Karlsruher Institut für Technologie

Fakultät für **Informatik**

Empirische Untersuchung zum Einsatz des Future-Datentyps zur automatischen Parallelisierung

Studienarbeit
von

Christopher Jung
Matrikelnummer 1057809

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

Bearbeitungszeit: 01. Mai 2012 – 31. August 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. August 2012

Christopher Jung

Inhaltsverzeichnis

1 Einleitung 7

2 Grundlagen 8

2.1 Fäden ... 8
2.2 Asynchrone Methodenaufrufe und Futures ... 8
2.3 Datenabhängigkeiten und Wettläufe ... 10
2.4 Codeanalysen ... 11
2.4.1 Statische Codeanalyse ... 12
2.4.2 Kontrollflussgraph ... 12
2.4.3 Aufrufgraph ... 13
2.4.4 Superkontrollflussgraph ... 13
2.4.5 Zeigeranalyse ... 14

3 Verwandte Arbeiten 15

3.1 Automatisierte Parallelisierung mit Auto-Futures ... 15
3.2 Pipeline-Parallelität durch Profiling-Informationen ... 16
3.3 Parallelisierung von Teile-und-Herrsche-Algorithmen ... 17

4 Konzept 18

4.1 Ziele und Anforderungen ... 18
4.2 Codebasis ... 19
4.3 Anwendungsdomänen ... 20
4.4 Voruntersuchung ... 21
4.5 Definitionen ... 22
4.5.1 Block ... 22
4.5.2 Kette von Blöcken ... 22
4.5.3 Kollektion und Iterator ... 22
4.5.4 Hotspot ... 22
4.6 Muster ... 23
4.6.1 M1: Funktionsextraktion ... 23
4.6.2 M2: Spekulative Berechnung ... 24
4.6.3 M3: Schleifenvervielfältigung ... 25
4.7 Automatisches Durchsuchen der Codebasis ... 27
4.8 Vermessung ... 27

5 Implementierung 28

5.1 Microsoft Roslyn ... 28
5.2 Datenabhängigkeitsanalyse ... 29
5.3 Kontrollflussanalyse ... 29
5.4 Mustersuche ... 30
5.4.1 Funktionsextraktion ... 30
5.4.2 Spekulative Berechnung ... 30
5.4.3 Schleifenvervielfältigung ... 30

5.5 Parallelisierung der Muster 31
5.5.1 Funktionsextraktion 31
5.5.2 Spekulative Berechnung 33
5.5.3 Schleifenvervielfältigung 35

6 Evaluation 40

6.1 Untersuchte Projekte 40
6.2 Domänenzuordnung 41
6.3 Verbreitung der Muster 43
6.4 Kosten und Nutzen 48
6.4.1 Funktionsextraktion 49
6.4.2 Spekulative Berechnung 50
6.4.3 Schleifenvervielfältigung 51
6.5 Fazit 52

7 Zusammenfassung und Ausblick 53

Anhänge und Verzeichnisse 54

1 EINLEITUNG

Die zunehmende Verbreitung von Mehrkernprozessoren legt nahe, diese neuen Ressourcen durch parallele Ausführung verschiedener Programmteile zu nutzen. Leider ist der Großteil bestehender Software nicht für parallele Verarbeitung entwickelt worden. Die manuelle Parallelisierung serieller Software ist sehr zeitaufwändig und nur mit großem Fachwissen im Bereich der Softwareparallelisierung machbar.

Einschlägige Literatur zu diesem Thema ist zwar verfügbar [MSM04][MRR12], den meisten Entwicklern steht allerdings weder das notwendige Fachwissen, noch die benötigte Zeit zur Verfügung, um sich in dieses Thema einzulesen bzw. eine händische Parallelisierung durchzuführen. Eine automatisierte Parallelisierung von Bestandssoftware wäre daher für viele Entwickler von großem Nutzen, selbst wenn das Ergebnis nur einen Teil des Leistungszuwachses nutzen könnte, der bei einer manuellen Parallelisierung durch Experten möglich wäre.

Ein bereits etabliertes Konzept zur Beschreibung paralleler Programme ist der asynchrone Methodenaufruf mit Future:

Eine Methode, deren Ergebnis erst zu einem späteren Zeitpunkt benötigt wird, wird frühzeitig (am Abspaltungspunkt) zur Ausführung angestoßen und ein sogenanntes Future wird als Platzhalter für das zukünftige Ergebnis zurückgeliefert. Während die Methode das Ergebnis im Hintergrund berechnet, werden andere Teile des Programms durchgeführt. Sobald das Ergebnis tatsächlich benötigt wird (am Synchronisierungspunkt), wird es beim Future angefragt.

Es werden zwei Situationen unterschieden:

1.) Falls die Methode bereits beendet ist, wird das Ergebnis sofort vom Future zurückgeliefert und der Anfrager kann ohne Unterbrechung weiterarbeiten.

2.) Falls die Methode noch nicht beendet ist, blockiert das Future die Ausführung des Anfragers so lange, bis die Methode endet und das Ergebnis zurückgeliefert werden kann.

Auto-Futures, wie in der Arbeit von Jochen Huck [Huc10] vorgestellt, stellen eine Möglichkeit dar, Bestandssoftware automatisch zu parallelisieren, indem geeignete synchrone Methoden identifiziert und in eine asynchrone Ausführungsform gebracht werden.

Hierbei wird in Software-Quelltext nach Methodenaufrufen gesucht, welche durch asynchrone Methodenaufrufe ersetzt werden, so dass diese parallel zum anschließenden Programmcode ausgeführt werden können. Die Wahl der Abspaltungs- und Synchronisierungspunkte ist hierbei ausschlaggebend für die Korrektheit und Ausführungsgeschwindigkeit des modifizierten Programms. Je näher sich Abspaltungs- und Synchronisierungspunkt am ursprünglichen Methodenaufruf befinden, desto geringer ist die potenzielle Beschleunigung des Programms. Wird der Abspaltungspunkt jedoch zu früh oder der Synchronisierungspunkt zu spät gewählt, so ist die Korrektheit des Programms nicht mehr gewährleistet.

Ziel dieser Arbeit ist es, Anwendungsfälle in Quelltexten zu identifizieren, die mithilfe des Future-Datentyps parallelisiert werden können. Des Weiteren soll untersucht werden, wie häufig diese Anwendungsfälle vorkommen und inwieweit sie automatisiert gefunden werden können. Schließlich sollen mögliche Laufzeitvorteile und zusätzlicher Speicherbedarf eines entsprechend parallelisierten Programms analysiert werden.

2 GRUNDLAGEN

In diesem Kapitel werden grundlegende Begriffe und Konzepte erläutert, welche das Verständnis der folgenden Kapitel verbessern sollen.

2.1 Fäden

Ein Faden bezeichnet einen Ausführungsstrang bei der Abarbeitung eines Programmes. Je nach Konzeption wird ein Faden mit eigenen dedizierten Ressourcen (Arbeitsspeicher, Ausführungseinheiten, …) ausgestattet oder die Ressourcen werden nur für begrenzte Zeit erteilt. Speicherressourcen können dabei vollständig logisch getrennt sein, oder auch zwischen Fäden geteilt werden.

Sind zu einem Zeitpunkt mehr Fäden als Ressourcen vorhanden, so muss entschieden werden, welche Fäden welche Ressourcen zur Verfügung gestellt bekommen, welche Fäden bis zu einem späteren Zeitpunkt warten müssen und wie lange die Ressourcen bei den aktiven Fäden verbleiben dürfen. Diese Entscheidung trifft üblicherweise der sogenannte Scheduler, welcher in modernen Systemen meist als Funktionalität des Betriebssystems bereitgestellt wird und für Anwendungsentwickler und Benutzer vollkommen transparent im Hintergrund arbeitet.

Die Behandlung von Fäden aus Sicht des Betriebssystems wird unter anderem in [Tan07] beschrieben.

2.2 Asynchrone Methodenaufrufe und Futures

Ein asynchroner Methodenaufruf wird, im Gegensatz zu den häufiger verwendeten synchronen Methodenaufrufen, nicht blockierend ausgeführt und besteht aus mindestens zwei Teilen: der Abspaltung und der Synchronisierung. Bei der Abspaltung wird die Ausführung der Methode angestoßen, bei der Synchronisierung wird das Ergebnis abgeholt bzw. auf die Beendigung der Methode gewartet.

Die Begriffe Future [BH77], Promise [FW76] und Eventual [Hib76] sind Bezeichner für gleiche bzw. sehr ähnliche Konzepte. Obwohl es ein paar Unterschiede zwischen den Konzepten Future und Promise gibt, werden Sie im Großteil der Literatur als äquivalent behandelt. Im Folgenden wird daher der Begriff Future stellvertretend für jedes dieser Konzepte benutzt.

Ein Future wird zur Kapselung der Funktionalität eines asynchronen Methodenaufrufs verwendet. Es dient als Platzhalter für den eigentlichen Ergebniswert und wird bei der Abspaltung eines asynchronen Methodenaufrufs zurückgegeben.

Sobald das Ergebnis des asynchronen Methodenaufrufs beim Future angefragt wird (am Synchronisierungspunkt), blockiert die Programmausführung des Anfragers so lange, bis der zugehörige asynchrone Methodenaufruf das Ergebnis berechnet hat. Anschließend wird das Ergebnis an den Anfrager zurückgeliefert und die Programmausführung wird fortgesetzt. Ist das Ergebnis zum Anfragezeitpunkt bereits vorhanden, so wird es sofort zurückgeliefert, ohne die Programmausführung zu blockieren.

In .NET 4 wird das Konzept Future durch die Klassen `Task` und `Task<T>` implementiert [Fre10]. `T` stellt hierbei einen sogenannten generischen Typparameter dar und gibt in diesem Fall den Typ des Rückgabewerts eines Futures an. Weitere Details zu generischen Typen in C# können in [Ric10] gefunden werden.

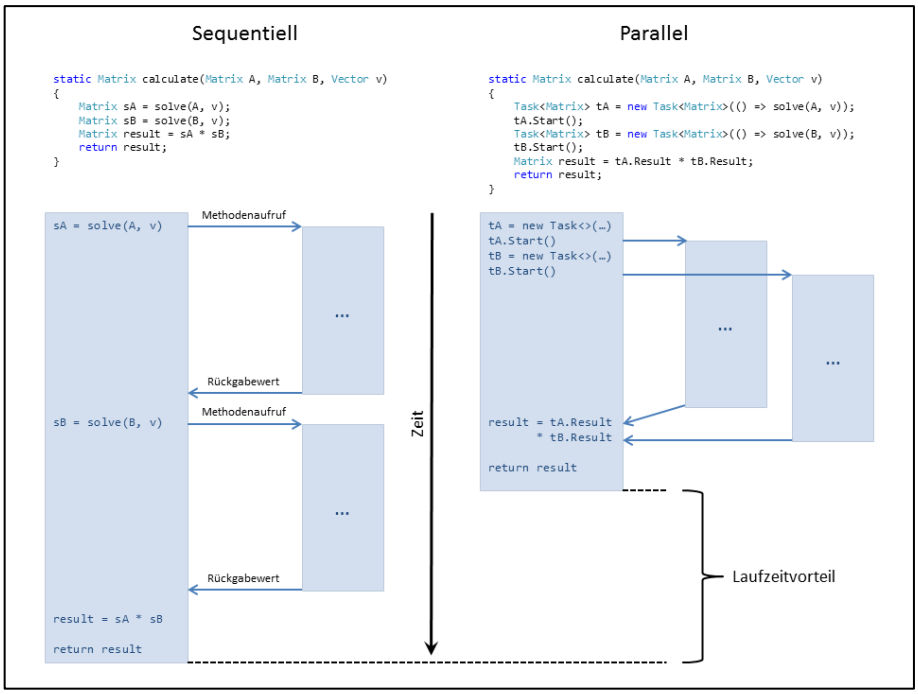

Abbildung 1: Futures in C#

Abbildung 1 illustriert an einem Codebeispiel den Unterschied zwischen sequentieller und paralleler Variante desselben Programms, sowie die zeitliche Verarbeitung ihrer Methodenaufrufe.

Futures werden in vielen Programmiersprachen unterstützt, teilweise direkt durch die Sprache selbst, teilweise durch zusätzliche Programmbibliotheken.

Eine unvollständige Auflistung von Programmiersprachen, die Futures unterstützen ist:

- C++ (mit der Boost-Bibliothek)
- C# (ab .NET 4)
- Common Lisp (mit Eager Future2)
- Flow Java (sprachunterstützt)
- Glasgow Haskell (sprachunterstützt)
- Java (mit java.util.concurrent)
- Javascript (mit dem Dojo Toolkit)
- Lucid (sprachunterstützt)
- MultiLisp (sprachunterstützt)
- Python 3.2 (sprachunterstützt)
- Objective-C (mit MAFuture)
- Perl (mit Reflex)
- Ruby (mit Promise gem)
- Visual Basic (ab .NET 4)

2.3 Datenabhängigkeiten und Wettläufe

Eine Datenabhängigkeit beschreibt den Umstand, dass Daten bzw. Speicherelemente, welche von einer Programmanweisung genutzt werden, auch von einer folgenden Anweisung genutzt werden.

Datenabhängigkeiten kommen grundsätzlich in 4 verschiedenen Formen vor:

- **Echte Datenabhängigkeit / read-after-write:**
  Eine Anweisung liest ein Datum, welches von einer vorangegangenen Anweisung geschrieben wird.
- **Gegenabhängigkeit / write-after-read:**
  Eine Anweisung überschreibt ein Datum, welches von einer vorangegangenen Anweisung gelesen wird.
- **Ausgabeabhängigkeit / write-after-write:**
  Eine Anweisung überschreibt ein Datum, welches von einer vorangegangenen Anweisung geschrieben wird.
- **Eingabeabhängigkeit / read-after-read:**
  Eine Anweisung liest ein Datum, welches von einer vorangegangen Anweisung gelesen wird.

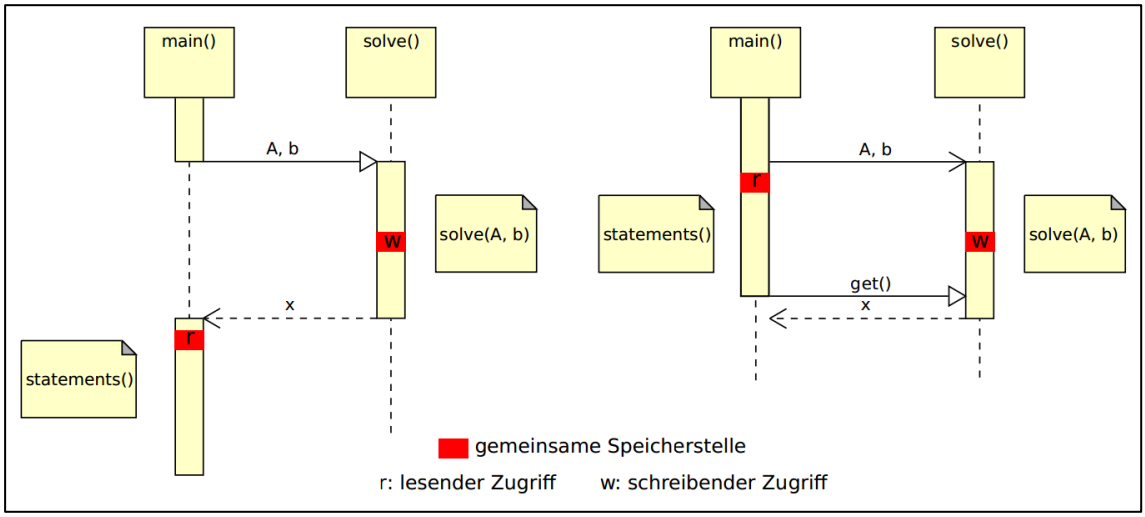

Abbildung 2: Wettlaufbedingung (Bild aus [Huc10])

In sequenziellen Programmen stellen Datenabhängigkeiten üblicherweise kein Problem dar, da die Ausführungsreihenfolge der Programmanweisungen leicht überblickt werden kann.

Bei der parallelen Ausführung mehrerer Programmfäden kann die Ausführungsreihenfolge der Anweisung hingegen nicht mehr so leicht abgeschätzt werden. Dies gilt umso mehr, wenn diese Reihenfolge, wie in den meisten modernen Programmiersprachen bzw. Ausführungsplattformen, von externen Hardware- oder Softwarekomponenten festgelegt werden und sich daher der Kontrolle des Entwicklers größtenteils entziehen. In diesen Fällen können die 3 erstgenannten Datenabhängigkeiten zu sogenannten Wettlaufbedingungen führen. Eine Wettlaufbedingung beschreibt dabei die Situation, dass ein Programm mit mehreren Programmfäden, je nach Ausführungsreihenfolge der Anweisungen, entweder korrekt ablaufen kann (falls die kritischen Anweisungen in einer vom Entwickler vorhergesehen Reihenfolge durchgeführt werden) oder auch nicht.

Abbildung 2 stellt ein Programm mit einer Wettlaufbedingung dar und illustriert, wie es hierdurch bei nicht-sequentieller Verarbeitung zu fehlerhaften Ergebnissen kommen kann. Im Beispiel greifen die Methoden `main()` und `solve()` auf eine gemeinsame Speicherstelle (rot markiert) zu. Im Sequentiellen Fall wird diese zunächst von `solve()` geschrieben und danach von `main()` gelesen. Bei asynchroner Verarbeitung der Methode `solve()` kann es jedoch dazu kommen, dass `main()` die Speicherstelle liest, bevor diese von `solve()` geschrieben wurde. Dies führt dazu, dass ein falsches Ergebnis erzeugt werden kann.

Weitere Details, Beispiele und Zusammenhänge können [ASL08] entnommen werden.

2.4 Codeanalysen

Die Codeanalyse ist eine Methode um Informationen über das Verhalten von Anwendungen zu gewinnen. Sie wird in zwei Hauptkategorien unterteilt:

- Die statische Codeanalyse untersucht ausschließlich Quellcode, ohne Information über Eingabedaten.
- Die dynamische Codeanalyse untersucht Anwendungen während der Laufzeit und kann hierbei auch Eingabedaten mit einbeziehen.

Die statische Codeanalyse kommt klassischerweise in Compilern zum Einsatz, wo sie unter anderem für die Zuordnung von Bezeichnern zu Klassen, Variablen und anderen Sprachkonstrukten eingesetzt wird. In [ASL08] findet der geneigte Leser einen guten Überblick über Quellcodeanalysen im Zusammenhang mit Compilern. Ein weiteres Einsatzgebiet ist die Softwareentwicklung, wo sie zur Sicherstellung von Coding-Standards, zur Überprüfung der Code-Abdeckung automatischer Tests und zum Finden einfacher funktionaler Fehler eingesetzt wird.

Die dynamische Codeanalyse wird u.a. in Just-In-Time-Compilern wie Hotspot [PVC01] oder dem .NET JIT-Compiler verwendet, um Code noch zur Laufzeit zu optimieren. Optimierungsmöglichkeiten sind z.B. Konstantenfaltung (constant folding) [WZ91][Muc97], Schleifenentfaltung (loop unrolling) [BH77][ASL08] und die Entfernung überflüssigen Codes (dead code elimination) [GBJ00][ASL08]. Hierbei wird das zur Laufzeit verfügbare zusätzliche Wissen genutzt, um zusätzliche Geschwindigkeitssteigerungen zu ermöglichen.

Allgemein hat die dynamische Codeanalyse gegenüber der statischen Analyse den Vorteil, Aussagen über das Programmverhalten bei konkreten Eingabedaten machen zu können. Dies sind z.B. Aussagen über die durchschnittliche Ausführungsdauer und Aufrufhäufigkeit von Methoden, weshalb sie auch in sogenannten Profilern zum Einsatz kommt, um besonders rechenintensive oder speicherhungrige Bereiche des Codes zu identifizieren. Eine gute Übersicht über die Nutzung von Profilern für .NET-Code ist in [Gou12] zu finden.

Im Gegensatz dazu hat die statische Codeanalyse vor allem den Vorteil der Unabhängigkeit von Eingabedaten. Alle Wege einer Verzweigung können erkannt und untersucht werden. Die dynamische Analyse kann eine Aussage über den Weg einer Verzweigung ausschließlich dann machen, wenn dieser auch im realen Programmablauf während der Analyse beschritten wurde.

Wegen der genannten Unabhängigkeit von Eingabedaten kommt in dieser Arbeit ausschließlich die statische Codeanalyse zum Einsatz, wobei jedoch eine Kombination mit dynamischer Analyse durchaus denkbar wäre.

2.4.1 Statische Codeanalyse

Statische Codeanalysen lassen sich grob in intra- und interprozedurale Analysen unterteilen.

Bei der intraprozeduralen Analyse werden Methodenaufrufe nicht weiter verfolgt. Seiteneffekte, welche beim Aufruf einer Methode entstehen, werden daher auch nicht berücksichtigt. Für diese Form der Codeanalyse ist ein Kontrollflussgraph des Programms ausreichend.

Interprozedurale Codeanalysen verfolgen hingegen auch jeden Methodenaufruf und können somit potenziell genauere Aussagen über mögliche Programmabläufe und Variableninhalte treffen. Für eine interprozedurale Analyse ist jedoch ein Aufrufgraph des zu analysierenden Programms nötig. Die Erzeugung von Aufrufgraphen in den meisten Fällen jedoch nicht trivial.

Eine gute Einführung in die interprozedurale Codeanalyse kann in [ASL08] gefunden werden.

2.4.2 Kontrollflussgraph

Kontrollflussgraphen sind gerichtete Graphen und beschreiben Programmabläufe. Sie haben einen definierten Wurzelknoten, wobei jeder Knoten im Graph vom Wurzelknoten aus erreichbar sein muss. Üblicherweise repräsentiert jeder Knoten eines Kontrollflussgraphen einen Basisblock. Basisblöcke sind Teile des Quellcodes, die vollkommen linear abgearbeitet werden, d.h. sie enthalten keinerlei Sprünge oder Verzweigungen. Die Kanten repräsentieren Übergänge vom Ende eines Blocks zum Anfang eines anderen. Die Übergänge können mit Bedingungen verknüpft sein, was die Erzeugung von bedingten Sprüngen bzw. Verzweigungen möglich macht.

Methodenaufrufe werden in Kontrollflussgraphen ebenfalls als Basisblöcke dargestellt. Was während dem Methodenaufruf geschieht ist daher im Graph nicht ersichtlich und kann auch während der Codeanalyse nicht untersucht werden. Die Untersuchung von Methodenaufrufen im Kontrollflussgraph ist theoretisch durch Ersetzen jedes Methodenaufrufs durch die zugehörige Methode zwar grundsätzlich möglich, erzeugt jedoch potenziell unendlich große Graphen (z.B. bei rekursiven Methoden) und ist daher nicht praktikabel.

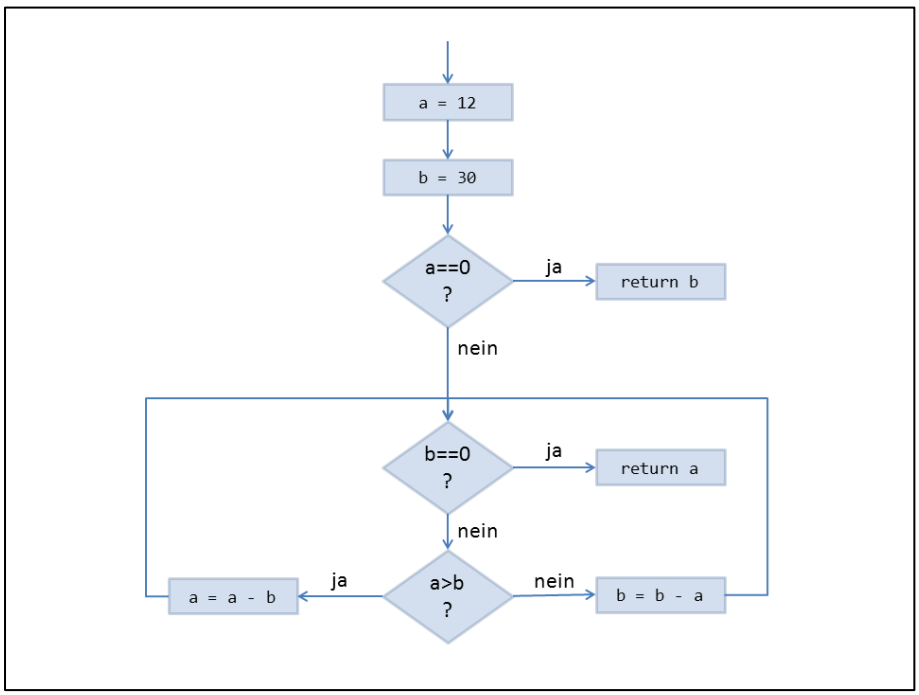

Abbildung 3: Kontrollflussgraph des Euklidischen Algorithmus

Aus diesem Grund werden Kontrollflussgraphen üblicherweise nur für die intraprozedurale Analyse verwendet und enthalten daher meistens auch nur Informationen über den Programmablauf innerhalb einer einzelnen Methode.

Abbildung 3 zeigt einen Kontrollflussgraph des Euklidischen Algorithmus, mit welchem der größte gemeinsame Teiler zweier Ganzzahlen berechnet werden kann. Jeder Knoten entspricht hier einer Anweisung, die Rauten entsprechen bedingten Sprüngen.

Weitere Informationen zu Kontrollflussgraphen sind in [ASL08] zu finden.

2.4.3 Aufrufgraph

Ein Aufrufgraph ist ein gerichteter Graph, der Abhängigkeiten zwischen Methoden beschreibt, welche aus Methodenaufrufen resultieren.

Jeder Knoten des Graphen stellt hierbei eine Methode des Programms oder eine Aufrufstelle dar, d.h. eine Stelle des Programms, an welcher eine Methode aufgerufen wird. Jede Kante stellt einen Methodenaufruf dar. Die Kanten zeigen dabei jeweils von der Aufrufstelle zur aufgerufenen Methode.

Zusätzlich können die Kanten mit Informationen angereichert sein, z.B. über Aufrufparameter, Rückgabewerte oder Speicherzugriffe.

Schwierig wird die Erzeugung von Aufrufgraphen unter anderem im Zusammenhang mit virtuellen Methodenaufrufen. Durch Konzepte wie Vererbung oder Schnittstellen (Interfaces) ist hierbei unter Umständen erst zur Laufzeit des Programms bekannt, welche konkrete Methode durch einen virtuellen Methodenaufruf tatsächlich ausgeführt wird.

Ein weiteres Problem ist der Einsatz von Reflexion [FS99][FF05]. Sogenannte Reflection-Frameworks gestatten es dem Entwickler zur Laufzeit eines Programms beliebige Methoden aufzurufen und neue Objekte oder sogar neue Objekt-Klassen zu erzeugen. Die Namen und Signaturen dieser Methoden und Objekte werden hierbei üblicherweise erst zur Laufzeit durch die Angabe einer Zeichenkette ausgewertet und sind nicht zwingend auf Typen beschränkt, die zur Compile-Zeit bekannt waren.

Falls der entsprechende Zeichenkettenausdruck variabel ist und zur Compile-Zeit nicht ausgewertet werden kann, kann das Ziel eines entsprechenden Methodenaufrufs daher nicht bestimmt werden und der Aufrufgraph ist unvollständig.

2.4.4 Superkontrollflussgraph

Ein Superkontrollflussgraph [ASL08] ist ein erweiterter Kontrollflussgraph. Zusätzlich zu den intraprozeduralen Kanten des Kontrollflussgraphs werden auch Kanten für Methodenaufrufe und –rücksprünge eingeführt. Dabei zeigen jeweils eine Kante von einer Aufrufstelle zum Anfang der aufgerufenen Methode und eine Kante von jeder Rücksprunganweisung der aufgerufenen Methode zurück zur Aufrufstelle.

Dieser Graphentyp ist gut geeignet um einfache interprozedurale Analysen durchzuführen, allerdings werden in diesem Modell Beziehungen zwischen Ein- und Ausgabewerten von Methodenaufrufen ignoriert, was die Ungenauigkeit einer Analyse erhöht.

Das Konzept des Superkontrollgraphs selbst wird in dieser Arbeit nicht verwendet. Die Spezifikation der gefundenen Muster verwendet allerdings einen ähnlichen Graphen (siehe Anhang A.1.1), sodass es sinnvoll erschien, dieses Konzept vorzustellen.

