierbei die Konfliktauflösung zwischen zwei konfligierenden `insert`-Operationen, sowie zwischen zwei konfligierenden `delete`-Operationen. Im Falle der `insert`-Operationen kann es sich von Anwendung zu Anwendung unterscheiden, wie die eingefügten Elemente nach der Transformation (Konfliktauflösung) angeordnet sein sollen (siehe Zeile 4). Stehen zwei `delete`-Operationen in Konflikt, so wurde das Element in beiden Kopien der Liste bereits gelöscht und die Operationen werden zu `no-op`-Operationen.

Um die Funktionsweise der Transformationsfunktion zu veranschaulichen, wird davon ausgegangen, dass zwei Prozesse P_1 und P_2 nebenläufig Operationen auf eine Liste L anwenden, die initial $[A, B, C]$ ist (siehe Abbildung 5.2). Angenommen der Prozess P_1 entfernt das Element C an der Stelle 2 (die Indizes der Liste starten bei 0), wodurch die Operation $op_{P_1} = delete(2)$ entsteht. Gleichzeitig fügt der Prozess P_2 das neue Element

Listing 5.1 Transformationsfunktion *TRANSFORM* nach [58].

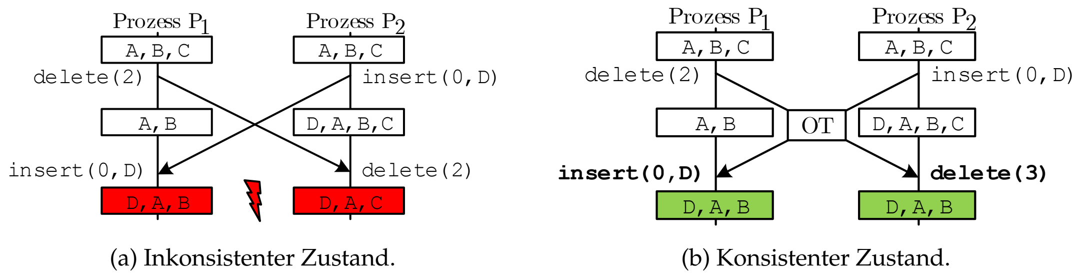
```
1:  function TRANSFORM(insert(i1, k1), insert(i2, k2)):
2:    if k1 <  k2: return(insert(i1, k1)    , insert(i2, k2 + 1))
3:    if k1 >  k2: return(insert(i1, k1 + 1), insert(i2, k2)    )
4:    if k1 == k2: // use application dependent priorities
5:
6:  function TRANSFORM(insert(i, k1), delete(k2)):
7:    if k1 <  k2: return(insert(i, k1)    , delete(k2 + 1))
8:    if k1 >  k2: return(insert(i, k1 - 1), delete(k2)    )
9:    if k1 == k2: return(insert(i, k1)    , delete(k2 + 1))
10:
11: function TRANSFORM(delete(k1), insert(i, k2)):
12:   if k1 <  k2: return(delete(k1)    , insert(i, k2 - 1))
13:   if k1 >  k2: return(delete(k1 + 1), insert(i, k2)    )
14:   if k1 == k2: return(delete(k1 + 1), insert(i, k2)    )
15:
16: function TRANSFORM(delete(k1), delete(k2)):
17:   if k1 <  k2: return(delete(k1)    , delete(k2 - 1))
18:   if k1 >  k2: return(delete(k1 - 1), delete(k2)    )
19:   if k1 == k2: return(no-op         , no-op         )
```

D am Anfang der Liste ein, wodurch die Operation $op_{P_2} = insert(0, D)$ generiert wird. Beide Prozesse wenden die eigenen Operationen op_{P_1} und op_{P_2} lokal an und senden sie anschließend an den jeweils anderen Prozess. Würden die beiden Prozesse die empfangenen Operationen ohne die Nutzung von Operational Transformation direkt anwenden, so würde dies zu inkonsistenten Ergebnissen führen ($[D, A, B]$ bei P_1 und $[D, A, C]$ bei P_2, siehe Abbildung 5.2a). Das liegt daran, dass sich bei Prozess P_2 der Index des Elementes C, das Prozess P_1 löschen wollte, von 2 zu 3 verschoben hat, als Prozess P_2 das Element D eingefügt hat. Der oben beschriebene OT-Algorithmus würde dementsprechend die Funktion $TRANSFORM(op_{P_1}, op_{P_2})$ anwenden. Durch Anwendung der Zeile 13 aus Listing 5.1 ergeben sich somit die transformierten Operationen $op'_{P_2} = insert(0, D)$ und $op'_{P_1} = delete(3)$. Wenn nun beide Prozesse die transformierten Operationen auf ihre lokale Liste anwenden, so erhalten beide als Ergebnis die Liste $[D, A, B]$ (siehe Abbildung 5.2b).

5.1.3 Klassifikation

OT-Algorithmen können anhand ihrer *Transformationseigenschaften* (*Transformation Properties (TP)*) klassifiziert werden [90]. Es gibt die Transformation Properties *TP1* und *TP2*.

Ein TP1-Algorithmus kann eine konsistente Konfliktauflösung zwischen *zwei* nebenläufig modifizierten Datenkopien erreichen, die von einem (beiden Prozessen bekannten) Zustand der Datenkopien ausgehen [85]. Bei der Konfliktauflösung müssen sich beide Pro-



(a) Inkonsistenter Zustand.

(b) Konsistenter Zustand.

Abbildung 5.2: Operational Transformation Beispiel.

zesse darüber einig sein, wessen Operationen als erster Parameter in die Transformation gegeben werden, und welche als zweiter Parameter. Hängt diese Reihenfolge von einer nichtdeterministischen Eigenschaft des Systems ab (z.B. dem Empfangszeitpunkt der Operationen), dann ist auch das Ergebnis der Konfliktauflösung nichtdeterministisch. Solange sich beide Prozesse auf dieselbe Reihenfolge einigen, ist das Ergebnis mindestens konsistent. Formel 5.10 beschreibt die TP1-Eigenschaft [60, 85], wobei O_1 eine einzelne Operation und O_1' das Ergebnis der Transformation mithilfe einer Transformationsfunktion TF von O_1 gegen O_2 darstellt ($O_1' = TF(O_1, O_2)$). Gleiches gilt für die Operationen O_2 und O_2'. Eine Anwendung der Operationen O_1 und O_2' nacheinander auf eine Liste hat dabei denselben Effekt wie die Anwendung der Operationen O_2 und O_1'.

$$\textbf{TP1: } O_1 \circ O_2' = O_2 \circ O_1' \tag{5.10}$$

Ein TP2-Algorithmus erreicht einen deterministischen konsistenten Zustand zwischen einer *beliebigen Anzahl* nebenläufig modifizierter Datenstrukturkopien, die von einem gemeinsamen Zustand abstammen [85]. Das Ergebnis der Konfliktauflösung ist dabei deterministisch, unabhängig von der Reihenfolge in der die Operationen der unterschiedlichen Prozesse zusammengeführt werden. Um zu erreichen, dass jeder Prozess seine Operationen mit jedem anderen Prozess zusammenführen kann, setzen TP2-Algorithmen auf die Nutzung von *Zustandsvektoren (Vector Clocks)*, um entscheiden zu können, welcher Prozess bereits welche Operationen von anderen Prozessen gesehen und angenommen hat [85]. Da diese Zustandsvektoren einen Zustand für jeden anderen an der Berechnung teilnehmenden Prozess beinhaltet, sind TP2-Algorithmen sehr teuer in Bezug auf den Speicherverbrauch und die notwendige Rechenzeit für die Transformation. Formel 5.11 beschreibt die TP2-Eigenschaft [60, 85]. Betrachtet werden hier zwei Teilnehmer eines Systems, die jeweils die Operationen $[O_1,\ O_2']$ beziehungsweise $[O_2,\ O_1']$ vorliegen haben (siehe vorangegangenes Beispiel). Empfangen beide Teilnehmer nun eine neue Operation O eines dritten Teilnehmers, so muss die gegen beide bereits vorliegenden Operationen transformierte Operation O'' bei beiden Teilnehmern gleich sein. Das bedeutet, dass in einem TP2-System eine Operation O gegen jede Kette $O_1, \ldots, O_n$ transformiert werden kann und das Ergebnis

der Transformation gleich ist, wenn auch die Konkatenation der Operationen O_1 bis O_n gleich ist. Dies ermöglicht auch dann ein deterministisches Ergebnis, wenn eine neue Operation O eines dritten Teilnehmers bei den ersten beiden Teilnehmern angewendet werden soll, die jeweils bereits unterschiedlich transformierte Operationen O_1 und O_2 vorliegen haben.

$$\textbf{TP2:}\ TF(TF(O,O_1),O_2') = TF(TF(O,O_2),O_1') \tag{5.11}$$

5.1.4 Alternativen zu Operational Transformation

Es gibt unterschiedliche Arten mit auftretenden Konflikten bei gemeinsam genutzten Datenstrukturen umzugehen. Zwei Möglichkeiten sind dabei *Konfliktauflösung* und *Konfliktvermeidung*. Operational Transformation, die im Kontext von Spawn & Merge vorgeschlagene Lösung für die Zusammenführung von Datenstrukturkopien, ist dabei ein Mechanismus zur Konfliktauflösung, da Konflikte im Rahmen der Transformationsfunktion aufgelöst werden.

Eine Alternative zur Konfliktauflösung stellt die Vermeidung von Konflikten dar. Eine bereits in Kapitel 2.1.1 beschriebene Möglichkeit ist die Verwendung von Locking-Mechanismen beim Zugriff auf geteilte Ressourcen, die allerdings aufgrund möglicher Race-Conditions keine deterministische Konfliktvermeidung darstellen. Eine weitere Möglichkeit ist die Nutzung von *Differential Synchronization* [38], bei der Veränderungen an einem geteilten Objekt zuerst anhand einer Schattenkopie (*shadow copy)* des geteilten Objektes nachvollzogen werden und anschließend als *Patch* zur Konfliktauflösung an andere Teilnehmer des Systems übertragen werden. Auch *Conflict-free Replicated Data Types (CRDTs)* [82] können genutzt werden, um eine Konvergenz zwischen geteilten Objekten zu erreichen, indem das Ergebnis der nebenläufigen Anwendung von Operationen auch dann konvergiert, wenn sich die Reihenfolge der Anwendung der Operationen unterscheidet (d.h. die Operationen sind *kommutativ*).

5.2 Operational Transformation in Spawn & Merge

In diesem Kapitel wird beschrieben, wie Operational Transformation im Kontext des Spawn & Merge Programmiermodells genutzt wird, um eine deterministische Programmausführung auf Applikationsebene zu ermöglichen. OT eignet sich dabei gut für den Einsatz als Standardmechanismus zur Konfliktauflösung, da OT ausführlich erforscht wurde und dabei gezeigt wurde, dass OT-Algorithmen für eine Vielzahl unterschiedlicher Datenstrukturen realisiert werden können (z.B. Strings, formatierter Text, Listen und Bäume [30, 63, 91]). Dabei ist es sowohl möglich TP1-Algorithmen zu verwenden, als auch TP2-Algorithmen.

Die Verwendung von TP1-Algorithmen hat dabei den Vorteil, dass diese günstiger zu realisieren und auch günstiger zur Laufzeit sind, wie bereits in Kapitel 5.1.3 beschrieben wurde. Ein deterministisches Ergebnis der (eigentlich nur konsistenten) TP1-Algorithmen wird dadurch ermöglicht, dass dem Spawn & Merge Framework durch die Spawn-Reihenfolge oder die Reihenfolge der übergebenen Parameter eines `Merge`-Aufrufs immer eine deterministische Sortierung der Tasks bekannt ist. Diese Reihenfolge kann vom Framework für eine deterministische Durchführung der Transformation der nebenläufig erzeugten Operationen genutzt werden.

Im Folgenden wird zuerst beschrieben zu welchem Zeitpunkt die Operational Transformation bei Spawn & Merge aufgerufen wird. Anschließend wird beschrieben, wie zusammenführbare Datenstrukturen die auf sie angewendeten Operationen nachverfolgen und welche Besonderheiten und Anforderungen sich durch die einzelnen Synchronisationsprimitive (insbesondere `Sync` und `SpawnSibling`) des Programmiermodells ergeben. Abschließend wird beschrieben, welche Nachteile sich durch die Verwendung von TP1-Algorithmen ergeben und wo der Ursprung dieser Nachteile liegt.

5.2.1 Operational Transformation im `Merge`-Vorgang

Die Zusammenführung von nebenläufig veränderten Datenstrukturkopien geschieht in Spawn & Merge zum einen zwischen genau zwei Tasks (Child-Task und Parent-Task) und zum anderen immer genau dann, wenn die `Merge`-Primitive durch den Parent-Task aufgerufen wurde (siehe Kapitel 4.2.7). Beim Aufruf der Transformationsfunktion der Datenstrukturen übergibt dabei das Framework immer die Datenstrukturkopien des Parent-Tasks als ersten Parameter und die des Child-Tasks als zweiten Parameter. Die Reihenfolge, in der die Datenstrukturkopien der Child-Tasks gemerged werden, hängt dabei von der Art der `Merge`-Primitive ab. Wird eine der deterministischen `Merge`-Primitive genutzt, so ist entsprechend auch diese Reihenfolge deterministisch (d.h. die Spawn-Reihenfolge oder die Reihenfolge der an die `Merge`-Primitive übergebenen Task-Handles). Dies ist notwendig, um ein deterministisches Ergebnis nach der Durchführung eines `Merge`-Aufrufes bei der Verwendung von TP1-OT-Algorithmen zu erreichen.

Sobald die Operationen beider Datenstrukturkopien gegeneinander transformiert wurden, werden die transformierten Operationen des Child-Tasks an die Liste der Operationen der Datenstruktur des Parent-Tasks angehängt. Handelt es sich bei dem durchgeführten `Merge`-Vorgang um einen `Sync`-Aufruf des Child-Tasks, so werden die transformierten Operationen des Parent-Tasks, die dem Child-Task noch nicht bekannt waren, an den Child-Task zurück übergeben und dort an die Liste der Operationen auf seiner Datenstrukturkopie angehängt (siehe Kapitel 4.2.9).

Welche Informationen dabei zwischen den Datenstrukturen für die Konfliktauflösung ausgetauscht werden, hängt von der Implementierung der Datenstrukturen und den ver-

wendeten Mechanismen ab. So können beispielsweise nur die Operationen (anstelle der gesamten Datenstruktur) übertragen werden, falls keine weiteren Informationen für die Durchführung des OT-Algorithmus notwendig sind. Dies ermöglicht es einem Entwickler bei der Erstellung einer zusammenführbaren Datenstruktur die zu übertragende Datenmenge zu reduzieren.

5.2.2 Datenstrukturen und versionsbasierter Merge

In diesem Kapitel wird beschrieben, welche Anpassungen an Datenstrukturen notwendig sind, um diese zu zusammenführbaren Datenstrukturen zu erweitern. Dabei wird insbesondere darauf eingegangen, welche Anforderungen sich aus den unterschiedlichen Spawn & Merge Synchronisationsprimitiven ergeben, und wie das Mergen zwischen Datenstrukturkopien, die zu unterschiedlichen Zeitpunkten erstellt wurden, ermöglicht wird.

Im Spawn & Merge Programmiermodell sind die Konfliktauflösungsmechanismen (z.B. die OT-Algorithmen) ein Teil der zusammenführbaren Datenstrukturen. Dementsprechend müssen zusammenführbare Datenstrukturen die auf sie angewendeten Operationen speichern, statt ihre Daten nur zu modifizieren. Die Implementierung der OT-Mechanismen innerhalb der Datenstrukturen soll zum einen eine hohe Wiederverwendbarkeit der Datenstrukturen ermöglichen, sodass die Entwicklung neuer Datenstrukturen selten notwendig ist. Zum anderen wird es den Entwicklern so ermöglicht ihre eigenen Datenstrukturen mit spezialisierten OT-Algorithmen zu entwickeln und anderen Entwicklern zur Verfügung zu stellen.

Damit das Spawn & Merge Framework diese Mechanismen von außen aufrufen kann, werden diese von den Datenstrukturen über eine Schnittstelle bereitgestellt. Die genaue Definition der Schnittstelle ist abhängig von der Implementierung des Spawn & Merge Frameworks, muss aber mindestens Funktionen für die folgenden Aufgaben bereitstellen:

1. Interne Bereitstellung und Aktualisierung der eigenen *Datenstruktur-Historie* Φ und der Historien abhängiger Datenstrukturkopien.
2. *Kopieren* einer Datenstruktur im Kontext von `Spawn` und `SpawnSibling`.
3. *Zusammenführung* mit einer Datenstrukturkopie im Kontext einer `Merge`-Primitive.
4. *Übernehmen des `Merge`-Ergebnisses* in die Datenstrukturkopie des Child-Tasks für `Sync`-Funktionalität (siehe Kapitel 4.2.9).

Datenstruktur-Historie Φ

Die Datenstruktur-Historie Φ, die in Kapitel 4.2.5 eingeführt wurde und für die Synchronisationsprimitive `SpawnSibling` in Kapitel 4.2.10 erweitert wurde, enthält alle Informa-

tionen die angeben, in welchem Verhältnis die Datenstrukturkopien einer Datenstruktur-Hierarchie zueinanderstehen (siehe dazu Tabelle 4.3).

Die in Φ enthaltenen Informationen sind für eine Zusammenführung zweier Datenstrukturkopien notwendig, da sie unter anderem angeben, welche Operationen die beiden Datenstrukturkopien bereits voneinander gesehen und übernommen haben. Dies verhindert eine doppelte Anwendung von Operationen. Daher müssen sie dem Algorithmus für die Zusammenführung durch die Datenstruktur zur Verfügung gestellt werden. Des Weiteren muss eine zusammenführbare Datenstruktur D immer dann Φ_D aktualisieren, wenn sich das Verhältnis zu anderen Datenstrukturkopien ändert. Die vorzunehmenden Änderungen wurden dabei für die einzelnen Primitive bereits in den Kapiteln 4.2.7, 4.2.9 und 4.2.10 beschrieben. Die Bereitstellung und Aktualisierung der Informationen, die in der Tabelle mit „(Sibling)“ gekennzeichnet sind, ist dabei nur dann notwendig, wenn die zusammenführbare Datenstruktur die Synchronisationsprimitive `SpawnSibling` unterstützen soll. Für eine Verwendung mit den Primitiven `Spawn`, `Merge` und `Sync` sind die in der Tabelle mit „(Spawn)“ gekennzeichneten Informationen ausreichend.

Kopieren

Wird eine zusammenführbare Datenstruktur kopiert, so muss sichergestellt werden, dass diese Kopie mit dem aktuellen Inhalt der Ursprungsdatenstruktur und mit einer leeren Liste an durchgeführten Operationen initialisiert wird. Zusätzlich ist es beim *Kopieren* einer zusammenführbaren Datenstruktur durch `Spawn` oder `SpawnSibling` notwendig, dass die zugehörige Datenstruktur-Historie Φ aktualisiert wird. Zu dieser Aktualisierung gehört auch das Setzen der initialen eigenen lokalen Versionsnummer auf die Versionsnummer der Ursprungsdatenstruktur zum Kopierzeitpunkt. Diese Information ermöglicht es dem Algorithmus für die Zusammenführung zu unterscheiden, welche Operationen seit dem Kopieren neu hinzugekommen sind, und welche zum Kopierzeitpunkt schon bekannt waren.

Zusammenführung

Die Zusammenführung zweier Datenstrukturkopien in Spawn & Merge geht über die reine Anwendung eines OT-Algorithmus auf zwei Listen von Operationen hinaus, da miteinbezogen werden muss, welche Operationen den beiden Datenstrukturkopien jeweils bereits bekannt sind. Dies wird insbesondere durch die `SpawnSibling`-Primitive verkompliziert, da diese dafür sorgen kann, dass dem Parent-Task Operationen von einem Child-Task bekannt sind, obwohl dieser bisher noch nicht selbst mit dem Parent-Task zusammengeführt wurde. Ein Beispiel zur Veranschaulichung dieser Problematik ist in Abbildung 5.3 zu sehen. Hier sind drei Tasks dargestellt, die Kopien derselben Liste haben. Eine Lis-

te wird dabei in der Form $[Element_1, Element_2, ...]_{Version}$ $Operation1_{Version}$ $Operation2_{Version}...$ beschrieben. Die Versionsnummer einer Operation gibt dabei an, zu welcher Version der Liste diese Operation geführt hat. Die Elemente der Liste sind hier zur Veranschaulichung angegeben, obwohl diese in der praktischen Umsetzung erst dann in der Liste enthalten sind, wenn die Operationen auf die Liste angewendet wurden. Vertikale Pfeile im Beispiel geben eine Veränderung der Liste durch die Operationen an, mit denen der Pfeil beschriftet ist. Horizontale Pfeile entsprechen Synchronisationen zwischen den einzelnen Tasks.

Im Beispiel spawnt der Task T_1 den Task T_2 mit einer leeren Liste. T_1 und T_2 führen nun nebenläufig die Operationen α_1 und β_1 auf ihrer Kopie der Liste aus. T_2 spawnt nun einen Sibling-Task T_3, der die von T_2 bereits mit β_1 veränderte Liste übergeben bekommt. Die Operation β_1 wird nun aktuell von zwei Tasks als eigene durchgeführte Operation, die an den Parent-Task gemeldet werden muss, angesehen. Dies ist notwendig, da es unklar ist, ob T_2 oder T_3 zuerst gemerged wird, und T_3 sich somit nicht darauf verlassen kann, dass der Parent-Task die Operation β_1 bereits kennen wird. T_2 und T_3 führen nun noch einmal nebenläufig die Operationen γ_2 und δ_2 auf ihrer Kopie der Liste aus. Wird nun T_3 von T_1 gemerged, so übernimmt T_1 die transformierten Operationen β_2' und δ_3' in seine Liste. Wird anschließend T_2 von T_1 gemerged, so muss der Transformationsalgorithmus erkennen, dass die erhaltene Operation β_1 der bereits bekannten Operation β_2' entspricht, während γ_2 noch nicht bekannt ist und nach der Transformation als γ_4'' in die eigene Liste eingefügt wird[4].

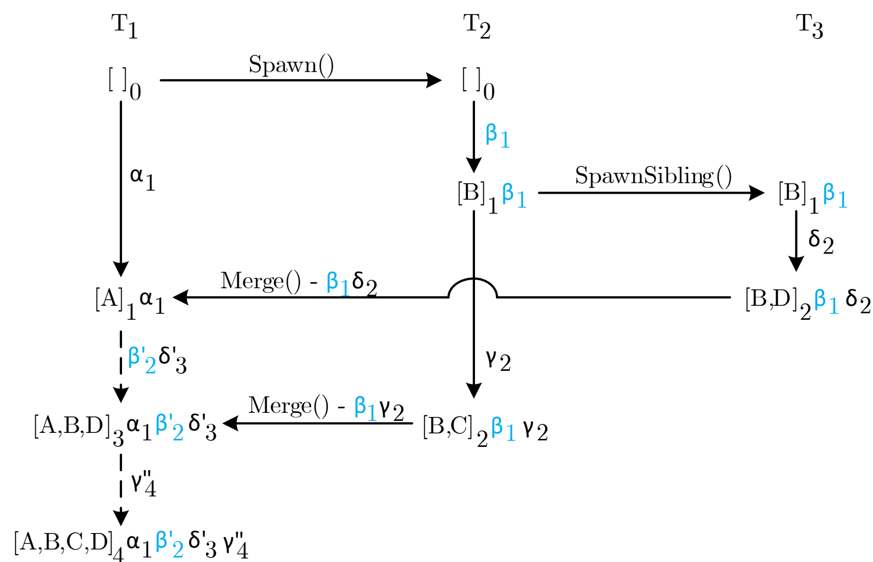


Abbildung 5.3: Ignorieren bereits bekannter Operationen.

[4]Obwohl bei der Liste von T_1 drei Operationen vorliegen wird γ_2 nur gegen α_1 und δ_3' transformiert, da die Effekte von β_2' bereits beachtet wurden.

Aus diesem Grund wird dem Zusammenführungsmechanismus neben zwei nebenläufig veränderten Datenstrukturkopien D_1 und D_2 auch die zugehörige Datenstruktur-Historie Φ übergeben. Aus diesen Informationen kann die Kontrollfunktion (durch ein Traversieren der darin abgebildeten Datenstruktur-Hierarchie) ableiten, welche Zustände den Datenstrukturen zuletzt bekannt waren und welche weiteren Operationen den einzelnen Datenstrukturkopien bereits bekannt sind und somit bei der Transformation ignoriert werden müssen. Bezogen auf das Beispiel aus Abbildung 5.3 bedeutet dies, dass beim `Merge` von T_3 die Datenstruktur-Historien dahingehend aktualisiert wurden, dass für T_2 angegeben ist, dass dessen Operation β_1 dem Parent-Task bereits als Operation β'_2 vorliegt und daher ignoriert werden muss.

Rückübertragung der Ergebnisse

Für die Nutzung der `Sync`-Primitive muss nach der Zusammenführung die Datenstrukturkopie des zusammengeführten Child-Tasks aktualisiert werden. Dabei ist es notwendig, dass die Operationen, die der Parent-Task durchgeführt hat, und die dem Child-Task noch nicht bekannt waren, gegen die Operationen des Child-Tasks transformiert und anschließend an den Child-Task zurück übertragen werden. Der naive Ansatz, einfach nach der durchgeführten Zusammenführung das Ergebnis des `Merge`-Aufrufs an den Child-Task zu übertragen, um sich die Transformation der Operationen des Parent-Tasks zu sparen, ist nicht möglich, da dies zu Problemen mit den gespawnten Tasks des Child-Tasks führen kann. Ein Beispiel dafür ist in Abbildung 5.4 dargestellt.

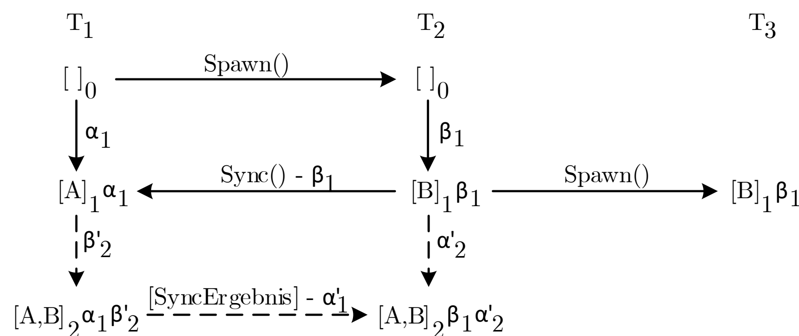


Abbildung 5.4: Rückübertragung der transformierten Parent-Operationen.

T_1 und T_2 führen nebenläufig die Operationen α_1 und β_1 auf der leeren Liste aus. T_2 spawnt nun mit dieser Liste einen neuen Task T_3, der die Operation β_1 kennt. Anschließend synchronisiert T_2 die Liste mit T_1 durch einen `Sync`-Aufruf und übergibt dabei β_1 an T_1. Dieser führt die Operational Transformation durch und wendet die transformierte Operation β'_1 als β'_2 bei sich an die Liste der Operationen an, da bereits eine Operation

für Version 1 vorhanden ist. Als letzten Schritt muss T_1 noch die gegen β_1 transformierte Operation α'_1 an T_2 senden, der diese dann wiederum an seine Liste der Operationen als α'_2 anhängt. Hätte T_1 seine gesamte Liste $[A, B]_2\ \alpha_1\beta'_2$ an T_2 übertragen, so könnte T_3 bei seiner Fertigstellung nicht die eigenen Operationen gegen die Operationen von T_2 transformieren, da es die Operation β_1 nicht geben würde. Aus diesem Grund ist für die Nutzung von `Sync` die Rückübertragung der transformierten Operationen (hier α'_1) des Parent-Tasks notwendig.

Datenstrukturen mit eingeschränkter Funktionalität

Die notwendigen Funktionalitäten einer zusammenführbaren Datenstruktur, um die Verwendung der Primitive `Sync` und `SpawnSibling` zu ermöglichen, sind komplex und nicht unbedingt mit jedem Mechanismus zur Konfliktauflösung realisierbar. Aus diesem Grund erlaubt das Spawn & Merge Programmiermodell dem Programmierer bei der Entwicklung von zusammenführbaren Datenstrukturen explizit anzugeben, ob eine Datenstruktur mit `Sync` und/oder `SpawnSibling` genutzt werden kann, oder nicht. Eine Möglichkeit für diese Kennzeichnung ist die Verwendung einer *Markierungsschnittstelle*, die den Vorteil hat, dass zur Compile-Zeit überprüft werden kann, ob die übergebenen Datenstrukturen mit den entsprechenden Primitiven genutzt werden können, oder nicht (siehe auch Kapitel 4.3). Dies vereinfacht die Entwicklung zusammenführbarer Datenstrukturen, die nicht zwingend im Kontext von `Sync` und `SpawnSibling` benötigt werden.

5.2.3 Alternative Mechanismen in Spawn & Merge

Da im Rahmen von Spawn & Merge die Realisierung der Konfliktauflösung innerhalb der zusammenführbaren Datenstrukturen realisiert wird, ist es für einen Entwickler auch mögliche eine Konfliktauflösung zu realisieren, die nicht auf Operational Transformation, sondern auf einem alternativen Verfahren basiert. Die datenstrukturinternen Mechanismen werden dabei, unabhängig von der Anwendung selbst, durch das Framework aufgerufen. Standardmäßig bekommt eine zusammenführbare Datenstruktur D_1 zur Konfliktauflösung eine Referenz auf die Datenstrukturkopie D_2, mit der sie zusammengeführt werden soll, sowie die in Kapitel 4.2.5 beschriebene Datenstruktur-Historie Φ_{D_1}, die die Operational Transformation ermöglichen. Neben komplexen Konfliktauflösungen oder Konfliktvermeidungen ist es so auch beispielsweise möglich, Datenstrukturen zu implementieren, die eine exklusive Nutzung für den Child-Task garantieren und deren Veränderungen somit nicht zusammengeführt, sondern nur übernommen werden brauchen. Weitere Möglichkeiten für alternative Datenstrukturen sind nicht veränderbare Datenstrukturen, die beim Child-Task nicht verändert werden können und somit auch nicht wieder zusammengeführt werden müssen, und Datenstrukturen, für welche die Konfliktauflösung sehr einfach

zu realisieren ist (z.B. Increment/Decrement Datentypen, Listen an die man nur Daten anhängen kann, oder Datentypen bei denen die letzte Schreibaktion gewinnt). So können mit anwendungsbezogenem Wissen spezialisierte Datenstrukturen in kritischen Bereichen einer Anwendung eingesetzt werden, um OT-Aufrufe zu vermeiden, wenn bereits bekannt ist, dass diese nicht benötigt werden.

Bei der Nutzung alternativer komplexer Mechanismen können weitere Metainformationen zur Durchführung notwendig sein. In diesem Falle muss das Spawn & Merge Framework für die Unterstützung dieser Datenstrukturen angepasst werden.

5.2.4 Kritik an Operational Transformation

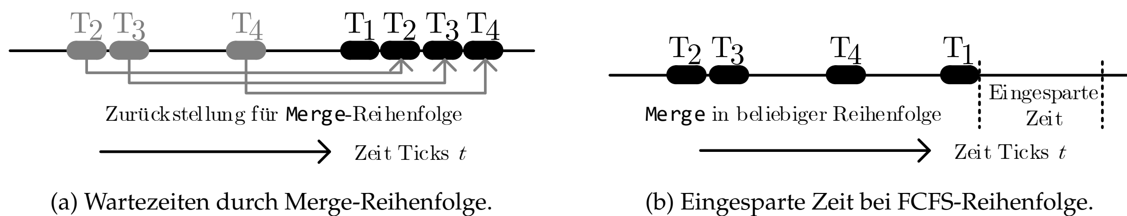
Das Spawn & Merge Programmiermodell erlaubt Entwicklern sowohl die Nutzung von TP1-OT-Algorithmen, als auch von TP2-OT-Algorithmen. Allerdings haben beide Arten eigene Nachteile, die im Folgenden betrachtet werden.

TP2-Algorithmen erreichen ein deterministisches Ergebnis, unabhängig von der Reihenfolge, in der divergente Kopien zusammengeführt werden. Allerdings sind diese Algorithmen sehr teuer im Hinblick auf den benötigten Arbeitsspeicher und die Komplexität der Transformationsalgorithmen [60]. Das liegt unter anderem daran, dass Zustandsvektoren genutzt werden müssen, um für jede Operation nachzuhalten, welche Operationen dieser Operation bereits bekannt waren [77].

Um die Komplexität der TP2-Algorithmen zu umgehen, sieht das Spawn & Merge Programmiermodell die Nutzung von TP1-Algorithmen vor. Das deterministische Ergebnis bei der Nutzung der eigentlich nur konsistenten TP1-Algorithmen wird dabei dadurch erreicht, dass fertiggestellte Child-Tasks in einer festen (deterministischen) Reihenfolge zusammengeführt werden (siehe auch Kapitel 4.2.7).

Diese einzuhaltende Reihenfolge für die Zusammenführung sorgt allerdings dafür, dass Wartezeiten beim Parent-Task entstehen können. Dies ist der Fall, wenn Child-Tasks bereits fertiggestellt wurden und theoretisch gemerged werden können, der Parent-Task aber erst auf die Fertigstellung und das Mergen eines anderen Child-Tasks warten muss, um die korrekte Reihenfolge und damit ein deterministisches Ergebnis zu erhalten. Ein Beispiel dafür ist in Abbildung 5.5a zu sehen. Hier wartet ein Task auf seine Child-Tasks $T_1 - T_4$, um sie in ebendieser Reihenfolge zu mergen. Die Tasks $T_2 - T_4$ sind früher fertiggestellt worden als T_1. Daher muss das Mergen der Tasks T_2 bis T_4 zurückgestellt werden, bis T_1 fertiggestellt und gemerged wurde. Die Möglichkeit, Tasks in beliebiger Reihenfolge zusammenführen zu können, würde insbesondere die Zusammenführung in einer *First-Come-First-Serve (FCFS)*-Reihenfolge ermöglichen und somit die eingeführten Wartezeiten reduzieren (siehe Abbildung 5.5b).

Die Häufigkeit des Auftretens einer solchen Wartesituation ist abhängig von der jeweiligen Anwendung. So kann es Anwendungen geben, bei denen nie gewartet werden



(a) Wartezeiten durch Merge-Reihenfolge.

(b) Eingesparte Zeit bei FCFS-Reihenfolge.

Abbildung 5.5: Einfluss der einzuhaltenden `Merge`-Reihenfolge.

muss und andere Anwendungen, in denen ein Task mit langer Laufzeit früh gemerged werden muss und die Wartezeit effektiv für andere `Merge`-Vorgänge genutzt werden könnte. Die effektiv eingesparte Zeit ist dabei wiederum davon abhängig, wie rechenintensiv die einzelnen vorgezogenen `Merge`-Vorgänge sind. Sind die `Merge`-Vorgänge sehr kurz, so ist die eingesparte Zeit sehr gering, da sie der Dauer der vorgezogenen `Merge`-Vorgänge entspricht.

Im folgenden Kapitel wird ein Operational Transformation Algorithmus entwickelt, der ein deterministisches Ergebnis für eine Zusammenführung in beliebiger Reihenfolge erreichen kann, ohne die negativen Eigenschaften der TP2-Algorithmen aufzuweisen. Dazu nutzt der Algorithmus spezifische Eigenschaften des Spawn & Merge Programmiermodells.

5.3 Deterministischer Merge in beliebiger Reihenfolge

Um einen deterministischen `Merge` in beliebiger Reihenfolge (und somit auch implizit in einer FCFS-Reihenfolge) zu ermöglichen, werden im Folgenden zuerst die Spawn & Merge spezifischen Eigenschaften untersucht, die einen solchen Algorithmus ermöglichen können. Anschließend wird beschrieben, wie ein bestehender TP1-Algorithmus erweitert werden kann, um einen `Merge` in beliebiger Reihenfolge zu ermöglichen, und was diese Veränderungen für die Nutzbarkeit mit den Synchronisationsprimitiven `Sync` und `SpawnSibling` bedeuten. Abschließend wird der erweiterte Algorithmus klassifiziert und in ein Verhältnis zu den Transformationseigenschaften TP1 und TP2 gesetzt. Dabei ist zu beachten, dass der hier vorgestellte Algorithmus eine Optimierung für das Spawn & Merge Programmiermodell darstellt und für die generelle Funktionsweise nicht erforderlich ist.

5.3.1 Herausforderungen bei beliebiger Merge-Reihenfolge

Soll eine Zusammenführung trotz beliebiger Reihenfolge deterministisch geschehen, so ergeben sich insbesondere zwei Herausforderungen. Die folgenden Betrachtungen gelten dabei unter der Annahme, dass dem Algorithmus eine deterministische Reihenfolge bekannt ist, nach der sich das Ergebnis der beliebigen Reihenfolge richten soll (im Folgenden

logische `Merge`*-Reihenfolge* genannt). Des Weiteren wird nur das Zusammenführen mit direkten Child-Tasks betrachtet, da die Operationen von deren Child-Tasks zum Zeitpunkt der Zusammenführung bereits in den Operationen des Child-Tasks beinhaltet sind.

Zum einen darf der Algorithmus für die beliebige `Merge`-Reihenfolge nur diejenigen Operationen für eine Transformation betrachten, die bei Einhaltung der logischen `Merge`-Reihenfolge auch für die Transformation zu berücksichtigen wären. Dabei ist es kein Problem, falls noch nicht alle Operationen, die sich auf eine Position *vor* der aktuell zu transformierenden Operation beziehen, vorliegen. Das liegt daran, dass diese fehlenden Operationen auch bei einer verspäteten Anwendung die nachfolgenden Operationen genauso verschieben, wie es bei einer früheren Anwendung der Fall gewesen wäre. Zum anderen muss sichergestellt werden, dass Konflikte immer auf die gleiche Weise aufgelöst werden. Dabei ist auch zu beachten, dass sich bei einer beliebigen `Merge`-Reihenfolge keine neuen Konflikte zwischen Operationen ergeben dürfen, die bei einer festen Reihenfolge nicht aufgetreten wären und dass sich die Art der Konfliktauflösung (d.h. die angewendete Transformation) bei bestehenden Konflikten nicht verändern darf[5].

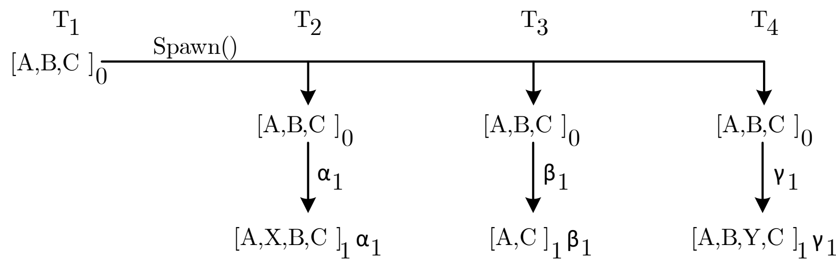


Abbildung 5.6: Operationen die bei beliebiger `Merge`-Reihenfolge zu Problemen führen (1).

Ein Beispiel, das zu einer veränderten Konfliktauflösung durch eine Veränderung der `Merge`-Reihenfolge führen kann, ist in Abbildung 5.6 zu sehen. Werden die Tasks $T_2 - T_4$ nacheinander gemerged, so ergeben sich beim Parent-Task nacheinander die Listen

$$[A, B, C]_0 \rightarrow [A, X, B, C]_1 \rightarrow [A, X, C]_2 \rightarrow [A, X, Y, C]_3. \tag{5.12}$$

Die in diesem Beispiel (Abbildung 5.6 und Formel 5.12) durchgeführten Operationen $\alpha_1 = insert(1, X)$ und $\gamma_3'' = insert(2, Y)$ stehen dabei nicht in Konflikt[6]. Betrachtet man nun als beliebige `Merge`-Reihenfolge die Reihenfolge T_3, T_4, T_2 (die sich z.B. als FCFS-Reihenfol-

[5]Im Folgenden schließt die Bezeichnung „veränderte Konfliktauflösung“ für eine bessere Lesbarkeit auch neu auftretende und wegfallende Konflikte mit ein.

[6]Die Operation γ_1 wird zuerst gegen α_1 transformiert, wodurch sich bei Anwendung der *TRANSFORM*-Funktion der Index (bzw. die Position) zu 3 ändert. Anschließend wird die Operation γ_1' gegen die Operation β_2' transformiert, wodurch sich der Index (bzw. die Position) zurück zu 2 ändert. Die Version ändert sich bei der Anwendung der Operation auf 3.

ge ergeben könnte), so wird bei der ersten Zusammenführung durch β_1 das B aus der Liste entfernt. Anschließend wird γ_1 gegen β_1 transformiert, sodass sich $\gamma'_1 = insert(1, Y)$ ergibt (die Liste verändert sich bei der Anwendung zu $[A, Y, C]_2$ und die Version der Operation wird auf 2 gesetzt (γ'_2)). Versucht der Parent-Task nun abschließend α_1 mit den bereits angewendeten Operationen zusammenzuführen, dann ergibt sich durch die Transformation gegen β_1 die Operation $\alpha'_1 = insert(1, X)$, da β_1 aufgrund der Position keinen Einfluss auf α_1 hat. Anschließend stehen jedoch α'_1 und γ'_2 in Konflikt, die beide eine `Insert`-Operation auf Position 1 anwenden wollen. Diese Art von neu auftretenden Konflikten und veränderten Konfliktauflösungen, die sich durch eine geänderte `Merge`-Reihenfolge ergeben können, müssen zur Sicherstellung der Korrektheit eines OT-Algorithmus für beliebige `Merge`-Reihenfolgen verhindert werden.

5.3.2 Analyse Spawn & Merge spezifischer Eigenschaften

Um die im vorangegangenen Abschnitt beschriebenen Probleme bei einer beliebigen `Merge`-Reihenfolge mit einem TP1-Algorithmus zu lösen, können spezifische Eigenschaften von auf Spawn & Merge basierenden Anwendungen genutzt werden: Die bekannte logische `Merge`-Reihenfolge und die Zusammenführung von Tasks exklusiv mit ihrem Parent-Task. Dieses anwendungsspezifische Wissen erlaubt die Erweiterung von TP1-Algorithmen, sodass diese eine Zusammenführung gespawnter Datenstrukturen in beliebiger Reihenfolge ermöglichen, während die Einführung speicherintensiver Zustandsvektoren vermieden werden kann. Dabei ist zu beachten, dass entsprechende OT-Systeme somit nur in Kontexten (hier das Spawn & Merge Framework) funktionieren können, in denen eine Anwendung diese Informationen bereitstellen kann.

Die *logische `Merge`-Reihenfolge* beschreibt die Reihenfolge, in der Child-Tasks bei der Verwendung normaler TP1-OT-Algorithmen zusammengeführt werden müssten, um ein deterministisches Ergebnis zu erreichen. Sie ergibt sich bei jeder Ausführung der Anwendung deterministisch aus dem im Programmcode vorliegenden Aufruf der deterministischen `Merge`-Primitive und entspricht somit entweder der Spawn-Reihenfolge[7] der Child-Tasks (`MergeAll`) oder der Reihenfolge der als Parameter übergebenen Task-Handles (`MergeAllByHandle`).

Weitere Eigenschaften von Spawn & Merge sind die baumförmige Task-Hierarchie, und die sich aus den Datenstruktur-Historien Φ ergebenden baumförmigen Datenstruktur-Hierarchien. Bei Spawn & Merge werden Tasks *ausschließlich* mit ihrem zugehörigen Parent-Task zusammengeführt. Im Gegensatz zu TP2-OT-Algorithmen müssen OT-Algorithmen im Kontext von Spawn & Merge somit keine beliebigen Merge-Graphen erlauben[8].

[7] Die Spawn-Reihenfolge ist ebenfalls durch den Programmcode deterministisch vorgegeben, inkl. der durch `SpawnSibling` gestarteten und in die Spawn-Reihenfolge eingeordneten Child-Tasks (siehe Einordnung in Kapitel 4.2.10).

[8] D.h. es muss nicht jeder Task in der Lage sein seine Datenstrukturen mit einem beliebigen anderen Task

Dies erlaubt es OT-Algorithmen in Spawn & Merge auf die Nutzung von Zustandsvektoren zu verzichten und stattdessen auf skalare Versionszähler zurückzugreifen, nie nur zwischen einem Task und seinen Child-Tasks gelten, beziehungsweise zwischen deren Datenstrukturkopien.

5.3.3 OT-Algorithmus für beliebige Merge-Reihenfolge

In diesem Kapitel wird beschrieben, wie die in Kapitel 5.3.2 genannten Spawn & Merge spezifischen Eigenschaften genutzt werden können, um einen TP1-OT-Algorithmus so zu erweitern, dass dieser eine deterministische Zusammenführung in beliebiger Reihenfolge erlaubt. Der resultierende Algorithmus ist dabei eine Spezialform, die nur unter der Annahme genutzt werden kann, dass die Anwendung eine deterministische logische Reihenfolge für die Zusammenführung bereitstellen kann, sowie dass Datenstrukturen nur mit ihrem direkten Elternknoten zusammengeführt werden.

Zur Veranschaulichung wird hier das in Kapitel 5.1.2 vorgestellte TP1-OT-System für eine Liste mit den Operationen `insert` und `delete` beispielhaft erweitert. Ein generalisierter Algorithmus ist nicht möglich, da die Funktionalität des Algorithmus abhängig von der Datenstruktur und den anwendbaren Operationen ist. Die Anpassung eines TP1-OT-Algorithmus für Zusammenführungen in beliebiger Reihenfolge muss dementsprechend für jeden TP1-OT-Algorithmus nach dem im Folgenden beschriebenen Schema einmal durchgeführt werden.

Ansatz des Algorithmus

Um die in Kapitel 5.3.1 beschriebenen Herausforderungen zu lösen, die sich bei einer Zusammenführung in beliebiger Reihenfolge ergeben, nutzt der hier beschriebene Algorithmus drei Informationsquellen: Die logische `Merge`-Reihenfolge, Tombstones und Merge-Epochen.

Ein kritisches Element der deterministischen Zusammenführung mit Operational Transformation ist die Auflösung von Konflikten, die immer gleich durchgeführt werden muss. Dazu kann die logische `Merge`-Reihenfolge in Betracht gezogen werden, um zu prüfen, wie ein Konflikt zwischen zwei Operationen bei einer Zusammenführung in der logischen Reihenfolge aufgelöst worden wäre. Hierbei gilt es für jede Transformation einer Child-Operation Op_C gegen eine Parent-Operation[9] Op_P zu prüfen, wie und ob die Transformation bei einer Zusammenführung der Datenstrukturen in der *logischen `Merge`-Reihenfolge* durchgeführt worden wäre. Somit muss die angepasste Kontrollfunktion des OT-Systems anhand der logischen Reihenfolge für zwei Operationen (Op_C und Op_P) in einem ersten

zusammenführen zu können.

