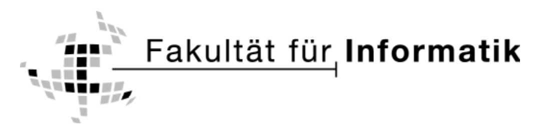


Automatische Parallelisierung mittels einer Beschreibungssprache für parameterisierbare Muster

Studienarbeit
von

Sebastian Stehle

Verantwortlicher Betreuer: Prof. Dr. Walter F. Tichy
Betreuender Mitarbeiter: Dipl.-Inform. Korbinian Molitorisz

Bearbeitungszeit: 01. Mai 2012 – 31.Juli 2012

Karlsruher Institut für Technologie · Institut für Programmstrukturen und Datenorganisation ·
Geb. 50.34 · Am Fasanengarten 5 · 76131 Karlsruhe

Ehrenwörtliche Erklärung

Ich erkläre hiermit, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Karlsruhe, den 31. Juli 2012

Sebastian Stehle

Inhaltsverzeichnis

1 Einleitung 7

2 Grundlagen 9

2.1 Entwurfsmuster für parallele Programmierung ... 9
2.1.1 Parallele Schleifen ... 9
2.1.2 Parallele Reduktion ... 9
2.1.3 Fork-Join ... 10
2.1.4 Pipeline (Fließband) ... 10
2.2 Architekturbeschreibungssprachen ... 11
2.3 Grundlagen für die Implementierung ... 11
2.3.1 .NET Framework und C# ... 11
2.3.2 Aufgabenbasierte parallele Programmierung ... 12

3 Verwandte Arbeiten 13

3.1 Beschreibungssprachen für parallele Programme ... 13
3.2 TADL ... 13
3.3 XJava ... 14

4 Eigener Ansatz 16

4.1 Parallelisierungsprozess ... 16
4.2 Ziele und Anforderungen ... 17
4.3 Zusammenfassung ... 19

***5* Technischer Entwurf 20**

5.1 *Tunable Architecture Description Language* (TADL) ... 20
5.1.1 Atomare Ausführungseinheiten (AE) ... 20
5.1.2 Operatoren ... 20
5.1.3 Sequenzieller Operator ... 25
5.2 Format zur Beschreibung von Tuning Parametern ... 26
5.3 Werkzeugunterstütze Quelltexttransformation ... 27
5.3.1 Entwurf des Rahmenwerks TPB (Tunable Parallel Blocks) ... 27
5.3.2 Automatische Quelltexttransformation ... 33
5.3.3 Laufzeitverhalten ... 34
5.4 Zusammenfassung ... 34

6 Implementierung 35

6.1 Implementierung von TPB ... 35
6.1.1 Nachrichten ... 35
6.1.2 Block Klassen ... 36
6.2 Quelltexttransformation ... 39
6.2.1 Analyse ... 39
6.2.2 Transformation ... 40

7 Evaluierung **42**

7.1 Beispiele 42

7.1.1 Desktopsuche 42
7.1.2 Bilder Konvertierung 43
7.1.3 Berechnen der konvexen Hülle 44

7.2 Fazit 45

8 Zusammenfassung und Ausblick **46**

Anhänge **47**

1 EINLEITUNG

Mit der steigenden Zahl an CPU-Kernen in heutigen Rechnern sowie zunehmend sogar in mobilen Endgeräten, erhöhen sich auch die Anforderungen an den Entwickler, effizienten und schnellen Quelltext zu schreiben. Die typischen Probleme bei der parallelen Programmierung, wie Datenwettläufe und Verklemmungen, und die schwierige Fehlerbehandlung erschweren die Softwareentwicklung zusätzlich. In einer Umfrage, die im Auftrag der Firma Intel durchgeführt wurde, wurden im Oktober und November 2011 275 Softwareentwickler über ihre Erfahrung mit paralleler Programmierung befragt [UWT11]. Davon antworteten mehr als 75 %, dass die parallele Programmierung einen wichtigen bis sehr wichtigen Stellenwert in ihrer Arbeit einnimmt, und mehr als die Hälfte verbringen mehr als sechs Stunden pro Monat mit der Performanzoptimierung ihrer Anwendungen. Allerdings bezeichnen sich nur ein Drittel der Entwickler als Experten oder fortgeschrittenen Entwickler auf diesem Gebiet. Obwohl die Werkzeuge immer besser werden, kann der Bedarf an Softwareentwickler mit umfassenden Kenntnissen in paralleler Programmierung also nicht gedeckt werden. Dazu kommt außerdem, dass noch viele Anwendungen im Einsatz sind, die für dedizierte Anwendungsfälle entwickelt wurden, aber nicht weiter betreut werden. Es wird zunehmend notwendig, diese an die neuen Rechner mit mehreren Kernen anzupassen, was oftmals mit hohen Kosten oder sogar einer kompletten Neuentwicklung verbunden ist.

Es ist von Interesse, die Parallelisierung von Anwendungen zu automatisieren. Die Parallelisierung kann auf Hardwareebene und Softwareebene durchgeführt werden. Moderne Prozessoren führen die Befehle in einer Pipeline aus und können daher mehrere Instruktionen gleichzeitig verarbeiten. Im Bereich der Softwareentwicklung ist unter anderem die automatische Parallelisierung von Schleifen im Fokus der Forschung: Dabei wird untersucht, wie Abhängigkeiten zwischen den einzelnen Iterationen so aufgelöst werden können, dass die Iterationen effizient nebenläufig ausgeführt werden können. Die besten Ergebnisse können bei der Parallelisierung erzielt werden, wenn die gesamte Anwendung parallelisiert werden kann. So werden bei komplexen Spielen in einer Schleife mehrmals pro Sekunde Benutzereingaben verarbeitet, Spielobjekte bewegt, physikalische Berechnungen durchgeführt und Zeichenoperationen ausgeführt. Da die einzelnen Schritte jeweils von ihrem Vorgänger abhängig sind, können diese nicht nebenläufig ausgeführt werden. Es ist aber möglich, im aktuellen Schleifendurchlauf schon die physikalischen Berechnungen für die nächste Iteration durchzuführen. Damit kann auf einem System mit mehreren Recheneinheiten eine sehr gute Beschleunigung erzielt werden, ohne dass einzelne Quelltextabschnitte noch weiter durch Parallelisierung optimiert werden müssen. Je nach Anwendung bieten sich also verschiedene Strategien zur Parallelisierung an.

Die Aufgaben bei der automatischen Parallelisierung sind die gleichen wie auch bei der manuellen Parallelisierung: Zuerst müssen Stellen im Programm identifiziert werden, die sich zur Parallelisierung eignen könnten. Anschließend muss entschieden werden, wie parallelisiert werden soll und der Quelltext muss entsprechend umgebaut werden. Zuletzt muss aus dem transformierten Quelltext ein lauffähiges Programm erstellt werden. Bei der Ermittlung der zu parallelisierenden Stellen und dem Erstellen der Parallelisierungsstrategie besteht noch Forschungsbedarf. Es wäre aber hilfreich, wenn ein Werkzeug zur Verfügung stehen würde, mit dem Quelltext auf Basis von beliebig komplexen Kompositionen von vorher definierten Entwurfsmustern transformiert werden kann. In zukünftigen Arbeiten könnte dann auf dieses Werkzeug zurückgegriffen werden, um die entwickelten Analyseverfahren zur automatischen Parallelisierung zu testen. Ein solches Werkzeug könnte auch dazu verwendet werden, manuell Entwurfsmuster für den eigenen Quelltext anzugeben. Dies erfordert aber ein sehr gutes Verständnis der Entwurfsmuster und der parallelen Programmierung, da das Werkzeug darauf angewiesen ist, dass der Parallelisierungsvorschlag eine gültige Lösung ist, bei der die Korrektheit gewährleistet werden kann.

Die Entwicklung eines solchen Werkzeugs ist Teil dieser Arbeit. Dazu wird eine Beschreibungssprache entworfen, mit der parallele Architekturen beschrieben werden können. Diese

Beschreibungssprache baut auf Entwurfsmustern auf. Dadurch, dass Entwurfsmuster Lösungen für Probleme beschreiben, die in realen Softwareprojekten häufig auftauchen, haben diese eine große praktische Relevanz. Das Werkzeug liest den sequenziellen Quelltext und die Architekturbeschreibung ein und führt eine Quelltexttransformation durch. Dabei wird ein Framework injiziert, das über Parameter von außen vor Programmausführung konfiguriert werden kann. Diese Parameter beeinflussen die Laufzeit des Programms, zum Beispiel indem der Parallelitätsgrad einer Schleife spezifiziert wird. Welche Parameter zur Verfügung stehen hängt von der Architekturbeschreibung ab. Eine Liste mit diesen Parametern wird von dem entwickelten Werkzeug zur Verfügung gestellt und kann dann verwendet werden, um das Programm an die Zielplattform anzupassen. Bei diesem Prozess, der Tuning genannt wird, wird die Parameterkonfiguration gesucht, bei der das Programm auf der Zielplattform die beste Laufzeit aufweist.

Der Schwerpunkt dieser Arbeit liegt darin, ein Werkzeug zu entwerfen, das für möglichst viele Anwendungsfälle verwendet werden kann. Außerdem müssen die folgenden Fragen beantwortet werden, um herauszufinden, in welchem Maße die automatische Quelltexttransformation durchgeführt werden kann:

- **Korrektheit:** Welche Maßnahmen müssen durchgeführt werden, damit das transformierte Programm korrekt arbeitet? Dabei soll davon ausgegangen, dass der Parallelisierungsvorschlag eine gültige Lösung ist.
- **Beschleunigung:** Welche Beschleunigung kann dabei erzielt werden und wie groß ist der Verwaltungsaufwand durch das zusätzlich eingeführte Framework?
- **Grad der Automatisierung:** Inwieweit kann der Quelltext automatisch transformiert werden und welche manuelle Anpassungen sind zuvor notwendig?

Der Rest der Arbeit gliedert sich wie folgt:

- In Kapitel 2 werden Grundlagen erläutert, die zum Verständnis der Arbeit notwendig sind. Die Entwurfsmuster, die von dem implementierten Werkzeug verarbeitet werden können, werden vorgestellt. Außerdem werden die bei der Implementierung verwendete Programmiersprache und Laufzeitumgebung und das Prinzip der aufgabenorientierten Programmierung eingeführt.
- In Kapitel 3 werden verwandte Arbeiten untersucht: Mit TADL und XJava werden zwei Architekturbeschreibungssprachen beschrieben, auf welche die hier entwickelte Sprache aufbaut.
- In Kapitel 4 wird der eigene Ansatz vorgestellt. Dazu wird zuerst der Prozess der automatischen Parallelisierung definiert und es wird erläutert, wie sich das Konzept der Quelltexttransformation in diesen Prozess eingliedert. Auf Basis dieses Prozesses werden Ziele und Anforderungen abgeleitet.
- Kapitel 5 beschreibt den technischen Entwurf der in Kapitel 4 beschriebenen Konzepte. Die Architekturbeschreibungssprache wird im Detail erläutert. Außerdem wird beschrieben, wie Informationen über Tuningparameter zur Verfügung gestellt werden. Im letzten Abschnitt werden das Konzept und die Architektur des implementierten Werkzeugs zur Quelltexttransformation vorgestellt.
- Kapitel 6 befasst sich mit der Implementierung des Werkzeugs zur Quelltexttransformation und dem Framework, mit dem die unterstützten Muster umgesetzt werden.
- In Kapitel 7 wird das entwickelte Werkzeug mit Hilfe von drei Beispielen evaluiert: Die Beschleunigung durch die Quelltexttransformation wird gemessen. Außerdem wird untersucht, ob das transformierte Programm weiterhin korrekt arbeitet.
- Kapitel 8 bildet mit der Zusammenfassung und einem Ausblick auf zukünftige Arbeiten den Schluss dieser Arbeit.

2 GRUNDLAGEN

In diesem Kapitel werden einige Grundlagen erläutert, die für das Verständnis der Arbeit wichtig sind. Wie bereits erläutert, werden Entwurfsmuster als Basis der Architekturbeschreibungssprache verwendet. Deshalb werden in diesem Kapitel die Muster beschrieben, die von dem implementierten Werkzeug verarbeitet werden können und zum besseren Verständnis die grundlegenden Arten der Parallelisierung eingeführt. Da sich ein großer Teil der Arbeit dem Entwurf einer Architekturbeschreibungssprache widmet, folgt im Anschluss die Definition einer solchen Sprache und es wird erläutert, aus welchen Bauteilen sich eine Architekturbeschreibungssprache im Allgemeinen zusammensetzt. Zuletzt werden die bei der Implementierung verwendete Programmiersprache und Laufzeitumgebung vorgestellt und das Prinzip der aufgabenorientierten parallelen Programmierung erläutert.

2.1 Entwurfsmuster für parallele Programmierung

Entwurfsmuster dienen als Basis der in dieser Arbeit entwickelten Architekturbeschreibungssprache. Wie bei der objektorientierten Programmierung haben sich Entwurfsmuster für die parallele Programmierung etabliert, um jeweils eine Lösung für eine Klasse von Problemen zu bieten. In [MS+04] stellen die Autoren einen umfangreichen Katalog von parallelen Entwurfsmustern vor und beschreiben drei Arten von Parallelität:

- **Datenparallelität:** Bei der Datenparallelität liegt der Fokus auf den zu verarbeitende Daten. Es werden Partitionen von Daten gebildet, die unabhängig voneinander verarbeitet werden können.
- **Aufgabenparallelität:** Hierbei findet eine funktionale Dekomposition statt, indem Aufgaben identifiziert werden, die parallel ausgeführt werden können. Die beste Effizienz wird dabei erreicht, wenn die Aufgaben keine Abhängigkeiten zueinander aufweisen und keinen geteilten Zustand haben.
- **Flussbasierte Parallelität**: Bei der flussbasierten Parallelität liegt der Fokus auf der Modellierung eines Datenflusses. Die Daten werden nacheinander oder nebenläufig von mehreren Aufgaben verarbeitet, wobei unterschiedliche Datenelemente von unterschiedlichen Aufgaben parallel verarbeitet werden können.

Im Folgenden werden die vier Entwurfsmuster vorgestellt, die von dem implementierten Werkzeug verarbeitet werden können und jeweils einem der drei Parallelitätsklassen zugeordnet. Bei der Auswahl der Muster wurde Wert darauf gelegt, alle Arten der Parallelität zu unterstützen.

2.1.1 Parallele Schleifen

Das parallele Ausführen von Schleifen ist das bekannteste Muster für **Datenparallelität**. Sind die einzelnen Iterationen unabhängig voneinander, können diese parallel ausgeführt werden. Um die Anzahl an Fäden zu reduzieren, werden dabei Partitionen von Daten gebildet oder ein Rahmenwerk für die aufgabenbasierte parallele Programmierung verwendet und pro Datum eine Aufgabe erstellt.

2.1.2 Parallele Reduktion

Eine parallele Reduktion ist ein Spezialfall einer parallelisierten Schleife und wird ebenfalls verwendet, um **Datenparallelität** abzubilden. Die Daten sollen hierbei auf ein einziges Element reduziert werden. Typische Beispiele sind die Bildung der Summe einer Menge von Zahlen oder das Bilden einer Matrix durch das Multiplizieren mehrerer Matrizen. Da die Iterationen alle auf die gleiche Variable zugreifen, kann die Schleife nicht direkt parallelisiert werden. Stattdessen müssen Partitionen gebildet werden und Teilergebnisse berechnet werden, die dann anschließend wieder zusammengefasst werden.

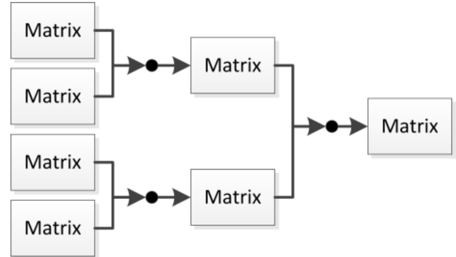


Abbildung 1: Reduktion von Matrizen

In Abbildung 1 wird die Reduktion von Matrizen schematisch dargestellt: Die vier Matrizen werden in zwei Iterationen paarweise multipliziert, bis sich eine einzige Matrix ergibt. Die Multiplikationen in jeder Iteration sind unabhängig voneinander und können daher parallel ausgeführt werden.

2.1.3 Fork-Join

Das Fork-Join-Muster wird verwendet, um **Aufgabenparallelität** zu realisieren. Dieses Muster bezeichnet üblicherweise eine parallele Region in einer sequenziellen Anwendung. Es findet eine Verzweigung statt (*fork*), indem die Berechnungen nebenläufig durchgeführt werden und eine Zusammenführung (*join*), nachdem alle Berechnungen abgeschlossen wurden. Anschließend folgt wieder ein sequenzieller Abschnitt. Dieses Muster findet sowohl in der Aufgabenparallelität als auch bei Datenparallelität Verwendung und ist eher als ein grundlegendes Prinzip zu verstehen und kein Entwurfsmuster, das direkt zur Parallelisierung verwendet werden kann. Auch bei parallelen Schleifen findet dieses Prinzip Verwendung. In dieser Arbeit wird Fork-Join aber auf die nebenläufige Ausführung verschiedener Aufgaben reduziert.