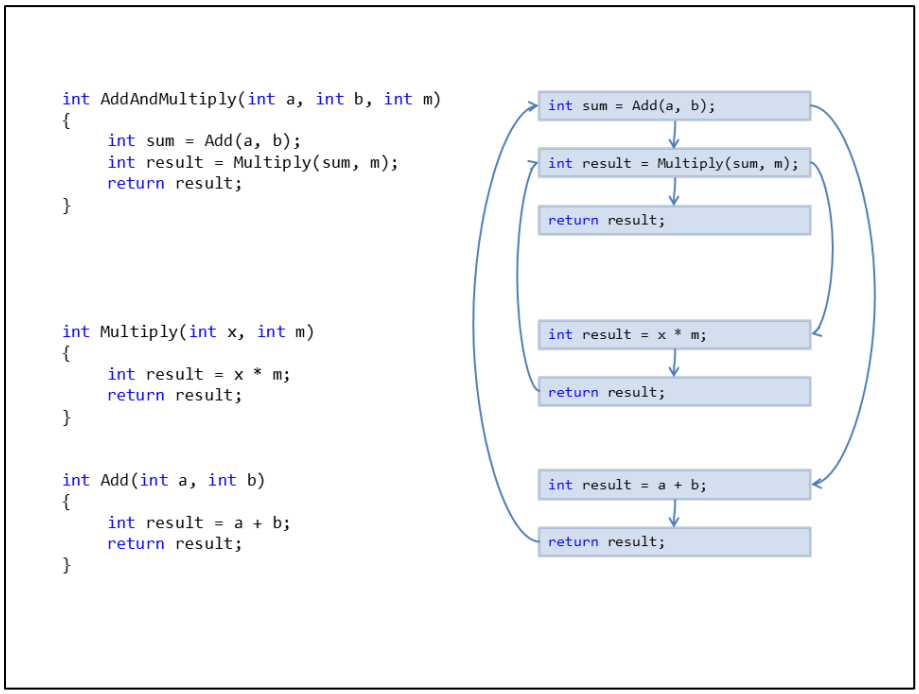

Abbildung 4: Ein Programm und sein Superkontrollflussgraph

Abbildung 4 zeigt ein Programm aus drei Methoden und den zugehörigen Superkontrollflussgraphen. Deutlich zu sehen sind die Kanten, welche aus Methodenaufrufen resultieren (rechts der Anweisungsknoten), sowie die Kanten aus Rücksprunganweisungen (links der Anweisungsknoten).

2.4.5 Zeigeranalyse

Die Zeigeranalyse soll die Frage beantworten, auf welche Objekte eine Variable zeigen kann. Dies ist wichtig für die Bestimmung von Datenabhängigkeiten. Falls zwei Variablen A und B niemals auf dieselben Objekte zeigen können, so ist ein (Schreib- oder Lese-)Zugriff auf A immer unabhängig von einem (Schreib- oder Lese-)Zugriff auf B.

Vollständig exakte Zeigeranalysen sind normalerweise sehr aufwendig und speicherhungrig, in den meisten Fällen jedoch sogar unmöglich, da die Existenz und Nicht-Existenz von Objekten oft direkt von den Eingabedaten abhängen und damit erst zur Laufzeit des Programms auswertbar sind. Daher werden üblicherweise Überapproximationen für die möglichen Zeiger und Vereinfachungen bezüglich der Umgebung gemacht, in welcher diese Annäherungen gültig sind:

- **Kontextsensitivität:** Falls eine Zeigeranalyse den Aufrufkontext einer Methode berücksichtigt, so wird sie kontextsensitiv genannt. Die Analyse hält also für verschiedene Aufrufkontexte auch verschiedene Approximationen der Variablenzeiger bereit. Je nach Implementierung kann der Aufrufkontext hierbei stärker oder schwächer unterschieden werden.
  Verschiedene Arten von Kontextsensitivität können z.B. in [Hen06] gefunden werden.
- **Flusssensitivität:** Falls die Zeigeranalyse die Stelle einer Anweisung im Kontrollflussgraphen berücksichtigt, so wird flusssensitiv genannt. Dies ist sinnvoll, da oftmals dieselben Variablen an verschiedenen Stellen eines Programms für verschiedene Zwecke verwendet werden.

Weitere Details zu Zeigeranalysen können [ASL08] entnommen werden.

3 VERWANDTE ARBEITEN

3.1 Automatisierte Parallelisierung mit Auto-Futures

Jochen Huck stellt in seiner Arbeit [Huc10] eine Methode vor, in welcher der Datentyp Future [FW76] [BH77] automatisiert in bestehenden Quellcode eingefügt und der parallelisierte Code anschließend auf Leistungszuwachs und Korrektheit geprüft wird.

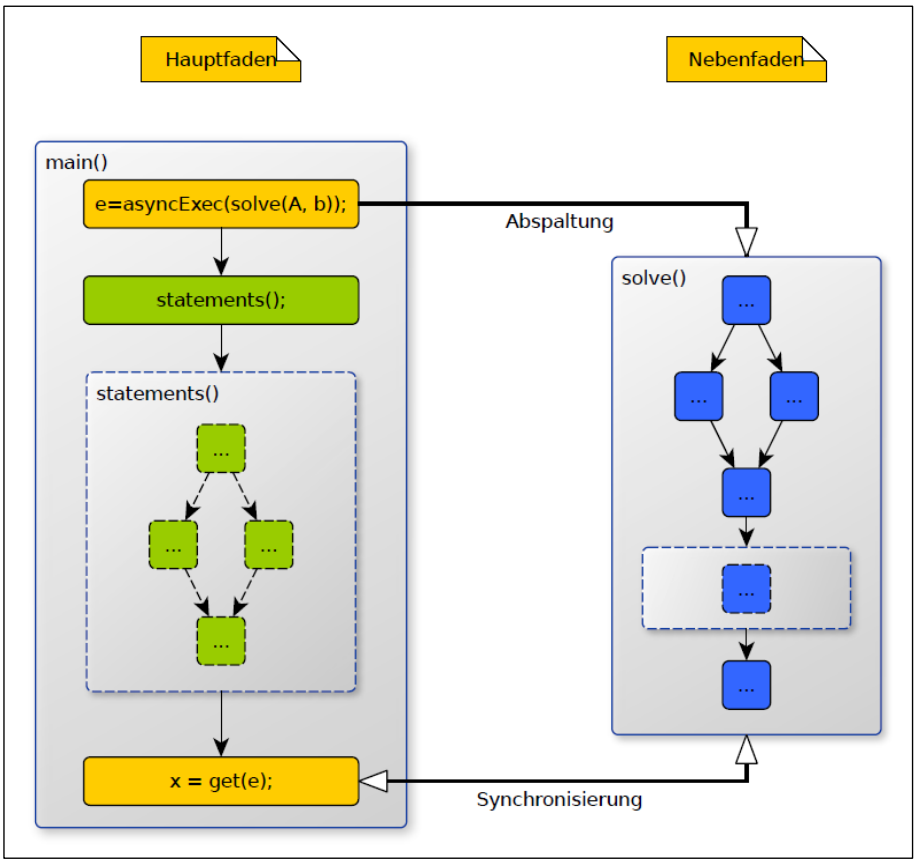

Abbildung 5: Asynchrone Methodenaufrufe mit Futures (Bild aus [Huc10])

Wie in Abbildung 5 zu sehen ist werden dazu ausgewählte (synchrone) Methodenaufrufe durch asynchrone Aufrufe ersetzt und statt des Ergebnisses ein Future zurückgegeben. Als Resultat entsteht für jede dieser Ersetzungen ein neuer Programmfaden, der parallel zum Hauptfaden abgearbeitet werden kann. Die verschiedenen Fäden werden später an Synchronisierungspunkten wieder mit dem Hauptfaden verbunden.

Für die Ermittlung der Synchronisierungspunkte wurden zwei Heuristiken benutzt:

1. Eine pessimistische Heuristik, welche durch Seiteneffektanalyse der zu parallelisierenden Methode immer eine korrekte Ausführung des Codes garantieren kann, dafür jedoch u.U. durch Wahl eines zu frühen Zeitpunktes der Synchronisierung Parallelisierungspotenzial verschenkt.
2. Eine optimistische Heuristik, welche den Synchronisationspunkt in ausgewählten Situationen auf einen späteren Zeitpunkt verschiebt, wodurch jedoch die Garantie der Korrektheit des resultierenden Codes verloren geht.
   Die optimistische Heuristik behandelt die folgenden Fälle speziell:

- simple-Invoke: Es wird angenommen, dass alleinstehende virtuelle Methodenaufrufe der Form `local.m()` ohne Rückgabewert und ohne Parameter nur lokale Daten des Objekts `local` ändern. Kann durch statische Datenabhängigkeitsanalyse gezeigt werden, dass bei zwei virtuellen Methodenaufrufen `local1.m()` und `local2.m()` die Schnittmenge der Objekte, auf welche `local1` und `local2` zeigen können, leer ist, so wird davon ausgegangen, dass die beiden Methodenaufrufe vollständig unabhängig voneinander sind.
- simple-Assign: Bei zwei Zuweisungen der Form `local = m(arg0, …, argn)` wird davon ausgegangen, dass `m` nur auf lokalen Daten arbeitet und somit keine Datenabhängigkeiten auftreten.
- Teile-und-Herrsche: Bei einer Anweisung, welche einen direkten rekursiven Aufruf besitzt, z.B. `local = m(arg1_0, …, arg1_n)` innerhalb der Methode `m(…)`, wird davon ausgegangen, dass diese Methode als Teile-und-Herrsche Algorithmus implementiert ist und keine Datenabhängigkeiten zu anderen Anweisungen auf derselben Ebene aufweist.

In diesen 3 Fällen wird davon ausgegangen, dass die entsprechenden Methodenaufrufe keine Datenabhängigkeiten zum umgebenden Code enthalten und daher nebenläufig ausgeführt werden können.

[Huc10] kommt nach Evaluation des Verfahrens an 5 Beispielen zu folgendem Schluss:

- Die Parallelisierung mittels pessimistischer Heuristik konnte in keinem Fall einen signifikanten Leistungszuwachs erzielen. In den meisten Fällen ist der parallelisierte Code sogar um ein vielfaches langsamer als der Ursprüngliche.
- Die Parallelisierung mit optimistischer Heuristik konnte in zwei der fünf Beispielen bei Erhaltung der Korrektheit auf einem Vierkernsystem eine 2,7 bzw. 3,3-fache Beschleunigung erzielen. Die Korrektheit des parallelisierten Codes konnte jedoch nicht mehr automatisch gewährleistet werden und musste in diesen Fällen manuell geprüft werden.

Der Ansatz in [Huc10] basiert ebenso wie die vorliegende Arbeit auf statischer Datenanalyse, im Gegensatz dazu beschränkt sich diese Arbeit allerdings nicht auf den asynchronen Aufruf bereits vorhandener synchroner Methodenaufrufe, sondern es wird versucht gezielt neue Methoden zu erzeugen, deren Aufrufe für die parallele Ausführung geeignet sind.

Zusätzlich soll die Evaluation nicht nur Aufschluss über die mögliche Geschwindigkeitssteigerung geben, sondern auch über den zusätzlichen Speicherbedarf der Parallelisierungen und die Auftrittshäufigkeit der Anwendungsfälle.

3.2 Pipeline-Parallelität durch Profiling-Informationen

[TF10] beschreibt einen Ansatz zur halbautomatischen Erzeugung von Pipeline-Parallelität.

Wie in Abbildung 6 dargestellt, wird der zu parallelisierende Quellcode zunächst durch den Compiler in eine interne Repräsentation umgeformt und anschließend instrumentiert. Anschließend wird der instrumentierte Code an vordefinierten Testfällen mit verschiedenen Eingabedaten ausgeführt und die Datenzugriffe werden aufgezeichnet.

Mithilfe der aufgezeichneten Datenzugriffe wiederum wird vom Abhängigkeitsanalysator ein Abhängigkeitsgraph ermittelt, welcher an den Partitionierer weitergereicht wird.

Der Partitionierer berechnet anhand des Abhängigkeitsgraphs eine Pipelinespezifikation, welche der Compiler für die Generierung des parallelen Codes verwendet.

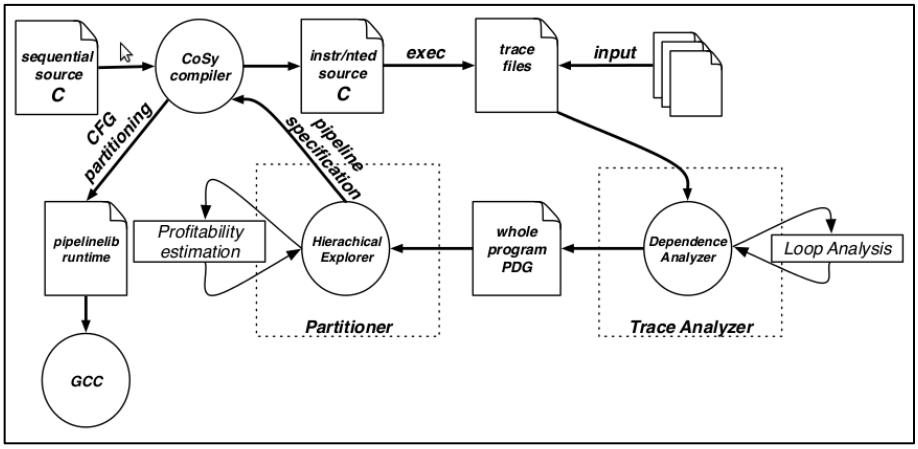

Abbildung 6: Extraktion von Pipeline-Parallelität (Bild aus [TF10])

Der Ansatz in [TF10] wird an Beispielen aus den Bereichen Multimedia (Audio, Video, Bildkompression) und Stromverarbeitenden Algorithmen (bzip2) evaluiert. Die Ergebnisse in diesen Domänen sind sowohl bezüglich der möglichen Beschleunigung, als auch bezüglich der Garantie der Korrektheit, sehr vielversprechend.

Allerdings bleibt die Frage offen, inwieweit der beschriebene Ansatz auch in anderen Domänen funktioniert, bzw. wie oft entsprechende Pipeline-Muster in anderen Anwendungen identifiziert werden können.

Im Unterschied zu [TF10] wird in dieser Arbeit ausschließlich Task-Parallelität erzeugt. Beide Ansätze könnten dadurch miteinander kombiniert werden. Die Evaluation dieser Arbeit erstreckt sich außerdem nicht nur über die Beschleunigung, sondern auch über den Bedarf an Arbeitsspeicher und die Auftrittshäufigkeit der Muster.

3.3 Parallelisierung von Teile-und-Herrsche-Algorithmen

In [RR99] wird ein Compiler vorgestellt, der Teile-und-Herrsche-Algorithmen parallelisieren kann, deren Teilprobleme jeweils auf disjunkten Regionen dynamisch zugeteilter Arrays arbeiten. Im Kern des Ansatzes steht ein kontextsensitiv, flusssensitiv und interprozedural arbeitender Algorithmus zur Zeigeranalyse.

Der Algorithmus ist für die Analyse von Zeiger-Arithmetik optimiert, kann aber nach Angabe des Autors auch für die Analyse einfacher Array-Zugriffe genutzt werden. Die Zeigeranalyse ist dabei technisch nicht auf die Analyse von Teile-und-Herrsche-Algorithmen beschränkt, sondern kann überall eingesetzt werden, wo dynamisch zugeteilte Arrays zum Einsatz kommen.

Die Implementierung der vorliegenden Arbeit verwendet für die Analyse von Array-Zugriffen eine wesentlich einfachere und damit auch ungenauere Heuristik (siehe Kapitel 5.2) und könnte von dem in [RR99] beschriebenen Ansatz profitieren.

Die Evaluation von [RR99] beschränkt sich wiederum nur auf die Angabe von Laufzeitvorteilen des parallelisierten Codes für zwei Beispielprogramme: Matrixmultiplikation und Sortieren. Eine Evaluation des Anwendungspotenzials für größere Softwareprojekte und Anwendungen ist nicht vorhanden.

4 KONZEPT

Dieses Kapitel beschreibt den konzeptuellen Aufbau der Untersuchung. Zunächst werden Ziele und Anforderungen dieser Arbeit (4.1) definiert. Es folgen je ein Kapitel über die Erstellung einer Codebasis für die Untersuchung (4.2) und die Definition von Anwendungsdomänen (4.3) für alle untersuchten Projekte. Anschließend werden die Ergebnisse der Voruntersuchung (4.4) vorgestellt, welche an einem Teil der Codebasis durchgeführt wurde. Es folgen die Definition wichtiger Konzepte (4.5) und der gefundenen Muster (4.6). Anschließend wird dargelegt, wie die Codebasis nach diesen Mustern durchsucht (4.7) und ein Teil der Fundstellen vermessen (4.8) wurde.

4.1 Ziele und Anforderungen

In diesem Kapitel werden, aufbauend auf der Arbeit von Jochen Huck [Huc10], die Schwächen des bisherigen Konzepts *Auto-Future* dargelegt und hieraus die Ziele und Anforderungen dieser Studienarbeit abgeleitet.

Die Ergebnisse aus [Huc10] stellen sich verallgemeinert wie folgt dar:

- Mit pessimistischer Heuristik konnte in den untersuchten Fällen kein Laufzeitvorteil erzielt werden. Der automatisch parallelisierte Code war korrekt, falls keine Reflektion eingesetzt wurde.
  Der Verlust der Korrektheit beim Einsatz von Reflektion kann auf die damit verbundene Ungenauigkeit der Zeigeranalyse und die dadurch resultierende unvollständige Seiteneffektanalyse zurückgeführt werden.
- Bei Einsatz einer optimistischen Heuristik wurden auf einem 4-Kern-Prozessor erhebliche Beschleunigungen erzielt, jedoch auf Kosten der Korrektheitsgarantie. Mit anschließender manueller Überprüfung konnten auf dem 4-Kern-System Beschleunigungen um bis zu Faktor 3,3 erzielt werden.

Es stellen sich folgende Fragen, die in dieser Studienarbeit genauer betrachtet werden sollen:

- Wie häufig sind Anwendungsfälle für Futures?
- Wie lassen sich Kandidaten für Auto-Futures identifizieren, die sowohl ein hohes Beschleunigungspotenzial, als auch eine hohe Wahrscheinlichkeit für anschließende Korrektheit besitzen?
- Wie lassen sich geeignete Abspaltungs- und Synchronisierungspunkte für die asynchronen Aufrufe finden?
- Wie kann der Anwendungsbereich für Auto-Futures vergrößert werden?

In der Folge ergeben sich die für diese Studienarbeit gesetzten Ziele:

- Mustersuche: Finden von Anwendungsfällen für den Einsatz von Futures
- Beschleunigungsanalyse: Untersuchung der Muster auf ihr Beschleunigungspotenzial
- Korrektheitsanalyse: Untersuchung der Muster auf Korrektheit nach der Code-Transformationen
- Vorkommensanalyse: Untersuchung der Muster auf Ihr Vorkommen in verschiedenen Anwendungsdomänen (Bildbearbeitung, Graphenalgorithmen, o.ä.)

Aus diesen Zielen wurden folgende Anforderungen abgeleitet:

- Finden der Codebasis: Quelltextdatenbanken werden nach *Open Source*-Projekten durchsucht, bei welchen hohes Parallelisierungspotenzial vermutet werden kann. Die Projekte werden zur weiteren Untersuchung gespeichert.

- Definition von Anwendungsdomänen: Eine Menge von Anwendungsdomänen wird definiert, welchen die Projekte zugeordnet werden. Dies soll später die Analyse der Anwendungsfälle in Abhängigkeit vom Einsatzzweck (der Anwendungsdomäne) erlauben.
- Voruntersuchung: Ein Teil der ausgewählten Projekte wird manuell nach Anwendungsfällen für Futures durchsucht. Danach wird analysiert, inwieweit diese Anwendungsfälle verallgemeinert werden können.
- Spezifikation und Implementierung einer Mustersuche: Die verallgemeinerten Anwendungsfälle werden formal spezifiziert und in einem Suchwerkzeug implementiert.
- Automatisches Durchsuchen der Codebasis: Ein Teil der Projekte wird mit dem entwickelten Werkzeug nach den spezifizierten Mustern durchsucht. Identifizierter Mustertyp und die Korrektheit der Fundstelle nach Parallelisierung werden anschließend manuell geprüft.
- Vermessung: Einige Fundstellen werden von Hand parallelisiert, um eine Untersuchung von Laufzeiten und Speicherverbrauch zu ermöglichen. Hierbei soll zu jedem Zeitpunkt die Korrektheit des Codes gewährleistet bleiben.

4.2 Codebasis

Zunächst wurde Quelltext von *Open Source*-Projekten gesucht, der potenziell von automatischer Parallelisierung profitieren könnte. Der Quelltext wurde aus den Datenbanken von sourceforge.net, code.google.com und codeplex.com bezogen. Die Untersuchung wurde auf C#-Code beschränkt, da der Autor in dieser Sprache über entsprechendes Fachwissen verfügt und die für die Analyse benötigten Werkzeuge existieren.

Um prozessorlastige Aktivitäten zu finden und dabei eine möglichst große Bandbreite unterschiedlicher Software abzudecken, wurden die Projekt-Datenbanken nach folgenden Schlüsselwörtern durchsucht, welche die beschriebenen Eigenschaften nahelegen:

- algorithm
- compare
- search
- find
- analyze
- convert
- solve

Folgende Projekte wurden für die Voruntersuchung in die Codebasis aufgenommen:

- *BigNumber* [BN09] enthält verschiedene Implementierungen von Datentypen und Methoden für das Rechnen mit sehr großen natürlichen und rationalen Zahlen.
- *Wordwheel Solver* [WS11] ist eine Anwendung zur Lösung sogenannter Wordwheel-Rätsel.
- *Sift String Distance and Similarity algorithm* [SSD09] enthält Algorithmen für die Ermittlung der Ähnlichkeit von Zeichenketten.

Das Projekt BigNumber wurde sowohl für die Voruntersuchung, wie auch für die automatische Mustersuche genutzt. Bei der automatischen Mustersuche wurden weitere 5 Projekte untersucht:

- *Computational Geometry* [CG11] ist eine Bibliothek mit geometrischen Algorithmen, z.B. für die Berechnung von konvexen Hüllen oder Voronoi-Diagrammen.
- *Data Structures and Algorithms* [DSA08] enthält verschiedene Datenstrukturen und Algorithmen, welche die in .NET 3.5 enthaltene Bibliothek ergänzen sollen. Darunter befinden sich unter anderem die Sortieralgorithmen Quicksort und Mergesort.
- *Evo - Framework for evolutionary computation* [Evo09] ist eine erweiterbare Bibliothek mit verschiedenen evolutionären Algorithmen.
- *GPdotNET - Genetic Programming Tool* [GP12] ist ein Werkzeug für die Modellbildung und –optimierung mithilfe genetischer Algorithmen.
- *AForge.NET* [AF12] enthält Algorithmen aus den Bereichen Bildverarbeitung, künstliche Intelligenz, Optimierung, Robotik, u.a.

Die Codebasis besteht damit aus 8 Projekten mit insgesamt ca. 131.000 Codezeilen.

4.3 Anwendungsdomänen

Die Projekte und Unterprojekte der Codebasis wurden klassifiziert und folgenden 9 Anwendungsdomänen zugeordnet.

- Framework / Klassenbibliothek
- Geometrie und Graphen
- Datenstrukturen
- Maschinelles Lernen und künstliche Intelligenz
- Allgemeine mathematische Funktionen und Datentypen
- Probabilistische Algorithmen
- Anwendungsprogramme
- Bildverarbeitung
- Optimierungsverfahren

Da die Ergebnisse der Voruntersuchung nicht direkt in die Auswertung der Auftrittshäufigkeit einfließen, wurden ausschließlich Projekte klassifiziert, welche in der automatischen Mustersuche analysiert wurden.

Unterprojekt	BigNumber	ComputationalGeometry	ComputationalGeometry.UnitTests	Dsa	Dsa.Test	Evo	gpNetLib	gpWpfTreeDrawerLib	GPdotNET	Core	Controls	Fuzzy	Genetic	Imaging	Imaging.Formats	MachineLearning	Math	Neuro	Lego	Video	Video.DirectShow	Video.VFW	Vision	Robotics.Surveyor	Video.Kinect	Video.Ximea
Projekt	BigNumber	ComputationalGeometry	ComputationalGeometry	Dsa	Dsa	Evo	GPdotNET	GPdotNET	GPdotNET	Aforge.NET	Aforge.NET	Aforge.NET	Aforge.NET	Aforge.NET	Aforge.NET	Aforge.NET	Aforge.NET	Aforge.NET	Aforge.NET	Aforge.NET	Aforge.NET	Aforge.NET	Aforge.NET	Aforge.NET	Aforge.NET	Aforge.NET
Domäne																										
Framework / Klassenbibliothek	x	x	x	x	x	x				x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Geometrie und Graphen		x	x																							
Datenstrukturen	x			x	x										x											
Maschinelles Lernen / künstliche Intelligenz							x	x	x			x				x		x								
Mathematik	x																x									
Probabilistische Algorithmen												x														
Anwendungsprogramme							x	x	x																	
Bildverarbeitung														x	x					x	x	x	x		x	x
Optimierungsverfahren						x	x	x	x				x					x								

Abbildung 7: Domänenzuordnung

Abbildung 7 zeigt die Zuordnung der Projekte bzw. ihrer Unterprojekte zu Anwendungsdomänen. Ein *x* bedeutet, dass das Projekt der Domäne zugeordnet wurde. Eine leere Zelle bedeutet dementsprechend, dass das Projekt der Domäne nicht zugeordnet wurde.

4.4 Voruntersuchung

Folgende Muster konnten in der Voruntersuchung identifiziert werden und wurden im Laufe der Studienarbeit weiter untersucht:

- **M1: Funktionsextraktion**
  Verkettete Anweisungsblöcke innerhalb von Funktionen/Methoden werden in eigene Funktionen extrahiert. Falls zwischen diesen Blöcken keine Datenabhängigkeiten bestehen, können resultierenden Methodenaufrufe mit Futures parallelisiert werden.
- **M2: Spekulative Berechnung**
  Die Berechnung eines Wertes wird vorgezogen und nebenläufig ausgeführt, obwohl zum Startzeitpunkt der (asynchronen) Berechnung noch nicht klar ist, ob das Ergebnis überhaupt benötigt wird. Für die optimale Implementierung dieses Konzepts ist eine Erweiterung des Future-Datentyps nötig, die es erlaubt asynchrone Methodenaufrufe frühzeitig abzubrechen.
- **M3: Schleifenvervielfältigung**
  Parallelisierung von Schleifen, welche in allen Iterationen (fast) nur auf Speicherstellen schreiben, die von anderen Iterationen weder gelesen noch geschrieben werden.
  Das Augenmerk wird auf Schleifen über Kollektionen gelegt.

4.5 Definitionen

In diesem Kapitel sollen die wichtigsten Begriffe definiert werden, welche für das Verständnis der anschließend beschriebenen Muster benötigt werden. Eine vollständige formale Spezifikation kann Anhang A entnommen werden.

4.5.1 Block

Im Kontext dieser Arbeit ist ein Block eine Menge von Basisblöcken bzw. von Knoten des Kontrollflussgraphs, die zusammenhängend sind und bei jedem beliebigen Programmablauf immer durch denselben Knoten betreten und durch denselben Knoten verlassen werden.

Ein Block kann daher im Gegensatz zu einem Basisblock durchaus Verzweigungen, Sprünge und Schleifen enthalten, jedoch nur solange diese sich komplett innerhalb des Blocks abspielen.

Die Definition dieser Art von Block ermöglicht es, im Rahmen der folgenden Muster, geeignete Abspaltungs- und Synchronisierungspunkte zu bestimmen.

4.5.2 Kette von Blöcken

Eine Kette von Blöcken ist eine direkte Aneinanderreihung von Blöcken. Dabei müssen die Blöcke jeweils komplett disjunkt sein und der Anfang eines Folgeblock muss direkt an das Ende seines Vorgängerblocks anschließen. Die Kette muss in jedem beliebigen Programmablauf immer in derselben Reihenfolge durchlaufen werden.

4.5.3 Kollektion und Iterator

Eine Kollektion ist eine geordnete oder ungeordnete Menge gleichartiger Datenelemente. Kollektionen können verschiedene Operationen unterstützen, z.B. das Hinzufügen von Elementen am Anfang oder Ende, das Vertauschen oder das Löschen von Elementen.

Ein Iterator dient dazu, auf jedes Element einer Kollektion zuzugreifen, ohne dass dabei die Datenstruktur, die Menge der Elemente oder die Position eines Elements innerhalb der Kollektion bekannt sein muss.

4.5.4 Hotspot

Als Hotspot wird eine Codestelle bezeichnet, an welcher viel Rechenzeit verbraucht wird, bevor die Stelle wieder verlassen wird.