[9] Die Parent-Operationen beinhalten dabei sowohl die Operationen, die der Parent selbst auf der Datenstruktur durchgeführt hat, sowie die Operationen der Child-Tasks, die bereits zusammengeführt wurden.

Schritt entscheiden können, ob in der logischen Reihenfolge Op_C von Op_P beeinflusst worden wäre, oder ob Op_P noch gar nicht vorliegen würde. Dies wäre genau dann der Fall, wenn Op_P eine Operation eines bereits zusammengeführten Child-Tasks ist, der nach der logischen Reihenfolge allerdings erst nach dem aktuellen Child-Task zusammengeführt werden sollte. Um dies zu ermöglichen, muss für jede Operation ersichtlich sein, wer diese Operation durchgeführt (erstellt) hat. Eine entsprechende Erweiterung der Operationen wird in Kapitel 5.3.3 beschrieben. In einem zweiten Schritt muss die Kontrollfunktion sicherstellen, dass die Transformation (und dabei insbesondere die Konfliktauflösung) auch bei einer beliebigen Reihenfolge gleich durchgeführt wird.

Um zu verhindern, dass es durch die beliebige Reihenfolge für die Zusammenführung zu veränderten Konfliktauflösungen zwischen Operationen kommt (siehe Kapitel 5.3.1), werden *Tombstones* (Grabsteine) für gelöschte Elemente eingeführt. Tombstones wurden bereits in einigen TP2-OT-Algorithmen verwendet, um zu verhindern, dass zwei nicht konfligierende Operationen durch eine dritte Operation in einen Konflikt geraten können [77]. Abbildung 5.7 zeigt hier, dass sich das beispielhafte Problem aus Abbildung 5.6 durch die Einführung von Tombstones lösen lässt. Das Anwenden der Operation β_1 markiert das Element B als gelöscht, entfernt dieses aber nicht aus der Liste. Die Operationen α_1 und γ_1 können nun eindeutig jeweils vor oder hinter dem B angewendet werden, ohne dass sie durch die verfrühte Anwendung von β_1 in einen Konflikt geraten können.

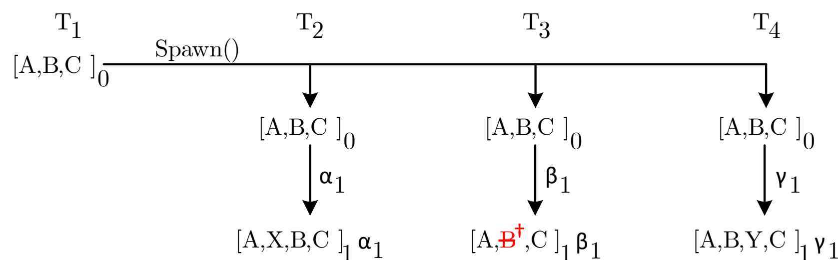


Abbildung 5.7: Operationen die bei beliebiger `Merge`-Reihenfolge zu Problemen führen (2).

Die dritte Herausforderung betrifft die Unterscheidbarkeit, ob zwei gegeneinander zu transformierende Operationen Op_1 und Op_2 im Rahmen desselben `Merge`-Aufrufes an den Parent übergeben wurden, oder ob eine der Operationen das Ergebnis eines früheren `Merge`-Aufrufes ist. Dies ist eine Besonderheit, die nur dann eintreten kann, wenn Child-Tasks die `Sync`- oder die `SpawnSibling`-Primitive verwenden. Entstammen die Operationen Op_1 und Op_2 nicht demselben `Merge`-Aufruf (d.h. derselben *Merge-Epoche*), so muss dies bei der Transformation beachtet werden (d.h. die Konfliktauflösung muss so durchgeführt werden, dass die Operation des Parent-Tasks priorisiert wird, unabhängig von Anpassungen, die sich durch die beliebige Reihenfolge ergeben könnten). Eine genauere Beschreibung des Problems anhand eines Beispiels folgt in Kapitel 5.3.3.

Erweiterung der Operationen

Die Anforderungen, dass für jede Operation erkennbar sein muss welcher Task diese erstellt hat und ob ein Element der Liste noch existiert oder bereits gelöscht wurde, machen eine Erweiterung der Operationen und Listen-Elemente notwendig.

Die Operationen müssen um ein Feld (hier *CreatingEntity*) erweitert werden, das den Task, der die Operation erstellt hat, erkennbar macht. Dazu wird die Definition der Operationen (siehe Kapitel 5.1.1), wie in Formel 5.13 zu sehen, erweitert.

$$Operation := (OperationCode, Element, Position, Version, CreatingEntity) \tag{5.13}$$

Das Feld *CreatingEntity* hat nur eine lokale Bedeutung. Wird eine neue Operation erstellt (d.h. ein Task modifiziert eine zusammenführbare Datenstruktur), so wird die *CreatingEntity* auf den Wert -1 gesetzt. Dieser Wert bedeutet, dass die Operation entweder vom aktuellen Task selbst durchgeführt, oder von seinem Parent-Task empfangen wurde. Für die Durchführung der OT sind diese beiden Fälle äquivalent, da in beiden Fällen die entsprechende Operation eine höhere Priorität besitzt als eingehende Operationen von Child-Tasks.

Werden Operationen im Rahmen der Fertigstellung oder eines `Sync`-Aufrufs eines Tasks T_1 an den Parent-Task T_P übertragen, so muss für alle übertragenen Operationen die *CreatingEntity* auf die Task-ID des Tasks T_1 geändert werden, um die Einhaltung der lokalen Bedeutung für den Parent-Task T_P sicherzustellen[10]. Bei der Übertragung der `Merge`-Ergebnisse zurück zum Parent-Task (im Kontext eines `Sync`-Aufrufes) müssen die *CreatingEntity*-Felder für alle Operationen, die der Child-Task noch nicht kannte, auf -1 gesetzt werden. So wird erreicht, dass für einen beliebigen Task die eigenen Operationen und die Operationen, die vom Parent-Task empfangen wurden ($CreatingEntity == -1$) von den Operationen, die von Child-Tasks empfangen wurden ($CreatingEntity == Child-Task-ID$), unterschieden werden können.

Eine besondere Betrachtung von Operationen, die durch mit `SpawnSibling` gestartete Child-Tasks erstellt wurden, ist nicht notwendig, obwohl sich durch die beliebige Reihenfolge der Zusammenführung eine nichtdeterministische Zuweisung der *CreatingEntity* ergeben kann. Angenommen aus einer Anwendung ergibt sich die in Abbildung 5.8 dargestellte Task-Hierarchie. Die Tasks T_1, T_2 und T_3 wurden vom Parent-Task T_P mithilfe der `Spawn`-Primitive gestartet. Task T_4 wurde des Weiteren von Task T_2 unter Verwendung von `SpawnSibling` erstellt und in der Liste der Child-Tasks von T_P direkt hinter T_2 eingeordnet (siehe Kapitel 4.2.10).

[10] Dabei werden die Operationen, die Child-Tasks von Task T_1 erzeugt haben, genauso behandelt wie Operationen, die von Task T_1 selbst erzeugt wurden.

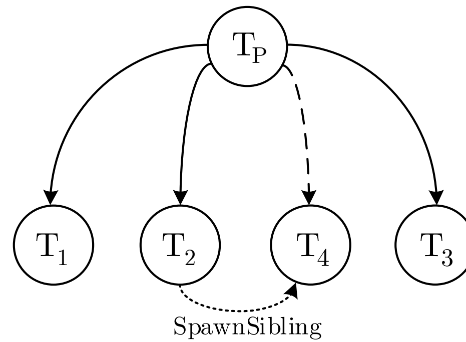


Abbildung 5.8: Einordnung des von Task T_2 per *SpawnSibling* gestarteten Tasks T_4.

Falls T_2 vor dem `SpawnSibling`-Aufruf bereits Operationen durchgeführt hat, so liegen diese nun sowohl in T_2 als auch in T_4 vor. Abhängig davon, welcher der beiden Child-Tasks zuerst von T_P zusammengeführt wird, bekommen diese Operationen als *CreatingEntity* entweder den Wert 2 oder den Wert 4 zugewiesen. Diese Zuweisung ist nichtdeterministisch, dennoch hat sie keinen Einfluss auf den Determinismus der Zusammenführung. Die *CreatingEntity* ist im Rahmen der Zusammenführung in beliebiger Reihenfolge notwendig, um zu unterscheiden, ob ein Child-Task T_x in der logischen `Merge`-Reihenfolge *vor* oder *nach* einem anderen Child-Task T_y zusammengeführt würde. Diese Unterscheidbarkeit ist auch in Anbetracht der nichtdeterministischen *CreatingEntity*-Zuweisungen weiterhin gegeben. Das liegt daran, dass mit `SpawnSibling` gestartete Tasks immer direkt hinter dem aufrufenden Task in der Child-Liste des Parent-Tasks eingeordnet werden. Die Formeln 5.14 und 5.15 zeigen, dass wenn für die logische `Merge`-Reihenfolge die Ordnung $T_{n-1} < T_n < T_{n+1}$ gilt, dann gilt auch $T_{n-1} < T_{n,Sibling_X} < T_{n+1}$, wobei $T_{n,Sibling_X}$ mit $1 \leq X \leq m$ einen der m per `SpawnSibling` durch T_n gestarteten Tasks angibt. Die Zusammenführung kann dementsprechend deterministisch durchgeführt werden.

$$[\ldots, T_{n-1},\ T_n,\ T_{n+1},\ \ldots] \tag{5.14}$$

$$[\ldots, T_{n-1},\ T_n,\ \underbrace{T_{n,Sibling_1},\ \ldots,\ T_{n,Sibling_m}}_{\text{Per } \texttt{SpawnSibling} \text{ von } T_n \text{ erstellt}},\ T_{n+1},\ \ldots] \tag{5.15}$$

Zur Vermeidung veränderter Konfliktauflösungen werden Tombstones für gelöschte Elemente eingeführt (siehe Kapitel 5.3.3). Dazu werden die Elemente der zusammenführbaren Datenstruktur um die Angabe des Erstellungszeitpunktes (*Version*) und des Löschzeitpunktes (*Tombstoned*) erweitert (siehe Formel 5.16).

$$Element := (ElementValue, Version, Tombstoned, TombstoneLength) \tag{5.16}$$

Das zusätzliche Feld *TombstoneLength* ermöglicht die Zusammenfassung mehrerer aufeinanderfolgender Tombstones (siehe Formel 5.17) zur Reduzierung der maximalen Anzahl erforderlicher Tombstones auf $m + 1$, wobei m die Anzahl der ungelöschten Listenelemente ist. Wird dabei ein neues Element innerhalb der Spannweite eines Tombstones eingefügt, so wird der Tombstone auf den Bereich vor und hinter dem neuen Element aufgeteilt (siehe Formel 5.18).

$$[A, \underbrace{\cancel{B}^{\dagger}, \cancel{C}^{\dagger}, \cancel{D}^{\dagger}}_{\substack{\text{Ein Tombstone} \\ \text{der Länge 3}}}, E] \tag{5.17}$$

$$[A, \underbrace{\cancel{B}^{\dagger}, \cancel{C}^{\dagger},}_{\substack{\text{Ein Tombstone} \\ \text{der Länge 2}}} X, \underbrace{\cancel{D}^{\dagger},}_{\substack{\text{Ein Tombstone} \\ \text{der Länge 1}}} E] \tag{5.18}$$

Unterscheidung von Merge-Epochen

Spawn & Merge erlaubt durch die `Sync`-Primitive die wiederholte Zusammenführung eines Tasks T_1 mit seinem Parent-Task T_P. Für Operationen, die von Task T_1 erstellt wurden, ist somit nicht direkt ersichtlich, zu welchem Zeitpunkt diese mit dem Parent-Task T_P zusammengeführt wurden. Bei einer Zusammenführung in beliebiger Reihenfolge ist es allerdings notwendig unterscheiden zu können, ob eine Child-Operation während des aktuell laufenden Aufrufs einer `Merge`-Primitive mit dem Parent-Task zusammengeführt wurde, oder ob die Operation bereits vor dem Aufruf der `Merge`-Primitive mit dem Parent-Task zusammengeführt wurde. In letzterem Falle dürfen die Mechanismen für eine Zusammenführung in beliebiger Reihenfolge nicht angestoßen werden, da die Operation nicht aus dem aktuellen `Merge`-Aufruf stammt. Um diese Unterscheidung zu ermöglichen, wird die sogenannte Merge-Epoche eingeführt.

Der Begriff *Merge-Epoche* beschreibt den Zeitraum während der Durchführung eines einzelnen Aufrufs einer `Merge`-Primitive. Innerhalb einer Merge-Epoche können mehrere Child-Tasks mit dem Parent-Task zusammengeführt werden[11]. Jeder neue Aufruf einer `Merge`-Primitive startet hierbei eine neue Merge-Epoche.

Zur Illustration der Problematik sei in Listing 5.2 ein Task gegeben, der zwei Child-Tasks T_1 und T_2 gestartet hat (Zeile 3 und Zeile 4). Die gespawnte Funktion `fct_y` nutzt intern die `Sync`-Primitive, sodass *task*2 auch nach dem Aufruf von `MergeAllByHandle` in Zeile 6 noch nicht beendet wurde. Anschließend werden sowohl *task*1 als auch *task*2 unter Verwendung von `MergeAllByHandle` zusammengeführt.

[11] Im Falle eines `MergeAll(TILL_ALL_FINISHED)`-Aufrufs auch einzelne Child-Tasks mehrfach, falls diese Gebrauch von der `Sync`-Primitive machen.

Listing 5.2 Notwendigkeit der Merge-Epochen.

```
1: [...]
2:
3: TaskHandle task1 = Spawn(fct_x, list);
4: TaskHandle task2 = Spawn(fct_y, list);
5:
6: MergeAllByHandle(task2);
7: MergeAllByHandle(task1, task2);
```

Falls eine Zusammenführung in beliebiger Reihenfolge erlaubt wird, so muss das Ergebnis der Zusammenführung weiterhin deterministisch sein, unabhängig davon, ob die Reihenfolge der Zusammenführung *task*1 gefolgt von *task*2 ist, oder umgekehrt. Das bedeutet insbesondere, dass bei der Transformation der Operationen von *task*1 (falls dieser nach *task*2 mit dem Parent-Task zusammengeführt wird) die Operationen $Op_{task2,1}$ von *task*2 aus dem `MergeAllByHandle`-Aufruf aus Zeile 6 von den Operationen $Op_{task2,2}$ von *task*2 aus dem `MergeAllByHandle`-Aufruf aus Zeile 7 unterschieden werden müssen. Dies ist notwendig, da bei einer Zusammenführung in fester Reihenfolge die Operationen $Op_{task2,1}$ *vor* den Operationen von *task*1 zusammengeführt worden wären, die Operationen $Op_{task2,2}$ hingegen erst *nach* den Operationen von *task*1. Die Transformationen und Konfliktauflösungen müssen sich dementsprechend in Abhängigkeit vom Ursprungszeitpunkt der Operationen unterschiedlich verhalten[12], um unter diesen Umständen ein deterministisches Ergebnis zu erreichen. Ohne diese Unterscheidung würden sowohl die Operationen $Op_{task2,1}$ als auch die Operationen $Op_{task2,2}$ bei der Zusammenführung von *task*1 so behandelt werden, als hätte *task*1 sie nach der logischen `Merge`-Reihenfolge noch nicht gesehen, obwohl die Operationen $Op_{task2,1}$ einem früheren `Merge`-Aufruf entstammen und somit *task*1 bereits bekannt sind. Die Einführung der Merge-Epochen dient der Unterscheidbarkeit dieser Fälle.

OT-System für Listen zur Zusammenführung in beliebiger Reihenfolge

Wie andere OT-Systeme besteht auch das hier beschriebene System zur Zusammenführung in beliebiger Reihenfolge aus zwei Komponenten, der Transformationsfunktion und der Kontrollfunktion. Der Ansatz des Systems besteht darin, mithilfe der logischen `Merge`-Reihenfolge die Transformationen einer Zusammenführung in deterministischer Reihenfolge reproduzieren zu können (siehe Kapitel 5.3.3).

Um dies zu ermöglichen, erhält die Kontrollfunktion neben den zwei Datenstrukturen die zusammengeführt werden sollen (die Datenstrukturkopie des Parent-Tasks sowie die Datenstrukturkopie des Child-Tasks, der gerade zusammengeführt wird) zusätzlich die

[12]Die genauen Unterschiede werden in Kapitel 5.3.3 beschrieben.

logische `Merge`-Reihenfolge aller Child-Tasks[13], die mit dem aktuellen `Merge`-Aufruf zusammengeführt werden sollen, die Datenstruktur-Historie Φ und die Angabe der aktuellen Merge-Epoche.

Die in Kapitel 5.1.2 vorgestellte Transformationsfunktion $TRANSFORM(op_1, op_2)$ wird hier verändert, um den Umgang mit Tombstones zu ermöglichen. Die abgewandelte Transformationsfunktion $FCFS_TRANSFORM(op_1, op_2)$ ist in Listing 5.3 beschrieben[14]. Das Präfix „FCFS" steht hierbei für *First-Come-First-Serve (FCFS)*. Verändert wurde hier die Transformationen einer beliebigen Operation Op_X gegen eine `delete`-Operation, da durch die Einführung von Tombstones die Operation Op_X nicht im Rahmen der Transformation verschoben werden muss. Daher entfällt in den Zeilen 8, 12, 17 und 18 die Anpassung der Position um -1. Eine weitere Anpassung der Transformationsfunktion ist nicht notwendig, da die weitere notwendige Funktionalität innerhalb der Kontrollfunktion realisiert wird.

Listing 5.3 Angepasste Transformationsfunktion *FCFS_TRANSFORM*.

```
 1:  function FCFS_TRANSFORM(insert(i1, k1), insert(i2, k2)):
 2:    if k1 <  k2: return(insert(i1, k1)    , insert(i2, k2 + 1))
 3:    if k1 >  k2: return(insert(i1, k1 + 1), insert(i2, k2)    )
 4:    if k1 == k2: return(insert(i1, k1)    , insert(i2, k2 + 1))
 5:
 6:  function FCFS_TRANSFORM(insert(i, k1), delete(k2)):
 7:    if k1 <  k2: return(insert(i, k1), delete(k2 + 1))
 8:    if k1 >  k2: return(insert(i, k1), delete(k2)    )
 9:    if k1 == k2: return(insert(i, k1), delete(k2 + 1))
10:
11:  function FCFS_TRANSFORM(delete(k1), insert(i, k2)):
12:    if k1 <  k2: return(delete(k1)    , insert(i, k2))
13:    if k1 >  k2: return(delete(k1 + 1), insert(i, k2))
14:    if k1 == k2: return(delete(k1 + 1), insert(i, k2))
15:
16:  function FCFS_TRANSFORM(delete(k1), delete(k2)):
17:    if k1 <  k2: return(delete(k1), delete(k2))
18:    if k1 >  k2: return(delete(k1), delete(k2))
19:    if k1 == k2: return(no-op     , no-op     )
```

Die Kontrollfunktion steuert die Zusammenführung eines Child-Tasks mit dem Parent-Task. Dabei werden alle neuen Operationen durchlaufen, die vom Parent-Task und dem gerade zusammengeführten Child-Task erzeugt wurden und dem jeweils anderen

[13]Die übergebene logische Reihenfolge der Child-Tasks muss hier bei jeder Zusammenführung eines weiteren Child-Tasks bei `MergeAll` aktualisiert werden, da sich die Liste der Child-Tasks des Parent-Tasks durch `SpawnSibling` verändert haben kann.

[14]Die applikationsabhängige Auflösung eines Konfliktes zweier `insert`-Operationen wird hier in Zeile 4 realisiert, indem die Operation, die als erster Parameter übergeben wurde, positionstechnisch *vor* der anderen Operation eingeordnet wird.

Task noch nicht bekannt sind. Die Kontrollfunktion entscheidet dabei für jedes Paar von Operationen, ob diese normal gegeneinander transformiert werden müssen, oder ob eine besondere Behandlung notwendig ist. Dies kann entweder ein Überspringen der Transformation sein (falls Parent-Operationen vom Child-Task ignoriert werden müssen[15]) oder das Umdrehen der Transformationsparameter. Ein Umdrehen der Parameter ist immer dann erforderlich, wenn beide Operationen derselben Merge-Epoche entstammen und die Parent-Operation dem Child-Task bei Einhaltung der logischen `Merge`-Reihenfolge noch unbekannt wäre. Das Umdrehen der Parameter sorgt hierbei dafür, dass die Transformation in der Weise durchgeführt wird, wie es bei einer Zusammenführung in deterministischer Reihenfolge der Fall gewesen wäre.

In Listing 5.4 ist die Kontrollfunktion für eine Zusammenführung in beliebiger Reihenfolge als Pseudocode dargestellt[16]. Die Kontrollfunktion *MERGE*() führt dabei zwei Listen (die des Parent-Tasks und die eines Child-Tasks) zusammen und kann für alle Child-Tasks in beliebiger Reihenfolge aufgerufen werden (insbesondere der FCFS-Reihenfolge der Fertigstellung der Child-Tasks). Als Parameter werden der Funktion, neben den beiden Listen *parentList* und *childList*, die logische `Merge`-Reihenfolge *logicalOrder*, die Datenstruktur-Historie *structureHistory* und die aktuelle Merge-Epoche *mergeEpoch* übergeben. In Zeile 15 werden die Task-ID des aktuell zusammengeführten Child-Tasks (*childTaskId*) und die Operationen *ignoreOps*, die durch `SpawnSibling` schon bekannt sind und bei der Zusammenführung ignoriert werden müssen, aus *structureHistory* inferiert. Anschließend werden in Zeile 16 und 17 die Operationen, die dem jeweils anderen Task noch unbekannt sind, für die Durchführung der Transformationen in die Variablen *parentOpsCopy* und *childOpsCopy* kopiert. Welche der Operationen noch unbekannt sind, ist aus der Datenstruktur-Historie *structureHistory* erkennbar. Die Kopie der Child-Operationen *childOpsCopy* entspricht den Operationen, die nach der Transformation an die Liste der Operationen des Parent-Tasks angehängt wird. Damit der Parent-Task dann weiterhin erkennen kann, welcher Child-Task die Operationen erstellt hat, werden in Zeile 18 die *CreatingEntity*-Felder aller Operationen in *childOpsCopy* auf die Task-ID des Child-Tasks gesetzt. Die Kopie der Parent-Operationen *parentOpsCopy* entspricht den Operationen, die im Falle eines `Sync`-Aufrufes nach Durchführung der Transformation an den Child-Task zurückgesendet werden.

Der Kern der Kontrollfunktion ist das Durchlaufen aller Child-Operationen (Zeile 23) und für jede dieser Child-Operationen wiederum aller Parent-Operationen (Zeile 24). Dabei werden alle Parent-Operationen, die dem Child-Task bereits bekannt sind, übersprungen, indem die *ignoreOps* überprüft werden (Zeile 25). Für jede Kombination aus

[15]Dies kann der Fall sein, wenn zwei Child-Tasks dieselben Operationen beinhalten, weil einer der beiden Child-Tasks mit `SpawnSibling` gestartet wurde. Gegen die bereits bekannten Operationen darf dementsprechend nicht noch einmal transformiert werden, da die Effekte der Operationen bereits miteinbezogen wurden.

[16]Zugunsten der Lesbarkeit wurde dabei auf die Darstellung der Anpassung von Versionsnummern einzelner Operationen verzichtet.

Listing 5.4 Kontrollfunktion für eine Zusammenführung in beliebiger Reihenfolge.

```
 1: function MERGE(parentList, childList, logicalOrder, structureHistory, mergeEpoch)
 2:     var
 3:         childTaskId                ▷ Task-ID of currently merged Child-Task
 4:         ignoreOps                  ▷ Operations already known to childList
 5:         childOpsCopy               ▷ Copy of operations performed on childList
 6:         parentOpsCopy              ▷ Copy of operations performed on parentList
 7:
 8:         cIdx                       ▷ childOpsCopy index
 9:         pIdx                       ▷ parentOpsCopy index
10:         sameEpoch                  ▷ If TRUE, both operations are in same epoch
11:         ChildBeforeParentOp        ▷ If TRUE, parent operation should not be visible to child
12:         reverseTransformationOrder ▷ If TRUE, next transformation will be reversed
13:     end var
14:
15:     childTaskId, ignoreOps ← ExtractFromStructureHistory(structureHistory)
16:     parentOpsCopy ← CopyUnknownOps(parentList, structureHistory)
17:     childOpsCopy ← CopyUnknownOps(childList, structureHistory)
18:     childOpsCopy ← SetCreatingEntity(childOpsCopy, childTaskId)
19:
20:     cIdx ← 1                       ▷ Start index of childOpsCopy
21:     pIdx ← 1                       ▷ Start index of parentOpsCopy
22:
23:     while cIdx ≤ |childOpsCopy| do
24:         while pIdx ≤ |parentOpsCopy| do
25:             if parentOpsCopy[pIdx] ∉ ignoreOps then
26:                 sameEpoch ← IsInEpoch(parentOpsCopy[pIdx], mergeEpoch)
27:                 ChildBeforeParentOp ← IsChildOpBeforeParentOp(childOpsCopy[cIdx],
                                                                  parentOpsCopy[pIdx],
                                                                  logicalOrder)
28:
29:                 reverseTransformationOrder ← sameEpoch and ChildBeforeParentOp
30:
31:                 if reverseTransformationOrder = TRUE then
32:                     FCFS_TRANSFORM(childOpsCopy[cIdx], parentOpsCopy[pIdx])
33:                 else
34:                     FCFS_TRANSFORM(parentOpsCopy[pIdx], childOpsCopy[cIdx])
35:                 end if
36:             end if
37:             pIdx ← pIdx + 1        ▷ Step to next copied parent operation
38:         end while
39:         cIdx ← cIdx + 1            ▷ Step to next copied child operation
40:     end while
41:
42:     parentOpsCopy ← SetCreatingEntity(parentOpsCopy, −1)  ▷ Prepare Sync results
43: end function
```

Parent-Operation und Child-Operation muss geprüft werden, ob die Parameterreihenfolge für den Aufruf der Transformationsfunktion getauscht werden muss, weil die Parent-Operation dem Child-Task noch nicht bekannt sein dürfte und bei Einhaltung der logischen `Merge`-Reihenfolge die Transformation in umgekehrter Reihenfolge stattgefunden hätte. Die Transformation muss genau dann getauscht werden, wenn beide Operationen derselben Merge-Epoche entstammen und wenn der Child-Task in der logischen `Merge`-Reihenfolge *vor* dem Task liegt, der die Parent-Operation erstellt hat (*CreatingEntity* der Parent-Operation). Daher wird in Zeile 26 zunächst geprüft, ob die Parent-Operation in der aktuellen Merge-Epoche oder zu einem früheren Zeitpunkt erstellt wurde. Anschließend wird in Zeile 27 geprüft, ob die Parent-Operation von einem Task durchgeführt wurde, dessen Operationen für den Child-Task nicht betrachtet werden dürfen, da der erstellende Task in der logischen `Merge`-Reihenfolge noch nicht zusammengeführt worden wäre. In diesem Falle ist die Position der *CreatingEntity* der Parent-Operation in *logicalOrder* größer als die Position des Child-Tasks in *logicalOrder*. Die Bedingung für das Tauschen der Transformationsparameter kann nun durch eine logische *Und*-Verknüpfung der vorherigen Erkenntnisse geprüft werden (Zeile 29). Trifft die Bedingung zu, so werden die Parameter in Zeile 32 in umgekehrter Reihenfolge an die Transformationsfunktion *FCFS_TRANSFORM* übergeben. Trifft sie nicht zu, so wird die Transformation mit der normalen Parameterreihenfolge durchgeführt (Zeile 34).

Wurden alle Transformationen durchgeführt, dann werden abschließend in Zeile 42 alle *CreatingEntity*-Felder der Operationen, die im Falle eines `Sync`-Aufrufes an den Child-Task zurückgesendet werden (*parentOpsCopy*), auf -1 gesetzt. Dies zeigt an, dass die Operationen vom Parent-Task stammen. Die Rückübertragung der Operationen im Rahmen eines `Sync`-Ergebnisses bleibt unverändert. Dies führt bei der Verwendung des Sync-Modus `RETURN_AFTER_MERGE_CYCLE_COMPLETED` zu keinen Problemen, da alle Operationen gleichermaßen vom Parent-Task kommen und die Reihenfolge der Operationen auf den Child-Task keinen Einfluss haben. Eine Betrachtung der Probleme, die sich bei einer Zusammenführung in beliebiger Reihenfolge für den Sync-Modus `RETURN_DIRECTLY` ergeben, folgt in Kapitel 5.3.3.

Abbildung 5.9 zeigt ein Beispiel, in dem die verschiedenen Herausforderungen für eine Zusammenführung in beliebiger Reihenfolge vereint sind. Ein Task T_1 startet drei Child-Tasks $T_2 - T_4$ mithilfe von `Spawn`, die jeweils eine Operation auf eine geteilte Liste anwenden. Der Child-Task T_4 startet anschließend wiederum einen Sibling-Task T_5 mithilfe von `SpawnSibling`. Der Parent-Task T_1 ruft zu einem späteren Zeitpunkt `MergeAll` auf. Daraus ergibt sich die logische `Merge`-Reihenfolge $[T_2, T_3, T_4, T_5]$ aus der Spawn-Reihenfolge und der Einordnung des Sibling-Tasks hinter T_4. Werden die Tasks in dieser Reihenfolge zusammengeführt[17], so ergibt sich nach dem `Merge`-Aufruf die Liste

[17] Für eine ausführliche Auflistung der durchgeführten Transformationen der Operationen siehe Kapitel

$[Z, A, X, B^{\dagger}, Y, C]_4\ \alpha_1\ \beta_2'\ \gamma_3''\ \delta_4'''$.

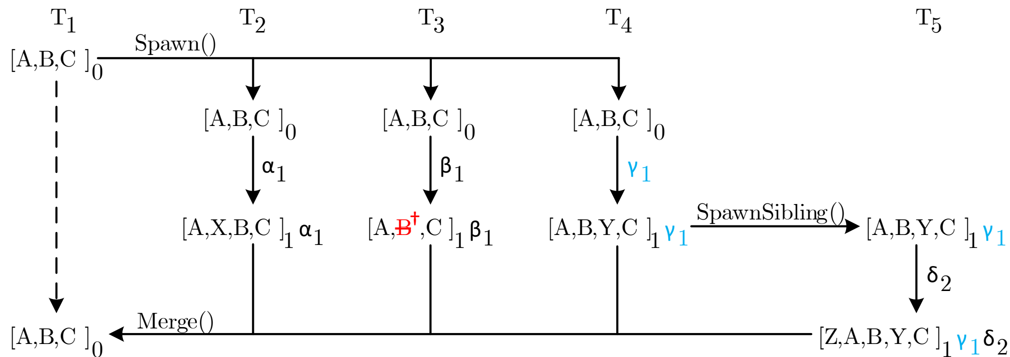


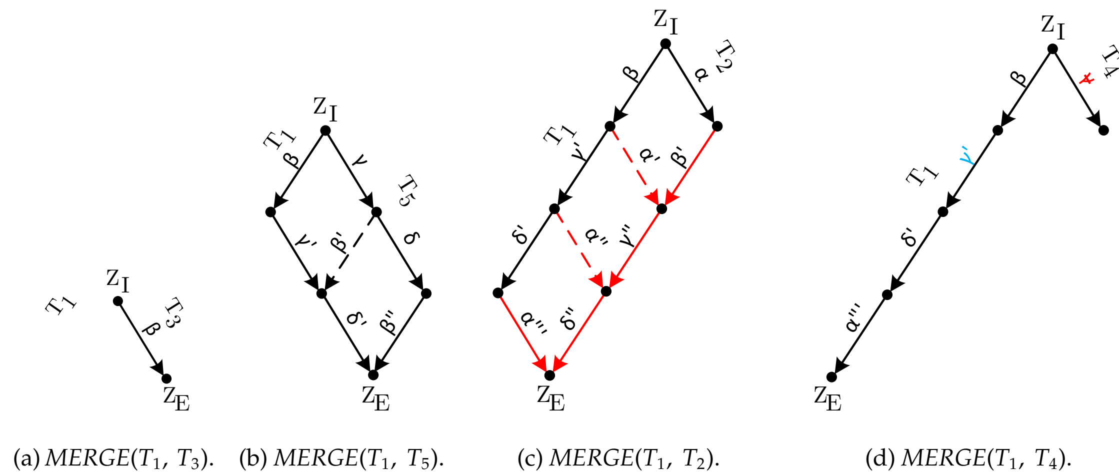
Abbildung 5.9: Beispielszenario für die Anwendung der Kontrollfunktion.

Die erste Herausforderung bei der Zusammenführung in beliebiger Reihenfolge ist in diesem Beispiel das Löschen des Elementes B durch Task T_3, da hier die Operationen α_1 (von Task T_2) und γ_1 (von Task T_4) in einen Konflikt miteinander geraten könnten, der vorher nicht aufgetreten ist. Dieser Punkt wird allerdings bereits durch die Einführung von Tombstones gelöst, die eine veränderte Konfliktauflösung (und somit auch neu auftretende Konflikte) verhindern. Die zweite Herausforderung ist, dass die Operationen eines früh zusammengeführten Tasks die Operationen eines anderen später zusammengeführten Tasks fälschlicherweise beeinflussen könnten. Das tritt genau dann auf, wenn ein Task (hier z.B. T_4), der in der logischen `Merge`-Reihenfolge erst zu einem späteren Zeitpunkt zusammengeführt werden sollte, als ein anderer Tasks (hier z.B. T_2), früher als T_2 zusammengeführt wird. Dieser Umstand muss von der Kontrollfunktion behandelt werden. Als dritte Herausforderung bleibt die Anforderung weiterhin bestehen (siehe Kapitel 5.2.2), dass durch `SpawnSibling` mehrfach vorliegende Operationen nicht mehrfach angewendet werden dürfen. Im Beispiel muss die Kontrollfunktion dementsprechend erkennen, dass die Operation γ_1 sowohl in Task T_4 als auch in Task T_5 vorliegt (durch `SpawnSibling`), und das zweite Auftreten der Operation ignorieren.

Die Abbildung 5.10 zeigt die Zusammenführungsgraphen für den Parent-Task T_1 mit seinen Child-Tasks $T_2 - T_5$. Die hier betrachtete Reihenfolge für die Zusammenführung ist $[T_3, T_5, T_2, T_4]$, in der alle drei Herausforderungen auftreten. Das Löschen des Elementes B wird zuerst übernommen, sodass die Operationen α_1 und γ_1 in einen neuen Konflikt geraten könnten. Der Task T_2 wird erst später als die Tasks T_3 und T_5 zusammengeführt, sodass hier die Reihenfolge der Parameter bei der Transformation umgedreht werden muss. Der Task T_4 muss des Weiteren erkennen, dass die Operation γ_1 bereits von T_5 an den Parent übertragen wurde. Für jede Zusammenführung eines Child-Tasks mit dem

C.1 im Anhang.

Parent-Task ist in Abbildung 5.10 ein eigener Zusammenführungsgraph gegeben.



(a) $MERGE(T_1,\ T_3)$. (b) $MERGE(T_1,\ T_5)$. (c) $MERGE(T_1,\ T_2)$. (d) $MERGE(T_1,\ T_4)$.

Abbildung 5.10: Zusammenführungsgraphen für T_1 mit $T_2 - T_5$.

Dadurch, dass der Parent-Task T_1 keine eigenen Operationen durchführt, können die Operationen des zuerst zusammengeführten Child-Tasks T_3 ohne eine Transformation in T_1 übernommen werden (Abbildung 5.10a). Wird nun der Task T_5 mit dem Parent-Task zusammengeführt, so werden die Operationen von T_5 (hier γ und δ) mit normaler Parameterreihenfolge gegen die Operation β, die nun im Parent-Task vorliegt, transformiert. Die transformierten Operationen γ' und δ' werden anschließend an die Liste der Operationen des Parent-Tasks angehängt (Abbildung 5.10b). Bei der Zusammenführung des Child-Tasks T_2 erkennt die Kontrollfunktion, dass T_2 in der logischen `Merge`-Reihenfolge vor den Tasks T_3 und T_5 liegt und alle Operationen aus derselben Merge-Epoche stammen. Aus diesem Grund wird die Reihenfolge der Parameter für die Transformation sowohl für die Operation β umgekehrt, die von Task T_3 erzeugt wurde, als auch für die Operationen γ' und δ', die von Task T_5 erzeugt wurden (in Abbildung 5.10c dargestellt durch rote Übergänge). Wird abschließend der Child-Task T_4 zusammengeführt, so erkennt die Kontrollfunktion, dass die Operation γ dem Parent-Task bereits bekannt ist und ignoriert diese entsprechend (Abbildung 5.10d). Nach der Zusammenführung ergibt sich als Ergebnis die Liste $[Z, A, X, B^{\dagger}, Y, C]_4\ \beta_1\ \gamma'_2\ \delta'_3\ \alpha'''_4$, die abgesehen von der Reihenfolge der Operationen dem Ergebnis der Zusammenführung in der logischen `Merge`-Reihenfolge entspricht[18].

Korrektheit und Determinismus

Um die Korrektheit des zuvor beschriebenen OT-Systems zu zeigen, wird in diesem Kapitel gezeigt, dass bei einer Zusammenführung in beliebiger Reihenfolge dieselben Konflikte zwischen Operationen auf dieselbe Art und Weise aufgelöst werden, wie bei einer

[18]Für eine ausführliche Auflistung der durchgeführten Transformationen der Operationen siehe Kapitel C.2 im Anhang.

Zusammenführung in der logischen `Merge`-Reihenfolge. Insbesondere wird dazu gezeigt, dass dieselben Konfliktauflösungen durchgeführt werden (d.h. dass keine neuen Konflikte hinzugekommen sind, keine vorher bestehenden Konflikte verloren gegangen sind und bestehende Konflikte auf dieselbe Weise aufgelöst werden). Unter diesen Annahmen ist das Ergebnis der Zusammenführung äquivalent zu dem deterministischen Ergebnis der Zusammenführung in der logischen `Merge`-Reihenfolge, da dieselben Transformationen durchgeführt wurden und sich lediglich die Reihenfolge der Anwendung der Operationen unterscheidet.

Satz 1. *Zwei Operationen Op_1 und Op_2, die nicht auf dieselbe Position zeigen, können nicht durch eine Transformation gegen eine dritte Operation Op_X auf dieselbe Position verschoben werden.*

Beweis von Satz 1 durch Widerspruch. Damit zwei Operationen Op_1 und Op_2, die auf unterschiedliche Positionen zeigen, durch eine Transformation gegen eine dritte Operation Op_X auf dieselbe Position verschoben werden, muss es eine Transformation geben, die das Feld *Position* für eine der beiden Operationen so verändert, dass sie der Position der anderen Operation entspricht. Die Operationen Op_1, Op_2 und Op_X sind in den Formeln 5.19, 5.20 und 5.21 beschrieben. Dadurch, dass sie *nicht* auf dieselbe Position zeigen ist entweder die Position von Op_1 kleiner als die Position von Op_2 (siehe Formel 5.22) oder umgekehrt (für die Betrachtung des umgekehrten Falles können im weiteren Beweis die Operationen Op_1 und Op_2 vertauscht werden).

$$Op_1 := (OperationCode_1,\ Element_1,\ Position_1,\ Version_1,\ CreatingEntity_1) \tag{5.19}$$

$$Op_2 := (OperationCode_2,\ Element_2,\ Position_2,\ Version_2,\ CreatingEntity_2) \tag{5.20}$$

$$Op_X := (OperationCode_X,\ Element_X,\ Position_X,\ Version_X,\ CreatingEntity_X) \tag{5.21}$$

$$Position_1 \overset{!}{<} Position_2 \tag{5.22}$$

In der Transformationsfunktion *TRANSFORM* (siehe Listing 5.3) gibt es keine Transformation, die die Position einer Operation dekrementiert. Wird die Position einer Operation verändert, so geschieht dies immer durch das Inkrementieren der Position um 1. Damit die Operationen Op_1 und Op_2 auf dieselbe Position verschoben werden, müsste es dementsprechend eine Situation geben, in der die Operation Op_X die Position von Op_1 inkrementiert, die Position von Op_2 hingegen nicht. Das Inkrementieren von $Position_1$ kann nur dann eintreten, wenn die $Position_X$ kleiner als $Position_2$ ist. In diesem Falle gilt aber gleichzeitig (durch Formel 5.22) auch, dass die $Position_X$ kleiner ist als $Position_2$ (siehe Formel 5.23). $Position_2$ würde somit ebenfalls inkrementiert und es gilt weiterhin $Position_1 < Position_2$. Die Operationen Op_1 und Op_2 können somit nicht von einer dritten Operation Op_X auf diesel-

be Position verschoben werden[19], was insbesondere bedeutet, dass keine neuen Konflikte entstehen können □.

$$Position_X < Position_1 \Rightarrow Position_X < Position_2 \tag{5.23}$$

Satz 2. *Zwei Operationen Op_1 und Op_2, die auf dieselbe Position zeigen, können nicht durch die Transformation gegen eine dritte Operation Op_X auf unterschiedliche Positionen verschoben werden.*

Beweis von Satz 2 durch Widerspruch. Damit zwei Operationen Op_1 und Op_2, die auf dieselbe Operation zeigen (siehe Formel 5.24), durch die Transformation gegen eine dritte Operation Op_X so verändert werden, dass sie nicht weiter auf dieselbe Position zeigen, müssen Op_1 und Op_2 unterschiedlich gegen Op_X transformiert werden. Dies bedeutet, dass die Position einer der Operationen verändert werden muss, während die andere Operation unverändert bleibt.

$$Position_1 \stackrel{!}{=} Position_2 \tag{5.24}$$

Die Transformationsfunktion *TRANSFORM* (siehe Listing 5.3) trifft die Entscheidung, ob die Position einer Operation inkrementiert werden muss. Dazu wird die Position der Operation, die als erster Parameter übergeben wurde (hier $Position_1$ oder $Position_2$), mit der Position der Operation, die als zweiter Parameter übergeben wurde (hier $Position_X$), verglichen. Für jedes mögliche Vergleichsergebnis von $Position_1$ und $Position_X$ gilt dabei, dass ein Vergleich von $Position_2$ und $Position_X$ dasselbe Ergebnis ausgibt (siehe die Formeln 5.25, 5.26 und 5.27). Beide Operationen Op_1 und Op_2 würden somit auf die gleiche Weise transformiert werden und nach der Transformation weiterhin auf dieselbe Position zeigen (d.h. insbesondere, dass bestehende Konflikte weiterhin bestehen bleiben) □.

$$Position_1 < Position_X \Rightarrow Position_2 < Position_X \tag{5.25}$$

$$Position_1 > Position_X \Rightarrow Position_2 > Position_X \tag{5.26}$$

$$Position_1 = Position_X \Rightarrow Position_2 = Position_X \tag{5.27}$$

Satz 3. *Die Konfliktauflösung innerhalb einer Merge-Epoche führt unabhängig von der Reihenfolge der Zusammenführung deterministisch zu demselben Ergebnis.*

Beweis von Satz 3. Satz 1 und Satz 2 stellen sicher, dass zwei Operationen, die nicht auf diesel-

[19] Aus dieser Argumentation ergibt sich des Weiteren, dass die Operationen Op_1 und Op_2 einander nicht überspringen können, da sich die Position bei jeder Transformation maximal um den Wert 1 verschiebt und die Operationen zuerst auf dieselbe Position geraten müssten, bevor sie einander überspringen können. Dies ist jedoch wie bereits beschrieben nicht möglich.

be Position zeigen auch durch eine Transformation nicht auf dieselbe Position verschoben werden können und dass ebenso zwei Operationen, die auf dieselbe Position zeigen nach einer Transformation weiterhin auf dieselbe Position zeigen. Um darauf aufbauend sicherzustellen, dass Konflikte innerhalb einer Merge-Epoche immer auf dieselbe Weise aufgelöst werden, kann die Kontrollfunktion die Parameterreihenfolge der Transformationsfunktion umkehren. Dies geschieht immer dann, wenn eine bereits beim Parent-Task vorliegende Operation Op_P von einem Child-Task erstellt wurde, der in der logischen `Merge`-Reihenfolge später zusammengeführt werden müsste, als die gerade neu hinzukommende Operation Op_C. Der Grund dafür ist, dass die Transformation in logischer `Merge`-Reihenfolge auch mit umgekehrter Parameterreihenfolge stattgefunden hätte. Somit wird erreicht, dass insbesondere für jedes Paar aus einer Menge aller konfligierenden Operationen an einer Position X (im Folgenden beschrieben durch die Menge $Ops_{K,X}$) eine deterministische Ordnung erreicht wird (*Trichotomie*, siehe Formel 5.28). Des Weiteren gilt, dass nach der Transformation zwischen allen Operationen eine *Transitivität* besteht (siehe Formel 5.29). Somit ergibt sich für alle konfligierenden Operationen an jeder Position eine *Strenge Totalordnung* [79]. Diese ist dabei deterministisch vorgegeben durch die logische `Merge`-Reihenfolge □.

$$x < y \lor x = y \lor x > y \quad \text{für alle } x, y \in Ops_{K,X} \tag{5.28}$$

$$x < y \land y < z \Rightarrow x < z \quad \text{für alle } x, y, z \in Ops_{K,X} \tag{5.29}$$

Als zusätzliche Validitätsprüfung wurde der Algorithmus wie hier beschrieben implementiert und anschließend ein Testszenario durchlaufen. In diesem Testszenario erstellt ein Parent-Task eine zusammenführbare Liste und nimmt an dieser sechs Modifikationen vor (5× `insert` und 1× `delete`). Während diese Modifikationen durchgeführt werden, werden zu unterschiedlichen Zeitpunkten (d.h. nach unterschiedlichen Modifikationen durch den Parent-Task) insgesamt drei Child-Tasks mit der Liste gestartet. Die Zeitpunkte für das Starten der Child-Tasks wurden so gewählt, dass jede mögliche Verschachtlung von Startzeitpunkten abgedeckt wird. Der initiale Zustand der Liste des Child-Tasks hängt dabei von dem Zeitpunkt ab, an dem der Child-Task gestartet wurde. Jeder gestartete Task (inklusive Parent-Task) führt seinerseits in jedem Testlauf eine einzelne zusätzliche Operation (`insert` oder `delete`) an einer Position in der Liste aus (in der Weise, dass jede Kombination von Operationen und Positionen der unterschiedlichen Tasks abgedeckt wird). Abschließend werden die drei Child-Tasks gezielt in jeder möglichen Reihenfolge zusammengeführt, um zu prüfen, dass sich deterministisch immer dasselbe Ergebnis ergibt. Hierbei wurden insgesamt 2.667.168 Kombinationen überprüft.