2.1.4 Pipeline (Fließband)

Mit dem Pipeline-Muster kann **flussbasierte Parallelität** erzeugt werden. Dieses Muster orientiert sich an moderne Produktionssysteme, in welchen Bauteile in mehreren Schritten verarbeitet werden. Die einzelnen Verarbeitungsschritte bauen dabei auf dem Ergebnisse des vorhergehenden Schritts auf. In der Softwareentwicklung wird Parallelität dadurch erreicht, dass mehrere Datensätze verarbeitet werden und die Stufen zeitlich überlappt ausgeführt werden. Das Schema einer Pipeline mit drei Stufen ist in Abbildung 2 dargestellt.

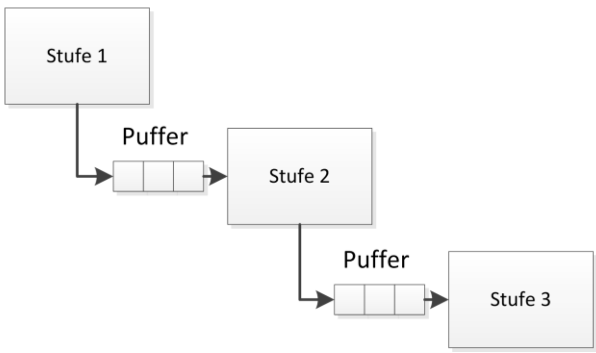


Abbildung 2: Schema einer Pipeline

Jede Stufe besitzt einen Pufferspeicher, in welchen Daten abgelegt werden. Ein Faden pro Stufe entnimmt Daten aus dem Puffer, verarbeitet diese und legt die Daten in den Puffer des Nachfolgers ab. Die Verarbeitungszeit sollte für alle Stufen möglichst gleich lang sein, weil sonst alle Stufen auf die langsamste Stufe warten. Ist eine Stufe nicht darauf angewiesen, dass die Daten innerhalb der Stufe sequenziell verarbeitet werden, können auch mehrere Fäden für eine Stufe verwendet werden. Dies kann dann zum Einsatz kommen, wenn sich die Verarbeitungszeit der Stufen deutlich unterscheidet. Das Muster Producer-Consumer ist ein Spezialfall der Pipeline.

Hierbei werden nur zwei Stufen verwendet, die über einen gemeinsamen Puffer kommunizieren. Die erste Stufe wird dabei als Producer und die Zweite als Consumer bezeichnet.

2.2 Architekturbeschreibungssprachen

Bei der Recherche konnte keine allgemein anerkannte Definition für Architekturbeschreibungssprachen gefunden werden. In einer Studie [MT00] untersuchen die Autoren Sprachen zur Beschreibung von Softwarearchitekturen, um die gemeinsamen Komponenten zu identifizieren und eine Klassifikation vorzunehmen. Dabei bedienen sie sich der Definition von Softwarearchitektur von Shaw und Garlan [SG96]:

Software architecture [is a level of design that] involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns.

Softwarearchitektur beschreibt also die Interaktion zwischen Komponenten einer Software und Muster für Kompositionen aus Komponenten. Mehr Informationen zu Softwarearchitektur können dem Handbuch von Reussner [RR06] entnommen werden. Eine Architekturbeschreibungssprache dient dazu, die konzeptionelle Architektur eines Softwaresystems ohne Berücksichtigung der konkreten Implementierung zu modellieren. Die Verfasser der Studie haben dabei die folgenden Bauteile als wesentlich für alle Architekturbeschreibungssprachen identifiziert:

- **Komponenten (engl. *components*):** Komponenten sind Bausteine der Architektur, die Berechnungen durchführen oder Daten speichern. Für den Datenaustausch stellen diese Schnittstellen (engl. *interfaces*) zur Verfügung.
- **Konnektoren (engl. *connectors*):** Mit Komponenten werden Interaktionen zwischen Komponenten modelliert. Konnektoren müssen nicht Bestandteil des zu beschreibenden Softwaresystems sein, sondern können auch von außen, zum Beispiel über das Betriebssystem zur Verfügung gestellt werden.
- **Konfigurationen (engl. *configurations*):** Konfigurationen sind verbundene Graphen aus durch Konnektoren verbundenen Komponenten. Sie werden benötigt, um festzustellen, ob die Schnittstellen zueinanderpassen und ob die verbundenen Komponenten das gewünschte Verhalten aufweisen.

2.3 Grundlagen für die Implementierung

In diesem Abschnitt sollen einige Grundlagen geliefert werden, die für das Verständnis von Kapitel 6 (Implementierung) notwendig sind. Zuerst wird dabei die verwendete Laufzeitumgebung und Programmiersprache beschrieben und anschließend das Prinzip der aufgabenbasierten parallelen Programmierung erläutert.

2.3.1 .NET Framework und C#

Für die Implementierung werden das .NET Framework und die Programmiersprache verwendet. C# kann als Erweiterung der Programmiersprachen C++ und Java gesehen werden und hat eine sehr ähnliche Syntax. Die Sprachkonstrukte wurden dabei weitestgehend von Java [JAVA] übernommen. Darüber hinaus wurde die Sprache aber auch um neue Funktionen, wie zum Beispiel Elementen aus der funktionalen Programmierung, erweitert. Mit C# können Programme auf Basis des .NET Frameworks entwickelt werden. Das .NET Frameworks ist Microsofts Implementierung des CLI Standards, welcher eine virtuelle Laufzeitumgebung, ein Typ-System und eine gemeinsame Zwischensprache beschreibt [CLI12]. C# Programme werden also wie Java von einer Laufzeitumgebung ausgeführt. Eine Ausführliche Beschreibung der Typen und Sprachkonstrukte des .NET Frameworks am Beispiel der Programmiersprache C# kann in [JR02] zu nachgelesen werden. Für das bessere Verständnis der Mittel die das Betriebssystem Windows für Programmierer zur Verfügung stellt, wie Prozesse und Speicherverwaltung kann [JN11] empfohlen werden.

2.3.2 Aufgabenbasierte parallele Programmierung

In einer Umgebung mit gemeinsam genutztem Speicher wird Parallelität im Allgemeinen über Fäden oder Prozesse erzeugt. Diese werden vom Betriebssystem verwaltet. Das Betriebssystem entscheidet auch, auf welchem Rechenkern der Faden oder Prozess ausgeführt werden soll. Das heißt, es gibt keine Garantie dafür, dass die Fäden wirklich parallel ausgeführt werden. Sollen nur kleinere Aufgaben parallel ausgeführt werden, bietet es sich nicht an eigene Fäden zu verwenden. Deren Erstellung ist relativ teuer, was Laufzeit und Speicherverbrauch betrifft. Wird viel parallelisiert, besteht zudem die Gefahr, dass innerhalb der ganzen Anwendung deutlich mehr Fäden wie Rechenkerne erstellt werden. Insbesondere dann, wenn externe Bibliotheken eingebunden werden, die ebenfalls parallel arbeiten. Deshalb wird bei der aufgabenbasierten parallelen Programmierung von Fäden und Prozessen abstrahiert und Datenstrukturen für Aufgaben angeboten. Eine Aufgabe ist eine begrenzt dauernde Rechenoperation, die parallel zum Hauptfaden ausgeführt werden soll. Diese können verwendet werden, um auch sehr kleine Berechnungen durchzuführen [PB+10]. Das Framework verwaltet eine begrenzte Menge an Fäden und entscheidet, wie die Aufgaben auf diese abgebildet werden. Für die aufgabenbasierte parallele Programmierung steht seit Version 4.0 im .NET Framework die *Task Parallel Library* [TPL] zur Verfügung.

3 VERWANDTE ARBEITEN

Da in dieser Arbeit eine Architektursprache zur automatischen Parallelisierung entwickelt wird, werden in diesem Kapitel Architekturbeschreibungssprachen untersucht. Abschnitt 3.2 ist ein grober Überblick der Sprache TADL. Diese wurde bislang zur Performanzoptimierung verwendet und soll im Rahmen dieser Arbeit für die automatische Parallelisierung ausgebaut werden. Ebenso findet die Sprache XJava Erwähnung, die in Abschnitt 3.3 beschrieben wird, weil der Syntax der darin definierten Sprachelemente für die hier definierte Sprache übernommen und um neue Elemente erweitert werden soll.

3.1 Beschreibungssprachen für parallele Programme

Bei der Untersuchung existierender Architekturbeschreibungssprache wurde auf die Arbeit von Medvidovic und Taylor [MT00] zurückgegriffen, in welcher sich die Autoren mit der Klassifikation von Architekturbeschreibungssprachen beschäftigen. Auffallend ist dabei, dass die untersuchten Sprachen für einen speziellen Einsatzbereich entwickelt wurden. Architekturbeschreibungssprachen dienen also nicht der allgemeinen Beschreibung von Anwendungen. Die dort untersuchten Sprachen können also nicht für diese Arbeit verwendet werden. Trotzdem ist die im Rahmen dieser Arbeit entworfene Sprache nicht von Grund auf neu gestaltet worden, sondern kann als Erweiterung der Sprachen TADL und XJava gesehen werden, die am gleichen Lehrstuhl entwickelt wurden, an dem auch diese Arbeit entstanden ist.

3.2 TADL

Schaefer untersucht in seiner Arbeit [SP+10] wie parallele Programme automatisch auf Performanz optimiert werden können. Dazu hat er die Beschreibungssprache TADL definiert, mit der parallele Architekturen auf der Basis von Entwurfsmustern strukturiert beschrieben werden können. Der Entwickler implementiert atomare Softwarekomponenten, die nicht weiter parallelisiert werden können und beschreibt über TADL, wie diese Komponenten miteinander interagieren. Mithilfe eines Werkzeuges kann daraus lauffähige Software generiert werden.

TADL setzt sich aus zwei Bausteinen zusammen:

1. Atomare Komponenten

Atomare Komponenten sind sequenzielle Rechenaufgaben, die vom Entwickler implementiert werden. Sie sollten keine interne Parallelität besitzen und führen Aufgaben aus, um die Anforderungen, die an die Anwendung gestellt werden, zu realisieren. In der Implementierung von Schaefer mit C# kann eine atomare Komponente mit einer Methode gleichgesetzt werden. Eine atomare Komponente verarbeitet einzelne Datenelemente und kann damit für alle Arten der Parallelität verwendet werden. Wenn die Methode einer atomaren Komponente über keinen geteilten Zustand besitzt, kann die Laufzeitumgebung eine atomare Komponente replizieren und die Kopien parallel ausführen.

2. Konnektoren

Konnektoren definieren die Verarbeitungsstrategie zwischen atomaren Komponenten und wie diese miteinander interagieren. Ein Konnektor hat zwei oder mehr Kinder, wobei sowohl atomare Komponenten als auch Konnektoren als Kinder verwendet werden können. Somit ergibt sich eine Baumstruktur, wobei die inneren Knoten und der Wurzelknoten von Konnektoren gebildet werden und die atomaren Komponenten die Blätter repräsentieren. Die folgenden Konnektoren werden von TADL unterstützt:

1. **Sequenzen:** Mit dem Sequenzoperator kann definiert werden, dass die Kinder dieses Operators hintereinander sequenziell aufgerufen werden. Damit können Teile des Programms voneinander getrennt werden, welche aufgrund von Abhängigkeiten nicht parallel ausgeführt werden können.

2. **Alternativen:** Dieser Konnektor drückt aus, dass die Kinder funktional gleichwertige Alternativen sind. Der Auto-Tuner, ein von Schaefer entwickeltes Optimierungsprogramm, wählt vor der Ausführung des mit Hilfe von TADL erstellten parallelen Programms eine der Möglichkeiten aus und misst die Laufzeitveränderungen.
3. **Fork Join:** Mit diesem Operator wird Aufgabenparallelität eingeführt. Die über diesen Operator verbundenen Kinder werden parallel ausgeführt und mit der Abarbeitung des ursprünglichen Fadens wird fortgefahren, wenn alle Kinder beendet wurden.
4. **Pipeline:** Die Kinder dieses Konnektors werden in einer Pipeline ausgeführt. Daten werden dabei von einer Stufe bearbeitet, in den Pufferspeicher der nächsten Stufe abgelegt, von dieser entnommen und wiederum verarbeitet.
5. **Producer-Consumer**: Dieser Konnektor kann als Spezialfall des Pipeline-Konnektors betrachtet werden. Es müssen immer zwei Kinder angegeben werden. Ein Produzent verarbeitet eine Liste von Daten und legt diese in einen Pufferspeicher ab, der von einem Konsumenten abgearbeitet wird.

Alle Komponenten haben dabei einen ähnlichen Syntax, der anhand eines Beispiels erläutert werden soll:

```
TunablePipeline ImagePipeline {
      TunableAlternative {
            AC_ReduceImage1,
            AC_ReduceImage2
      },
      AC_Filter1,
      AC_Filter2 [replicable]
}
```

In diesem Beispiel sollen mehrere Bilder durch eine Pipeline in mehreren Schritten bearbeiten werden. Zuerst wird eine Reduktion der Bildgröße durchgeführt, wobei zwei alternative Implementierungen bereitgestellt werden. Anschließend werden die Bilder hintereinander durch zwei Filter bearbeitet. Die atomaren Komponenten müssen dabei mit dem Präfix `AC_` und dem Namen der atomaren Komponenten gekennzeichnet werden. Die die Verarbeitungszeit des zweiten Filters länger als die Zeit des ersten Filters ist wurde dieser mit dem `[replicable]` Attribute versehen um die Verarbeitung einzelner Bilder in dieser Stufe parallel durchzuführen. Konnektoren werden über einen eindeutigen Namen angegeben, die Kinder können durch Komma separiert innerhalb der geschweiften Klammern aufgelistet werden.

3.3 XJava

Otto und Pankratius entwickeln mit XJava eine Erweiterung von Java [OP+09], mit der Softwareentwickler auf einem hohen Abstraktionsniveau parallelen Quelltext in Java entwickeln können. Entwickler sollen sich nicht um die oftmals tückischen Details der parallelen Programmierung, wie zum Beispiel die manuelle Synchronisation, kümmern müssen. Dabei soll vor allem die Wartbarkeit und die Lesbarkeit des Quelltexts verbessert werden. XJava besteht aus zwei Bestandteilen: Parallel ausführbare Aktivitäten werden als Aufgabe *(*engl. *task)* bezeichnet. Diese können mit parallelen Ausdrücken (engl. *parallel statement*) verbunden werden. Beide Sprachelemente sollen im Folgenden kurz beschrieben werden.

1. Tasks

Den wichtigsten Baustein bildet dabei ein *task*. Ein *task* ist eine spezielle Methode und stellt eine Aktivität dar, die parallel in einem separaten Faden ausgeführt werden kann. Ein *task* besitzt optional einen typisierten Eingabekanal oder Ausgabekanal und kann mit anderen *tasks* über parallele Ausdrücke verknüpft werden. Wie normale Methoden können *task* Parameter, allerdings keinen Rückgabetypen besitzen. Ein öffentlicher *task* zum Verschlüsseln einer Zeichenkette kann in XJava wie folgt definiert werden:

```
public String => Byte[] Encrypt(String password) { … }
```

Der Eingabekanal nimmt in diesem Beispiel eine Zeichenkette entgegen genommen. Das Ergebnis der Verschlüsselung wird als Byte-Feld in den Ausgabekanal geschrieben.

2. Parallele Ausdrücke

Parallele Ausdrücke dienen dazu mehrere *tasks* über Operatoren zu komplexeren, parallelen Bausteinen zu verknüpfen. Dazu stehen bisher nur zwei Operatoren zur Verfügung:

1. Der "=>" Operator erstellt ein *pipe statement*, indem mehrere *tasks* über die Eingabe- und Ausgabekanäle miteinander verbunden werden. Damit kann mit diesem Operator das Pipeline-Muster realisiert werden. Als Beispielszenario soll gezeigt werden, wie eine Pipeline definiert werden kann, um Dateien zu verschlüsseln. Dazu sollen zusätzlich zu dem oben definierten *task* zwei weitere *tasks* zum Lesen und Schreibe von Dateien definiert werden:

   ```
   public void => String ReadFile(File input) { … }
   public Byte[] => void WriteFile(File output) { … }
   ```

   Die *tasks* können nun zu einer Pipeline verbunden werden. In drei Schritten wird eine Datei ausgelesen, verschlüsselt und das Ergebnis zurück in die Datei geschrieben:

   ```
   ReadFile(file) => Encrypt(password) => WriteFile(file)
   ```

2. Der „|||" Operator erzeugt ein *concurrent statement.* Mehrere *tasks* werden gleichzeitig ausgeführt. Die Ausführung wird fortgesetzt, nachdem beide Aufgaben erfolgreich abgearbeitet wurden. Die so verknüpften *tasks* dürfen keinen Eingabekanal und Ausgabekanal besitzen, die Ergebnisse parallelen Aktivitäten müssen also über globale Zustandsvariablen weitergeben werden. Es ist daher auch nicht möglich, ein *pipe statement* mit einem concurrent statement zu verknüpfen. Als Beispiel soll ein Programm dienen, dass Produktdaten aus einem HTML-Fragment extrahiert. Mit Hilfe von regulären Ausdrücken werden daraus Preis und Beschreibung für das Produkt ermittelt. Dafür werden folgende *tasks* definiert:

   ```
   public void => void ExtractPrice(Html html, Product p) { … }
   public void => void ExtractName(Html html, Product p) { … }
   ```

   Da beide *tasks* unabhängig voneinander sind, können diese parallel ausgeführt werden:

   ```
   ExtractPrice(html, product) ||| ExtractName(html, product)
   ```

4 EIGENER ANSATZ

In diesem Kapitel wird das eigene Konzept zur Quelltexttransformation beschrieben. Die Quelltexttransformation ist nur ein Teil des komplexen Prozesses der Parallelisierung. Deshalb wird im ersten Abschnitt beschrieben, wie dieser Prozess beschaffen ist und welche Position innerhalb dieses Prozesses die Quelltexttransformation einnimmt. Daraus ergeben Ziele und Anforderungen an das eigene Konzept, die im zweiten Abschnitt beschrieben werden soll. In den Folgekapiteln wird dann auf den technischen Entwurf und die Implementierung des hier vorgestellten Ansatzes eingegangen.