Die einzelnen Muster sind unabhängig von Hotspots definiert. Die Mustersuche hingegen wurde im Rahmen dieser Arbeit so gestaltet, dass versucht wurde einzelne Hotspot voneinander zu trennen und in verschieden Fäden auszuführen. Dies verkleinert den Suchraum sehr stark und erhöht gleichzeitig das durchschnittliche Parallelisierungspotenzial der gefundenen Anwendungsfälle.

Im Rahmen dieser Arbeit wurde angenommen, dass folgende Bestandteile des Quelltexts Hotspots darstellen:

- Schleifenanweisungen,
- rekursiv alle Blockanweisungen, welche mindestens einen Hotspot enthalten und
- rekursiv alle Methodenaufrufe auf Methoden, welche mindestens einen Hotspot enthalten.

4.6 Muster

Im Folgenden werden die in der Voruntersuchung gefundenen Muster beschrieben. Eine vollständige formale Spezifikation kann Anhang A entnommen werden.

4.6.1 M1: Funktionsextraktion

Die Funktionsextraktion ist im allgemeineren Sinne die Isolisierung eines Stücks Code in eine eigene Funktion bzw. Methode. Sie ist eine Form des Code-Refactorings (Umbau von Quellcode ohne Änderung der Funktionalität mit dem Zweck die Performance oder Wartbarkeit zu Erhöhen [Mcc04]) und kann in vielen Programmieroberflächen und Sprachen bereits komplett automatisch, nach Auswahl des zu extrahierenden Codes, durchgeführt werden.

Im Kontext dieser Arbeit beschränkt sich der Begriff Funktionsextraktion nicht ausschließlich auf den Teil der Überführung von eingebetteten Code in eigene Methoden, sondern bezeichnet ebenso die Umsetzung dieser Methoden in eine asynchrone Ausführungsform, sowie die Identifikation von geeigneten Abspaltungs- und Synchronisierungspunkten.

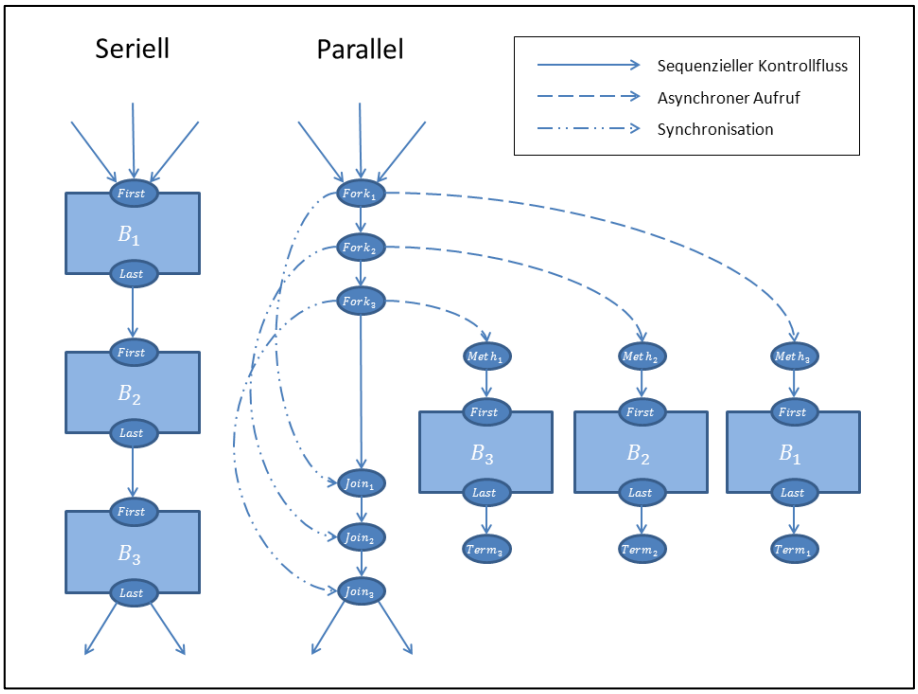

Abbildung 8: Parallelisierung durch Funktionsextraktion

Für die Umsetzung des Musters wird eine Kette aus mindestens 2 Blöcken benötigt, deren Blöcke paarweise datenunabhängig sind. Abbildung 8 illustriert die Parallelisierung dieses Musters.

Die einzelnen Blöcke werden zunächst in eigene Methoden isoliert und durch entsprechende Methodenaufrufe ersetzt. Die synchronen Methodenaufrufe werden anschließend durch asynchrone Methodenaufrufe mit Future ersetzt. Der Abspaltungspunkt aller Blöcke kann man sich daher direkt vor dem ersten Block vorstellen.

Nach den Methodenaufrufen wird für jeden Block ein Synchronisierungspunkt, in Form eines Rückgabewertzugriffs auf dem Future, eingesetzt. Dies führt dazu, dass der Programmablauf nicht fortgesetzt wird, solange nicht alle Blöcke abgearbeitet wurden.

4.6.2 M2: Spekulative Berechnung

Für das Muster „Spekulative Berechnung“ sind zwei ineinander verschachtelte Blöcke notwendig. Um die Korrektheit bei paralleler Ausführung zu garantieren, müssen die Anweisungen des inneren Blocks datenunabhängig zu allen Anweisungen sein, die sich im äußeren, aber nicht im inneren Block befinden.

Außerdem darf der innere Block auf keine Speicherstellen schreiben, auf welche Außerhalb des äußeren Blocks zugegriffen wird. Dies garantiert, dass das Programm auch nach Verlassen des äußeren Blocks korrekt durchgeführt wird, falls die Ausführung des inneren Blocks nicht benötigt wurde.

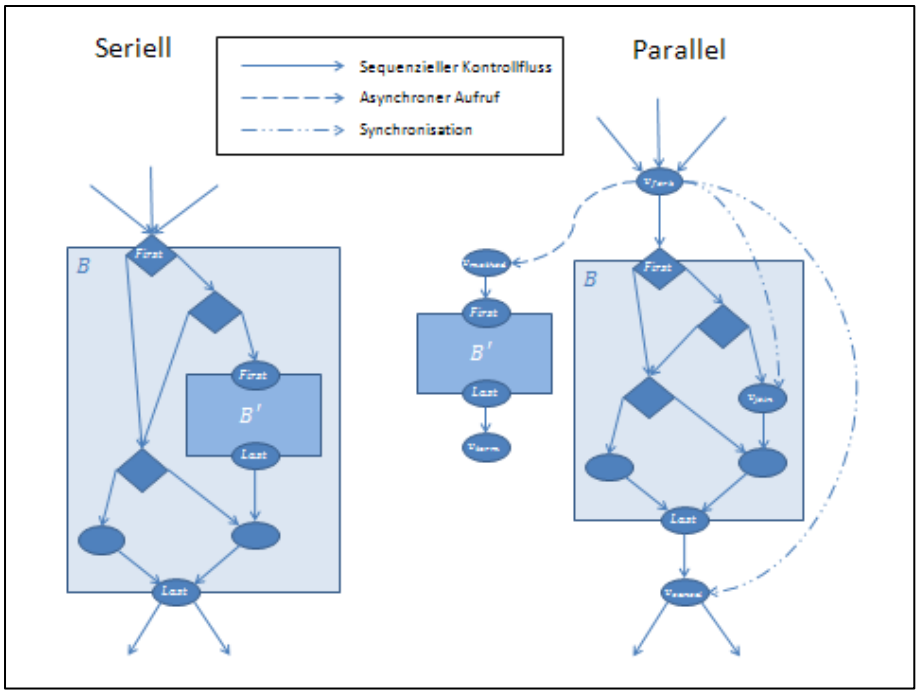

Abbildung 9: Parallelisierung durch spekulative Berechnung

Abbildung 9 stellt die Transformation dieses Musters in eine parallele Variante dar.

Wie bei der Funktionsextraktion wird zunächst der innere Block in eine eigene Methode isoliert, der asynchrone Methodenaufruf (der Abspaltungspunkt) wird jedoch vor der ersten Anweisung des äußeren Blocks eingefügt. Der innere Block wird durch einen Synchronisierungspunkt ersetzt, welcher in Form eines Rückgabewertzugriffs auf dem Future umgesetzt wird.

Da die Ausführung des inneren Blocks im sequenziellen Fall abhängig vom Kontrollfluss des Programms ist, muss am Ende des Äußeren Blocks sichergestellt sein, dass sich durch die Ausführung des inneren Blocks keine ungewollten Nebeneffekte ergeben, falls das Ergebnis nicht benötigt wurde. Dies wird durch den Synchronisierungspunkt am Ende des äußeren Blocks, sowie durch die Definition des inneren Blocks (siehe Anhang A.3) garantiert: der innere Block darf demnach auf keinerlei Speicher schreiben, welcher außerhalb des äußeren Blocks genutzt wird.

Zur Performancesteigerung kann an dieser Stelle ein eventuell noch laufender asynchroner Methodenaufruf abgebrochen werden um Ressourcen freizugeben. Alternativ kann einfach das Ende der Berechnung abgewartet werden.

4.6.3 M3: Schleifenvervielfältigung

In vielen modernen Sprachen gibt es eine spezielle Art von Schleifen, welche über Arrays bzw. Kollektionen iteriert. In C# werden diese Arten von Schleifen mit dem Schlüsselwort `foreach` eingeleitet.

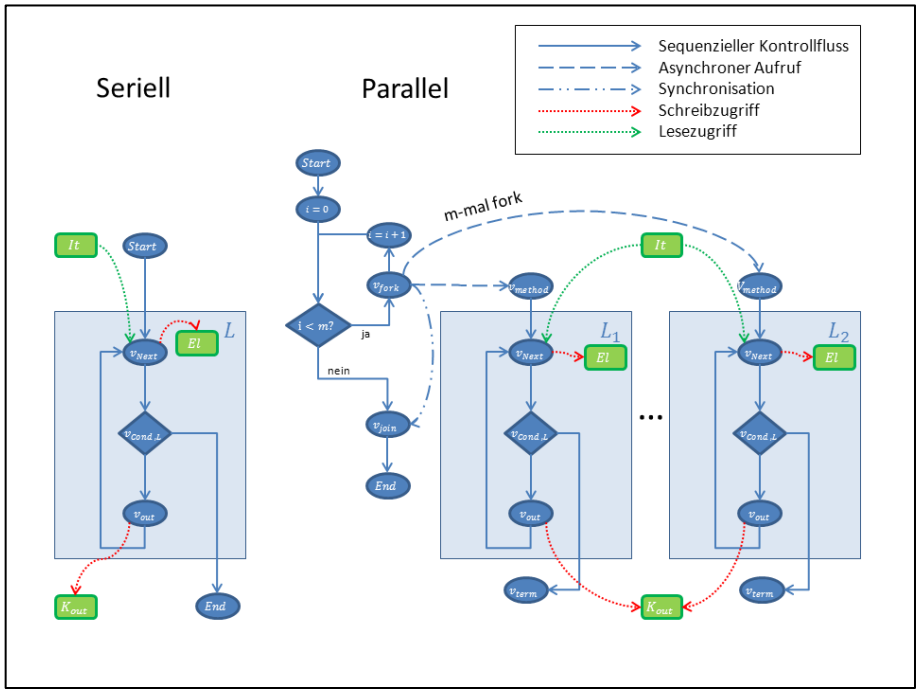

Abbildung 10: Schleifenvervielfältigung

Voraussetzung für die Anwendung des Musters Schleifenvervielfältigung ist, dass zwei verschiedene Schleifeniterationen der zu parallelisierenden Schleife niemals auf denselben Speicher schreiben. Einzige Ausnahme sind Ausgabekollektionen, zu welchen die Schleifeniterationen jedoch ausschließlich Elemente hinzufügen dürfen. Das Lesen oder Löschen von Elementen in Ausgabekollektionen ist nicht erlaubt.

Abbildung 10 verdeutlich den Parallelisierungsprozess solcher Schleifen. Die Grundidee ist es, die Schleife mehrfach gleichzeitig ausführen zu lassen, hierbei jedoch denselben Iterator zu verwenden. Falls der Iterator Thread-sicher ist, wird hierdurch garantiert, dass jedes Element der Eingabekollektion exakt einmal verarbeitet wird.

Zunächst wird die Schleife in eine eigene Methode extrahiert. Anstelle der ursprünglichen Schleife wird die extrahierte Schleifenmethode nun so oft asynchronen aufgerufen, bis die gewünschte Anzahl an Fäden erzeugt worden ist. Anschließend wird auf das Ende jedes erzeugten asynchronen Methodenaufrufs gewartet.

Der mögliche Geschwindigkeitsvorteil hat einen Haken: Die Reihenfolge von Elementen in Ausgabekollektionen ist nicht definiert. In manchen Fällen hat dies für den weiteren Programmverlauf keine Bedeutung, in vielen Fällen ist die Reihenfolge jedoch Teil des Ergebnisses. Um dieses Problem zu lösen gibt es verschiedene Möglichkeiten, welche jeweils gewisse Vor- und Nachteile haben.

```
FOREACH (e in Input) {
      IF (e ≠ C AND e ≠ E) {
            e‘ = Process(e)
            Output.Add(e‘)
      }
}
```

Abbildung 11: Pseudocode einer Schleife über einer Kollektion

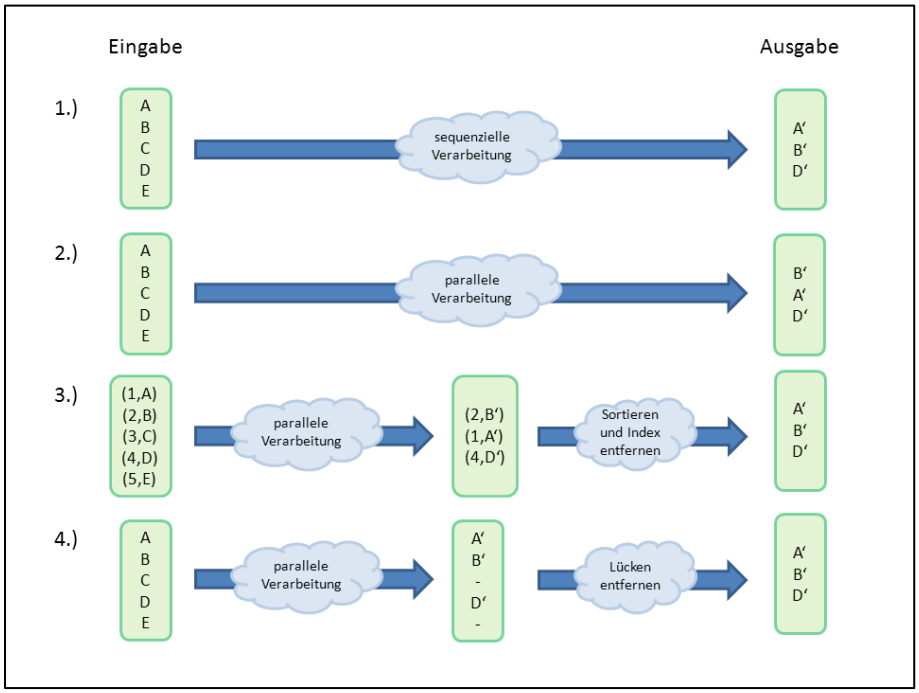

Abbildung 12: Behandlung der Ausgabereihenfolge

Abbildung 11 zeigt Pseudocode einer Schleife über einer Kollektion `Input`. Für jedes Element `e` der Eingabekollektion, außer für die Elemente `C` und `E`, wird ein Ausgabeelement `e‘` berechnet und in die Ausgabekollektion `Output` eingefügt. Abbildung 12 zeigt verschiedene Möglichkeiten, die Reihenfolge von Elementen in Ausgabekollektionen zu behandeln:

1.) Sequenzielle Verarbeitung:
Die Ausgabekollektion enthält 3 Elemente, deren Reihenfolge derjenigen der Eingabekollektion entspricht.

2.) Parallele Verarbeitung ohne Behandlung der Ausgabereihenfolge:
Die Reihenfolge der Elemente in der Ausgabekollektion ist in diesem Fall undefiniert und kann in jedem Programmlauf verschieden sein.

3.) Reihenfolgenerhaltung durch Sortieren:
Jedem Ausgabeelement wird, abhängig von der Position des Eingabeelements, ein Index zugeordnet. Nach Verarbeitung aller Eingabeelemente werden die Ausgabekollektionen nach diesem Index sortiert und der Index wird entfernt.

4.) Falls eine maximale Anzahl von Ausgabeelementen pro Iteration bekannt ist, können Ausgabeelemente in eine Kollektion eingefügt werden, so dass zwar Lücken zwischen den einzelnen Elementen bleiben, die Reihenfolge insgesamt jedoch korrekt ist. Anschließend müssen die Lücken aus der Kollektion entfernt werden.

Je nachdem, ob die tatsächliche Anzahl der Ausgabeelemente groß oder klein gegenüber der maximal möglichen Anzahl der Ausgabeelemente ist, kann Variante 3 oder 4 laufzeittechnisch günstiger sein.

4.7 Automatisches Durchsuchen der Codebasis

Das in dieser Arbeit entwickelte Werkzeug wurde genutzt um die beschriebenen Muster innerhalb der Codebasis zu identifizieren. Die einzelnen gefundenen Anwendungsfälle wurden danach manuell auf folgende Eigenschaften untersucht:

- Parallele Korrektheit:
  Kann eines der beschriebenen Muster umgesetzt werden, ohne die Korrektheit des Codes zu gefährden?
- Mustertyp:
  Wie genau kann ein Muster von anderen Mustern abgegrenzt werden?

Anschließend wurden die Ergebnisse zusammengefasst um folgende Fragen zu beantworten:

- Wie häufig treten die verschiedenen Mustertypen auf:
  - Innerhalb eines Projekts
  - Innerhalb einer Domäne
- Wie hoch ist die Erkennungsqualität des Werkzeugs?
  - Anteil an korrekt und falsch identifizierten Anwendungsfällen
  - Anteil an korrekt und falsch identifizierten Mustertypen

4.8 Vermessung

Für die Auswertung des Beschleunigungspotenzials der einzelnen Mustertypen wurden einige der gefundenen Anwendungsfälle, welche als besonders lohnenswert identifiziert wurden, von Hand parallelisiert. Diese wurden anschließend mittels eines Benchmarks auf verschiedenen Rechnersystemen vermessen.

Gemessen wurde, neben der Laufzeit der parallelisierten Methoden, auch der vom Prozess durchschnittlich allokierte Speicher. Anhand dieses Werts kann abgeschätzt werden, inwieweit sich der Speicherbedarf eines Programms durch die Parallelisierung erhöht.

Um die Korrektheit des parallelisierten Codes zu gewährleisten, wurde nach jedem Lauf die Ausgabe des parallelen Programms mit der Ausgabe des sequenziellen Programms bei gleicher Eingabe verglichen.

5 IMPLEMENTIERUNG

Im Verlauf dieser Arbeit wurde ein Werkzeug entwickelt, welches nach möglichen Anwendungsfällen der in Kapitel 4.6 beschriebenen Muster sucht. In diesem Kapitel soll näher auf die Implementierung dieses Werkzeugs eingegangen werden.

Zunächst wird das Framework *Microsoft Roslyn* vorgestellt (5.1), auf welchem die Implementierung des Werkzeugs aufbaut. Anschließend wird beschrieben, auf welche Weise das Werkzeug Datenabhängigkeiten (5.2), sowie den Kontrollfluss (5.3) analysiert. Es folgt die Beschreibung der eigentlichen Mustersuche (5.4). Abschließend wird gezeigt, wie eine Parallelisierung der Muster (5.5) umgesetzt werden könnte.

5.1 Microsoft Roslyn

Roslyn ist ein von Microsoft entwickeltes Framework, welches die Lücke zwischen Softwareentwickler und Compiler verringern soll. Es bietet eine Programmierschnittstelle für die syntaktische und semantische Analyse, sowie für die Transformation von Quellcode.

Die Mustersuche des in dieser Arbeit entwickelten Werkzeugs wurde auf Basis des *Microsoft „Roslyn" Oct 2011 CTP (Community Technology preview)* [Ros11] umgesetzt, welches zu Beginn der Arbeit bereits verfügbar war. Gegen Ende der Arbeit wurde das *June 2012 CTP* veröffentlicht, welches jedoch mangels Zeit nicht mehr zum Einsatz kam.

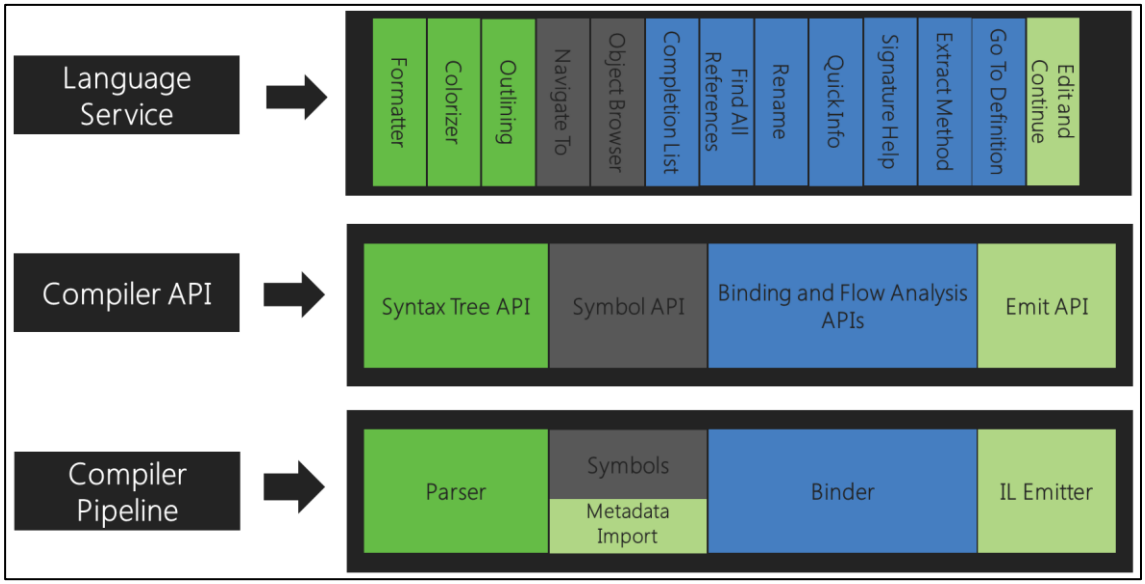

Abbildung 13: Komponentenübersicht von Roslyn (Bild aus [Ros11])

Abbildung 13 zeigt eine Übersicht der in Roslyn enthaltenen Komponenten. Diese beinhalten Funktionen für die Unterstützung von Benutzern in Entwicklungsumgebungen (Language Service), Funktionen für die Analyse und Erzeugung von Quelltext (Compiler API), sowie Funktionen für die Erzeugung von IL-Code (Compiler Pipeline).

Für diese Arbeit wurden ausschließlich Funktionen der Compiler API für die Syntax- und Semantikanalyse verwendet. Diese ermöglichen unter anderem die Erstellung eines Syntaxbaums, sowie der Zuordnung verschiedener Bezeichner (u.a. für Datentypen, Variablen, Properties, Methoden, etc.) zu sogenannten Symbolen, welche eine Unterscheidung von Bezeichnern mit gleichem Namen aber unterschiedlichem Gültigkeitsraum möglich macht.

5.2 Datenabhängigkeitsanalyse

Die Entwicklung einer möglichst genauen Datenabhängigkeitsanalyse wurde im Verlauf dieser Arbeit nicht weiter verfolgt. Stattdessen wurde eine Heuristik verwendet um abzuschätzen, ob zwischen zwei Quelltextabschnitten eine kritische Datenabhängigkeit besteht. Hierbei wurde folgendermaßen vorgegangen:

- Die Datenabhängigkeit verschiedener Codestellen wurde ausschließlich intraprozedural geprüft. Eine Interprozedurale Analyse hätte erheblichen zusätzlichen Implementierungsaufwand bedeutet und wurde daher in dieser Arbeit nicht weiter verfolgt.
- Der Vergleich der Variablenzugriffe wurde auf Basis einer hierarchischen Verkettung der von Roslyn bereitgestellten Symbole durchgeführt. Zwei Variablenzugriffe werden als gleich identifiziert, falls alle Symbole der Kette (z.B. `meinAuto.Fahrer.Name`) identisch sind.
- Indexzugriffe (`meineAutos[i,j]`) werden als identisch betrachtet, falls alle Parameter des Index identisch sind.
- Falls einer der Indizes keine Variable, sondern eine komplexer Ausdruck (z.B. `i * j + 3`) ist, so wird angenommen, dass die Indexzugriffe unterschiedliche Elemente liefern, falls der komplexe Ausdruck nicht in beiden Zugriffen exakt derselbe ist. Der Ausdruck selbst wird nicht weiter untersucht.
- Falls eine Symbolkette einen Methodenaufruf enthält (z.B. `autos.mitFarbe(rot).Laenge`), so wird angenommen, dass der Methodenaufruf eine beliebige Instanz des Rückgabetyps liefert. Eine solche Symbolkette wird grundsätzlich als ungleich zu jeder anderen, auch zu sich selbst, betrachtet um potenzielle Anwendungsfälle nicht vorzeitig auszuschließen.

5.3 Kontrollflussanalyse

Die Granularität für die Analyse von Blöcken (siehe Kapitel 4.5.1) wurde auf Anweisungen beschränkt. Eine Fortführung der Untersuchung auf kleineren Strukturen (Ausdrücke) könnte zusätzliche Anwendungsfälle aufdecken, steht derzeit aber noch aus.

Die Codeanalyse wird ohne die explizite Erzeugung eines Kontrollflussgraphen durchgeführt. Dies ist unter der Voraussetzung möglich, dass die Analyse nur intraprozedural durchgeführt wird, und dass es im untersuchten Code keine Sprünge gibt, außer solche, die zur Folgeanweisung einer Elternanweisung springen. Dies ist z.B. am Ende eines IF .. THEN .. Blocks der Fall.

Explizit bedeutet dies, dass

- der untersuchte Code keine `goto`- Anweisungen enthalten darf und
- `return`- Anweisungen als terminaler Knoten des Kontrollflussgraphen interpretiert werden (und nicht als Rücksprung in eine andere Methode).

Unter diesen Bedingungen kann davon ausgegangen werden, dass

- eine Anweisung alle nachfolgenden Anweisungen auf selber Ebene (d.h. innerhalb desselben Elternblocks `{}`) erreichen kann,
- von einer Anweisung innerhalb eines IF/ELSE-Anweisungen aus jede andere Anweisung unterhalb und auf selber Ebene wie die IF-Anweisung erreicht werden kann und

- dass eine Anweisung eine vorangegangene Anweisung nur erreichen kann, falls es eine Schleife gibt, welche beide Anweisungen enthält.

Auf eine explizite Erzeugung eines Kontrollflussgraphen wurde daher verzichtet. Stattdessen wurde direkt auf dem von Roslyn erzeugten Syntaxbaum gearbeitet.

5.4 Mustersuche

In diesem Kapitel wird die Implementierung der Suche nach den Mustern Funktionsextraktion (5.4.1), spekulative Berechnung (5.4.2) und Schleifenvervielfältigung (5.4.3) genauer beschrieben.

5.4.1 Funktionsextraktion

Die Implementierung sucht nach dem Muster Funktionsextraktion ausschließlich mit zwei Blöcken. Es kann davon ausgegangen werden, dass ein Anwendungsfall für Funktionsextraktion mit mehr als zwei Blöcken dann ebenfalls gefunden und manuell identifiziert werden kann.

Zunächst werden Methoden mit mindestens zwei Hotspots identifiziert. Diese Hotspots werden nach deren Elternblöcken gruppiert. Jeweils zwei aufeinanderfolgende Hotspots mit demselben Elternblock sind Kandidaten für die weitere Untersuchung.

Falls es möglich ist die beiden Hotspots, einschließlich aller dazwischenstehenden Anweisungen, in zwei datenunabhängige Blöcke zu unterteilen, so ist dies mit hoher Wahrscheinlichkeit ein Anwendungsfall für die Funktionsextraktion und wird für die manuelle Untersuchung markiert.

5.4.2 Spekulative Berechnung

Die Implementierung sucht nach zwei ineinander verschachtelten Blöcken, wobei mindestens ein Hotspot im inneren Block, sowie ein weiterer Hotspot ausschließlich im äußeren Block (d.h. nicht im inneren) vorhanden sein soll.

Dafür werden zunächst alle Methoden mit mindestens zwei Hotspots identifiziert.