Abschließende Betrachtung

In diesem Kapitel wurde gezeigt, wie ein bestehendes TP1-OT-System erweitert werden kann, um eine deterministische Zusammenführung in beliebiger Reihenfolge (insbesondere der FCFS-Reihenfolge) zu ermöglichen. Da OT-Systeme (und insbesondere die darin verwendeten Transformationsfunktionen) abhängig von der Datenstruktur stark variieren können, ist es nicht möglich eine feste Anleitung zu entwerfen, nach der sich ein beliebiges TP1-OT-System zu einem OT-System für eine Zusammenführung in beliebiger Reihe erweitern lässt. Allerdings können (in Anlehnung an den Beweis für die Korrektheit des im Vorhinein beschriebenen OT-Systems in Kapitel 5.3.3) Eigenschaften definiert werden, mit deren Hilfe eine deterministische Zusammenführung in beliebiger Reihenfolge ermöglicht werden kann. So kann ein deterministisches Ergebnis für die Zusammenführung in beliebiger Reihenfolge sichergestellt werden, wenn unabhängig von der Reihenfolge immer *dieselben Konflikte* auf *dieselbe Weise* aufgelöst werden, die bei einer Zusammenführung in der deterministischen logischen `Merge`-Reihenfolge aufgelöst worden wären. Sollte eine Zusammenführung in beliebiger Reihenfolge notwendig, aber eine Erweiterung eines bestehenden TP1-OT-Systems nicht möglich oder die Umsetzung zu aufwändig sein, so gibt es weiterhin die Möglichkeit zusammenführbare Datenstrukturen zu verwenden, die intern ein TP2-OT-System verwenden.

Es gibt allerdings auch Einschränkungen, die sich aus der Zusammenführung in beliebiger Reihenfolge ergeben. Wird eine Zusammenführung in beliebiger Reihenfolge durchgeführt, so ist im Kontext von Spawn & Merge die Verwendung der `Sync`-Primitive mit dem Sync-Modus `RETURN_DIRECTLY` nicht möglich[20]. Bei `RETURN_DIRECTLY` sollen, im Gegensatz zum dem Sync-Modus `RETURN_AFTER_MERGE_CYCLE_COMPLETED`, nur diejenigen Veränderungen aller Child-Tasks im Ergebnis enthalten sein, die in der logischen `Merge`-Reihenfolge *vor* dem aktuell zusammengeführten Child-Task liegen (siehe Kapitel 4.2.9). Dies kann jedoch nicht sichergestellt werden, da die Reihenfolge der Zusammenführung nichtdeterministisch ist. Somit können bereits Child-Task Veränderungen zusammengeführt worden sein, die nicht im `Sync`-Ergebnis des aktuellen Child-Tasks enthalten sein dürfen, während die benötigten Veränderungen anderer Child-Tasks noch nicht vorliegen. Der Parent-Task kann zu Beginn der Merge-Epoche nicht wissen, ob ein Child-Task mit einem `RETURN_DIRECTLY` `Sync`-Aufruf erwartet wird. Somit kann diese Information nicht in die Entscheidung, ob eine Zusammenführung in beliebiger Reihenfolge durchgeführt werden kann, miteinbezogen werden. Als „Notlösung“ werden `RETURN_DIRECTLY` `Sync`-Aufrufe im Falle einer Zusammenführung in beliebiger Reihenfolge zum Sync-Modus `RETURN_AFTER_MERGE_CYCLE_COMPLETED` geändert und eine Warnung ausgegeben, sodass weiterhin ein deterministisches Ergebnis erreicht wird.

Ebenso verhindert die Zusammenführung in beliebiger Reihenfolge die Verwendung

[20] Die hier beschriebenen Probleme gelten dabei auch bei der Verwendung eines TP2-OT-Systems.

von Bedingungsfunktionen (siehe Kapitel 4.2.8), da diese eine deterministische Eingabe erwarten. Der Zustand der zusammengeführten Datenstrukturen ist hier (innerhalb der `Merge`-Primitive) allerdings abhängig von der nichtdeterministischen Reihenfolge in der die Child-Tasks zusammengeführt werden. Somit können Bedingungsfunktionen nicht zusammen mit einer Zusammenführung in beliebiger Reihenfolge verwendet werden.

Für den Entwickler einer auf Spawn & Merge basierenden Anwendung ergibt sich aus der Einführung der Zusammenführung in beliebiger Reihenfolge keine zusätzliche Komplexität, da die notwendige Funktionalität komplett innerhalb des Frameworks und den zusammenführenbaren Datenstrukturen realisiert ist. Das Framework kann dabei automatisch für jede Merge-Epoche entscheiden, ob eine Zusammenführung in beliebiger Reihenfolge erlaubt ist, oder ob für ein deterministisches Ergebnis eine Zusammenführung in der festen Spawn-Reihenfolge notwendig ist. Die Entscheidungsgrundlage bieten hierbei die Datenstrukturkopien aller Child-Tasks, die im Verlaufe des `Merge`-Aufrufes zusammengeführt werden *könnten*[21]. Nur wenn alle an die Child-Tasks übergebenen Datenstrukturen eine Zusammenführung in beliebiger Reihenfolge unterstützen, dürfen die Child-Tasks der aktuellen Merge-Epoche auch in der FCFS-Reihenfolge zusammengeführt werden.

5.3.4 Komplexität des neuen Algorithmus

Auch ohne die Nutzung von Zustandsvektoren (die in vielen TP2-OT-Systemen verwendet werden) sorgt die Einführung von Tombstones (die eine Veränderung der Konfliktauflösung verhindern) für die in diesem Kapitel beschriebene Liste dafür, dass das entstehende OT-System einen höheren Speicherverbrauch hat als das ursprüngliche TP1-OT-System. Dies liegt daran, dass gelöschte Elemente nicht verworfen werden können, sondern weiterhin (als gelöscht markiert) beibehalten werden müssen. Zusätzlich erhöht sich mit der Anzahl an Tombstones auch die Menge der zu übertragenden Daten im Rahmen der Synchronisationsprimitive. Die in Kapitel 5.3.3 beschriebene Zusammenfassung nebeneinanderliegender Tombstones beschränkt dabei den Overhead auf maximal $m+1$ Tombstones bei einer Liste mit m ungelöschten Elementen.

Auch die Komplexität der Anwendung von Operationen auf die Liste erhöht sich durch die Einführung von Tombstones. Dies liegt daran, dass Operationen nicht mehr direkt auf die (in der Operation) angegebene Position innerhalb der Liste angewendet werden können, sondern die Position in der Liste erst (unter Berücksichtigung der Tombstones) ermittelt werden muss.

Die Komplexität des neuen Transformationssystems entspricht weiterhin der Komplexität des ursprünglichen Transformationssystems ($O(n^2)$ wobei n der Anzahl der durchge-

[21] „Könnten", da bei einer Verwendung von `Sync` nicht notwendigerweise tatsächlich alle Datenstrukturen zusammengeführt werden, die dem Child-Task initial übergebenen wurden.

führten Operationen entspricht). Die hinzugefügten Vergleiche zur Entscheidung, ob die Parameter der Transformationsfunktion getauscht werden müssen, fallen als lineare Komponente im Rahmen der $\mathcal{O}$-Notation weg, auch wenn sie in der Implementierung einen Mehraufwand für die Berechnung bedeuten.

Die hier beschriebenen Nachteile des erweiterten OT-Systems im Vergleich zum ursprünglichen OT-System werden in Kapitel 7.5.1 im Rahmen der Evaluation weiter untersucht.

5.4 Zusammenführbare Map

Die deterministische Zusammenführung nebenläufig modifizierter Listen erfordert komplexe Mechanismen, die im Verlaufe dieses Kapitels beschrieben wurden. Es gibt allerdings auch einfachere zusammenführbare Datenstrukturen, deren Mechanismen weniger komplex sind. Ein Beispiel dafür ist die *Map* [56]. Eine *Map* ist eine Datenstruktur, die unter einem *Schlüssel* (*Key*) einen *Wert* (*Value*) speichern kann. Im Folgenden wird beschrieben, wie eine zusammenführbare Map funktionieren kann.

Eine Map erlaubt als Operationen das *Zuweisen* (`set`) eines Wertes für einen Schlüssel und das *Löschen* (`delete`) eines Wertes unter einem Schlüssel. Die Operationen sind dabei ein 5-Tupel bestehend aus dem *OperationCode* (`set`, `delete` oder `no-op`), dem *Wert*, dem *Schlüssel*, der *Version* und der Angabe des Erstellers (*CreatingEntity*), wie in Formel 5.30 zu sehen.

$$Map_Operation := (OperationCode, Wert, Schluessel, Version, CreatingEntity) \tag{5.30}$$

Für die Zusammenführung zweier Maps gilt, dass Konflikte so aufgelöst werden, dass die Operation des Teilnehmers mit der höchsten Priorität (z.B. der Teilnehmer, der zuletzt versucht hat die Daten zu verändern („last write wins“)) gewinnt. Eine anderweitige Beeinflussung der Operationen untereinander (wie beispielsweise das Verschieben der Positionen im Falle der zusammenführbaren Liste) gibt es im Falle der Map nicht. Eine Transformationsfunktion für die Zusammenführung zweier Maps ist in Listing 5.5 zu sehen. Hier wird die Operation des Teilnehmers, der Priorität hat, als erster Parameter (`op1`) übergeben.

Listing 5.5 Transformationsfunktion *MAP_TRANSFORM*.

```
1: function MAP_TRANSFORM(op1(val1, key1), op2(val2, key2)):
2:   if key1 == key2: return (op1(val1, key1), no-op)
3:   else: return (op1(val1, key1), op2(val2, key2)
```

In Zeile 2 ist die Konfliktauflösung realisiert, die sicherstellt, dass die Operation mit niedrigerer Priorität (`op2`) bei der Map mit höherer Priorität nicht angewendet wird. Dazu wird die Operation zu einem `no-op` umgewandelt. Falls die Operationen nicht im Konflikt stehen, dann können sie ohne Veränderung beibehalten werden, da sie sich nicht gegenseitig beeinflussen.

Für eine deterministische Zusammenführung in beliebiger Reihenfolge müssen die Konflikte zwischen Operationen immer auf dieselbe Weise aufgelöst werden. Dies kann hierbei, ebenso wie bei der im Vorhinein beschriebenen zusammenführbaren Liste, durch das Speichern der *CreatingEntity* für jede Operation zusammen mit der Betrachtung einer festen Reihenfolge (im Kontext von Spawn & Merge der logischen `Merge`-Reihenfolge) realisiert werden. Hierbei muss immer diejenige Operation priorisiert werden, deren Ersteller (*CreatingEntity*) in der festen Reihenfolge am weitesten hinten eingeordnet ist, sofern beide Operationen in derselben Merge-Epoche liegen. Neu entstehende Konflikte oder wegfallende Konflikte kann es hier nicht geben, da sich die Schlüssel der Operationen nicht verändern.

5.5 Verschachtlung zusammenführbarer Datenstrukturen

Um vielseitigere und komplexere zusammenführbare Datenstrukturen zu erstellen, ist es möglich, zusammenführbare Datenstrukturen zu verschachteln. In diesem Kapitel wird dies beispielhaft durch die theoretische Verschachtlung der bereits beschriebenen zusammenführbaren Liste und zusammenführbaren Map demonstriert. Die komplexe Datenstruktur soll in einer Liste oder in einer Map als Elemente wiederum zusammenführbare Listen oder Maps erlauben. Die sich ergebende Baumstruktur ähnelt somit dem Datenformat der *JavaScript Object Notation (JSON)* [17]. Hauptaugenmerk dieses Kapitels liegt dabei auf der Funktionsweise eines verschachtelten OT-Systems.

Wird eine zusammenführbare Datenstruktur D_1 in eine andere zusammenführbare Datenstruktur D_2 eingebettet so wird dies hier als *Verschachtlung* bezeichnet. D_1 wird hier als „untergeordnet" und D_2 als „übergeordnet" bezeichnet. Diese Verschachtlung kann dabei auf beliebig viele Ebenen erweitert werden (d.h. D_1 können weitere zusammenführbare Datenstrukturen untergeordnet sein). Werden zwei Datenstrukturen zusammengeführt, die wiederum untergeordnete Datenstrukturen beinhalten, so muss das OT-System in der Lage sein, dies nachzuvollziehen. Diese Funktionalität liegt dabei nur in der Datenstruktur und ist unabhängig vom Spawn & Merge Framework.

Zu diesem Zweck wird eine neue Operation `change` eingeführt. Die `change`-Operation gibt an, dass ein untergeordnetes Element (d.h. eine zusammenführbare Datenstruktur) verändert wurde. Die *Position*/der *Schlüssel* identifiziert hierbei das veränderte Element. `change`-Operationen können dabei verschachtelt werden, um Veränderungen über meh-

rere Ebenen hinweg zu verfolgen. Das in der Operation enthaltene Feld *Element* / *Wert* beschreibt die Operation, die auf der untergeordneten Datenstruktur durchgeführt wurde (siehe Formeln 5.31 und 5.32).

$$SimpleChangeOp := (change,\ \underbrace{(insert,\ "X",\ 5,\ 1,\ "Task1")}_{\text{Untergeordnete Operation}},\ "key",\ 1,\ "Task1") \tag{5.31}$$

$$NestedChangeOp := (change,\ \underbrace{(change,\ (\ldots),\ 5,\ 1,\ "Task1")}_{\text{Verschachtelte change-Operation}},\ "key",\ 1,\ "Task1") \tag{5.32}$$

Die Operationen einer verschachtelten zusammenführbaren Datenstruktur müssen dabei immer von der höchsten übergeordneten Datenstruktur ausgehend (in eben dieser) gespeichert werden. Dies ist aus zwei Gründen notwendig: Zum einen wird die Transformation (im Kontext von Spawn & Merge) im Rahmen der `Merge`-Primitive für eben diese höchste Datenstruktur aufgerufen. Falls diese nicht über Veränderungen innerhalb der untergeordneten Datenstrukturen informiert ist, müssten alle untergeordneten Datenstrukturen durchlaufen werden, um eventuell vorliegende Veränderungen festzustellen. Zum anderen kann es passieren, dass eine untergeordnete Datenstruktur D_1 durch eine neue Datenstruktur D_2 ersetzt wird und anschließend (fälschlicherweise) Operationen für D_1 auf D_2 angewendet werden. Dies kann dadurch verhindert werden, dass die übergeordnete Datenstruktur anhand der `change`-Operationen den Konflikt (Änderung eines bereits überschriebenen Elementes) feststellen und auflösen kann.

Durch die Erweiterung der Menge an möglichen Operationen müssen auch die entsprechenden Transformationsfunktionen der zusammenführbaren Datenstrukturen um die hinzugefügte `change`-Operation erweitert werden. Für die Map bedeutet dies, dass sich die Auflösung eines Konfliktes ändert, falls es sich um zwei `change`-Operationen handelt. Die erweiterte Transformationsfunktion ist in Listing 5.6 zu sehen. Hier wird im Konfliktfall für zwei `change`-Operationen eine Transformationsfunktion *TRANSFORM* für die beiden Operationen aufgerufen (Zeile 4). Anschließend werden die in den `change`-Operationen als Werte enthaltenen Operationen durch die transformierten Operationen ersetzt und zurückgegeben (Zeile 5). Die tatsächlich aufgerufene Transformationsfunktion ist dabei abhängig von der untergeordneten zusammenführbaren Datenstruktur, auf die sich die Operationen in `val1` und `val2` beziehen.

Um die in Kapitel 5.3.3 beschriebene zusammenführbare Liste um `change`-Operationen zu erweitern, wird im Folgenden die Transformationsfunktion aus Listing 5.3 erweitert (siehe Listing 5.7). Die bereits in Kapitel 5.3.3 beschriebenen Transformationen zwischen `insert`- und `delete`-Operationen untereinander sind hierbei nicht abgebildet (Zeile 1). Stehen zwei `change`-Operationen in Konflikt miteinander (d.h. sie beziehen sich auf dieselbe

Listing 5.6 Erweiterte Transformationsfunktion *MAP_TRANSFORM*.

```
1: function MAP_TRANSFORM(op1(val1, key1), op2(val2, key2)):
2:   if key1 == key2:
3:     if op1 == change && op2 == change:
4:       val1upd, val2upd = TRANSFORM(val1, val2)
5:       return (op1(val1upd, key1), op2(val2upd, key2))
6:     else:
7:       return (op1(val1, key1), no-op)
8:   else: return (op1(val1, key1), op2(val2, key2)
```

Position $k1 == k2$), dann werden die in `i1` und `i2` enthaltenen Operationen gegeneinander transformiert (Zeile 5) und anschließend zurückgegeben (Zeile 6). Auch hier ist die tatsächlich in Zeile 5 aufgerufene Transformation abhängig vom Typ der untergeordneten Datenstruktur. Stehen die beiden Operationen nicht in Konflikt, so beeinflussen sie sich nicht und brauchen nicht verändert werden, da sich durch eine `change`-Operation keine Verschiebung ergibt.

Wird eine `change`-Operation gegen eine `insert`-Operation transformiert, so entspricht die durchzuführende Transformation der Transformation einer `delete`-Operation gegen eine `insert`-Operation, da sich bei `delete`-Operationen ebenfalls keine Verschiebung ergibt (Zeile 10 – Zeile 18). Die Transformation einer `change`-Operation gegen eine `delete`-Operation stellt eine weitere Besonderheit dar: Stehen beide Operationen in Konflikt, so bedeutet dies, dass entweder das Element zuerst gelöscht und anschließend bearbeitet werden oder zuerst bearbeitet und anschließend gelöscht werden soll. Aus diesem Grund wird die `change`-Operation hier in eine `no-op`-Operation umgewandelt, da eine Anwendung der Modifikation keinen Sinn macht (Zeile 21 und Zeile 25). Zeigen beide Operationen hingegen auf unterschiedliche Elemente der Liste, so beeinflussen sich diese nicht und können unverändert zurückgegeben werden (Zeile 22 und Zeile 26).

Zur Veranschaulichung sei das Beispiel in Abbildung 5.11 gegeben, in dem ein Task T_1 eine zusammenführbare Map besitzt, die unter dem Schlüssel $k1$ eine zusammenführbare Liste gespeichert hat. Diese Map wird beim Starten eines neuen Tasks T_2 übergeben. Beide Tasks führen nun nebenläufig die Operationen α und β (Task T_1) und γ (Task T_2) durch. Anschließend führt Task T_1 die Veränderungen von T_2 unter Verwendung einer `Merge`-Primitive wieder zusammen.

Werden nun beide Maps wieder zusammengeführt, so stellen diese im Rahmen der Transformation einen Konflikt zwischen α und γ, sowie zwischen β und γ' fest[22]. Dadurch wird die Transformation für die in den `change`-Operationen enthaltenen Operationen gestartet (siehe Listing 5.6). Dabei wird zuerst $insert(2, 'X')$ gegen $insert(1, 'Y')$ transformiert,

[22]Bei letzterem Konflikt hat sich γ bereits durch die Transformation gegen α zu γ' verändert.

Listing 5.7 Um change erweiterte Transformationsfunktion *FCFS_TRANSFORM*.

```
1:  [...]
2:
3:  function FCFS_TRANSFORM(change(i1, k1), change(i2, k2)):
4:    if k1 == k2:
5:      i1upd, i2upd = TRANSFORM(i1, i2)
6:      return (change(i1upd, k1), change(i2upd, k2)
7:    else:
8:      return (change(i1, k1), change(i2, k2))
9:
10: function FCFS_TRANSFORM(change(k1), insert(i, k2)):
11:   if k1 <  k2: return(change(k1)    , insert(i, k2))
12:   if k1 >  k2: return(change(k1 + 1), insert(i, k2))
13:   if k1 == k2: return(change(k1 + 1), insert(i, k2))
14:
15: function FCFS_TRANSFORM(insert(i, k1), change(k2)):
16:   if k1 <  k2: return(insert(i, k1), change(k2 + 1))
17:   if k1 >  k2: return(insert(i, k1), change(k2)    )
18:   if k1 == k2: return(insert(i, k1), change(k2 + 1))
19:
20: function FCFS_TRANSFORM(change(k1), delete(k2)):
21:   if k1 == k2: return(no-op, delete(k2))
22:   else: return(change(k1), delete(k2))
23:
24: function FCFS_TRANSFORM(delete(k1), change(k2)):
25:   if k1 == k2: return(delete(k1), no-op)
26:   else: return(delete(k1), change(k2))
```

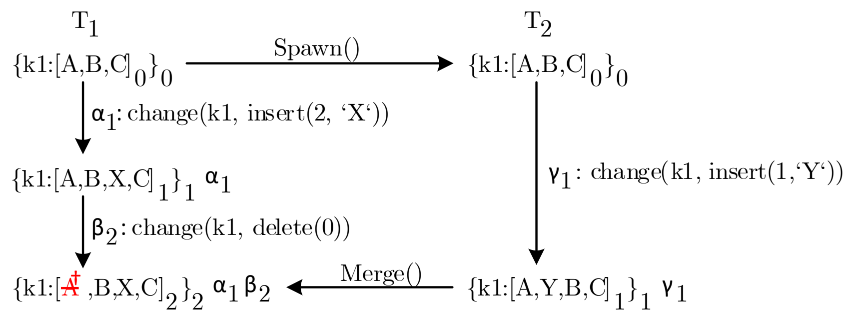


Abbildung 5.11: Beispielszenario für verschachtelte Operationen.

wodurch sich die Operationen $\alpha' = change(k1, insert(3,' X'))$ und $\gamma' = change(k1, insert(1,' Y'))$ ergeben (siehe Listing 5.3). Anschließend wird $delete(0)$ gegen $insert(1,' Y')$ transformiert, wodurch sich $\beta' = change(k1, delete(0))$ und $\gamma'' = change(k1, insert(1,' Y'))$ ergeben, da beide

Operationen nicht verändert werden. Wird γ'' nun bei Task T_1 angewendet, so ergibt sich als Ergebnis der verschachtelten Transformation die Map $\{\text{k1} : [A^{\dagger}, Y, B, X, C]_3\}_3\ \alpha_1\ \beta_2\ \gamma_3''$. Das Ergebnis entspricht somit dem Ergebnis, das bei einer direkten Transformation der untergeordneten Listen entstanden wäre und zeigt, dass zusammenführbare Datenstrukturen zu komplexeren zusammenführbaren Datenstrukturen (für die Nutzung im Spawn & Merge Framework) kombiniert werden können.

Kapitel 6

Spawn & Merge für Verteilte Systeme

Im folgenden Kapitel wird die Umsetzung des Spawn & Merge Programmiermodells auf einem Verteilten System beschrieben. So soll gezeigt werden, dass das Spawn & Merge Konzept realisierbar ist und sich mit Hilfe von Spawn & Merge deterministische Verteilte Systeme entwickeln lassen. Der in diesem Kapitel beschriebene Proof-of-Concept Prototyp wird anschließend in Kapitel 7 evaluiert.

Für die Beschreibung der Umsetzung wird zunächst das zugrundeliegende Systemmodell definiert. Anschließend wird beschrieben, wie sich das Konzept des Determinismus auf Applikationsebene und das Spawn & Merge Programmiermodell auf das definierte Systemmodell abbilden lassen. Dabei wird auch beschrieben, welche Optimierungen durchgeführt werden können, damit eine verteilte Spawn & Merge Anwendung vom Vorteil eines gemeinsam genutzten Arbeitsspeichers Gebrauch machen kann (falls die Anwendung auf nur einem Rechenknoten ausgeführt wird).

6.1 Systemmodell

In diesem Kapitel wird das Systemmodell definiert, auf dem die prototypische Umsetzung des Spawn & Merge Programmiermodells für Verteilte Systeme realisiert wird. Dabei wird insbesondere auf die Architektur des Verteilten Systems und das Fehlermodell eingegangen. Des Weiteren wird beschrieben, welche Implikationen sich aus dem Umstand ergeben, dass im Spawn & Merge Programmiermodell geteilte Datenstrukturen für jeden Task kopiert werden müssen.

6.1.1 Architektur

Für die Ausführung der Anwendung wird davon ausgegangen, dass das Verteilte System aus Rechenknoten besteht, die ein lokal verbundenes Cluster bilden und eine stabile Verbindung zueinander haben. Die in diesem Kapitel beschriebene Umsetzung des Spawn

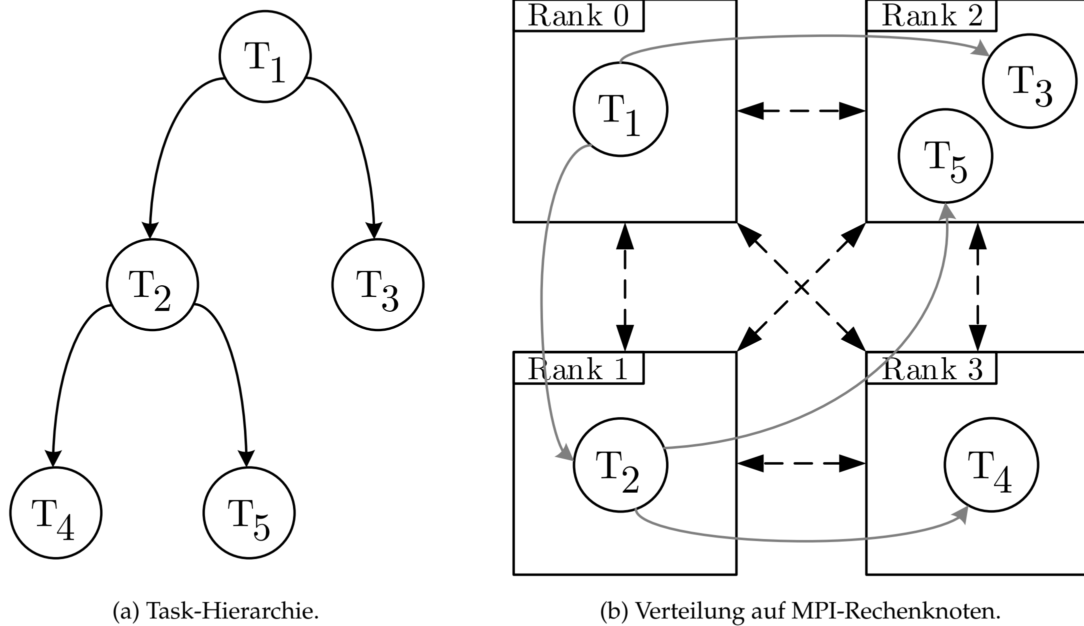
& Merge Programmiermodells für Verteilte Systeme nutzt Nachrichtenaustausch [51] als interne Synchronisationsmechanik (auf Frameworkebene) zur Kommunikation zwischen den einzelnen Rechenknoten. Da in Spawn & Merge die geteilten Datenstrukturen für jeden gespawnten Task kopiert werden und die Tasks isoliert voneinander ausgeführt werden, eignet sich ein auf Nachrichtenaustausch basierendes Protokoll für die Umsetzung des Frameworks. Einzelne gestartete Tasks können so in eine einzelne Nachricht gekapselt werden, die an einen Rechenknoten mit freien Kapazitäten gesendet werden kann. Die Nutzung von Nachrichtenaustausch zur Synchronisation ermöglicht es außerdem, dass die Anwendungen nicht nur in Systemen mit geteiltem Arbeitsspeicher eingesetzt werden können. Zur Umsetzung der Synchronisationsprimitive könnten allerdings auch andere Ansätze genutzt werden, wie beispielsweise Remote Procedure Calls [93].

Die hier konzipierte Umsetzung des Spawn & Merge Programmiermodells setzt auf dem Message Passing Interface (MPI) [69] auf. Das MPI ist für viele Programmiersprachen und Systemarchitekturen geeignet (z.B. Multiprozessor-Systeme mit verteiltem Arbeitsspeicher, Netzwerke von Rechenknoten und Mehrkernprozessoren mit geteiltem Arbeitsspeicher [69]) und ermöglicht den Nachrichtenaustausch zwischen einzelnen Rechenknoten, sowie das Management der Rechenknoten. Auf jedem Rechenknoten in dem Verteilten System, auf dem die verteilte Anwendung ausgeführt werden soll, wird eine Instanz der Anwendung ausgeführt. Die Anwendung beinhaltet wiederum eine MPI-Implementierung, die sich eigenständig mit den weiteren Instanzen koordiniert.

MPI bietet der darauf aufbauenden Anwendung eine klar definierte Schnittstelle für die Kommunikation zwischen Rechenknoten und das Knotenmanagement an, die unabhängig von der Implementierung des MPI-Frameworks ist (siehe Kapitel 2.1.3). Über diese Schnittstelle können Nachrichten direkt von einem Rechenknoten zu einem anderen Rechenknoten gesendet werden, ohne dass das Spawn & Merge Framework selbst eine Verbindung zwischen den Knoten herstellen muss.

Jeder Rechenknoten kann einen oder mehrere Tasks einer Task-Hierarchie gleichzeitig ausführen. Der erreichte Grad an Parallelität ist dabei abhängig von der Anzahl verfügbarer Prozessorkerne. Abbildung 6.1 zeigt, wie eine Task-Hierarchie (Abbildung 6.1a) beispielhaft auf vier verfügbare Rechenknoten verteilt werden kann (Abbildung 6.1b). Hier werden die Tasks T_1–T_5 auf die MPI-Knoten 0–3 verteilt. Die Auswahl von MPI-Knoten für die Ausführung der einzelnen Tasks ist dabei unabhängig von der Task-Hierarchie. Die logischen Abhängigkeiten zwischen Parent-Tasks und Child-Tasks ergeben sich weiterhin aus den Informationen, die innerhalb der einzelnen Tasks gespeichert sind (siehe Kapitel 4.2.3).

Bei diesem Systemmodell ergeben sich mehrere Herausforderungen für die Umsetzung. Zum einen muss sichergestellt werden, dass die Anwendung auf Applikationsebene auch dann deterministisch ausgeführt wird, wenn das Framework nichtdeterministisch arbeitet. Dazu müssen die Spawn & Merge Synchronisationsprimitive auf ein Nachrichtenprotokoll



(a) Task-Hierarchie.

(b) Verteilung auf MPI-Rechenknoten.

Abbildung 6.1: Beispielhafte Verteilung gespawnter Tasks auf MPI-Knoten.

abgebildet, und die Schnittstellen zwischen der Applikationsebene und der Frameworkebene, wie in Kapitel 4.2 beschrieben, auf diese Primitive beschränkt werden. Zum anderen soll es durch die Nutzung eines dynamischen Scheduling-Algorithmus ermöglicht werden, dass die Anwendung mit den verfügbaren Ressourcen skalieren kann.

6.1.2 Nachrichtenfluss

In diesem Kapitel wird beschrieben, wie die einzelnen Komponenten des Prototyps für verteiltes Spawn & Merge miteinander über ein Protokoll, das auf Nachrichtenaustausch basiert, kommunizieren können. Nicht jede Komponente kann mit jeder anderen Komponente kommunizieren, da jede der Komponenten einen klaren Zuständigkeitsbereich besitzt.

Die grundlegende Komponente des verteilten Spawn & Merge Prototyps ist die MPI-Komponente, die die Kommunikation zwischen einzelnen MPI-Knoten ermöglicht. Der *SchedulingMaster* (siehe Kapitel 6.2.3), der für das Scheduling der Tasks auf MPI-Knoten zuständig ist, und die *NodeMaster* (siehe Kapitel 6.2.2), die für das Ausführen der Tasks auf dem eigenen MPI-Knoten zuständig sind, können auf die MPI-Komponente zugreifen und diese für das Versenden und Empfangen von Nachrichten nutzen, um untereinander zu kommunizieren. Der NodeMaster ist die einzige Komponente, die direkt mit Tasks kommunizieren kann und umgekehrt. Jegliche Kommunikation zwischen Tasks muss dementsprechend mindestens über den NodeMaster des eigenen MPI-Knotens laufen. Diese Zusammenhänge sind noch einmal in Abbildung 6.2 dargestellt.

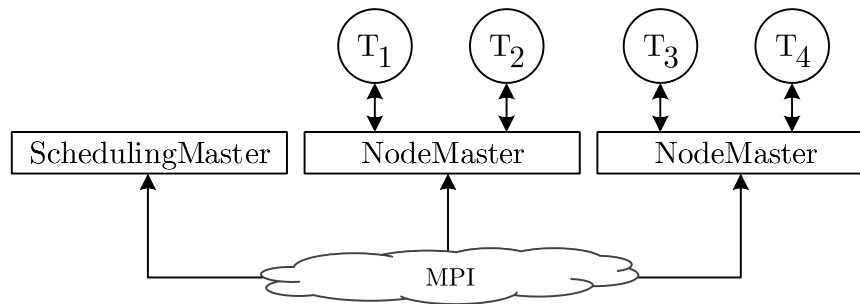


Abbildung 6.2: Kommunikation zwischen den Komponenten des Prototyps.

Die Nachrichten, die durch den SchedulingMaster und den NodeMaster über die MPI-Komponente ausgetauscht werden, haben alle denselben grundsätzlichen Aufbau, der in Abbildung 6.3 dargestellt wird. Das Feld `DestinationMpi` wird für die Zustellung der Nachricht an den korrekten MPI-Knoten benötigt. Das Feld `OpCode` beschreibt den Zweck der Nachricht und ermöglicht dem empfangenden MPI-Knoten die Zuordnung, ob die Nachricht an den NodeMaster oder an den SchedulingMaster gerichtet ist, sodass er die Nachricht weitergeben kann[1]. Die Felder `SourceMpi`, `SourceTaskId` und `ReferenceID` ermöglichen die Zuordnung einer Antwort zu dem ursprünglichen MPI-Knoten und der ursprünglichen Anfrage. Das Feld `Payload` beinhaltet die Nutzdaten, deren Struktur sich aus dem vorangegangenen `OpCode` ergibt.

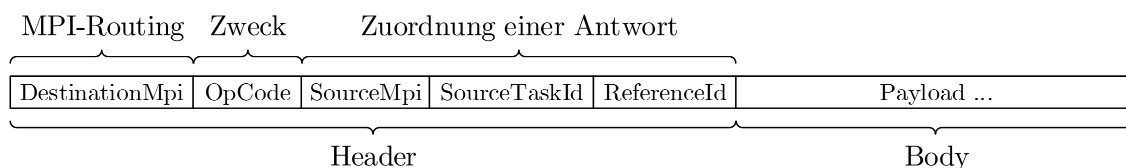


Abbildung 6.3: Aufbau einer über MPI versendeten Nachricht.

6.1.3 Fehlermodell

Bei der Ausführung von Anwendungen, die auf Spawn & Merge basieren, kann es zu unterschiedlichen Fehlerfällen kommen, die zum Teil bereits in Kapitel 4.3 beschrieben wurden. Im Folgenden wird der Umgang mit Fehlern beschrieben, die sich durch die Nutzung eines Verteilten Systems ergeben, sowie mit Fehlern, die auf der Applikationsebene zum Abbruch eines Tasks führen. Die Betrachtung von Sicherheit, im Sinne von Betrügern und Angriffen auf das System, ist nicht Teil der prototypischen Umsetzung des Spawn & Merge Programmiermodells für Verteilte Systeme.

[1] Eine vollständige Auflistung der validen `OpCodes` ist in Anhang A.4 zu sehen.

Nachrichtenübertragung

Das Message Passing Interface (MPI) bietet einer darauf aufbauenden Anwendung eine zuverlässige Nachrichtenübertragung an. Dementsprechend kann die Anwendung davon ausgehen, dass Nachrichten nicht verloren gehen, nicht doppelt zugestellt werden und nach der Zustellung auf Korrektheit geprüft wurden [69]. Treten Fehler beim Aufruf der MPI-Schnittstelle auf, die das MPI-Framework nicht selbstständig beheben kann (wie beispielsweise die Nichterreichbarkeit eines MPI-Knotens), dann wird die Anwendung standardmäßig terminiert.

MPI gibt allerdings keine Garantien darüber, in welcher Reihenfolge oder mit welcher Verzögerung Nachrichten beim Empfänger eintreffen. Die Verzögerung ist hierbei abhängig von dem zugrundeliegenden Rechencluster, während die Reihenfolge der Nachrichten neben der Übertragungsverzögerung auch von der nebenläufigen Ausführung der einzelnen Tasks abhängig ist. Im Kontext von Spawn & Merge für Verteilte Systeme bedeutet eine nichtdeterministische Nachrichtenreihenfolge, dass die Reihenfolge, in der Tasks auf den einzelnen Rechenknoten ausgeführt werden, beeinflusst wird. Diese Reihenfolge ist jedoch bereits durch die Nebenläufigkeit der einzelnen Tasks und die damit verbundene Nebenläufigkeit der `Spawn`- und `SpawnSibling`-Aufrufe nichtdeterministisch. Durch die, vom Spawn & Merge Programmiermodell sichergestellte, deterministische Reihenfolge für die Zusammenführung fertiggestellter Tasks bleibt das Ergebnis der Anwendung dennoch deterministisch und reproduzierbar.

Fehler auf Applikationsebene

Treten Fehler bei der Ausführung eines Tasks auf, dann wird die Anwendung terminiert und der Fehler auf der Konsole ausgegeben. Eine frameworkinterne, deterministische Behandlung von Fehlern innerhalb eines Tasks auf Applikationsebene ist aus mehreren Gründen nur schwer zu realisieren.

Der Absturz eines Tasks bedeutet den Verlust des gesamten Teilbaums der Task-Hierarchie, der von ihm ausgehend über seine Child-Tasks aufgespannt wird. Grundlegend laufen Tasks isoliert voneinander, sodass der Neustart eines Tasks und der neue Aufbau des verlorenen Teilbaums keinen Einfluss auf andere Tasks hat. Dies gilt allerdings nicht mehr, sobald der Task Gebrauch von der `Sync`- oder `SpawnSibling`-Primitive macht. In diesen Fällen könnte ein abgestürzter Task bereits vorläufige Ergebnisse an seinen Parent-Task übertragen oder neue Sibling-Tasks gestartet haben. Würde das Framework den abgestürzten Task einfach neu starten, so würden diese Aktionen vom Task noch einmal wiederholt werden und zu einem nichtdeterministischen Ergebnis führen.

Um dies zu verhindern, wäre ein Nachverfolgen und Aufzeichnen der Kommunikation zwischen allen Tasks notwendig (ähnlich der in Kapitel 2.2 beschriebenen Ansätze

für eine deterministische Wiederholung der Anwendungsausführung). Diese aufgezeichneten Nachrichten könnten anschließend genutzt werden, um auf wiederholte `Sync`- und `SpawnSibling`-Aufrufe dem Task die aufgezeichneten Antworten zur Verfügung zu stellen. Diese Aufzeichnung und die Analyse aller Nachrichten würde allerdings einen stark erhöhten Arbeitsspeicherverbrauch der Anwendung, sowie eine Erhöhung der benötigten Rechenzeit bedeuten. Im Rahmen dieser Arbeit liegt der Fokus auf der deterministischen Synchronisation nebenläufiger Systeme, sodass die Behandlung von Fehlern auf der Applikationsebene ausgeklammert wurde[2]. Daher wird die Anwendung beim Auftreten eines Fehlers innerhalb eines Tasks terminiert, und dem Entwickler die Entscheidung überlassen, wie er mit dem Auftreten des entsprechenden Fehlers auf Applikationsebene umgeht.

6.1.4 Serialisierbarkeit von Datenstrukturen

Neu gespawnte Tasks bekommen, wie in Kapitel 4.2.3 beschrieben, eine Kopie der geteilten Datenstrukturen übergeben. Die Übertragung dieser Datenstrukturkopien muss, im Kontext der hier entwickelten prototypischen Umsetzung von Spawn & Merge für Verteilte Systeme, auch durch Nachrichtenübertragung funktionieren. Um die Datenstrukturen in eine übertragbare Form zu bringen, wird das Konzept der *Serialisierung* [93] angewendet. Serialisierung beschreibt die Umwandlung einer komplexen Datenstruktur in eine Repräsentation, die gespeichert oder übertragen werden kann (z.B. eine Bytesequenz). Die Rückumwandlung der serialisierten Daten in die entsprechenden Datenstrukturen heißt *Deserialisierung*.

Somit ergibt sich für die Verwendung von Datenstrukturen, im hier entwickelten Prototypen für ein verteiltes Spawn & Merge Framework, die Einschränkung, dass diese nicht nur *kopierbar* (siehe Kapitel 4.2.5), sondern auch *serialisierbar* sein müssen. Beispiele für nicht serialisierbare Datenstrukturen sind plattformabhängige Objekte wie File-Handles und Sockets [93], die für eine verteilte Spawn & Merge basierte Anwendung nicht genutzt werden können. Diese Einschränkung gilt allerdings nicht nur für die hier beschriebene Umsetzung eines verteilten Spawn & Merge Frameworks, sondern generell für alle verteilte Anwendungen ohne geteilten Arbeitsspeicher.

Eine Möglichkeit, um nicht serialisierbare Datenstrukturen im Kontext eines Verteilten Systems dennoch nutzen zu können, ist die Verwendung von Remote-Referenzen (remote references) [76]. Rechenknoten, die keinen direkten Zugriff auf das nicht serialisierbare Objekt haben, können stattdessen eine Remote-Referenz auf das Objekt verwenden, welche die Zugriffe registriert und an den Rechenknoten mit dem wirklichen Objekt weiterleitet. Die Zugriffe können nun von dem Objekt verarbeitet und die Ergebnisse wieder an die Remote-Referenz zurückgegeben werden.

[2] In Kapitel 8 wird die Thematik der Fehler innerhalb der Applikationsebene noch einmal aufgegriffen.

Im Kontext von Spawn & Merge ist bei diesem Vorgehen zu beachten, dass ein Task für die Determinismusgarantie (siehe Kapitel 4.1.1) immer dieselben Eingabedaten erhalten muss. Wird ein nicht serialisierbares Objekt, das als Remote-Referenz übergeben werden soll, nicht kopiert oder wird nicht anderweitig (z.B. durch Versionierung) ein deterministischer Zustand bei der Ausführung des Tasks sichergestellt, so kann es passieren, dass das Objekt bereits wieder modifiziert wurde, bevor der Task ausgeführt wurde. Eine deterministische Ausführung wäre somit nicht mehr garantiert.

Als letzter Punkt ist zu beachten, dass bei *Containerobjekten* auch der interne Datentyp serialisierbar sein muss. So kann eine Liste beispielsweise nur dann serialisiert werden, wenn die einzelnen Elemente der Liste auch serialisierbar sind. Sowohl die Serialisierbarkeit der Datenstruktur selbst, als auch der beinhalteten Elemente kann, wie in Kapitel 4.3 beschrieben, durch die Nutzung einer *Markierungsschnittstelle* zur Compile-Zeit überprüft werden.

6.2 Umsetzung der Programmierabstraktion

In diesem Kapitel wird beschrieben, wie das Spawn & Merge Programmiermodell auf ein Verteiltes System übertragen werden kann. Dazu werden die Mechanismen beschrieben, die eine skalierende deterministische Ausführung basierend auf Nachrichtenaustausch ermöglichen. Dabei wird darauf eingegangen, welche Mechanismen auf der Frameworkebene umgesetzt werden können und welchen Einfluss diese auf den Determinismus auf Applikationsebene haben.

Um eine möglichst gleichmäßige Auslastung der verfügbaren Rechenknoten zu erreichen, müssen die Tasks zur Laufzeit auf die Rechenknoten verteilt und die Ausführung einzelner Tasks auf den Rechenknoten koordiniert werden. Dazu werden ein *NodeMaster* und ein *SchedulingMaster* beschrieben, die zentralen Komponenten der Umsetzung, die eine Skalierung der Anwendung durch ein dynamisches Scheduling ermöglichen. Anschließend wird beschrieben, wie die Synchronisationsprimitive auf ein Nachrichtenprotokoll abgebildet werden können und welche Besonderheiten es bei der Initialisierung des Frameworks auf einem Verteilten System zu beachten gilt. Abschließend werden Optimierungen vorgestellt, die im Spezialfall einer Ausführung der verteilten Anwendung auf einem einzelnen Rechenknoten eine Reduktion der Laufzeit ermöglichen.

6.2.1 Einordnung der neuen Mechanismen

Wie bereits in Kapitel 4.1.2 beschrieben, darf eine Spawn & Merge basierte Anwendung Nichtdeterminismus auf der Frameworkebene beinhalten. Im hier beschriebenen Konzept für verteiltes Spawn & Merge wird dieser Umstand genutzt, um auf der Frameworkebene Mechanismen einzuführen, die es der Anwendung erlauben mit den verfügbaren

Ressourcen zu skalieren.

Nichtdeterministisches Scheduling in der Frameworkebene

Damit eine Spawn & Merge basierte Anwendung skaliert, müssen die zur Laufzeit gespawnten Tasks auf die verfügbaren Ressourcen verteilt werden. Im hier beschriebenen Prototypen wird ein dynamisches Scheduling zur Verteilung der Tasks auf einzelne Rechenknoten genutzt, das in Kapitel 6.2.3 vorgestellt wird. Dabei wird die Auslastung der Rechenknoten in Betracht gezogen und die Möglichkeit gegeben, dass Tasks neu verteilt werden, bevor sie gestartet werden. So wird ermöglicht, dass das System reagieren kann, wenn Tasks mit unterschiedlichen Laufzeiten ausgeführt werden.

Ein statisches Scheduling, bei dem die Tasks fest einzelnen Rechenknoten zugewiesen werden, kann im Gegensatz dazu nicht auf variierende Task-Laufzeiten reagieren. Eine Scheduling-Entscheidung anhand der Laufzeit eines Tasks im Voraus ist bei Spawn & Merge nicht möglich, da die Laufzeit eines Tasks nicht bekannt ist.

Die Einordnung der Scheduling Mechanismen in die Frameworkebene sorgt zusätzlich dafür, dass Scheduling-Entscheidungen (für den Entwickler transparent) im Hintergrund gefällt werden. Der Entwickler hat keine Möglichkeit, das Scheduling zu steuern, da es sich außerhalb der von ihm implementierten Applikationsebene (und somit außerhalb seines Einflussbereichs) befindet.

Neben dem dynamischen Scheduling bietet die Frameworkebene für die Applikationsebene auch eine Abstraktion für die nichtdeterministischen internen Sperren und ausgetauschten Nachrichten, die für die Realisierung von Spawn & Merge notwendig sind. Diese Sperren und Nachrichten sind notwendig, um die Funktionalität der Synchronisationsprimitive zu realisieren, und werden in Kapitel 6.2.4 genauer beschrieben.

Einfluss auf den Determinismus auf Applikationsebene

In Kapitel 3.3 wurde definiert, dass eine Anwendung genau dann auf Applikationsebene deterministisch ist, wenn der für den Entwickler beobachtbare Ausführungspfad bei jeder Ausführung (unabhängig von der Ausführungsumgebung) dieselben Funktionsaufrufe mit identischen Berechnungen, Eingaben und Ausgaben beinhaltet. In diesem Abschnitt wird gezeigt, dass diese Anforderungen auch dann (durch die in Kapitel 4.2 vorgestellten Spawn & Merge Synchronisationsprimitive) erfüllt werden, wenn sich der Ausführungsort und die Ausführungsreihenfolge der Tasks durch das Scheduling in der Frameworkebene verändern. Die Definition bezieht sich dabei auf die Funktionseingaben (Inputs), Funktionsberechnungen und die Funktionsausgaben (Outputs) für alle Funktionsaufrufe (Tasks).