4.1 Parallelisierungsprozess

Ein Prozess besteht im Allgemeinen aus hintereinander oder nebenläufig ausgeführten Aufgaben. Damit ergibt sich, dass jede Aufgabe einen Vorgänger oder Nachfolger hat, mit dem ein Austausch an Informationen oder realen Objekten stattfindet. Diese Informationen oder Objekte bilden die Schnittstelle der Aufgabe, welche nur dann beschrieben werden kann, wenn die genaue Position der Aufgabe innerhalb des Prozesses bekannt ist. Deshalb soll in diesem Abschnitt der Prozess der Parallelisierung genauer betrachtet werden und daraus Ziele und Anforderungen an die Quelltexttransformation abgeleitet werden. Leider konnte für die automatische Parallelisierung keine allgemein anerkannte Formulierung eines Prozesses gefunden werden. Das liegt daran, dass sich viele Arbeiten mit der Parallelisierung von einzelnen Kontrollstrukturen wie Schleifen beschäftigen. Im Folgenden sind die Schritte zur manuellen Refaktorisierung von sequentiellem Quellcode dargestellt. Er umfasst die Schritte, die typischerweise notwendig sind um korrekten parallelen Code zu erzeugen. Im Rahmen der *Multicore*-Forschung am Lehrstuhl wird daran gearbeitet, diese wesentlichen Schritte durch Werkzeuge zu unterstützen. Da die vorliegende Arbeit Teil dieser Forschungsbeschäftigung ist, gliedert sie sich auch in den Prozess ein, wie in Abbildung 3 zu erkennen ist.

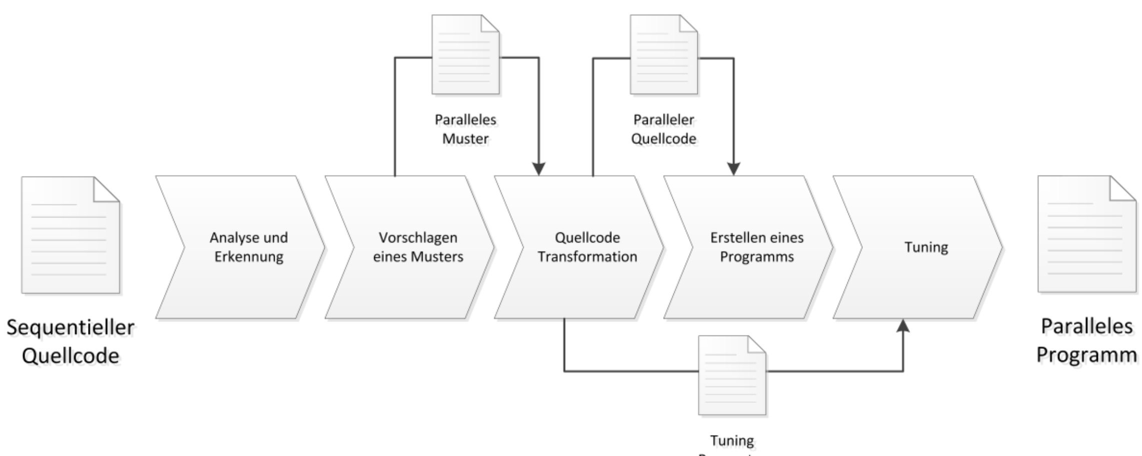


Abbildung 3: Prozess der automatischen Parallelisierung

Als Eingabe wird dem Prozess der sequenzielle Quelltext übergeben und am Ende des Prozesses wird ein paralleles Programm erwartet. Dieser Prozess besteht aus den folgenden fünf Schritten:

1. **Analyse und Erkennung**: Im ersten Schritt wird der sequenzielle Quelltext analysiert um Stellen zu identifizieren, die sich für die Parallelisierung eignen. Wie genau die Analyse durchgeführt werden soll, wird hier offen gelassen. Hierbei besteht noch Forschungsbedarf. Die typischen Probleme bei der parallelen Parallelisierung sind ein Indikator dafür, wie komplex ein solches Analyseverfahren werden kann, insbesondere wenn individuellen Lösungen für jedes analysierte Programm gefunden werden sollen.

2. **Vorschlag einer Parallelisierungsstrategie:** Nachdem ermittelt wurde, wie das untersuchte sequenzielle Programm parallelisiert werden kann, muss die Parallelisierungsstrategie formuliert werden, damit diese für den nächsten Schritt in dem Prozess der automatischen Parallelisierung zur Verfügung steht.
3. **Quelltexttransformation:** Mithilfe des zuvor generierten Parallelisierungsvorschlages wird der ursprüngliche Quelltext entsprechend umgebaut. Dabei können nach Bedarf auch zusätzliche Bibliotheken integriert werden. Diese Bibliotheken können vor der Ausführung des parallelen Programms über Parameter konfiguriert werden. Solch ein Parameter könnte zum Beispiel verwendet werden, um die Anzahl der Fäden zu spezifizieren.
4. **Erstellen eines Programms:** Aus dem transformierten Quelltext erstellt der Kompilierer ein paralleles, ausführbares Programm.
5. **Tuning auf der Zielplattform:** Im letzten Schritt wird das parallelisierte Programm an die Zielplattform angepasst. Dieser Schritt wird Tuning genannt. Mithilfe eines Optimierungsprogramms wird in mehreren Iterationen die Parameterkonfiguration gesucht, bei der das parallele Programm die beste Laufzeit aufweist. In jeder Iteration werden dazu die Parameter gesetzt, das Programm ausgeführt und die Laufzeit gemessen, bis eine ausreichend gute Konfiguration gefunden wurde oder alle Kombinationen aus Parametern untersucht wurden. Dieser Schritt könnte zum Beispiel Teil einer Installationsroutine sein, bei der das Programm an jeden Rechner individuell angepasst wird.

Die Quelltexttransformation, bei der aus einem sequenziellen Quelltext eine optimierte, parallele Version erstellt wird, wird in dieser Arbeit behandelt. Wie aus Abbildung 3 entnommen werden kann, ist es notwendig, dass die Quelltexttransformation Informationen mit den vorhergehenden und nachfolgenden Prozessschritten austauscht: Der parallele Quelltext muss an den nächsten Schritt zur Erzeugung des ausführbaren Programms übergeben werden. Zudem erwartet die Quelltexttransformation den Parallelisierungsvorschlag als Eingabe. Da die Tuningparameter von den Bibliotheken bereitgestellt werden, die während der Transformation hinzugefügt werden, ist es außerdem notwendig, dass eine Liste von verfügbaren Parametern für das Tuning erzeugt wird, die dann von dem Tuningprogramm eingelesen werden kann. Damit ergeben sich zwei Schnittstellen für die Quelltexttransformation zu den Nachfolgern und Vorgänger innerhalb des Prozesses der automatischen Parallelisierung: Es muss definiert werden, wie die gefundenen, parallelen Muster beschrieben werden können und wie Tuningparameter bereitgestellt werden sollen. Insgesamt ergeben sich für diese Arbeit damit die folgenden Teilaufgaben:

1. Definition einer Sprache für parallele Architekturen aus zusammengesetzten Mustern.
2. Definition eines Austauschformates für Tuningparameter.
3. Implementierung eines Werkzeuges zur Quelltexttransformation.

Für jede Teilaufgabe sollen im folgenden Ziele und Anforderungen definiert werden:

4.2 Ziele und Anforderungen

Um auch komplexe Muster mit einer Sprache beschreiben zu können, ist es notwendig, die einzelnen Komponenten dieser Sprache miteinander verknüpfen zu können. Wie in Kapitel 3 geschildert ist das in XJava und TADL nur bedingt möglich. So können die Operatoren in XJava zwar verknüpft werden, allerdings dürfen die Methoden eines parallelen Ausdrucks nicht über Ports mit anderen Methoden, wie zum Beispiel mit den Methoden einer umschließenden Pipeline, verbunden werden. Ziel dieser Arbeit ist es, diese Entwurfsmuster frei kombinieren zu können und auch Daten zwischen den Mustern auszutauschen.

Ziel Z_1: Beschreibungssprache für parallele Muster

In dem Standardwerk über Entwurfsmuster in objektorientierter Software [GH+96] beschreiben die Autoren wie Entwurfsmuster in einem Lernprozess entstehen: Entwickler müssen oft sich wiederholende Probleme lösen. Experten gehen dabei nicht von Grund auf jedes Problem neu

an, sondern greifen auf Lösungen zurück, die sich bereits bewährt haben. Die Autoren haben Software und Projekte studiert, um die Muster für Klassen von Problemen in einem Katalog zusammenzufassen. Dadurch, dass Entwurfsmuster Lösungen für Probleme beschreiben, die in realen Softwareprojekten häufig auftauchen, haben diese eine hohe Relevanz. Es bietet sich daher an, bei der Entwicklung einer Beschreibungssprache auf Muster zu setzen. Damit kann gewährleistet werden, dass die entworfene Sprache auch tatsächlich eingesetzt werden kann. Die Sprache soll dabei nicht von Grund auf neu entwickelt werden, sondern ist die Erweiterung einer am Lehrstuhl entwickelten Sprache für parallele Softwarearchitekturen, genannt TADL (*Tunable Architecture Description Language*). Die Syntax soll vereinfacht und von der Sprache XJava übernommen werden. Außerdem soll zu jedem Baustein der Sprache definiert werden, welche Tuningparameter von der Implementierung für diesen Baustein mindestens bereitgestellt werden müssen. Diese Sprache bildet die Grundlage der Quelltexttransformation. Es ergeben sich die folgenden Anforderungen:

- **Anforderung A1.1:** Es soll eine ausgewählte Menge an Mustern mit dieser Architekturbeschreibungssprache beschrieben werden können.
- **Anforderung A1.2:** Muster sollen beliebig kombiniert werden können. Die zu bearbeitenden Daten sollen dabei zwischen den Komponenten, welche die Muster bilden, ausgetauscht werden. Wenn zum Beispiel innerhalb der Stufe einer Pipeline zwei Aufgaben parallel ausgeführt werden sollen, werden die an diese Stufe übergebenen Daten den parallel auszuführenden Aufgaben übergeben. Nachdem alle Aufgaben innerhalb der Pipelinestufe erfolgreich durchgeführt wurden, werden die Ergebnisse zusammengefasst und an die nächste Stufe der umgebenden Pipeline weitergegeben.

Ziel Z_2: Format für Tuning Parameter

Um dem Optimierungsprogramm die verfügbaren Parameter mitzuteilen, soll ein einfaches Datenformat definiert werden, mit dem diese Parameter beschrieben werden. An dieses Format werden folgende Anforderungen gestellt:

- **Anforderung A2.1:** Die Parameter müssen eindeutige Namen besitzen.
- **Anforderung A2.2:** Jeder Typ muss über einen fest definierten Typen aus der Menge der eingebauten Typen der Programmiersprache des zu transformierenden Quelltexts besitzen. Um den Suchraum für das Tuning zu reduzieren, kann der Wertebereich des Typs mit zusätzlichen Restriktionen eingegrenzt werden.

Ziel Z_3: Werkzeugunterstützte Quelltexttransformation

Der letzte Teil der Arbeit umfasst die Entwicklung eines Werkzeugs zur automatischen Quelltexttransformation. Hierbei sollen folgende Anforderungen umgesetzt werden:

- **Anforderung A3.1**: Unter der Annahme, dass der Parallelisierungsvorschlag eine gültige Lösung ist, soll der Quelltext nach der Transformation weiterhin korrekt arbeiten. Unter Umständen arbeiten die ursprünglichen sequenziellen Methoden die Daten in einer festen Reihenfolge ab und sind darauf angewiesen, dass diese Reihenfolge bei der Parallelisierung erhalten bleibt. Das ist zum Beispiel der Fall, wenn eine Methode ein Datenelement mit dem zuvor verarbeiteten Element vergleicht. Deshalb ist es notwendig, die Reihenfolge von Objekten in Feldern und Listen zu erhalten und die Methoden in der gleichen Reihenfolge wie im sequenziellen Programm auszuführen.
- **Anforderung A3.2:** Es sollen möglichst keine Anforderungen an den ursprünglichen Quelltext gestellt werden. Solch eine Anforderung könnte zum Beispiel darin bestehen, dass eine spezielle Basisklasse implementiert werden muss. Das würde zwar die Quelltexttransformation erleichtern, würde aber auch das Einsatzgebiet auf denjenigen Quelltext einschränken, der mit Hinblick auf eine spätere Parallelisierung entwickelt wurde.
- **Anforderung A3.4:** Das parallelisierte Programm soll skalierbar sein und auch auf Architekturen mit vielen Recheneinheiten möglichst effizient laufen.

- **Anforderung A3.5:** Die verfügbaren Parameter sollen im Rahmen der Quelltexttransformation als Datei dem Tuningprogramm zur Verfügung gestellt werden.
- **Anforderung A3.6:** Das parallele Programm muss die Werte für Tuningparameter, die von einem Tuningprogramm vor der Ausführung des Programms festgelegt und in eine Datei geschrieben wurden, unmittelbar nach dem Start einlesen und anwenden können.

4.3 Zusammenfassung

In diesem Kapitel wurde der Parallelisierungsprozess definiert. Die Rolle der Quelltexttransformation auf Basis einer Architekturbeschreibungssprache innerhalb dieses Prozesses wurde erläutert und die drei Teilkonzepte dieser Arbeit beschrieben. Außerdem wurden Ziele und Anforderungen formuliert. In den folgenden Kapiteln soll nun gezeigt werden, wie diese Teilkonzepte entworfen und implementiert wurden.

5 TECHNISCHER ENTWURF

Im vorangegangenen Kapitel wurde der Parallelisierungsprozess definiert und beschrieben, wie die eigenen Konzepte in diesen Prozess integriert sind. Außerdem wurden die verschiedenen Teilkonzepte genannt und Ziele und Anforderungen festgesetzt. In diesem Kapitel soll nun im Detail auf den technischen Entwurf dieser Teilkonzepte eingegangen werden und wie mit diesem die Ziele und Anforderungen realisiert werden können. Der Entwurf soll dabei unabhängig von der eingesetzten Programmiersprache und Laufzeitumgebung sein. Da die Implementierung mit C# erfolgte, liegen aber alle Beispiele in dieser Programmiersprache vor. Außerdem muss der zu transformierende Quelltext in einer objektorientierten Sprache vorliegen und es müssen Bibliotheken für die aufgabenorientierte parallele Programmierung bereitstehen. Im ersten Abschnitt wird die hier entworfene Architekturbeschreibungssprache beschrieben. Im zweiten Abschnitt wird erläutert, wie Informationen über verfügbare Tuningparameter bereitgestellt werden. Der letzte Abschnitt befasst sich mit der werkzeugunterstützten Quelltexttransformation und beschreibt das Rahmenwerk, mit dem die unterstützen Entwurfsmuster realisiert werden. Außerdem wird beschrieben, wie die Quelltexttransformation durchgeführt wird und wie sich der transformierte Quelltext zur Laufzeit verhält.

5.1 *Tunable Architecture Description Language* (TADL)

TADL ist eine Erweiterung der von Schaefer entwickelten Architekturbeschreibungssprache mit gleichem Namen. Der Syntax wurde dabei von der Sprache XJava übernommen. TADL ist eine textbasierte Sprache, die mit dem Fokus auf Einfachheit entwickelt wurde und daher keinen speziellen Editor benötigt. Ein einfacher Texteditor, wie auf jedem Rechner verfügbar, ist ausreichend. Mit dieser Sprache können parallele Architekturen beschrieben werden, die sich aus Kombinationen von parallelen Entwurfsmustern zusammensetzen. TADL besteht aus zwei Sprachkonstrukten: Die **atomaren Ausführungseinheiten** repräsentieren sequenziell arbeitende Methoden. Die parallele Verarbeitung dieser Methoden wird über **Operatoren** beschrieben. Die beiden Sprachkonstrukte werden in den folgenden Abschnitten genauer beschrieben.