Für jede Kombination aus zwei dieser Hotspots H_1, H_2 wird nun zunächst geprüft,

- ob einer der beiden Hotspots eine Elternanweisung (z.B. `if(){ … }`) des anderen ist (ineinander verschachtelte Anweisungen können nicht voneinander getrennt werden)
- oder ob beide Hotspots dieselbe Elternanweisung haben. (Dies wäre ein Anwendungsfall für Funktionsextraktion)

Falls eine der beiden Bedingungen erfüllt ist, so wird das Hotspot-Paar verworfen und das nächste Paar untersucht.

Falls nicht, wird der kleinste gemeinsame Block B, in welchem sich beide Hotspots befinden, berechnet. Es gibt nun zwei mögliche Anwendungsfälle für das Muster Spekulative Berechnung, die aus diesem Hotspot-Paar hervorgehen können:

- Falls $B \setminus H_1$ und H_1 datenunabhängig sind, so werden die Blöcke B, H_1 für die manuelle Untersuchung markiert.
- Falls $B \setminus H_2$ und H_2 datenunabhängig sind, so werden die Blöcke B, H_2 für die manuelle Untersuchung markiert.

5.4.3 Schleifenvervielfältigung

Für eine genaue Analyse der Datenabhängigkeit zweier Iterationen wäre eine exakte Kenntnis der Schreib- und Zugriffsmengen der einzelnen Iterationen nötig, welche nicht geliefert werden kann. Stattdessen wird eine Überapproximation verwendet: Die Schnittmenge aus der

Schreibmenge einer und der Zugriffsmenge einer anderen Iteration ist immer Untermenge der Schreibmenge aller Iterationen zusammen.

Von allen Symbolketten aus der Schreibmenge der untersuchten Schleife wird das Symbol untersucht, das am weitesten links steht. Ist es eine lokale Variable (d.h. sie liegt auf dem Stack) und ist sie innerhalb der Schleife deklariert worden, so wird davon ausgegangen, dass die Variable in zwei verschiedenen Schleifeniteration auch verschiedene Objekte enthält und an dieser Stelle somit keine Datenabhängigkeiten zwischen zwei Schleifeniterationen auftreten.

Ist das Symbol jedoch ein Feld oder eine Eigenschaft (des `this`-Objekts) oder wurde es außerhalb der Schleife deklariert, so kann mit großer Wahrscheinlichkeit davon ausgegangen werden, dass sich zwei Iterationen gegenseitig stören. Die Schleife wird in diesem Fall als Kandidat für die Schleifenvervielfältigung ausgeschlossen.

Schreibzugriffe durch Hinzufügen von Elementen zu Ausgabekollektionen sind in .NET Methodenaufrufe, welche von der benutzten Datenabhängigkeitsanalyse nicht betrachtet werden und daher auch keinen Einfluss auf die Kandidatenauswahl haben. Diese Schreibzugriffe müssen bei der manuellen Betrachtung analysiert werden.

5.5 Parallelisierung der Muster

Im Folgenden wird beschrieben, auf welche Weise die in Kapitel 4.6 beschriebenen Muster parallelisiert wurden. Die automatische Parallelisierung ist kein Teil dieser Arbeit, daher werden ausschließlich die Ideen beschrieben, nach welchen eine solche Parallelisierung von C#-Code umgesetzt werden kann.

5.5.1 Funktionsextraktion

Für die Erklärung der Codetransformation *Funktionsextraktion* (siehe Kapitel 4.6.1 und Anhang A.2) wird das Codebeispiel in Listing 1 benutzt. Es entstammt dem Source-Projekt *BigNumber* [BN09].

```
System.UInt32 top = 0;
int shiftComplement = DigitsArray.DataSizeBits - partShift;
while (source < bufLen)
{
       if (target > 0)
       {
              top = buffer[source] & mask;
              top = top << shiftComplement;
              buffer[target - 1] = buffer[target - 1] | top;
       }
       buffer[target] = buffer[source] >> partShift;
       target++;
       source++;
}

while (bufLen > 1 && buffer[bufLen - 1] == 0)
{
       bufLen--;
}

return bufLen;
```

Listing 1: Funktionsextraktion

Die beiden hellblau markierten Bereiche wurden vom entwickelten Werkzeug als unabhängige Blöcke des Musters Funktionsextraktion markiert.

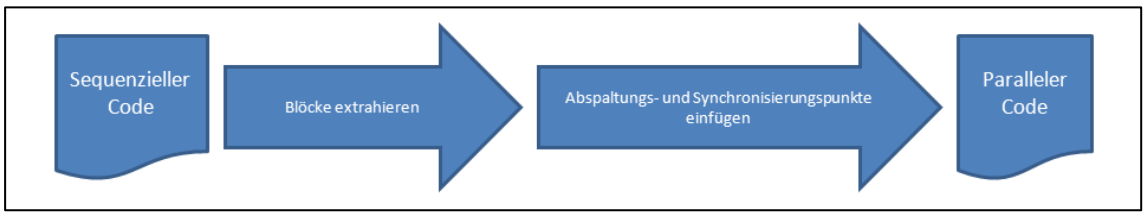

Abbildung 14: Parallelisierungsprozess Funktionsextraktion

Abbildung 14 zeigt die Stufen des Parallelisierungsprozesses beim Muster *Funktionsextraktion*. Zunächst werden die einzelnen Blöcke der gefundenen Blockkette in eigene Methoden extrahiert. Anschließend werden die resultierenden Methodenaufrufe durch asynchrone Methodenaufrufe (die Abspaltungspunkte) ersetzt und am Ende der Kette Abfragen auf den Rückgabewert der asynchronen Methodenaufrufe (die Synchronisierungspunkte) eingefügt.

Die Codetransformation wird unter Zuhilfenahme der Funktion *Extract Method* von Visual Studio durchgeführt.

1.) Im ersten Schritt wird jeder Block der Kette in Visual Studio markiert und mit „Extract Method“ extrahiert. Die entsprechenden Codestellen werden hierdurch von Visual Studio durch Methodenaufrufe in zwei verschiedenen Ausprägungen ersetzt:

- `value = Method1([ref/out] par1, [ref/out] par2, … )` oder
- `Method2([ref/out] par1, [ref/out] par2, … )`

Aus dem oben aufgeführten Codestück wird also:

```
System.UInt32 top = 0;
int shiftComplement = DigitsArray.DataSizeBits - partShift;

Method1(buffer, bufLen, partShift, ref target, ref source, mask, ref top,
shiftComplement);

bufLen = Method2(buffer, bufLen);

return bufLen;
```

Listing 2: Isolierung der Blöcke

2.) Im zweiten Schritt werden diese Methodenaufrufe nun mithilfe der Klassen `System.Threading.Tasks.Task` und `System.Threading.Tasks.Task<T>` und anonymer Methoden durch asynchrone Aufrufe ersetzt, wobei zuerst alle Tasks erstellt und anschließend alle eventuellen Rückgabewerte in der ursprünglichen Reihenfolge von den Tasks abgeholt bzw. auf die Beendigung der Tasks Beendigung gewartet wird.

Die erste Art des Methodenaufrufs wird ersetzt durch eine Anweisung der Form

```
Task<T> task1 = new Task<T>(() => Method1(par1, par2, …));
```

und Entgegennahme des Rückgabewerts in der Form

```
value = task1.Result;
```

wobei <T> dem Typ des Rückgabewerts `value` entspricht.

Die zweite Art des Methodenaufrufs wird ersetzt durch eine Anweisung der Form

```
Task task2 = new Task(() => Method2(par1, par2, …));
```

und Entgegennahme des Rückgabewerts in der Form

```
task2.Wait();
```

Hierdurch wird der Code folgendermaßen abgeändert:

```
System.UInt32 top = 0;
int shiftComplement = DigitsArray.DataSizeBits - partShift;

Task task1 = new Task(() => Method1(buffer, bufLen, partShift, ref target, ref
source, mask, ref top, shiftComplement));
task1.Start();
Task<int> task2 = new Task<int>(() => Method2(buffer, bufLen));
task2.Start();

task1.Wait();
bufLen = task2.Result;

return bufLen;
```

Listing 3: Ersetzen synchroner Methodenaufrufe durch asynchrone

Damit ist die Codetransformation abgeschlossen.

5.5.2 Spekulative Berechnung

Für die Beschreibung der *spekulativen Berechnung* (siehe auch Kapitel 4.6.2 und Anhang A.3) konnte im untersuchten Code leider kein Beispiel gefunden werden, das keinerlei zusätzliche Codetransformationen benötigt und klein genug ist, damit die Übersicht erhalten bleibt. Für die Erklärung wird daher auf ein hypothetisches Beispiel ausgewichen.

```
int result;

result = DoSomeWork1();
if (result < 100)
{
        result = DoSomeWork2();
}
```

Listing 4: Spekulative Berechnung

Die markierten Bereiche in Listing 4 stellen den äußeren, sowie den inneren Block des Musters dar. Auf die Funktionsextraktion wird in diesem Fall verzichtet. Sollte statt eines Methodenaufrufs im inneren Block ein größerer Codeabschnitt stehen, so kann dieser äquivalent zum Vorgehen in Kapitel 5.4.1 durch *Extract Method* in einen Methodenaufruf umgewandelt werden.

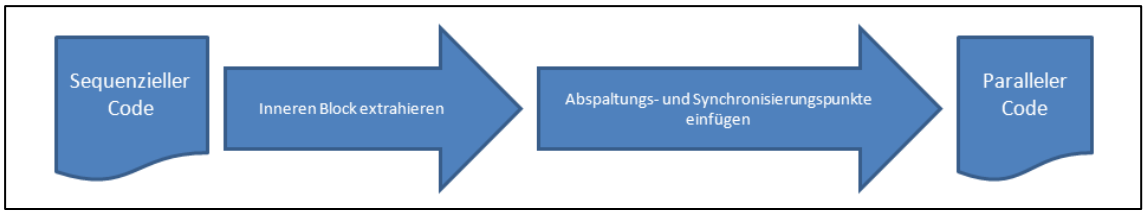

Abbildung 15: Parallelisierungsprozess spekulative Berechnung

Abbildung 15 zeigt den Parallelisierungsprozess beim Muster *spekulative Berechnung*. Zunächst wird der innere Block, analog zur Funktionsextraktion, extrahiert. Anschließend werden der asynchrone Methodenaufruf vor der ersten Anweisung des äußeren Blocks, eine Synchronisierungsanweisung anstelle des inneren Blocks, sowie eine Synchronisierungsanweisung ganz am Ende des äußeren Block eingefügt.

1.) Der innere Block wird durch *Extract Method* in eine eigene Methode überführt und gleichzeitig durch einen synchronen Methodenaufruf ersetzt. Im Beispiel muss das nicht gemacht werden, da der innere Block bereits ein Methodenaufruf ist.

2.) Im zweiten Schritt wird vor dem äußeren Block ein `Task` erzeugt und ein asynchroner Methodenaufruf auf die Methode des inneren Blocks eingefügt. Der Methodenaufruf des inneren Block wird zudem durch einen Zugriff auf die `Result`-Eigenschaft des erzeugten Tasks ersetzt. Außerdem wird nach dem äußeren Block noch ein Aufruf der `Wait()`-Methode des erzeugten Tasks eingefügt, um sicherzustellen, dass der Methodenaufruf vor dem Fortsetzen des Programms beendet ist. Im Optimalfall würde hier statt der `Wait()`-Anweisung eine Abbruchsanweisung stehen, welche den asynchronen Methodenaufruf sofort abbricht. Da diese Funktionalität jedoch in den bereitgestellten Klassen nicht zur Verfügung steht, muss auf die langsamere Variante des Wartens ausgewichen werden.

Nach diesen Transformationen sieht der Programmcode folgendermaßen aus:

```
int result;

Task<int> task = new Task<int>(() => DoSomeWork2());
Task.Start();

result = DoSomeWork1();
if (result < 100)
{
       result = task.Result;
}
task.Wait();
```

Listing 5: Parallelisiertes Muster Spekulative Berechnung

Bereits während `DoSomeWork1()` ausgeführt wird, wird auch die Ausführung von `DoSomeWork2()` angestoßen. Sollte das Ergebnis tatsächlich benötigt werden, so kann es mit verkürzter Restbearbeitungszeit abgeholt werden.

5.5.3 Schleifenvervielfältigung

Die Iteratoren der Standard .NET-Collections (`IEnumerator<T>`) enthalten leider keine atomare Methode für das gleichzeitige Fortschreiten und Lesen des nächsten Elements (siehe Anhang A.4.1). Stattdessen wird das Fortschreiten zum nächsten Element mit der Methode

```
IEnumerator<T>.MoveNext()
```

und das Auslesen desselben mit der Iterator-Eigenschaft

```
IEnumerator<T>.Current
```

durchgeführt, was eine Thread-sichere Nutzung verhindert.

Aus diesem Grund muss eine alternative Implementierung für den Iterator gewählt werden. Eine solche ist ab .NET Framework 4 mit dem Datentyp

```
System.Collections.Concurrent.ConcurrentQueue<T>
```

möglich. Der Iterator wird mit

```
new ConcurrentQueue<T>(inputCollection)
```

initialisiert. Als Iterationsmethode wird

```
ConcurrentQueue<T>.TryDequeue(out T result)
```

gewählt, welche die nötige Atomarität und damit Thread-Sicherheit besitzt.

```
private static IList<T> QuickSortInternal<T>(IList<T> list, ref Comparer<T> comparer)
{
        if (list.Count <= 1)
        {
                return list;
        }

        // lists to store relevant items
        List<T> less = new List<T>();
        List<T> greater = new List<T>();
        List<T> equal = new List<T>();

        // put the median value at index 0 of list
        list = MedianLeft(list);

        // place values in correct list {less, greater, equal}
        foreach (T item in list)
        {
                if (Compare.IsLessThan(item, list[0], comparer))
                {
                        less.Add(item);
                }
                else if (Compare.IsGreaterThan(item, list[0], comparer))
                {
                        greater.Add(item);
                }
                else
                {
                        equal.Add(item);
                }
        }

        // return list with items in the following order: less -> equal -> greater
        return Concatenate(QuickSortInternal(less, ref comparer), equal,
                QuickSortInternal(greater, ref comparer));
}
```

Listing 6: Schleifenvervielfältigung

Der Beispielcode in Listing 6 wurde aus einer Quicksort-Implementierung des Projekts *Data Structures and Algorithms* [DSA08] gewählt.

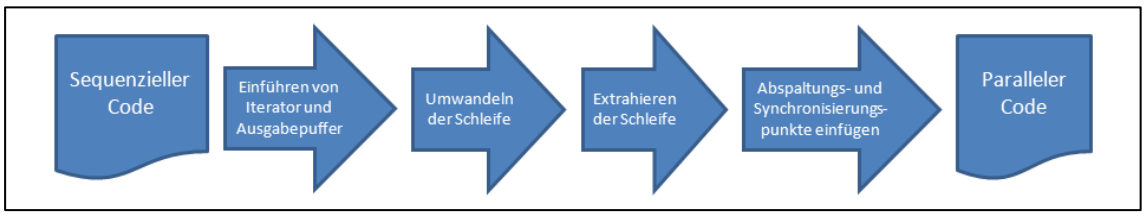

Abbildung 16: Parallelisierungsprozess Schleifenvervielfältigung

Abbildung 16 kann der Ablauf des Parallelisierungsprozesses beim Muster *Schleifenvervielfältigung* entnommen werden. Zunächst werden ein thread-sicherer Iterator für die Eingabekollektion, sowie thread-sichere Pufferkollektionen für die Ausgabe eingeführt. Anschließend wird die Schleife so transformiert, dass sie den neuen Iterator nutzt. Nun wird die Schleife in eine eigene Methode extrahiert und schließlich werden Abspaltungs- und Synchronisierungspunkte erzeugt.

1.) Im ersten Schritt wird die Eingabekollektion `list` in eine `ConcurrentQueue input` überführt. Die Schleife im Beispiel schreibt außerdem auf 3 Ausgabelisten, welche im Zuge der Parallelisierung durch Thread-sichere Varianten ersetzt werden müssen. In unserem Beispiel verwenden wir dazu den Kollektionstyp `ConcurrentBag<T>`. Nach der Ausführung der Schleife müssen diese wieder zurück in die ursprünglichen Ausgabelisten überführt werden.

2.) Im zweiten Schritt wird die Schleife in eine `while`-Schleife überführt, welche in jeder Iteration die Warteschlange `input` auf vorhandene Elemente geprüft und gleichzeitig ein Element aus der Warteschlange entnimmt. Die Schleife endet, sobald sich keine Elemente mehr in der Warteschlange befinden.

In diesem konkreten Beispiel mussten neben den Ausgabelisten zusätzlich lokale Variablen für den `ref`-Parameter `comparer`, sowie für das erste Element der Liste `list[0]` eingeführt werden. Ersteres ist wegen einer Einschränkung der Sprache oder der Plattform nötig, da `ref`-Parameter in anonymen Methoden und Lambda-Ausdrücken nicht unterstützt werden. Warum das Listenelement jedoch privatisiert werden musste, konnte nicht abschließend geklärt werden. Ein Unterlassen führte jedoch zu einer fehlerhaften Auswertung des Elementwerts und somit zu Falschergebnissen.

Das Resultat der Transformation in Schritt 1 und 2 kann Listing 7 entnommen werden.

```
private static IList<T> QuickSortInternal<T>(IList<T> list, ref Comparer<T> comparer)
{
        if (list.Count <= 1)
        {
                return list;
        }

        // lists to store relevant items
        List<T> less = new List<T>();
        List<T> greater = new List<T>();
        List<T> equal = new List<T>();

        // put the median value at index 0 of list
        list = MedianLeft(list);

        ConcurrentQueue<T> input = new ConcurrentQueue<T>(list);
        ConcurrentBag<T> less_buffer = new ConcurrentBag<T>(less);
        ConcurrentBag<T> greater_buffer = new ConcurrentBag<T>(greater);
        ConcurrentBag<T> equal_buffer = new ConcurrentBag<T>(equal);
        Comparer<T> ref_comparer = comparer;
        T list_0 = list[0];

        // place values in correct list {less, greater, equal}

        T item;
        while(input.TryDequeue(out item))
        {
                if (Compare.IsLessThan(item, list_0, ref_comparer))
                {
                        less_buffer.Add(item);
                }
                else if (Compare.IsGreaterThan(item, list_0, ref_comparer))
                {
                        greater_buffer.Add(item);
                }
                else
                {
                        equal_buffer.Add(item);
                }
        }

        less.Clear(); less.AddRange(less_buffer);
        greater.Clear(); greater.AddRange(greater_buffer);
        equal.Clear(); equal.AddRange(greater_buffer);

        // return list with items in the following order: less -> equal -> greater
        return Concatenate(QuickSortInternal(less, ref comparer), equal,
                QuickSortInternal(greater, ref comparer));
}
```

Listing 7: Umformung der Schleife

3.) Im dritten Schritt wird die Schleife inklusive der Deklaration der Schleifenvariable durch „Extract Method“ in einen Methodenaufruf extrahiert. Der resultierende Code ist in Listing 8 dargestellt.

```
private static IList<T> QuickSortInternal<T>(IList<T> list, ref Comparer<T> comparer)
{
        if (list.Count <= 1)
        {
                return list;
        }

        // lists to store relevant items
        List<T> less = new List<T>();
        List<T> greater = new List<T>();
        List<T> equal = new List<T>();

        // put the median value at index 0 of list
        list = MedianLeft(list);

        ConcurrentQueue<T> input = new ConcurrentQueue<T>(list);
        ConcurrentBag<T> less_buffer = new ConcurrentBag<T>(less);
        ConcurrentBag<T> greater_buffer = new ConcurrentBag<T>(greater);
        ConcurrentBag<T> equal_buffer = new ConcurrentBag<T>(equal);
        Comparer<T> ref_comparer = comparer;
        T list_0 = list[0];

        // place values in correct list {less, greater, equal}

        ExtractedMethod<T>(input, less_buffer, greater_buffer, equal_buffer, ref_comparer,
        list_0);

        less.Clear(); less.AddRange(less_buffer);
        greater.Clear(); greater.AddRange(greater_buffer);
        equal.Clear(); equal.AddRange(greater_buffer);

        // return list with items in the following order: less -> equal -> greater
        return Concatenate(QuickSortInternal(less, ref comparer), equal,
                QuickSortInternal(greater, ref comparer));
}

private static void ExtractedMethod<T>(ConcurrentQueue<T> input, ConcurrentBag<T> less_buffer,
ConcurrentBag<T> greater_buffer, ConcurrentBag<T> equal_buffer, Comparer<T> ref_comparer, T
list_0)
{
        T item;
        while (input.TryDequeue(out item))
        {
                if (Compare.IsLessThan(item, list_0, ref_comparer))
                {
                        less_buffer.Add(item);
                }
                else if (Compare.IsGreaterThan(item, list_0, ref_comparer))
                {
                        greater_buffer.Add(item);
                }
                else
                {
                        equal_buffer.Add(item);
                }
        }
}
```

Listing 8: Isolierung der Schleife

4.) Letztendlich wird, wie in Listing 9 zu sehen, die gewünschte Anzahl von Tasks für die extrahierte Methode in einer neuen Schleife erzeugt und in einer Liste `tasks` gespeichert. Nach der Taskerzeugung und vor der Rückführung der Ausgabelisten wird noch eine Synchronisationsanweisung eingefügt, welche auf das Ende aller in `tasks` gespeicherten Tasks wartet.

```
private static IList<T> QuickSortInternal<T>(IList<T> list, ref Comparer<T> comparer)
{
        if (list.Count <= 1)
        {
                return list;
        }

        // lists to store relevant items
        List<T> less = new List<T>();
        List<T> greater = new List<T>();
        List<T> equal = new List<T>();

        // put the median value at index 0 of list
        list = MedianLeft(list);

        ConcurrentQueue<T> input = new ConcurrentQueue<T>(list);
        ConcurrentBag<T> less_buffer = new ConcurrentBag<T>(less);
        ConcurrentBag<T> greater_buffer = new ConcurrentBag<T>(greater);
        ConcurrentBag<T> equal_buffer = new ConcurrentBag<T>(equal);
        Comparer<T> ref_comparer = comparer;
        T list_0 = list[0];

        // place values in correct list {less, greater, equal}

        // fork
        List<Task> tasks = new List<Task>();
        for (int i = 0; i < 4; i++)
        {
                Task task = new Task(() => ExtractedMethod<T>(input, less_buffer,
                greater_buffer, equal_buffer, ref_comparer, list_0));

                tasks.Add(task);
                task.Start();
        }

        // sync
        tasks.ForEach(t => t.Wait());

        less.Clear(); less.AddRange(less_buffer);
        greater.Clear(); greater.AddRange(greater_buffer);
        equal.Clear(); equal.AddRange(greater_buffer);

        // return list with items in the following order: less -> equal -> greater
        return Concatenate(QuickSortInternal(less, ref comparer), equal,
                QuickSortInternal(greater, ref comparer));
}
```

Listing 9: Die parallele Schleife

6 EVALUATION

In diesem Kapitel werden die Ergebnisse der automatischen Mustersuche genauer beleuchtet. Zunächst werden die untersuchten Projekte (6.1) kurz vorgestellt. Dann wird die Zuordnung von Projekten und gefundenen Anwendungsfällen zu Anwendungsdomänen (6.2) beschrieben. Anschließend wird genauer auf die Verbreitung der vorgestellten Muster (6.3) eingegangen. Es folgt eine Analyse von Kosten und Nutzen (6.4) der einzelnen Muster. Das Kapitel schließt mit einer Zusammenfassung der Ergebnisse (6.5).

6.1 Untersuchte Projekte

Mit dem entwickelten Werkzeug wurden 6 verschiedene Open Source-Projekte mit insgesamt ungefähr 131.000 Codezeilen untersucht. Es wurden insgesamt 133 Anwendungsfälle der oben beschriebenen Muster gefunden, davon entfallen 58 auf den Typ Funktionsextraktion, 26 auf den Typ Spekulative Berechnung und 49 auf den Typ Schleifenvervielfältigung.

Die folgenden Projekte wurden im Rahmen der Evaluation analysiert:

- *BigNumber* [BN09] enthält verschiedene Implementierungen von Datentypen und Methoden für das Rechnen mit sehr großen natürlichen und rationalen Zahlen.
- *Computational Geometry* [CG11] ist eine Bibliothek mit geometrischen Algorithmen, z.B. für die Berechnung von konvexen Hüllen oder Voronoi-Diagrammen.
- *Data Structures and Algorithms* [DSA08] enthält verschiedene Datenstrukturen und Algorithmen, welche die in .NET 3.5 enthaltene Bibliothek ergänzen sollen. Darunter befinden sich unter anderem die Sortieralgorithmen Quicksort und Mergesort.
- *Evo - Framework for evolutionary computation* [Evo09] ist eine erweiterbare Bibliothek mit verschiedenen evolutionären Algorithmen.
- *GPdotNET - Genetic Programming Tool* [GP12] ist ein Werkzeug für die Modellbildung und –optimierung mithilfe genetischer Algorithmen.
- *AForge.NET* [AF12] enthält Algorithmen aus den Bereichen Bildverarbeitung, künstliche Intelligenz, Optimierung, Robotik, u.a.

Abbildung 17 enthält Angaben über die Größe der untersuchten Projekte. Neben der Anzahl der Codezeilen (inklusive Kommentare) wird die Größe des Projekts auch in der Anzahl von Anweisungen (Statements) angegeben.

Das Messen der Codegröße anhand der Anzahl von Anweisungen bietet sich aus zwei Gründen an:

- Anweisungen sind unabhängig von vielen Faktoren, die für diese Arbeit unwichtig sind, wie z.B. Zeilenumbrüche, Kommentare, oder die Länge von Variablen-, Methoden- und Klassennamen.
- Während der Analysephase wird der untersuchte Code bereits in seine Syntaxbestandteile zerlegt. Die Anzahl an Anweisungen fällt somit ohne zusätzliche Kosten als Nebenprodukt ab.

Projekt	Unterprojekt	Zeilen	Statements
BigNumber	BigNumber	5422	2793
ComputationalGeometry	ComputationalGeometry	1100	439
ComputationalGeometry	ComputationalGeometry.UnitTests	291	107
Dsa	Dsa	4762	1362
Dsa	Dsa.Test	4219	1330
Evo	Evo	3547	1225
GPdotNET	gpNetLib	3438	1172
GPdotNET	gpWpfTreeDrawerLib	1533	547
GPdotNET	GPdotNET	5621	2839
Aforge	Core	2750	413
Aforge	Controls	4264	1190
Aforge	Fuzzy	2636	365
Aforge	Genetic	4719	937
Aforge	Imaging	46442	13457
Aforge	Imaging.Formats	1700	529
Aforge	MachineLearning	937	190
Aforge	Math	11283	2853
Aforge	Neuro	3123	572
Aforge	Lego	3113	580
Aforge	Video	2180	558
Aforge	Video.DirectShow	6490	1048
Aforge	Video.VFW	2030	354
Aforge	Vision	2953	726
Aforge	Robotics.Surveyor	2513	556
Aforge	Video.Kinect	1838	522
Aforge	Video.Ximea	2142	409

Abbildung 17: Codegröße der untersuchten Projekte

6.2 Domänenzuordnung

Die einzelnen Projekte und Unterprojekte wurden jeweils einer oder mehreren der folgenden 9 Domänen zugeordnet:

- Framework / Klassenbibliothek
- Geometrie und Graphen
- Datenstrukturen
- Maschinelles Lernen und künstliche Intelligenz
- Allgemeine mathematische Funktionen und Datentypen
- Probabilistische Algorithmen
- Anwendungsprogramme
- Bildverarbeitung
- Optimierungsverfahren

Domäne	Projekt	Unterprojekt																									
	BigNumber	BigNumber	ComputationalGeometry	ComputationalGeometry	Dsa	Dsa.Test	Evo	gpNetLib	gpWpfTreeDrawerLib	GPdotNET	Core	Controls	Fuzzy	Genetic	Imaging	Imaging.Formats	MachineLearning	Math	Neuro	Lego	Video	Video.DirectShow	Video.VFW	Vision	Robotics.Surveyor	Video.Kinect	Video.Ximea
	ComputationalGeometry	ComputationalGeometry	ComputationalGeometry.UnitTests																								
	Dsa	Dsa	Dsa.Test																								
	Evo	Evo	gpNetLib																								
	GPdotNET	GPdotNET	gpWpfTreeDrawerLib																								
	Aforge.NET	GPdotNET	Aforge.NET																								
Framework / Klassenbibliothek	x	x	x	x	x	x				x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Geometrie und Graphen		x	x																								
Datenstrukturen	x			x	x											x											
Maschinelles Lernen / künstliche Intelligenz							x	x	x				x				x		x								
Mathematik	x																	x									
Probabilistische Algorithmen													x														
Anwendungsprogramme								x	x	x																	
Bildverarbeitung															x	x					x	x	x	x		x	x
Optimierungsverfahren							x	x	x	x				x					x								

Abbildung 18: Domänen der untersuchten Projekte

Abbildung 18 zeigt die Zuordnung der untersuchten Projekte zu den genannten Domänen. Die Namen der Projekte bzw. deren Unterprojekte sind verschiedenen Spalten aufgetragen, die Domänen dementsprechend in verschiedenen Zeilen. Ein x in einer Zelle bedeutet, dass das Projekt/Unterprojekt der entsprechenden Spalte der Domäne zugeordnet wurde. Eine leere Zelle bedeutet dementsprechend, dass das Projekt/Unterprojekt der Domäne nicht zugeordnet wurde.