Die Funktionseingaben entsprechen den Parametern, die einem Task bei einem `Spawn`- oder `SpawnSibling`-Aufruf übergeben wurden. Im Spawn & Merge Programmiermodell

werden die Parameter für die gestarteten Tasks kopiert, um einen höheren Grad an Parallelität zu ermöglichen indem Wechselwirkungen zwischen nebenläufigen Tasks verhindert werden. Da alle Parameter beim Starten eines neuen Tasks serialisiert und in Nachrichten verpackt werden müssen, werden diese Datenstrukturkopien automatisch in dieser Nachricht konserviert (Snapshot des Zustands). So wird sichergestellt, dass auch bei einer unterschiedlichen Ausführungsreihenfolge der gespawnten Tasks die Eingabedaten bei jeder Ausführung dieselben sind, solange `Spawn` und `SpawnSibling` in deterministischer Weise aufgerufen werden.

Der Funktionskörper eines Tasks wird sequenziell in einem eigenen Thread ausgeführt und ist dementsprechend deterministisch[3]. Synchronisationen zwischen nebenläufig ausgeführten Tasks werden nur mit den von Spawn & Merge bereitgestellten Synchronisationsprimitiven durchgeführt. `Spawn` und `SpawnSibling` beeinflussen nicht die deterministische Ausführung des Funktionskörpers, da sie keine Daten innerhalb des Funktionskörpers verändern können[4]. Die deterministischen[5] `Merge`-Varianten erhalten die Ausgaben der fertiggestellten Child-Tasks in einer zufälligen Reihenfolge, die abhängig von der nichtdeterministischen Ausführungsreihenfolge der Tasks (Scheduling) und den Laufzeiten der einzelnen Tasks ist. Das deterministische Verhalten wird innerhalb der `Merge`-Primitive durch die deterministische Konfliktauflösung erreicht, die unabhängig von der Fertigstellungsreihenfolge der Child-Tasks ist (siehe Kapitel 4.2.7 und Kapitel 5.2). Die `Sync`-Primitive übergibt bei jeder Ausführung dieselben geänderten Datenstrukturkopien zurück an den Parent-Task. Das deterministische `Merge`-Ergebnis des Parent-Tasks wird wiederum an den Child-Task übergeben, wodurch ein deterministisches Verhalten bei Nutzung der `Sync`-Primitive sichergestellt ist. Aufgrund der deterministischen Funktionseingaben und der deterministischen Berechnungen im Funktionskörper ist somit auch die Funktionsausgabe (d.h. die Operationen, die auf den Datenstrukturkopien durchgeführt wurden) deterministisch.

Um zu zeigen, dass sich aus der oben beschriebenen deterministischen Ausführung eines einzelnen Tasks ergibt, dass bei jeder Ausführung dieselben Funktionsaufrufe durchgeführt werden, wird im Folgenden die Task-Hierarchie (siehe Kapitel 4.2.3), die sich bei der Ausführung der Anwendung ergibt, betrachtet. Ausgehend vom Main-Task werden, bedingt durch die deterministischen Funktionskörper aller Tasks, deterministisch die Child-Tasks gespawnt (die wiederum rekursiv Child-Tasks spawnen können). Die Blattknoten der Task-Hierarchie (d.h. die Tasks, die keine eigenen Child-Tasks haben) werden

[3]Unter der Annahme, dass keine von Haus aus nichtdeterministischen Funktionen innerhalb des Tasks aufgerufen werden (z.B. Zufallszahlengeneratoren ohne Seed, das Auslesen nichtdeterministischer Nutzereingaben oder die Nutzung der nichtdeterministischen `Merge`-Varianten).

[4]Einzige Ausnahme bildet das zurückgegebene `TaskHandle`, das aber deterministisch bei jeder Ausführung für denselben Task steht.

[5]Die von Haus aus nichtdeterministischen `Merge`-Varianten, die den zuerst fertiggestellten Task mergen, werden hier nicht betrachtet.

deterministisch sequenziell ausgeführt und die Ergebnisse werden an den zugehörigen Parent-Task übergeben. Dadurch, dass der `Merge`-Aufruf beim Parent-Task deterministisch ist, solange die Ergebnisse der Child-Tasks deterministisch sind, wird der Parent-Task (sobald er alle eigenen Child-Tasks zusammengeführt hat) zu einem deterministisch ausgeführten Blattknoten der Task-Hierarchie. Somit propagiert sich die deterministische Ausführung der Tasks aufwärts in der Task-Hierarchie, sodass der Schluss gezogen werden kann, dass sich alle Funktionsaufrufe (Tasks) bei jeder Ausführung der Anwendung gleich verhalten werden und die Anwendung auf der Applikationsebene deterministisch ist.

6.2.2 NodeMaster

Auf jedem MPI-Knoten müssen verschiedene Aufgaben durchgeführt werden. Zum einen muss die Kommunikation zwischen den MPI-Knoten und dabei insbesondere die Abbildung der Spawn & Merge Synchronisationsprimitive auf einzelne Nachrichten gehandhabt werden. Zum anderen muss ein MPI-Knoten in der Lage sein, die ihm zugewiesenen gespawnten Tasks anzunehmen und diese zu starten, wenn er genügend freie Ressourcen zur Verfügung hat.

Zu diesem Zweck wird hier die *NodeMaster*-Komponente eingeführt, die auf jedem MPI-Knoten instanziiert wird. Der NodeMaster zeichnet sich durch einen geringen Berechnungsmehraufwand aus. Er stellt eine Schnittstelle für den Zugriff auf MPI bereit und verwaltet die zugewiesenen Tasks und verfügbaren Ressourcen des MPI-Knotens. Für den Entwickler ist der NodeMaster transparent.

Funktionsweise

Auf jedem MPI-Knoten befindet sich ein NodeMaster. Die Einordnung des NodeMasters in die Architekturdarstellung aus Kapitel 6.1.1 ist in Abbildung 6.4 dargestellt. Zu sehen ist hier der MPI-Knoten 0, der aktuell einen Task T_1 ausführt, der wiederum zwei Child-Tasks besitzt, die auf anderen MPI-Knoten ausgeführt werden.

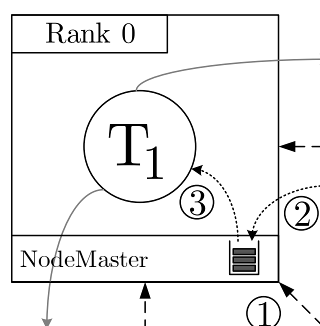


Abbildung 6.4: NodeMaster im Kontext eines MPI-Knotens.

Der NodeMaster nimmt alle eingehenden MPI-Nachrichten entgegen und bietet der Frameworkebene (z.B. für das Scheduling) eine Schnittstelle für das Senden von MPI-Nachrichten an (die Kommunikation mit anderen MPI-Knoten ist in der Abbildung mit (1) gekennzeichnet). Somit übernimmt der NodeMaster die gesamte Kommunikation zwischen einzelnen MPI-Knoten, inklusive dem Annehmen von Nachrichten, die neu gespawnte Tasks beinhalten, sowie die Übertragung der Ergebnisse eines Tasks zurück an den MPI-Knoten, der den zugehörigen Parent-Task ausführt.

Eingehende Nachrichten mit neu gestarteten Tasks, werden im folgenden als *Workitems* bezeichnet. Ein Workitem kapselt alle Informationen, die für das nebenläufige Ausführen eines Tasks benötigt werden. Es beinhaltet eine ID für die zu startende Funktion, eine Referenz auf den Parent-Task[6] (d.h. den MPI-Knoten auf dem der Parent-Task ausgeführt wird, sowie die ID des Parent-Tasks) und die serialisierten Funktionsparameter. Empfängt der NodeMaster ein solches Workitem von einem anderen MPI-Knoten, so speichert er dieses in einer Liste von wartenden Workitems für die spätere Ausführung zwischen (in der Abbildung mit (2) gekennzeichnet). Diese Liste wird als *WorkQueue* bezeichnet.

Im Rahmen der Verwaltung zugewiesener Tasks entscheidet der NodeMaster, wann er ein Workitem aus der WorkQueue herausnehmen und ausführen möchte (in der Abbildung mit (3) gekennzeichnet). Die Mechanismen, die für die Entscheidungsfindung genutzt werden, werden im folgenden Abschnitt beschrieben.

Ressourcenmanagement

Eine der Aufgaben des NodeMasters ist die Verwaltung der Tasks in Anbetracht verfügbarer Ressourcen auf dem eigenen MPI-Knoten. Dabei stellt der NodeMaster sicher, dass immer eine konfigurierbare Anzahl von n Tasks parallel ausgeführt wird, sofern genügend zugewiesene Tasks in der WorkQueue vorliegen[7]. Diese Anzahl n soll dabei so gewählt werden, dass nach Möglichkeit eine optimale Last auf jedem MPI-Knoten erreicht wird, die den Prozessor (zusammen mit der Ausführung des NodeMasters und des MPI-Prozesses) möglichst zu 100% auslastet.

Die Einschränkung, dass nur eine begrenzte Anzahl an Tasks parallel ausgeführt wird, ist darin begründet, dass einmal gestartete Tasks nicht mehr (ohne großen Mehraufwand) auf einen anderen MPI-Knoten migriert werden können. Die Migration von bereits zugewiesenen (aber noch nicht gestarteten) Tasks soll allerdings explizit möglich sein, um im Rahmen des dynamischen Schedulings die Umsetzung eines Workstealing-Algorithmus zu ermöglichen (siehe Kapitel 6.2.3).

[6] Da es keine günstige Möglichkeit gibt, um jedem neu gespawnten Task eine einzigartige ID zuzuweisen, sind die Task-IDs nur innerhalb eines MPI-Knotens eindeutig. Für eine eindeutige Identifizierung eines Tasks im gesamten System wird daher das Tupel bestehend aus dem MPI-Knoten und der TaskID genutzt.

[7] Diese Anzahl n kann dabei so gewählt werden, dass sie optimal für die gewählte Ausführungsumgebung ist. Standardmäßig wird n auf die Anzahl verfügbarer Rechenkerne pro MPI-Knoten gesetzt.

Es gibt verschiedene Situationen, in denen ein Task blockieren und eine gewisse Zeit lang keine Berechnungen durchführen können. Dies ist beispielsweise der Fall, wenn ein Task bei einem `Merge`-Aufruf auf die Fertigstellung seiner Child-Tasks, oder ein Child-Task auf die Fertigstellung eines `Sync`-Aufrufs wartet. In diesen Fällen signalisieren blockierende Tasks dem NodeMaster, dass sie sich von nun an im Ruhemodus befinden, sodass der NodeMaster weiß, dass er einen anderen neuen Task starten kann, um keine wertvolle Rechenzeit zu verlieren (*signal-sleeping*). Wichtig ist hier anzumerken, dass diese Funktionalität nur dafür sorgt, dass keine nennenswerten Leerlaufzeiten auf dem Prozessor entstehen. Auf die deterministische Ausführung der Anwendung hat dies keinen Einfluss. Sobald für einen Task das Ereignis, auf das er gewartet hat, eingetreten ist, signalisiert er dem NodeMaster, dass er gerne seine Ausführung fortsetzen würde (*wakeup-wish*).

Bevor der Task seine Ausführung fortsetzen darf, muss er darauf warten, dass er vom NodeMaster dafür die Erlaubnis bekommt. Andernfalls würde die Grenze von n Tasks durch aufgewachte Tasks missachtet und überschritten. Die Erlaubnis vergibt der NodeMaster, sobald die Anzahl laufender Tasks wieder unter die Begrenzung n fällt. Das Fortsetzen bereits laufender Tasks hat dabei eine höhere Priorität als das Starten neuer Workitems.

NodeMaster Hinweise auf Applikationsebene

Wie bereits beschrieben kann das Framework bei Verwendung der Spawn & Merge Synchronisationsprimitive automatisch den NodeMaster darüber informieren, ob ein Task in den Ruhemodus versetzt wird (d.h. auf den Eintritt eines Ereignisses wartet), oder ob er nach Eintritt eines Ereignisses die Ausführung fortsetzen möchte. Diese Informationen können dazu genutzt werden, um Prozessorleerlauf zu verhindern.

Werden blockierende Aufrufe in der Applikationsebene getätigt, so kann dies nicht automatisch an den NodeMaster gemeldet werden. Im schlimmsten Fall kann es dazu kommen, dass alle n Tasks auf einem MPI-Knoten auf Applikationsebene auf ein Ereignis warten (z.B. auf eine Nutzereingabe) und somit kein Kern auf diesen Rechenknoten arbeitet. Um diesen Performanceverlust verhindern zu können, werden in diesem Abschnitt die Befehle `BlockingCallStart` und `BlockingCallEnd` eingeführt, die ein Entwickler nutzen kann, um den NodeMaster auf ruhende und aufwachende Tasks hinzuweisen.

`BlockingCallStart` dient zwei Zielen. Zum einen signalisiert der Aufruf dem NodeMaster, dass der aufrufende Task einen blockierenden Funktionsaufruf absetzen wird. Der NodeMaster weiß dementsprechend anschließend, dass er wieder mehr freie Ressourcen zur Verfügung hat und markiert den Task als schlafend. Zum anderen wird eine Wartebedingung vorbereitet, die dafür sorgt, dass ein `BlockingCallEnd`-Aufruf solange blockiert, bis der NodeMaster dem Task die Fortsetzung der Ausführung erlaubt. Dementsprechend informiert der `BlockingCallEnd`-Aufruf den NodeMaster über den Wunsch die Ausführung fortzusetzen und wartet auf die entsprechende Erlaubnis in Form der Auflösung der

Wartebedingung. Die abstrakten Schnittstellendefinitionen von `BlockingCallStart` und `BlockingCallEnd` sind in den Listings 6.1 und 6.2 definiert. Beide Befehle dienen als reine Hinweise an den NodeMaster und haben dementsprechend keinen Rückgabewert und erwarten keine Parameter.

Listing 6.1 Definition der *BlockingCallStart* Schnittstelle.

```
void BlockingCallStart()
```

Listing 6.2 Definition der *BlockingCallEnd* Schnittstelle.

```
void BlockingCallEnd()
```

Listing 6.3 zeigt, wie die beiden Befehle um einen blockierenden Aufruf herum genutzt werden können. Hier wird der NodeMaster in Zeile 3 darüber informiert, dass der Task einen blockierenden Aufruf startet, der in Zeile 4 durchgeführt wird. Anschließend blockiert der `BlockingCallEnd`-Aufruf (Zeile 5) solange, bis der NodeMaster die weitere Ausführung erlaubt (d.h. bis freie Ressourcen verfügbar sind).

Listing 6.3 Hinweis auf einen blockierenden Funktionsaufruf.

```
1: [...] // Task execution
2:
3: BlockingCallStart();
4: getline(std::cin, readStr);
5: BlockingCallEnd();
6:
7: [...] // Task continues execution
```

Ein verbleibendes Problem ist, dass die Verwendung von `BlockingCallEnd` nicht erzwungen werden kann. Ohne die Verwendung würde der Task nach Beendigung des blockierenden Aufrufs weiterlaufen, unbemerkt vom NodeMaster (der weiterhin denkt, dass der Task ruht) und ungeachtet der verfügbaren Ressourcen auf dem NodeMaster. Dies kann bei der weiteren Ausführung der Anwendung zu (deterministischem) Fehlverhalten führen.

Deadlockvermeidung

Um zu verhindern, dass es innerhalb der Frameworkebene und insbesondere im NodeMaster zu Verklemmungen (Deadlocks) kommen kann, kommunizieren die einzelnen Komponenten des Frameworks auch innerhalb des MPI-Knotens nur über asynchronen Nachrichtenaustausch miteinander. Der NodeMaster wartet blockierend (um keine Re-

chenzeit zu verbrauchen) darauf, dass ein Befehl in seine *CommandQueue* geschrieben wird.

Sobald ein Befehl in der CommandQueue vorliegt, wird dieser vom NodeMaster ausgelesen und abgearbeitet. Dieser Befehl kann beispielsweise eine empfangene Nachricht von der MPI-Komponente sein, oder ein Signal von einem der aktuell ausgeführten Tasks (z.B. dass eine Synchronisationsprimitive aufgerufen oder der Task fertiggestellt wurde). Nachdem der NodeMaster den aktuellen Befehl abgearbeitet hat, wird geprüft, ob sich durch den Befehl etwas an den verfügbaren Ressourcen geändert hat. Dabei wird insbesondere überprüft, ob ein Task fertiggestellt oder in den Ruhemodus versetzt wurde und somit freie Ressourcen verfügbar sind, um einen anderen Task zu starten. Anschließend bearbeitet der NodeMaster den nächsten Befehl oder blockiert beim Auslesen der CommandQueue (falls diese aktuell leer ist).

6.2.3 SchedulingMaster

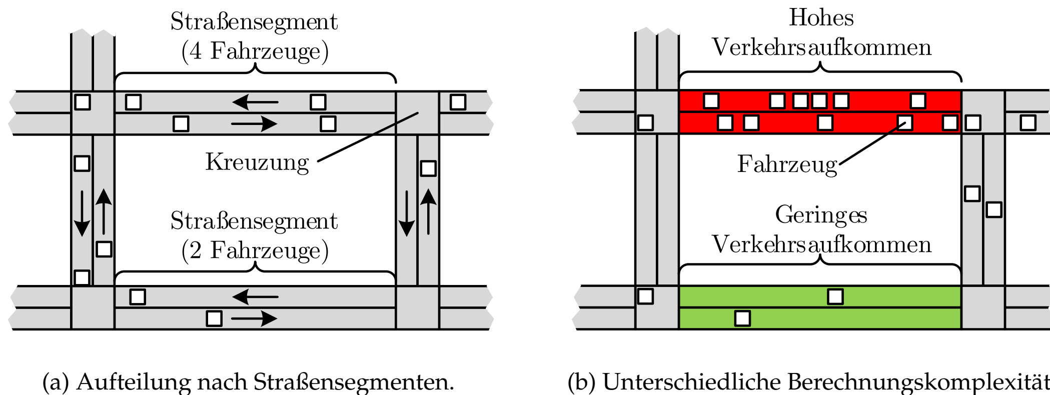
Für die dynamische Zuweisung von gespawnten Tasks zu MPI-Knoten wird ein zentraler *SchedulingMaster* eingeführt. Ein *dynamisches Scheduling*[8] im Rahmen von verteiltem Spawn & Merge ist notwendig, da ein statisches Scheduling nicht dazu in der Lage ist, zur Laufzeit auf eine sich ändernde Berechnungskomplexität der gespawnten Tasks zu reagieren.

Zur Veranschaulichung der Problematik wird hier noch einmal das Verkehrssimulationsbeispiel, das in Kapitel 4.2.2 beschrieben wurde, herangezogen. Abbildung 6.5a zeigt die Aufteilung der gesamten Simulation in einzelne Straßensegmente, die von unterschiedlichen Tasks simuliert werden. Diese Straßensegmente unterscheiden sich in ihrer Berechnungskomplexität, in Abhängigkeit von der Anzahl an Fahrzeugen auf dem simulierten Straßensegment. Ein Segment, auf dem ein Stau ist, ist komplexer zu simulieren als ein Segment mit wenigen Fahrzeugen (siehe Abbildung 6.5b). Diese Berechnungskomplexität kann sich dabei auf benachbarte Segmente verschieben, wenn sich die Fahrzeuge im Laufe der Simulation bewegen.

Ein *statisches Scheduling*[9] würde hier bedeuten, dass ein Task immer auf demselben MPI-Knoten ausgeführt wird, auch wenn die diesem MPI-Knoten zugewiesenen Tasks zusammen nur eine geringe Berechnungskomplexität aufweisen, während andere MPI-Knoten mit derselben Anzahl auszuführender Tasks insgesamt eine sehr hohe Berechnungskomplexität zu bewältigen haben. Das Scheduling der Tasks hat dabei nur einen Einfluss auf die Laufzeit der Anwendung, nicht aber auf den Determinismus auf Applikationsebene.

[8] Dynamisches Scheduling bedeutet, dass die Scheduling-Entscheidungen zur Programmlaufzeit getroffen werden [92].

[9] Statisches Scheduling bedeutet, dass die Scheduling-Entscheidungen bereits vor der Ausführung der Anwendung getroffen werden [92].



(a) Aufteilung nach Straßensegmenten.

(b) Unterschiedliche Berechnungskomplexität.

Abbildung 6.5: Berechnungskomplexität der Verkehrssimulation.

Dynamisches Scheduling

Das in diesem Kapitel beschriebene dynamische Scheduling setzt an zwei Stellen an, um eine möglichst gleichmäßige Verteilung der Berechnungskomplexität zu erreichen. Zum einen werden Tasks anhand der aktuellen Auslastung vorhandener MPI-Knoten verteilt. Zum anderen kann ein NodeMaster, der freie Ressourcen aber keine weiteren Workitems in seiner WorkQueue hat, einem anderen ausgelasteten MPI-Knoten Workitems abnehmen, um die Last gleichmäßiger im System zu verteilen. Hierbei ist zu beachten, dass einem Task seine Berechnungskomplexität nicht *a priori* anzusehen ist, sodass es nicht möglich ist ein statisches Scheduling durchzuführen, das die Komplexität eines Tasks einbezieht. Ein Scheduling-Algorithmus kann dementsprechend nur auf einen aktuellen Systemzustand *reagieren* (dynamisches Scheduling).

Eine Anwendung mit dynamischem Scheduling ist in der Regel schneller als eine Anwendung mit statischem Scheduling. Das liegt daran, dass ein statisches Scheduling keine Laufzeitinformationen mit in die Scheduling-Entscheidungen einbeziehen kann. Es gibt allerdings auch seltene Umstände, in denen das statische Scheduling schneller ist als ein dynamisches Scheduling.

Einer dieser Umstände tritt ein, wenn es beispielsweise nur einen einzelnen Rechenknoten zum Ausführen der gespawnten Tasks gibt. In diesem Falle gibt es für dynamische und statische Scheduler nur eine Möglichkeit Tasks auf Rechenknoten zu verteilen. Da das dynamische Scheduling mit einem Berechnungsmehraufwand verbunden ist, der in diesem Fall nicht durch ein besseres Scheduling aufgeholt werden kann, wird die Anwendung mit dynamischem Scheduling hier langsamer ausgeführt werden als die Anwendung mit dem statischen Scheduling. In Kapitel 6.2.6 werden Optimierungen für die Ausführung einer verteilten Spawn & Merge basierten Anwendung beschrieben, die unter anderem auch das Deaktivieren des Schedulers beinhalten, um dieses Problem zu beheben.

Ein anderer Umstand tritt ein, wenn der statische Scheduler durch Zufall einen perfekten Zeitplan für die Tasks trifft. Der dynamische Scheduler bräuchte für die Errechnung dieses Zeitplans mehr Zeit als der statische Scheduler, oder könnte diesen perfekten Zeitplan gegebenenfalls gar nicht erreichen. Das Eintreten dieses Umstandes ist allerdings in einer durchschnittlichen Anwendung ausreichend unwahrscheinlich, sodass ein dynamisches Scheduling im Durchschnitt eine bessere Performance für die Anwendung erreicht.

Einordnung des SchedulingMasters in das Systemmodell

Der SchedulingMaster wird auf einem der MPI-Knoten ausgeführt[10] und dient für alle NodeMaster als Anlaufstelle für die Entscheidung, zu welchem MPI-Knoten ein neu gespawnter Task gesendet werden soll. Die Abbildung 6.6 stellt das um den Scheduling-Master erweiterte Systemmodell dar. In der Abbildung möchte der Task T_2 einen neuen Child-Task T_4 spawnen und teilt dies dem NodeMaster auf seinem MPI-Knoten mit. Der NodeMaster fragt beim SchedulingMaster an, zu welchem MPI-Knoten der neue Task gesendet werden soll (1). Anschließend wird das Workitem für den Task T_4 in die WorkQueue des entsprechenden MPI-Knotens eingefügt (2).

Im Folgenden wird beschrieben, wie das dynamische Scheduling im Rahmen des hier entwickelten Prototyps durchgeführt und wie die Scheduling-Entscheidungen getroffen werden. Anschließend wird ein Workstealing-Algorithmus beschrieben, der es der Anwendung ermöglicht, auf eine ungünstige Task-Verteilung zu reagieren, indem bereits zugewiesene Tasks neu verteilt werden.

Scheduling-Ziele

Der im SchedulingMaster angesiedelte Scheduling-Algorithmus soll bestimmte Ziele verfolgen, die im Folgenden beschrieben werden. Zur Kategorisierung werden hier die von Tanenbaum beschriebenen möglichen Scheduling-Ziele herangezogen [92].

Die Prioritäten des Scheduling-Algorithmus sollen darauf liegen, nach Möglichkeit alle MPI-Knoten beschäftigt zu halten (*Balance*) und somit den Durchsatz an fertiggestellten Tasks pro Zeiteinheit hochzuhalten (*Throughput*). Innerhalb des NodeMasters wird des Weiteren dafür Sorge getragen, dass der Prozessor nach Möglichkeit durchgängig ausgelastet ist (*CPU utilization*). Dabei wird automatisch darauf geachtet, dass jeder Task einen möglichst gleichen Anteil an der Prozessorzeit bekommt (*Fairness*), da immer genau ein Task pro Prozessorkern ausgeführt wird.

Eine Minimierung der Zeitspanne zwischen dem Starten eines Tasks und dessen Fertigstellung (*Turnaround time*) gehört nicht zu den Zielen des Scheduling-Algorithmus. Ebenso

[10] Der SchedulingMaster wird standardmäßig auf dem MPI-Knoten mit der ID 0 ausgeführt. Die Verwendung einer höheren ID ist nicht ratsam, da Probleme auftreten, sobald die Anwendung auf einem System mit wenigen MPI-Knoten ausgeführt werden, sodass kein Knoten mit der gewählten ID vorhanden ist.

Abbildung 6.6: Einordnung des SchedulingMasters in das Systemmodell.

gibt es keine Verhaltensregeln, die durch das Scheduling eingehalten werden müssen (*Policy enforcement*).

Funktionsweise des Scheduling-Algorithmus

Der in diesem Abschnitt beschriebene Scheduling-Algorithmus dient als Beispiel dafür, dass ein dynamisches Scheduling auf der Frameworkebene von verteiltem Spawn & Merge genutzt werden kann, ohne den Determinismus auf Applikationsebene zu beeinflussen. Der implementierte Scheduling-Algorithmus zeigt dabei, dass das dynamische Scheduling eine Verbesserung der Laufzeit erreicht (siehe Kapitel 7.3). Eine Optimierung des verwendeten Scheduling-Algorithmus könnte möglicherweise weitere Verbesserungen der Laufzeit erreichen, wird allerdings im Rahmen dieser Arbeit nicht betrachtet.

Da der Berechnungsaufwand eines Tasks (wie bereits beschrieben) a priori unbekannt ist, werden alle Tasks vom Scheduling-Algorithmus als gleich aufwändig behandelt. Der SchedulingMaster nutzt daher als Entscheidungsgrundlage nur eine Schätzung, wie viele Tasks den einzelnen MPI-Knoten gerade zur Ausführung zugewiesen sind und dort ausgeführt werden[11] und versucht die Verteilung gleichmäßig zu halten (*Work-Approximation*).

[11]Die Work-Approximation beinhaltet somit gestartete (d.h. laufende und schlafende Tasks) und bisher

Die Work-Approximation beinhaltet dazu für jeden MPI-Knoten die zuletzt bekannte Menge zugewiesener und gestarteter Tasks.

Empfängt der SchedulingMaster eine `GetSpawnDestination`-Nachricht von einem NodeMaster, so prüft der SchedulingMaster, ob einer der MPI-Knoten aktuell noch nicht voll ausgelastet ist (d.h. gestartete Tasks < festgelegtes n). Sind alle MPI-Knoten ausgelastet, sucht der SchedulingMaster den MPI-Knoten, der aktuell die geringste Anzahl an Tasks zugewiesen bekommen hat. Die ID des entsprechenden MPI-Knotens wird an den NodeMaster zurückgegeben und die Anzahl zugewiesener Tasks in der Work-Approximation für diesen MPI-Knoten um eins erhöht. Anschließend übernimmt der NodeMaster die Übertragung des Workitems an den Zielknoten für den neu gespawnten Task.

Perfomancebetrachtung für den SchedulingMaster

Um zu verhindern, dass der SchedulingMaster die Systemperformance stark beeinträchtigt, wurden bei der Konzeption zwei Ziele verfolgt. Zum einen sollte der Berechnungsaufwand beim SchedulingMaster gering gehalten werden, um die Ressourcen des MPI-Knotens zu schonen, auf dem der SchedulingMaster ausgeführt wird. Dies wurde durch die einfach zu berechnende und zu verwaltende Work-Approximation erreicht, die einen Kompromiss zwischen aufwändiger Berechnung und akkurater Abbildung des Systemzustandes darstellt. Zum anderen sollte die Kommunikation der NodeMaster mit dem SchedulingMaster minimal gehalten werden, damit der SchedulingMaster nicht zu einem Flaschenhals im Rahmen des Nachrichtenaustausches innerhalb des Systems wird.

Die Anzahl zugewiesener Tasks pro MPI-Knoten ist dem SchedulingMaster bekannt, da er diese Zuweisung vorgenommen hat. Um die Kommunikation zu reduzieren, wird der SchedulingMaster zur Aktualisierung der Work-Approximation nur über fertiggestellte Tasks informiert. Dazu sendet der NodeMaster dedizierte `WorkFinished`-Nachrichten an den SchedulingMaster.

Der Kenntnisstand des SchedulingMasters bezüglich der Anzahl der Tasks, die tatsächlich gerade auf einem MPI-Knoten ausgeführt werden oder schlafen (blockieren), ist nicht zu jeder Zeit aktuell. Hier werden Aktualisierungen nur dann an den SchedulingMaster übertragen, wenn bereits aus einem anderen Grund eine Kommunikation mit dem SchedulingMaster notwendig ist. So wird beispielsweise bei Scheduling-Anfragen oder Workstealing-Anfragen (siehe nächster Abschnitt) der aktuelle Zustand des eigenen MPI-Knotens (d.h. zugewiesene und gestartete Tasks) vom NodeMaster *Huckepack (piggyback)* mit an den SchedulingMaster gesendet (siehe z.B. *Piggyback*-Updates in [18]). Das reduziert die notwendige Kommunikation mit dem SchedulingMaster, beeinträchtigt aber gleichzeitig

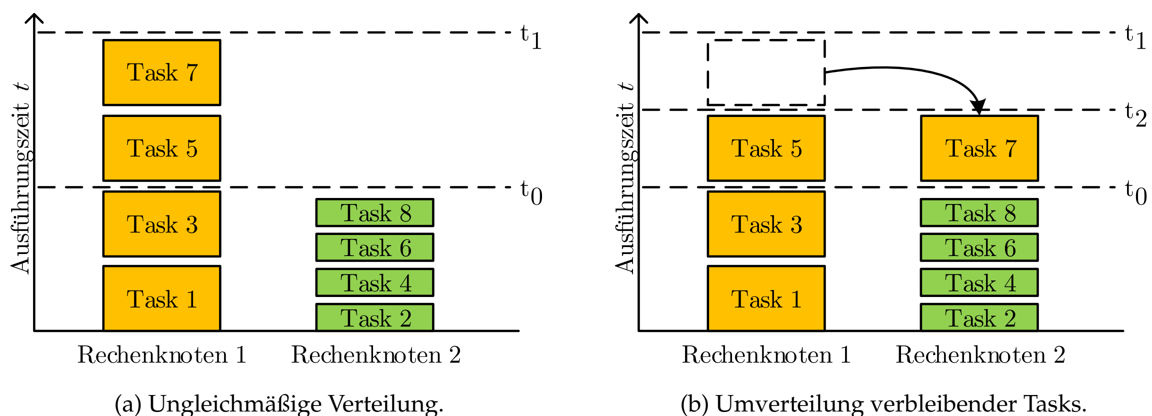
ungestartete Tasks. Die CPU-Auslastung wird nicht mit einbezogen, da das System zu jeder Zeit auf eine Auslastung von 100% abzielt und diese durch den NodeMaster und das Starten einer festgelegten Anzahl von n Tasks bereits erreicht wird.

die Genauigkeit der Work-Approximation und somit die Qualität der Scheduling-Entscheidungen. Besonders in der finalen Phase der Anwendungsausführung kann es dazu kommen, dass keine neuen Anfragen mehr an den SchedulingMaster gesendet werden, und somit der über Piggyback-Updates aktualisierte Teil der Work-Approximation nicht weiter aktualisiert werden kann.

Workstealing

Da die Berechnungskomplexität einzelner Tasks nicht in das Scheduling mit einbezogen werden kann, ist es möglich, dass mehrere aufwändig zu berechnende Tasks demselben MPI-Knoten zugewiesen werden, während ein anderer MPI-Knoten mehrere, sehr einfach zu berechnende Tasks zugewiesen bekommt. Ein Beispiel dafür ist in Abbildung 6.7a zu sehen, wo der Rechenknoten 2 ab dem Zeitpunkt t_0 die zugewiesenen Tasks fertig abgearbeitet hat, während der Rechenknoten 1 noch weitere Tasks zur Ausführung vorliegen hat.

Um diese ungleichmäßige Verteilung zu beheben, wird ein Workstealing-Algorithmus eingeführt, der eine Umverteilung bereits zugewiesener (aber noch nicht gestarteter) Tasks ermöglicht. Der Ansatz ist in Abbildung 6.7b dargestellt, wo der Rechenknoten 2, sobald er keine eigene Arbeit mehr vorliegen hat, einen noch nicht gestarteten Task von Rechenknoten 1 stiehlt. So wird die Ausführungszeit der Anwendung von t_2 auf t_1 reduziert. Das Workstealing ermöglicht es dementsprechend einer Anwendung mit ungünstigen Scheduling-Entscheidungen umzugehen, indem Tasks (falls notwendig) automatisch umverteilt werden können.



(a) Ungleichmäßige Verteilung.

(b) Umverteilung verbleibender Tasks.

Abbildung 6.7: Workstealing bei ungleichmäßig verteilter Berechnungskomplexität.

Funktionsweise des Workstealings

Sobald die Auslastung eines MPI-Knotens unter einen vorher festgelegten Grenzwert g fällt, startet der NodeMaster den Workstealing Vorgang. Der Grenzwert g wurde im hier konzipierten Prototypen so gewählt, dass er unterschritten wurde, sobald ein MPI-Knoten freie Ressourcen für mindestens zwei weitere Tasks hat, da in diesem Falle keine Workitems mehr in der eigenen WorkQueue vorliegen.

Um den Workstealing Vorgang zu starten, sendet der NodeMaster dem SchedulingMaster eine `GetWorkstealingDestination`-Nachricht. Diese Nachricht repräsentiert die Bitte des NodeMasters, dass der SchedulingMaster ihm mitteilen soll, auf welchem MPI-Knoten noch viele Tasks vorliegen, die möglicherweise noch unbearbeitet sind. Um zu entscheiden, ob von einem MPI-Knoten Tasks gestohlen werden können, prüft der SchedulingMaster die lokal gespeicherte Work-Approximation. Dabei sucht er nach dem MPI-Knoten mit der höchsten Anzahl an zugewiesenen Tasks (`Assigned`). Falls die Anzahl der zugewiesenen Tasks größer als 0 ist, dann handelt es sich um einen MPI-Knoten, der für Workstealing in Frage kommt und der SchedulingMaster antwortet dem NodeMaster mit der ID des ausgelasteten MPI-Knotens. Der NodeMaster kann nun direkt bei dem entsprechenden MPI-Knoten ein Workitem anfragen, um diesen zu entlasten.

Falls der SchedulingMaster keinen geeigneten MPI-Knoten in der Work-Approximation findet, wird die Anfrage des NodeMasters verworfen. Eine negative Antwort auf eine Workstealing-Anfrage wird nicht versendet, um die Kommunikation mit dem SchedulingMaster gering zu halten. Bekommt ein NodeMaster keine Antwort auf seine Workstealing-Anfrage, so geht er davon aus, dass es im System aktuell keine Workitems gibt, die von ihm übernommen werden könnten. Der NodeMaster sendet des Weiteren keine neuen Workstealing-Anfragen[12].

Um sicherzustellen, dass keine Workstealing-Anfragen versendet werden während die Anwendung sich noch in der Startphase befindet, wird der Workstealing-Mechanismus auf einem MPI-Knoten erst dann aktiviert, wenn dieser Knoten das erste Mal selbst vollständig ausgelastet ist.

Verteilungstransparenz

Dadurch, dass der SchedulingMaster in der Frameworkebene angesiedelt ist, wird ermöglicht, dass der Entwickler auf der Applikationsebene keine Scheduling- und Workstealing-Entscheidungen fällen braucht und die Verteilung der Tasks „transparent“ wird. Somit wird durch das dynamische Scheduling eine Teilmenge der von Tanenbaum definierten *Verteilungstransparenz* [93] erreicht: *Location-* und *Migration-Transparency* [93]. Loca-

[12] Dieses Verhalten wird erst dann wieder auf den Ursprungszustand zurückgesetzt, sobald der NodeMaster auf normalem Wege ein neues Workitem erhalten hat. Erst dann geht er davon aus, dass sich die Auslastung im System wieder grundlegend geändert hat.

tion-Transparency gibt dabei an, dass nicht sichtbar ist, wo sich eine Ressource im Verteilten System befindet. Im Kontext von verteiltem Spawn & Merge ist für den Entwickler nicht ersichtlich, auf welchem MPI-Knoten ein Task ausgeführt wird. Migration-Transparency gibt an, dass das Verschieben einer Ressource an einen anderen Ort im Verteilten System nicht sichtbar ist. Im Kontext von verteiltem Spawn & Merge wird das Verschieben von bereits zugewiesenen Tasks im Rahmen des Workstealings auf andere MPI-Knoten transparent durchgeführt.

Das Spawn & Merge Programmiermodell ermöglicht zusätzlich *Access-Transparency*, da Ressourcen immer als Kopie beim eigenen Task vorliegen. *Relocation-* und *Replication-Transparency* sind nicht anwendbar auf verteiltes Spawn & Merge, da laufende Tasks und deren Datenstrukturkopien nicht verschoben und auch nicht repliziert werden [93]. *Concurrent-Transparency* ist ebenfalls nicht anwendbar, da Tasks (z.B. durch mehrfache `Sync`-Aufrufe) Änderungen anderer Tasks an übergebenen Datenstrukturen sehen können [93]. Des Weiteren wird *Failure-Transparency* [93] von Spawn & Merge nicht erreicht, da Fehler in den meisten Fällen nicht automatisch behoben werden können und zur Terminierung der Anwendung führen (siehe Kapitel 4.3).

6.2.4 Nachrichtenprotokoll

In diesem Kapitel wird anhand der Synchronisationsprimitive `Spawn`, `Merge`, `Sync` und `SpawnSibling` gezeigt, wie deren Funktionen auf ein Nachrichtenprotokoll abgebildet werden[13].

Spawn & Merge Ablauf

In Abbildung 6.8 wird dargestellt, wie die `Spawn`-Primitive und die `Merge`-Primitive auf ein Nachrichtenprotokoll abgebildet wurde. Hier startet ein Parent-Task auf dem MPI-Knoten 1 einen neuen Child-Task und merged diesen anschließend wieder. Beim Aufruf der `Spawn`-Primitive meldet der Task dem zugehörigen NodeMaster, dass dieser ein Workitem versenden soll (`SendWork`-Nachricht). Der NodeMaster fordert beim SchedulingMaster mit einer `GetSpawnDestination`-Anfrage ein Ziel für die neu gespawnte Arbeit an und erhält eine Antwort auf seine Anfrage in Form einer `SpawnDestination`-Nachricht. Sobald der NodeMaster dieses Ziel (in Form eines MPI-Knotens mit geringer Auslastung) erhalten hat, sendet er das Workitem (die `SendWork`-Nachricht) weiter an den NodeMaster des entsprechenden MPI-Knotens. Dieser NodeMaster wird nun das empfangene Workitem, das den Child-Task beinhaltet, ausführen, sobald genügend freie Ressourcen vorhanden sind.

Nach der Ausführung des Child-Tasks werden dessen Ergebnisse unter Verwendung einer `SendData`-Nachricht, über den eigenen NodeMaster und den NodeMaster auf dem

[13] Der Aufruf einer `Abort`-Primitive geschieht analog zum Senden einer `Spawn`-Nachricht.

MPI-Knoten des Parent-Tasks, an den Parent-Task zurück übertragen. Die Zuordnung der empfangenen Daten zu dem gespawnten Task auf der Seite des Parent-Tasks geschieht über die Header-Informationen des Workitems, mit dem der Child-Task gestartet wurde (siehe Abbildung 6.3). Beim Parent-Task wird dabei mit dem Erhalt dieser Nachricht die Wartebedingung der `Merge`-Primitive aufgelöst.

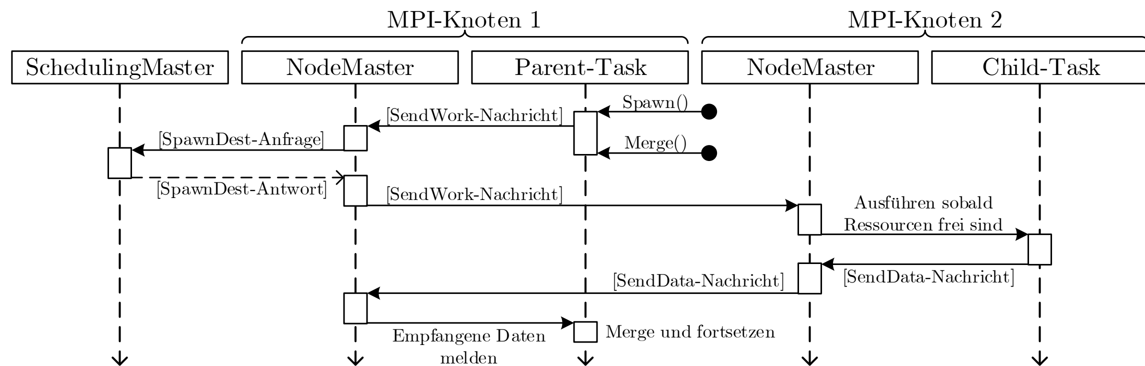


Abbildung 6.8: Nachrichtenfluss für *Spawn* und *Merge*.

Sync Ablauf

Der Nachrichtenfluss für die Durchführung eines `Sync`-Aufrufs ist im Vergleich zu den `Spawn`- und `Merge`-Primitiven komplexer, da nach dem `Merge` auf der Seite des Parent-Tasks die Ergebnisse zurück an den Child-Task gesendet werden müssen. Der Nachrichtenfluss eines vollständigen `Sync`-Aufrufs ist in Abbildung 6.9 dargestellt.

Der Child-Task, der `Sync` aufgerufen hat, sendet seine Zwischenergebnisse in einer `SendData`-Nachricht (über den eigenen NodeMaster und den NodeMaster des MPI-Knotens des Parent-Tasks) an seinen Parent-Task. Der Parent-Task nimmt diese Nachricht im Rahmen eines `Merge`-Aufrufs entgegen. Im Anschluss an die Zusammenführung werden die Ergebnisse des `Merge`-Aufrufs in einer `SendData`-Nachricht über die entsprechenden NodeMaster an den Child-Task gesendet. Dort werden bei Erhalt der Nachricht die `Merge`-Ergebnisse übernommen und die Wartebedingung für `Sync` erfüllt, sodass die Ausführung des Tasks fortgesetzt wird.

SpawnSibling Ablauf

Das Protokoll für einen `SpawnSibling`-Aufruf unterscheidet sich von den vorangegangenen Protokollen dadurch, dass vor dem Starten des Sibling-Tasks Informationen zwischen dem Child-Task (der `SpawnSibling` aufgerufen hat) und dem Parent-Task ausgehandelt werden müssen (`Notify`-Nachrichten). Unter diese Informationen fällt unter anderem die

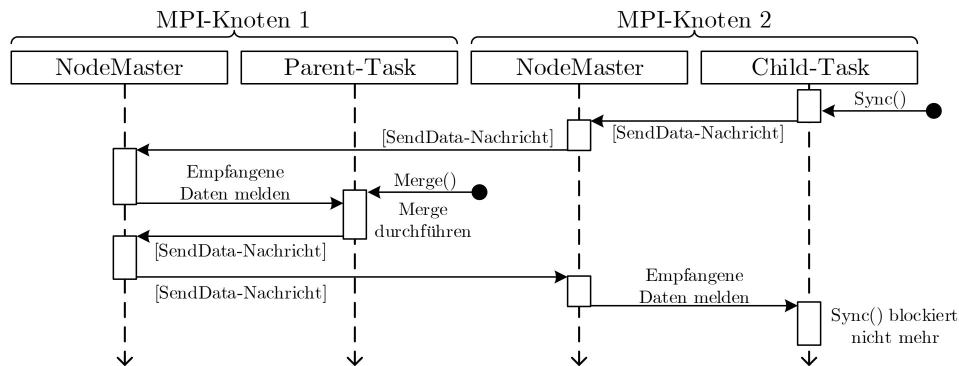


Abbildung 6.9: Nachrichtenfluss für *Sync*.

Zuweisung einer Task-ID für den neu gespawnten Sibling-Task und die Erstellung eines entsprechenden Task-Handles.

Es ist notwendig diese Zuweisung vom Parent-Task vornehmen zu lassen bevor der Sibling-Task gestartet wird. Zum einen wird so sichergestellt, dass es beim Parent-Task keine Kollision von Task-IDs gibt. Zum anderen wird der Parent-Task so zu einem Zeitpunkt über seinen neuen (nicht selbst gestarteten) Child-Task informiert, an dem er selbst auf jeden Fall noch nicht beendet ist (da zumindest der Child-Task noch ausgeführt wird, der aktuell `SpawnSibling` aufruft). Würde der Parent-Task erst beim Start der Ausführung des Sibling-Tasks über den neuen Child-Task informiert werden, so wäre es möglich, dass alle anderen Child-Tasks bereits fertiggestellt wurden und sich der Parent-Task bereits beendet hat.

Sobald die Informationen zwischen dem Child-Task und dem Parent-Task ausgehandelt wurden, schreibt der Child-Task die ausgehandelte Task-ID für den Sibling-Task in das Workitem und beginnt mit dem Versenden des Workitems (wie bei einem normalen `Spawn`-Aufruf).