5.1.1 Atomare Ausführungseinheiten (AE)

Atomare Ausführungseinheiten stellen die Komponenten der Architekturbeschreibungssprache dar (vgl. Kapitel 2.2). Es sind Methoden, die von der TADL-Laufzeitumgebung direkt ausgeführt werden. Diese Methoden werden durch den Compiler nicht verändert. Da Daten an Methoden übergeben und die Ergebnisse wieder entgegen genommen werden müssen, werden Bedingungen an die Signatur gestellt: Jede Methode muss genau einen Parameter und einen Rückgabetyp besitzen. Manche Operatoren machen es notwendig, spezielle Signaturen zu verwenden, die dann im jeweiligen Abschnitt zu diesem Operator genannt werden. Um Methoden eindeutig zu identifizieren, wird der vollqualifizierte Namen verwendet. In C# entspricht das der Kombination aus Namensraum, Klassennamen und Methodennamen.

Syntax: *Namespace.ClassName.MethodName*

5.1.2 Operatoren

Operatoren sind die Konnektoren der Architekturbeschreibungssprache (vgl. Kapitel 2.2). Mit diesen kann beschrieben werden, wie atomare Ausführungseinheiten parallel verarbeitet werden sollen. TADL unterscheidet zwischen **unären und binären Operatoren**. Unäre Operatoren verlangen eine AE als Operand, die Operanden eines binären Operators können sowohl atomare Ausführungseinheiten als auch binäre oder unäre Operatoren sein. Damit bildet ein TADL-Ausdruck eine Baumstruktur, wobei der Wurzelknoten und die inneren Knoten von Operatoren gebildet werden und atomare Komponenten die Blätter bilden.

Operatoren können nicht beliebig verkettet werden. Manche Operatoren verlangen eine Liste als Eingabeparameter und können nur dann als Operanden verwendet werden, wenn die zu bearbei-

tenden Daten als Listen vorliegen und der äußere Operator die Daten auch als Liste weitergibt. Zu den meisten Operatoren werden zudem Tuningparameter definiert, die von der Laufzeitumgebung unterstützt werden müssen. Die folgenden Operatoren werden von TADL unterstützt:

Binärer Operator: Replikation

Syntax: *[AE]+*

Der Replikationsoperator definiert, dass eine atomare Ausführungseinheit AE mehrfach parallel zur Ausführung gebracht werden kann. Die von jeder AE zu bearbeitenden Daten müssen dabei als Listen vorliegen. Außerdem muss die Signatur der Methode angepasst werden: Sowohl Parameter als auch Rückgabetyp müssen eine Liste sein. Außerdem darf die Methode keinen gemeinsamen Zustand besitzen. Ansonsten kann die Korrektheit nicht gewährleistet werden. Die im Zuge der Optimierung zu bestimmende Zahl an AE hängt sowohl von der Zahl an freien Kernen auf der Zielplattform ab, vom Umfang der zu verarbeitenden Daten als auch von der Zeit, die für die Bearbeitung eines der Datenelemente erforderlich ist. Wird die Anzahl der AE nicht weiter über den Tuningparameter `MaxParallelism` spezifiziert, wird für jede Recheneinheit eine AE erstellt.

Vor der Bearbeitung der Daten findet eine Dekomposition statt: Die zu bearbeitende Liste wird in Teillisten mit jeweils einem Element zerlegt und diese in einen Puffer ablegt. Die AE entnehmen Teillisten aus dem Puffer nach dem *First-come-first-serve* (FCFS) Prinzip und geben die Ergebnisse ihrer Berechnungen als Liste zurück. Wurden alle Teillisten erfolgreich bearbeitet, findet eine Aggregation statt, indem die Elemente aller Ergebnislisten unter Berücksichtigung der richtigen Reihenfolge zu einer einzelnen Liste hinzugefügt werden. Die Ergebnisse müssen in dieser Liste in der gleichen Reihenfolge vorliegen, wie diese auch von einer nicht replizierten AE zurückgegeben worden wären. Das soll durch folgendes Beispiel verd

```
// AE:
List<string> Execute(List<int> input) {}

// Ursprünglicher Code:
void main() {
    List<int> input = ...;

    List<string> output = Execute(input);
}

// Manuell Optimierter Code:
// In diesem Beispiel wird einer Methode eine Liste von Datenelementen
// übergeben. Diese Aufgabe kann jedoch dadurch parallelisiert werden,
// indem die Liste der Datenelemente verkleinert und die Aufgabe vervielfacht
// wird. Dies kommt in TADL durch Execute+ zum Ausdruck.
void main() {
    List<int> input = ...;
    List<string> output = new List<string>(input.Count);

    Parallel.For(0, input.Count, i => {
        List<int> newInput = new List<int> { input[i] };
        List<string> newOutput = Execute(newInput);
        output[i] = newOutput[0];
    });
}
```

Verwendung innerhalb einer Pipeline:

Wird eine AE als Stufe innerhalb einer Pipeline verwendet, so kann der Replikationsoperator verwendet werden, um zu kennzeichnen, dass eingehende Daten in dieser Stufe parallel verarbeitet werden sollen. Da in einer Pipeline die Datendekomposition vor der Bearbeitung durch die erste Stufe stattfindet, werden an die Stufen Einzelelemente übergeben, die direkt verarbeitet werden können. Die Verteilung der Daten auf die Instanzen der AE erfolgt je nach Wahl des

Pipelineoperators. Um die Reihenfolge der Daten im Datenstrom zu erhalten, findet nach der Bearbeitung der Daten durch die atomare Ausführungseinheit eine Sicherung der Reihenfolge statt.

Tuningparameter

`MaxParallelism` ∈ {0,1,2, ….}
Gibt an, wie oft die AE repliziert werden soll. Jede AE soll dabei von einem Faden ausgeführt werden. Wird der Standardwert 0 verwendet, so werden so viele AE wie Anzahl Kerne auf dem System erstellt.

Unärer Operator: Reduktion

Syntax: *[AE]-*

Der Reduktionsoperator ist ein Spezialfall der Replikation und definiert, dass die AE verwendet werden soll, um eine Liste von Daten zu einem einzelnen Datum zusammenzuführen, indem die Daten paarweise zusammengefasst werden. Dazu muss die Signatur der Methode angepasst werden: Es werden zwei Parameter und ein Rückgabetyp verlangt, die Typen müssen identisch sein. Die Verbesserung der Laufzeit wird dadurch erreicht, dass mehrere Paare baumartig parallel zusammengefasst werden. Soll zum Beispiel eine Liste mit acht Elementen auf einem Rechner mit vier Kernen reduziert werden, können die vier Paare parallel zusammengeführt werden. Anschließend werden die vier Ergebnisse auf zwei Kernen zusammengefasst werden, bis schließlich die letzten zwei Ergebnisse nur noch auf einem Kern zusammengefasst werden. Es müssen die gleichen Tuning Parameter wie bei der Replikation unterstützt werden.

```
// AE:
Set Union(Set left, Set right) {}

// Ursprünglicher Code:
void main() {
    List<Set> input = ...;
    Set finalSet = input.First();
    foreach (Set set in input.SkipFirst()) {
        finalSet = Union(finalSet, set)
    }
}

// Optimierter Code:
// Hier sollen mehrere Mengen vereinigt werden. Der naive Ansatz ähnlich
// des plus Operators kann optimiert werden indem die Mengen parallel immer
// paarweise zusammengefasst werden, bis nur noch eine Menge vorhanden ist.
// Das kann in TADL durch "Union-" ausgedrückt werden.
void main() {
    List<Set> input = ...
    List<Set> temp = new List<Set>();

    while (temp.Count > 1) {
        Parallel.For(0, input.Count / 2, i => {
            Set a = input[i / 2];
            Set b = input[i / 2 + 1];

            temp.Add(input);
        });
        input = temp;
        temp = new List<Set>();
    }
}
```

Binärer Operator: Pipeline

Syntax: *([Operand] => [Operand]) oder ([Operand]=>* [AE]+)*

Eine Pipeline stellt ein paralleles Entwurfsmuster dar, bei dem Daten in einer Art Fließband hintereinander von mehreren Stufen verarbeitet werden. Eine Verbesserung der Laufzeit entsteht dadurch, dass mehrere Daten gleichzeitig verarbeitet werden. Jede Stufe besitzt dazu einen Puffer, in welchen die Daten von der vorgehenden Stufe abgelegt werden, nachdem diese verarbeitet wurden. Für jede Stufe wird ein Faden verwendet, in welchem die Daten aus dem Puffer entnommen, verarbeitet und weitergegeben werden. Sollen mehrere Datenelemente verarbeitet werden, kann die erste Stufe ein Datenelement verarbeiten, während die nächste Stufe den Vorgänger verarbeitet. Die beste Beschleunigung wird dabei erreicht, wenn jede Stufe genau gleich lang rechnet. Da das in einem Programm oft nicht der Fall ist, kann eine zusätzliche Beschleunigung erreicht werden, indem für lang dauernde AE in der Pipeline der Replikationsoperator verwendet wird.

TADL sieht zwei Arten der Datenvermittlung durch den Befehlsstrom vor:

=> Regulärer Befehlsstrom: In diesem Fall werden die Daten in einem Puffer abgelegt. Die Entnahme der Daten aus dem Puffer nach dem *First-come-first-serve* (FCFS) Prinzip.

=>* Broadcast-Pipeline: Diese Art der Datenvermittlung verteilt die Daten an alle Instanzen der AE der folgenden Stufe. Dazu muss der Operand replizierbare atomare Einheit sein.

Die Pipeline nimmt eine Liste entgegen und führt vor der ersten Stufe eine Dekomposition durch. Nach dem letzten Bearbeitungsschritt werden alle Einzelelemente unter Berücksichtigung der korrekten Reihenfolge wieder in einer Liste zusammengeführt und gegebenenfalls an einen Nachfolger weitergegeben.

```
// AE:
Image ReadImage(string file) {}
void Filter1(Image image) {}
void Filter2(Image image) {}
void Save(Image image) {}

// Ursprünglicher Code:
// In diesem Beispiel sollen mehrere Bilder geladen, durch zwei Filter
// bearbeitet und wieder auf die Festplatte geschrieben werden.
// Dieses Programm kann durch eine Pipeline optimiert werden. Im Gegensatz
// zu einer parallel Foreach Schleife bleibt auch der gleichzeitige
// Zugriff auf die Festplatte gering.
// Dies kann in TADL durch ‚ReadImage => Filter1 => Filter2 => Save‛
// ausgedrückt werden.
void main() {
    List<string> files = ...;

    foreach (string file in files)
    {
        Image image = ReadImage(file);
        Filter1(image);
        Filter2(image);
        Save(image);
    }
}
```

Binärer Operator: Fork-Join

Syntax: *([Operand] || [Operand])*

Neben der Datenparallelität unterstützt TADL Aufgabenparallelität. Hierunter versteht man das parallele Ausführen unterschiedlicher Aufgaben. Der Fork-Join-Operator spezifiziert, dass mehrere Operanden parallel ausgeführt werden können. Der TADL-Compiler erstellt eine Barriere, das heißt, mit der Ausführung wird erst fortgesetzt, nachdem alle Teilaufgaben beendet wurden. Das Eingabedatum wird an alle Operanden übergeben. Die Ergebnisse der Operanden werden in einem Tuple zusammengefasst und kann dann anschließend von dem äußeren Operator weiterverarbeitet werden.

```
// AE:
void Execute1(int input) {}
void Execute2(int input) {}

// Ursprünglicher Code:
void main() {
    int input = ...;

    Execute1(input);
    Execute2(input);

    ...
}

// Manuell Optimierter Code:
// Da beide Methoden nur lesend auf die Eingabedaten zugreifen können, kann
// hier optimiert werden, indem beide Methoden parallel ausgeführt werden.
// Um Probleme mit Zustandsänderungen zu vermeiden wird auf die
// Beendigung beider Aufgaben gewartet.
// Dies lässt sich in TADL durch „Execute1 || Execute2“ ausdrücken.
void main() {
    int input = ...;

    Task task1 = new Task(() => Execute1(input));
    Task task2 = new Task(() => Execute2(input));

    Task.WaitAll(task1, task2);

    ...
}
```

Binärer Operator: Ausführungsalternative

Syntax: *([Operand] ? [Operand])*

Je nach Problemstellung kann es für ein Teilproblem mehrere alternative Lösungen geben. Die Wahl der Lösung könnte von der Beschaffenheit der Daten und der Konfiguration des ausführenden Rechensystems abhängen. Es ist die Aufgabe dieses Operators einen Austausch von alternativen Operanden zu ermöglichen. Wird die zu wählende Alternative nicht über den Parameter `AlternativeIndex` spezifiziert, wird der erste Operand gewählt.

```
// AE:
void Quicksort(List<int> input) {}
void Heapsort(List<int> input) {}

// Ursprünglicher Code:
// In diesem Beispiel wird ein Sortierverfahren auf eine Liste angewendet.
// Je nach Aufbau dieser Liste kann das Verfahren QuickSort zu besseren
// Ergebnissen führen als HeapSort.
// In diesem Beispiel könnte in TADL über „QuickSort ? HeapSort"
// das beste Verfahren vermessen werden.
void main() {
    List<int> input = ...;

    if (/* Condition */) {
        Quicksort(input);
    } else {
        Heapsort(input);
    }
}
```

Tuningparameter

`AlternativeIndex` $\in \{0,1,2,\dots,N-1\}, N = \text{Anzahl der Alternativen}$
Gibt an, welcher der alternativen Operanden aufgerufen werden soll.

5.1.3 Sequenzieller Operator

Syntax: *(Operand ; Operand)*

Jedes Programm verfügt über sequenzielle und parallele Abschnitte. In aller Regel wechseln sich diese im Programmlauf auf, sodass ein paralleles Programm ohne Beschränkung der Allgemeinheit als sequenzielles Programm gesehen werden muss, das über parallele Abschnitte verfügt. Da sich getrennte parallele Abschnitte nicht überlappen können, müssen sie auch als getrennt angesehen werden, was die Optimierung anbelangt. Der Sequenzoperator ermöglicht es, aufeinander folgende Programmteile zu definieren.

```
// AE:
Index CreateIndex(List<FileInfo> files) { }
void Search(Index index, string query) { }

// Ursprünglicher Code:
// Hier wird eine Indexstruktur über dem Inhalt von Dateien erzeugt.
// Im Anschluss daran wird der soeben erzeugt Index abgefragt. Es besteht
// die Möglichkeit, die Erzeugung der Indexstruktur oder die Abfrage in
// sich parallel auszuführen, aber die Abfrage selbst kann erst geschehen
// wenn der Index vollständig erzeugt wurde.
// Somit wären in dieser Situation unter anderem folgende
// TADL-Ausdrücke möglich:
// 1. CreateIndex  ; QueryIndex
// 2. CreateIndex+ ; QueryIndex
void main() {
    List<FileInfo> files = ...;
    string query = ...;

    Index index = CreateIndex();
    Search(index, query);
}
```

5.2 Format zur Beschreibung von Tuning Parametern

Für das Tuning ist es notwendig, Informationen über die verfügbaren Parameter zur Verfügung zu stellen. Diese Parameter werden von TADL spezifiziert. Es ist aber auch denkbar, dass das während der Quelltexttransformation injizierte Rahmenwerk weitere Parameter unterstützt. Um konkrete Werte für die Parameter zu spezifizieren, müssen eindeutige Namen für die Parameter vergeben werden. Deshalb werden vollqualifizierte Namen verwendet, die sich aus dem Namen der Operatoren in TADL und dem Namen der Parameter zusammensetzen. Das soll an folgendem TADL-Ausdruck demonstriert werden:

```
(A || B+) ; (C => D)
```

Betrachtet man den Ausdruck als Baum, kommt folgende Struktur zustande:

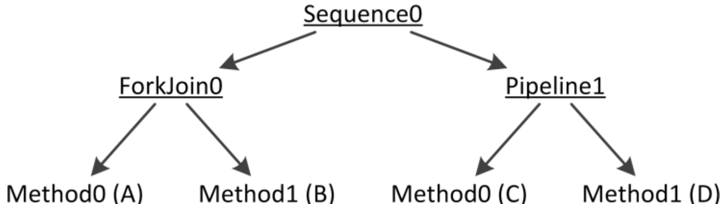


Abbildung 4: Baumstruktur des TADL Ausdrucks

Wie in der Abbildung 4 zu erkennen, wird jedem Element (Operator oder Methode) ein Namen zugeordnet, der sich aus dem Typ und einem mit Null beginnenden Index, der die Position innerhalb des Elternknotens repräsentiert, zusammensetzt. Um ein Element eindeutig zu identifizieren, kann der Pfad innerhalb des Baums verwendet werden. Für den vollqualifizierten Namen des Parameters werden die Elemente dieses Pfades beginnend mit dem Wurzelknoten und zuletzt dem Namen des Parameters mit Punkten separiert aneinandergehängt. Unäre Operatoren werden dabei mit der Methode zusammenfasst, auf dass sie sich beziehen, und kommen daher in obiger Abbildung nicht vor. Auf diese Weise kann die Methode B über den Namen `Sequence0.ForkJoin0.Method1` identifiziert werden. Nimmt man noch den Namen des Parameters hinzu, kann mit `Sequence0.ForkJoin0.Method1.MaxParallelism` die Anzahl der zu erstellenden Fäden spezifiziert werden.