Die Zuordnung von potenziellen Anwendungsfällen der Muster zu Domänen wurde folgendermaßen gemacht: Befand sich der Anwendungsfall in einem Projekt bzw. Unterprojekt, welches einer bestimmten Domäne zugeordnet war, so wurde auch der Anwendungsfall dieser Domäne zugeordnet. Auf diese Weise konnte untersucht werden, ob bestimmte Mustertypen in manchen Anwendungsdomänen stärker oder schwächer vertreten sind als in anderen.

	Domänen									
Codegröße	Framework / Klassen-bibliothek	Geometrie / Graphen	Daten-strukturen	Maschinelles Lernen / künstliche Intelligenz	Mathematik	Probabilistische Algorithmen	Anwendungs-programme	Bild-verarbeitung	Optimierungs-verfahren	Gesamt
Zeilen	120454	1391	16103	17288	16705	2636	10592	65775	21981	131046
Statements	32515	546	6014	5685	5646	365	4558	17603	7292	37073

Abbildung 19: Codegröße der untersuchten Domänen

Abbildung 19 enthält die Größe jeder untersuchten Domäne jeweils in Codezeilen, sowie in Anweisungen (Statements). Wegen der Mehrfachzuordnung von Projekten zu Domänen überlappen sich die Domänen teilweise, daher entspricht die Gesamtanzahl der Codezeilen bzw. Anweisungen nicht der Summe der entsprechenden Domänenwerte.

6.3 Verbreitung der Muster

Nach der Domänenzuordnung wurden die automatisch gefundenen potenziellen Anwendungsfälle der Muster manuell auf Korrektheit einer parallelen Umsetzung geprüft. Ein Anwendungsfall wurde als korrekt klassifiziert, falls ein beliebiger der 3 Mustertypen (Funktionsextraktion, spekulative Berechnung, Schleifenvervielfältigung) an der entsprechenden Codestelle umgesetzt werden konnte, ohne die Korrektheit des Programms zu gefährden. Abbildung 20 und Abbildung 21 zeigen die Ergebnisse dieser Analyse.

	Domänen									
Anwendungsfälle	Framework / Klassen-bibliothek	Geometrie / Graphen	Daten-strukturen	Maschinelles Lernen / künstliche Intelligenz	Mathematik	Probabilistische Algorithmen	Anwendungs-programme	Bild-verarbeitung	Optimierungs-verfahren	Gesamt
Gefunden	366	**3**	102	122	86	5	107	150	185	473
- Korrekt	110	**2**	26	28	20	1	23	29	63	133
- Unkorrekt	256	**1**	76	94	66	4	84	121	122	340
Genauigkeit	30,05%	**66,67%**	25,49%	22,95%	23,26%	20,00%	21,50%	19,33%	34,05%	28,12%
Häufigkeit (Zeilen)	0,91	1,44	1,61	1,62	1,20	0,38	2,17	0,44	2,87	1,01
Häufigkeit (Statements)	3,38	3,66	4,32	4,93	3,54	2,74	5,05	1,65	8,64	3,59

Abbildung 20: Korrektheit der gefundenen Anwendungsfälle (gruppiert nach Domänen)

In Abbildung 20 wurden die untersuchten Domänen spaltenweise angeordnet. Die einzelnen Zeilen sind wie folgt zu interpretieren:

- Gefunden: Die Gesamtanzahl der gefundenen Anwendungsfälle in der jeweiligen Domäne.
- Korrekt: Die Anzahl der korrekten Anwendungsfälle. D.h. diejenigen Anwendungsfälle, die parallelisiert werden können ohne die Korrektheit des Programms zu gefährden.
- Unkorrekt: Die Anzahl der gefundenen Anwendungsfälle, die nicht korrekt sind.
- Genauigkeit: Das Verhältnis korrekter Anwendungsfälle zur Gesamtzahl
- Häufigkeit (Zeilen): Die durchschnittliche Anzahl an korrekten Anwendungsfällen, die pro 1000 durchsuchte Codezeilen gefunden wurde.
- Häufigkeit (Statements): Die durchschnittliche Anzahl an korrekten Anwendungsfällen, die pro 1000 durchsuchte Anweisungen gefunden wurde.

Leider konnte die Trefferquote (das Verhältnis der gefundenen korrekten Anwendungsfälle zur Gesamtzahl aller im Code befindlichen korrekten Anwendungsfälle) nicht ermittelt werden, da der Aufwand für die manuelle Untersuchung des gesamten Codes zu aufwendig war.

Unter den korrekten Anwendungsfällen befinden sich sowohl solche, die direkt umgesetzt werden können, als auch solche, die vor der Transformation des Musters zusätzliche Codetransformationen wie Privatisierung von Variablen, Verschieben von Anweisungen, Ersetzen von Datentypen durch Thread-sichere Varianten, o.ä. benötigen. Eine Quantifizierung dieser zusätzlichen Transformationen ist bis jetzt jedoch noch nicht geschehen und wurde daher in der Untersuchung nicht weiter betrachtet.

		Anwendungsfälle					
Projekt	**Unterprojekt**	Gefunden	Korrekt	Unkorrekt	Genauigkeit	Häufigkeit (Zeilen)	Häufigkeit (Statements)
BigNumber	BigNumber	61	13	48	21,31%	2,40	4,65
ComputationalGeometry	ComputationalGeometry	2	2	0	100,00%	1,82	4,56
ComputationalGeometry	ComputationalGeometry.UnitTests	1	0	1	0,00%	0,00	0,00
Dsa	Dsa	23	11	12	47,83%	2,31	8,08
Dsa	Dsa.Test	6	2	4	33,33%	0,47	1,50
Evo	Evo	44	29	15	65,91%	8,18	**23,67**
GPdotNET	gpNetLib	26	5	21	19,23%	1,45	**4,27**
GPdotNET	gpWpfTreeDrawerLib	28	13	15	46,43%	8,48	**23,77**
GPdotNET	GPdotNET	53	5	48	9,43%	0,89	1,76
AForge	Core	2	0	2	0,00%	0,00	0,00
AForge	Controls	1	0	1	0,00%	0,00	0,00
AForge	Fuzzy	5	1	4	20,00%	0,38	2,74
AForge	Genetic	25	8	17	32,00%	1,70	**8,54**
AForge	Imaging	119	28	91	23,53%	0,60	2,08
AForge	Imaging.Formats	12	0	12	0,00%	0,00	0,00
AForge	MachineLearning	1	1	0	100,00%	1,07	5,26
AForge	Math	25	7	18	28,00%	0,62	2,45
AForge	Neuro	9	3	6	33,33%	0,96	**5,24**
AForge	Lego	0	0	0	-	0,00	0,00
AForge	Video	0	0	0	-	0,00	0,00
AForge	Video.DirectShow	8	0	8	0,00%	0,00	0,00
AForge	Video.VFW	1	0	1	0,00%	0,00	0,00
AForge	Vision	9	1	8	11,11%	0,34	1,38
AForge	Robotics.Surveyor	11	4	7	36,36%	1,59	7,19
AForge	Video.Kinect	1	0	1	0,00%	0,00	0,00
AForge	Video.Ximea	0	0	0	-	0,00	0,00
	Gesamt	473,00	133,00	340,00	28,12%	1,01	3,59

Abbildung 21: Korrektheit der gefundenen Anwendungsfälle (gruppiert nach Projekten)

Die Angaben in Abbildung 21 entsprechen im Wesentlichen denen aus Abbildung 20, jedoch wurden Anwendungsfälle nach Projektzugehörigkeit zusammengefasst und die Projekte aus Platzgründen auf der y-Achse angeordnet. Die Bedeutungen der einzelnen Spalten ist äquivalent der Zeilen aus Abbildung 20. Die letzte Zeile enthält die zusammengefassten Werte für den gesamten untersuchten Code.

Betrachtet man in Abbildung 20 die durchschnittliche Genauigkeit der Suche, so fällt auf, dass diese sich über sämtliche Domänen hinweg in ähnlichen Größenordnungen bewegt. Einzig die Domäne Geometrie und Graphen fällt hier mit einer überdurchschnittlichen Genauigkeit von fast 67% aus dem Rahmen. Da in dieser Domäne jedoch nur sehr wenig Code untersucht wurde, welcher zudem noch aus demselben Projekt *Computational Geometry* [CG11] stammt, kann diese Abweichung nicht als statistisch relevant angesehen werden.

Bei der Häufigkeit von Treffern sticht die Domäne *Optimierungsverfahren* mit überdurchschnittlich hohen Werten hervor. In dieser Domäne wurden 3 verschiedene Projekte, nämlich *Evo* [Evo09], *GPDotNet* [GP12] und *AForge* [AF12] mit insgesamt 6 Unterprojekten untersucht. Abbildung 21 können die zu diesen Projekten (blau) gehörenden Werte entnommen werden. Auffällig ist, dass die Häufigkeit (Statements) in 4 dieser 6 Projekte überdurchschnittlich groß ist. Das legt die Vermutung nahe, dass diese Häufung nicht ausschließlich vom Programmierstil abhängig ist und die Anwendungsdomäne *Optimierungsverfahren* grundsätzlich mehr Optimierungspotenzial birgt als die anderen untersuchten Domänen.

Als nächstes wurde untersucht, inwieweit die gefundenen korrekten Anwendungsfälle auch den Mustertypen entsprechen, als welche sie vom entwickelten Werkzeug identifiziert wurden. Abbildung 22 und Abbildung 23 geben einen Überblick über die Ergebnisse dieser Untersuchung.

		Domänen									
Muster	Typisierung	Framework / Klassen-bibliothek	Geometrie / Graphen	Daten-strukturen	Maschinelles Lernen / künstliche Intelligenz	Mathematik	Probabilistische Algorithmen	Anwendungs-programme	Bild-verarbeitung	Optimierungs-verfahren	Gesamt
Funktions-extraktion	Korrekt	45	2	6	14	7	0	13	6	36	58
	Falsch-negativ	0	0	0	0	0	0	0	0	0	0
	Falsch-positiv	9	0	7	1	8	0	0	0	0	9
	Genauigkeit	83,33%	100,00%	46,15%	93,33%	46,67%	-	100,00%	100,00%	100,00%	86,57%
	TrefferQuote	100,00%	100,00%	100,00%	100,00%	100,00%	-	100,00%	100,00%	100,00%	100,00%
Spekulative Berechnung	Korrekt	17	0	1	0	1	0	0	16	0	17
	Falsch-negativ	9	0	7	1	8	0	0	0	0	9
	Falsch-positiv	0	0	0	0	0	0	0	0	0	0
	Genauigkeit	100,00%	-	100,00%	-	100,00%	-	-	100,00%	-	100,00%
	TrefferQuote	65,38%	-	12,50%	0,00%	11,11%	-	-	100,00%	-	65,38%
Schleifen-vervielfältigung	Korrekt	39	0	12	13	4	1	10	7	27	49
	Falsch-negativ	0	0	0	0	0	0	0	0	0	0
	Falsch-positiv	0	0	0	0	0	0	0	0	0	0
	Genauigkeit	100,00%	-	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%
	TrefferQuote	100,00%	-	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%

Abbildung 22: Typisierung der Anwendungsfälle (gruppiert nach Domänen)

In der Tabelle in Abbildung 22 werden für jede untersuchte Domäne und für jeden der 3 Mustertypen (Funktionsextraktion, spekulative Berechnung und Schleifenvervielfältigung) folgende Werte angegeben:

- Korrekt: Die Anzahl der Anwendungsfälle, welche sowohl automatisch, wie auch manuell als dieser Mustertyp identifiziert wurden.
- Falsch-negativ: Die Anzahl der Anwendungsfälle, die manuell als dieser, automatisch jedoch als anderer Mustertyp identifiziert wurde.
- Falsch-positiv: Die Anzahl der Anwendungsfälle, die automatisch als dieser, manuell jedoch als anderer Mustertyp identifiziert wurde.
- Genauigkeit: Der Anteil korrekt typisierter Anwendungsfälle von allen Anwendungsfällen, die automatisch als dieser Mustertyp identifiziert wurden (also korrekte und falsch-positive).
- Trefferquote: Der Anteil korrekt typisierter Anwendungsfälle von allen Anwendungsfällen, die manuell als dieser Mustertyp identifiziert wurden (also korrekt und falsch-negative).

Als Beispiel sollen die Angaben des Musters *Funktionsextraktion* in der Domäne *Datenstrukturen* näher beleuchtet werden:

- Das Muster *Funktionsextraktion* wurde insgesamt in 54 (45+9) Anwendungsfällen erkannt.
- Davon sind 45 tatsächlich vom Muster *Funktionsextraktion* (korrekt),
- weitere 9 sind von einem der anderen beiden Mustertypen (falsch-positiv).
- 0 Anwendungsfälle wurden als anderer Mustertyp identifiziert, obwohl sie eigentlich vom Typ *Funktionsextraktion* sind (falsch-negativ).
- Bei 83,33% (=45/(45+9)) der als *Funktionsextraktion* typisierten Anwendungsfälle war die Typisierung auch korrekt.
- 100% (=45/(45+0)) aller Anwendungsfälle, die tatsächlich vom Typ *Funktionsextraktion* sind, wurden auch als *Funktionsextraktion* typisiert.

		Funktionsextraktion					Spekulative Berechnung					Schleifenvervielfältigung				
Projekt	Unterprojekt	Korrekt	Falsch-negativ	Falsch-positiv	Genauig-keit	Treffer-quote	Korrekt	Falsch-negativ	Falsch-positiv	Genauig-keit	Treffer-quote	Korrekt	Falsch-negativ	Falsch-positiv	Genauig-keit	Treffer-quote
BigNumber	BigNumber	3	0	7	30,00%	100,00%	1	7	0	100,00%	12,50%	2	0	0	100,00%	100,00%
ComputationalGeometry	ComputationalGeometry	2	0	0	100,00%	100,00%	0	0	0	-	-	0	0	0	-	-
ComputationalGeometry	ComputationalGeometry.UnitTests	0	0	0	-	-	0	0	0	-	-	0	0	0	-	
Dsa	Dsa	3	0	0	100,00%	100,00%	0	0	0	-	-	8	0	0	100,00%	100,00%
Dsa	Dsa.Test	0	0	0	-	-	0	0	0	-	-	2	0	0	100,00%	100,00%
Evo	Evo	16	0	0	100,00%	100,00%	0	0	0	-	-	13	0	0	100,00%	100,00%
GPdotNET	gpNetLib	5	0	0	100,00%	100,00%	0	0	0	-	-	0	0	0	-	-
GPdotNET	gpWpfTreeDrawerLib	3	0	0	100,00%	100,00%	0	0	0	-	-	10	0	0	100,00%	100,00%
GPdotNET	GPdotNET	5	0	0	100,00%	100,00%	0	0	0	-	-	0	0	0	-	-
Aforge	Core	0	0	0	-	-	0	0	0	-	-	0	0	0	-	-
Aforge	Controls	0	0	0	-	-	0	0	0	-	-	0	0	0	-	-
Aforge	Fuzzy	0	0	0	-	-	0	0	0	-	-	1	0	0	100,00%	100,00%
Aforge	Genetic	6	0	0	100,00%	100,00%	0	0	0	-	-	2	0	0	100,00%	100,00%
Aforge	Imaging	6	0	0	100,00%	100,00%	16	0	0	100,00%	100,00%	6	0	0	100,00%	100,00%
Aforge	Imaging.Formats	0	0	0	-	-	0	0	0	-	-	0	0	0	-	-
Aforge	MachineLearning	0	0	1	0,00%	-	0	1	0	-	0,00%	0	0	0	-	-
Aforge	Math	4	0	1	80,00%	100,00%	0	1	0	-	0,00%	2	0	0	100,00%	100,00%
Aforge	Neuro	1	0	0	100,00%	100,00%	0	0	0	-	-	2	0	0	100,00%	100,00%
Aforge	Lego	0	0	0	-	-	0	0	0	-	-	0	0	0	-	-
Aforge	Video	0	0	0	-	-	0	0	0	-	-	0	0	0	-	-
Aforge	Video.DirectShow	0	0	0	-	-	0	0	0	-	-	0	0	0	-	-
Aforge	Video.VFW	0	0	0	-	-	0	0	0	-	-	0	0	0	-	-
Aforge	Vision	0	0	0	-	-	0	0	0	-	-	1	0	0	100,00%	100,00%
Aforge	Robotics.Surveyor	4	0	0	100,00%	100,00%	0	0	0	-	-	0	0	0	-	-
Aforge	Video.Kinect	0	0	0	-	-	0	0	0	-	-	0	0	0	-	-
Aforge	Video.Ximea	0	0	0	-	-	0	0	0	-	-	0	0	0	-	-
	Gesamt	58	0	9	86,57%	100,00%	17	9	0	100,00%	65,38%	49	0	0	100,00%	100,00%

Abbildung 23: Typisierung der Anwendungsfälle (gruppiert nach Projekten)

Abbildung 23 enthält dieselben Werte wie Abbildung 22, jedoch aufgegliedert nach Projekten, statt nach Domänen. Es soll daher nicht weiter auf die Bedeutung der einzelnen Zeilen und Spalten eingegangen werden. Zu beachten ist, dass x- und y-Achse aus Platzgründen vertauscht wurden.

Folgende Auffälligkeiten können sowohl in Abbildung 23, wie auch in Abbildung 22 beobachtet werden:

- Das Muster Funktionsextraktion wurde nie falsch-negativ typisiert. D.h. alle gefundenen Anwendungsfälle vom Typ Funktionsextraktion wurden automatisch auch als solche identifiziert.
- Das Muster spekulative Berechnung wurde nie falsch-positiv typisiert. D.h. wenn ein Anwendungsfall automatisch als spekulative Berechnung identifiziert wurde, so war er das auch.
- Das Muster Schleifenvervielfältigung wurde immer korrekt typisiert.

Letzteres ist nicht weiter verwunderlich, da die Suche nach Schleifenvervielfältigung erheblich anders strukturiert ist, als die der anderen beiden Mustertypen.

In Abbildung 24 und Abbildung 25 wurden die Auftrittshäufigkeiten der einzelnen Mustertypen festgehalten. Wie auch in den vorangegangenen Abbildungen wird jeweils eine Tabelle für Angaben pro Domäne (Abbildung 24), sowie eine Tabelle für Angabe pro Projekt (Abbildung 25) abgebildet.

		Domänen									
Muster		Framework / Klassen-bibliothek	Geometrie / Graphen	Daten-strukturen	Maschinelles Lernen / künstliche Intelligenz	Mathematik	Probabilistische Algorithmen	Anwendungs-programme	Bild-verarbeitung	Optimierungs-verfahren	Gesamt
Funktions-extraktion	Treffer	45	2	6	14	7	0	13	6	36	58
	Häufigkeit (Zeilen)	0,37	1,44	0,37	0,81	0,42	0,00	1,23	0,09	1,64	0,44
	Häufigkeit (Statements)	1,38	3,66	1,00	2,46	1,24	0,00	2,85	0,34	**4,94**	1,56
Spekulative Berechnung	Treffer	26	0	8	1	9	0	0	16	0	26
	Häufigkeit (Zeilen)	0,22	0,00	0,50	0,06	0,54	0,00	0,00	0,24	0,00	0,20
	Häufigkeit (Statements)	0,80	0,00	1,33	0,18	1,59	0,00	0,00	0,91	0,00	0,70
Schleifen-vervielfältigung	Treffer	39	0	12	13	4	1	10	7	27	49
	Häufigkeit (Zeilen)	0,32	0,00	0,75	0,75	0,24	0,38	0,94	0,11	1,23	0,37
	Häufigkeit (Statements)	1,20	0,00	2,00	2,29	0,71	2,74	2,19	0,40	**3,70**	1,32

Abbildung 24: Häufigkeit der Muster (gruppiert nach Domänen)

		Funktionsextraktion			Spekulative Berechnung			Schleifenvervielfältigung		
Projekt	Unterprojekt	Treffer	Häufigkeit (Zeilen)	Häufigkeit (Statements)	Treffer	Häufigkeit (Zeilen)	Häufigkeit (Statements)	Treffer	Häufigkeit (Zeilen)	Häufigkeit (Statements)
BigNumber	BigNumber	3	0,55	1,07	8	1,48	2,86	2	0,37	0,72
ComputationalGeometry	ComputationalGeometry	2	1,82	4,56	0	0,00	0,00	0	0,00	0,00
ComputationalGeometry	ComputationalGeometry.UnitTests	0	0,00	0,00	0	0,00	0,00	0	0,00	0,00
Dsa	Dsa	3	0,63	2,20	0	0,00	0,00	8	1,68	5,87
Dsa	Dsa.Test	0	0,00	0,00	0	0,00	0,00	2	0,47	1,50
Evo	Evo	16	4,51	**13,06**	0	0,00	0,00	13	3,67	**10,61**
GPdotNET	gpNetLib	5	1,45	**4,27**	0	0,00	0,00	0	0,00	0,00
GPdotNET	gpWpfTreeDrawerLib	3	1,96	**5,48**	0	0,00	0,00	10	6,52	**18,28**
GPdotNET	GPdotNET	5	0,89	1,76	0	0,00	0,00	0	0,00	0,00
Aforge	Core	0	0,00	0,00	0	0,00	0,00	0	0,00	0,00
Aforge	Controls	0	0,00	0,00	0	0,00	0,00	0	0,00	0,00
Aforge	Fuzzy	0	0,00	0,00	0	0,00	0,00	1	0,38	2,74
Aforge	Genetic	6	1,27	**6,40**	0	0,00	0,00	2	0,42	**2,13**
Aforge	Imaging	6	0,13	0,45	16	0,34	1,19	6	0,13	0,45
Aforge	Imaging.Formats	0	0,00	0,00	0	0,00	0,00	0	0,00	0,00
Aforge	MachineLearning	0	0,00	0,00	1	1,07	5,26	0	0,00	0,00
Aforge	Math	4	0,35	1,40	1	0,09	0,35	2	0,18	0,70
Aforge	Neuro	1	0,32	1,75	0	0,00	0,00	2	0,64	**3,50**
Aforge	Lego	0	0,00	0,00	0	0,00	0,00	0	0,00	0,00
Aforge	Video	0	0,00	0,00	0	0,00	0,00	0	0,00	0,00
Aforge	Video.DirectShow	0	0,00	0,00	0	0,00	0,00	0	0,00	0,00
Aforge	Video.VFW	0	0,00	0,00	0	0,00	0,00	0	0,00	0,00
Aforge	Vision	0	0,00	0,00	0	0,00	0,00	1	0,34	1,38
Aforge	Robotics.Surveyor	4	1,59	7,19	0	0,00	0,00	0	0,00	0,00
Aforge	Video.Kinect	0	0,00	0,00	0	0,00	0,00	0	0,00	0,00
Aforge	Video.Ximea	0	0,00	0,00	0	0,00	0,00	0	0,00	0,00
	Gesamt	58	0,44	1,56	26	0,20	0,70	49	0,37	1,32

Abbildung 25: Häufigkeit der Muster (gruppiert nach Projekten)

Da die grundsätzliche Tabellenstruktur bereits aus den vorangegangen Abbildungen bekannt ist, wird hier ausschließlich auf die Spalten- bzw. Zeilenbezeichnungen eingegangen:

- Treffer: Die Anzahl der korrekten Anwendungsfälle von diesem Mustertyp.
- Häufigkeit (Zeilen): Die durchschnittliche Anzahl an korrekten Anwendungsfällen von diesem Mustertyp, die pro 1000 durchsuchte Codezeilen gefunden wurde.
- Häufigkeit (Statements): Die durchschnittliche Anzahl an korrekten Anwendungsfällen von diesem Mustertyp, die pro 1000 durchsuchte Anweisungen gefunden wurde.

Besonderes Augenmerk soll hier wieder auf die Domäne Optimierungsverfahren gelenkt werden. Die Mustertypen Funktionsextraktion und Schleifenvervielfältigung treten in dieser Domäne jeweils mit überdurchschnittlich großer Häufigkeit hervor, das Muster spekulative Berechnung ist jedoch überhaupt nicht vertreten. Bei der Untersuchung der zu dieser Domäne zugeordneten Projekte fällt auf, dass die beiden Mustertypen jeweils in 4 der 6 Unterprojekte überdurchschnittlich häufig auftreten.

Grundsätzlich kann beobachtet werden, dass die Muster Funktionsextraktion und Schleifenvervielfältigung in erheblich mehr untersuchten Projekten auftraten als das Muster spekulative Berechnung. Von den 26 untersuchten Unterprojekten trat die spekulative Berechnung nur in 4 auf, davon in 2 Projekten nur jeweils 1mal und in 2 weiteren Projekten jeweils 8- und 16-mal. Wenn man davon ausgeht, dass zwei unterschiedliche (Unter-)Projekte auch von unterschiedlichen Personen entwickelt wurden, so könnte dies ein Hinweis darauf sein, dass das Muster spekulative Berechnung stark vom Programmierstil des Entwicklers abhängig ist.

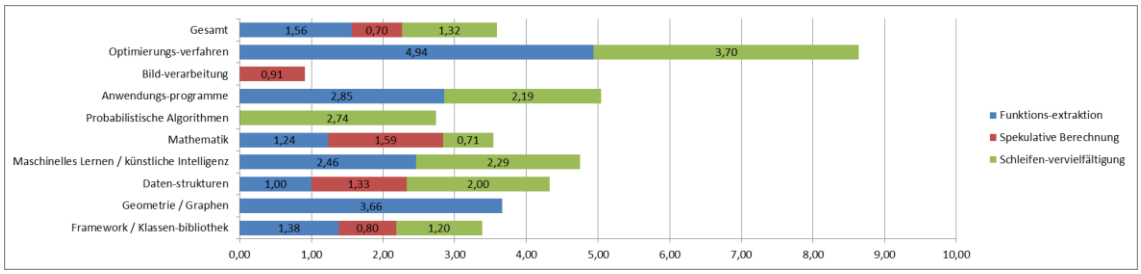

Abbildung 26: Auftrittshäufigkeit der Muster nach Domänen

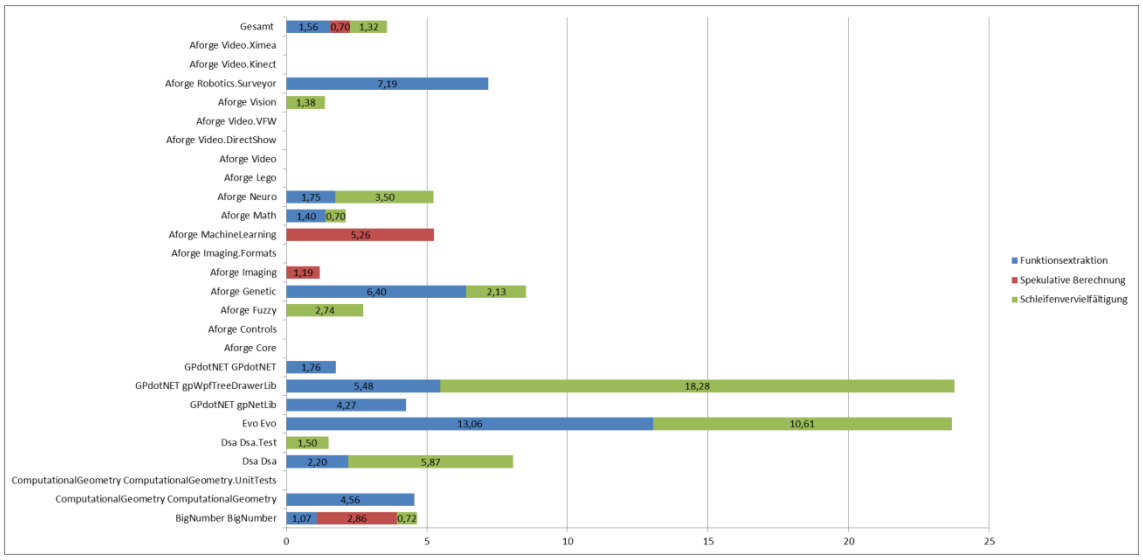

Abbildung 27: Auftrittshäufigkeit der Muster nach Projekten

In Abbildung 26 und Abbildung 27 wurden die Häufigkeitswerte der einzelnen Mustertypen nochmals graphisch zusammengefasst. Deutlich zu sehen sind die stark erhöhten Mustervorkommnisse in der Domäne *Optimierungsverfahren*, sowie in den Projekten *AForge Genetic*, *GPdotNET* und *Evo*.