6.2.5 Parameterübertragung

Da das Spawn & Merge Programmiermodell die Nutzung beliebiger Funktionen als Tasks erlaubt, muss ein Remote Procedure Call [93] für beliebige Funktionen realisiert werden. Dazu müssen insbesondere alle möglichen Parameterkombinationen in ein Workitem serialisierbar sein. Die Definition einer eigenen `SendWork`-Nachricht für jede Parameterkombination ist dabei nicht praktikabel. Im Rahmen des hier beschriebenen Prototyps wurden Templates (siehe Kapitel 4.3) für die Umsetzung der Serialisierung der Datenstrukturen und des Funktionsaufrufes genutzt, aus denen zur Compile-Zeit der notwendige Quelltext für die Serialisierung und den Funktionsaufruf (inklusive Deserialisierung) generiert wird. Dies ermöglicht den Verzicht auf die Nutzung aufwändiger Mechanismen (z.B. *Reflection*

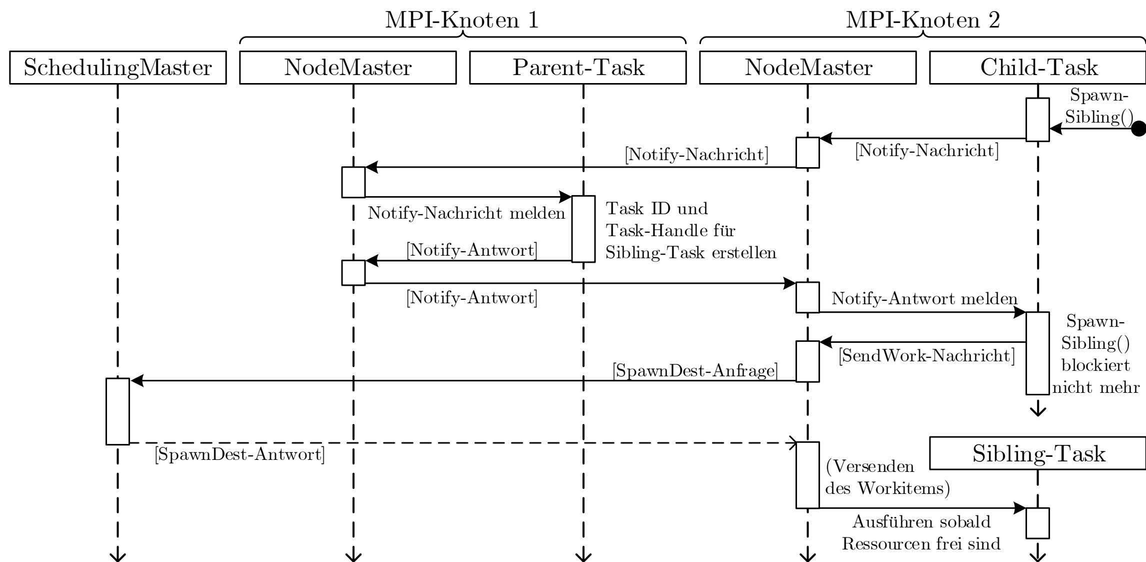


Abbildung 6.10: Nachrichtenfluss für *SpawnSibling*.

[37]), um beispielsweise zur Laufzeit zu analysieren, welchem Datentyp eine zusammenführbare Datenstruktur genau entspricht.

Die Serialisierung geschieht bei der rekursiven Auflösung des variadischen Templates für die entsprechende Synchronisationsprimitive, wie z.B. `Spawn` (siehe Kapitel 4.3). Dabei wird bei jedem Rekursionsschritt ein einzelner Parameter des Templates herausgenommen und verarbeitet. Beim `Spawn`-Template ist der erste Parameter ein Pointer auf die zu spawnende Funktion, der in einen Funktionsidentifier umgewandelt und in den Bytestream geschrieben wird. Anschließend folgen die Parameter der Funktion, sofern es sich nicht um eine parameterlose Funktion handelt. Mit jedem Rekursionsschritt wird ein Parameter abgearbeitet, indem er serialisiert und hinten an den Bytestream angehängt wird, bis keine weiteren Parameter mehr vorliegen. Dieses Vorgehen ist in Abbildung 6.11 zu sehen[14], wo der Funktionsidentifier zusammen mit zwei Parametern in den Payload-Bytestream einer `SendWork`-Nachricht serialisiert wird.

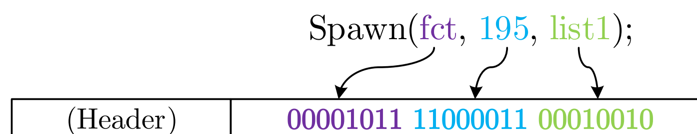


Abbildung 6.11: Serialisierung eines `Spawn`-Aufrufs in eine `SendWork`-Nachricht.

[14] Die Abbildung zeigt eine vereinfachte Version des Mechanismus. Im Prototypen werden noch zusätzliche Steuerinformationen in den Bytestream kodiert, die beispielsweise angeben, ob es sich um einen `Sync`-Aufruf handelt und welche Datenstrukturen übergeben wurden.

Zur Durchführung der Deserialisierung einer Datenstruktur ist es notwendig, den Datentypen der zu deserialisierenden Datenstruktur zu kennen. Dazu wird aus dem Bytestream im Workitem (nach dem First-in-first-out Prinzip) zuerst der Funktionsidentifier deserialisiert und die entsprechende Funktionssignatur aus der Zuordnung von Funktionsidentifiern zu Funktionssignaturen ausgelesen. Um die zu spawnende Funktion mit den deserialisierten Parametern ausführen zu können, wird als Mechanismus *Currying* [29] verwendet. Currying ermöglicht es dem Framework, im Verlaufe der Deserialisierung eine parameterlose Funktion zu erstellen, die anschließend als Task ausgeführt werden kann. Die Verwendung parameterloser Funktionen hat hierbei unter anderem den Vorteil, dass alle deserialisierten Tasks vom Framework gleich behandelt werden können.

Listing 6.4 zeigt die (vereinfachte) Funktion `curryFunction`, die im Prototypen genutzt wird, um den in einem Workitem enthaltenen ByteStream zu deserialisieren und eine parameterlose Funktion zu erstellen. Dazu erwartet die Funktion zum einen einen Pointer auf die Funktion, die umgewandelt werden soll (`fctToCurry`), um daraus die Funktionssignatur ableiten zu können. In Zeile 2 ist zu sehen, dass die Funktionssignatur von `fctToCurry` aus einem aktiven Argument `ActiveArgument` und einem variadischen Argument `Args...` besteht. Dies impliziert, dass die Funktion mindestens einen Parameter besitzen muss[15], der bei diesem Aufruf von `curryFunction` in die zu erreichende parameterlose Funktion integriert wird. In Zeile 3 wird der ByteSteam `incomingStream` als Parameter übergeben, der die serialisierten Datenstrukturen beinhaltet.

Im Rahmen der Funktion wird zuerst der aktive Parameter deserialisiert. Der Typ des Parameters ist durch `ActiveArgument` gegeben[16]. In Zeile 6 wird dazu ein Objekt des entsprechenden Typs angelegt und in Zeile 7 aus dem ByteStream `incomingStream` heraus deserialisiert. Anschließend wird in den Zeilen 10 bis 12 eine neue Funktion erstellt, die einen Parameter weniger als die Ausgangsfunktion hat (der Parameter `ActiveArgument` fällt weg). Die neu erstellte Funktion `curried` wird dabei als *Lambda-Funktion* [55] definiert und erhält, durch die Verwendung der *capture* `[=]`, eine Kopie aller im Funktionskörper verwendeten Symbole (d.h. insbesondere die deserialisierte Datenstruktur `argument`). Somit kann die resultierende Funktion `curried` mit einem Parameter weniger aufgerufen werden, da dieser bereits im Kontext der Lambda-Funktion enthalten ist. Abschließend wird die (um einen Parameter reduzierte) Funktion `curried` zusammen mit dem verbleibenden ByteStream `incomingStream` an den nächsten Rekursionsschritt übergeben, um weitere Parameter zu deserialisieren. Sobald alle Parameter entfernt wurden, wird die Rekursion abgebrochen und die zuletzt übergebene Funktion `curried` zurückgegeben.

Wird die parameterlose Funktion vom Framework aufgerufen, so werden die Parameter einzeln aus dem Funktionskörper der jeweiligen Lambda-Funktionen ausgelesen

[15] Falls die umzuwandelnde Funktion keinen Parameter besitzt wird die Rekursion sofort abgebrochen.

[16] Dabei ist zu beachten, dass es sich hierbei um einen Pointer oder eine Referenz handeln kann. In diesen Fällen muss der Typ in seinen Basistypen umgewandelt werden.

Listing 6.4 Deserialisierung eines Spawn-Aufrufs und *Currying*.

```
1:  std::function<void(void)> curryFunction(
2:    std::function<void(ActiveArgument,Args...)>& fctToCurry,
3:    ByteStream& incomingStream) {
4:
5:    // Deserialize active argument
6:    ActiveArgument argument = new ActiveArgument();
7:    argument->deserialize(incomingStream);
8:
9:    // Create function with one parameter less
10:   std::function<void(Args...)> curried = [=](Args... args) {
11:     fctToCurry(argument, args...);
12:   }
13:
14:   // Recursion for subsequent argument
15:   return curryFunction(curried, incomingStream);
16: }
```

und weitergegeben, bis die ursprüngliche Funktion des Tasks aufgerufen werden kann. Der Ablauf ist, für das Beispiel aus Abbildung 6.11, in Listing 6.5 dargestellt. Hier wird in Zeile 2 die parameterlose Funktion aufgerufen, in deren Funktionskörper der bereits deserialisierte `list`-Parameter vorliegt. In Zeile 8 wird mit diesem Parameter die `curriedFct`-Funktion aufgerufen, die einen Parameter erwartet und wiederum den deserialisierten `int`-Parameter beinhaltet. Diese ruft wiederum die ursprünglich gespawnte `fct`-Funktion mit beiden übergebenen Parametern auf, die somit als eigener Task gestartet werden kann.

6.2.6 Spezialfall: Lokale Ausführung

Wird die verteilte Spawn & Merge Anwendung auf nur einem einzelnen Rechenknoten ausgeführt (beispielsweise zum Testen während der Entwicklung), dann können Optimierungen vorgenommen werden, um die durch das Spawn & Merge Programmiermodell eingeführte Berechnungskomplexität zu reduzieren. Die Optimierungen umfassen dabei das Abschalten des Scheduling-Algorithmus und der Kommunikation zwischen MPI-Knoten, sowie das Tauschen der Serialisierung und Deserialisierung gegen ein direktes Kopieren der Datenstrukturen.

Der Scheduling-Algorithmus und die MPI-Komponente können abgeschaltet werden, wenn die Anwendung auf nur einem Rechenknoten ausgeführt wird. Das liegt zum einen daran, dass keine Kommunikation zwischen MPI-Knoten notwendig ist. Zum anderen gibt es nur einen Rechenknoten, sodass alle gespawnten Tasks auf dem einzigen laufenden

Listing 6.5 Aufruf der parameterlosen Funktion.

```
1:  // Call parameterless curried function
2:  curriedFct();
3:
4:  curriedFct(){
5:    list argument;
6:
7:    // Call curried function with one more parameter
8:    curriedFct(argument);
9:  }
10:
11: curriedFct(list parameter1){
12:   int argument;
13:
14:   // Call initial task function
15:   fct(argument, parameter1);
16: }
```

NodeMaster ausgeführt werden müssen. Die Ausführung eines SchedulingMasters und eines Scheduling-Algorithmus würde dementsprechend einen reinen Mehraufwand an Berechnungen bedeuten, ohne dass dadurch ein Vorteil erzielt werden kann. Das interne Scheduling der Tasks durch den NodeMaster ist davon nicht betroffen. So stellt der NodeMaster weiterhin sicher, dass immer nur die konfigurierten n Tasks ausgeführt werden. Ungestartete Tasks werden dementsprechend weiterhin in Form von Workitems in der WorkQueue vorgehalten.

Bei der lokalen Ausführung der verteilten Spawn & Merge Anwendung brauchen keine Workitems über die Grenzen des eigenen Rechenknotens hinaus verschoben werden. Dies ermöglicht den Verzicht auf das Serialisieren und Deserialisieren der Datenstrukturen, die an einen Task als Parameter übergeben werden. Stattdessen können die Parameter für den Task direkt kopiert werden, um den Deserialisierungsschritt einzusparen. Des Weiteren wird bei Verwendung der `Sync`-Primitive die Übertragung der Zwischenergebnisse an den MPI-Knoten des Parent-Tasks eingespart, da die Datenstrukturkopien bereits auf demselben Rechenknoten liegen. Ein für die lokale Ausführung optimiertes Kopieren erfordert dabei die Unterstützung der Datenstruktur. Wird dies nicht von der Datenstruktur angeboten, so kann als `Fallback` weiterhin auf die weniger effiziente Erstellung einer Datenstrukturkopie, durch das Serialisieren und Deserialisieren der Datenstruktur, zurückgegriffen werden.

Die Ausführung eines Child-Tasks auf demselben Knoten, auf dem auch der Parent-Task ausgeführt wird, ermöglicht des Weiteren die Umsetzung von *copy-on-write* Mechanismen. Statt eine als Parameter übergebene Datenstruktur direkt beim `Spawn`-Aufruf

zu kopieren, kann sowohl vom Child-Task als auch vom Parent-Task solange dieselbe Datenstruktur verwendet werden, wie nur lesende Operationen durchgeführt werden. Erst wenn einer der Tasks, die dieselbe Datenstruktur verwenden, eine Veränderung an den Daten vornimmt, muss (vor Anwendung der Veränderung) eine Kopie für alle Tasks erstellt werden. Im besten Fall brauchen Datenstrukturen so nicht kopiert werden, falls keine Veränderungen an den Daten durchgeführt werden. Ansonsten wird die Berechnungszeit für das Kopieren lediglich auf einen späteren Zeitpunkt der Anwendungsausführung verschoben (d.h. der Zeitpunkt, an dem einer der Tasks die geteilte Datenstruktur verändert).

In Kapitel 7.4 wird evaluiert, wie sich die beschriebenen Optimierungen auf die Ausführung einer verteilten Spawn & Merge Anwendung auf nur einem Rechenknoten auswirken.

6.3 Einordnung in den Stand der Forschung

In diesem Kapitel wird der beschriebene Prototyp für verteiltes Spawn & Merge mit den in Kapitel 2.2.4 beschriebenen Ansätzen für verteilte deterministische Anwendungen verglichen. Dazu wurde die Tabelle 2.1 aus Kapitel 2.2.4 um eine Spalte für den Prototypen für verteiltes Spawn & Merge erweitert (siehe Tabelle 6.1).

Der Spawn & Merge Prototyp unterscheidet sich von den Ansätzen *DDOS* und *Determinator* dadurch, dass er keine voll-deterministische Ausführung erzwingt, sondern speziell die Applikationslogik der Anwendung deterministisch und reproduzierbar ausführt. Das ermöglicht dem Prototypen mit hinzugefügten Ressourcen zu skalieren (siehe Kapitel 7.3). Bei Szenarien mit einem hohen parallelisierbaren Anteil ist es dem Prototypen dabei möglich, die Performance einer optimalen Parallelisierung zu erreichen. Ein direkter Vergleich einer nichtdeterministischen Ausführung einer Anwendung mit einer auf Spawn & Merge basierenden deterministischen Ausführung derselben Anwendung ist nicht möglich, da Spawn & Merge (im Gegensatz zu *DDOS* und *Determinator*) nicht das Ausführen beliebiger Anwendungen erlaubt. Aus diesem Grund kann die Verlangsamung der Anwendung, die sich aus der Nutzung von Spawn & Merge ergibt, nicht so ermittelt werden, wie es bei *DDOS* und *Determinator* der Fall ist.

Die Erweiterung bestehender General Purpose Languages (GPLs) um eine geringe Anzahl neuer Synchronisationsprimitive, zusammen mit der standardmäßig deterministischen Ausführung[17] der Anwendungsebene einer auf Spawn & Merge basierenden Anwendung, erleichtern die Entwicklung deterministischer verteilter Anwendungen. Liegt eine Anwendung bereits in derselben General Purpose Language (GPL) vor, so erleichtert dies die Anpassung der Applikation an das Spawn & Merge Programmiermodell, da die Anwendung nicht von Grund auf neu entwickelt werden muss. Auch einzelne Komponenten, die mit dem Spawn & Merge Framework entwickelt wurden (z.B. zusammenführbare

[17] Unter Einhaltunge der in Kapitel 4.1.1 genannten Anforderungen.

Datenstrukturen), können für weitere auf Spawn & Merge basierende Anwendungen wiederverwendet werden, um den Entwicklungsprozess zu beschleunigen.

Der im Spawn & Merge Programmiermodell verfolgte Ansatz des Determinismus auf Applikationsebene ähnelt, durch die Unterscheidung zwischen Logik und Scheduling, den Ansätzen der Coordination Languages (*Orc* und *CnC*). Die Ansätze sind dabei durch ihre Spezialisierung auf eine bestimmte Domäne oder funktionale Programmierung eingeschränkt. Spawn & Merge hingegen erlaubt die Entwicklung beliebiger Anwendungen und ist nicht auf eine bestimmte Programmiersprache festgelegt. Dabei ist zu beachten, dass sich nicht jedes Anwendungsszenario gleichermaßen für eine Umsetzung mit dem Spawn & Merge Programmiermodell eignet (siehe Evaluation in Kapitel 7).

Ein weiterer Punkt, in dem sich das Spawn & Merge Framework von *DDOS*, *Determinator* und *Orc* unterscheidet, ist, dass Konflikte nicht verhindert, sondern explizit zugelassen werden. So werden Wartezeiten zwischen nebenläufigen Tasks verhindert, die sich aus dem Sperren kritischer Codebereiche ergeben. Treten Konflikte auf, so werden diese deterministisch aufgelöst. Hierbei wird Operational Transformation als Standardmechanismus genutzt. *DDOS*, *Determinator* und *Orc* benötigen hingegen keine Mechanismen zur Konfliktauflösung, da Konflikte (auf Kosten der Performance) verhindert werden oder immer in der gleichen Art und Weise auftreten. *CnC* überlässt die Auflösung auftretender Konflikte dem Entwickler und bietet keine Mechanismen zur Konfliktauflösung an.

Wie hoch die Performancekosten sind, die sich durch die Verwendung von Operational Transformation ergeben, hängt von der Art der Anwendung und der zu erwartenden Häufigkeit von Konflikten ab. Für welche Anwendungen das Spawn & Merge Framework geeignet ist und in welchen Fällen hohe Performancekosten zu erwarten sind, wird in Kapitel 7.5 evaluiert.

Eigenschaft	DDOS	Determinator	Orc (CnC)	Verteiltes Spawn & Merge
Verteilt	Ja	Ja	Ja / (Möglich)	Ja
Determinismus	Voll/Replay	Voll/Replay	Funktionen	Applikationslogik
Reproduzierbarkeit	Alles	Alles	Funktionsaufrufe	Applikationslogik
Skalierbarkeit	Eingeschränkt durch Barrieren	Embarrassingly parallel	Skaliert	Skaliert
Generalität	Alles	Alles	Speziell	Alles, aber unterschiedliche Eignung
Wiederverwendbarkeit	Ja	Ja	Nein	Ja
Einschränkungen	Skalierbarkeit	Skalierbarkeit	Domäne / Keine Zwischenergebnisse	OT Algorithmen & Komplexität
Verlangsamung	10x	bis zu 10x	–	–
Einfache Verwendung	Ja	Ja	Nein, spezielle Graphsprache	Ja
Konfliktauflösung	–	–	–/(Entwickler)	OT-Algorithmen

Tabelle 6.1: Einordnung von Spawn & Merge in die deterministischen Verteilten Systeme.

Kapitel 7

Evaluation

Im Rahmen der Evaluation wird untersucht, für welche Art von Anwendungen sich das Spawn & Merge Programmiermodell eignet und wie hoch der Performanceverlust durch die eingeführte Garantie einer deterministischen Programmausführung auf der Applikationsebene ist. Im Einzelnen wird dazu untersucht, inwiefern unterschiedliche Beispielszenarien mit einer wachsenden Ausführungsumgebung skalieren und wie groß die Kosten der einzelnen Mechanismen zur Realisierung der deterministischen Programmausführung in Abhängigkeit von verschiedenen Charakteristika einer Anwendung sind. Diese Untersuchungen sollen es Entwicklern ermöglichen, im Vorhinein abzuschätzen, ob sich das Spawn & Merge Programmiermodell zur Umsetzung einer Anwendung eignet.

7.1 Versuchsaufbau

Grundlage für die Evaluation ist eine prototypische Implementierung des in Kapitel 6 entwickelten Konzeptes für Spawn & Merge in einem Verteilten System. Der Prototyp wurde dabei in C++11 entwickelt und nutzt als Middleware zur Abstraktion des Nachrichtenaustausches MPI [69] (konkret in der Implementierung MPICH2 [44]). Der NodeMaster (siehe Kapitel 6.2.2) des Prototyps ist so konfiguriert, dass er so viele Tasks parallel ausführt, wie es Prozessorkerne auf dem entsprechenden MPI-Knoten gibt (n = Anzahl Prozessorkerne). Der Prototyp ist des Weiteren so aufgebaut, dass im Rahmen der Evaluation einzelne Funktionalitäten zugeschaltet oder abgeschaltet werden können, um die Auswirkungen einzelner Mechanismen evaluieren zu können. In Bezug auf das Scheduling von Tasks auf MPI-Knoten kann so unterschieden werden, ob die Verteilung statisch (*Round-robin* [92]) oder dynamisch geschehen soll. Bei dynamischem Scheduling übernimmt ein MPI-Knoten zusätzlich die Aufgabe des SchedulingMasters, der angefragt werden kann, um einen passenden (möglichst wenig ausgelasteten) MPI-Knoten für einen neuen Task zu finden. Zusätzlich können MPI-Knoten, die keine weiteren Workitems zur Bearbeitung

vorliegen haben, bei dem SchedulingMaster anfragen, ob einem überlasteten MPI-Knoten zugewiesene und bisher nicht gestartete Workitems abgenommen werden können (siehe Workstealing-Algorithmus in Kapitel 6.2.3). Die letzte konfigurierbare Funktionalität, die im Rahmen dieser Evaluation genutzt wird, ist die Verwendung der in Kapitel 6.2.6 beschriebenen Mechanismen zur Optimierung der Ausführung auf einem einzelnen Rechenknoten. Ist diese aktiviert, dann müssen Datenstrukturen für die Übertragung zu einem anderen MPI-Knoten nicht serialisiert (und dort deserialisiert) werden und es wird auf den SchedulingMaster verzichtet. Zusätzlich können *copy-on-write* Mechanismen (sofern von der Datenstruktur implementiert) dafür sorgen, dass eine Datenstruktur nicht kopiert werden muss, sofern nur lesend auf diese zugegriffen wird.

Als Ausführungsumgebung für die Messungen dienen 10 vernetzte Virtuelle Maschinen (VMs)[1]. Um die Aussagekraft der Messungen zu erhöhen, wurde hierbei eine homogene Zusammenstellung der VMs verwendet, sodass alle VMs als gleichwertig betrachtet werden können. Bei der Verwendung eines heterogenen Systems sind Aussagen bezüglich der Skalierbarkeit oder der generellen Performance des Systems nur erschwert möglich, da die zusätzliche Rechenkraft des Verteilten Systems von der Rechenkraft des zusätzlichen Rechenknotens abhängt und nicht nur davon wie viele neue Rechenknoten hinzugefügt wurden.

7.2 Szenarien

In diesem Kapitel werden die verschiedenen Szenarien vorgestellt, die zur Evaluation des Prototyps für Spawn & Merge in einem Verteilten System genutzt werden. Gemessen wird dabei die Ausführungszeit zwischen dem Zeitpunkt, an dem die Anwendung ihre Initialisierung abgeschlossen hat (d.h. das Anlegen und Initialisieren benötigter Variablen für das Szenario) bis zu dem Zeitpunkt an dem alle Ergebnisse vorliegen (bevor diese ausgegeben werden).

Neben den bereits beschriebenen Konfigurationsmöglichkeiten für den Prototyp können auch die unterschiedlichen Szenarien mit verschiedenen Eingabeparametern gestartet werden, um deren Verhalten zu steuern. Um Ausreißer der Messung abzuschwächen, wird jede getestete Konfiguration (des Prototyps zusammen mit dem Szenario) fünfmal ausgeführt und der Mittelwert der Ausführungszeit gebildet. Zusätzlich wird in den jeweiligen Ergebnissen die Standardabweichung innerhalb der Messung angegeben.

Die Szenarien und deren Besonderheiten im Rahmen der Evaluation werden im Folgenden beschrieben. Dabei wird insbesondere auf die Gründe für die Verwendung ebendieser Szenarien eingegangen, sowie die Konfigurierbarkeit beschrieben.

[1]Amazon EC2 *c4.xlarge*, Stand Juni 2017. 4 × 2,9-GHz Intel Xeon E5-2666 v3-Hochfrequenzprozessoren, 7,5 GByte Arbeitsspeicher, 750 Mbit/s [4].

7.2.1 Normale Verteilung

Die Zielsetzung des *Normale Verteilung* Szenarios ist das Testen des dynamischen Schedulings des Prototypen und der generellen Skalierbarkeit. Eine unterschiedliche Komplexität der Child-Tasks sorgt dabei für eine ungleichmäßige Verteilung der Arbeitslast auf die verfügbaren MPI-Knoten, die durch das dynamische Scheduling (und dabei insbesondere durch den Workstealing-Algorithmus) behoben werden soll. Die Skalierbarkeit soll dadurch gezeigt werden, dass die Ausführungszeit mit den hinzugefügten Rechenknoten skaliert.

Im Rahmen dieses Szenarios wird eine Arbeitslast l angegeben, die auf k nebenläufig ausgeführte Child-Tasks aufgeteilt wird. Die Tasks bekommen dabei unterschiedliche Anteile an der Arbeitslast l zugewiesen, sodass sich eine ungleichmäßige Verteilung der Arbeitslast auf die verfügbaren Rechenknoten ergibt. So soll abgebildet werden, dass sich Tasks in einer Anwendung in ihrer Komplexität unterscheiden können. Damit die Zuweisung der Arbeitslast deterministisch ist, werden die k Tasks auf drei Klassen aufgeteilt, die jeweils für unterschiedlich große Anteile an der Arbeitslast stehen. 65% der Tasks werden der Klasse *Niedrig* zugewiesen und berechnen einen kleinen Anteil der Arbeitslast l. 30% der Tasks werden der Klasse *Mittel* zugewiesen und berechnen den fünffachen Anteil der Klasse *Niedrig*. Der kleinsten Klasse *Hoch* werden 5% der Tasks zugewiesen. Diese berechnen den dreißigfachen Anteil der Klasse *Niedrig*. Die Spawn-Reihenfolge der Tasks wird dabei durch einen Pseudozufallszahlengenerator (PRNG) mit einem statischen Seed ausgewählt und ist somit auch bei unterschiedlichen Durchführungen der Messungen deterministisch.

Zur Realisierung dieser Arbeitslast werden l $SHA3_{256}$ Hashwerte errechnet. Jeder Task speichert dabei eine Liste der h kleinsten gefundenen Hashwerte. Nachdem ein Task seinen Anteil der Arbeitslast abgearbeitet hat wird seine Liste der h kleinsten Hashwerte an den Parent-Task zurückgegeben und dort zusammengeführt. Als Synchronisationspunkte ergeben sich somit die `Spawn`-Aufrufe für das Starten der k Child-Tasks, sowie das Zusammenführen eben dieser Child-Tasks (inklusive der Operational Transformation für die Zusammenführung der Ergebnislisten mit jeweils h Elementen).

7.2.2 Extreme Verteilung

Das *Extreme Verteilung* Szenario ist eine Variante des in Kapitel 7.2.1 beschriebenen *Normale Verteilung* Szenarios. Das Szenario *Extreme Verteilung* unterscheidet sich in der Verteilung der Arbeitslast auf die Child-Tasks. Im Gegensatz zum *Normale Verteilung* Szenario soll hier eine hochgradig ungleichmäßige Verteilung der Arbeitslast erreicht werden, um die Skalierbarkeit einer Spawn & Merge Anwendung auch bei extrem ungleichmäßig verteilter Berechnungskomplexität in der Task-Hierarchie zu evaluieren.

Im *Extreme Verteilung* Szenario wird die Arbeitslast l ebenfalls auf k nebenläufig ausgeführte Child-Tasks verteilt, um eine Liste der h kleinsten $SHA3_{256}$ Hashwerte zu errechnen. Die Szenarien unterscheiden sich jedoch in der Verteilung der Child-Tasks auf die Klassen, sowie in den Anteilen der einzelnen Klassen an der gesamten Arbeitslast. 95,725% der Tasks werden der Klasse *Niedrig* zugewiesen und berechnen einen kleinen Anteil der Arbeitslast l. 4% der Tasks werden der Klasse *Mittel* zugewiesen, in der sie die fünffache Arbeitslast der Klasse *Niedrig* berechnen. Die restlichen 0,125% der Tasks berechnen in der Klasse *Hoch* die fünfzigfache Arbeitslast der Klasse *Niedrig*. Die so erreichte ungleichmäßige Verteilung der Arbeitslast auf die Child-Tasks steht beispielhaft für Anwendungen, in denen einzelne Teilaufgaben eine besonders hohe Komplexität besitzen, die nicht weiter aufgeteilt werden kann. In diesem Falle sollen die dynamischen Scheduling-Mechanismen die restliche Arbeitslast auf die übrigen Rechenknoten umverteilen, um dennoch eine möglichst skalierende Ausführung zu erreichen.

Ein weiterer Unterschied ist die Spawn-Reihenfolge der Tasks. Im *Extreme Verteilung* Szenario werden die Tasks der Klasse *Hoch* zuerst gestartet, sodass diese in der logischen `Merge`-Reihenfolge zuerst zusammengeführt werden müssten. Dadurch soll eine Evaluation des in Kapitel 5.3.3 beschriebenen Mechanismus für eine Zusammenführung in beliebiger Reihenfolge (insbesondere FCFS) ermöglicht werden. Dies ist notwendig, da eine Zusammenführung in FCFS Reihenfolge nur dann einen positiven Einfluss hat, wenn es einen lange laufenden Child-Task gibt, auf den der Parent-Task warten muss, bevor er weitere Child-Tasks zusammenführen kann. Die Synchronisationspunkte entsprechen wiederum den Synchronisationspunkten des *Normale Verteilung* Szenarios.

7.2.3 Damenproblem

Das *Damenproblem* [89] befasst sich mit der Fragestellung, wie viele mögliche Anordnungen es für n Damen auf einem $n \times n$ großen Schachbrett gibt, in denen keine Dame eine andere Dame schlagen kann[2]. In Abbildung 7.1a ist ein Beispiel für eine mögliche Lösung des Damenproblems für $n = 4$ dargestellt, während Abbildung 7.1b eine ungültige Platzierung der Damen auf dem Spielfeld zeigt, da sich die obere rechte Dame in derselben Reihe wie die obere mittlere Dame befindet. Für die Evaluation wurde das Damenproblem als Beispiel für ein Problem der realen Welt ausgewählt.

Um eine Lösung des *Damenproblems* zu finden, wird hier zur Illustration eine *Tiefensuche* über den Lösungsraum des *Damenproblems* durchgeführt, obwohl es auch komplexere, weitergehende Ansätze zur Lösung des *Damenproblems* gibt [16, 98]. Der Algorithmus geht für jede Zeile des Schachbrettes (beginnend mit der ersten Zeile) alle möglichen Posi-

[2] Bei einem Schachspiel kann sich eine Dame-Spielfigur beliebig viele Schritte entlang der X-Achse und der Y-Achse des Spielbretts, sowie entlang der Diagonalen (ausgehend vom eigenen eingenommenen Spielfeld) bewegen.

	D		
			D
D			
		D	

(a) Valide Platzierung von 4 Damen.

	D		D
D			
		D	

(b) Invalide Platzierung von 4 Damen.

Abbildung 7.1: Das Damenproblem für $n = 4$.

tionierungen einer Dame innerhalb dieser Zeile durch. Wird eine valide Positionierung gefunden, dann springt der Algorithmus in die darunterliegende Zeile und versucht wiederum eine Dame zu positionieren. Dieses Vorgehen wird wiederholt, bis entweder einer Dame in der letzten Zeile eine valide Position zugewiesen werden kann (d.h. eine Lösung wurde gefunden) oder bis es für eine Zeile keine valide Positionierung einer Dame gibt. In letzterem Falle springt der Algorithmus eine Zeile zurück und prüft die nächste mögliche Positionierung innerhalb dieser Reihe.

Im Rahmen dieser Evaluierung kann neben der Größe n des Schachbrettes noch die Anzahl nebenläufig bearbeiteter Zeilen des Schachbrettes z angegeben werden. Für die ersten z Zeilen des Schachbrettes werden dabei alle möglichen Positionierungen einer Dame innerhalb der Zeile parallel getestet indem für jede Positionierung ein eigener Child-Task gestartet wird. Ab der Teile $z + 1$ übernimmt der entsprechende Child-Task die Durchführung der restlichen Tiefensuche innerhalb der verbleibenden $n - z$ Zeilen. Während der Ausführung ergibt sich somit eine Task-Hierarchie mit z Ebenen von Child-Tasks und insgesamt einer Anzahl von $N \approx n^z$ gestarteten Tasks[3].

Die Implementierung des *Damenproblems* nutzt für das Zählen der gefundenen Möglichkeiten eine spezialisierte Datenstruktur, die eine vereinfachte Konfliktauflösung ermöglicht. Die Möglichkeit alternative Mechanismen für die Konfliktauflösung zu nutzen (sofern anwendungsspezifisches Wissen dies erlaubt) wurde bereits in Kapitel 5.2.3 beschrieben. Im Falle des *Damenproblems* wurde hier ein zusammenführbarer Zähler implementiert, bei dem in jeder Kopie des Zählers lediglich die Veränderungen Δ nachgehalten werden müssen. Dieses Δ kann bei der Zusammenführung ohne Veränderung auf den Wert des Zählers innerhalb des Parent-Tasks addiert werden. Dies dient als Beispiel dafür, wie durch anwen-

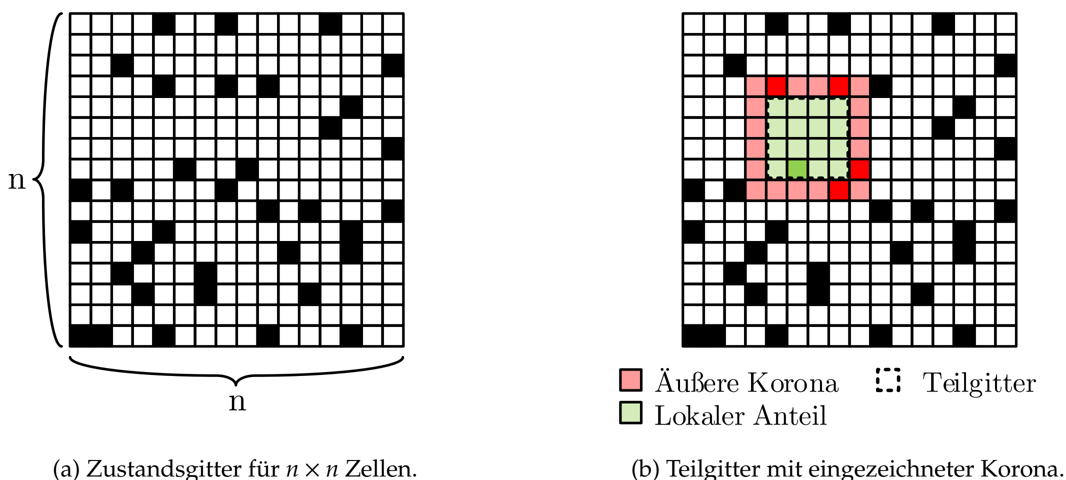
[3]Genauer handelt es sich um $\sum_{i=0}^{z} n^i$ Tasks, da ansonsten nur die Blattknoten der Task-Hierarchie gezählt werden.

dungsspezifisches Wissen die Verwendung eines teuren OT-Systems vermieden werden kann.

7.2.4 Game of Life

Das *Game of Life* Szenario ist eine Implementierung von Conway's Game of Life [40]. Beim *Game of Life* Szenario handelt es sich um einen *zellulären Automaten*, der als weiteres Beispiel für ein Problem der realen Welt ausgewählt wurde. Zelluläre Automaten können genutzt werden, um natürliche Phänomene zu modellieren und zu simulieren.

Der zelluläre Automat wird beim *Game of Life* in Form eines zweidimensionalen Gitters der Größe $n \times n$ abgebildet. Jede Zelle des Gitters entspricht einer Zelle des zellulären Automaten und kann entweder lebendig (Schwarz) oder tot (Weiß) sein. Zu Beginn der Simulation werden die Zellen zufällig als lebendig oder tot initialisiert. Ein Beispiel dafür ist in Abbildung 7.2a zu sehen.



(a) Zustandsgitter für $n \times n$ Zellen.

(b) Teilgitter mit eingezeichneter Korona.

Abbildung 7.2: Das *Game of Life* Zustandsgitter.

Zur Simulation wird beim *Game of Life* ausgehend vom Zustand (einer Zellenbelegung) zu einem Zeitpunkt t_x der Zustand zum Zeitpunkt t_{x+1} bestimmt. Dazu wird eine Menge vorgegebener Regeln (siehe [40]) auf jede einzelne Zelle Z angewandt, die in Abhängigkeit von den Nachbarn der Zelle (N_Z) entscheiden, ob Z zum Zeitpunkt t_{x+1} lebendig ist, oder nicht. Jede Zelle hat genau acht benachbarte, angrenzende Zellen. Für Zellen, die am Rand des Gitters liegen, gelten dabei die gegenüberliegenden Zellen als Nachbarn (*Wrapping*). Das Ergebnis der Simulation ist der Zustand (die Belegung des Gitters) nach s Simulationsschritten.

Zur Parallelisierung von *Game of Life* wird das $n \times n$ Gitter in t kleinere Teilgitter aufgeteilt (wobei $n \mod \sqrt{t} \stackrel{!}{=} 0$ gelten sollte), die parallel innerhalb einzelner Child-Tasks

simuliert werden. Um die äußeren Zellen des Teilgitters simulieren zu können, muss zusätzlich noch ein eine Zelle breiter Ring um das Teilgitter herum an die Tasks übergeben werden (im Folgenden *Korona* genannt, siehe Abbildung 7.2b). Die Korona entspricht hierbei den Zellen, die eigentlich von einem anderen Child-Task simuliert werden, aber für die eigenen Berechnungen erforderlich sind. Da auch die weiteren Child-Tasks eine Korona besitzen, entsprechen die äußeren Zellen des eigenen Teilgitters den Zellen, die Teil der Koronen anderer Child-Tasks sind. Die Koronen stellen somit die Schnittstelle zwischen den einzelnen Child-Tasks dar. Nach jedem Simulationsschritt müssen alle Zustände aller in Koronen enthaltenen Zellen zwischen den benachbarten Child-Tasks ausgetauscht werden, damit jeder Child-Task den nachfolgenden Simulationsschritt berechnen kann. Dieser Austausch wird durch einen Aufruf der `Sync`-Primitive, mit den Zellen der Korona als Parameter, realisiert. Ein Austausch der restlichen lokalen Zellen ist nicht notwendig, da diese keinen unmittelbaren Einfluss auf die Berechnungen anderer Child-Tasks haben können. Nach der Durchführung aller Simulationsschritte werden alle berechneten Zustände der Zellen an den Parent-Task zurückgesendet, der das Gesamtergebnis zusammenstellt.

Für den Aufruf der `Sync`-Primitive ist zu beachten, dass diese mit dem Parameter `RETURN_AFTER_MERGE_CYCLE_COMPLETED` aufgerufen wird, da der Child-Task zur Aktualisierung der Korona die Ergebnisse aller anderen Child-Tasks benötigt. Für die Zusammenführung nutzt die Implementierung des *Game of Life* Szenarios zwei zusammenführbare Datenstrukturen die ineinander verschachtelt werden (siehe Kapitel 5.5). Die innere Datenstruktur der Verschachtlung repräsentiert eine einzelne Zelle (`MergeableCell`). Die äußere Datenstruktur repräsentiert ein Array von Zellen (d.h. das Teilgitter oder eine Korona). Wie in Kapitel 5.5 beschrieben, löst hierbei eine Zusammenführung zweier Arrays von Zellen eine Zusammenführung aller enthaltenen Zellen aus.

7.2.5 Operational Transformation Szenario

Das *Operational Transformation* Szenario dient der Evaluation eines OT-Systems in Hinblick auf dessen Nutzbarkeit für unterschiedliche Anwendungsarten. Die Nutzbarkeit hängt im Kontext eines OT-Systems sowohl von der erwarteten Anzahl aufzulösender Konflikte (*anwendungsabhängig*) als auch von der generellen Berechnungskomplexität der Konfliktauflösung sowie der Operationen auf der darunterliegenden Datenstruktur ab (*datenstrukturabhängig*). Das Ziel des *Operational Transformation* Szenarios ist die Quantifizierung der durch Operational Transformation entstehenden Kosten in Abhängigkeit von verschiedenen Nutzungsprofilen der Datenstruktur. Im Kontext dieser Evaluation zielt das Szenario dabei explizit auf die in Kapitel 5 vorgestellten OT-Systeme für Listen.

In diesem Szenario startet ein Parent-Task einen Child-Task. Um unterschiedliche Nutzungsprofile abbilden zu können, kann das Verhalten der Tasks im Rahmen des *Operational Transformation* Szenarios konfiguriert werden. Zu diesen Einstellungsmöglichkeiten zählt

zum einen die Anzahl n der Elemente, die zum Startzeitpunkt des Child-Tasks bereits in der Liste enthalten sind (Listengröße). Als zweiter Parameter kann die Anzahl o der Operationen (`insert`- und `delete`-Operationen), die vom Parent-Task und vom Child-Task nebenläufig auf der übergebenen Liste durchgeführt werden sollen, angegeben werden. Der dritte Parameter definiert die Arbeitslast l, die von beiden Tasks bewältigt werden muss. Haben beide Tasks ihre zugewiesenen Aufgaben erfüllt, werden sie wieder zusammengeführt. Das *Operational Transformation* Szenario kann dabei mit unterschiedlichen OT-Systemen s gestartet werden, die den vierten Parameter darstellen. Der Anteil der OT an der Ausführungszeit kann ermittelt werden, indem dieselbe Szenariokonfiguration zusätzlich mit einem Prototyp ausgeführt wird, in dem die Operational Transformation Mechanismen abgeschaltet sind.

Im Rahmen des *Operational Transformation* Szenarios ist zu beachten, dass eine initiale Liste mit n Elementen bedeutet, dass auf dieser Liste bereits n Operationen durchgeführt wurden. Auch wenn diese Operationen im Rahmen der Zusammenführung übersprungen werden können, so ergibt sich doch ein Einfluss auf die Laufzeit für die Durchführung von Operationen auf der Datenstruktur. So ist insbesondere für die Liste des (für eine Zusammenführung in beliebiger Reihenfolge) erweiterten OT-Systems eine höhere Komplexität für die Durchführung von Operationen zu erwarten, da bedingt durch die Einführung von Tombstones die Positionen der Operationen vor ihrer Anwendung angepasst werden müssen, damit die Tombstones mit einbezogen werden (siehe Kapitel 5.3.3).

7.3 Skalierbarkeit

In diesem Kapitel wird untersucht, ob der Prototyp für Spawn & Merge in Verteilten Systemen mit der Anzahl verfügbarer Rechenknoten (hier MPI-Knoten) skalieren kann und ob es anwendungsabhängige Unterschiede in der Skalierbarkeit gibt. In letzterem Falle werden des Weiteren die Gründe für die ausbleibende Skalierbarkeit untersucht. Dabei wird insbesondere evaluiert, wie sich dynamisches und statisches Scheduling bei einer wachsenden Ausführungsumgebung auswirken.

Dazu werden die im Folgenden beschriebenen Szenariokonfigurationen mit einer ansteigenden Anzahl von n MPI-Knoten ausgeführt. Die Anzahl verfügbarer MPI-Knoten steigt dabei von $n = 1$ zu $n = 10$. Dieses Vorgehen wird sowohl mit dynamischem Scheduling als auch mit statischem Scheduling durchgeführt.

Zur Visualisierung der Skalierbarkeit werden in diesem Kapitel *Speedup-Graphen* (Beschleunigungsgraphen) genutzt. In den Speedup-Graphen ist auf der X-Achse die Anzahl genutzter MPI-Knoten abgebildet während auf der Y-Achse der Beschleunigungsfaktor (relativ zur Ausführungsgeschwindigkeit mit einem MPI-Knoten) abgebildet ist. Eine perfekt

skalierende verteilte Anwendung[4] würde bei einer Verdopplung der verfügbaren Ressourcen (hier MPI-Knoten) doppelt so schnell ausgeführt werden (siehe Amdahl's Law [5]). Diese optimale Skalierbarkeit wird in den Graphen der folgenden Kapitel als Referenz mit eingezeichnet.

Für die Speedup-Graphen der Messungen ergibt sich somit, dass die Skalierbarkeit gegeben ist, falls es sich bei dem Graphen um eine Gerade handelt, wobei die Steigung der Gerade angibt, wie „gut" diese Skalierbarkeit ist. Je steiler die Gerade ist, desto größer ist der Anteil der zusätzlichen Ressourcen, der tatsächlich zur Beschleunigung der Ausführung genutzt werden kann und desto geringer ist der Abstand der tatsächlichen Beschleunigung von der optimalen Beschleunigung der Anwendung. Dies ist abhängig davon, wie viel zusätzliche Berechnungszeit für die Lösung der Aufgabe der Anwendung verwendet werden kann und wie viel Berechnungszeit für die Verteilungsmechanismen und Mechanismen zur Sicherstellung des Determinismus aufgewendet werden muss.

7.3.1 Normale Verteilung

Das *Normale Verteilung* Szenario wird für die Evaluation der Skalierbarkeit mit einer niedrigen Arbeitslast (NV_N) und einer hohen Arbeitslast (NV_H) ausgeführt. Die niedrige Arbeitslast entspricht dabei $l \approx 5,08 \cdot 10^{10}$ CPU-Zyklen, während die hohe Arbeitslast $l \approx 1,016 \cdot 10^{12}$ CPU-Zyklen entspricht[5]. In beiden Fällen wird die Arbeitslast auf insgesamt $t = 1.000$ Tasks verteilt und die $h = 5$ kleinsten Hashwerte gesucht.

Die Graphen in Abbildung 7.3 (NV_N) und in Abbildung 7.4 (NV_H) zeigen die erreichten Beschleunigungen, die durch hinzugefügte Rechenknoten erreicht werden konnten. Die Messungen wurden dazu auf $n \in \{1, \ldots, 10\}$ der in Kapitel 7.1 beschriebenen VMs ausgeführt und die Ausführungszeit gemessen. Anhand der Ausführungszeit für $n = 1$ wurde anschließend der abgebildete Beschleunigungsfaktor berechnet.