In einer XML-Datei werden die verfügbaren Parameter. Für jeden Parameter wird dabei der Name, der Typ, der Standardwert und zusätzliche Einschränkungen des erlaubten Wertebereichs (*constraints*) definiert. Dazu die XML-Repräsentation von obigem Beispiel:

```
<tuneableParameters>
  <tunableParameter name="Sequence0.ForkJoin0.MaxParallelism"
    type="System.Int32"
    defaultValue="0">
    <constraints>
      <range description="Minimum=1" />
    </constraints>
  </tunableParameter>
</tuneableParameters>
```

Da der volle Pfad Teil der Namen ist, kann noch eine weitere Information gewonnen werden: Um den Suchraum beim Tuning zur verkleinern, können die einzelnen sequenziellen Teile eines Programms unabhängig voneinander optimiert werden. Über die Namen kann bestimmt werden, in welchem sequenziellen Abschnitt ein Parameter Verwendung findet, um damit voneinander unabhängige Gruppen von Parametern zu bilden.

5.3 Werkzeugunterstütze Quelltexttransformation

Für die Entwicklung eines Werkzeugs wird ein Framework benötigt, mit dem die Operatoren aus TADL realisiert werden können. Die Aufgabe des Transformationswerkzeugs besteht dann nur noch darin, den Quelltext zur Initialisierung dieses Rahmenwerks in den ursprünglichen Quelltext zu injizieren und den Teil, in dem die zu parallelisierenden Methoden aufgerufen werden mit Quelltext zum Ausführen des Rahmenwerks zu ersetzten.

5.3.1 Entwurf des Rahmenwerks TPB (Tunable Parallel Blocks)

Das Rahmenwerk hat die Aufgabe, die Operatoren aus TADL zu realisieren und die in der TADL-Datei verwendeten Methoden auszuführen. Da die Methoden unter Umständen auf globale Variablen der umschließenden Klasse zugreifen, muss das Rahmenwerk in die Klasse integriert werden können, in welcher die Methoden definiert sind. Außerdem müssen unmittelbar nach dem Starten der Anwendung die Tuningparameter eingelesen und bei der Ausführung berücksichtigt werden. Es werden also folgende Anforderungen an dieses Rahmenwerk gestellt:

1. Alle TADL-Operatoren müssen unterstützt werden und gemäß den Regeln der Spezifikation verknüpft werden können.
2. Die Datenstrukturen müssen über Tuningparameter konfiguriert werden können.
3. Existierende Methoden müssen direkt aufgerufen werden können.
4. Die Datenstrukturen müssen in eine bestehende Klasse integrierbar sein.

Zu Beginn der Arbeit wurden mehrere existierende Lösungen evaluiert. Da die Implementierung mit C# erfolgen sollte, wurde die Auswahl auf Bibliotheken aus dem .NET Umfeld begrenzt. Außerdem wurden nur Bibliotheken untersucht, die aktiv weiterentwickelt werden und für die eine ausreichend gute Dokumentation vorhanden ist. Dabei wurden folgende Rahmenwerks genauer betrachtet:

1. **Windows Workflow Foundation** [WF]**:** Mit der Workflow Foundation, die Teil des .NET Frameworks ist können Geschäftsprozesse mithilfe eines grafischen Editors modelliert werden und dann als XML-Datei gespeichert werden. WF bietet umfangreiche Möglichkeiten Parameter zu übergeben und Zustände zu verwalten. Außerdem werden Strukturen zur parallelen Verarbeitung bereitgestellt. Erweiterungen können in Form von Aktivitäten entwickelt werden: Damit könnte auch das Pipeline-Muster implementiert werden. Allerdings müssen auch Methodenaufrufe immer als Aktivitäten realisiert werden. Zudem sind ein Workflow und Aktivitäten immer Klassen und lassen sich nicht direkt in eigene Klassen integrieren, so dass in den Methoden nicht auf globale Variablen zugegriffen werden kann.
2. **TPL DataFlow** [TPLDF]**:** Dieses Projekt wurde von der Forschungsabteilung *Microsoft Research* entwickelt und dient dazu, parallele Datenflüsse zu modellieren. Verschiedene Bausteine können miteinander zu komplexen Mustern kombiniert werden. Leider verfügen die eingebauten Komponenten nicht über alle Parameter, die für das Tuning benötigt werden. Diese müssten also neu entwickelt werden, was sich aufgrund der hohen Komplexität und schlechten Dokumentation als zu schwierig gestaltet hat.

Da keine existierende Bibliothek alle Anforderungen unterstützt, wurde ein eigenes System entworfen und implementiert. Dieses Rahmenwerk wird *TPB (Tunable Parallel Blocks)* genannt. Beim Entwurf des Konzeptes diente [GB03] als Vorlage. Obwohl sich das Buch der Beschreibung von Entwurfsmuster für verteilte, nachrichtenbasierte Systeme widmet, kann das Prinzip auch für normale, parallele Programme verwendet werden: Typischerweise besitzen verteilte Systeme keinen globalen Zustand. Eine zentrale Datenbank kann zwar dazu dienen, einen solchen Zustand zu speichern, verringert aber oft die Skalierbarkeit. Außerdem werden bei vielen Systemen auch Dienste von Drittanbieter integriert, deren interne Zustände nicht bekannt sind. Deshalb werden Informationen über Nachrichten ausgetauscht, die alle benötigten Daten enthalten. Ein solches Nachrichtensystem ist skalierbar, relativ einfach zu implementie-

ren und gut erweiterbar, da die einzelnen Komponenten, welche Nachrichten erzeugen oder verarbeiten, prinzipiell unabhängig von anderen Komponenten sind. Das kann auch für ein paralleles Programm übernommen werden, weshalb das Rahmenwerk als nachrichtenbasiertes System entworfen wurde. Im Folgenden soll erst demonstriert werden, wie Nachrichten innerhalb des Rahmenwerks verwendet werden. Anschließend werden die verschiedenen Bausteine des Rahmenwerks aufgelistet und kurz beschrieben.

Nachrichtenbasiertes System

Kern des Systems sind einzelne Komponenten und Nachrichten, mit welchen Daten zwischen diesen Komponenten ausgetauscht werden können.

Zuerst sollen auf die Komponenten eingegangen werden, wobei die Instanz einer solchen Komponente als *Block* bezeichnet wird. Ein *Block* kann wie ein Rechner oder ein Lastenausgleichsystem in einem verteilten System betrachtet werden. Er erfüllt also programmabhängige Aufgaben, indem Methoden ausgeführt werden, oder dient als Vermittler zwischen Blöcken. Dazu kann ein *Block* sowohl Nachrichten entgegen nehmen und auch Nachrichten erzeugen. Um verschiedene Instanzen miteinander zu verketten, kann jeder Block einen oder mehrere Nachfolger haben. Das Versenden einer Nachricht erfolgt, indem die Nachricht an die Nachfolger per Methodenaufruf weitergegeben wird. Um Parallelität zu erzeugen, kann ein Block auch Aufgaben mittels Fäden bearbeiten und auch die Nachricht innerhalb dieses Fadens weitergeben. Deshalb müssen alle *Block* Typen threadsicher implementiert werden.

Der Weg einer Nachricht durch diese verketteten Komponenten kann als Nachrichtenfluss betrachtet werden: Bei Verzweigungen werden aus einer Nachricht neue Nachrichten erzeugt. Bei einer Zusammenführung wird aus einer Menge von Nachrichten eine zusammenfassende Nachricht generiert, welche die Daten der Ursprungsnachrichten in aggregierter Form enthält. Ein solcher Nachrichtenfluss wird in Abbildung 5 beispielhaft dargestellt. Zur Vereinfachung werden nur Methodenaufrufe und Nachrichten betrachtet. Ein mit TPB realisierter Nachrichtenfluss besitzt zudem noch eine Vielzahl an Strukturen, die den Nachrichtenfluss kontrollieren und hier nicht abgebildet sind.

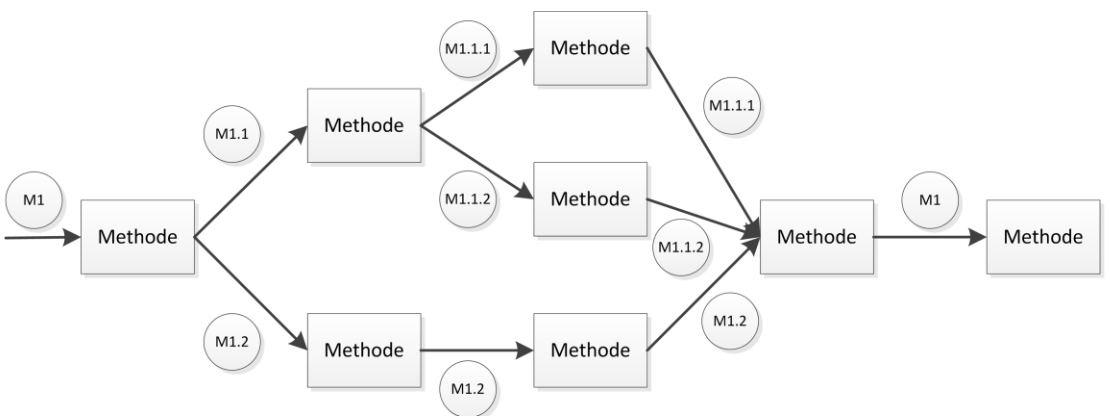


Abbildung 5: Nachrichtenfluss

Jede Verzweigung bedeutet, dass Methoden parallel ausgeführt werden können, wobei auch ohne eine Verzweigung über das Pipeline-Muster parallelisiert werden kann. Die Methoden wurden nicht mit Hinblick auf Threadsicherheit entwickelt. Daher ist es notwendig, dass die Aufrufe in derselben Reihenfolge erfolgen, wie im sequenziellen Fall. Die Sicherstellung dieser Reihenfolge von Nachrichten und auch die Zusammenführung von Verzweigungen stellen die Herausforderungen dieses Konzeptes dar. Diese Aufgabe wird zusätzlich dadurch erschwert, dass die einzelnen Komponenten keine Informationen über den Aufbau des Systems haben und nur ihre Nachfolger kennen. Deshalb müssen alle benötigten Informationen von den Nachrichten selbst bezogen und von diesen transportiert werden.

Dafür werden folgende Metadaten benötigt.

- ***BranchSize***: Eine Ganzzahl, die angibt, wie viel Nachrichten bei einer Verzweigung generiert wurden. Hat ein Block zum Beispiel vier Nachfolger, so werden aus den Ursprungsnachrichten vier weitere Nachrichten erzeugt, wobei jeweils die BranchSize Eigenschaft jeder Nachricht auf vier gesetzt wird.
- ***IterationId***: Eine Ganzzahl welche für jede neue Nachricht, die von dem ersten Block erstellt wurde, hochgezählt wird. Damit kann leicht ermittelt werden, ob die Reihenfolge von Nachrichten, zum Beispiel durch parallele Verarbeitung, vertauscht wurde.

Diese Metadaten liegen in jeder Nachricht auf einem Kellerspeicher vor, bei dem neue Metadaten oben hinzugefügt und von oben entnommen werden können. Die Bedeutung dieses Kellerspeichers soll mit den oben erwähnten Szenarien der Zusammenführung von Verzweigungen und der Sortierung von Nachrichten erläutert werden.

Verzweigungen

Verzweigungen können auf folgende Art geniert werden:

- Hat ein Block mehrere Nachfolger, so werden für eine eingehende Nachricht so viele neue Nachrichten erzeugt, wie es Nachfolger gibt. Die von der Ursprungsnachricht transportierten Daten werden kopiert und die Kopien an die zuvor erstellten Nachrichten übergeben. Auch die Metadaten werden dabei von der ursprünglichen Nachricht übernommen, zusätzlich werden aber neue Metadaten auf den Kellerspeicher jeder neuen Nachricht gelegt, welche über die Eigenschaft `BranchSize` kennzeichnen, wie viele Nachrichten insgesamt für die Verzweigung erstellt wurden. Die neuen Nachrichten werden durchnummeriert und jeder Nachricht wird über die Eigenschaft `IterationId` der obersten Metadaten auf dem Kellerspeicher eine eindeutige Nummer zugeordnet.
- Soll eine Verzweigung zusammengeführt werden, müssen die obersten Metadaten betrachtet werden. Ist der `BranchSize` Wert größer als eins, so muss die Nachricht vorgehalten werden, bis alle anderen Nachrichten eingetroffen sind. Erst dann kann aus diesen eine neue Nachricht erzeugt werden, die dann weitergegeben wird. Diese neue Nachricht enthält die Daten aller zusammengeführten Nachrichten und alle Metadaten von dem Kellerspeicher einer der zusammenzuführenden Nachrichten außer dem obersten Eintrag.
- Mithilfe des Kellerspeichers können auch einfach verschachtelte Verzweigungen realisiert werden. Da die Metadaten nicht ersetzt werden, können Verzweigungen beliebiger Tiefe existieren und innerhalb einer Verzweigung eine neue geöffnet werden. Dabei muss aber zu jeder Verzweigung wieder eine Zusammenführung existieren.

Sortierung

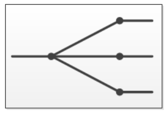
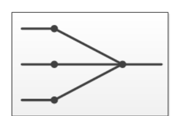
Um Nachrichten zu sortieren, muss nur die Eigenschaft `IterationId` der obersten Metadaten auf dem Kellerspeicher einer Nachricht betrachtet werden. Da diese für jede neue Nachricht hochgezählt wird, kann leicht ermittelt werden, ob die Reihenfolge vertauscht wurde. Dazu muss nur die `IterationId` der zuletzt behandelten Nachricht gespeichert werden. Ist der Abstand der `IterationId` der aktuellen Nachricht zu dem gespeicherten Wert größer als eins, so muss die Komponente auf weitere Nachrichten warten. Da durch Verzweigungen mehrere Nachrichten existieren können, welche die gleiche `IterationId` haben, dürfen nur diejenigen Nachrichten verglichen werden, welche den gleichen Pfad genommen haben, was leicht aus den Metadaten auf dem Kellerspeicher ermittelt werden kann.

Das generieren und behandeln von Nachrichten und das Ausführen von Methoden ist Aufgabe der verschiedenen Block Typen, die im nächsten Abschnitt beschrieben werden sollen.

Beschreibung der verfügbaren Block Typen

In der folgenden Tabelle werden die Block Typen aufgelistet und erläutert. Für jeden Typen wird dabei ein grafisches Symbol verwendet.

Symbol	Beschreibung
Method	
→ M →	Dieser Block ist für Ausführung einer Methode zuständig und stellt damit den einfachsten Typ dar. Optional kann er einen Nachfolger haben. Die auszuführende Methode muss einen Rückgabewert liefern und muss genau einen Parameter haben, wobei der Typ des Parameters dem Typ der eingehenden Nachrichten und der Typ des Rückgabewertes dem der ausgehenden Nachrichten entsprechen muss. Da Nachrichten typisiert sind, kann die eingehende Nachricht nicht verwendet werden, um den Rückgabewert zu transportieren. Deshalb wird eine neue Nachricht erstellt, wobei die Metadaten von der eingegangen Nachricht übernommen werden. Für die Replikation und Reduktion werden Varianten dieses Typs verwendet, welche die Replikation und Reduktion durchführen und abhängig von den Daten die Methode mehrmals ausführen.
Task	
	In diesem Block werden eingehende Nachrichten in eine Queue eingefügt und von einem oder mehreren Threads bearbeitet. Die Anzahl der maximalen Threads kann über einen Parameter eingestellt werden. Werden mehrere Threads zur Bearbeitung der Nachrichten verwendet, kann die Reihenfolge der ausgehenden Nachrichten nicht garantiert werden.
Sorter	
	Um die Reihenfolge garantieren zu können, steh der *Sorter Block* zur Verfügung. Er stellt sicher, dass die ausgehenden Nachrichten in der richtigen Reihenfolge weitergegeben werden. Wird festgestellt, dass zu einer eingehenden Nachricht einer oder mehrere Vorgänger noch nicht behandelt wurden, so wird diese Nachricht erst weitergegeben, wenn auch alle Vorgänger eingetroffen sind. Dieser Block wird immer in Kombination mit einem *Task Block* verwendet.
Splitter und *Aggregator*	
	Diese Komponenten dienen der Behandlung von Listen. Empfängt der *Splitter* (erstes Symbol) eine Nachricht, die eine Liste von Elementen transportiert, so werden für alle Elemente dieser Liste einzelne Nachrichten erstellt und weitergegeben. Die Metadaten der einzelnen Nachrichten werden dabei von der Ursprungsnachricht übernommen. Darüber hinaus werden neue Metadaten auf den Stack gelegt. Damit enthält jede Nachricht die Information, welchen Teil der ursprünglichen Liste sie repräsentiert und wie groß diese Liste ursprünglich war. Diese Informationen werden vom *Aggregator Block* (zweites Symbol) benutzt um aus den einzelnen Nachrichten wieder eine einzelne Nachricht zu bilden, die eine Liste mit den Daten der ursprünglichen Nachrichten enthält.
Alternative	
	Mit dem Alternative Block können verschiedene alternative Implementierungen getestet werden. Damit findet dieser Block besonders in der Performanz Optimierung Verwendung. Der Block kann mehrere Nachfolger haben, von denen der erste Block ausgewählt wird, der über einen Tuning Parameter

	spezifiziert wurde.
Branch und *Join*	
 	Diese beiden Blöcke können verwendet werden, um mehrere Aufgaben parallel auszuführen. Der *Branch Block* (erstes Symbol) ist neben dem *Alternative Block* der einzige Block der mehrere Nachfolger haben kann und gibt ein Nachricht an diese weiter, indem neue Nachrichten erzeugt werden. Als Nachfolger könnten beispielsweise *Task Blocks* mit Methoden verwendet werden, um eine einfache Parallelität zu erzeugen, der Block könnte aber auch mit komplexen Konstruktionen wie einer Pipeline verknüpft werden. Damit eine solche Verzweigung wieder zusammengeführt werden kann, enthalten die Nachrichten neue Metadaten, welche die Größe der Verzweigung und den Index in der Verzweigung enthält. Mit diesen Metadaten kann der *Join Block* (zweites Symbol) Verzweigungen wieder zusammenführen. Dazu werden alle Nachrichten so lange vorbehalten, bis alle Nachrichten einer Verzweigung eingetroffen sind.