6.4 Kosten und Nutzen

Für jeden Mustertyp wurde ein Anwendungsfall im untersuchten Quellcode ausgewählt, welcher bei der manuellen Analyse als besonders geeignet identifiziert wurde. Diese Anwendungsfälle wurden, wie in Kapitel 5.5 beschrieben, parallelisiert. Die parallelen, sowie die sequentiellen Versionen des Quellcodes wurden jeweils bezüglich Laufzeit und Speicherbedarf auf folgenden Systemen vermessen:

- AMD Phenom II (2x 3,3GHz)
- AMD FX-8120 (8x 3,1GHz)

Um die Korrektheit des parallelisierten Codes zu gewährleisten wurden sämtliche Ausgaben der parallelen Programme mit denen der sequenziellen Programme bei gleicher Eingabe verglichen. Die Ergebnisse waren in allen untersuchten Fällen identisch, so dass davon ausgegangen werden kann, dass durch die Parallelisierung keine Wettlaufbedingungen in die Programme eingebaut wurden.

Im Folgenden werden die vermessenen Anwendungsfälle kurz beschrieben, bevor die entsprechenden Messergebnisse näher beleuchtet werden.

6.4.1 Funktionsextraktion

Für die Untersuchung der Funktionsextraktion wurde eine Funktion des Projekts *Computational Geometry* [CG11] parallelisiert, mit welcher der geometrische Schwerpunkt eines Polygons berechnet werden kann. Die Umsetzung des Musters konnte in diesem Anwendungsfall exakt wie in 5.5.1 beschrieben bei einer Kette aus 2 Blöcken umgesetzt werden.

Die Eingabe der Funktion besteht aus einem Polygon in Form einer Liste von Punkten im 2-dimensionalen Raum. Die Ausgabe ist der geometrische Schwerpunkt in Form eines Punktes im 2-dimensionalen Raum. Die Eingabegröße wurde über die Anzahl der Punkte des Polygons angepasst.

Eingabegröße	Laufzeit (par.)	Laufzeit (seq.)	Speedup	Ø Speicher (par.)	Ø Speicher (seq.)	Mehrverbrauch
500000	0,032	0,047	1,469	43544102	40488390	7,55%
1250000	0,082	0,123	1,500	65366145	63355995	3,17%
5000000	0,333	0,509	1,529	194171600	189267600	2,59%
20000000	1,356	2,056	1,516	678399100	668538480	1,47%

Abbildung 28: Funktionsextraktion auf dem AMD Phenom II (2 Kerne)

Eingabegröße	Laufzeit (par.)	Laufzeit (seq.)	Speedup	Ø Speicher (par.)	Ø Speicher (seq.)	Mehrverbrauch
500000	0,027	0,041	1,519	46510453	37917673	22,66%
1250000	0,067	0,098	1,463	65951315	66195728	-0,37%
5000000	0,256	0,399	1,559	193953780	192958000	0,52%
20000000	0,986	1,576	1,598	720305280	676231020	6,52%

Abbildung 29: Funktionsextraktion auf dem AMD FX-8120 (8 Kerne)

Abbildung 28 und Abbildung 29 zeigen die Ergebnisse der Laufzeit- und Speicherverbrauchsmessungen. Die Spalten sind wie folgt zu interpretieren:

- Eingabegröße: Die Anzahl der Punkte des Polygons
- Laufzeit (par.): Die Laufzeit des parallelen Programms
- Laufzeit (seq.): Die Laufzeit des sequentiellen Programms
- Speedup: Die Beschleunigung des Parallelen Programms gegenüber dem sequenziellen
- Ø Speicher (par.): Der durchschnittliche Speicherbedarf des parallelen Programms
- Ø Speicher (seq.): Der durchschnittliche Speicherbedarf des sequenziellen Programms
- Mehrverbrauch: Der zusätzliche Speicherbedarf des parallelen Programms gegenüber dem sequenziellen

Auffallend ist, dass der Speedup fast unabhängig von der Eingabegröße ist. Der zusätzliche Speicherbedarf liegt außerdem in fast allen Fällen unter 10%. Warum der Verbrauch auf dem 8-Kern-System bei der kleinsten Eingabegröße so stark von den anderen Werten abweicht konnte nicht abschließend geklärt werden

Der Speedup ist zwischen den untersuchten Systemen weitgehend gleich. Dieser Umstand ist leicht einzusehen, wenn man bedenkt, dass in diesem Fall nur 2 Blöcke parallelisiert wurden. Bei längeren Block-Ketten sind auf Systemen mit mehr als 2 Kernen höhere Leistungssteigerungen zu erwarten. Die Effizienz der Funktionsextraktion kann daher für 2 Kerne mit ungefähr 0,75 angegeben werden. Für Systeme mit mehr Kernen kann keine sinnvolle Aussage gemacht werden.

6.4.2 Spekulative Berechnung

Dieser Anwendungsfall ist dem Projekt *AForge* [AF12] entnommen. Die parallelisierte Methode untersucht, ob die Form eines Polygons einem bestimmten Typ (z.B. Kreis oder Dreieck) entspricht und gibt dann diesen Typ zurück. Die Eingabe der Methode ist ein Polygon in Form einer Liste von 2-dimensionalen Punkten, die Ausgabe ist der Formtypus des Polygons. Die Methode enthält 2 Arbeitsintensive Blöcke, wobei der zweite Block im sequenziellen Fall nur dann ausgeführt wird, falls der erste Block kein Ergebnis für die Ausgabe liefern kann.

Hierdurch ergeben sich zwei unterschiedliche Fälle, die zu berücksichtigen sind:

- Fall 1: Der erste arbeitsintensive Block kann ein Ergebnis zurückliefern, das Ergebnis des zweiten Bereichs ist dann irrelevant.
- Fall 2: Der erste Block kann kein Ergebnis zurückliefern. Die Ausgabe der Methode hängt damit vom Ergebnis des zweiten Blocks ab.

Konzeptbedingt kann die spekulative Berechnung nur im zweiten Fall zu einem Laufzeitvorteil führen. Im ersten Fall kann sie bestenfalls so schnell sein wie das sequenzielle Programm.

Ein Abbruch asynchroner Methodenaufrufe vor dem Ende ihrer Berechnung ist mit den vorhandenen Mitteln leider nicht möglich gewesen, daher konnte die spekulative Berechnung nur korrekt umgesetzt werden, indem am Ende des äußeren Blocks auf das Ende aller Methodenaufrufe gewartet wird (siehe 4.6.2). Um trotzdem eine Vorstellung über die im optimalen Fall zu erwartende Laufzeit zu erhalten, wurde auch eine Messung ohne abschließendes Warten durchgeführt.

Eingabegröße	Fall 2	Warten	Laufzeit (par.)	Laufzeit (seq.)	Speedup	Ø Speicher (par.)	Ø Speicher (seq.)	Mehrverbrauch
1250000			0,070	0,073	1,043	27860736	26760530	4,11%
1250000		X	0,208	0,073	0,351	27866213	26817880	3,91%
1250000	X		0,226	0,292	1,292	27887444	27275195	2,24%
1250000	X	X	0,226	0,307	1,358	27980084	26832210	4,28%
5000000			0,286	0,283	0,990	58575258	57779540	1,38%
5000000		X	0,831	0,287	0,345	58737664	57935190	1,39%
5000000	X		0,923	1,178	1,276	58945672	58353118	1,02%
5000000	X	X	0,924	1,180	1,277	58706466	57853950	1,47%
20000000			1,135	1,070	0,943	185987480	179324200	3,72%
20000000		X	3,307	1,117	0,338	186389300	179479900	3,85%
20000000	X		3,579	4,698	1,313	185949820	184216620	0,94%
20000000	X	X	3,612	4,677	1,295	186148860	179415000	3,75%

Abbildung 30: Spekulative Berechnung auf dem auf dem AMD Phenom II (2 Kerne)

Eingabegröße	Fall 2	Warten	Laufzeit (par.)	Laufzeit (seq.)	Speedup	Ø Speicher (par.)	Ø Speicher (seq.)	Mehrverbrauch
1250000			0,086	0,087	1,012	27158630	26158420	3,82%
1250000		X	0,212	0,087	0,410	27086542	26151590	3,58%
1250000	X		0,245	0,319	1,302	27154944	26607581	2,06%
1250000	X	X	0,249	0,317	1,273	27320523	26354350	3,67%
5000000			0,354	0,337	0,952	58037658	57121450	1,60%
5000000		X	0,861	0,330	0,383	58172622	57320790	1,49%
5000000	X		0,950	1,250	1,316	57934644	57328844	1,06%
5000000	X	X	0,958	1,273	1,329	57931228	57006080	1,62%
20000000			1,325	1,307	0,986	184129540	183235900	0,49%
20000000		X	3,399	1,280	0,377	185026160	179926400	2,83%
20000000	X		3,821	4,925	1,289	184622180	183192380	0,78%
20000000	X	X	3,753	5,090	1,356	184630760	179410900	2,91%

Abbildung 31: Spekulative Berechnung auf dem AMD FX-8120 (8 Kerne)

Die Spalten in Abbildung 30 Abbildung 31 sind folgendermaßen zu interpretieren:

- Eingabegröße: Die Anzahl der Punkte im zu untersuchenden Polygon
- Fall 2: Eine leere Zelle bedeutet, dass Fall 1 eintritt. Ein X bedeutet, dass Fall 2 eintritt (siehe oben)
- Warten: X bedeutet, dass nach Ende des äußeren Blocks auf das Ende aller asynchronen Methodenaufrufe gewartet wird. Eine leere Zelle bedeutet, dass direkt mit dem anschließenden Code weitergemacht wird.

Die restlichen Spalten entsprechen in ihrer Bedeutung jenen in Abbildung 28 und Abbildung 29.

Deutlich zu erkennen ist, dass in Fall 1 (weiße und rote Zellen in Spalte Speedup) keinerlei Laufzeitvorteil erreicht wurde. Falls auf das Ende der asynchronen Methodenaufrufe gewartet wird (rote Zellen), so ist sogar eine schlechtere Laufzeit möglich.

In Fall 2 (blaue Zellen) sind auf beiden Systemen maximale Speedups von über 1,35 gemessen worden. Ähnlich wie bei der Funktionsextraktion können Systeme mit mehr als 2 Kernen in diesem Fall leider keine zusätzlichen Leistungssteigerungen erreichen, da im Rahmen dieses Anwendungsfalls nur 2 Blöcke parallelisiert wurden. Eine Erweiterung der Parallelisierung auf mehr Blöcke würde den Speedup auf solchen System vermutlich steigern. Die Effizienz auf Systemen mit 2 Kernen liegt damit bei ca. 0,67. Anhand der bisher gemachten Messungen kann für Systeme mit mehr Kernen kann leider keine sinnvolle Aussage über die Effizienz gemacht werden.

6.4.3 Schleifenvervielfältigung

Da der Quicksort-Algorithmus auf dem Prinzip *Teile und Herrsche* basiert, wurde die Schleifenvervielfältigung so implementiert, dass sie nur zum Einsatz kommt, falls die zu sortierende Liste mindestens 50 Einträge enthält. Bei weniger Einträgen wird der sequenzielle Algorithmus genutzt. Diese Maßnahme ist nötig, damit der gewonnene Laufzeitvorteil nicht bei kleinen Listen durch ineffiziente Nutzung der Ressourcen wieder aufgefressen wird.

Die Schleifenvervielfältigung ist effizienter, je größer der Aufwand innerhalb einer Schleifeniteration ist. Um diesen Faktor auch in den Messungen berücksichtigen zu können wurde folgendes Szenario gewählt:

Die Eingabe bestand aus *m* zufällig gewählten *n*-dimensionalen Vektoren, welche nach ihrer Euklidischen Norm im m-dimensionalen Raum sortiert wurde. Die Eingabegröße wurde über die zwei Parameter *m* und *n* verändert.

Dimensionen	Anzahl	Laufzeit (par.)	Laufzeit (seq.)	Speedup	Ø Speicher (par.)	Ø Speicher (seq.)	Mehrverbrauch
20	10000	0,478	0,637	1,333	36905002	26330995	40,16%
20	25000	1,287	1,767	1,373	46614390	35369666	31,79%
20	100000	5,865	8,270	1,410	105872064	86939322	21,78%
80	10000	1,675	2,526	1,508	47122910	37115714	26,96%
80	25000	4,569	6,814	1,491	78510628	62820764	24,98%
80	100000	20,483	32,382	1,581	234280820	183089480	27,96%
320	10000	6,566	9,815	1,495	124518304	84868632	46,72%
320	25000	17,358	26,304	1,515	212215540	164355284	29,12%
320	100000	76,347	121,434	1,591	704034460	584886620	20,37%

Abbildung 32: Schleifenvervielfältigung auf dem AMD Phenom II (2 Kerne)

Dimensionen	Anzahl	Laufzeit (par.)	Laufzeit (seq.)	Speedup	Ø Speicher (par.)	Ø Speicher (seq.)	Mehrverbrauch
20	10000	0,758	1,217	1,606	37789390	27266658	38,59%
20	25000	1,970	3,309	1,680	54830270	36699864	49,40%
20	100000	8,609	16,085	1,868	121818600	84583772	44,02%
80	10000	2,610	4,847	1,857	63838122	38405799	66,22%
80	25000	6,696	13,077	1,953	86525886	59739032	44,84%
80	100000	29,170	62,441	2,141	234195960	181138220	29,29%
320	10000	10,018	19,060	1,903	139360176	77489610	79,84%
320	25000	25,380	51,712	2,038	234428200	157549402	48,80%
320	100000	108,628	245,455	2,260	708893200	528898160	34,03%

Abbildung 33: Schleifenvervielfältigung auf dem AMD FX-8120 (8 Kerne)

Die Spalten in Abbildung 32 und Abbildung 33 sind folgendermaßen zu interpretieren:

- Dimensionen: Die Anzahl der Dimensionen der zu sortierenden Vektoren
- Anzahl: Die Anzahl der zu sortierenden Vektoren

Die Bedeutung aller anderen Spalten sind äquivalent der Tabellen in 6.4.1 und 6.4.2.

Bei 2 Kernen konnte bei entsprechend großen Problemen eine maximale Beschleunigung von ca. 1,6 erreicht werden. Die Effizienz liegt somit bei ca. 0,8. Der Mehrverbrauch an Arbeitsspeicher schwankt je nach Eingabegröße zwischen 20% und 47%.

Bei 8 Kernen konnte eine maximale Beschleunigung von ca. 2,26 erreicht werden. Damit liegt die Effizienz bei ungefähr 0,28. Der Mehrverbrauch an Speicher liegt auf diesem System zwischen 29% und 80%.

6.5 Fazit

Die Auswertung der Musterverbreitung zeigt, dass Die vorgestellten Muster in jedem größeren Projekt gefunden werden können. Im Durchschnitt kann das Muster Funktionsextraktion am häufigsten, mit 1,56 Anwendungsfällen pro 1000 Anweisungen, gefunden werden. Es folgen die Muster Schleifenvervielfältigung mit 1,32 Anwendungsfällen pro 1000 Anweisungen und die spekulative Berechnung mit 0,7 Anwendungsfällen pro 1000 Anweisungen.

Die einzelnen Muster können in bestimmten Anwendungsdomänen noch erheblich häufiger angetroffen werden. In der Domäne Optimierungsverfahren konnten 4,94 Anwendungsfälle der Funktionsextraktion pro 1000 Anweisungen, sowie 3,7 Anwendungsfälle der Schleifenvervielfältigung pro 1000 Anweisungen gefunden werden.

Die Ergebnisse der Laufzeitmessungen zeigen, dass mit Auto-Futures Laufzeitvorteile erreicht werden können. Auf einem 2-Kernrechner konnten mit den Mustern Funktionsextraktion und spekulative Berechnung ein Speedup von 1,6 bzw. 1,35 erreicht werden. Mit dem Muster Schleifenvervielfältigung wurde auf einem 8-Kernrechner ein Speedup von 2,26 erreicht.

Die Muster Funktionsextraktion und spekulative Berechnung benötigen hierfür nur wenig zusätzlichen Speicher. Der erhöhte Speicherbedarf der Schleifenvervielfältigung ist auf das Kopieren von Ein- und Ausgabekollektionen zurückzuführen. Mit effizienteren Datenstrukturen, könnten hier sowohl ein geringerer Speicherbedarf wie auch zusätzliche Geschwindigkeitssteigerungen erreicht werden.

Für die effiziente Nutzung der spekulativen Berechnung ist es nötig, asynchrone Methodenaufrufe schon vor dem Ende der Berechnung abbrechen zu können.

7 ZUSAMMENFASSUNG UND AUSBLICK

Auto-Futures, wie in der Arbeit von Jochen Huck [Huc10] beschrieben, bieten die Möglichkeit, synchrone Methodenaufrufe in Quelltext automatisch durch asynchrone Methodenaufrufe zu ersetzen. Die Menge der Anwendungsfälle, in welchen dieses Verfahren gleichzeitig einen Geschwindigkeitsvorteil bringt und die Korrektheit des Programms nicht verletzt, scheint jedoch äußerst begrenzt zu sein. In der vorliegenden Arbeit wurden weitere Anwendungsfälle beschrieben, um welche das Konzept Auto-Futures erweitert werden kann.

In einer Voruntersuchung wurden die Muster Funktionsextraktion, spekulative Berechnung und Schleifenvervielfältigung durch die Analyse von *Open Source*-Projekten identifiziert und anschließend formal spezifiziert.

Im weiteren Verlauf wurde ein Werkzeug für die Identifikation relevanter Stellen im Quellcode entwickelt. Die Qualität der implementierten Mustersuche wurde wiederum manuell anhand der Analyse von weiteren *Open Source*-Projekten ermittelt.

Mithilfe des entwickelten Werkzeugs wurde eine Codebasis, bestehend aus 8 Projekten mit insgesamt ca. 131.000 Codezeilen, nach den spezifizierten Mustern durchsucht und deren Auftrittshäufigkeit analysiert. Die ermittelten Ergebnisse zeigen, dass vor allem die Muster Funktionsextraktion und Schleifenvervielfältigung in den meisten größeren Projekten angetroffen werden können.

Die Codetransformationen der Muster wurden an ausgewählten Stellen von Hand durchgeführt und in Bezug auf Laufzeit, sowie Speicherverbrauch auf einem 2-Kernsystem und einem 8-Kernsystem vermessen. Die Ergebnisse liefern eine Vorstellung über die Leistungsfähigkeit der Muster, können jedoch noch keine Aussage über die zu erwartenden Ergebnisse bei der vollständigen Parallelisierung eines kompletten Projekts liefern.

Die Parallelisierung eines ganzen Projekts anhand der gefundenen Anwendungsfälle und anschließende Laufzeituntersuchung war für diese Studienarbeit vorgesehen, konnte jedoch aus Zeitgründen nicht mehr umgesetzt werden.

Die Orientierung der Mustersuche an Hotspots ist auf die Tatsache zurückzuführen, dass eine vollständige Analyse jeder Anweisung des Quelltexts sehr zeitaufwändig ist. Selbst wenn hierbei parallelisierbare Stellen gefunden werden, so ist nicht sofort abschätzbar ob diese auch wirklich messbares Beschleunigungspotenzial bergen. Das initiale Ausspähen von Hotspots ermöglicht es, die Suche nach Mustern zielgerichtet an diejenigen Stellen zu führen, welche bereits hohes Beschleunigungspotenzial enthalten. Bisher wurden allerdings ausschließlich Hotspots untersucht, welche auf Schleifen zurückzuführen sind. Andere Identifikatoren könnten sicherlich ebenso auf arbeitsintensive Stellen schließen. Beispielsweise werden bisher keinerlei Aufrufrekursionen oder lange sequenzielle Methoden betrachtet. Dynamische Codeanalyseverfahren könnten ebenso gut, oder evtl. sogar noch besser, auf weitere Hotspots schließen.

Datenabhängigkeiten werden bisher ausschließlich intraprozedural analysiert. Der Symbolketten-basierte Ansatz scheint zwar zu funktionieren, jedoch ist bisher unklar welche impliziten Beschränkungen dieser Ansatz hat und wie er auf interprozedurale Analyse ausgeweitet werden kann. Insbesondere könnten andere Heuristiken für die Datenabhängigkeitsanalyse eine starke Wirkung auf die Qualität der Ergebnisse haben.

Im Verlauf der Arbeit ist außerdem noch ein weiterer Typ von Schleifenparallelisierung für herkömmliche Schleifen (ohne Kollektionen) aufgetaucht, welcher allerdings bis zum Abgabezeitpunkt nicht mehr spezifiziert werden konnte. Insoweit scheint der Vorrat an ausnutzbaren Mustern noch nicht ausgeschöpft zu sein.

ANHÄNGE UND VERZEICHNISSE

A. Musterspezifikation

A.1. Grundlegende Definitionen

In diesem Kapitel werden Grundlegende Begriffe definiert, welche für alle folgenden Muster von Belang sind.

Alle Definitionen gehen als Basis von einem vollständig sequenziellen Programm aus, welches durch die weiter unten Beschriebenen Programmtransformationen in ein paralleles Programm umgewandelt wird.

A.1.1. Programmablaufgraph

Der hier beschriebene Programmablaufgraph ist eine Erweiterung des Kontrollflussgraphs und ähnelt dem Superkontrollflussgraph (siehe Kapitel 2.4.4). Er enthält für jede Methode eines Programms alle Knoten und Kanten des zugehörigen Kontrollflussgraphen. Zusätzlich enthält er für jede Methode einen speziellen Methodenknoten, welcher den Anfang der Methode definiert, sowie für jeden Methodenaufruf eine spezielle Aufrufkante von der Aufrufstelle zum zugehörigen Methodenknoten. Die Rücksprungkanten des Superkontrollflussgraphen werden nicht benötigt.

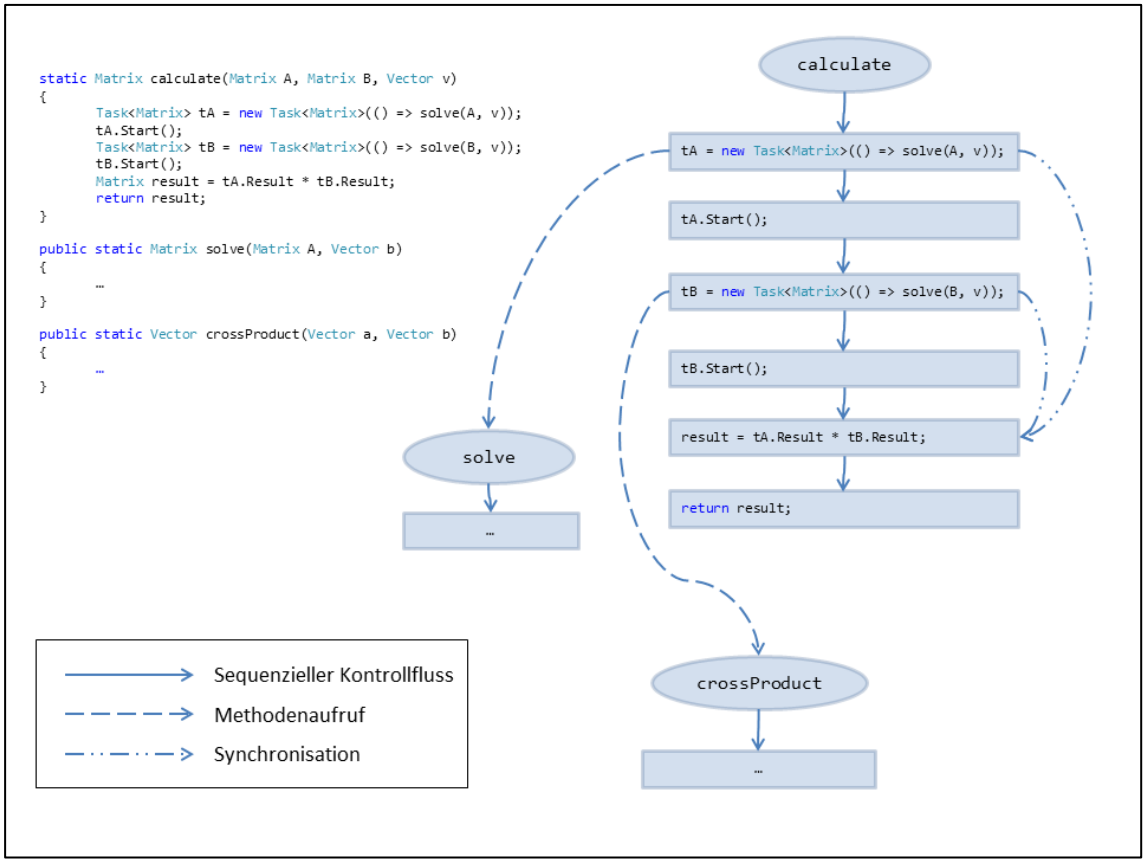

Abbildung 34: Ein Programm und sein Programmablaufgraph

Für die Darstellung von Parallelität in Form von asynchronen Methodenaufrufen wird außerdem eine Synchronisierungskante von der Aufrufstelle zur zugehörigen Synchronisierungsstelle eingeführt. Abbildung 34 illustriert einen Programmablaufgraphen anhand eines Codebeispiels in C#.

Sei $G_p\langle V, E, u\rangle$ der Programmablaufgraph eines Programms p mit Knotenmenge V, Kantenmenge $E \subseteq V \times V$ und Startknoten $u \in V$.

Definition 1 Eingehende und ausgehende Kanten

Sei $v \in V$ ein Knoten.

- Die Kantenmenge $E_{\text{in}}(v) = \{(v', v) \mid v' \in V \wedge (v', v) \in E\}$ heißt **eingehende Kanten von** v.
- Die Kantenmenge $E_{out}(v) = \{(v, v') \mid v' \in V \wedge (v, v') \in E\}$ heißt **ausgehende Kanten von** v.

Definition 2 Knoten- und Kantentypen

- V enthalte die Teilmengen **Methodenknoten** V_{method}, **Anweisungsknoten** V_{instr}, **Terminierungsknoten** V_{term}, **Verzweigungsknoten** V_{branch}, **synchrone Aufrufknoten** V_{call}, **asynchrone Aufrufknoten** V_{fork}, **Synchronisierungsknoten** V_{join} und **Abbruchknoten** V_{cancel}.
  $V_{method}, V_{instr}, V_{term}, V_{branch}, V_{call}, V_{fork}, V_{join}$ und V_{cancel} seien paarweise disjunkt.
- E enthalte die Teilmengen **bedingungslose Folgekanten** E_{succ}, **Verzweigungskanten** E_{branch}, **synchrone Aufrufkanten** E_{call}, **asynchrone Aufrufkanten** E_{fork}, **Synchronisationskanten** E_{join} und **Abbruchkanten** E_{cancel}.
  $E_{succ}, E_{branch}, E_{call}, E_{fork}, E_{join}$ und E_{cancel} seien paarweise disjunkt.

Für alle Knoten $v \in V$ gelte:

- Falls $v \in V_{method}$, dann $E_{\text{in}}(v) \subseteq (E_{call} \cup E_{fork})$, $E_{out}(v) \subseteq E_{succ}$ und $\#E_{out}(v) = 1$.
- Falls $v \in V_{instr}$, dann $E_{\text{in}}(v) \subseteq (E_{succ} \cup E_{branch})$, $\#E_{\text{in}}(v) \geq 1$, $E_{out}(v) \subseteq E_{succ}$ und $\#E_{out}(v) = 1$.
- Falls $v \in V_{term}$, dann $E_{\text{in}}(v) \subseteq (E_{succ} \cup E_{branch})$, $\#E_{\text{in}}(v) \geq 1$ und $E_{out}(v) = \emptyset$.
- Falls $v \in V_{branch}$, dann $E_{\text{in}}(v) \subseteq (E_{succ} \cup E_{branch})$, $\#E_{\text{in}}(v) \geq 1$, $E_{out}(v) \subseteq E_{branch}$ und $\#E_{out}(v) \geq 2$.
- Falls $v \in V_{call}$, dann $E_{\text{in}}(v) \subseteq (E_{succ} \cup E_{branch})$, $\#E_{\text{in}}(v) \geq 1$, $E_{out}(v) \subseteq (E_{succ} \cup E_{call})$, $\#(E_{out}(v) \cap E_{succ}) = 1$ und $\#(E_{out}(v) \cap E_{call}) = 1$.
- Falls $v \in V_{fork}$, dann $E_{\text{in}}(v) \subseteq (E_{succ} \cup E_{branch})$, $\#E_{\text{in}}(v) \geq 1$, $E_{out}(v) \subseteq (E_{succ} \cup E_{fork} \cup E_{join} \cup E_{cancel})$, $\#(E_{out}(v) \cap E_{succ}) = 1$ und $\#(E_{out}(v) \cap E_{fork}) = 1$.
- Falls $v \in V_{join}$, dann $E_{\text{in}}(v) \subseteq (E_{succ} \cup E_{branch} \cup E_{join})$, $\#(E_{\text{in}}(v) \cap (E_{succ} \cup E_{branch})) \geq 1$, $\#(E_{\text{in}}(v) \cap E_{join}) = 1$, $E_{out}(v) \subseteq E_{succ}$ und $\#E_{out}(v) = 1$.
- Falls $v \in V_{cancel}$, dann $E_{\text{in}}(v) \subseteq (E_{succ} \cup E_{branch} \cup E_{cancel})$, $\#(E_{\text{in}}(v) \cap (E_{succ} \cup E_{branch})) \geq 1$, $\#(E_{\text{in}}(v) \cap E_{cancel}) = 1$, $E_{out}(v) \subseteq E_{succ}$ und $\#E_{out}(v) = 1$.