Bei den Ergebnissen der NV_N und NV_H Messungen mit statischem Scheduling ist erkennbar, dass der Verlauf des Beschleunigungsfaktors zwar mit den hinzugefügten Ressourcen skaliert, jedoch im Vergleich zu den Ergebnissen der Messungen mit dynamischem Scheduling ungleichmäßiger ist und eine geringere Steigung hat. Das ergibt sich daraus, dass beim statischen Scheduling kein Wissen über die Auslastung des Systems mit einbezogen wird. Eine ungleichmäßige Verteilung der Arbeitslast l auf die t Tasks kann dabei nicht zur Laufzeit behoben werden (siehe auch Kapitel 6.2.3). Dadurch ergeben sich zum einen die geringeren Beschleunigungsfaktoren bei $n = 4$, $n = 6$ und $n = 7$. Zum anderen ergibt sich daraus der Abstand zwischen dem Beschleunigungsfaktor für statisches

[4] D.h. eine verteilte Anwendung ohne nicht parallelisierbare Anteile.

[5] Die Arbeitslast wird hier durch die Berechnung von 10.000.000 bzw. 200.000.000 $SHA3_{256}$ Hashwerten erzeugt. Für die Abschätzung der CPU-Zyklen wurde die Ausführungszeit für $SHA3_{256}$ Hashwert-Berechnungen gemessen und die CPU-Zyklen anhand der CPU-Taktung approximiert.

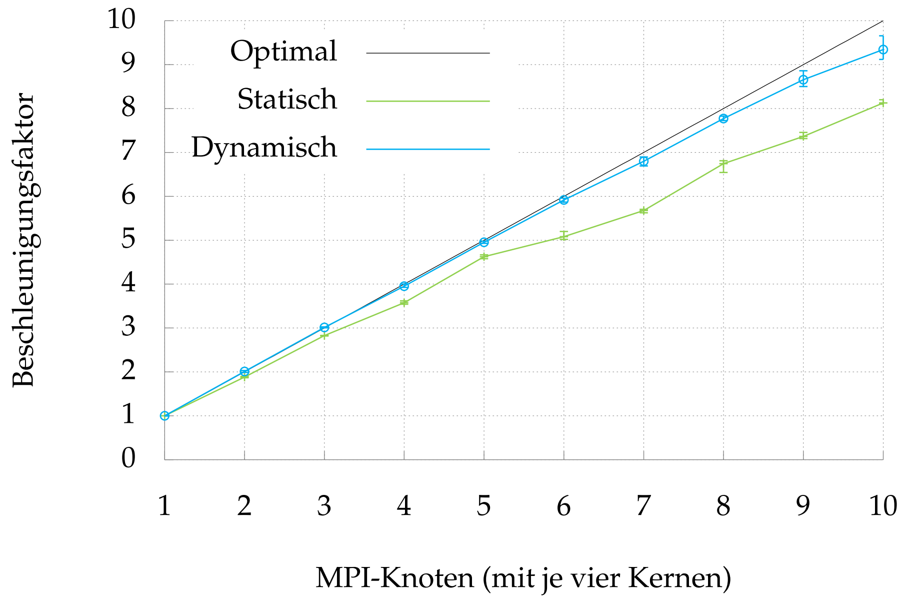


Abbildung 7.3: Speedup-Graph der NV_N Messung.

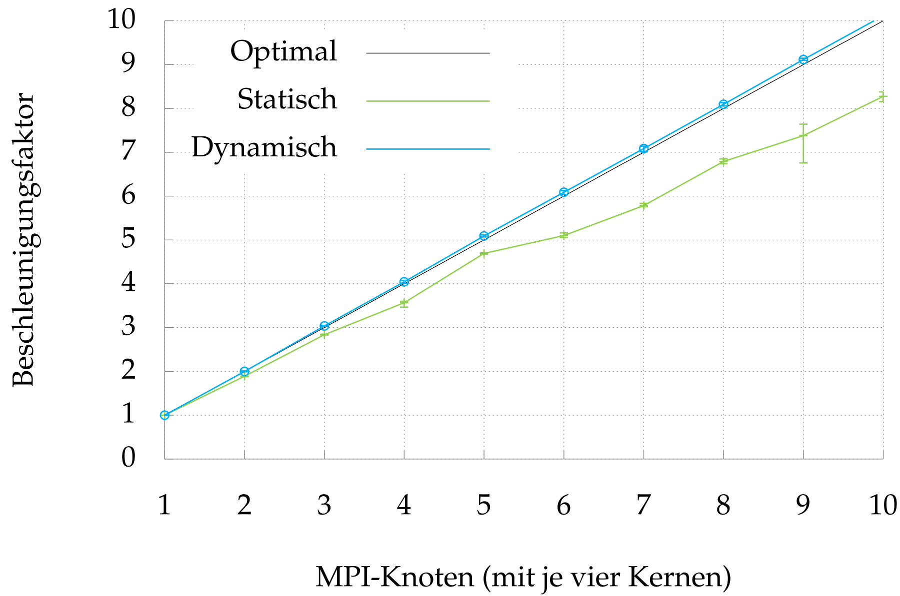


Abbildung 7.4: Speedup-Graph der NV_H Messung.

und dynamisches Scheduling. Während der Beschleunigungsfaktor für $n = 10$ beim statischen Scheduling 18,7% (NV_N) und 17,3% (NV_H) unter dem theoretischen Optimum liegt, liegt der Beschleunigungsfaktor beim dynamischen Scheduling nur 6,6% (NV_N) un-

ter dem Optimum. Bei der Messung NV_H mit dynamischem Scheduling entspricht der Beschleunigungsfaktor dem Optimum, was einer sehr guten Skalierbarkeit für hohe Arbeitslasten entspricht. Hier zeigt sich durch den Vergleich mit statischem Scheduling, dass das dynamische Scheduling eine bessere Verteilung der Arbeitslast auf die Rechenknoten erreicht und dabei auch ungünstige Verteilungen wieder besser verteilen kann. Zusätzlich zeigt der Vergleich der Messungen für niedrige Arbeitslast und hohe Arbeitslast, dass die Anwendung bei einer hohen Arbeitslast besser skaliert. Das liegt daran, dass die Kosten der nicht parallelisierbaren Mechanismen für eine deterministische Ausführung und die Verteilung der Anwendung konstant sind und deren Anteil an der gesamten Laufzeit bei höherer Arbeitslast geringer ausfällt.

In Abbildung 7.4 ist erkennbar, dass der Graph für dynamisches Scheduling teilweise über dem Optimum verläuft. Dies liegt an der beschriebenen Approximation des optimalen Beschleunigungsfaktors anhand der Laufzeit für $n = 1$. Ausreißer für $n = 1$ beeinflussen somit überdurchschnittlich die Approximation für höhere n.

7.3.2 Extreme Verteilung

Das *Extreme Verteilung* Szenario nutzt für die Arbeitslast ebenfalls die Größenordnungen $l \approx 5,08 \cdot 10^{10}$ CPU-Zyklen für die niedrige Arbeitslast (EV_N) und $l \approx 1,016 \cdot 10^{12}$ CPU-Zyklen für die hohe Arbeitslast (EV_H) während $t = 1.000$ Tasks die $h = 5$ kleinsten $SHA3_{256}$ Hashwerte suchen. Das *Extreme Verteilung* Szenario unterscheidet sich vom *Normale Verteilung* Szenario durch die besonders ungleichmäßige Verteilung der Arbeitslast l auf die t Tasks (siehe Kapitel 7.2.2). Die Beschleunigung der Anwendungsausführung bei einer steigenden Anzahl von $n \in \{1, \ldots, 10\}$ MPI-Knoten ist in den Speedup-Graphen in Abbildung 7.5 (EV_N) und in Abbildung 7.6 (EV_H) dargestellt.

Im Vergleich zu den Ergebnissen der *Normale Verteilung* Messung ist erkennbar, dass sowohl bei statischem als auch bei dynamischem Scheduling die Ausführungszeit schlechter skaliert. So liegt der Beschleunigungsfaktor für statisches Scheduling 29,8% (EV_N) und 28,1% (EV_H) unter dem Optimum und für dynamisches Scheduling 21,6% (EV_N) und 16,8% (EV_H). Dieser Abstand erscheint erst ab einer Anzahl von $n = 7$ und ergibt sich aus den wenigen Tasks, die einen sehr hohen Anteil an der Arbeitslast berechnen. Während bei den *NV* Szenarien die Arbeitslast durch das Workstealing so neu verteilt werden kann, dass alle Rechenknoten in etwa einen gleich großen Anteil an der Arbeitslast l berechnen, ist dies bei der extremen Verteilung der Arbeitslast in den *EV* Szenarien ab einer gewissen Anzahl von Rechenknoten (hier $n > 6$) nicht möglich, da die Arbeitslast der wenigen Tasks mit hoher Arbeitslast nicht weiter parallelisiert werden kann. Somit ergibt sich ein Systemzustand, in dem einige Rechenknoten noch mit der Ausführung von Tasks beauftragt sind, während andere Rechenknoten keine weiteren Workitems mehr zur Ausführung vorliegen haben und deren Rechenkapazitäten nicht genutzt werden können.

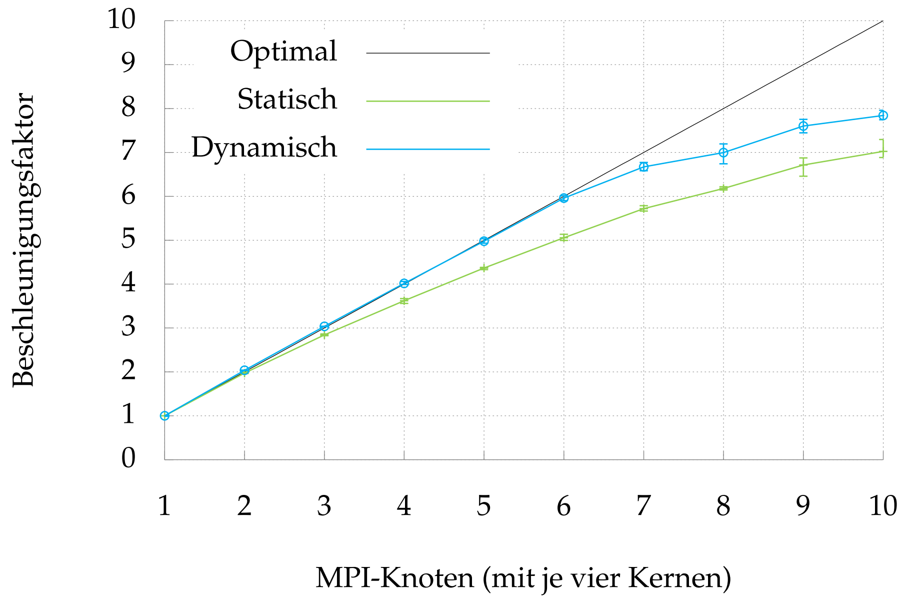


Abbildung 7.5: Speedup-Graph der EV_N Messung.

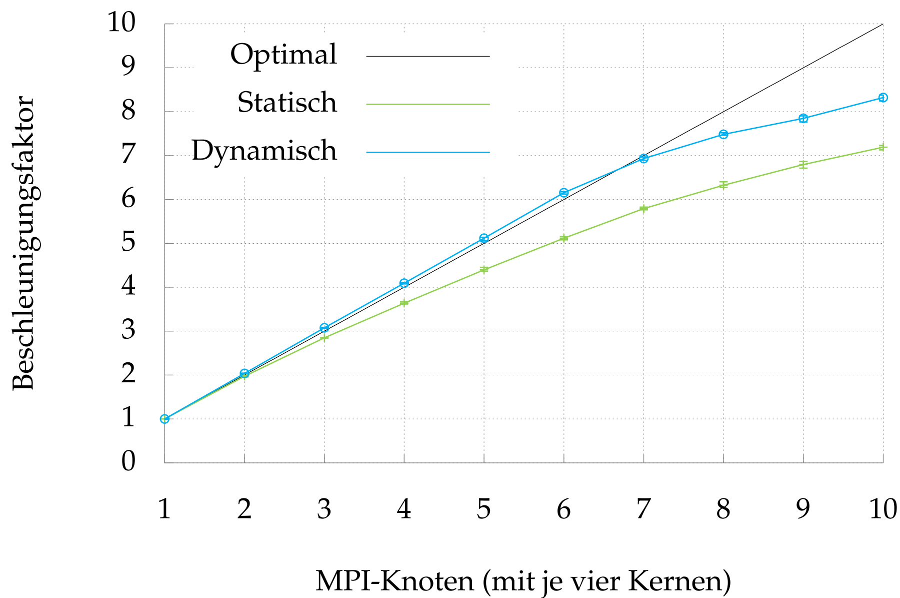


Abbildung 7.6: Speedup-Graph der EV_H Messung.

7.3.3 Damenproblem

Im Rahmen der Evaluation wird das *Damenproblem* für ein Schachbrett der Größe 14×14 Felder verwendet (im Folgenden als DP_{14} bezeichnet). Dabei werden die ersten $z = 3$ Zei-

len nebenläufig durch Child-Tasks berechnet (was einer Menge von 2.744 Tasks entspricht) die jeweils für eine Damen-Konstellation der ersten drei Zeilen die möglichen Damen-Positionierungen der verbleibenden elf Zeilen durchprobieren. Alle Tasks haben somit eine vergleichbare Arbeitslast zu bewältigen. Anschließend wird von jedem Task die Anzahl der gefundenen validen Platzierungen an den Parent-Task übergeben, der die Summe über alle Ergebnisse bildet und zu dem Ergebnis 365.596 kommt. Der Beschleunigungsgraph für die Messung DP_{14} (in Abhängigkeit von der Anzahl verwendeter MPI-Knoten) ist in Abbildung 7.7 dargestellt.

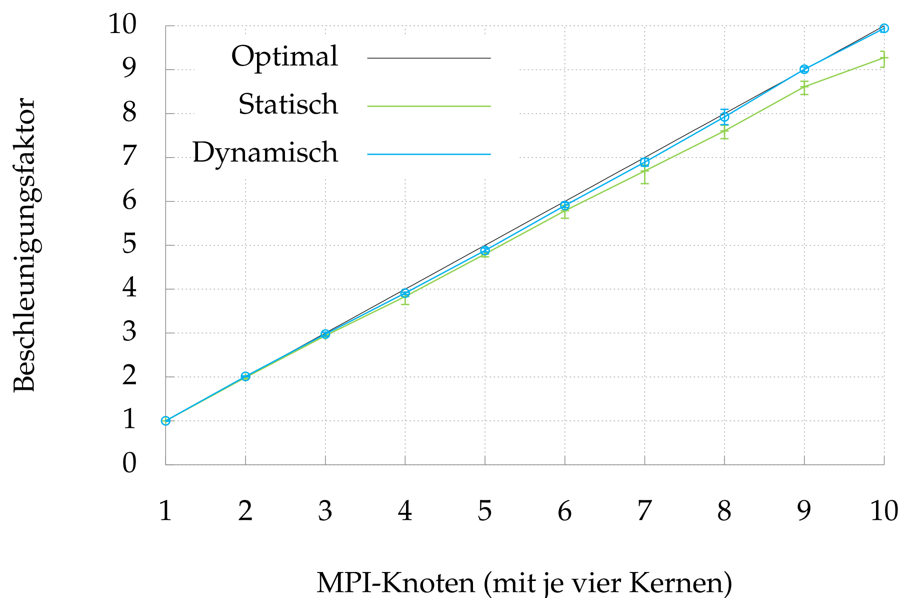


Abbildung 7.7: Speedup-Graph der DP_{14} Messung.

Wie die vorangegangenen Szenarien, skaliert auch bei der Messung des Damenproblems die Implementierung mit dynamischem Scheduling besser als die Implementierung mit statischem Scheduling. Konkret liegt der Beschleunigungsfaktor für $n = 10$ bei statischem Scheduling 7,3% unter dem Optimum, während mit dynamischem Scheduling der optimale Wert knapp erreicht wird ($-0,6\%$). Das Damenproblem skaliert sehr gut, da sich die Arbeitslast gleichmäßig auf die verfügbaren Ressourcen aufteilen lässt. Die Kommunikation zwischen den Tasks beschränkt sich dabei auf das Starten der Tasks sowie die Übertragung der Ergebnisse. Auch die zusätzlichen Kosten für die Zusammenführung der Ergebnisse sind durch die spezialisierten Datenstrukturen sehr gering. Hier ist erkennbar, dass spezialisierte Datenstrukturen, die auf komplexe Mechanismen zur Zusammenführung verzichten, genutzt werden können, um die Berechnungskomplexität der Verteilung zu reduzieren, sofern anwendungsspezifisches Wissen die Nutzung solcher Datenstrukturen ermöglicht.

7.3.4 Game of Life

Das *Game of Life* Szenario wird für die Evaluation der Skalierbarkeit mit einer Konfiguration für eine niedrige Arbeitslast (GoL_N) und eine für hohe Arbeitslast ausgeführt (GoL_H). Im Falle von GoL_N wird ein Gitter der Größe 500×500 durch $t = 100$ Tasks simuliert. Im Falle von GoL_H hat das Gitter eine Größe von 4.000×4.000 Zellen, die ebenfalls durch $t = 100$ Tasks simuliert werden. Die Anzahl der Simulationsschritte entspricht dabei jeweils $s = 20$. Die Ergebnisse für die Messungen sind in den Abbildungen 7.8 (GoL_N) und 7.9 (GoL_H) dargestellt.

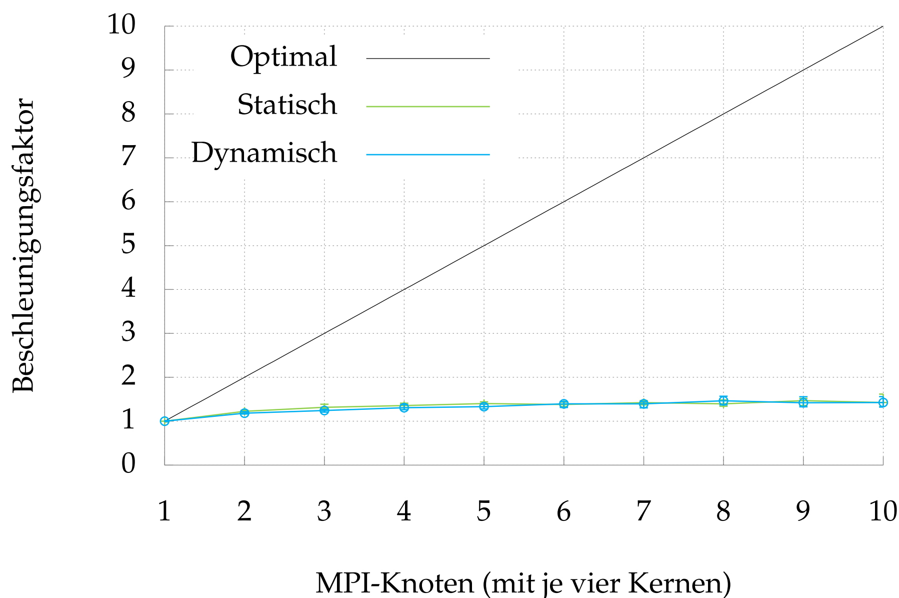


Abbildung 7.8: Speedup-Graph der GoL_N Messung.

An den Graphen für die Messungen GoL_N und GoL_H ist erkennbar, dass die Anwendung nicht mit den hinzugefügten Ressourcen skaliert (der Abstand des Beschleunigungsfaktors zum Optimum liegt hier bei $85,8\%$ für GoL_N und $84,3\%$ für GoL_H). Das liegt unter anderem an den häufigen Aufrufen der `Sync`-Primitive nach jedem Simulationsschritt. Daraus ergibt sich eine sehr hohe Anzahl an Synchronisationspunkten, die abhängig von der Anzahl der Simulationsschritte s, sowie der Anzahl der Child-Tasks (bzw. der Teilgitter) t ist. Das Spawn & Merge Programmiermodell ist dazu gedacht, die Synchronisationspunkte innerhalb von Anwendungen zu reduzieren, um so eine bessere Skalierbarkeit zu erreichen. Dies ist jedoch nicht für jedes Szenario möglich. Im Kontext des *Game of Life* Szenarios zeigt sich dies sehr deutlich. Damit eine skalierende Ausführung der Anwendung möglich ist, muss der Performancegewinn durch die parallele Simulation der Teilgitter die Kosten für die häufige Synchronisation übersteigen. Dies ist allerdings nicht gegeben, da die

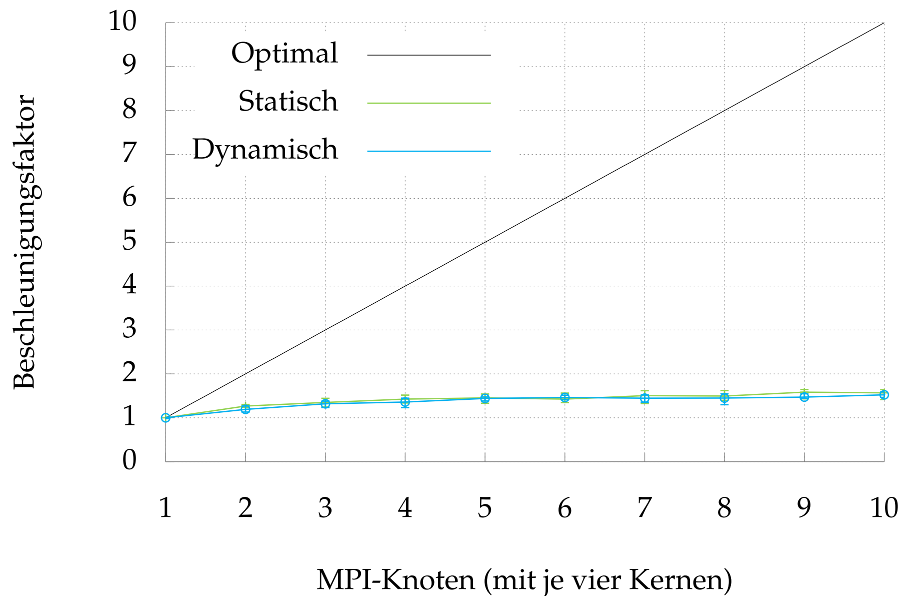


Abbildung 7.9: Speedup-Graph der GoL_H Messung.

Simulationskomplexität für einzelne Simulationsschritte sehr gering ausfällt. Dies spiegelt sich auch in der schlechten Skalierung bei einer Erhöhung der Rechenknoten wieder. Um diese Annahme zu verifizieren, wird eine zusätzliche Messung des *Game of Life* Szenarios durchgeführt, bei dem die Komplexität der Simulationsschritte künstlich erhöht wurde.

Das Ergebnis für die Messung GoL_X ist in Abbildung 7.10 zu sehen. Für $n = 10$ liegt hier der Beschleunigungsfaktor 53,5% (statisches Scheduling) und 56,7% (dynamisches Scheduling) unter dem Optimum, was einer deutlichen Verbesserung im Vergleich zu den Ergebnissen der GoL_N und GoL_H Messungen entspricht. Diese Unterschiede ergeben sich daraus, dass es sich bei den Kosten für die Synchronisation um nicht parallelisierbare Anteile der Anwendung handelt, während die Komplexität der Simulationsschritte parallelisierbar ist. Bei einer Verteilung der Simulation kann dementsprechend nur die Zeit für die Durchführung der Simulationsschritte beschleunigt werden. Übersteigen die nicht parallelisierbaren Kosten für die Verteilung die parallelisierbaren Anteile der Anwendung, so kann die Anwendung nicht effizient skalieren.

7.3.5 Skalierbarkeit von Spawn & Merge

Die Ergebnisse der Messungen in diesem Kapitel zeigen, dass sich Anwendungen mit einem hohen parallelisierbaren Anteil effizient mit dem Spawn & Merge Programmiermodell verteilen lassen. Anwendungen, die wenig Synchronisation zwischen den einzelnen Tasks benötigen (z.B. das *Damenproblem*) skalieren dabei erwartungsgemäß besser als An-

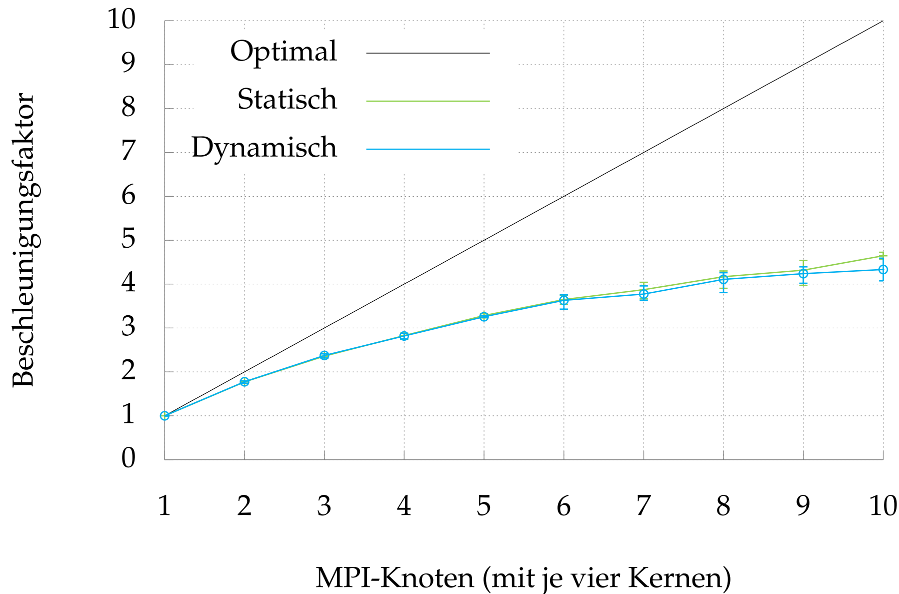


Abbildung 7.10: Speedup-Graph der GoL_X Messung.

wendungen, die eine häufige Synchronisation zwischen Tasks erfordern (z.B. das *Game of Life* Szenario). Je größer der parallelisierbare Anteil, desto weniger fallen dabei die nicht parallelisierbaren Teile der Anwendung ins Gewicht. Dies wurde durch den Vergleich der GoL_H Messung mit der GoL_X Messung gezeigt. Das zeigt sich auch an den Abständen der gemessenen Szenarien zum theoretischen optimalen Beschleunigungsfaktor, die in Tabelle 7.1 für $n = 10$ Rechenknoten aufgelistet sind. Hier ist erkennbar, dass (vor allem bei dynamischem Scheduling) die Konfigurationen mit einer höheren Arbeitslast besser skalieren (einen geringeren Abstand zum Optimum haben) als Konfigurationen mit einer niedrigeren Arbeitslast. So reduziert sich beispielsweise der Abstand des Beschleunigungsfaktors für das *Normale Verteilung* Szenario von 6,6% auf 0% und für das *Extreme Verteilung* Szenario von 21,6% auf 16,8%. Das liegt (wie beschrieben) daran, dass der nicht parallelisierbare Anteil der Anwendung konstant bleibt.

	Abstand zur optimalen Skalierung (für $n = 10$)							
	NV_N	NV_H	EV_N	EV_H	DP_{14}	GoL_N	GoL_H	GoL_X
Statisch	18,7%	17,3%	29,8%	28,1%	7,3%	85,8%	84,3%	53,5%
Dynamisch	6,6%	0%	21,6%	16,8%	0,6%	85,7%	84,8%	56,7%

Tabelle 7.1: Abstand der gemessenen Beschleunigung zum Optimum (in Prozent).

Beim Vergleich der Laufzeiten für die einzelnen Szenarien mit statischem und dynamischem Scheduling ist für die Szenarien *Normale Verteilung*, *Extreme Verteilung* und

das *Damenproblem* eine Verbesserung der Laufzeit zu erkennen. So verringert sich der Abstand des Beschleunigungsfaktors beispielsweise für das *Normale Verteilung* Szenario von 18,7% auf 6,6% und für das *Damenproblem* von 7,3% auf 0,6%. Zusätzlich wird der Speedup-Graph geglättet, was sich in den Messungen dadurch bemerkbar macht, dass die Messwerte bei dynamischem Scheduling eine geringere Standardabweichung aufweisen. Tabelle 7.2 zeigt die Standardabweichungen der unterschiedlichen gemessenen Szenarien und Konfigurationen. Für die Szenarien *Normale Verteilung*, *Extreme Verteilung* und das *Damenproblem* wird bei hoher Arbeitslast[6] bei einem Wechsel zu dynamischem Scheduling die Standardabweichung zwischen 35,8% und 50,3% reduziert. Das liegt daran, dass mit dynamischem Scheduling eine ungleichmäßige Verteilung der Arbeitslast auf die verfügbaren Rechenknoten zur Laufzeit an die aktuellen Begebenheiten angepasst werden kann (siehe Workstealing in Kapitel 6.2.3). Bei statischem Scheduling ist es der Anwendung hingegen nicht möglich, die Verteilung der Arbeitslast zur Laufzeit anzupassen.

	Standardabweichung in Millisekunden							
	NV_N	NV_H	EV_N	EV_H	DP_{14}	GoL_N	GoL_H	GoL_X
Statisch	10,6	231,8	13,6	159,6	87,7	26,0	1.089,7	889,4
Dynamisch	11,4	115,2	13,1	102,5	53,3	29,1	854,2	927,5
Differenz	+7,5%	−50,3%	−3,7%	−35,8%	−39,2%	+11,9%	−21,6%	+4,3%

Tabelle 7.2: Veränderung der Standardabweichungen abhängig vom Scheduling.

Für die *Game of Life* Szenarien hingegen gibt es keinen eindeutig positiven Einfluss durch dynamisches Scheduling. So verändert sich für die GoL_N Messung der Abstand zum theoretischen Optimum von 85,8% zu 85,7%, für die GoL_H Messung von 84,3% zu 84,8% und für die GoL_X Messung von 53,5% zu 56,7%. Das liegt daran, dass alle Child-Tasks im *Game of Life* Szenario dieselbe Berechnungskomplexität besitzen und dementsprechend ein statisches Scheduling auf die verfügbaren Rechenknoten bereits optimal ist. Die Berechnungskomplexität, die der dynamische Scheduler hier der Anwendung hinzufügt, kann dementsprechend nicht zu einer Verbesserung der Ausführungszeit führen, sondern führt zu einer Verlangsamung der Anwendung. Dieser Umstand kann auch bei den weiteren Szenarien für $n = 1$ Rechenknoten nachvollzogen werden, wo ein dynamisches Scheduling keine bessere Verteilung als ein statisches Scheduling erreichen kann (z.B. NV_H 112,5 Sekunden (statisch) im Vergleich zu 113,6 Sekunden (dynamisch)). Die Standardabweichung der Messwerte der *Game of Life* Konfigurationen verbessert sich dementsprechend ebenfalls nicht deutlich, da durch die bereits optimale Verteilung der Workitems auf die Rechenknoten keine Verbesserung erzielt werden kann.

[6] Die Veränderungen bei niedriger Arbeitslast sind hier nicht aussagekräftig, da sich die Werte nur weniger als eine Millisekunde unterscheiden.

Die Messungen in diesem Kapitel zeigen, dass auf Spawn & Merge basierende Anwendungen skalieren können und das dynamische Scheduling dabei die Performance (insbesondere bei nicht homogen verteilten Arbeitslasten) signifikant verbessert. Wie gut diese skalieren hängt dabei zum einen vom Verhältnis der nicht parallelisierbaren Anteile zu den parallelisierbaren Anteilen der Anwendung ab. Die nicht parallelisierbaren Anteile beinhalten dabei auch die Synchronisationsmechanismen (d.h. die Verteilungskosten und die Kosten für die deterministische Zusammenführung). Je höher der parallelisierbare Anteil, desto weniger fallen diese statischen Kosten für die Laufzeit und die Skalierbarkeit ins Gewicht. Eine genauere Aufschlüsselung der Kosten für die Synchronisationsmechanismen in Abhängigkeit der Anwendung folgt in Kapitel 7.5.

7.4 Lokale Optimierungen

In diesem Kapitel wird untersucht, inwieweit sich die in Kapitel 6.2.6 beschriebenen Optimierungen, die für den Fall einer Ausführung des Prototyps auf einem einzelnen MPI-Knoten implementiert wurden, auf die Laufzeit der unterschiedlichen Szenarien auswirken. Ziel dieser Optimierungen ist es, dem Entwickler einer auf Spawn & Merge basierenden Anwendung das Testen auf einem einzelnen Entwicklungsrechner zu erleichtern.

Zu den implementierten Optimierungen zählen der Verzicht auf den SchedulingMaster, der Verzicht auf das Serialisieren und Deserialisieren von Nachrichten und die Umsetzung von *copy-on-write* Mechanismen. Letztere ermöglichen es dem Framework auf das Kopieren einer übergebenen Datenstruktur zu verzichten, solange sowohl der Parent-Task als auch der Child-Task nur lesend auf die entsprechende Datenstruktur zugreifen. Der Einfluss dieser Optimierungen ist dabei abhängig davon, wie groß der Anteil eben jener umgangener Mechanismen an der Gesamtlaufzeit der Anwendung bei einer Ausführung auf einem MPI-Knoten ist.

Gemessen wird die Ausführungszeit der unterschiedlichen Szenariokonfigurationen, die bereits in Kapitel 7.3 verwendet wurden. Die Messung wird dabei sowohl mit den genannten Optimierungen als auch ohne die Optimierungen auf einem einzelnen Rechenknoten durchgeführt.

In Abbildung 7.11 sind die Ergebnisse der Messungen in Form eines Säulendiagramms abgebildet. Die grünen Säulen geben die Ausführungszeit der verschiedenen Szenariokonfigurationen ohne Optimierungen (für die Ausführung auf einem einzelnen Rechenknoten) als Referenzwert wieder. Die blauen Säulen geben in Form eines Prozentwertes an, wie sich die Ausführungszeit durch die Aktivierung der Optimierungen verändert hat (im Vergleich zur Ausführung ohne Optimierungen).

Für die Szenarien mit normaler Verteilung der niedrigen und hohen Arbeitslast (NV_N und NV_H) sowie für die entsprechenden Szenarien mit extremer Verteilung der Arbeitslas-

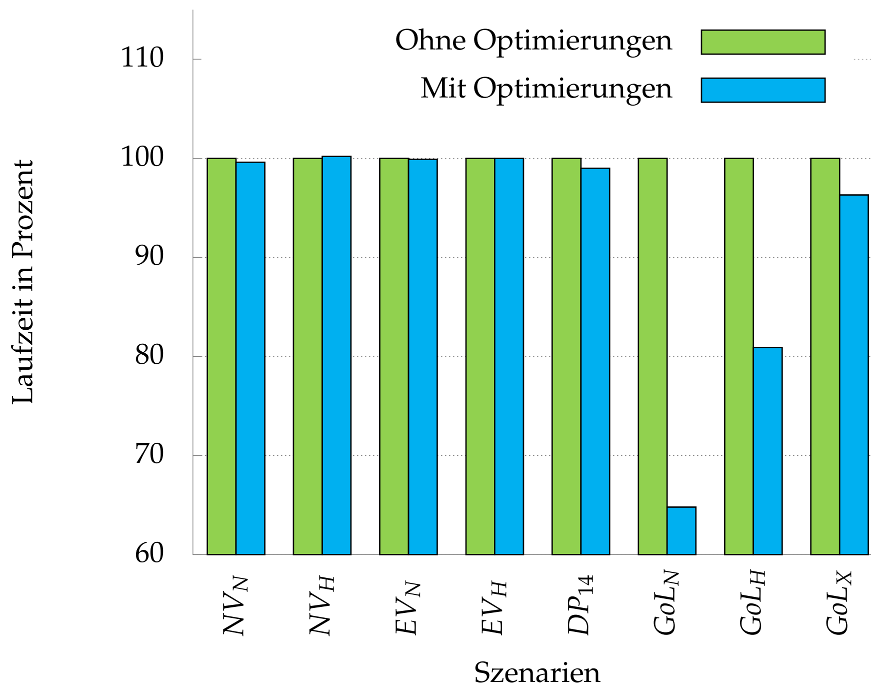


Abbildung 7.11: Einfluss der Optimierungen für lokale Ausführung.

ten (EV_N und EV_H) ist zu erkennen, dass sich die Ausführungszeit nur in der Größenordnung von ±1% verändert, was innerhalb der generellen Schwankung der Ausführungszeit liegt. Selbiges trifft auf die Ausführungszeit des Damenproblem-Szenarios für ein Schachbrett der Größe 14×14 zu (DP_{14}).

Das Ausbleiben sichtbarer Auswirkungen der Optimierungen hat unterschiedliche Ursachen. Das statische Scheduling hat einen so geringen Overhead, dass das Abschalten des Scheduling-Mechanismus keinen Performancegewinn erreichen kann. Auch der Anteil der Serialisierung und der Deserialisierung an der Programmlaufzeit ist, verglichen mit den Kosten für das Kopieren der Datenstrukturen, sehr gering. In allen Szenarien werden die übergebenen Datenstrukturen verändert, sodass sie früher oder später kopiert werden müssen und auch die *copy-on-write* Mechanismen hier keinen Performancegewinn erzielen können.

Bei den Ergebnissen der Game of Life Messungen mit niedriger und hoher Arbeitslast (GoL_N und GoL_H) ist eine Reduzierung der Ausführungszeit um 35% (GoL_N) bzw. 19% (GoL_H) erkennbar. Diese Einsparung ergibt sich aus dem Verzicht auf das Serialisieren und Deserialisieren der übergebenen Datenstrukturen. Wie bereits in Kapitel 7.3.4 beschrieben, beinhalten die GoL_N und GoL_H Messungen einen hohen Anteil nicht parallelisierbarer Synchronisationen, da jeder Task seine Zwischenergebnisse nach jedem Schritt

durch den Aufruf der `Sync`-Primitive mit dem Parent-Task zusammenführen muss. Zusätzlich bestehen die zusammenführbaren Datenstrukturen hier wiederum aus einzelnen zusammenführbaren Zellen. Diese haben einen binären Wert (*lebendig* oder *tot*).

Für die Mechanismen zur Zusammenführung müssen allerdings viele zusätzliche Informationen gespeichert werden (z.B. die Operationen und Versionsnummern). Somit ergibt sich für jede der Zellen, im Vergleich mit den eigentlichen Daten (hier die Lebendigkeit der Zelle) der Datenstruktur, ein hoher Overhead an gespeicherten Informationen. Der parallelisierbare Anteil (die Durchführung der Simulation) ist hingegen sehr gering. Der Verzicht auf die Serialisierung und Deserialiserung bringt somit Laufzeiteinsparungen im Bereich der nicht parallelisierbaren Synchronisationsmechanismen mit sich. Da diese Synchronisationsmechanismen bei GoL_N und GoL_H einen hohen Anteil ausmachen, fallen die Vorteile durch die Optimierung für eine lokale Ausführung stärker ins Gewicht. Hierbei ist zu beachten, dass beide Szenarien nicht skalieren. Bei einer Betrachtung der zusätzlichen GoL_X Messung ist zu sehen, dass die Reduzierung der Ausführungszeit mit $3,7\%$ wieder geringer ausfällt, da sich das Verhältnis von nicht parallelisierbaren Anteilen zu parallelisierbaren Anteilen verändert hat.

Die Messergebnisse zeigen auf, dass die Vorteile durch die Optimierungen für eine Ausführung auf einem einzelnen Rechenknoten abhängig von den nicht parallelisierbaren Anteilen der Anwendung sind. Je geringer der Anteil der nicht parallelisierbar ist, desto geringer der Einfluss der Optimierungen für eine lokale Ausführung. Das liegt insbesondere daran, dass die Optimierungen nur für die Synchronisationsmechanismen greifen. Wie stark die einzelnen Optimierungen die Laufzeit beeinflussen, ist abhängig von der Anwendung. So kann der *copy-on-write* Mechanismus nur dann Rechenzeit einsparen, wenn es in der Anwendung übergebene Datenstrukturen gibt, die sowohl vom Parent-Task als auch vom Child-Task nur gelesen werden. Der Wegfall der Serialisierung und Deserialisierung sorgt des Weiteren für eine signifikante Reduzierung der Ausführungszeit bei Anwendungen, bei denen viele Daten zwischen den Tasks übertragen werden müssen, sei es durch das Starten und Fertigstellen von Tasks oder durch die Übertragung von Zwischenergebnissen über die `Sync`-Primitive. Die Optimierung durch das Abschalten des Scheduling-Mechanismus ist in jedem Falle verschwindend gering, da das statische Scheduling bereits nur aus der Rückgabe und dem Inkrementieren eines einzelnen `Integer`-Wertes besteht. Dadurch, dass die Optimierungen keinen negativen Einfluss auf die Programmlaufzeit haben, dafür aber einen positiven Einfluss haben *können*, ist die Verwendung der Optimierungen dennoch sinnvoll.

7.5 Determinismuskosten

In diesem Kapitel soll untersucht werden, was die Verwendung der Mechanismen, die eine deterministische Ausführung auf Applikationsebene im Kontext von Spawn & Merge ermöglichen, auf die Performance bezogen kostet. Die Erkenntnisse dieses Kapitels sollen Entwicklern als Anhaltspunkte dafür dienen, ob sich eine Anwendung für die Umsetzung mit dem Spawn & Merge Programmiermodell eignet oder nicht.

Eine direkte Bezifferung der Determinismuskosten im Sinne der Frage „Wie hoch ist der Performanceverlust, wenn ich die nichtdeterministische Anwendung X mit Spawn & Merge deterministisch ausführen möchte?“ ist nicht möglich, da das Spawn & Merge Framework nicht die deterministische Ausführung *beliebiger* Anwendungen ermöglicht. Bei Verwendung von Spawn & Merge muss eine Anwendung entsprechend an das Spawn & Merge Programmiermodell angepasst werden. Somit ist ein direkter Vergleich mit der Laufzeit nichtdeterministischer Anwendungen, wie es beispielsweise bei *DDoS* [52] durchgeführt wurde, nicht aussagekräftig.

Aus diesem Grund werden hier die Komponenten einer auf Spawn & Merge basierenden Anwendung, die die deterministische Ausführung der Anwendung ermöglichen, in Bezug auf ihre Kosten untersucht. Zu diesen Komponenten zählen die Operational Transformation, die Wartebedingungen zur Einhaltung der deterministischen Reihenfolge für die Zusammenführung von Child-Tasks und die Kosten, die mit der Verteilung der Anwendung auf mehrere Rechenknoten einhergehen[7]. Die Determinismuskosten werden somit als Summe der Kosten der einzelnen Komponenten betrachtet, die eine deterministische Ausführung ermöglichen. Die Skalierbarkeit des Frameworks wurde bereits anhand unterschiedlicher Szenarien in Kapitel 7.3 gezeigt.

Im Folgenden wird insbesondere der Einfluss unterschiedlicher Eigenschaften der Anwendung, die Auswirkungen auf die Kosten der einzelnen Komponenten haben können, evaluiert, um Entwicklern eine Entscheidungsgrundlage für die Frage, ob sich das Spawn & Merge Framework für eine bestimmte Anwendung eignet, zu bieten.

7.5.1 Operational Transformation Kosten

In diesem Kapitel werden die Kosten für die Durchführung einer Konfliktauflösung mittels Operational Transformation für unterschiedliche Anwendungstypen ermittelt. Dazu wird der Anteil der Operational Transformation Durchführung an der Ausführungszeit für unterschiedliche Konfigurationen des *Operational Transformation* Szenarios gemessen (siehe Kapitel 7.2.5).

Die Konfigurationen verwenden dabei als OT-System s zum einen das originale TP1-OT-

[7] Dabei ist zu beachten, dass die Verteilungskosten nicht spezifisch für Spawn & Merge sind, sondern bei jeder verteilten Anwendung anfallen.

System für Listen, das in Kapitel 5.1 vorgestellt wurde, und zum anderen das angepasste OT-System, das in Kapitel 5.3 vorgestellt wurde, um eine Zusammenführung in beliebiger Reihenfolge zu ermöglichen. Damit wird evaluiert, inwiefern sich die Veränderungen des angepassten OT-Systems (d.h. die Einführung von Tombstones und die Anpassung der Kontrollfunktion) auf dessen Komplexität in unterschiedlichen Kontexten ausgewirkt haben.

Um unterschiedliche Anwendungstypen abbilden zu können, wird die Anzahl n der Elemente, die dem Child-Task übergeben wird, zwischen den Einstellungen *leer* (0 Elemente), *mittel* (1.000 Elemente) und *viele* (1.000.000 Elemente) variiert. So können Anwendungen abgebildet werden, die unterschiedlich große Datenstrukturen übergeben bekommen. Des Weiteren wird die Anzahl o der von beiden Tasks durchgeführten Operationen je nach Konfiguration auf *wenige* (50 Operationen), *mittel* (50.000 Operationen) oder *viele* (200.000 Operationen) gesetzt. So kann zusätzlich in die Konfiguration mit einbezogen werden, wie stark eine Datenstruktur von einem Task nebenläufig modifiziert wird. Zusätzlich berechnen beide Tasks wieder eine niedrige Arbeitslast von $l \approx 5,08 \cdot 10^{10}$ CPU-Zyklen oder eine hohe Arbeitslast von $l \approx 1,016 \cdot 10^{12}$ CPU-Zyklen.

Die Messung wird auf zwei MPI-Knoten ausgeführt, sodass je ein Task auf einem der MPI-Knoten ausgeführt wird. Das führt dazu, dass die geteilte Datenstruktur von einem MPI-Knoten auf einen anderen MPI-Knoten übertragen werden muss, was sich bei großen Datenstrukturen auf die Laufzeit auswirken kann und dementsprechend in der Messung abgebildet werden soll. Für jede Konfiguration wird die Ausführungszeit mit und ohne Operational Transformation gemessen (siehe Kapitel 7.2.5).

Größe	Ops	s = *Originales OT-System* Laufzeit in Sek. (OT-Anteil)				s = *Angepasstes OT-System* Laufzeit in Sek. (OT-Anteil)			
n	o	l = hoch		l = niedrig		l = hoch		l = niedrig	
0	50	131,5	(0,8%)	6,6	(0%)	130,4	(0,5%)	6,5	(0%)
	50k	139,0	(7,0%)	15,6	(58,3%)	139,2	(6,8%)	15,0	(54,7%)
	200k	275,4	(53,0%)	152,0	(95,7%)	263,7	(48,3%)	139,4	(91,2%)
1.000	50	130,2	(0,5%)	6,5	(0%)	130,0	(0,3%)	6,5	(0%)
	50k	145,6	(11,2%)	22,2	(70,7%)	145,9	(11,0%)	22,3	(69,1%)
	200k	381,0	(66,0%)	256,9	(97,5%)	377,8	(64,2%)	254,7	(95,2%)
1.000.000	50	130,9	(0,5%)	6,7	(1,5%)	130,2	(0,5%)	6,7	(0%)
	50k	153,9	(13,0%)	30,2	(63,6%)	186,2	(14,3%)	62,7	(41,6%)
	200k	402,9	(63,4%)	279,2	(91,3%)	565,9	(50,1%)	447,6	(64,3%)

Tabelle 7.3: Laufzeiten der OT-Szenarien (mit Angabe des OT-Anteils).