Transformation von TADL zu TPB

Dieser Abschnitt beschreibt wie die Muster, die in TADL durch vordefinierte Operatoren realisiert werden, auch in TPB erzeugt werden können, und stellt für beispielhafte TADL Ausdrücke die entsprechenden Kompositionen aus verlinkten TPB Komponenten dar.

Sequence

```
TADL: (A ; B)
```

Um eine Sequenz zu erstellen, müssen die Blöcke nur verlinkt werden. Wie in Abbildung 6 wird für den obigen TADL-Ausdruck für jede Methode ein *Method Block* erstellt und diese miteinander in der gewünschten Ausführungsreihenfolge verknüpft.

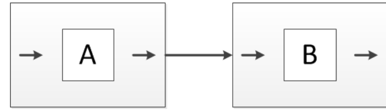


Abbildung 6: Realisierung einer Sequenz in TPB

Pipeline

```
TADL: (A => B+)
```

Da eine Pipeline immer eine Liste von Elementen entgegen nimmt, muss diese Liste zuerst in Einzelelemente zerlegt werden und zuletzt die Einzelelemente wieder als Liste zusammengeführt werden. Dazu wird ein Splitter verwendet. Für jede Methode innerhalb der Pipeline wird ein *Method Block* und ein *Task Block* erstellt. Da jeder Task Block eine Warteschleife zur Abarbeitung der eingehenden Nachrichten besitzt, wird damit das Pipeline-Muster wie in der Literatur beschrieben realisiert. Wurden alle Nachrichten verarbeitet, werden diese über einen *Aggregator* wieder zu einer Liste zusammengefasst.

Ist eine Methode innerhalb einer Pipeline replizierbar, so werden innerhalb des *Task Blocks* mehrere Threads zur Verarbeitung der eingehenden Nachrichten verwenden. In diesem Fall ist die Reihenfolge nicht mehr gewährleistet. Deshalb wird ein *Sorter Block* hinter diese Methoden einfügt, welche die Reihenfolge für nachfolgende Methoden sicherstellt. Wird nur ein Thread verwendet, ist der *Sorter Block* nicht notwendig. Die TPB-Struktur des obigen TADL Ausdrucks ist in Abbildung 7 zu sehen.

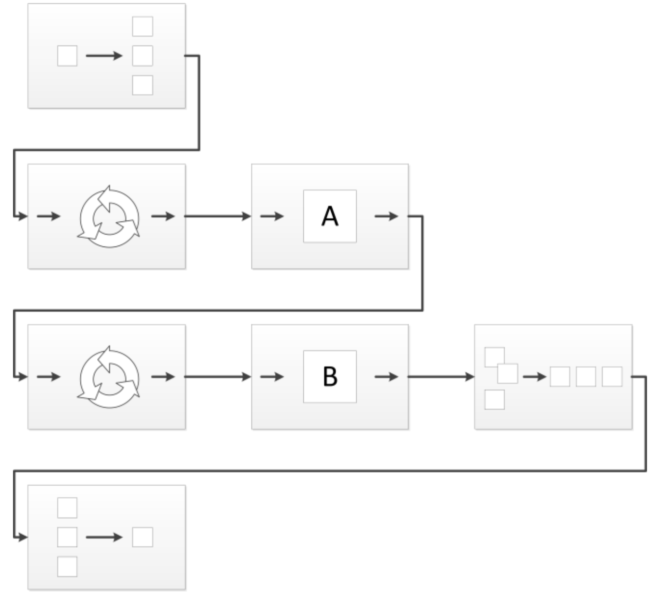


Abbildung 7: Realisierung einer Pipeline mit TPB

ForkJoin

`TADL: (A1 || B1)`

Um das obige ForkJoin Muster zu realisieren, wird ein *Broadcast Block* benötigt, der mit allen Komponenten verbunden wird, welche die parallel auszuführenden Methoden aufrufen. Um die Verzweigung wieder zusammenführen, wird ein *Join Block* benötigt, der wartet, bis alle Methoden erfolgreich ausgeführten wurden. Da keiner dieser Komponenten Parallelität erzeugen kann, wird vor jeden *Method Block* ein *Task Block* eingeführt, der die Ausführung der Methoden in separaten Thread indirekt realisiert.

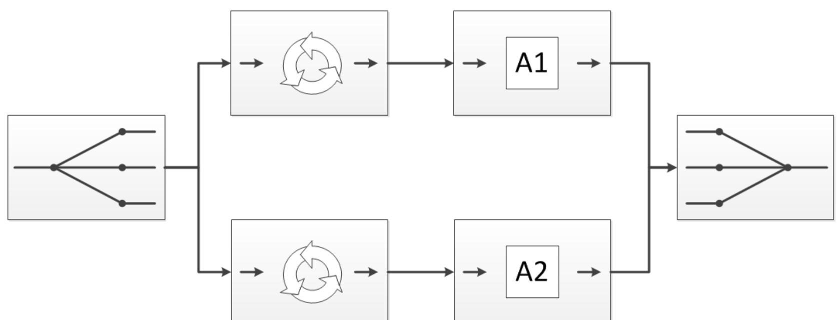


Abbildung 8: Realisierung von Fork-Join in TPB

Alternative

TADL: `((A1 ?? A2) ; B)`

Dieses Muster kann durch einen *Alternative Block* realisiert werden, welcher mit allen *Method Block* Instanzen verbunden wird. Diese werden wiederum mit dem Nachfolger Block verlinkt, wobei in diesem Beispiel wiederum eine Methode aufgerufen werden soll.

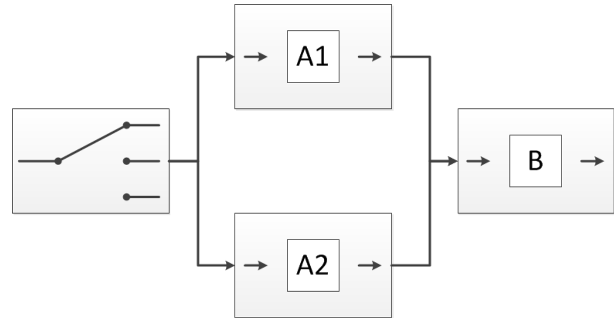


Abbildung 9: Realisierung einer Alternative in TPB

Jedes der oben beschriebenen Muster kann als Block verwendet werden, indem der Eingang der ersten und der Ausgang der letzten Instanz verwendet wird. Dadurch ist es möglich, die verschiedenen Muster beliebig zu kombinieren, jeder *Method Block* in den obigen Beispielen könnte auch durch ein komplexes Konstrukt aus verketteten Komponenten ersetzt werden. Somit ist es möglich, auch komplexe TADL Instanzen nach TPB zu transformieren.

Diese Transformation ist Teil der automatischen Quelltexttransformation, die im letzten Abschnitt dieses Kapitels beschrieben werden soll.

5.3.2 Automatische Quelltexttransformation

Die Transformation des ursprünglichen Programms könnte auf Basis des Quelltexts geschehen oder es könnte direkt die ausführbaren Dateien bearbeitet werden. Bei Systemen mit virtueller Laufzeitumgebung auf Ebene der Zwischensprachen oder auch direkt durch Bearbeiten des Binärcodes. Die Quelltexttransformation ist deutlich einfacher, da oft Bibliotheken zum Einlesen und Bearbeiten des Quelltexts zur Verfügung stehen, die zum Beispiel auch von Entwicklungsumgebungen eingesetzt werden. Außerdem kann vor dem Kompilieren manuell eingegriffen werden und durch den Compiler werden detaillierte Fehlermeldungen erzeugt, wenn der erzeugte Quelltext nicht kompiliert werden kann. Deshalb wurde für diese Arbeit der Weg der Quelltexttransformation gewählt.

Die Quelltexttransformation erfolgt in mehreren Schritten:

1. **Syntaxanalyse:** Zuerst wird eine TADL-Datei eingelesen, auf syntaktische Korrektheit überprüft und ein abstrakter Syntaxbaum erstellt.
2. **Optimierung:** Im zweiten Schritt wird der abstrakte Syntaxbaum untersucht, um Optimierungen durchzuführen.
3. **Validierung:** Anschließend wird der Syntaxbaum unter Zuhilfenahme der zu optimierenden Quelltextdatei validiert. Die Namen der Methoden müssen gültig und in der Datei zu finden sein. Eine Validierung der Typen, das heißt der Parameter und Rückgabetypen der Methoden, findet nicht statt. Eventuelle Probleme können vom Compiler wesentlich effizienter aufgedeckt werden, auch unter Berücksichtigung aller möglichen Sonderfälle, wie zum Beispiel generische Typen und Konvertierungsoperatoren in C#.
4. **Erzeugung Initialisierungsquelltextes:** Auf Basis eines gültigen Syntaxbaums kann die Transformation von TADL in TPB durchgeführt werden, indem der Quelltext zur Erzeugung der TPB-Komponenten und deren Verlinkung durch Methodenaufrufe erzeugt wird. Die Komposition aus verlinkten TPB Komponenten wird anschließend über eine Wrapperklasse zur Verfügung gestellt. Der Initialisierungsquelltext wird in den statischen Konstruktor injiziert oder alternativ eine statische Methode zur Initialisierung erzeugt.
5. **Parameter Ausgabe:** Aus den erzeugten TPB-Komponenten wird ermittelt, welche Parameter zur Verfügung stehen. Die Parameter werden aus den Typinformationen der verschiedenen Block Typen extrahiert und können somit leicht erweitert werden. Diese

werden inklusive der Beschreibung der gültigen Werte über Randbedingungen in eine Datei geschrieben und stehen somit für das Tuning bereit.

6. **Ersetzung Methodenaufrufe:** Der Quelltextabschnitt, welcher ursprünglich diejenigen Methoden beinhaltet hat, die nun Teil der erzeugten TPB-Instanz sind, wird durch einen Aufruf der Wrapperklasse ersetzt. Dazu müssen die Namen der Variablen, welche den Eingangswert und Ausgangswert speichern festgelegt werden und der zu ersetzende Quelltextabschnitt durch Angabe der ersten und letzten Quelltextzeile definiert werden.

5.3.3 Laufzeitverhalten

Da der Quelltext zur Initialisierung in den statischen Konstruktor eingefügt wird, kann sichergestellt werden, dass alle Blöcke initialisiert wurden, bevor diese zum ersten Mal verwenden werden. Unter anderem lesen die Komponenten während der Initialisierung Werte für ihre Parameter aus einer zentralen Konfigurationsdatei aus. Ungültige Werte werden ignoriert. Diese Datei kann von einem Auto-Tuner erzeugt werden. Da weder TADL noch TPB Möglichkeiten bieten, direkt solche Parameter mit Werten zu belegen ist eine erneute Transformation nicht notwendig. Die Trennung von Parameter und Struktur erleichtert somit das Tuning.

5.4 Zusammenfassung

In diesem Kapitel wurde der technische Entwurf zur Realisierung der Konzepte beschrieben, die im vorherigen Kapitel vorgestellt wurden. Die Architekturbeschreibungssprache TADL wurde beschrieben und das Austauschformat für Tuningparameter erläutert. Außerdem wurde ein Rahmenwerk zur nachrichtenbasierten, parallelen Programmierung entworfen und beschrieben, wie dieses Rahmenwerk in der Quelltexttransformation Verwendung findet. Es sollte möglich sein, diesen Entwurf mit einer beliebigen Programmiersprache zu realisieren. Auf die Implementierung mit C# wird im nächsten Kapitel eingegangen.

6 IMPLEMENTIERUNG

Dieses Kapitel beschreibt die Implementierung der in den vorherigen Kapitel beschriebenen Konzepte mit dem .NET Framework und der Programmiersprache C#. Im ersten Abschnitt wird auf den Aufbau des TPB Rahmenwerks eingegangen und anschließend die Implementierung des Transformationsprogramms erläutert.

6.1 Implementierung von TPB

In diesem Abschnitt wird die Implementierung von TPB beschrieben. Aufgrund des Umfangs von mehr als zwanzig Klassen kann nicht auf jede Klasse im Detail eingegangen werden. Stattdessen werden die wichtigsten Basisklassen und Schnittstellen und die Implementierung ausgewählter Blöcke beschrieben.

6.1.1 Nachrichten

Um beliebige Daten typsicher transportieren zu können, wurde die Klasse `Message` generisch gestaltet. Neben den Daten, deren Typ über einen Typparameter definiert wird, speichert jede Nachricht die Metadaten in einem Kellerspeicher. Um den Umgang mit dieser Klasse zu erleichtern, werden Methoden und Eigenschaften zur Verfügung gestellt, um Metadaten auf den Kellerspeicher zu legen oder zu entfernen. Im folgenden Quelltextausschnitt
werden zwei Methoden zum Hinzufügen oder Entnehmen von Metadaten beispielhaft dargestellt.

```
public sealed class Message<T>
{
    private readonly Stack<MessageMetadata> history;

    public T Data { get; set; }

    public void PushMeta(int iterationId, int branchSize)
    {
        history.Push(new MessageMetadata(iterationId, branchSize));
    }

    public MessageMetadata PopMeta()
    {
        return history.Pop();
    }

    …
}
```

Die Metadaten, welche bereits in Kapitel 5 detailliert beschrieben wurden, werden über die Klasse `MessageMetadata` verfügbar gemacht. Einzelne Metadaten sind unveränderbar, die Metadaten einer Nachricht können daher nur über Hinzufügen von Einträgen zum Kellerspeicher der Nachricht oder Entnahme geändert werden. Ein nachträgliches Erhöhen der Eigenschaft `BranchSize` könnte zum Beispiel bewirken, dass ein Block auf Nachrichten wartet, obwohl bereits alle eingetroffen sind.

```
public sealed class MessageMetadata
{
    public int IterationId { get; private set; }

    public int BranchSize { get; private set; }
}
```

6.1.2 Block Klassen

Aufbauend auf diesen Nachrichten werden Blöcke definiert. Um beliebige Nachrichten behandeln zu können, sind alle Typen generisch, wobei der Typ der Daten einer Nachricht an den Typ-Parameter der Klassen gebunden wird. Das erhöht die Performanz und vermeidet die Verwendung von unsicheren Downcasts.

Um Nachrichten empfangen zu können, muss ein Block die Schnittstelle `ITargetBlock` implementieren, wobei als Typparameter der Typ der eingehenden Daten angegeben wird, der von der Nachricht transportiert wird. Über die Methode `Execute`, können Nachrichten an den Block übermittelt werden.

```
public interface ITargetBlock<T>
{
    void Execute(Message<T> message);
}
```

Um Nachrichten an andere Blöcke zu versenden, muss die Schnittstelle `ISourceBlock` implementiert werden. Über den Typparameter wird der Typ der ausgehenden Daten definiert. Mit der Methode `LinkTo` kann ein solcher Block mit jedem Block, der Nachrichten empfangen kann, verbunden werden. Die meisten Block Typen können nur einen Nachfolger haben und werfen eine Ausnahme, wenn der Block mit einem weiteren Nachfolger verbunden werden soll.

```
public interface ISourceBlock<T>
{
     void LinkTo(ITargetBlock<T> block);
}
```

Da für die Behandlung von eigehenden und ausgehenden Nachrichten verschiedene Schnittstellen verwendet wurden, können bei der Implementierung auch verschiedene Typ-Parameter angegeben werden. Somit kann ein Block auch eine Transformation durchführen und den Typ der Nachricht ändern. Dazu muss eine neue Nachricht konstruiert werden, wobei die Metadaten von der ursprünglichen Nachricht übernommen werden.