Für alle Kanten $e \in E$ mit $e = (v, v')$ und $v, v' \in V$ gelte:

- Falls $e \in E_{succ}$, dann $v \in (V_{method} \cup V_{instr} \cup V_{call} \cup V_{fork} \cup V_{join} \cup V_{cancel})$ und $v' \in (V_{instr} \cup V_{term} \cup V_{branch} \cup V_{call} \cup V_{fork} \cup V_{join} \cup V_{cancel})$.
- Falls $e \in E_{branch}$, dann $v \in V_{branch}$ und $v' \in (V_{instr} \cup V_{term} \cup V_{branch} \cup V_{call} \cup V_{fork} \cup V_{join} \cup V_{cancel})$.
- Falls $e \in E_{call}$, dann $v \in V_{call}$ und $v' \in V_{method}$.
- Falls $e \in E_{fork}$, dann $v \in V_{fork}$ und $v' \in V_{method}$.
- Falls $e \in E_{join}$, dann $v \in V_{fork}$ und $v' \in V_{join}$.
- Falls $e \in E_{cancel}$, dann $v \in V_{fork}$ und $v' \in V_{cancel}$.

Für vollständig sequenzielle Programme gelte:

- $E_{fork} \cup E_{join} \cup E_{cancel} = \emptyset$ und $V_{fork} \cup V_{join} \cup V_{cancel} = \emptyset$.

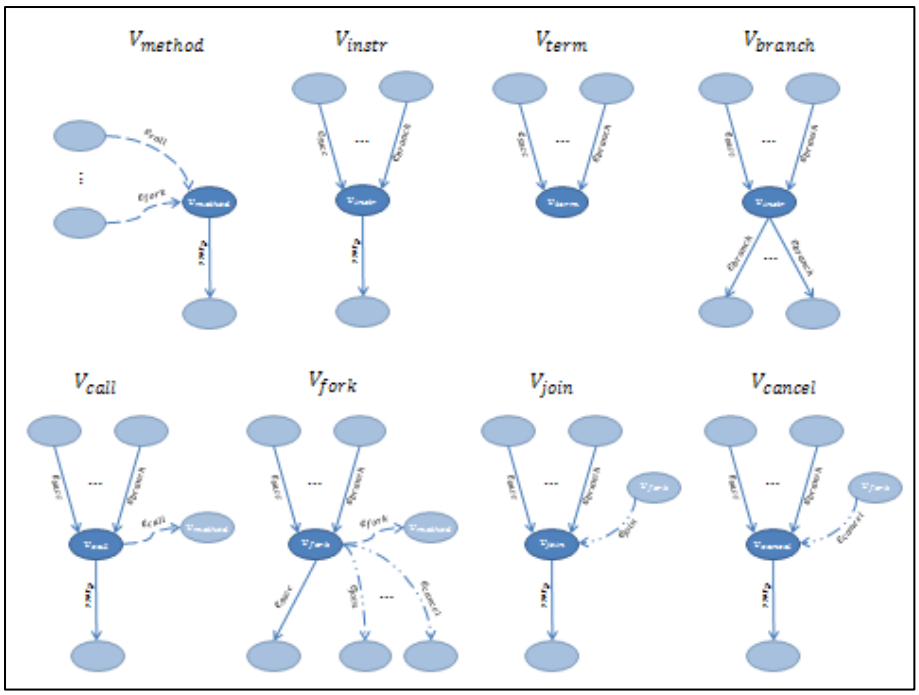

Abbildung 35: Knoten- und Kantentypen

Definition 3 Intraprozedurale Kanten, Vorgänger, Nachfolger und Wege

- Die Kantenmenge $E_{intra} = E_{succ} \cup E_{branch}$ heißt **intraprozedurale Kanten**.
- Die Knotenmenge $succ_v = \{v_r \in V \mid (v, v_r) \in E_{intra}\} \subseteq V$ heißt **intraprozedurale Nachfahren** von v.
- Die Knotenmenge $pred_v = \{v_l \in V \mid (v_l, v) \in E_{intra}\} \subseteq V$ heißt **intraprozedurale Vorgänger** von v.
- Eine Folge von Knoten $W = (v_1, \ldots, v_n)$ mit $v_1, \ldots, v_n \in V$ heißt **intraprozeduraler Weg** von v_1 nach v_n, falls gilt: $\forall i \in \{1, \ldots, n-1\}: (v_i, v_{i+1}) \in E_{intra}$.
  $W(i) = v_i$ heißt **der i-te Knoten** von W.
  v_1 heißt **erster Knoten von** W oder $First_W$.
  v_n heißt **letzter Knoten von** W oder $Last_W$.

- Die Menge aller intraprozeduraler Wege heißt $\boldsymbol{Path}$.

A.1.2. Operanden und Werte

Definition 4 Operanden, Werte, Belegung und Belegungstransformation

- Die Menge Op heißt **Operandenmenge** und besteht aus den Teilmengen Op_{global} und Op_{local}. $Op_{global} \cap Op_{local} = \emptyset$.
- Die Menge Val heißt **Wertemenge**. Der Wert $NIL \in Val$ heißt **Nullwert**.
- Eine **Operandenbelegung** ist eine Abbildung $Op \to Val$.
  Die Menge aller möglicher Operandenbelegungen heißt $\boldsymbol{Stat}$.
- Eine **aufruflokale Operandenbelegung** ist eine Abbildung $Op_{local} \to Val$.
  Die Menge aller möglicher aufruflokaler Operandenbelegungen heißt $\boldsymbol{Stat_{local}}$.
- Eine **aufrufglobale Operandenbelegung** ist eine Abbildung $Op_{global} \to Val$.
  Die Menge aller möglicher aufrufglobaler Operandenbelegungen heißt $\boldsymbol{Stat_{global}}$.
- Eine **Belegungstransformation** ist eine Abbildung $Stat \to Stat$.
  Die Menge aller möglicher Belegungstransformationen heißt $\boldsymbol{TransStat}$.

Definition 5 Lokale und globale Operandenbelegung

Sei $s \in Stat$, dann heißen die Abbildungen:

- $local: Stat \to Stat_{local}$, $local(s) = s_{local}$ mit $\forall o \in Op_{local}: s_{local}(o) = s(o)$ **die lokale Operandenbelegung von** s.
- $global: Stat \to Stat_{global}$, $global(s) = s_{global}$ mit $\forall o \in Op_{global}: s_{global}(o) = s(o)$ **die globale Operandenbelegung von** s.

Definition 6 Belegungsvereinigung $\oplus$

Sei $s_{local} \in Stat_{local}$ und $s_{global} \in Stat_{global}$.

Die Operation $\oplus$ heißt **Belegungsvereinigung** und sei definiert durch:

$$\forall o \in Op: (s_{local} \oplus s_{global})(o) = (s_{global} \oplus s_{local})(o) = \begin{cases} s_{local}(o) \text{ , falls } o \in Op_{local} \\ s_{global}(o) \text{ , falls } o \in Op_{global} \end{cases}$$

In der folgenden Arbeit wird davon ausgegangen, dass Val die folgenden Teilmengen enthält:

- Den **Nullwert** NIL.
- Die **Wahrheitswerte** $true$ und $false$;
- Die Menge der natürlichen Zahlen $\mathbb{N}$.
- Die Operandenmenge Op.

A.1.3. Intraprozeduraler Folgeknoten und Folgebelegungen

Definition 7 $Succ_v$ und $TransStat_v$

- Für jeden Knoten $v \in (V \setminus V_{term})$ und Belegung s sei der intraprozedurale Folgeknoten durch die Abbildung $\boldsymbol{Succ_v}$: $\boldsymbol{Stat \to succ_v}$ definiert.

- Für jeden Knoten $v \in V_{instr}$ und Belegung s sei die Folgebelegung durch die Abbildung $\boldsymbol{TransStat_v} \in \boldsymbol{TransStat}$ definiert.
- Für jeden Knoten $v \in (V \setminus V_{instr})$ und Belegung s sei die Folgebelegung konstant als $\boldsymbol{TransStat_v(s) = s}$ definiert.

A.1.4. Funktionsaufrufe

Definition 8 Aufrufparameterzuordnung und Rückgabewertzuordnung

- Eine **Aufrufparameterzuordnung** ist eine partielle Abbildung $Op_{local} \to Op$.
  Die Menge aller möglicher Aufrufparameterzuordnungen heißt $\boldsymbol{Par}$.
- Eine **Rückgabewertzuordung** ist eine partielle Abbildung $Op \to Op_{local}$.
  Die Menge möglicher Rückgabewertzuordungen heißt $\boldsymbol{Ret}$.
- Für jeden Knoten $v \in (V_{call} \cup V_{fork})$ und Belegung s sei die Aufrufparameterzuordnung des erzeugten Aufrufs durch eine Abbildung $\boldsymbol{Par_v}$: **Stat** $\to \boldsymbol{Par}$ definiert.
- Für jeden Knoten $v \in (V_{call} \cup V_{fork})$ und Belegung s sei e die Rückgabewertzuordung des erzeugten Aufrufs durch eine Abbildung $\boldsymbol{Ret_v}$: $\boldsymbol{Stat} \to \boldsymbol{Ret}$ definiert.

Definition 9 Aufruf

- Ein Tupel $(v, s, r) \in V \times Stat_{local} \times Ret$ mit Knoten $v \in V$, Belegung $s \in Stat_{local}$ und Rückgabewertzuordnung r heißt **Aufruf**.
  Die Menge aller möglicher Aufrufe heißt $\boldsymbol{Call}$.
- Ein Aufruf $c = (v, s, r) \in Call$ heißt **terminiert**, wenn $v \in V_{term}$.

Definition 10 $Trans_{call}$

Sei $p \in Par$ und $s \in Stat$.

- Die Parameterübergabetransformation $\boldsymbol{Trans_{call}}$: $\boldsymbol{Stat \times Par \to Stat_{local}}$ sei definiert als
  $Trans_{call}(s, p) = s'$ mit
  $\forall o \in Op_{local}: s'(o) = \begin{cases} s(p(o)) & \text{, falls } o \in \text{Def}(p) \\ NIL & \text{, sonst} \end{cases}$

Definition 11 $Trans_{return}$

Sei $r \in Ret$, $s \in Stat$ und $s_{local} \in Stat_{local}$.

- Die Rückgabewerttransformation $\boldsymbol{Trans_{return}}$: $\boldsymbol{Stat \times Ret \times Stat_{local}}$ sei definiert als
  $Trans_{return}(s, \text{r}, s_{local}) = s'$ mit
  $\forall o \in Op: s'(o) = \begin{cases} s_{local}(r(o)) & \text{, falls } o \in \text{Def}(r) \\ s(o) & \text{, sonst} \end{cases}$

A.1.5. Programmfäden

Definition 12 Faden und laufender Aufruf

- Eine geordnete Folge von Aufrufen $t = (c_{1,\dots}, c_n)$ mit $\forall 1 \leq i \leq n: c_i = (v_i, s_i, r_i) \in Call$ heißt **Programmfaden** oder kurz **Faden**. Der jeweils letzte Aufruf $c_n = (v_n, s_n, r_n)$ heißt auch der **laufende Aufruf** von t.
  Für jeden Aufruf eines Fadens $c = (v, s, r) \in Call$, außer dem laufenden Aufruf, gilt: $v \in V_{call}$.
  Die Menge aller möglichen Programmfäden heißt $\boldsymbol{Thread}$.
- Ein Faden t heißt **terminiert**, wenn er nur noch einen einzigen Aufruf $(v, s, r) \in Call$ enthält und $v \in V_{term}$.
- $Thr_p \subseteq Thread$ sei die Menge der Fäden, die im Programm p zum aktuellen Zeitpunkt existieren.
- Die Abbildung $\boldsymbol{Thr}: \boldsymbol{Call} \rightarrow \boldsymbol{Thread}$ weise jedem existierenden Aufruf denjenigen Faden zu, der den Aufruf beinhaltet.
- Die Abbildung $\boldsymbol{Top}: \boldsymbol{Thread} \rightarrow \boldsymbol{Call}$ weise jedem Faden innerhalb des angegebenen Programmzustands seinen laufenden Aufruf zu: $Top(t) = c$.

Definition 13 Parent

Sei $t \in Thr_p$ ein Programmfaden.

- Die Abbildung $\boldsymbol{parent}: \boldsymbol{Thr_p} \rightarrow (\boldsymbol{Thr_p} \times \boldsymbol{Call} \times \boldsymbol{V_{fork}}) \cup \emptyset$ weise jedem Faden t denjenigen Faden, Aufruf und Aufrufknoten zu, in welchem der Faden erzeugt wurde.
  Für t_{main} gelte: $parent(t_{main}) = \emptyset$.

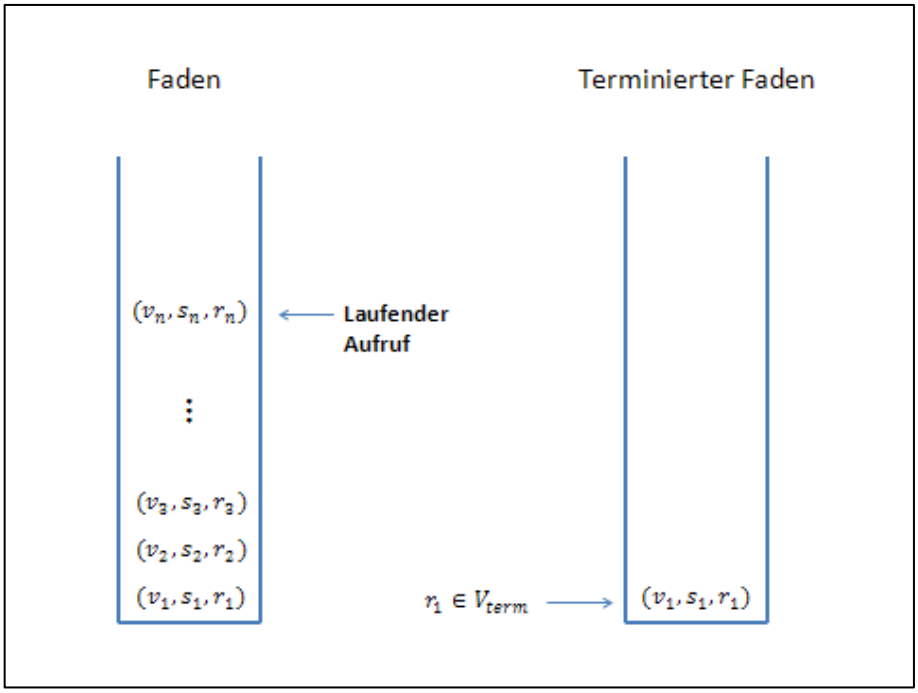

Abbildung 36: Programmfäden

A.1.6. Programmablauf

Der Ablauf eines Programms p mit Startknoten $u \in V_{method}$ und Startzustand $s_0 \in Stat$ kann nun folgendermaßen beschrieben werden:

1. Initialisierung:
   a) Die globale Belegung des Programms s_{global} wird mit $global(s_0)$ initialisiert.
   b) Der Hauptfaden wird mit $t_{main} = (c)$ mit Aufruf $c = (u, local(s_0))$ initialisiert und in Thr_p eingefügt.
   c) Fahre mit Schritt 2. fort
2. Prüfung:
   a) Falls $\forall t \in Thr_p: t$ ist terminiert, dann beende das Programm, ansonsten fahre mit Schritt 3. fort.
3. Programmschritt: Wähle einen beliebigen nicht terminierten Faden $t \in Thr_p$.
   Im folgenden sei $c_{top} = (v_{top}, s_{top}, r_{top}) \in Call$ der laufende Aufruf und $c_{top-1} = (v_{top-1}, s_{top-1}, r_{top-1}) \in Call$ der darunterliegende Aufruf von t.
   a) Falls $v_{top} \in V_{term}$:
      - Berechne $s_{comp} = s_{global} \oplus s_{top-1}$
      - Berechne $s_{new} = Trans_{return}(s_{comp}, r_{\text{top}}, s_{top})$
      - Berechne $v_{new} = Succ_{v_{top-1}}(s_{new})$
      - Weise s_{global} die Belegung $global(s_{new})$ zu
      - Entferne c_{top} aus t
      - Ersetze c_{top-1} durch $(v_{new}, local(s_{new}), r_{top-1})$
      - Gehe zu Schritt 2
   b) Falls $v_{top} \in V_{call}$:
      Sei $v' \in V_{method}$ mit $(v_{top}, v') \in E_{call}$ derjenige Knoten, auf welchen die Aufrufkante von v_{top} zeigt.
      - Berechne $s_{comp} = s_{global} \oplus s_{top}$
      - Berechne $s_{newlocal} = Trans_{call}(s_{comp}, Par_{v_{top}}(s_{comp}))$
      - Berechne $r_{new} = Ret_{v_{top}}(s_{comp})$
      - Füge einen neuen Aufruf $c_{top+1} = (v', s_{newlocal}, r_{new})$ über c_{top} in t ein. c_{top+1} ist ab jetzt der neue laufende Aufruf von t.
      - Gehe zu Schritt 2
   c) Falls $v_{top} \in V_{fork}$:
      Sei $v' \in V_{method}$ mit $(v_{top}, v') \in E_{fork}$ derjenige Knoten, auf welchen die Aufrufkante von v_{top} zeigt.
      - Berechne $s_{comp} = s_{global} \oplus s_{top}$
      - Berechne $s_{newlocal} = Trans_{call}(s_{comp}, Par_{v_{top}}(s_{comp}))$
      - Berechne $r_{new} = Ret_{v_{top}}(s_{comp})$

- Erzeuge einen neuen Faden $t' \in Thr_p$
  mit Aufruf $c_1 = (v', s_{newlocal}, r_{new})$
- Berechne $v_{new} = Succ_{v_{top}}(s_{comp})$
- Ersetze c_{top} durch $(v_{new}, s_{top}, r_{top})$
- Gehe zu Schritt 2

d) Falls $v_{top} \in V_{join}$:
Sei $v' \in V_{fork}$ mit $(v', v_{top}) \in E_{join}$ derjenige Knoten, von welchem aus die Synchronisationskante auf v_{top} zeigt.

- Falls es einen Faden $t' \in Thr_p$ gibt mit $parent(t) = (t, c_{top}, v')$ und t' ist terminiert:
  Sei $c' = (v', s', r')$ der terminierte Aufruf von t'
  - Berechne $s_{comp} = s_{global} \oplus s_{top}$
  - Berechne $s_{new} = Trans_{return}(s_{comp}, \mathrm{r}', \mathrm{s}')$
  - Weise s_{global} die Belegung $global(s_{new})$ zu
  - Ersetze c_{top} durch $(v_{top}, local(s_{new}), r_{top})$
  - Entferne t' aus Thr_p
  - Gehe zu Schritt 2
- Falls es keinen Faden $t' \in Thr_p$ mehr gibt mit $parent(t) = (t, c_{top}, v')$:
  - Berechne $s_{comp} = s_{global} \oplus s_{top}$
  - Berechne $v_{new} = Succ_{v_{top}}(s_{comp})$
  - Ersetze c_{top} durch $(v_{new}, s_{top}, r_{top})$
  - Gehe zu Schritt 2
- Ansonsten:
  - Gehe zu Schritt 2

e) Falls $v_{top} \in V_{cancel}$:
Sei $v' \in V_{fork}$ mit $(v', v_{top}) \in E_{join}$ derjenige Knoten, von welchem aus die Synchronisationskante auf v_{top} zeigt.

- Entferne alle Fäden $t' \in Thr_p$ gibt mit $parent(t) = (t, c_{top}, v')$ aus Thr_p.
- Gehe zu Schritt 2

f) Falls $v_{top} \in (V_{instr} \cup V_{branch})$:

- Berechne $s_{comp} = s_{global} \oplus s_{top}$
- Berechne $v_{new} = Succ_{v_{top}}(s_{comp})$
- Berechne $s_{new} = TransStat_{v_{top}}(s_{comp})$
- Weise s_{global} die Belegung $global(s_{new})$ zu
- Ersetze c_{top} durch $(v_{new}, local(s_{new}), r_{top})$

- Gehe zu Schritt 2

A.1.7. Programmzustand

Definition 14 Programmzustand

- Ein **Programmzustand** zu einem bestimmten Zeitpunkt sei definiert durch ein Tupel $z = (s_{global}, Thr_p)$, wobei $s_{global} \in Stat$ und $Thr_p \subseteq Thread$.
- Die Menge aller möglicher Programmzustände heißt $\boldsymbol{ProgStat}$.

Definition 15 Programmzustandsfolge und Ausführungsfaden

- Eine **mögliche Programmzustandsfolge** mit Startzustand $z \in ProgStat$ ist eine Folge von Programmzuständen $Z = (z_0, \dots, z_n)$ mit $\forall 0 \leq i \leq n: z_i \in ProgStat$, für die gilt:
  - $z_0 = z$ und
  - $\exists t_1, \dots, t_n \in Thr_p$, so dass $\forall 1 \leq i \leq n$:
    - z_{i-1} ist der aktuelle Programmzustand in Schritt 2 des Programmablaufs (siehe Kapitel A.1.6Programmablauf),
    - der danach verarbeitete Faden in Schritt 3 des Programmablaufs ist t_i
    - und der aktuelle Programmzustand beim nächsten Erreichen von Schritt 2 ist z_i.

    Im Folgenden nennen wir den zu einem z_i gehörenden Faden t_i den **Ausführungsfaden** vor z_i $\boldsymbol{Exec(z_i)} = t_i$.
- Die Menge aller möglicher Programmzustandsfolgen heißt $\boldsymbol{ProgPath}$.

Im Folgenden wird davon ausgegangen, dass innerhalb einer möglichen Programmzustandsfolge die Fäden und Aufrufe eines beliebigen Programmzustands den Fäden und Aufrufen seines Folgezustands eindeutig zugeordnet werden können.
Ein Bezeichner für einen Faden bzw. Aufruf behält damit auch über folgende Programmzustände seine Gültigkeit, solange der Faden im Programmzustand existiert.
Im Folgenden wird für die eindeutige Kennzeichnung der Bezeichner des Programmzustands hochgestellt an den Bezeichner des Fadens bzw. Aufrufs gehängt, z.B. t^{z_0} oder c^{z_0}.

A.1.8. Blöcke

Definition 16 Erreichbarkeit

Sei $B \subseteq V$ und $v, v' \in B$, dann heißt:

- v' **von** v **innerhalb** B **erreichbar**, falls es einen Weg $W = (v, \dots, v') \in Path$ gibt mit $W \subseteq B$

Definition 17

Sei $B \subseteq V$ und $W = (v_1, \dots, v_n)$ ein intraprozeduraler Weg, dann heißt:

- $First_{W,B} = \{v_i \mid i \in (1, \dots, n); v_i \in B; \forall 1 \leq j < i: v_j \notin B\}$ **erster Knoten** von W in B.
- $Last_{W,B} = \{v_i \mid i \in (1, \dots, n); v_i \in B; \forall n \geq j > i: v_j \notin B\}$ **letzter Knoten** von W in B.

Definition 18 Blöcke

Sei $B \subseteq V$, dann heißt:

- B **eingangskonsistent**, falls für alle Wege $W, W' \subseteq V$ mit $W \cap B \neq \emptyset$, $W' \cap B \neq \emptyset$, $First_W \notin B$und $First_{W'} \notin B$ gilt: $First_{W,B} = First_{W',B}$, d.h. alle Wege durch B, die außerhalb B beginnen, haben denselben ersten Knoten in B.
  Für eingangskonsistente B gilt die Kurzschreibweise:
  $First_B = First_{W,B}$ für beliebigen Weg $W \subseteq V$ mit $W \cap B \neq \emptyset$.
- B **ausgangskonsistent**, falls für alle Wege $W, W' \subseteq V$ mit $W \cap B \neq \emptyset$, $W' \cap B \neq \emptyset$, $Last_W \notin B$ und $Last_{W'} \notin B$ gilt: $Last_{W,B} = Last_{W',B}$, d.h. alle Wege durch B, die außerhalb B enden, haben denselben letzten Knoten in B.
  Für ausgangskonsistente B gilt die Kurzschreibweise:
  $Last_B = Last_{W,B}$ für beliebigen Weg $W \subseteq V$ mit $W \cap B \neq \emptyset$.
- B **zusammenhängend**, falls ein $v \in B$existiert mit $\forall v' \in B$: v' ist von v innerhalb B erreichbar.
- B **Block**, falls B eingangskonsistent, ausgangskonsistent und zusammenhängend ist und $Last_B \in V_{instr}$.
- B **nichtterminal**, falls $\forall v \in B: succ(v) \neq \emptyset$.

A.1.9. Schreib-, Lese- und Zugriffsoperandenmengen

Definition 19 Schreib-, Lese- und Zugriffsmenge von Knoten

Sei $U \subseteq V$, $Z = (z_0, \dots) \in ProgPath$, $z_0 = (s_{global}, Thr_p) \in ProgStat$ und
$c = (v_0, s_{local}, r) \in Call$ mit $Thr(c) \in Thr_p$.

- Die **Schreiboperandenmenge** $\boldsymbol{Write_U^{Z,c}}$ enthalte alle Operanden, die im Verlauf der Programmzustandsfolge Z innerhalb des Aufrufs c von Knoten $u \in U$ geschrieben werden.
  Falls $u \in V_{call}$, so werden auch alle globalen Operanden hinzugezählt, auf die der erzeugte Aufruf im Verlauf von Z schreibt.
  Falls $u \in V_{fork}$, so werden auch alle globalen Operanden hinzugezählt, auf die der erzeugte Faden im Verlauf von Z schreibt.
- Die **Leseoperandenmenge** $\boldsymbol{Read_U^{Z,c}}$ enthalte alle Operanden, die im Verlauf der Programmzustandsfolge Z innerhalb des Aufrufs c von Knoten $u \in U$ gelesen werden.
  Falls $u \in V_{call}$, so werden auch alle globalen Operanden hinzugezählt, die der erzeugte Aufruf im Verlauf von Z liest.
  Falls $u \in V_{fork}$, so werden auch alle globalen Operanden hinzugezählt, die der erzeugte Faden im Verlauf von Z liest.
- Die **Zugriffsoperandenmenge** $\boldsymbol{Access_U^{Z,c}}$ sei definiert durch
  $Access_U^{Z,c} = Write_U^{Z,c} \cup Read_U^{Z,c}$

Definition 20 Lokaler Pfad

Sei $Z = (z_0, \dots, z_n) \in ProgPath,\ c \in Call$

- Der **lokale Pfad von c in Z** sei definiert als Folge von Knoten:
  $\boldsymbol{Path(Z,c) = (v_i | \forall 0 \leq i \leq n: c^{z_i} = (v_i, s_i, r_i))}$

Definition 21 Schreib-, Lese- und Zugriffsmenge von Knoten innerhalb eines Blocks

Sei $B \subseteq V$ ein Block, $U \subseteq B$ eine Knotenmenge, $z \in ProgStat$, $c \in Call,\ t = Thr(c)$

- $Write_U^{z,c,B} = \bigcup_Z Write_U^{Z,c}$ für die gilt:
  $Z = (z_0, \dots, z_n) \in ProgPath \wedge \exists Z' = (z, \dots, z_0, \dots, z_n) \in ProgPath$:
  $Path(Z', c) \subseteq B \wedge Path(Z, c) \subseteq U$
- $Read_U^{z,c,B} = \bigcup_Z Read_U^{Z,c}$ für die gilt:
  $Z = (z_0, \dots, z_n) \in ProgPath \wedge \exists Z' = (z, \dots, z_0, \dots, z_n) \in ProgPath$:
  $Path(Z', c) \subseteq B \wedge Path(Z, c) \subseteq U$
- $Access_U^{z,c,B} = Write_U^{z,c,B} \cup Read_U^{z,c,B}$

Definition 22 Anweisungsgetrennt, Kette, Datenabhängigkeit

Eine Folge von Blöcken $M = (B_1, \dots B_n)$ heißt:

- **anweisungsgetrennt**, falls $\forall B, B' \in M: B \cap B' = \emptyset$, d.h. ein Knoten ist in höchstens einem der Blöcke enthalten.
- **Kette**, falls
  - M ist anweisungsgetrennt,
  - $First_M = First_{B_1}$,
  - $Last_M = Last_{B_n}$,
  - $\forall 2 \leq i \leq n: (Last_{B_{i-1}}, First_{B_i}) \in E$,
  - $\nexists 2 \leq i \leq n, v \in V: v \neq First_{B_i} \wedge (Last_{B_{i-1}}, v) \in E$ und
  - $\nexists 2 \leq i \leq n, v \in V: v \neq Last_{B_{i-1}} \wedge (v, First_{B_i}) \in E$.
- **datenabhängig**, falls
  $\forall z \in ProgStat, c \in Call, B, B' \in M: Write_B^{z,c,M} \cap Access_{B'}^{z,c,M} \neq \emptyset$
- **datenunabhängig**, falls M nicht datenabhängig ist.