Die Ergebnisse der Messung sind in Tabelle 7.3 dargestellt, wobei der Anteil der Operational Transformation Mechanismen an der gesamten Laufzeit für die unterschiedlichen

Messungen jeweils in Klammern mit angegeben ist. Generell spiegelt sich im stark ansteigenden OT-Anteil an der Laufzeit der Messungen in Abhängigkeit der durchgeführten Operationen o die Laufzeitkomplexität von $\mathcal{O}(n^2)$ für beide OT-Systeme wieder. So erhöht sich der Anteil der Operational Transformation überproportional schnell und vervierzehnfacht sich im Mittel ($\approx 14,17\times$) bei einer Vervierfachung der Operationen (von 50.000 auf 200.000). Für Listen mit $n = 0$ und $n = 1.000$ Elementen ist die Laufzeit der Messungen des angepassten OT-Systems geringer als die Laufzeit mit dem originalen OT-System. Das ändert sich allerdings, sobald die zu modifizierende Liste besonders viele Elemente beinhaltet. So macht sich bei $n = 1.000.000$ Elemente die in Kapitel 5.3.3 beschriebenen zusätzliche Komplexität für das Durchführen von Operationen auf der Liste im angepassten OT-System durch eine längere Laufzeit bemerkbar ($20,99\%$ bis $107,62\%$ langsamer für $o = 50.000$ bis $o = 200.000$ Operationen), die sich aus der Einführung der Tombstones und die damit einhergehende komplexere Index-Suche ergibt ($\mathcal{O}(n)$ für jede Operation). Dass sich die zusätzliche Laufzeit nicht durch die angepasste Kontrollfunktion ergibt, ist dadurch ersichtlich, dass der OT-Anteil der entsprechenden Messungen an der Laufzeit deutlich geringer ist, als es bei der vergleichbaren Messung des originalen OT-Systems der Fall ist ($50,1\%$ statt $63,4\%$ und $64,3\%$ statt $91,3\%$).

Bei einer Erhöhung der Arbeitslast l ist erkennbar, dass sich der OT-Anteil reduziert. So verringert sich beispielsweise im originalen OT-System der OT-Anteil bei $n = 1.000$ und $o = 200.000$ von $97,5\%$ (niedrige Arbeitslast) zu $66,0\%$ (hohe Arbeitslast). Dies war zu erwarten, da sich die OT-Kosten aus der Größe n der Liste und der Anzahl der durchgeführten Operationen o ergibt und dabei unabhängig von der Arbeitslast l sind. Daraus ergibt sich die Schlussfolgerung, dass sich auch hohe OT-Kosten bei höheren Arbeitslasten amortisieren können, da die OT-Kosten konstant bleiben.

Die Nutzbarkeit der einzelnen OT-Systeme für eine Spawn & Merge basierte Anwendung hängt somit davon ab, wie viele Aufrufe der Operational Transformation Mechanismen im Rahmen der Anwendung zu erwarten sind. Die Anzahl der Aufrufe hängt dabei von den Synchronisationspunkten der Anwendung ab, die sich durch die verwendeten Synchronisationsprimitive ergeben. Des Weiteren ist die Komplexität der durchzuführenden Operational Transformation zu beachten. So kann eine Anwendung mit vielen Synchronisationspunkten unproblematisch sein, solange die zu erwartenden OT-Aufrufe günstig sind (wenige durchzuführende Transformationen). Gleichzeitig können auch wenige Synchronisationspunkte bei der Verwendung von OT-Mechanismen zu einer stark eingeschränkten Performance führen, falls die zu erwartenden OT-Aufrufe sehr teuer sind (viele durchzuführende Transformationen). Diese Betrachtungen müssen von einem Entwickler für jede Anwendung neu durchgeführt werden, um zu überprüfen, ob sich das Spawn & Merge Programmiermodell für die Umsetzung der entsprechenden Anwendung eignet.

7.5.2 Wartezeiten durch die deterministische Reihenfolge

Um eine deterministische Zusammenführung auch mit TP1-OT-Systemen erreichen zu können, erzwingt das Spawn & Merge Framework eine Zusammenführung in einer festen Reihenfolge. Dies kann dazu führen, dass ein Parent-Task auf die Fertigstellung eines bestimmten (langsamen) Child-Tasks warten muss, um diesen zusammenführen zu können, obwohl bereits andere Child-Tasks fertiggestellt wurden. In diesem Falle kann es passieren, dass potenziell nutzbare Rechenzeit ungenutzt verstreicht. Um dem entgegenzuwirken, wurde (wie bereits beschrieben) ein OT-System für eine Liste angepasst, um eine Zusammenführung in beliebiger Reihenfolge (bei deterministischem Ergebnis) zu ermöglichen.

In diesem Kapitel wird untersucht, inwiefern sich Wartezeiten durch das Warten auf die Fertigstellung von Child-Tasks ergeben, indem der Einfluss einer Zusammenführung in First-Come-First-Serve (FCFS) Reihenfolge auf die Programmlaufzeit gemessen wird. Wird eine Zusammenführung in FCFS Reihenfolge ermöglicht, so werden die entsprechenden Wartebedingungen vermieden. Ein Vergleich der Laufzeiten mit und ohne eine Zusammenführung in FCFS Reihenfolge ermöglicht somit eine Aussage über den Anteil der Wartezeiten an der Ausführungszeit der Anwendung.

Als Messungen werden die Szenarien NV_H, EV_H, DP_{14} und GoL_H jeweils mit einer deterministischen `Merge`-Reihenfolge und mit FCFS `Merge`-Reihenfolge auf $n = 10$ MPI-Knoten ausgeführt und die Laufzeit gemessen[8].

Tabelle 7.4 zeigt die Laufzeiten der durchgeführten Messungen zusammen mit den jeweiligen Standardabweichungen der Ausführungszeiten. Für die Konfigurationen, in denen eine Zusammenführung in FCFS Reihenfolge erlaubt war, ist zusätzlich die eingesparte Laufzeit in Prozent angegeben. In den Ergebnissen ist zu erkennen, dass die Zusammenführung in FCFS Reihenfolge eine Veränderung der Ausführungszeit von $+0,5\%$ bis hin zu $-2,6\%$ erreicht. Allerdings liegen diese Abstände fast alle im Bereich der Standardabweichung, sodass diese nicht als signifikante Verbesserungen angesehen werden können.

Wie in Kapitel 5.2.4 beschrieben, ergibt sich der Vorteil einer Zusammenführung in FCFS Reihenfolge nur in ganz spezifischen Situationen. Zum einen muss die Summe der Laufzeit der vorgezogenen Operational Transformation Durchführungen Δ_{OT} kleiner sein als die verbleibende Wartezeit Δ_T auf den (nach der logischen `Merge`-Reihenfolge) nächsten Task T_1 der zusammengeführt werden müsste (siehe Beispiel in Abbildung 7.12). In diesem Fall kann die Operational Transformation Laufzeit Δ_{OT} für die Tasks T_2–T_4, die eigentlich erst nach der Zusammenführung mit Task T_1 beginnen kann, vorgezogen und somit ein-

[8]Es wurden hier nur die Szenarien mit einer *hohen* Arbeitslast gewählt, da die Laufzeiten der Szenarien mit einer *niedrigen* Arbeitslast zu gering sind, um ein aussagekräftiges Ergebnis in Bezug auf Veränderungen der Ausführungszeit zu ermitteln.

Szenario	`Merge`-Reihenfolge	Laufzeit in Sek.	(Änderung)
NV_H	Deterministisch	$11,23 \pm 0,04$	
	FCFS	$11,28 \pm 0,02$	(+0,5%)
EV_H	Deterministisch	$13,80 \pm 0,07$	
	FCFS	$13,66 \pm 0,06$	(−1,01%)
DP_{14}	Deterministisch	$18,47 \pm 0,57$	
	FCFS	$17,99 \pm 0,49$	(−2,6%)
GoL_H	Deterministisch	$2,63 \pm 0,03$	
	FCFS	$2,60 \pm 0,01$	(−1,14%)

Tabelle 7.4: Einfluss der Zusammenführung in FCFS Reihenfolge.

gespart werden. Ist Δ_{OT} so groß, dass die Durchführung der vorgezogenen Operational Transformation länger dauert als die verbleibende Wartezeit Δ_T, so wird die Überschneidung der beiden Zeiträume von der eingesparten Zeit abgezogen. Das liegt daran, dass nur die vorgezogene Zeit eingespart werden kann.

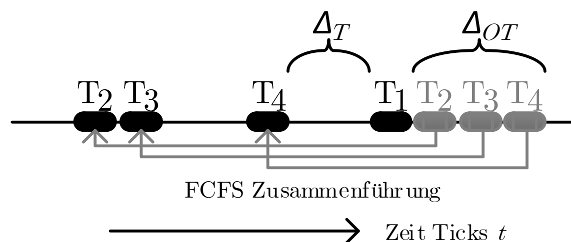


Abbildung 7.12: Mögliche eingesparte Laufzeit bei FCFS Reihenfolge.

Um ein Szenario zu testen in dem Δ_{OT} und Δ_T entsprechend groß ausfallen, wird im Folgenden die Skalierbarkeit einer zusätzlichen Konfiguration EV_X des *Extreme Verteilung* Szenarios gemessen. Dabei wird die Anzahl der gesuchten $SHA3_{256}$ Hashwerte auf $h = 500$ erhöht, um die Laufzeit der Operational Transformation Δ_{OT} zu erhöhen. Eine Arbeitslast $l \approx 4,064 \cdot 10^{11}$ CPU-Zyklen wird dabei auf $t = 800$ Tasks verteilt. Diese Konfiguration hat die Zielsetzung die Voraussetzungen zu schaffen, unter denen sich eine Zusammenführung in FCFS Reihenfolge auf die Laufzeit der Anwendung signifikant auswirkt.

Szenario	`Merge`-Reihenfolge	Laufzeit in Sek.	(Einsparung)
EV_X	Deterministisch	$62,24 \pm 0,68$	
	FCFS	$57,51 \pm 0,37$	(−7,6%)

Tabelle 7.5: Einfluss der Zusammenführung in FCFS Reihenfolge (Szenario EV_X).

Die Ergebnisse der Messung sind in Tabelle 7.5 abgebildet. Es ist zu erkennen, dass die

Aktivierung der Zusammenführung in FCFS Reihenfolge eine Reduzierung der Ausführungszeit um $7,6\%$ zur Folge hat. Diese Verbesserung der Laufzeit liegt dabei außerhalb der Schwankung der Ergebnisse.

Die Erkenntnis dieses Kapitels ist, dass die Zusammenführung in FCFS Reihenfolge für viele Anwendungen keine oder nur moderate Auswirkungen bedeutet. Das liegt daran, dass das dynamische Scheduling bereits viele Wartebedingungen auf langsame Tasks dadurch auflöst, dass in der Zwischenzeit andere noch zu berechnende Workitems vorgezogen und auf andere Rechenknoten verteilt werden können. Die Zusammenführung in FCFS Reihenfolge kann somit nur in spezifischen Situationen ihre Wirksamkeit entfalten: Zum einen muss die Laufzeit der Zusammenführung hoch sein. Zum anderen muss es einen Zeitraum geben, in dem einzelne Rechenknoten keine weiteren Berechnungen durchführen können bis ein anderer Rechenknoten einen länger laufenden Task fertiggestellt hat. In diesen Fällen kann die Rechenzeit der wartenden Tasks für die Durchführung der Zusammenführung genutzt und somit die Laufzeit der Zusammenführung eingespart werden. Da sich aus der Verwendung einer Zusammenführung in FCFS Reihenfolge keine signifikanten Nachteile für die Laufzeit einer Anwendung ergeben, ist es sinnvoll, diese Optimierung zu aktivieren, sofern die genutzten Datenstrukturen dies erlauben. So kann die Laufzeitverbesserung eintreten, falls die Anwendung eine entsprechende Wartesituation beinhaltet, während sich ansonsten keine Nachteile für die Ausführungszeit ergeben.

7.5.3 Verteilungskosten

Die *Verteilungskosten* umfassen die Mechanismen für das Kopieren von Datenstrukturen zur Bereitstellung für einen Child- oder Sibling-Task sowie die Mechanismen für die Verteilung der Anwendung (d.h. im Kontext von Spawn & Merge das Scheduling und die Nachrichtenübertragungen). Streng genommen handelt es sich bei diesen Aufwänden nicht um einen Teil der Determinismuskosten, da beide Mechanismen systembedingt in allen verteilten Anwendungen vorhanden sind. Dennoch soll hier kurz betrachtet werden, ob und wie der Einfluss entsprechender Mechanismen reduziert werden könnte, um die Kosten für eine Verteilung des Spawn & Merge Programmiermodells zu verringern.

Damit Tasks auf unterschiedlichen Rechenknoten auf geteilte Datenstrukturen zugreifen können, werden die Datenstrukturen im hier evaluierten Prototyp kopiert und zu dem entsprechenden Rechenknoten übertragen. Eine Alternative zum Kopieren und Übertragen der Daten ist das Kopieren der Daten[9] und das Vorhalten dieser Kopie auf dem ursprünglichen Rechenknoten (d.h. der Knoten des Parent-Tasks). Child-Tasks könnten somit auf die Datenstruktur zugreifen, indem die Zugriffe an den Parent-Task gesendet werden und

[9] Hierbei ist zu beachten, dass auf das Kopieren der Daten nicht verzichtet werden kann. Das liegt daran, dass der Zugriffszeitpunkt, an dem ein Child-Task auf die Datenstruktur zugreifen möchte, nichtdeterministisch ist und somit auch kein deterministisches Ergebnis für den Zugriff garantiert werden kann.

dieser das Ergebnis zurück überträgt. Auf den ersten Blick würde sich diese Herangehensweise bei großen geteilten Datenstrukturen, auf die ein Child-Task nur selten zugreifen muss, eignen. Allerdings ist dabei zu beachten, dass der Parent-Tasks zu einem Flaschenhals für den Zugriff auf die entsprechende Datenstruktur wird. Das einmalige Kopieren der Datenstrukturen für jeden Child-Task wird durch eine Kommunikation zwischen dem Parent-Task und seinen Child-Tasks für jeden Datenstrukturzugriff ersetzt, die bei vielen Anfragen dazu führen kann, dass der Parent-Task überlastet wird. Das bedeutet gleichzeitig, dass für die Modifikation der Datenstrukturen (die wie in Kapitel 5.3.3 beschrieben aufwändig ausfallen kann) nicht mehr die Parallelität und die Rechenkraft der Child-Tasks verwendet werden kann. Ein solcher Mechanismus sollte daher nur unter entsprechenden Umständen eingesetzt werden, falls im Vorhinein erkennbar ist, dass Child-Tasks selten auf sehr große Datenstrukturen zugreifen müssen.

Die Verteilung von Tasks auf verfügbare Rechenknoten bedeutet, dass eine Kommunikation zwischen den Rechenknoten sowie ein Mechanismus zur Entscheidung, wo einzelne Tasks gestartet werden sollen, erforderlich ist. Lediglich die Komplexität der Kommunikation sowie der Scheduling-Mechanismen variiert in Abhängigkeit vom gewählten Ansatz für die Verteilung. Der hier evaluierte Prototyp nutzt zur Kommunikation ein Nachrichtenprotokoll, das nur einen geringen Overhead erzeugen soll (siehe Kapitel 6.2.4). Die Nachrichten beschränken sich dazu auf die Durchführung der Synchronisationsprimitive sowie die Steuerung und das Anfragen des SchedulingMasters, falls dynamisches Scheduling genutzt wird. Bei Verwendung des statischen Schedulings werden keine Nachrichten versendet, da jeder NodeMaster eine lokale Entscheidung für das Scheduling fällt. Wird dynamisches Scheduling verwendet, so müssen zusätzliche Nachrichten versendet werden, um einen passenden MPI-Knoten für neu gestartete Tasks zu erfragen, die Work-Approximation des SchedulingMasters anzupassen oder anzufragen, ob einem anderen MPI-Knoten Workitems abgenommen werden können. In Kapitel 7.3 wurde gezeigt, dass die dadurch erreichte Reduzierung der Ausführungszeit (im Vergleich zu einer Ausführung mit statischem Scheduling) die Kosten für die zusätzlich zu versendenden Nachrichten in den meisten Fällen überwiegt. Ein Verzicht auf ein dynamisches Scheduling mit dem Ziel die Anzahl versendeter Nachrichten zu reduzieren ist somit nicht empfehlenswert.

7.6 Zusammenfassung der Ergebnisse

Die Messungen, die in diesem Kapitel durchgeführt wurden, haben gezeigt, dass verteilte Anwendungen, die auf dem Spawn & Merge Framework basieren, effizient skalieren können während eine deterministische Ausführung auf Applikationsebene garantiert wird. Dazu ist es allerdings erforderlich, dass die verteilte Anwendung einen entsprechenden parallelisierbaren Anteil hat.

Zum nicht parallelisierbaren Anteil einer auf Spawn & Merge basierenden Anwendung zählen die Kosten für die Verteilung und die Kosten für die deterministische Zusammenführung. Die Verteilungskosten sind abhängig von der Anzahl der verwendeten Rechenknoten und steigen somit bei größeren Ausführungsumgebungen. Die Verwendung des in Kapitel 6.2.4 beschriebenen, minimalen Nachrichtenprotokolls zur Umsetzung der Synchronisationsprimitive sowie das dynamische Scheduling für eine optimale Auslastung der Rechenknoten sorgen hierbei dafür, dass die Kosten für die Verteilung möglichst gering bleiben (siehe Kapitel 7.3). Die Kosten für die Mechanismen, die eine deterministische Ausführung der Anwendung ermöglichen, setzen sich aus den OT-Kosten und den Wartebedingungen zwischen Child-Tasks zusammen. Der Performanceverlust durch die Wartebedingungen zwischen Child-Tasks wird durch das dynamische Scheduling bereits zu einem Großteil verhindert (wie in Kapitel 7.5.2 zu sehen ist). Für die verbleibenden Wartebedingungen, die nur in besonderen Situationen auftreten können, können des Weiteren Mechanismen für eine Zusammenführung in beliebiger Reihenfolge genutzt werden, um auch diese aufzuheben (siehe Kapitel 7.5.2). Die OT-Kosten ergeben sich aus den Anforderungen der Anwendung. Diese Kosten können hoch ausfallen, sind allerdings für eine Anwendung konstant, unabhängig von der Ausführungsumgebung. Somit skaliert die Anwendung mit der Anzahl verfügbarer Rechenknoten bei konstanten OT-Kosten, sodass auch hohe OT-Kosten bei vielen verfügbaren Rechenknoten weniger ins Gewicht fallen.

Die Evaluation hat allerdings auch Grenzen des Systems aufgezeigt. So sollte beispielsweise nur dann das angepasste OT-System aus Kapitel 5.3 verwendet werden, wenn im Vorhinein klar ist, dass entweder keine Listen mit besonders vielen Elementen modifiziert werden müssen, oder dass eine derartige extreme Verteilung der Arbeitslast auf Tasks zu erwarten ist, dass die eingesparte Laufzeit durch die Zusammenführung in FCFS Reihenfolge die Nachteile durch die höhere Komplexität der Modifikation der Datenstruktur aufwiegen kann. Des Weiteren hat die Umsetzung des *Game of Life* Szenarios gezeigt, dass spezialisierte Datenstrukturen nicht notwendigerweise die optimale Lösung für ein Problem sind. Bei spezialisierten Datenstrukturen muss neben der Komplexität der Zusammenführung auch das Kosten-Nutzen-Verhältnis in Bezug auf den benötigten Arbeitsspeicher beachtet werden. So sollte eine spezialisierte Datenstruktur beispielsweise nicht ein Vielfaches des Arbeitsspeichers benötigen, den die grundlegende Datenstruktur vor der Spezialisierung verbraucht hat (siehe die zusammenführbaren Zellen in Kapitel 7.4). Dies gilt insbesondere dann, wenn ein einzelner Parent-Task viele Kopien dieser Datenstruktur an Child-Tasks übergibt und für die Zusammenführung anschließend alle Kopien wieder gleichzeitig auf dem eigenen Rechenknoten vorhalten muss (wie die zusammenführbaren Zellen im *Game of Life* Szenario).

Kapitel 8

Fazit

In dieser Arbeit wurde die Hypothese untersucht, dass eine deterministische und somit reproduzierbare Ausführung einer verteilten Anwendung derart realisiert werden kann, dass geringere Performancekosten entstehen als bei einer voll-deterministischen Ausführung und dass die entsprechende Anwendung mit den verfügbaren Ressourcen skalieren kann. Zur Untersuchung der Hypothese wurde das Programmiermodell Spawn & Merge für deterministische nebenläufige Anwendungen entwickelt und anschließend anhand eines Prototyps evaluiert.

Zuerst wurde dazu das Konzept des *Determinismus auf Applikationsebene* eingeführt. Das Konzept baut auf der Erkenntnis auf, dass die voll-deterministische Ausführung einer Anwendung auch diejenigen Codebereiche in deterministischer Reihenfolge ausführt, deren nebenläufige Ausführung keinen Einfluss auf das deterministische Ergebnis der Anwendung haben. Dies führt zu einem vermeidbaren Performanceverlust. Eine Beschränkung der Determinismusgarantien auf die deterministische Ausführung der Anwendungslogik (Applikationsebene) stellt eine effizientere Alternative zu vollem Determinismus dar. Sie garantiert dabei die Reproduzierbarkeit sowie den Determinismus der beobachtbaren Funktionsaufrufe und Ergebnisse unabhängig von der Nebenläufigkeit und den Timings der Funktionsaufrufe. Die darunterliegende Frameworkebene erlaubt hingegen eine nicht-deterministische Ausführungsreihenfolge der Funktionsaufrufe. Diese ermöglicht, zusammen mit der Reduktion der Synchronisationspunkte durch den Verzicht auf das Sperren von Datenstrukturen, einen erhöhten Grad an Parallelität und somit die Skalierbarkeit einer deterministischen Anwendung in Abhängigkeit von verfügbaren Ressourcen.

Aufbauend auf diesem Konzept wurde das *Spawn & Merge Programmiermodell* konzipiert, das die Entwicklung von skalierenden Applikationen, deren Anwendungslogik deterministisch ausgeführt werden soll, erleichtert. Anwendungen, die auf Spawn & Merge basieren, können dabei sowohl auf Systemen mit geteiltem Arbeitsspeicher als auch auf Verteilten Systemen ausgeführt werden. Die eingeführten Synchronisationsprimitive

erlauben Entwicklern die einfache Parallelisierung einer Anwendung, die inhärent deterministisch ist, da alle gestarteten Tasks seiteneffektfrei ausgeführt werden.

Um diese Seiteneffektfreiheit zu erreichen, arbeiten gestartete Tasks auf Kopien der Datenstrukturen. An die Stelle des Sperrens der Zugriffe auf geteilte Datenstrukturen treten dabei Mechanismen zur Konfliktauflösung. Im Spawn & Merge Programmiermodell werden insbesondere Operational Transformation Systeme verwendet, um eine deterministische Auflösung von Konflikten zu ermöglichen. Konflikte können hier durch eine nebenläufige Modifikation von Datenstrukturkopien, die von derselben original Datenstruktur abstammen, entstehen. Da die Verwendung von TP1-OT-Systemen eine feste Reihenfolge für die Zusammenführung von modifizierten Datenstrukturkopien zum Erreichen eines deterministischen Ergebnisses voraussetzt und TP2-OT-Systeme mit einem großen Berechnungsmehraufwand einhergehen, wurde ein neues OT-System entwickelt. Dieses neue OT-System ermöglicht eine deterministische Zusammenführung in beliebiger Reihenfolge (First-Come-First-Serve (FCFS)). Somit sorgt es dafür, dass ein deterministisches Ergebnis (mit einer Berechnungskomplexität, die einem TP1-System entspricht) erreicht werden kann, während die Wartezeiten, die sich aus der Einhaltung einer festen Reihenfolge der Zusammenführung ergeben hätten, eingespart werden können.

Zur Überprüfung der Hypothese wurde das Spawn & Merge Programmiermodell für die Entwicklung verteilter Anwendungen prototypisch implementiert und anschließend evaluiert. Dabei wurde gezeigt, dass Anwendungen, die auf dem Spawn & Merge Programmiermodell basieren, durch die Verwendung des dynamischen Schedulings skalieren können, sofern die Anwendung einen entsprechend hohen parallelisierbaren Anteil im Vergleich zum nicht parallelisierbaren Anteil der Anwendung hat (siehe Kapitel 7.3.5). Zu dem nicht parallelisierbaren Anteil der Anwendung zählen dabei die Kosten für die Verteilung, die mit der Anzahl der Rechenknoten der Ausführungsumgebung steigen, und die Kosten für die Mechanismen, die eine deterministische Ausführung der Anwendung ermöglichen. Diese setzen sich aus den Kosten für die OT und den Wartebedingungen zwischen Child-Tasks zusammen. Im Rahmen der Evaluation wurde gezeigt, dass ein Großteil des Performanceverlustes durch die Wartebedingungen zwischen Child-Tasks durch das dynamische Scheduling bereits verhindert wird. Aus diesem Grund kann der zusätzlich eingeführte Mechanismus (für eine Zusammenführung in beliebiger Reihenfolge (FCFS)) nur in besonderen Situationen die verbleibenden Wartebedingungen auflösen. Die Höhe der OT-Kosten ergibt sich aus dem Aufbau und den Anforderungen der Anwendung und kann sehr hoch ausfallen, wenn viele Modifikationen an geteilten Datenstrukturen durchgeführt werden. Die OT-Kosten sind allerdings für eine Anwendung konstant (für feste Eingabedaten) und unabhängig von der Ausführungsumgebung. Für eine Anwendung, die auf Spawn & Merge basiert, bedeutet dies dass die Anwendung mit den verfügbaren Ressourcen skalieren kann, während die OT-Kosten konstant bleiben. So sinkt der Anteil der

OT-Kosten an der Gesamtlaufzeit bei einer steigenden Anzahl verfügbarer Rechenknoten. Das Spawn & Merge Programmiermodell eignet sich somit für Applikationen, in denen jeder Task hauptsächlich Berechnungen und/oder Netzwerkkommunikation durchführt und letzten Endes nur eine geringe Anzahl an Modifikationen an den geteilten Datenstrukturen vornimmt. Für Anwendungen, die nur einen geringen parallelisierbaren Anteil besitzen oder die eine sehr hohe Anzahl nebenläufiger Modifikationen geteilter Datenstrukturen (im Verhältnis zum parallelisierbaren Anteil der Anwendung) erfordern, kann das Spawn & Merge Programmiermodell keine Skalierbarkeit erreichen.

Die Evaluation des Prototyps zeigt, dass verteilte Anwendungen, die auf dem Spawn & Merge Framework basieren und sich für dieses eignen, effizient skalieren können, während eine deterministische Ausführung der Applikationsebene garantiert wird. Die Evaluation bestätigt somit die Hypothese dieser Arbeit. Die Kernerkenntnis dieser Arbeit ist, dass die Entwicklung skalierender deterministischer nebenläufiger Anwendungen möglich ist, sofern das Programmiermodell die Unterscheidung ermöglicht, ob die Einhaltung der Reihenfolge zweier nebenläufiger Ereignisse in einer Anwendung für ein deterministisches Ergebnis notwendig ist oder ob zugunsten der Performance auf die Einhaltung der Reihenfolge verzichtet werden kann. Die erreichte deterministische Ausführbarkeit der Anwendungen, die auf Spawn & Merge basieren, erleichtert dabei den Entwicklungsprozess und die Fehlersuche in verteilten Anwendungen. Im Entwicklungsprozess garantiert der Determinismus auf Applikationsebene, dass eine fehlerfreie Ausführung der Anwendung auf einem Testsystem gleichzeitig bedeutet, dass die Anwendung auch auf dem Zielsystem fehlerfrei ausgeführt wird. Bei der Fehlersuche werden Entwickler in gleicher Weise dadurch unterstützt, dass sich ein Fehlerfall, der auf dem Zielsystem aufgetreten ist, bei gleichen Eingabedaten auch auf einem Testsystem deterministisch reproduzieren lässt.

8.1 Weiterer Forschungsbedarf

Die Erkenntnisse dieser Arbeit können als Ausgangspunkt für zukünftige Forschungsarbeiten im Gebiet der deterministischen Ausführung nebenläufiger Anwendungen dienen, von denen zwei Möglichkeiten im Folgenden betrachtet werden: Erstens die Übertragung des Konzeptes des Determinismus auf Applikationsebene auf andere Ansätze für die deterministische Ausführung nebenläufiger Anwendungen und zweitens die Optimierung und Erweiterung des Spawn & Merge Programmiermodells.

Im Gegensatz zum Spawn & Merge Programmiermodell ermöglichen Ansätze wie *DDOS* (siehe Kapitel 2.2.3) die voll-deterministische Ausführung beliebiger Anwendungen, ohne dass eine Anpassung notwendig ist. In zukünftigen Forschungsarbeiten könnte untersucht werden, inwiefern es einer Laufzeitumgebung oder einer statischen Code-Analyse möglich ist, Aussagen darüber zu treffen, inwiefern die Einhaltung der Reihenfolge

nebenläufiger Ereignisse zum Erreichen eines deterministischen Ergebnisses notwendig ist. Falls solch eine Unterscheidung (ähnlich dem Determinismus auf Applikationsebene) für beliebige Events einer Anwendung erreicht werden könnte, so könnte damit eine Verbesserung der Performance für die deterministische Ausführung beliebiger Anwendung erreicht werden, ohne dass eine Anpassung oder Neuimplementierung der Anwendung erforderlich ist.

Zur Verbesserung der Performance von Anwendungen, die auf dem Spawn & Merge Programmiermodell basieren, könnte die Lokalität von Tasks und Daten in das Scheduling der Tasks auf Rechenknoten einbezogen werden. Als Entscheidungsgrundlage für die Zuweisung von Tasks zu Rechenknoten könnten dazu die Größe der Datenstrukturen, die übertragen werden müssen, dienen. Eine alternative Möglichkeit wäre die Einführung neuer Synchronisationsprimitive für das Starten von Task-Gruppen. Die Herausforderungen, die sich aus diesen Ansätzen ergeben, liegen darin begründet, dass eine Laufzeitabschätzung für noch nicht gestartete Tasks schwer möglich ist. Dadurch kann die Laufzeitumgebung nicht automatisiert entscheiden, ob beispielsweise die eingesparte Übertragungszeit beim Senden von x Tasks zu einem einzelnen Rechenknoten den Vorteil der parallelen Ausführung der x Tasks auf mehreren Rechenknoten überwiegt. Das Erreichen einer optimalen Ausführungszeit wird somit erschwert. Wird die Entscheidung, ob Tasks zusammen an einen einzelnen Knoten gesendet werden sollen, hingegen dem Entwickler überlassen, dann verliert eine Anwendung möglicherweise ihre Skalierbarkeit, da die Scheduling-Entscheidungen des Entwicklers nicht zwingend für jede Ausführungsumgebung optimal sind. Zukünftige Forschungsarbeiten auf diesem Gebiet könnten sich dementsprechend mit der Frage beschäftigen, inwiefern diese Herausforderungen gelöst werden können.

Um die Fehlersuche in auf Spawn & Merge basierenden Anwendungen weiter zu erleichtern, könnten Techniken aus dem Bereich der deterministischen Wiederholung verwendet werden (siehe Kapitel 2.2.1). Da jedes Workitem einen Schnappschuss aller Informationen beinhaltet, die für die Ausführung eines Tasks notwendig sind, könnte die Aufzeichnung aller Synchronisationsnachrichten (insbesondere der Workitems) dazu genutzt werden, diejenigen Tasks, bei deren Ausführung ein Fehler aufgetreten ist, neu zu starten und somit deren Verhalten zu reproduzieren. Dieser Ansatz würde die Fehlersuche insofern erleichtern, als für die Reproduktion eines Fehlers nicht die gesamte Anwendung auf einem Testsystem neu ausgeführt werden müsste, sondern nur ein Teilbaum der Task-Hierarchie.

Die Ergebnisse dieser Arbeit stellen somit einen weiteren Schritt im Forschungsgebiet der deterministischen Ausführung nebenläufiger Anwendungen dar. So wird die Entwicklung korrekter nebenläufiger Anwendungen erleichtert und eine Grundlage für weitere Forschungsarbeiten gelegt, die zukünftig eine effiziente deterministische Ausführung beliebiger Anwendungen ermöglichen könnten.

Liste der Abkürzungen

CPU Central Processing Unit

FAS Fahrerassistenzsystem

FCFS First-Come-First-Serve

GPL General Purpose Language

GPU Graphics Processing Unit

MPI Message Passing Interface

OT Operational Transformation

RPC Remote Procedure Call

STM Software Transactional Memory

TP Transformation Properties

VM Virtuelle Maschine

Glossar

Abort Die `Abort`-Primitive ermöglicht das Abbrechen eines laufenden Tasks. Beim Abbruch von *innen* entscheidet ein Task selbst, dass er abbricht und dass seine Modifikationen verworfen werden sollen. Beim Abbruch von *außen* entscheidet der Parent-Task, dass ein Child-Task abgebrochen werden soll und dass dessen Modifikationen nicht zusammengeführt werden sollen.

Abort-Flag Das *Abort-Flag* kennzeichnet einen Task als abgebrochen und kann mit der `Abort`-Primitive gesetzt werden.

Agentenbasiert Bei einem *agentenbasierter* Ansatz zur Simulation wird jeder einzelne Agent (z.B. ein einzelnes Fahrzeug und dessen Fahrassistenzsystem im Straßenverkehr) für sich selbst gesehen simuliert.

Applikationsebene Die *Applikationsebene* umfasst den Teil der Anwendung, der die Anwendungslogik beinhaltet und deterministisch ausgeführt wird. Im Kontext von Spawn & Merge geschieht die Trennung zwischen Applikationsebene und Frameworkebene durch die eingeführten Synchronisationsprimitive.

Ausführungsumgebung Die *Ausführungsumgebung* beschreibt den Computer oder das Rechencluster, auf dem eine Anwendung ausgeführt wird.

Baum Ein *Baum* ist ein azyklischer Graph. Die Knoten eines Baumes haben Kind-Knoten und/oder Eltern-Knoten. Knoten, die nur einen Eltern-Knoten besitzen werden Blattknoten genannt. Der Knoten, der nur Kind-Knoten besitzt, ist der Wurzelknoten.

Bedingungsfunktion Die *Bedingungsfunktion* ermöglicht es dem Entwickler festzulegen, unter welchen Umständen die Ergebnisse eines Child-Tasks in die Datenstrukturen eines Tasks integriert werden und wann die Ergebnisse verworfen werden sollen.

Berechnungskomplexität Unter dem Begriff *Berechnungskomplexität* versteht man die Abschätzung des Aufwandes, der für die Berechnung einer Aufgabe (meist in Abhängigkeit einer variablen Anzahl von involvierten Elementen) aufgewendet werden muss.

Blockierende Aufrufe *Blockierende Aufrufe* stoppen die weitere Ausführung des aktiven Threads bis der Aufruf abgearbeitet wurde. Ein Beispiel ist das Warten (`listen`) auf eine eingehende TCP-Verbindung.

Call by Reference Bei der Übergabe von Parametern an eine Funktion bezeichnet *Call by Reference* die Übergabe von Referenzen auf die entsprechenden übergebenen Objekte (Referenzparameter). Veränderungen an den referenzierten Objekten betreffen somit auch das originale übergebene Objekt.

Call by Value Bei der Übergabe von Parametern an eine Funktion bezeichnet *Call by Value* die Übergabe von Kopien der übergebenen Objekte (Werteparameter). Veränderungen an den Parametern betreffen somit nicht die originalen übergebenen Objekte.

Child-Task Die Bezeichnung *Child-Task* (von Task P) sagt aus, dass dieser Task von einem anderen Task P gestartet wurde. Dieser Task P wird auch als *Parent-Task* bezeichnet.

CommandQueue Der NodeMaster nutzt intern eine *CommandQueue*, in der alle noch auszuführenden Befehle gesammelt werden. Liegen aktuell keine Befehle vor, so blockiert der NodeMaster, um keine Rechenzeit zu verbrauchen.

Copy-on-Write Verwendet eine zusammenführbare Datenstruktur die *Copy-on-Write* Optimierung, so wird die Datenstruktur erst dann kopiert, wenn die Kopie oder das Original der Datenstruktur verändert wird. Vorher können sowohl der Parent-Task als auch der Child-Task lesend auf derselben Datenstruktur arbeiten.

CreatingEntity Das Feld *CreatingEntity* der in Kapitel 5.3.3 erweiterten Operationen für eine Zusammenführung in beliebiger Reihenfolge beschreibt durch eine Task-ID, welcher Task die Operation durchgeführt hat. Dabei werden die CreatingEntity Task-IDs der Operationen von Child-Tasks und deren Child-Tasks zusammengefasst zur Task-ID des direkten Child-Tasks. Gleiches gilt für die CreatingEntity Task-IDs der Operationen von Parent-Tasks und deren Parent-Tasks.

Currying Unter *Currying* versteht man das Integrieren von Parametern in den Funktionskörper einer neu erstellten Funktion. So lassen sich parameterlose Funktionen erstellen, deren ursprüngliche Parameter in den Funktionskörper integriert sind.

***Damenproblem* Szenario** Beim *Damenproblem* geht es um die Fragestellung, wie viele mögliche Stellungen es für n Damen auf einem $n \times n$ Schachbrett gibt, in denen sich die Damen gegenseitig nicht schlagen können.

Datenparallelität Bei einer *datenparallelen* Anwendung werden die Eingabedaten zu gleichen Teilen auf die verfügbaren Ressourcen aufgeteilt. Jeder Rechenknoten berechnet dieselbe Funktion parallel auf dem ihm zugewiesenen Segment der Daten.

Daten-Race-Condition Eine *Daten-Race-Condition* ist eine Race-Condition, die sich auf Lese- und Schreib-Operationen auf geteilten Datenstrukturen (bei gemeinsam genutztem Arbeitsspeicher) bezieht.

Datenstruktur-Hierarchie Die *Datenstruktur-Hierarchie* ist eine Verbildlichung einer Datenstruktur und aller von dieser abstammenden Datenstrukturkopien.

Datenstruktur-Historie Die *Datenstruktur-Historie* (Φ) enthält die Informationen über die Relationen zwischen Datenstrukturen und allen von ihnen abstammenden Datenstrukturkopien, sowie die Informationen zu durchgeführten Modifikationen an den Datenstrukturen.

Deadlock Ein *Deadlock* ist eine Situation, in der eine zirkuläre Wartebedingung dafür sorgt, dass die Ausführung einer Menge nebenläufiger Prozesse nicht weiter fortschreiten kann.

Determinismus Unter *Determinismus* versteht man den Umstand, dass sich das Verhalten und das Ergebnis einer Berechnung (hier auch Programmausführung) eindeutig nach einem festen Schema aus den Eingabeparametern ergeben.

Dynamisches Scheduling Bei *dynamischem Scheduling* werden Laufzeitinformationen mit in die Scheduling-Entscheidung mit einbezogen.

Event *Events* sind Ereignisse innerhalb eines Prozesses. Ereignisse beschreiben dabei im Rahmen dieser Arbeit insbesondere Synchronisationspunkte, die sich aus der Synchronisation zwischen nebenläufigen Prozessen ergeben.

Eventgraph Der *Eventgraph* ist eine visuelle Repräsentation der Events (Ereignisse) innerhalb nebenläufiger Prozesse und zwischen nebenläufigen Prozessen (im Falle der Kommunikation zweier Prozesse).

***Extreme Verteilung* Szenario** Das *Extreme Verteilung* Szenario ist eine Variante des *Normale Verteilung* Szenarios. $95,725\%$ der Tasks bekommen eine niedrige Arbeitslast, 4% eine mittlere Arbeitslast und $0,125\%$ eine hohe Arbeitslast zugewiesen.

Fahrerassistenzsystem (FAS) Ein *Fahrerassistenzsystem (FAS)* unterstützt den Fahrer eines Fahrzeuges. Dazu werden die gelesenen Daten der im Fahrzeug verbauten Sensoren ausgewertet und verarbeitet, um dem Fahrer zusätzliche Information bereitzustellen oder ihn bei der Steuerung des Fahrzeuges zu unterstützen.

finishedChildren Die Liste `finishedChildren` beinhaltet alle eigenen Child-Tasks eines Tasks, die bereits fertig ausgeführt, aber noch nicht wieder mit den eigenen Datenstrukturen zusammengeführt wurden.

First-Come-First-Serve Im Rahmen dieser Arbeit werden Tasks bei einer Zusammenführung in einer *First-Come-First-Serve*-Reihenfolge genau in der Reihenfolge zusammengeführt, in der sie auch fertiggestellt wurden.

Framework Ein *Framework* ist ein Grundgerüst, das grundlegende Funktionalitäten für die weitere Verwendung in einer Anwendung bereitstellt. Im Kontext von Spawn & Merge werden hier unter anderem die Klassen für Tasks und Task-Handles, sowie die eingeführten Synchronisationsprimitive bereitgestellt.

Frameworkebene Die *Frameworkebene* umfasst den Teil der Anwendung, der die nichtdeterministiscchen Interna der Anwendung beinhaltet (z.B. die Konfliktauflösung und das Scheduling der Tasks). Im Kontext von Spawn & Merge geschieht die Trennung zwischen Applikationsebene und Frameworkebene durch die eingeführten Synchronisationsprimitive.

Fuzzy Testing *Fuzzy Testing* beschreibt das Testen einer Anwendung durch die Eingabe zufälliger Daten. Dadurch soll die Robustheit der Anwendung für möglichst unterschiedliche Eingabedaten getestet werden.

***Game of Life* Szenario** Das *Game of Life* ist ein Beispiel für einen zellulären Automaten. Ein Gitter der Größe $n \times n$ beinhaltet in jedem Feld eine *Zelle*, die entweder lebendig oder tot ist. Über ein Regelwerk wird für jede Zelle entschieden, ob diese im nachfolgenden Simulationsschritt lebendig oder tot sein wird. Über mehrere Simulationsschritte hinweg kann somit die Entwicklung des Gitters beobachtet werden.

General Purpose Language *General Purpose Languages* ist der Überbegriff für Programmiersprachen, die für die Entwicklung beliebiger Anwendungen konzipiert wurde.

Guter Determinismus Im Kontext von Spawn & Merge bezeichnet *guter Determinismus* diejenigen nebenläufigen Ausführungen, deren deterministische Reihenfolge ein deterministisches Ergebnis ermöglichen.

Happened-Before Relation Die Happened-Before Relation (auch beschrieben als „$\rightarrow$“) trifft eine Aussage darüber, ob zwei Events a und b in einer Anwendung in einer definierten Reihenfolge geschehen (z.B. $a \rightarrow b$), oder ob die Events nebenläufig sind. Für zwei nebenläufige Events a und b gilt sowohl $a \nrightarrow b$ als auch $b \nrightarrow a$.

Konsistenz Im Gegensatz zu den identischen Ergebnissen einer deterministischen Ausführung gibt ein *konsistentes* Verhalten an, dass sich alle Prozesse auf eine gemeinsame Wahrheit einigen.

Kontrollfunktion Die *Kontrollfunktion* eines Operational Transformation Systems steuert die Zusammenführung zweier nebenläufig modifizierter Datenstrukturkopien. Dazu wird, sofern notwendig, die Transformationsfunktion des Operational Transformation Systems für Paare von Operationen aufgerufen.

Livelock Ein *Livelock* ist eine Situation, in der eine Endlosschleife dafür sorgt, dass die Ausführung eines Prozesses nicht weiter fortschreiten kann.

Liveness Die *Liveness*-Eigenschaft gibt an, ob noch einmal ein bestimmter Zustand (z.B. ein späterer erfolgreicher `Merge`-Aufruf) erreicht werden kann.

Logische `Merge`-Reihenfolge Die *logische `Merge`-Reihenfolge* beschreibt die Reihenfolge, in der die Tasks in Abhängigkeit von der verwendeten `Merge`-Primitive deterministisch zusammengeführt werden müssten, um ein deterministisches Ergebnis zu erreichen.

Main-Task Der *Main-Task* einer auf Spawn & Merge basierenden Anwendung ist der Task, der als erstes gestartet wird. Im Gegensatz zu allen anderen Tasks der Anwendung besitzt der Main-Task keinen Parent-Task.

Map Eine *Map* ist ein Schlüssel/Wert Speicher (Key/Value Store). Unter einem Schlüssel eines Typs `x` können Werte eines Typs `y` gespeichert werden.

Markierungsschnittstelle Bei der Verwendung einer *Markierungsschnittstelle* erbt eine Klasse von einem leeren Interface, wodurch bestimmte Eigenschaften der Klasse markiert werden können.

Merge Die `Merge`-Primitiven erlauben die Festlegung der Merge-Reihenfolge, in der die Ergebnisse der gestarteten Child-Tasks mit den eigenen Daten zusammengeführt werden sollen. Diese logische `Merge`-Reihenfolge hat dabei möglicherweise Einfluss auf das Ergebnis der Zusammenführung. Es gibt die folgenden `Merge`-Primitive: `MergeAll`, `MergeAllByHandle`, `MergeAny` und `MergeAnyByHandle`.

Mergeable Das `Mergeable` Interface definiert die Funktionen, die eine zusammenführbare Datenstruktur implementieren muss.

MergeAll Die `MergeAll`-Primitive löst eine deterministische Zusammenführung aller gestarteten Child-Tasks in der Spawn-Reihenfolge aus.

MergeAllByHandle Die `MergeAllByHandle`-Primitive löst eine deterministische Zusammenführung aller Child-Tasks aus, die der Primitive als Parameter übergeben wurden. Die Reihenfolge entspricht dabei der Reihenfolge der Parameter.

MergeAny Die `MergeAny`-Primitive gibt an, dass der nächste fertiggestellte Child-Task zusammengeführt werden soll. Das Ergebnis ist dabei nichtdeterministisch, da die Fertigstellungsreihenfolge der Child-Tasks nichtdeterministisch ist.

MergeAnyByHandle Die `MergeAnyByHandle`-Primitive gibt an, dass der nächste fertiggestellte Child-Task, aus der Menge der als Parameter übergebenen Child-Tasks, zusammengeführt werden soll. Das Ergebnis ist dabei nichtdeterministisch, sofern mindestens zwei Child-Tasks übergeben wurden, da die Fertigstellungsreihenfolge dieser Child-Tasks nichtdeterministisch ist.

Merge-Epoche *Merge-Epochen* unterscheiden aufeinanderfolgende Aufrufe von `Merge`-Primitiven. Sie sind notwendig zur Entscheidung ob die Mechanismen der Zusammenführung in beliebiger Reihenfolge aktiviert werden dürfen.