Tuneable Parameters

Um Parameter zu unterstützen, wird die Klasse `TuneableParameter` bereitgestellt, die den Namen des Parameters, den Typ, den Standardwert und eine Liste der Randbedingungen enthält. Die Instanzen der Klasse können in XML Knoten konvertiert werden und können somit während der Quelltexttransformation direkt in eine Datei geschrieben werden. Bei der Initialisierung der Block Klassen wird jeder Instanz ein eindeutiger Name zugeordnet. Mithilfe des Namens und einer Instanz der `TuneableParameter` Klasse kann der Block beim Konfigurationssystem nach dem Wert eines Parameters nachfragen. Die Deklaration eines Parameters soll an einem Beispiel verdeutlicht werden.

```
public sealed class TaskBlock<T> : FlowBlockBase<T, T>
{
    ...
    public const string MaxDegreeOfParallelismParameterName
        = "MaxDegreeOfParallelism";

    public static TuneableParameter MaxDegreeOfParallelismParameter =
        TuneableParameter.Create<int>(
            MaxDegreeOfParallelismParameterName, 0,
            new RangeConstraint<int>(1, null));

    public override void Initialize(string typeIdentifier)
    {
        int maxDegreeOfParallelism =
            TuneableParameterProvider.GetParameter<int>(
                typeIdentifier,
                MaxDegreeOfParallelismParameter);
        ...
    }
    ...
}
```

Im obigen Beispiel definiert die Klasse `TaskBlock` einen Parameter mit dem Namen `MaxDegreeOfParallelism`, wobei über `RangeConstraint` der Wertebereich auf Zahlen größer als Null eingegrenzt wird. Klasse `TuneableParameterProvider` liest die Parameter aus der Datei aus der Konfigurationsdatei im Arbeitsverzeichnis der Anwendung, falls diese vorhanden ist. In dieser Datei können Parameter zeilenweise im Format

```
[TypeIdentifier].[ParameterName] = [ParameterWert]
```

angegeben werden. Ist der Wert eines Parameters in der Datei gesetzt, kann er in den entsprechenden Typ konvertiert werden und erfüllt dabei die Randbedingungen, so wird der Wert dem Block übergeben, andernfalls wird der Standardwert übergeben.

Implementierte Blöcke

Das in Abbildung 10 dargestellte UML Diagramm zeigt die Hierarchie der implementierten Block Typen. Bis auf eine Ausnahme erben alle Klassen von der gemeinsamen Basis Klasse `FlowBlockBase`, die beide vorgestellten Schnittstellen implementiert und Basisfunktionalität bereitstellt. Die Typen der Eingabe- und Ausgabedaten können jeweils einen eigenen Typparameter angegeben werden.

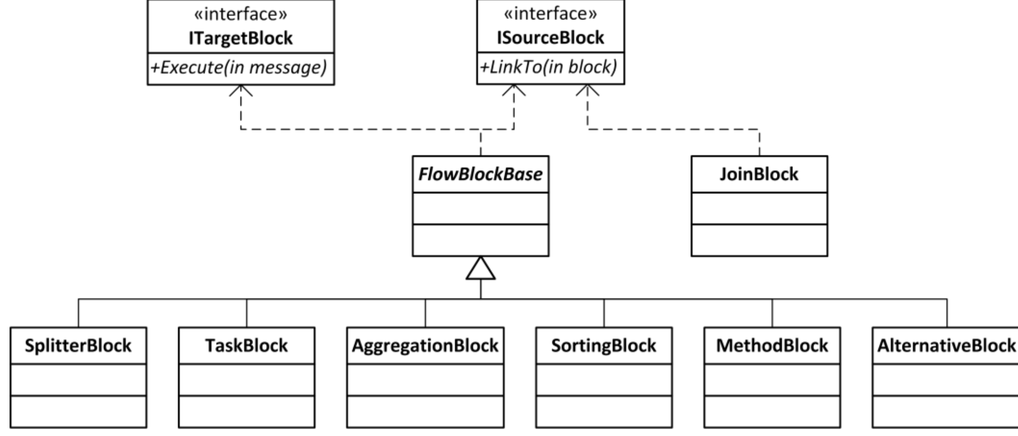


Abbildung 10: Block Hierarchie

Die meisten Block Klassen verwenden aber den gleichen Typen. Die Klasse `JoinBlock` implementiert nur das Interface `ISourceBlock`, obwohl dieser Block auch Nachrichten empfangen kann. Auf diese Besonderheit soll in einem späteren Abschnitt eingegangen werden. Im Folgenden soll nun die Implementierung von drei Blöcken exemplarisch gezeigt werden.

Beispiel Block 1: Method Block

Die Klasse `MethodBlock` im folgenden Quelltextausschnitt verfügt über zwei Typparameter, um den Typ der Eingabe- und Ausgabedaten getrennt angeben zu können. Bei der Initialisierung nimmt sie ein Funktionszeiger entgegen und ruft diesen in der `Execute` Methode auf. Dabei wird das Datum der eingehenden Nachricht als Parameter übergeben. Nachdem die Methode erfolgreich ausgeführt worden ist, wird aus dem Rückgabewert eine neue Nachricht generiert und diese an die Nachfolger übergeben. Die Metadaten werden dabei von der Eingabenachricht übernommen.

```
public sealed class MethodBlock<TIn, TOut> : FlowBlockBase<TIn, TOut>
{
    private readonly Func<TIn, TOut> nestedFunc;

    public override void Execute(Message<TIn> message)
    {
        TOut output = nestedFunc(message.Data);

        Message<TOut> newMessage =
            Message<TOut>.CreateFromParent(output, message);

        ForwardMessage(newMessage);
    }
}
```

Beispiel Block 2: Task Block

Der `TaskBlock` aus dem folgenden Quelltextausschnitt wird verwendet, um Nachrichten parallel zu verarbeiten und weiterzuleiten.

```
public sealed class TaskBlock<T> : FlowBlockBase<T, T>
{
    private TaskScheduler scheduler;

    public override void Initialize(string typeIdentifier)
    {
        scheduler = new LimitedThreadsScheduler(maxThreads);
    }

    public override void Execute(Message<T> message)
    {
        Task task = new Task(ForwardMessage(message));

        task.Start(scheduler);
    }
}
```

Er verwendet die TPL (Task Parallel Library) um die Nachfolger asynchron auszuführen. Die TPL ist ein Rahmenwerk zur aufgabenorientierten parallelen Programmierung und wurde mit .NET 4.0 eingeführt. Damit kann eine Anwendung abhängig von der Anzahl der Recheneinheiten zur Laufzeit optimiert werden. Tasks werden hierbei von einem `Scheduler` ausgeführt. Die Standardimplementierung verwendet dazu den `ThreadPool`, der schon seit Version 1.0 Teil des .NET Frameworks ist. Diese Klasse verwaltet eine Menge an Fäden und führt Aufgaben, die über Methodenzeiger definiert werden mit diesen Fäden aus. Allerdings gibt es keine

Möglichkeit gibt, die Anzahl der verwendeten Fäden einzuschränken. Deshalb wurde mit der Klasse `LimitedThreadsScheduler` ein eigener Scheduler eingeführt, dem bei der Initialisierung die Anzahl der zu verwendenden Threads übergeben werden kann. Damit kann auch nur ein Thread verwendet werden.

Beispiel Block 3: Join Block

Die Klasse `JoinBlock` führt mehrere Verzweigungen wieder zusammen. Da TPB typsicher implementiert ist, gibt es mehrere Implementierungen für zwei bis vier Verzweigungen. Die Klasse implementiert nicht die Schnittstelle `ITargetBlock`, da, falls die zu zusammenzuführenden Verzweigungen den gleichen Ausgangstyp besitzen mehrere Methoden `Execute` mit gleichem Parameter existieren würden. Deshalb stellt die Klasse mehrere Eingänge (engl. *ports*) zur Verfügung, mit denen die letzten Blöcke der Verzweigungen verbunden werden können. Wird eine Nachricht empfangen, so wird überprüft, ob die Nachrichten aus den anderen Verzweigungen mit gleicher IterationId bereits eingetroffen sind. Ist dies der Fall, werden die Daten beider Nachrichten mit der Klasse `Tuple` neu verpackt und weitergeleitet. Andernfalls wird die Nachricht gespeichert, um für die Überprüfung beim Empfang der anderen Nachrichten zur Verfügung zu stehen. Bis auf den obersten Eintrag werden beim Versenden die Metadaten von einer der Ursprungsnachrichten übernommen.

```
public sealed class JoinBlock<T1, T2> : ISourceBlock<Tuple<T1, T2>>
{
    private readonly Dictionary<int, Message<T1>> messages1;
    private readonly Dictionary<int, Message<T2>> messages2;

    public ITargetBlock<T1> Port0 { get; private set; }
    public ITargetBlock<T2> Port1 { get; private set; }

    private void TryForward<T, TOther>(Message<T> message)
    {
        Message<TOther> otherMessage = null;

        if (messages2.TryGetValue(message.CurrentId, out otherMessage))
        {
            messages2.Remove(message.CurrentId);
            message.PopMeta();

            var newMessage = CreateMessage(message, otherMessage);
            ForwardMessage(newMessage);
        }
        else
        {
            messages1[message.CurrentId] = message;
        }
    }
}
```

6.2 Quelltexttransformation

Da mit TPB ein umfangreiches Rahmenwerk eingeführt worden ist, besteht die Aufgabe der Quelltexttransformation nur darin, eine TADL-Architektur einzulesen und in eine TPB-Struktur zu überführen. Der Quelltext zur Initialisierung der Blöcke muss dann in das ursprüngliche Programm integriert werden.

6.2.1 Analyse

Vor der Quelltexttransformation muss der ursprüngliche Quelltext analysiert werden. Es wird überprüft, ob die Methoden existieren und eine gültige Signatur besitzen. Diese müssen statisch sein, über einen Rückgabetypen verfügen und einen oder zwei Parameter entgegen nehmen.

Außerdem werden die Typen der Parameter und Rückgabewerte ermittelt. Für die Analyse wird die Bibliothek *NRefactor* eingesetzt, welche Teil der quelloffenen Entwicklungsumgebung *MonoDevelop* ist und für die Werkzeuge zur Refaktorisierung implementiert wurde. Mit dieser Bibliothek können C#-Dateien eingelesen und in einen abstrakten Syntaxbaum (AST) überführt werden. Durch Verwendung des Entwurfsmusters *Besucher* können leicht eigene Analysen durchgeführt werden, indem die Basis Klasse `DepthFirstAstVisitor` wie in folgendem Quelltextausschnitt verwendet wird.

```
class FindMethodsVisitor : DepthFirstAstVisitor
{
    public override void VisitMethodDeclaration(MethodDeclaration m)
    {
        if (IsValidSignature(methodDeclaration))
        {
            string paramType  = m.Parameters.First().GetText();
            string returnType = m.ReturnType.GetText();

            methods.Add(new Method(paramType, returnType));
        }
    }
}
```

In diesem Beispiel wurde ein eigener Visitor `FindMethodsVisitor` implementiert, um alle Methoden mit einer gültigen Signatur zu ermitteln. Dazu reicht es aus, die Methode `VisitMethodDeclaration` zur überschreiben, detaillierte Kenntnisse über den Aufbau des AST werden nicht benötigt.

6.2.2 Transformation

Sind alle Methoden erfolgreich validiert worden und konnten die Typen ermittelt werden, so findet die Quelltext-Transformation statt.

Beispielhaft soll dazu folgender Quelltext transformiert werden:

```
class Program
{
    static void Main(string[] args)
    {
        int input = 12, output = 0;
        int temp = A(input);
        temp = B(temp);
        output = C(temp);

        Console.WriteLine(o);
    }
}
```

Das Beispiel entspricht dem TADL-Ausdruck:

```
A ; B ; C
```

Es werden mehrere Quelltextteile in das ursprüngliche Programm injiziert. Das wird anhand des transformierten Quelltexts in den Kommentaren aufgezeigt:

```
class Program
{
    // 1. Einführen einer statischen Variable vom Typ BlockExecutor
    private static readonly BlockExecutor<int, int> executor;

    // 2. Statischer Konstruktor
    static Program()
    {
        // 3. Instanziierung der Blöcke
        MethodBlock<int, int> block0 = new MethodBlock<int, int>(A);
        MethodBlock<int, int> block1 = new MethodBlock<int, int>(B);
        MethodBlock<int, int> block2 = new MethodBlock<int, int>(C);
        FlowEndBlock<int> block3 = new FlowEndBlock<int>();
        // 4. Verlinkung der Blöcke
        block0.LinkTo(block1);
        block1.LinkTo(block2);
        block2.LinkTo(block3);
        // 5. Initialisierung der Blöcke
        block0.Initialize("Sequence0.Method0");
        block1.Initialize("Sequence0.Method1");
        block2.Initialize("Sequence0.Method2");
        // 6. Instanziierung des Block Executors
        executor = new BlockExecutor<int, int>(block0, block3);
    }

    static void Main(string[] args)
    {
        int input = 12, output = 0;
        // 7. Ersetzen der ursprünglichen Methodenaufrufe
        output = executor.Execute(input);

        Console.WriteLine(output);
        Console.Read();
    }
}
```

7 EVALUIERUNG

Für die Evaluierung werden Beispielanwendungen untersucht und mithilfe des implementierten Werkzeuges transformiert um die folgenden Fragen zu beantworten:

1. Kann das untersuchte Programm direkt ohne manuelle Änderungen mithilfe des implementierten Werkzeugs transformiert werden oder sind manuelle Anpassungen notwendig? Wie gestalten sich diese Anpassungen?
2. Welche Beschleunigung kann mit der automatischen Quelltexttransformation auf den Testsystemen erreicht werden und wie groß ist der zusätzliche Verwaltungsaufwand durch das verwendete Rahmenwerk? Falls mehrere Alternativen zur Optimierung zur Verfügung stehen: Welche Alternative hat den besten Leistungszuwachs?
3. Wurde die Korrektheit gewährleistet? Sind die richtigen Ergebnisse zurückgegeben worden und liegen diese in der richtigen Reihenfolge vor?

7.1 Beispiele

Die Evaluation erfolgt an drei Beispielen. Zuerst wurden dazu die Originalprogramme analysiert und wenn nötig angepasst und die Laufzeit des sequentiellen Originalprogramms und des transformierten Programms gemessen um die Beschleunigung zu ermitteln. Die Korrektheit wird durch einfache Tests im Quelltext oder manuelle Überprüfung untersucht.

7.1.1 Desktopsuche

Im ersten Beispiel werden alle Textdateien innerhalb eines Verzeichnisses gesucht, die das angegebene Teilwort beinhalten. Die Untersuchung des Originalprogramms hat zwar Parallelisierungspotenzial aufgezeigt, allerdings kann der Optimierer nicht direkt auf den ursprünglichen Quelltext angewendet werden. Das liegt daran, dass die einzelnen Aufgaben nicht sauber auf mehrere Methoden verteilt wurden. Außerdem werden mehrere Klassen verwendet, der Optimierer kann aber nur Quelltext innerhalb einer Klasse und Datei anpassen. Deshalb wurde eine eigene Implementierung erstellt, wobei der Quelltext soweit wie möglich von der ursprünglichen Version übernommen wurde. Nach der Anpassung hat sich folgender Ablauf ergeben:

1. `GetFiles`: Es werden alle Dateipfade im Zielverzeichnis ermittelt.
2. `CreateIndices`: Es werden Indizes erstellt, bei dem einem Dateipfad die Menge der darin gefundenen verschiedenen Wörter zugeordnet wird. Es gibt so viele Indizes wie Dateien, wobei jeder Index nur einen Eintrag besitzt.
3. `InvertIndices`: Die Indizes werden invertiert, jedem Wort wird der Dateipfad zugeordnet.
4. `MergeIndices`: Die Indizes werden zusammengefasst, zu jedem Wort existiert nun eine Liste an Pfaden zu Dateien, in welchen das Wort enthalten ist.
5. `Find`: Die Pfade zu dem zu suchenden Teilwort werden durch Nachschlagen im Index ermittelt.

Das ursprüngliche Programm könnte in TADL wie folgt definiert werden:

```
GetFiles; CreateIndices ; InvertIndices; MergeIndices ; Find
```

Zur Optimierung wurde zuerst die Vereinigung der Indizes mithilfe des Reduktionsoperators verbessert (1). Die Laufzeit konnte dadurch allerdings nur minimal verbessert werden. Das Replizieren des Erstellens der Indizes (2) zeigt aber eine deutliche Verbesserung der Laufzeit. Diese konnte durch Anwenden des Replikationsoperators auf die Umkehrung der Indizes (3) noch mal leicht verbessert werden.

```
(1) GetFiles ; CreateIndices  ; RevertIndices  ; MergeIndices- ; Find
```

```
(2) GetFiles ; CreateIndices+ ; RevertIndices  ; MergeIndices- ; Find

(3) GetFiles ; CreateIndices+ ; RevertIndices+ ; MergeIndices- ; Find
```

Die gemessenen Laufzeiten sind in Abbildung 11 zusammengefasst.

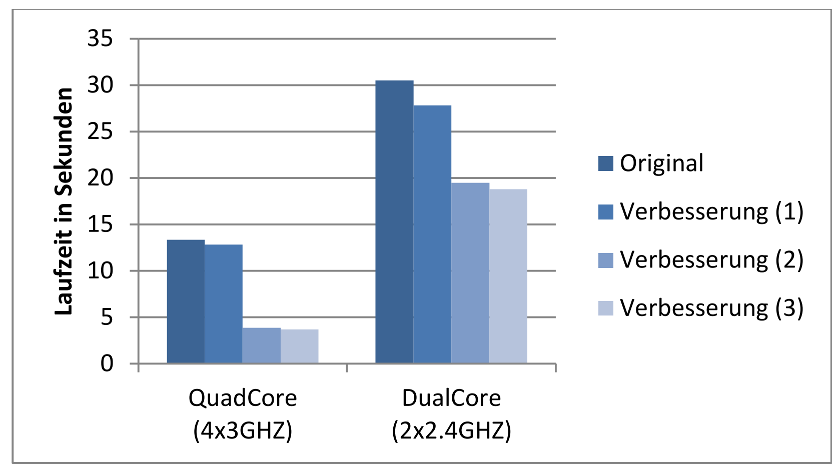


Abbildung 11: Laufzeitmessung Desktop Suche

Die Überprüfung der Korrektheit erfolgt, indem ursprüngliche sequenzielle Programm und die transformierte parallele Version nacheinander ausgeführt werden und die Ergebnisse verglichen werden.

7.1.2 Bilder Konvertierung

Diese Beispielanwendung liest eine Menge von Bildern ein, führt verschiedene Filter aus und speichert diese Bilder anschließend wieder auf der Festplatte. Um den gleichzeitigen Zugriff auf das Dateisystem zu reduzieren, wird das Entwurfsmuster Pipeline verwendet. Hierbei wurde ein eigenes Beispielprogramm geschrieben. Soweit möglich wurden dabei Quelltext und Beispieldateien von einem bereits existierenden Projekt übernommen. Das sequenzielle Programm liest alle Dateinamen aus und führt in einer Schleife mehrere Operationen aus, was durch folgenden TADL-Ausdruck beschrieben werden kann:

```
ReadFile ; AdaptSmoothing ; Blobs ; Convolution ; SaveImage
```

Da eine Pipeline eine Liste von Eingabewerten verlangt, muss auch der Schleifenrumpf ersetzt werden. Die Pipeline kann durch folgenden TADL-Ausdruck beschrieben werden.

```
(1) ReadFile => AdaptSmoothing => Blobs => Convolution => SaveImage
```

Die Laufzeitmessung hat nicht den gewünschten SpeedUp gezeigt, eine Überprüfung mit einem Profiler hat ergeben, dass die Methoden unterschiedlich lange brauchen und besonders der erste Filter die meiste Zeit in Anspruch nimmt. Deshalb wurden in zwei Schritten Replikationen eingeführt:

```
(2) ReadFile => AdaptSmoothing+ => Blobs  => Convolution => SaveImage

(3) ReadFile => AdaptSmoothing+ => Blobs+ => Convolution => SaveImage
```

Die Ergebnisse der Laufzeitmessung können aus Abbildung 12 entnommen werden.

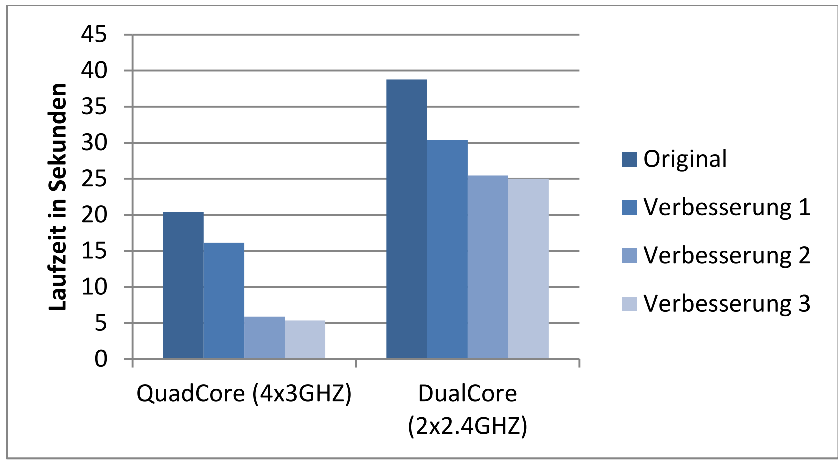


Abbildung 12: Laufzeitmessung Bilder Konvertierung

In diesem Beispiel wurde manuell überprüft, ob das transformierte Programm die korrekten Ergebnisse geliefert hat. Außerdem wurde automatisch überprüft, ob die Reihenfolge erhalten wurde.

7.1.3 Berechnen der konvexen Hülle

Im letzten Beispiel soll die konvexe Hülle zu einer Menge von Punkten berechnet werden. Das Beispiel stammt aus einem quelloffenen Projekt und arbeitet wie folgt: Die Punkte werden erst nach den x und y Koordinaten sortiert und dann die obere und untere Hülle berechnet. Aus beiden Teilergebnissen indem die Punkte der unteren Hälfte zu den oberen Punkte in umgekehrter Reihenfolge hinzugefügt werden. Das genaue Verständnis ist in diesem Fall nicht notwendig, es genügt zu erkennen, dass bei der Berechnung der beiden Teile nur lesend auf die Daten zugegriffen wird. Zur Optimierung war es wieder notwendig eine Anpassung vorzunehmen, da für die komplette Berechnung nur eine Methode verwendet wurde. Die angepasste sequenzielle Version kommt wie folgt in TADL zum Ausdruck:

```
SortPoints ; CreateUpperHull ; CreateLowerHull ; CreateHull
```

Das Programm kann optimiert werden, indem die beiden Teilhüllen parallel berechnet werden:

```
(1) SortPoints ; (CreateUpperHull || CreateLowerHull) ; CreateHull
```

Leider sind die beiden Teilberechnungen nicht aufwendig, die Beschleunigung der Laufzeit fällt daher nur relativ gering aus. Aufgrund geringen Aufwands könnte sich eine Optimierung in realen Anwendungen aber trotzdem lohnen. Die Ergebnisse der Laufzeitmessung können aus Abbildung 13 entnommen werden.

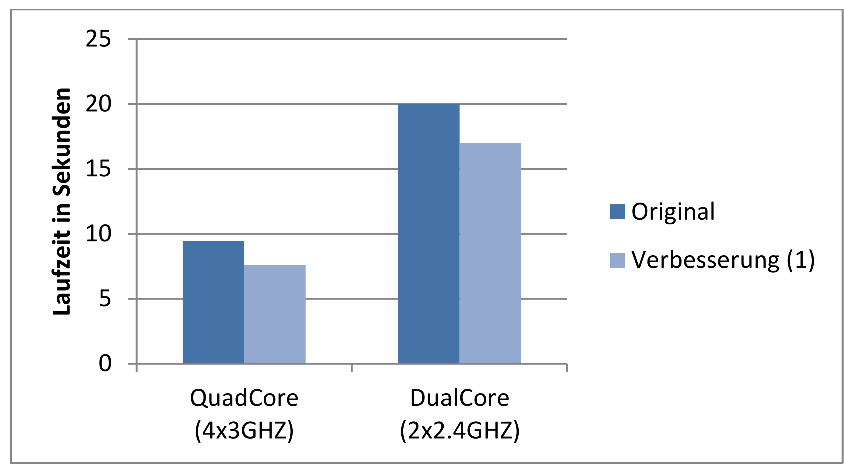


Abbildung 13: Laufzeitmessung konvexe Hülle

7.2 Fazit

Die Evaluierung zeigt, dass in Anwendungsfällen mit Parallelisierungspotenzial durchaus eine gute Beschleunigung der Laufzeit erreicht werden kann. Der zusätzliche Verwaltungsaufwand durch die implementierten Datenstrukturen ist wie Analysen mit einem Profiler zeigen vernachlässigbar. Allerdings zeigt die praktische Arbeit mit dem implementierten Werkzeug eigene Schwachstellen auf:

1. Komplexe TADL Ausdrücke können je nach Benennung der Methoden sehr lang werden. Da der Aufbau horizontal ist, kann der Entwickler immer nur einen kleinen Ausschnitt im Blick halten. Außerdem ist es schwierig, durch die fehlende Einrückung, Strukturen zu erkennen. Die Verschachtelung durch Klammern ist zwar hilfreich aber nicht sehr übersichtlich.
2. Viele Programme sind nicht so in Methoden aufgeteilt, dass diese direkt in TADL verwendet werden können. Oft wird nur eine Methode verwendet oder die Anweisungen anders in Methoden gruppiert, wie es notwendig wäre. Auch die Verwendung vieler Klassen für einen Algorithmus ist problematisch. Damit ist es notwendig, manuelle Anpassungen vorzunehmen, was dem Konzept der nachträglichen automatischen Parallelisierung widerspricht.
3. Durch die einfache Syntax von TADL müssen viele Anforderungen an das Programm gestellt werden, zum Beispiel welche Parameter und Rückgabetypen eine Methode besitzen muss, um verwendet werden zu können. Diese Anforderungen werden von den untersuchten Programmen nicht unterstützt und es konnten auch keine anderen Beispiele gefunden werden, die direkt parallelisiert werden konnten. Zudem befinden sich in vielen Anwendungen kleinere Ausdrücke zwischen Methodenaufrufe, wie zum Beispiel um Typkonvertierungen durchzuführen. Es ist nicht möglich, dies direkt mit TADL auszudrücken.

Insgesamt konnte in der Evaluierung gezeigt werden, dass mit der Implementierung in den untersuchten Beispielen durchaus eine gute Verbesserung der Ausführungszeit erreicht werden konnte. Allerdings war es nicht möglich, direkt existierenden Quelltext ohne manuelle Anpassungen zu optimieren. Bei der Entwicklung neuer Anwendungen kann aber TADL jetzt schon hilfreich sein, um auf einem hohen Abstraktionsniveau Parallelität auszudrücken und verschiedene Varianten zu probieren. Außerdem könnte in Verbindung mit der Schnittstelle für einen Auto-Tuner in diesem Fall schon jetzt ein hoher Mehrwert für Entwickler generiert werden.

8 ZUSAMMENFASSUNG UND AUSBLICK

In dieser Arbeit wurde ein möglicher Prozess der automatischen Parallelisierung vorgestellt und ein Ansatz für den Schritt der Quelltext-Transformation entwickelt und erörtert. Die Idee besteht darin, eine Sprache zur Beschreibung von parallelen Architekturen zu entwerfen und im Transformationsprozess ein Rahmenwerk in den ursprünglichen Quelltext zu injizieren, welches diese Architektur umsetzt. Für das Tuning, also das Anpassen der Anwendung auf eine konkrete Plattform, stellt das Rahmenwerk Parameter zur Verfügung. Eine Liste dieser Parameter wird während der Transformation generiert.

Der Ansatz wurde mit Hilfe von drei Beispielanwendungen evaluiert. Es konnte gezeigt werden, dass eine automatische Quelltexttransformation möglich ist und es konnte eine zum Teil deutliche Verbesserung der Laufzeit gemessen werden. Manuelle Untersuchungen haben auch ergeben, dass der zusätzliche Verwaltungsaufwand durch das injizierte Rahmenwerk mehr als ausgeglichen werden kann. Um die Sprache zur Architekturbeschreibung möglichst einfach zu halten, wurden Anforderungen an den Quelltext gestellt. Bei der Formulierung der Anforderungen wurden die Richtlinien von sauberem Quelltext [RM08] beachtet: Eine Methode soll nur für eine Aufgabe zuständig sein (Prinzip der einen Verantwortlichkeit) und dürfen nur einen Parameter besitzen. Solche Methoden können als atomare Komponenten parallel ausgeführt werden. Es hat sich aber gezeigt, dass alle Beispielanwendungen diese Richtlinien nicht befolgen. Auf der Suche nach weiteren Beispielen konnte ebenfalls kein Quelltext gefunden werden, der alle Anforderungen direkt unterstützt. Oft ist der zu optimierende Quelltextabschnitt innerhalb einer einzigen Methode zu finden. Außerdem werden oft mehrere Parameter verwendet oder eine Methode speichert das Ergebnis in einem globalen Feld, hat also keinen Rückgabetyp. Eine automatische Quelltexttransformation zu realisieren konnte also nicht erreicht werden. Während der Evaluierung mussten die Beispielprogramme erst manuell angepasst werden.

Trotzdem ist das Konzept vielversprechend. Um wirklich beliebige Architekturen beschreiben zu können, müsste die Beschreibungssprache erweitern werden, um die Weitergabe von Parameter und Rückgabewerte zu beschreiben. Außerdem wäre es sinnvoll kleinere Ausdrücke direkt einbetten zu können, um beispielsweise vor der Parameterübergabe eine Konvertierung durchzuführen oder Daten zusammenzufassen. Hier kann die Windows Workflow Foundation einige Ideen liefern. Außerdem müsste der Quelltexttransformation ein Schritt im Parallelisierungsprozess vorgelagert werden, bei dem Abschnitte im Quelltext extrahiert und zu Methoden umgewandelt werden. Mit diesen Maßnahmen könnte das implementierte Werkzeug so erweitert werden, dass das Ziel erreicht werden kann.

ANHÄNGE

A. Abbildungsverzeichnis

Abbildung 1: Reduktion von Matrizen 10
Abbildung 2: Schema einer Pipeline 10
Abbildung 3: Prozess der automatischen Parallelisierung 16
Abbildung 4: Baumstruktur des TADL Ausdrucks 26
Abbildung 5: Nachrichtenfluss 28
Abbildung 6: Realisierung einer Sequenz in TPB 31
Abbildung 7: Realisierung einer Pipeline mit TPB 32
Abbildung 8: Realisierung von Fork-Join in TPB 32
Abbildung 9: Realisierung einer Alternative in TPB 33
Abbildung 10: Block Hierarchie 37
Abbildung 11: Laufzeitmessung Desktop Suche 43
Abbildung 12: Laufzeitmessung Bilder Konvertierung 44
Abbildung 13: Laufzeitmessung konvexe Hülle 45

B. Literaturverzeichnis

[MT00] Nenad Medvidovic and Richard N. Taylor: *A Classification and Comparison Framework for Software Architecture Description* Languages, 2000

[AT05] Andrew S. Tanenbaum: Computerarchitektur. Strukturen - Konzepte - Grundlagen, Addison-Wesley Verlag; 5.Auflage, 2005

[RN11] Jeffrey Richter, Christoph Nassarre: Windows via C/C++, Microsoft Press, 2011

[JR02] Jeffref Richter: *Applied Microsoft .NET Framework Programming*, Microsoft Press, 2002

[RR06] Ralf Reussner: *Handbuch der Software-Architektur*, Dpunkt Verlag, 2006

[SG96] Mary Shaw, David Garlan: *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, April 1996.

[PB+10] Artur Podobas, Mats Brorsson Karl-Filip Faxén: *A Comparison of some recent Task-based Parallel Programming Models*, 2010, Pisa, Italy.

[MS+04] Timothy G. Mattson, Beverly A. Sanders, Berna L. Massingill: *Patterns for Parallel Programming,* Addison-Wesley. 2004.

[OP+09] Frank Otto, Victor Pankratius, Walter F. Tichy: *XJava: Exploiting Parallelism with Object-Oriented Stream Programming,* 2009

[GH+96] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Entwurfsmuster, Elemente wiederverwendbarer objektorientierter Software*, Addision Wesley, 1998

[GB03] Greggor Hohpe, Bobby Woolf: *Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solution,* Addison-Wesley Longman, 2002

[SP+10] Christoph A. Schaefer, Victor Pankratius, Walter F. Tichy: *Engineering Parallel Applications with Tunable Architectures,* 2010

[RM08] Robert C. Martin: Clean Code: *A Handbook of Agile Software Craftmanship*, Prentice Hall International, 2008

[UWT11] *http://www.cs.pitt.edu/~melhem/courses/xx45p/ParallelProg_intelReport.pdf,* zugegriffen am 19.07.2012

[JAVA] Java Spezifikation in der dritten Auflage: *http://docs.oracle.com/javase/specs/jls/se5.0/html/j3TOC.html*

[CLI12] http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf

[TPL] http://msdn.microsoft.com/de-de/library/dd460717.aspx

[WF] http://msdn.microsoft.com/en-us/netframework/aa663328.aspx

[TPLDF] http://msdn.microsoft.com/en-us/devlabs/gg585582.aspx