Merge-Modus Der *Merge-Modus* steuert das Verhalten der beiden *MergeAll*-Primitive. Dabei kann unterschieden werden, ob in diesem `Merge`-Aufruf alle Tasks einmal zusammengeführt werden sollen (`ALL_TASKS_ONCE`) oder ob der Aufruf solange wiederholt werden soll, bis alle Child-Tasks fertiggestellt wurden (`TILL_ALL_FINISHED`).

Merge-Reihenfolge Die *Merge-Reihenfolge* ist die Reihenfolge, in der die Child-Tasks eines Tasks zusammengeführt werden sollen. Sie wird dabei vom Entwickler durch die verwendete `Merge`-Primitive festgelegt.

Metaprogrammierung Unter *Metaprogrammierung* versteht man die Anreicherung des Quelltextes um Befehle, die vom Compiler zur Compile-Zeit ausgewertet werden, beispielsweise um zusätzlichen Quelltext zu generieren (z.B. bei der Verwendung von Templates).

Nachbedingung (Post-Condition) Eine *Nachbedingung (Post-Condition)* ist eine Bedingung, die nach der Modifikation von Daten überprüft wird. Wird die Nachbedingung von den modifizierten Daten nicht erfüllt, so können die Modifikationen beispielsweise verworfen werden.

Nachrichtenaustausch *Nachrichtenaustausch* ist eine Möglichkeit für die Synchronisation nebenläufiger Prozesse. Hierbei koordinieren sich die Prozesse durch das Versenden von Nachrichten untereinander.

Nebenläufig Eine *nebenläufige* Ausführung zweier Prozesse bedeutet, dass es keine festgelegte Reihenfolge für die Ausführung des Codes der beiden Prozesse gibt. Der Code kann dabei in beliebiger Reihenfolge von einem einzelnen Prozessorkern, oder aber auch parallel von zwei Prozessorkernen ausgeführt werden. Die nebenläufige Ausführung beinhaltet somit die parallele Ausführung.

Nichtdeterminismus Unter *Nichtdeterminismus* versteht man den Umstand, dass etwas sich nicht deterministisch verhält. D.h. dass das Verhalten und das Ergebnis einer Berechnung oder Programmausführung unter anderem von nicht vorhersehbaren Ereignissen oder Timings abhängen.

NodeMaster Der *NodeMaster* ist ein Thread, der auf jedem MPI-Knoten läuft und dort die Ausführung der zugewiesenen Tasks (Workitems) koordiniert. Er stellt dabei unter anderem soweit möglich sicher, dass zu jedem Zeitpunkt alle Prozessorkerne des MPI-Knotens ausgelastet sind.

***Normale Verteilung* Szenario** Das *Normale Verteilung* Szenario verteilt eine Arbeitslast auf verschiedene Child-Tasks, die Anteile unterschiedlicher Größe von der Arbeitslast zugewiesen bekommen. 65% der Tasks bekommen eine niedrige Arbeitslast, 30% eine mittlere Arbeitslast und 5% eine hohe Arbeitslast zugewiesen.

Operation Eine *Operation* (im Kontext von Operational Transformation) kapselt eine einzelne Modifikation einer Datenstruktur. Die Operation beinhaltet dabei den Zeitpunkt an dem die Operation durchgeführt wird, die Position an der die Operation durchgeführt wurde, hinzugefügte Elemente und nach Bedarf weitere Informationen.

Operational Transformation *Operational Transformation* ist ein Mechanismus zur Konfliktauflösung. Hierbei werden die Operationen, die auf zwei Kopien desselben Dokumentes angewendet wurden, so gegeneinander transformiert, dass diese anschließend den Effekt der jeweils anderen Operation beinhalten. Nach Anwendung der transformierten Operationen konvergieren die Kopien der Dokumente zu demselben Zustand.

***Operational Transformation* Szenario** Das *Operational Transformation* Szenario dient der Evaluation eines OT Systems in Hinblick auf dessen Nutzbarkeit für unterschiedliche Anwendungsarten. Dazu führt es o zufällige Operationen auf einer Liste mit n Elementen aus, während eine Arbeitslast l bewältigt wird.

Parallel Eine *parallele* Ausführung zweier Prozesse bedeutet, dass der Code der beiden Prozesse tatsächlich zur selben Zeit (beispielsweise auf zwei Prozessorkernen) ausgeführt wird.

Parent-Task Die Bezeichnung *Parent-Task* (von Task *C*) sagt aus, dass dieser Task mindestens einen anderen Task *C* gestartet hat. Dieser andere Task *C* wird auch als *Child-Task* bezeichnet.

Piggyback-Updates Unter *Piggyback-Updates* versteht man das Übertragen von Informationen (hier Aktualisierungen) als Zusatz zu anderen wichtigeren Informationen. Somit werden für die weniger wichtigen Informationen keine zusätzlichen Nachrichten versendet sondern die Informationen an wichtigere Nachrichten mit angehängt.

Quelltext-Annotationen *Quelltext-Annotationen* sind Schlüsselworte, die von Entwicklern genutzt werden können, um beispielsweise bestimmte Codebereiche als nebenläufig ausführbar zu markieren.

Race-Condition Unter einer *Race-Condition* versteht man eine Situation innerhalb einer nebenläufigen Anwendung, in der die Reihenfolge der Ausführung zweier Codebereiche nicht eindeutig geregelt ist und von den Timings der Prozessoren oder dem Scheduling des Betriebssystems abhängt.

Rechencluster Ein *Rechencluster* ist ein Zusammenschluss mehrerer Rechenknoten.

Rechenknoten Ein *Rechenknoten* beschreibt hier einen Computer (MPI-Knoten), der seine Rechenkapazitäten für die Ausführung der verteilten Anwendung zur Verfügung stellt.

Reflection *Reflection* bezeichnet die Möglichkeit zur Laufzeit Informationen (z.B. Funktionsnamen oder Funktionssignaturen) eines Objektes abzufragen.

Rekursion Unter einer *Rekursion* versteht man ein selbstreferenzielles Verhalten (z.B. eine Funktion, die sich selbst aufruft).

Reproduzierbarkeit Unter der *Reproduzierbarkeit* einer Programmausführung versteht man die Möglichkeit eine Programmausführung genau zu wiederholen.

Rollback Ein *Rollback* bezeichnet das Zurücksetzen aller Veränderungen (z.B. einer Datenstruktur) auf einen bestimmten Zustand vor der Modifikation.

Round-robin Unter *Round-robin* versteht man einen statischen Scheduling-Algorithmus, bei dem die Ressourcen in einer festen Reihenfolge nacheinander zugewiesen werden. Ist die letzte Ressource zugewiesen worden, so beginnt die Zuweisung in derselben Reihenfolge von vorne.

Scheduling Das *Scheduling* beschreibt den Mechanismus für die möglichst gleichmäßige Verteilung anfallender Arbeitslast (in Form von gestarteten Tasks) auf die verfügbaren Rechenknoten. Hier wird das Scheduling durch den SchedulingMaster vorgenommen.

SchedulingMaster Der *SchedulingMaster* ist ein Thread, der auf einem der MPI-Knoten des Systems ausgeführt wird und die Verteilung der Tasks auf MPI-Knoten koordiniert.

Schlechter Determinismus Im Kontext von Spawn & Merge bezeichnet *schlechter Determinismus* diejenigen nebenläufigen Ausführungen, die von voll-deterministischen Ansätzen in deterministischer Reihenfolge ausgeführt werden, aber das deterministische Ergebnis nicht beeinflussen. Hier kann auf die deterministische Reihenfolge verzichtet werden, um eine höhere Performance einer Anwendung zu ermöglichen.

Seiteneffektfreiheit Mit *Seiteneffektfreiheit* wird der Umstand beschrieben, dass ein Prozess keinen Effekt auf einen nebenläufig ausgeführten anderen Prozess ausübt.

Serialisierung Im Kontext der Übertragung von Datenstrukturen bezeichnet die *Serialisierung* den Prozess der Umwandlung von Datenstrukturen in eine Bytesequenz. Die *Deserialisierung* bezeichnet die Umkehrung dieses Prozesses. Im Kontext der Reihenfolge von Synchronisationsevents bezeichnet die *Serialisierung* die Ordnung von Events oder Operationen in eine konkrete Reihenfolge.

Sibling-Relation Die *Sibling-Relation* bildet die Information ab, dass ein Child-Task von einem anderen Child-Task desselben Parent-Tasks gestartet wurde.

Sibling-Task Die Bezeichnung *Sibling-Task* (von Task C) sagt aus, dass dieser Task denselben Parent-Task P hat wie der Task C. Ein Sibling-Task kann dabei entweder durch den Parent-Task P direkt, oder unter Verwendung der `SpawnSibling`-Primitive durch einen Child-Task von P gestartet worden sein.

Spawn Die `Spawn`-Primitive erlaubt das Starten einer Funktion in Form eines nebenläufig ausgeführten Child-Tasks. Die übergebenen Parameter werden dabei für den Task kopiert.

Spawn-Reihenfolge Die *Spawn-Reihenfolge* ist die Reihenfolge, in der die Child-Tasks von einem Task gestartet wurden. Sie beinhaltet dabei sowohl diejenigen Tasks, die mit der `Spawn`-Primitive gestartet wurden, als auch die Tasks, die von eigenen Child-Tasks mit der `SpawnSibling`-Primitive gestartet wurden.

SpawnSibling Die `SpawnSibling`-Primitive ermöglicht einem Child-Task des Parent-Tasks P das Starten eines neuen Tasks, der ebenfalls den Task P als Parent-Task besitzt.

Speedup-Graph (Beschleunigungsgraph) Ein Beschleunigungsgraph stellt die erreichte Beschleunigung (hier einer Anwendungsausführung) im Verhältnis zur optimal erreichbaren Beschleunigung dar. Eine optimale Beschleunigung entspricht dementsprechend der Funktion $f(x) = x$.

Sperren (Locking) Das *Sperren (Locking)* ist ein Synchronisationsmechanismus zur Realisierung des gegenseitigen Ausschlusses aus einem kritischen Codebereich. Hierbei kann immer nur genau ein nebenläufig ausgeführter Prozess auf eine geteilte Ressource zugreifen. Dazu sperrt er beispielsweise einen Mutex, der stellvertretend für den Codebereich steht. Andere Prozesse können selbst erst den Mutex sperren, wenn dieser wieder freigegeben wurde.

startingStructures Die Liste `startingStructures` beinhaltet alle Datenstrukturen, die kopiert und an eigene Child-Tasks übergeben wurden.

Statisches Scheduling Bei *statischem Scheduling* werden die Scheduling-Entscheidungen bereits bei der Programmierung der Anwendung getroffen. Laufzeitinformationen können hier nicht mit einbezogen werden.

Sync Die `Sync`-Primitive ermöglicht Child-Tasks die Übermittlung von Zwischenergebnissen an den Parent-Task. Dabei werden Modifikationen der übergebenen Datenstrukturkopien an den Parent-Task übertragen und Modifikationen des Parent-Tasks zurück an den Child-Task übergeben.

Synchronisation *Synchronisation* bezeichnet den Prozess der Koordination nebenläufiger Prozesse, z.B. im Bezug auf den Zugriff auf geteilte Ressourcen.

Synchronisationsgraph Ein *Synchronisationsgraph* stellt im Gegensatz zu einem Eventgraphen die Reihenfolge der Synchronisationspunkte im Zusammenhang mit den Prozessorkernen, auf denen die Prozesse ausgeführt wurden, dar.

Synchronisationsprimitive *Synchronisationsprimitive* sind Schlüsselworte einer Programmiersprache, die für die Steuerung der Synchronisation zwischen nebenläufigen Prozessen verwendet werden.

Synchronisationspunkt Ein *Synchronisationspunkt* ist ein Event innerhalb eines Prozesses, das sich aus der Koordination des Prozesses mit einem anderen nebenläufig ausgeführten Prozess ergibt (z.B. das Senden oder Empfangen einer Nachricht).

Sync-Modus Der *Sync-Modus* bestimmt den Zeitpunkt, an dem der Parent-Task die modifizierten Datenstrukturkopien an den Child-Task zurück übergibt. Entweder werden die Datenstrukturen direkt nach der Zusammenführung mit dem Child-Task zurückgegeben (`RETURN_DIRECTLY`) oder erst wenn die aktuelle Merge-Epoche abgeschlossen ist (`RETURN_AFTER_MERGE_CYCLE_COMPLETED`).

Task Ein *Task* kapselt alle Informationen, die eine nebenläufig ausgeführte Funktion im Rahmen des Spawn & Merge Programmiermodells benötigt. Dazu gehören unter

anderem die Informationen über den Parent-Task und die übergebenen Parameter, sowie Abhängigkeiten zu anderen Tasks (d.h. eigene Child-Tasks oder Sibling-Tasks).

Task-Handle Das *Task-Handle* kapselt alle für einen Entwickler notwendigen Informationen zu einem gestarteten Task. Diese Indirektion ist notwendig, da der tatsächliche Task auf einem anderen Rechenknoten ausgeführt werden kann und ein direkter Zugriff somit nicht unbedingt möglich ist.

Task-Hierarchie Die *Task-Hierarchie* beschreibt die Struktur einer auf Spawn & Merge basierenden Anwendung, die sich aus den Parent- und Child-Relationen zwischen den gestarteten Tasks ergibt.

Task-Informationen Die *Task-Informationen* beschreiben zum einen die Parent-Child-Relationen eines Tasks, die die baumartige Task-Hierarchie ergeben. Zum Anderen enthalten sie die Entwicklung des Tasks und seiner Child-Tasks (Task-Historie).

Taskparallelität Bei einer *taskparallelen* Anwendung können unterschiedliche Funktionen nebenläufig ausgeführt werden, die auf (möglicherweise gemeinsam genutzten) Teilen der Eingabedaten arbeiten können. Die Anzahl der Tasks, die gestartet werden können, ist dabei nicht auf die Anzahl der verfügbaren Rechenknoten beschränkt. Tasks können außerdem weitere (Sub-)Tasks starten.

Template Ein *Template* (in C++) ist eine Vorlage (z.B. für eine Funktion oder eine Klasse) mit Platzhaltern für Typen, aus der bei Verwendung im Quelltext zur Compile-Zeit vom Compiler, durch das Einsetzen entsprechender Typen (die zu der Verwendung passen) in die Platzhalter, Code generiert wird.

Thread Ein *Thread* ist Teil eines Prozesses und ermöglicht das Starten der nebenläufigen Ausführen eines Teils des Programmcodes.

Thread-Pool Ein *Thread-Pool* bezeichnet eine für eine spätere Verwendung vorbereitete Menge an Threads. Werden zu einem späteren Zeitpunkt Threads benötigt, so werden diese dem Thread-Pool entnommen.

Tiefensuche Bei einer *Tiefensuche* in einem Baum wird zuerst bis zu einem Blattknoten hinabgestiegen. Anschließend steigt der Algorithmus wieder soweit auf, bis er in einen noch nicht besuchten Teilbaum absteigen kann. Auf diese Weise wird der gesamte Baum rekursiv durchsucht.

Tombstones (Grabsteine) Gelöschte Elemente werden für die Zusammenführung in beliebiger Reihenfolge zu *Grabsteinen (Tombstones)* umgewandelt. Dies ermöglicht es dem Algorithmus gelöschte Elemente auch zu einem späteren Zeitpunkt noch in die Durchführung der Transformation mit einzubeziehen.

Transformation Properties (Transformationseigenschaften) Die Transformation Properties (Transformationseigenschaften) beschreiben die Eigenschaften eines Transformationssystems. Dabei wird zwischen Systemen des Typs TP1 und des Typs TP2 unterschieden (siehe Kapitel 5.1.3).

Transformationsfunktion Die *Transformationsfunktion* eines Operational Transformation Systems transformiert zwei nebenläufig durchgeführte Operationen gegeneinander. Sie wird dazu von der Kontrollfunktion aufgerufen.

Variadische Parameter Unter *variadischen Parametern* versteht man in einer Funktionssignatur die Angabe, dass eine beliebige Anzahl an Parametern eines bestimmten Typs erwartet wird.

Verschachtlung *Verschachtlung* bezieht sich hier auf Datenstrukturen, die ineinander enthalten sind (z.B. eine Map beinhaltet ein Array, dessen Elemente wiederum Maps sind).

Verteiltes System Unter einem *Verteilten System* versteht man mehrere miteinander verbundene Computer, die gemeinsam eine Aufgabe erfüllen (z.B. die Ausführung einer Anwendung oder die Bereitstellung eines Dienstes).

Verteilungstransparenz Unter *Verteilungstransparenz* versteht man, dass ein Nutzer eines Verteilten Systems nicht erkennen kann, dass das System auf einem Zusammenschluss mehrerer Computer ausgeführt wird.

Work-Approximation Die *Work-Approximation* stellt die Sicht des SchedulingMasters auf die aktuelle Arbeitsverteilung der gestarteten Tasks dar. Für jeden MPI-Knoten ist die Anzahl zugewiesener Tasks gespeichert. Die Work-Approximation ist dabei, abhängig von den verwendeten Mechanismen für die Aktualisierung, nicht zwingend aktuell.

Workitems Ein *Workitem* kapselt alle Informationen, die einem neu gestarteten Task mitgegeben werden. Diese Informationen umfassen, neben der Funktion und der Angabe des zugehörigen Parent-Tasks, auch die Kopien der übergebenen Datenstrukturen zum Zeitpunkt des Aufrufs der `Spawn`-Primitive.

WorkQueue Die *WorkQueue* eines NodeMasters beinhaltet alle diesem MPI-Knoten zugewiesenen, aber noch nicht gestarteten Workitems.

Workstealing Das *Workstealing* beschreibt einen Mechanismus für die Neuverteilung bereits zugewiesener Workitems. Hierbei fragt ein nicht ausgelasteter NodeMaster beim SchedulingMaster an, ob er einen anderen NodeMaster entlasten kann, indem er ihm Workitems abnimmt.

Zellulärer Automat Ein *zellulärer Automat* ist ein Modell, das ein System beschreibt, das aus miteinander benachbarten Zellen besteht. In einer Simulation hängt der kommende Zustand einer Zelle in erster Linie vom aktuellen Zustand ihrer benachbarten Zellen ab.

Zusammenführbare Datenstrukturen Eine *zusammenführbare Datenstruktur* beinhaltet Mechanismen, die es dem Spawn & Merge Framework ermöglichen eine Kopie der Datenstruktur zu erstellen und diese Kopie anschließend wieder deterministisch mit dem Original zusammenzuführen. Die dafür notwendigen Mechanismen zur Konfliktauflösung (z.B. Operational Transformation) sind dabei ein Teil der zusammenführbaren Datenstruktur, da die Mechanismen datenstrukturabhängig sind.

Anhang A

Tabellen

A.1 Task-Handle Schnittstelle

Schnittstelle	Funktion
bool isRunning()	TRUE, wenn der Task noch nicht beendet wurde
int getTaskId()	Gibt die Task-ID des abgebildeten Tasks zurück
std::vector<TaskHandle> GetSpawnedSiblings()	Gibt die gestarteten Sibling-Tasks zurück
void Abort()	Bricht die Task-Ausführung von außen ab
void setCondFct(**bool**(*fPtr)(**const std::vector<Mergeable*>** child1structures, **const std::vector<Mergeable*>** child2structures))	Weist dem Task eine Bedingungsfunktion zu, die zwei zusammenführbare Datenstrukturen als Argumente entgegennimmt
void resetCondFct()	Entfernt eine zugewiesene Bedingungsfunktion

Tabelle A.1: Task-Handle Schnittstelle.

A.2 Zusammenführbare Datenstrukturen

Die nachfolgende Tabelle listet alle Methoden der Klasse `Mergeable` auf, die bei der Implementierung einer zusammenführbaren Datenstruktur überschrieben werden müssen.

Schnittstelle	Funktion
`void ApplyTransformation( Mergeable* secondMergeable, std::map<int,int>& knownOps, SiblingInfo* knownSiblings, bool isSync, std::vector<int>* mergeOrder, int mergeEpoch)`	Transformiert die Operationen einer zweiten zusammenführbaren Datenstruktur und wendet diese auf sich selbst an
`void ExtractOperations( bool isSpawnSibling, Mergeable* fromStructure, int fromVersion)`	Extrahiert die Operationen einer Datenstruktur für `Sync` und `SpawnSibling` ab einer angegebenen Version
`void serialize( QDataStream& stream, bool isSpawnSibling)`	Serialisiert eine Datenstruktur in einen DataStream zur Übertragung
`void deserialize( QDataStream& stream)`	Deserialisiert eine Datenstruktur aus einem empfangenen Stream
`void serializeSyncResult( QDataStream& stream)`	Serialisiert Operationen der Child-Datenstruktur nach einer Transformation bei `Sync`-Aufrufen
`void deserializeSyncResult( QDataStream& stream)`	Deserialisiert ein empfangenes `Sync`-Ergebnis

Tabelle A.2: Task-Handle Schnittstelle.

A.3 Informationen in der Datenstruktur-Historie Φ

Zugehörigkeit	Gespeicherte Information
Datenstruktur-Hierarchie	Parent-Task Datenstruktur Referenz (Spawn)
	Kopien der eigenen Datenstruktur (Spawn)
	Referenz auf den Ursprung der Kopie (Sibling)
	Selbst erstellte Kopien (Sibling)
Änderungsverfolgung	Versionsnummer beim Kopieren (Spawn)
	Eigene aktuelle Versionsnummer (Spawn)
	Durchgeführte Veränderungen (Spawn)
	Versionsnummer beim Kopieren (Sibling)
	Dem Ursprung bekannte Parent-Version (Sibling)
	Zu ignorierende Veränderungen (Sibling)

Tabelle A.3: Datenstruktur-Historie Φ (vollständig).

A.4 Vollständige Liste valider `OpCodes`

OpCode	Zweck
SendWork	Starten eines Tasks
SendWorkMaster	Starten des Main-Tasks
SendData	Ergebnisübermittlung
Notify	Informationsaustausch
GetSpawnDestination	Scheduler-Anfrage
SpawnDestination	Scheduler-Antwort
GetWorkstealingDestination	Workstealing-Anfrage
WorkstealingDestination	Workstealing-Antwort
WorkFinished	Task-Fertigstellungshinweis
FetchWork	Workstealing Durchführung
Shutdown	Beenden der MPI-Knoten

Tabelle A.4: Nachrichten `OpCodes` und ihre Bedeutung.

Anhang B

Code-Beispiele und Syntax

B.1 `MergeAll`-Syntax

Listing B.1 Vollständige Definition der *MergeAll* Schnittstelle.

```
void MergeAll(mergeMode)
```

B.2 `MergeAllByHandle`-Syntax

Listing B.2 Definition der *MergeAllByHandle* Schnittstelle.

```
void MergeAllByHandle(mergeMode,[taskHandles...])
```

B.3 Anwendung der Synchronisationsprimitive

Listing B.3 Anwendung der Synchronisationsprimitive.

```
1:  // Parallel modification of list-copy within the Child-Task
2:  void ModifyList(MergeableList* mList, int i){
3:    mList->append(i);
4:  }
5:
6:  // Start of application-level deterministic execution
7:  void mainTask(){
8:    // Create new list with elements 1 and 2
9:    MergeableList* list = new MergeableList(1,2);
10:
11:   // Spawn three Child-Tasks
12:   TaskHandle task1 = Spawn(ModifyList, list, 4);
13:   TaskHandle task2 = Spawn(ModifyList, list, 5);
14:   TaskHandle task3 = Spawn(ModifyList, list, 6);
15:
16:   // Modification of the list within the Parent-Task
17:   list->append(3)
18:
19:   // Variant 1: MergeAll
20:   MergeAll(TILL_ALL_FINISHED);
21:
22:   // Variant 2: MergeAllByHandle
23:   MergeAllByHandle(TILL_ALL_FINISHED, task1, task2);
24:
25:   // Variant 3: MergeAny
26:   TaskHandle mergedTask = MergeAny();
27:
28:   // Variant 4: MergeAnyByHandle
29:   TaskHandle mergedTask = MergeAnyByHandle(task1, task2);
30:
31:   if (mergedTask == task1){
32:     print("Task1␣merged!");
33:   } else {
34:     print("Task2␣merged!");
35:   }
36: }
37:
38: void main(int argc, const char* argv[]){
39:   // Initialization of Spawn & Merge Framework
40:   InitSpawnMerge(mainTask);
41: }
```

B.4 Dining Philosophers mit Spawn & Merge

Listing B.4 Dining Philosophers mit Spawn & Merge (trivial).

```
1:  #define N 5                // Amount of philosophers
2:  #define LEFT (i+N-1)%N     // ID of left neighbor
3:  #define RIGHT (i+1)%N      // ID of right neighbor
4:
5:  int forks[N];              // Forks on the tables
6:
7:  void philosopher(int i, int fork_left, int fork_right){
8:    while (true){
9:      think();               // Think
10:     take_forks(i);         // Take copy of fork
11:     eat();                 // Eat
12:     put_forks(i);          // Put copy of fork on table
13:   }
14: }
15:
16: for (int i = 0; i < N; i++){
17:   Spawn(philosopher, i, forks[LEFT], forks[RIGHT]);
18: }
19:
20: MergeAll();
```

Anhang C

Operational Transformation Beispiele

C.1 Vollständige Transformationen für Abbildung 5.9

Initiale Operationen der Tasks:

$T_2 : \alpha_1 = insert(1, X)$ $T_3 : \beta_1 = delete(1)$ $T_4 : \gamma_1 = insert(2, Y)$

$T_5 : \gamma_1 = insert(2, Y),\ \delta_2 = insert(0, Z)$

Initiale Liste des Parent-Tasks:

$[A,\ B,\ C]_0$

Merge-Reihenfolge:

$[T_2,\ T_3,\ T_4,\ T_5]$

Merge von T_2:

Operationen des Parent-Tasks: $\emptyset$

Operationen von T_2: α_1

α_1 wird ohne Transformation vom Parent-Task übernommen

$\Rightarrow [A,\ X,\ B,\ C]_1\ \alpha_1$

Merge von T_3:

Operationen des Parent-Tasks: α_1

Operationen von T_3: β_1

$FCFS_TRANSFORM(\alpha_1,\ \beta_1) \rightarrow \beta'_1 = delete(2)$

$\Rightarrow [A,\ X,\ \cancel{B}^{\dagger},\ C]_2\ \alpha_1\ \beta'_2$

Merge von T_4:

Operationen des Parent-Tasks: $\alpha_1,\ \beta'_2$

Operationen von T_4: γ_1

$FCFS_TRANSFORM(\alpha_1,\ \gamma_1) \rightarrow \gamma'_1 = insert(3, Y)$

$FCFS_TRANSFORM(\beta'_2,\ \gamma'_1) \rightarrow \gamma''_1 = insert(3, Y)$

$\Rightarrow [A,\ X,\ \cancel{B}^{\dagger},\ Y,\ C]_3\ \alpha_1\ \beta'_2\ \gamma''_3$

Merge von T_5:

Operationen des Parent-Tasks: $\alpha_1,\ \beta_2',\ \gamma_3''$

Operationen von T_5: δ_2

$FCFS_TRANSFORM(\alpha_1,\ \delta_1) \rightarrow \delta_1' = insert(0, Z)$

$FCFS_TRANSFORM(\beta_2',\ \delta_1') \rightarrow \delta_1'' = insert(0, Z)$

$FCFS_TRANSFORM(\gamma_3'',\ \delta_1'') \rightarrow \delta_1''' = insert(0, Z)$

$\Rightarrow [Z,\ A,\ X,\ \cancel{B}^{\dagger},\ Y,\ C]_4\ \alpha_1\ \beta_2'\ \gamma_3''\ \delta_4'''$

Finale Liste des Parent-Tasks:

$[Z,\ A,\ X,\ \cancel{B}^{\dagger},\ Y,\ C]_4\ \alpha_1\ \beta_2'\ \gamma_3''\ \delta_4'''$

C.2 Vollständige Transformationen für Abbildung 5.9 (FCFS)

Initiale Operationen der Tasks:

$T_2 : \alpha_1 = insert(1, X)$ $\quad T_3 : \beta_1 = delete(1)$ $\quad T_4 : \gamma_1 = insert(2, Y)$

$T_5 : \gamma_1 = insert(2, Y),\ \delta_2 = insert(0, Z)$

Initiale Liste des Parent-Tasks:

$[A,\ B,\ C]_0$

Merge-Reihenfolge:

$[T_3,\ T_5,\ T_2,\ T_4]$

Merge von T_3:

Operationen des Parent-Tasks: $\emptyset$

Operationen von T_3: β_1

β_1 wird ohne Transformation vom Parent-Task übernommen.

$\Rightarrow [A,\ \cancel{B}^{\dagger},\ C]_1\ \beta_1$

Merge von T_5:

Operationen des Parent-Tasks: β_1

Operationen von T_5: $\gamma_1,\ \delta_2$

$FCFS_TRANSFORM(\beta_1,\ \gamma_1) \rightarrow \gamma_1' = insert(2, Y)$

$FCFS_TRANSFORM(\beta_1,\ \delta_2) \rightarrow \delta_2' = insert(0, Z)$

$\Rightarrow [Z,\ A,\ \cancel{B}^{\dagger},\ Y,\ C]_3\ \beta_1\ \gamma_2'\ \delta_3'$

Merge von T_2:

Operationen des Parent-Tasks: $\beta_1,\ \gamma_2',\ \delta_3'$

Operationen von T_2: α_1

Umgekehrte Parameterreihenfolge ist rot markiert.

$FCFS_TRANSFORM(\alpha_1,\ \beta_1) \rightarrow \alpha_1' = insert(1, X)$

$FCFS_TRANSFORM(\alpha_1',\ \gamma_2') \rightarrow \alpha_1'' = insert(1, X)$

$FCFS_TRANSFORM(\alpha_1'',\ \delta_3') \rightarrow \alpha_1''' = insert(2, X)$

$\Rightarrow [Z,\ A,\ X,\ \cancel{B}^{\dagger},\ Y,\ C]_4\ \beta_1\ \gamma_2'\ \delta_3'\ \alpha_4'''$

Merge von T_4:

Operationen des Parent-Tasks: β_1, γ'_2, δ'_3, α'''_4

Operationen von T_4: γ_1

γ_1 *wurde bereits mit Task* T_5 *vom Parent-Task übernommen.*

$\Rightarrow [Z, A, X, \cancel{B}^{\dagger}, Y, C]_4\ \beta_1\ \gamma'_2\ \delta'_3\ \alpha'''_4$

Finale Liste des Parent-Tasks:

$[Z, A, X, \cancel{B}^{\dagger}, Y, C]_4\ \beta_1\ \gamma'_2\ \delta'_3\ \alpha'''_4$

Literaturverzeichnis

[1] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *Computer*, 19(8):26–34, August 1986.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeño virtual machine. *IBM Syst. J.*, 39(1):211–238, January 2000.

[3] Bowen Alpern, Jong-Deok Choi, Ton Ngo, Manu Sridharan, and John M. Vlissides. A perturbation-free replay platform for cross-optimized multithreaded applications. In *Proceedings of the 15th International Parallel &Amp; Distributed Processing Symposium*, IPDPS '01, pages 23–, Washington, DC, USA, 2001. IEEE Computer Society.

[4] Amazon Web Services. New Compute-Optimized EC2 Instances. `https://aws.amazon.com/cn/blogs/aws/new-c4-instances/`, 2017. Zugriff: 02.06.2017.

[5] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[6] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[7] H. E. Bal and M. Haines. Approaches for integrating task and data parallelism. *IEEE Concurrency*, 6(3):74–84, Jul 1998.

[8] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Comput. Surv.*, 21(3):261–322, September 1989.

[9] Assia Belbachir, Jean-Christophe Smal, Jean-Marc Blosseville, and Dominique Gruyer. Simulation-driven validation of advanced driving-assistance systems. *Procedia - Social and Behavioral Sciences*, 48:1205 – 1214, 2012. Transport Research Arena 2012.

[10] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: A compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 53–64, New York, NY, USA, 2010. ACM.

[11] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in dos. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[12] Andrew P. Black, Kim B. Bruce, Michael Homer, and James Noble. Grace: The absence of (inessential) difficulty. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2012, pages 85–98, New York, NY, USA, 2012. ACM.

[13] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[14] Robert L. Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 4–4, Berkeley, CA, USA, 2009. USENIX Association.

[15] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. *SIGPLAN Not.*, 44(10):97–116, October 2009.

[16] M. Bozikovic, M. Golub, and L. Budin. Solving n-queen problem using global parallel genetic algorithm. In *The IEEE Region 8 EUROCON 2003. Computer as a Tool.*, volume 2, pages 104–107 vol.2, Sept 2003.

[17] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159, March 2014.

[18] Erik Buchmann, Sven Apel, and Gunter Saake. Piggyback meta-data propagation in distributed hash tables. In *Proceedings of the 1st International Conference on Web*

Information Systems and Technologies (WEBIST'05), 2005, pages 72–81, Portugal, 2005. INSTICC Press.

[19] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşırlar. Concurrent collections. *Sci. Program.*, 18(3-4):203–217, August 2010.

[20] Roy H. Campbell and A. Nico Habermann. The specification of process synchronization by path expressions. In *Operating Systems, Proceedings of an International Symposium*, pages 89–102, London, UK, UK, 1974. Springer-Verlag.

[21] Franck Cappello, Amina Guermouche, and Marc Snir. On communication determinism in parallel hpc applications. In *ICCCN*, pages 1–8. IEEE, 2010.

[22] Richard H. Carver and Kuo-Chung Tai. Replay and testing for concurrent programs. *IEEE Softw.*, 8(2):66–74, March 1991.

[23] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, SPDT '98, pages 48–59, New York, NY, USA, 1998. ACM.

[24] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, June 1971.

[25] Duncan Coutts and Andres Löh. Deterministic parallel programming with haskell. *Computing in Science & Engineering*, 14(6):36–43, 2012.

[26] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 388–405, New York, NY, USA, 2013. ACM.

[27] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 337–351, New York, NY, USA, 2011. ACM.

[28] Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. Stable deterministic multithreading through schedule memoization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 207–221, Berkeley, CA, USA, 2010. USENIX Association.

[29] H.B. Curry, R. Feys, and W. Craig. *Combinatory Logic*. Number Bd. 1 in Studies in logic and the foundations of mathematics. North-Holland, 1968.

[30] Aguido Horatio Davis, Chengzheng Sun, and Junwei Lu. Generalizing operational transformation to the standard general markup language. In *Proceedings of the 2002 ACM conference on Computer supported cooperative work*, CSCW '02, pages 58–67, New York, NY, USA, 2002. ACM.

[31] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[32] Joseph Devietti, Luis Ceze, and Dan Grossman. The Case For Merging Execution- and Language-level Determinism with MELD. In *Workshop on Determinism and Correctness in Parallel Programming w/ International Conference on Architectural Support for Programming Languages and Operating Systems (WoDet w/ ASPLOS)*, 3 2012.

[33] E. W. Dijkstra. Cooperating sequential processes, technical report ewd-123. Technical report, Eindhoven University of Technology, 1965.

[34] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1(2):115–138, June 1971.

[35] Romain Dolbeau, Stéphane Bihan, François Bodin, and Caps Entreprise. Hmpp: A hybrid multi-core parallel programming environment, 2007.

[36] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, SIGMOD '89, pages 399–407, New York, NY, USA, 1989. ACM.

[37] Ira R. Forman and Nate Forman. *Java Reflection in Action (In Action Series)*. Manning Publications Co., Greenwich, CT, USA, 2004.

[38] Neil Fraser. Differential synchronization. In *Proceedings of the 9th ACM Symposium on Document Engineering*, DocEng '09, pages 13–20, New York, NY, USA, 2009. ACM.

[39] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 79–90, New York, NY, USA, 2009. ACM.

[40] M. Gardner. The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223:120–123, October 1970.

[41] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, February 1992.

[42] Google. The Go Programming Language Specification [May 31, 2016]. Technical report, Google, 2016.

[43] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[44] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Comput.*, 22(6):789–828, September 1996.

[45] Dhiraj Gupta and V.K. Gupta. Approaches for Deadlock Detection and Deadlock Prevention for Distributed systems. *Research Journal of Recent Sciences*, 1, 2012.

[46] Per Brinch Hansen. *Joyce—A Programming Language for Distributed Systems*, pages 464–492. Springer New York, New York, NY, 2002.

[47] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[48] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[49] Mark Hill. Racey: A Stress Test for Deterministic Execution. `http://pages.cs.wisc.edu/~markhill/racey.html`, 2009. Zugriff: 31.01.2017.

[50] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, December 1986.

[51] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[52] Nicholas Hunt, Tom Bergan, Luis Ceze, and Steven D. Gribble. Ddos: Taming nondeterminism in distributed systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 499–508, New York, NY, USA, 2013. ACM.

[53] ISO/IEC. Pascal [ISO 7185:1990]. Technical report, International Organization for Standardization (ISO), 1990.

[54] ISO/IEC. Standard for Programming Language C++ [Working Draft, N3337]. Technical report, International Organization for Standardization (ISO), 2012.

[55] ISO/IEC. Lambda functions on CppReference.com. `http://de.cppreference.com/w/cpp/language/lambda`, 2017. Zugriff: 31.07.2017.

[56] ISO/IEC. std::map on CppReference.com. `http://en.cppreference.com/w/cpp/container/map`, 2017. Zugriff: 02.02.2017.

[57] Simon Peyton Jones. Beautiful concurrency. In Andy Oram and Greg Wilson, editors, *Beautiful Code*. O'Reilly, 2007.

[58] Tim Jungnickel and Tobias Herb. Tp1-valid transformation functions for operations on ordered n-ary trees. *CoRR*, abs/1512.05949, 2015.

[59] Khronos Group. OpenCL 2.2 (Provisional) API Specification [March 11, 2016]. Technical report, Khronos Group, 2016.

[60] Ajay Khunteta and Santosh Kumawat. Article: A survey on operational transformation algorithms: Challenges, issues and achievements. *International Journal of Computer Applications*, 3(12):30–38, July 2010. Published By Foundation of Computer Science.

[61] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[62] Leslie Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Trans. Program. Lang. Syst.*, 1(1):84–97, January 1979.

[63] Claudia lavinia Ignat and Moira C. Norrie. Customizable collaborative editor relying on treeopt algorithm. In *In Proc. of the European Conf. of Computer-supported Cooperative Work*, pages 315–334. Kluwer Academic Publishers, 2003.

[64] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 327–336, New York, NY, USA, 2011. ACM.

[65] Li Lu and Michael L. Scott. Toward a formal semantic framework for deterministic parallel programming. In *Proceedings of the 25th International Conference on Distributed Computing*, DISC'11, pages 460–474, Berlin, Heidelberg, 2011. Springer-Verlag.

[66] Bertrand Meyer. *Object-oriented Software Construction (2Nd Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[67] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[68] Jayadev Misra and William R. Cook. Computation orchestration. *Software & Systems Modeling*, 6(1):83–110, 2007.

[69] MPI Forum. Document for a standard message-passing interface. Technical report, MPI Forum, Knoxville, TN, USA, 1993.

[70] Kai Nagel and Marcus Rickert. Parallel implementation of the transims micro-simulation. *Parallel Computing*, 27:200–1, 2001.

[71] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, UIST '95, pages 111–120, New York, NY, USA, 1995. ACM.

[72] NumFocus. Documentation for Programming Language Julia. `http://docs.julialang.org/en/stable/`, 2016. Zugriff: 31.01.2017.

[73] NVIDIA. CUDA Runtime API (v8.0). Technical report, NVIDIA, 2016.

[74] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 97–108, New York, NY, USA, 2009. ACM.

[75] OpenMP Architecture Review Board. OpenMP Application Program Interface. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, 2014.

[76] Oracle. Java Remote Method Invocation - Distributed Computing for Java. `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html`. Zugriff: 20.07.2017.

[77] G. Oster, P. Molli, P. Urso, and A. Imine. Tombstone transformation functions for ensuring consistency in collaborative editing systems. In *Collaborative Computing: Networking, Applications and Worksharing, 2006. CollaborateCom 2006. International Conference on*, pages 1–10, Nov 2006.

[78] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, July 1982.

[79] P.J. Pahl and R. Damrath. *Mathematische Grundlagen Der Ingenieurinformatik*. Springer, 2000.

[80] Benjamin C. Pierce. Foundational calculi for programming languages. In *in the CRC Handbook of Computer Science and Engineering. Available electronically*, 1995.

[81] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 12–25, New York, NY, USA, 2011. ACM.

[82] Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, pages 395–403, Washington, DC, USA, 2009. IEEE Computer Society.

[83] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[84] Shiru Ren, Le Tan, Chunqi Li, Zhen Xiao, and Weijia Song. Samsara: Efficient deterministic replay in multiprocessor environments with hardware virtualization extensions. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 551–564, Denver, CO, June 2016. USENIX Association.

[85] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work*, CSCW '96, pages 288–297, New York, NY, USA, 1996. ACM.

[86] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *SIGGRAPH Comput. Graph.*, 21(4):25–34, August 1987.

[87] Davide Sangiorgi and David Walker. *PI-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001.

[88] Cedomir Segulja and Tarek S. Abdelrahman. What is the cost of weak determinism? In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 99–112, New York, NY, USA, 2014. ACM.

[89] Rok Sosic. A parallel search algorithm for the n-queens problem. *IEEE Transactions on Systems, Man, and Cybernetics*, 21:Nov/Dec, 1991.

[90] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 59–68. ACM, 1998.

[91] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact.*, 5(1):63–108, March 1998.

[92] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 4th edition, 2014.

[93] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[94] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. Prentice Hall Press, Upper Saddle River, NJ, USA, 5th edition, 2010.

[95] Terracotta Inc. *The Definitive Guide to Terracotta: Cluster the JVM for Spring, Hibernate and POJO Scalability*. Apress, Berkely, CA, USA, 2008.

[96] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

[97] University of Illinois at Urbana-Champaign. Deterministic Parallel Java (DPJ) project. http://dpj.cs.uiuc.edu/DPJ/Home.html, 2014.

[98] Zhaocai Wang, Dongmei Huang, Jian Tan, Taigang Liu, Kai Zhao, and Lei Li. A parallel algorithm for solving the n-queens problem based on inspired computational model. *Biosystems*, 131:22 – 29, 2015.

[99] Torben Weis and Arno Wacker. Federating websites with the google wave protocol. *IEEE Internet Computing*, 15(3):51–58, 2011.

[100] Tian Xiao, Jiaxing Zhang, Hucheng Zhou, Zhenyu Guo, Sean McDirmid, Wei Lin, Wenguang Chen, and Lidong Zhou. Nondeterminism in mapreduce considered harmful? an empirical study on non-commutative aggregators in mapreduce programs. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 44–53, New York, NY, USA, 2014. ACM.

[101] Junfeng Yang, Heming Cui, and Jingyue Wu. Determinism is overrated: What really makes multithreaded programs hard to get right and what can be done about it? In *Proceedings of the Fifth USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, USA, 2013. USENIX Association.

[102] Junfeng Yang, Heming Cui, Jingyue Wu, Yang Tang, and Gang Hu. Making parallel programs reliable with stable multithreading. *Commun. ACM*, 57(3):58–69, March 2014.

[103] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

Liste der Publikationen

Christopher Boelmann hat im Jahr 2011 das Angewandte Informatik Studium an der Universität Duisburg-Essen mit dem Abschluss Diplom-Informatiker abgeschlossen. Seine Diplomarbeit behandelte thematisch die Konzeption und Entwicklung von Entwicklungswerkzeugen für Autoren für Videospiele. Nach seinem Studium hat er sich 2011 dem Verteilte Systeme Lehrstuhl an der Universität Duisburg-Essen angeschlossen. Christopher hat unter anderem Forschungsergebnisse auf den Gebieten Verteilte Systeme, Peer-to-Peer Systeme und Privatsphäre und Sicherheit im Internet veröffentlicht.

Begutachtete Publikationen

1. Matthäus Wander, Christopher Boelmann, and Torben Weis. Domain Name System without Root Servers. In *Proceedings of the 12th International Conference on Risks and Security of Internet and Systems*, CRiSIS '17, 2017. Springer.

2. Christopher Boelmann, Lorenz Schwittmann, Marian Waltereit, Matthäus Wander, and Torben Weis. Application-level Determinism in Distributed Systems. In *Proceedings of the 2016 International Conference on Parallel and Distributed Systems*, ICPADS '16, Washington, DC, USA, 2016. IEEE Computer Society.

3. Lorenz Schwittmann, Christopher Boelmann, Viktor Matkovic, Matthäus Wander, and Torben Weis. Identifying TV Channels & On-Demand Videos using Ambient Light Sensors. *Pervasive and Mobile Computing*, 2016.

4. Torben Weis and Christopher Boelmann. Automatismen zur Strukturbildung und Selbst-Organisation in verteilten Systemen. In *Logiken strukturbildender Prozesse: Automatismen*, editors: Norbert Otto Eke, Lioba Foit, Timo Kaerlein, and Jörn Künsemöller, pages 145–160. Wilhelm Fink, Paderborn, Germany, Oct 2014.

5. Matthäus Wander, Lorenz Schwittmann, Christopher Boelmann, and Torben Weis. GPU-Based NSEC3 Hash Breaking. In *Network Computing and Applications (NCA), 2014 IEEE 13th International Symposium on*, pages 137–144, Aug 2014.

6. Christopher Boelmann, Lorenz Schwittmann, and Torben Weis. Deterministic Synchronization of Multi-threaded Programs with Operational Transformation. In *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, IPDPSW '14, pages 381–390, Washington, DC, USA, 2014. IEEE Computer Society.

7. Matthäus Wander, Christopher Boelmann, Lorenz Schwittmann, and Torben Weis. Measurement of Globally Visible DNS Injection. *Access, IEEE*, 2:526–536, 2014.

8. Lorenz Schwittmann, Matthaus Wander, Christopher Boelmann, and Torben Weis. Privacy Preservation in Decentralized Online Social Networks. *Internet Computing, IEEE*, 18(2):16–23, Mar 2014.

9. Christopher Boelmann and Torben Weis. Development of Efficient Role-Based Sensor Network Applications with Excel Spreadsheets. In *Proceedings of the 2013 International Conference on Parallel and Distributed Systems*, ICPADS '13, pages 365–371, Washington, DC, USA, 2013. IEEE Computer Society.

10. Lorenz Schwittmann, Christopher Boelmann, Matthäus Wander, and Torben Weis. SoNet – Privacy and Replication in Federated Online Social Networks. In *Distributed Computing Systems Workshops (ICDCSW), 2013 IEEE 33rd International Conference on*, pages 51–57, July 2013.

11. Christopher Boelmann, Torben Weis, Michael Engel, and Arno Wacker. Self-Stabilizing Micro Controller for Large-Scale Sensor Networks in Spite of Program Counter Corruptions Due to Soft Errors. In *Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems*, ICPADS '12, pages 506–513, Washington, DC, USA, 2012. IEEE Computer Society.