A.2. Funktionsextraktion (M1)

Definition 23 Funktionsextraktion

Sei $M = \{B_1, \dots, B_n\}$ eine datenunabhängige Kette von nichtterminalen Blöcken.

Die folgende Programmtransformation heißt **Funktionsextraktion**:

1. Füge für jeden Block B_i neue Knoten $v_{fork,i}$ in V_{fork}, $v_{join,i}$ in V_{join}, $v_{method,i}$ in V_{method} und $v_{term,i}$ in V_{term} ein.
   Die Parameterübergabezuordnung von $v_{fork,i}$ ist jeweils so zu wählen, dass alle innerhalb B_i gelesenen lokalen Operanden an den Aufruf übergeben werden.
   Die Rückgabewertzuordnung von $v_{join,i}$ ist jeweils so zu wählen, dass alle innerhalb B_i geschriebenen und vom Aufrufer gelesenen lokalen Operanden an den Aufrufer zurückgegeben werden.

2. Ersetze jede Kante $(v, First_{B_1}) \in E_{succ} \cup E_{branch}$ mit $v \in V \setminus M$ durch eine Kante $(v, v_{fork,1})$ und jede Kante $(Last_{B_n}, v) \in E_{succ}$ mit $v \in V \setminus M$ durch eine Kante $(v_{join,n}, v)$.
3. Füge neue Kante $(v_{fork,n}, v_{join,1})$ in E_{succ} ein.
4. $\forall 2 \leq i \leq n$: Füge neue Kante $(v_{fork,i-1}, v_{fork,i})$ und $(v_{join,i-1}, v_{join,i})$ in E_{succ} ein.
5. $\forall 2 \leq i \leq n$: Entferne Kante $(Last_{B_{i-1}}, First_{B_i})$ aus E_{succ}.
6. $\forall 1 \leq i \leq n$: Füge neue Kante $(v_{method,i}, First_{B_i})$ in E_{succ} ein.
7. $\forall 1 \leq i \leq n$: Füge neue Kante $(Last_{B_i}, v_{term,i})$ in E_{succ} ein.
8. $\forall 1 \leq i \leq n$: Füge neue Kante $(v_{fork,i}, v_{method,i})$ in E_{fork} ein.
9. $\forall 1 \leq i \leq n$: Füge neue Kante $(v_{fork,i}, v_{join,i})$ in E_{join} ein.

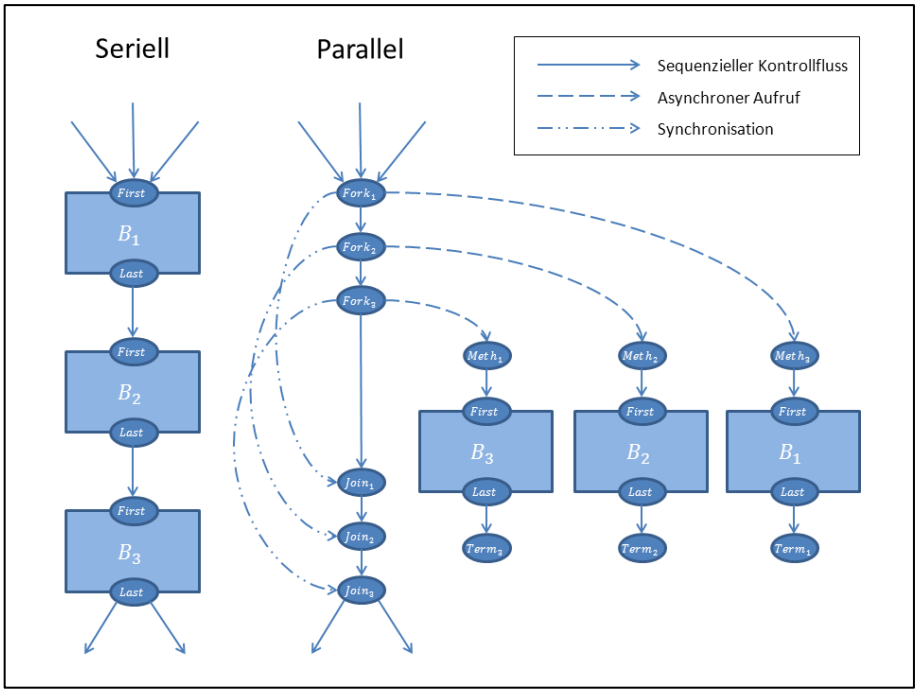

Abbildung 37: Parallelisierung durch Funktionsextraktion

Wie in Abbildung 37 dargestellt, wird hierbei für jeden Block der Kette in eine eigene Methode erzeugt und aus dem aktuellen Faden heraus per asynchronem Aufruf ausgeführt. Danach wird auf das Ende jedes neu erzeugten Fadens gewartet und die Ergebnisse anschließend zurück in den Speicher des ursprünglichen Fadens geschrieben.

A.3. Spekulative Berechnung (M2)

Definition 24 Spekulative Berechnung

Seien B, B' nicht-terminale Blöcke mit $B' \subset B \subseteq V$ und $\forall z \in ProgStat, c \in Call$ gelte:

- $Write^{z,c,B}_{(B \setminus B')} \cap Access^{z,c,B}_{B'} = \emptyset$ und
- $Access^{z,c,B}_{(B \setminus B')} \cap Write^{z,c,B}_{B'} = \emptyset$ und

- $Access_{(V\setminus B)}^{z,c,V} \cap Write_{B'}^{z,c,V} = \emptyset$

Die folgende Programmtransformation heißt **Spekulative Berechnung**:

1. Füge neue Knoten v_{fork} in V_{fork}, v_{join} in V_{join}, v_{cancel} in V_{cancel}, v_{method} in V_{method} und v_{term} in V_{term} ein.
   Die Parameterübergabezuordnung von v_{fork}, sowie die Rückgabewertzuordnung von v_{join} sind wie in Kapitel A.2 Funktionsextraktion (M1) zu wählen.
2. Ersetze jede Kante $(v, First_B) \in E_{succ} \cup E_{branch}$ mit $v \in (V \setminus B)$ durch eine Kante (v, v_{fork}) und jede Kante $(Last_B, v) \in E_{succ}$ mit $v \in (V \setminus B)$ durch eine Kante (v_{cancel}, v).
3. Füge neue Kanten $(v_{fork}, First_B), (Last_B, v_{cancel})$ in E_{succ} ein.
4. Ersetze jede Kante $(v, First_{B'}) \in E_{succ} \cup E_{branch}$ mit $v \in (B \setminus B')$ durch eine Kante (v, v_{join}) und jede Kante $(Last_{B'}, v) \in E_{succ}$ mit $v \in (B \setminus B')$ durch eine Kante (v_{join}, v).
5. Füge neue Kanten $(v_{method}, First_{B'}), (Last_{B'}, v_{term})$ in E_{succ} ein.
6. Füge neue Kanten (v_{fork}, v_{method}) in E_{fork}, (v_{fork}, v_{join}) in E_{join}, sowie (v_{fork}, v_{cancel}) in E_{cancel} ein.

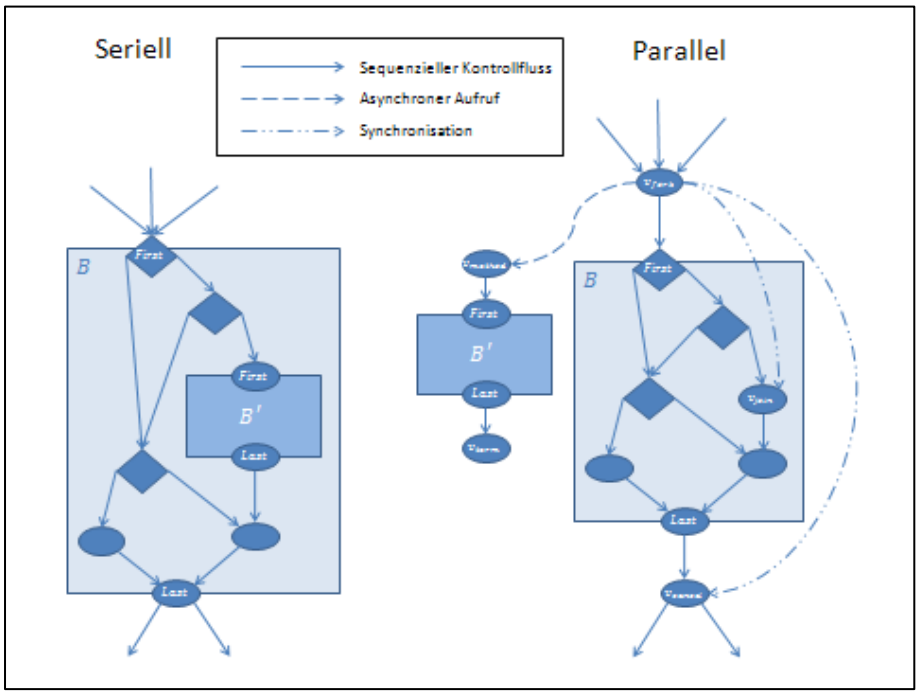

Abbildung 38: Parallelisierung durch spekulative Berechnung

Wie in Abbildung 40 dargestellt, wird der innere Block per Funktionsextraktion isoliert und die erzeugte Methode vor Ausführung des äußeren Blocks asynchron aufgerufen. Im Gegensatz zur puren Funktionsextraktion in Abschnitt A.2 ist zum Zeitpunkt des asynchronen Aufrufs jedoch noch nicht klar, ob das Ergebnis der Berechnung tatsächlich benötigt wird.

Falls die Programmausführung am ursprünglichen Ort der Berechnung ankommt, wird das Ergebnis des erzeugten Fadens in den Speicher des ursprünglichen Fadens geschrieben.

Falls die Programmausführung vorher den äußeren Block verlässt, so wird entweder auf das Ende des erzeugten Fadens gewartet oder dieser wird aktiv abgebrochen. Anschließend wird die Ausführung des Programms fortgesetzt.

A.4. Schleifenparallelisierung

Die Parallelisierung von Schleifen ist ein intensiv erforschtes Gebiet und es gibt bereits vielfältige Ansätze. Im Zuge dieser Studienarbeit wurden folgende Mechanismen für die Parallelisierung von Schleifen eingesetzt: Funktionsextraktion und asynchroner Aufruf (siehe Abschnitt A.2), sowie Work-Stealing.

Definition 25 Schleifen

Ein Block $L \subseteq V$ heißt:

- **Schleife**, falls $\forall v \in L$: $First_L$ ist von v in L erreichbar.
- **Bedingte Schleife**, falls $\exists v \in L, v' \notin L: v' \in succ(v)$.
  v heißt dann **Bedingungsknoten** $v_{Cond,L}$ und
  $Cond_L = Succ_v$ heißt **Schleifenbedingung** von L.
  Für bedingte Schleifen gilt immer: $v_{Cond,L} = Last_L$.

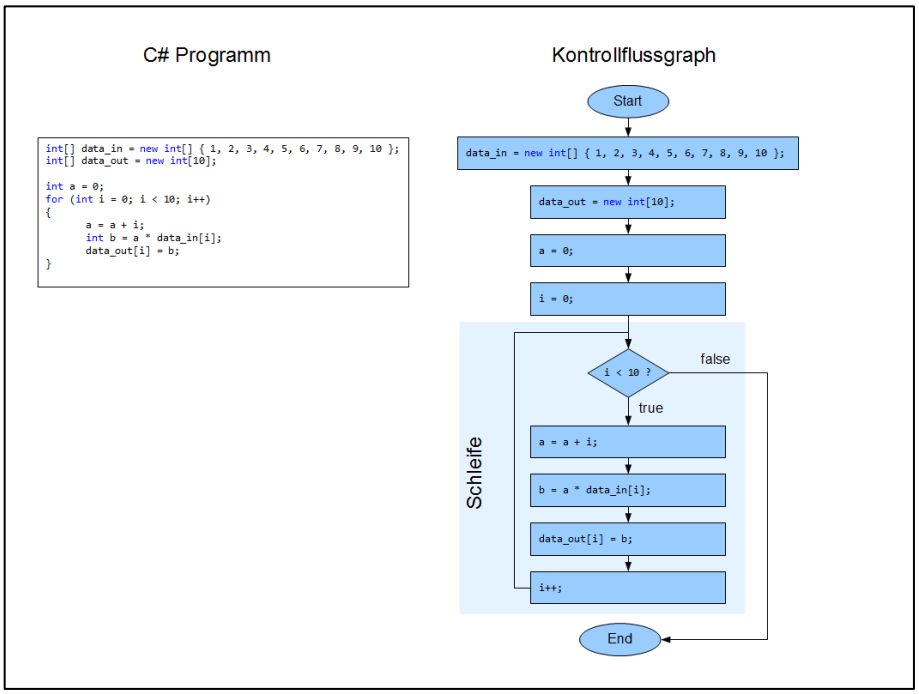

Abbildung 39: Programmablaufgraph einer Schleife

A.4.1. Schleifen über Kollektionen

Eine Kollektion ist eine Gruppe von Elementen desselben Typs. Sie kann geordnet oder ungeordnet sein.

Bekannte Kollektionsdatentypen sind z.B.:

- Menge (Set): ungeordnet, keine Duplikate erlaubt
- Mehrfachmenge (Bag): ungeordnet, Duplikate erlaubt
- Assoziatives Array: Menge von Schlüssel/Wert-Paaren, normalerweise ungeordnet

- Liste: geordnet
- Warteschlange (Queue): geordnet, Definiertes Kopfelement (FIFO)
- Stapel (Stack): geordnet, Definiertes Kopfelement (LIFO)

Schleifen über Kollektionen werden in vielen Hochsprachen z.B. mit dem Schlüsselwort FOREACH eingeleitet. Hierbei eine definierte Schleifenvariable in genau einer Schleifeniteration mit einem Element der Eingabekollektion belegt und der Schleifenrumpf für diese Belegung ausgeführt. Die Schleife wird endet, wenn alle Elemente der Kollektion verarbeitet wurden.

Die Reihenfolge, in welcher die einzelnen Elemente der Kollektion durchlaufen werden, hängt dabei direkt von der Implementierung des Kollektionsdatentyps selbst ab.

Schleifen über Kollektionen bieten unter bestimmten Bedingungen die im Folgenden beschriebenen Parallelisierungsmöglichkeiten.

Definition 26 Kollektion und Iterator

- Eine **Kollektion** sei ein Tupel $K = (Elem_K, Cnt_K)$ mit $Cnt_K \in Op$,
  $\forall s \in Stat: s(Cnt_K) \in \mathbb{N}$ und Operandenfolge $Elem_K = (k_0, \dots, k_{Cnt_K}), k_i \in Op$.
  Die Menge aller möglichen Kollektionen heißt $\boldsymbol{Collection}$.
- Ein **Iterator** über einer Kollektion K_{It} sei ein Tupel $It = (K_{It}, Idx_{It})$ mit
  $K_{It} \in Collection$ und $\forall s \in Stat: s(Idx_{It}) \in \mathbb{N}$.
  Die Menge aller möglicher Iteratoren über K heißt $\boldsymbol{Iterator_K}$.
- Die folgenden Belegungstransformationen für Kollektionen seien für alle
  $K \in Collection, It = (K, Idx_{It}) \in Iterator_K, o, q \in Op$ definiert:
  - $Count_K^{q\leftarrow}(s) = s'$ mit $s'(o) = \begin{cases} \text{falls } o = q & : s(Cnt_K) \\ \text{sonst} & : s(o) \end{cases}$
  - $Add_K^{q\rightarrow}(s) = s'$ mit $s'(o) = \begin{cases} \text{falls } o = Cnt_K & : s(Cnt_K) + 1 \\ \text{falls } o = k_i \text{ mit } i = s(Cnt_K) + 1 & : q \\ sonst & : s(o) \end{cases}$
  - $Iterator_K^{It\leftarrow}(s) = s'$ mit $s'(o) = \begin{cases} \text{falls } o = Idx_{It} & : 0 \\ sonst & : s(o) \end{cases}$
- Die folgenden Belegungstransformationen für Iteratoren seien für alle
  $K = (Elem_K, Cnt_K) \in Collection, Elem_K = (k_0, \dots, k_{Cnt_K}),\ k_i \in Op,$
  $It = (K, Idx_{It}) \in Iterator_K, o, Chk, El \in Op$ definiert:
  - $Reset_{It}(s) = s'$ mit $s'(o) = \begin{cases} falls\ o = Idx_{It} & : 0 \\ sonst & : s(o) \end{cases}$
  - $Next_{It}^{Chk,El\leftarrow}(s) = s'$ mit

$$s'(o) = \begin{cases} \text{falls } o = Idx_{It} \wedge s(Idx_{It}) < s(Cnt_K) & : s(Idx_{It}) + 1 \\ \text{falls } o = Idx_{It} \wedge s(Idx_{It}) \geq s(Cnt_K) & : s(Idx_{It}) \\ \text{falls } o = Chk \wedge s(Idx_{It}) < s(Cnt_K) & : true \\ \text{falls } o = Chk \wedge s(Idx_{It}) \geq s(Cnt_K) & : false \\ \text{falls } o = El \wedge s(Idx_{It}) < s(Cnt_K) & : s(k_i) \text{ mit } i = s(Idx_K) + 1 \\ \text{falls } o = El \wedge s(Idx_{It}) \geq s(Cnt_K) & : NIL \\ \text{sonst} & : s(o) \end{cases}$$

Im Folgenden gehen wir davon aus, dass die oben beschriebenen Belegungstransformationen (insbesondere $Next_{It}^{Chk,El\leftarrow}$) atomar durchgeführt werden.

Definition 27 Schleife über einer Kollektion

Sei $Chk, El \in Op$, $K \in Collection$, $It \in Iterator_K$ und $L \subseteq V$ eine Schleife mit Bedingungsknoten $v_{Cond,L}$, Schleifenbedingung $Cond_L$ und ein Knoten $v_{Next} \in L$. Es gelte:

- $\forall v \in \left(succ\left(v_{Cond,L}\right) \cap L\right), W = \left(v, \dots, v_{Cond,L}\right) \in Path: v_{Next} \in W$
- $TransStat_{v_{Next}} = Next_K^{chk,El\leftarrow}$
- $\forall s \in Stat$: wenn $s(Chk) = true \Rightarrow Succ_{v_{Cond,L}}(s) \in L$
- $\forall s \in Stat$: wenn $s(chk) = false \Rightarrow Succ_{v_{Cond,L}}(s) \notin L$

Dann heißt L **Schleife über der Kollektion K** mit Iterator It, Iterationsknoten v_{Next} und Schleifenvariable El.

A.4.2. Schleifenvervielfältigung (M3)

Definition 28 Schleifenvervielfältigung

Sei L eine nicht-terminale Schleife über der Kollektion $K \in Collection$ mit Iterator $It \in Iterator_K$, Iterationsknoten $v_{Next} \in L$ und Schleifenvariable $El \in Op$, $v_{out,1}, \dots, v_{out,n} \in L$ mit $n \geq 0$. Sei $m \geq 2$ der gewünschte Parallelitätsgrad.
Es gelte:

- $\#(succ(Last_L) \cap \{V \setminus L\}) = 1$
  Der erste Knoten nach dem Verlassen der Schleife ist eindeutig.
- $\forall 0 < i \leq n \; \exists K_{out,i} \in (Collection \setminus \{K\}),\ q \in Op: TransStat_{v_{out,i}} = Add_{K_{out,i}}^{q\rightarrow}$
  An allen $v_{out,i}$ werden ausschließlich Elemente zu Ausgabekollektionen hinzugefügt.
- $\forall Z_1 = (z_{beg}, \dots, z_{end}), Z_2 = (z'_{beg}, \dots, z'_{end}), Z = (\dots, Z_1, \dots, Z_2, \dots) \in ProgPath$, $c \in Call$ mit
  $z_{beg}, z'_{beg} \in succ(v_{next})$, $Path(Z, c) \subseteq L$, $Path(Z_1, c) \subseteq (L \setminus \{v_{next}\})$, $Path(Z_2, c) \subseteq (L \setminus \{v_{next}\})$ gilt:
  - $Write_{L\setminus\{v_{next},v_{out,1},\dots,v_{out,n}\}}^{\boldsymbol{Z_1},c} \cap Access_{L\setminus\{v_{next},v_{out,1},\dots,v_{out,n}\}}^{\boldsymbol{Z_2},c}$
  - $Write_{L\setminus\{v_{next},v_{out,1},\dots,v_{out,n}\}}^{\boldsymbol{Z_2},c} \cap Access_{L\setminus\{v_{next},v_{out,1},\dots,v_{out,n}\}}^{\boldsymbol{Z_1},c}$

  Verschiedene Iterationen sind datenunabhängig, abgesehen von Iterations- und Ausgabeknoten.

Die folgende Transformation heißt **Schleifenvervielfältigung**:

1. Füge neue Knoten v_{fork} in V_{fork}, v_{join} in V_{join}, v_{method} in V_{method} und v_{term} in V_{term} ein. Die Parameterübergabezuordnung von v_{fork} ist wie in Abschnitt A.2 Funktionsextraktion (M1) zu wählen.
2. Füge neue Knoten v_{init}, v_{inc} in V_{instr} und v_{chk} in V_{branch} ein.
3. Ersetze jede Kante $(v, First_L) \in E_{succ} \cup E_{branch}$ durch eine Kante (v, v_{init})

4. Ersetze die Kante $(Last_L, v_{leave}) \in E_{branch}$ mit $v_{leave} \notin L$ durch eine Kante $(Last_L, v_{term})$, füge eine Kante (v_{join}, v_{leave}) in E_{succ} ein und ersetze $Cond_L$ durch $Cond'_L$, wobei $Cond'_L(s) = \begin{cases} falls\ Cond_L(s) = v_{leave} & : v_{term} \\ sonst & : Cond_L(s) \end{cases}$
5. Füge eine neue Kante $(v_{method}, First_L)$ in E_{succ} ein.
6. Füge neue Kanten (v_{fork}, v_{method}) in E_{fork} und (v_{fork}, v_{join}) in E_{join} ein.
7. Füge neue Kanten $(v_{init}, v_{chk}), (v_{fork}, v_{inc}), (v_{inc}, v_{chk})$ in E_{succ} und $(v_{chk}, v_{fork}), (v_{chk}, v_{join})$ in E_{branch} ein.
8. Wähle einen ungenutzten Operanden $i \in Op$. Setze:
   $TransStat_{v_{init}}(s) = s'$ mit $s'(o) = \begin{cases} \text{falls } o = i & : 0 \\ \text{sonst} & : s(o) \end{cases}$
   $TransStat_{v_{inc}}(s) = s'$ mit $s'(o) = \begin{cases} \text{falls } o = i & : s(i) + 1 \\ \text{sonst} & : s(o) \end{cases}$
   $Succ_{v_{chk}}(s) = \begin{cases} \text{falls } s(i) < m & : v_{fork} \\ sonst & : v_{join} \end{cases}$

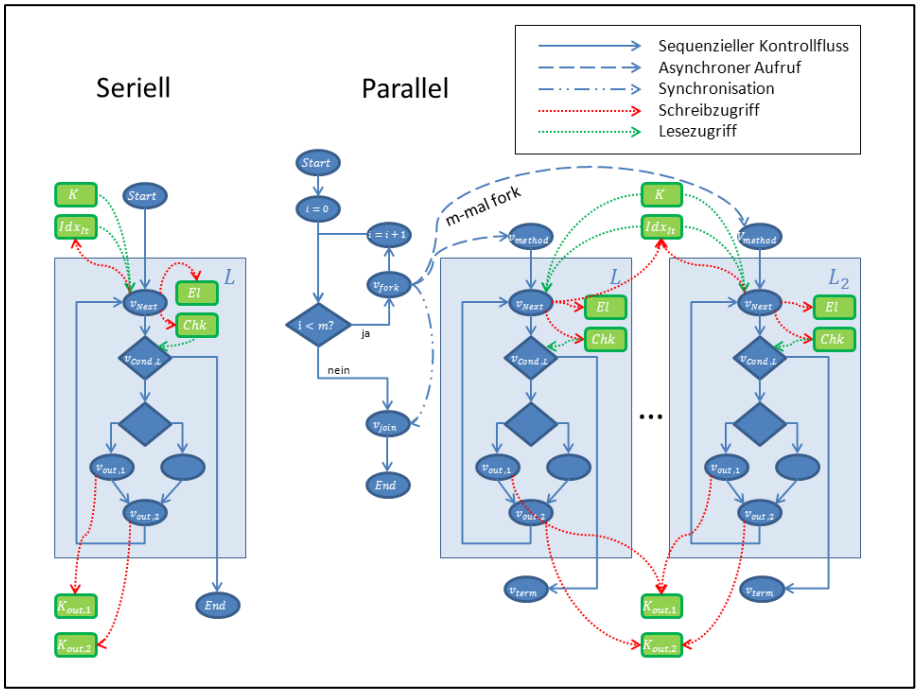

Abbildung 40: Parallelisierung durch Schleifenvervielfältigung

Abbildung 40 zeigt, wie eine Schleife gemäß der Programmtransformation nach Definition 28 mehrfach parallel ausgeführt werden kann.

Die zu parallelisierende Schleife wird per Funktionsextraktion (siehe Definition 23) in eine eigene Methode isoliert. Eine neue Schleife erzeugt die gewünschte Anzahl an Fäden durch asynchronen Aufruf der extrahierten Methode. Hierdurch kann die Anzahl der Fäden genau an die Anzahl der Ausführungseinheiten angepasst werden, was den Overhead für die Fadenerzeugung bei maximaler Parallelität minimiert.

Mittels Work-Stealing werden die einzelnen Elemente der Kollektion an die einzelnen Fäden verteilt und abgearbeitet. Wichtig ist hierbei, dass alle Fäden auf **demselben Iterator** arbeiten

und nicht auf getrennten. Das heißt, der Iterator muss außerhalb der extrahierten Methode erzeugt werden.

Die neuen Fäden terminieren, sobald der Iterator alle Elemente durchlaufen hat. Ein Rückgabewert der Fäden wird nicht benötigt, daher wird auch keine Rückgabewertzuordnung zu v_{join} definiert.

Zu beachten ist außerdem, dass die Reihenfolge der Elemente in den Ausgabekollektionen undefiniert ist. Falls die Reihenfolge für den weiteren Programmverlauf wichtig ist, so müssen zusätzliche Maßnahmen ergriffen werden, wie im Folgenden beschrieben.

A.4.3. Reihenfolgenerhaltung durch Sortieren

Die Elemente der Eingabekollektion werden zunächst durch einen Hauptindex ergänzt, so dass nun in jeder Iteration bekannt ist, an welcher Stelle der Eingabkollektion sich das aktuelle Element befindet.

Statt des Hinzufügens eines Elements El in eine Ausgabekollektion $K_{out,i}$ wird nun ein Tupel aus Ausgabeelement, der Hauptindex des Eingabeelements, sowie der aktuelle Subindex zu einer Pufferkollektion $K_{buffer,i}$ hinzugefügt. Der Subindex ist definiert als die Gesamtanzahl der Elemente, die in der aktuellen Iteration zu Kollektionen hinzugefügt wurden.

Um die Ausgabekollektionen mit korrekter Reihenfolge zu erhalten werden die entsprechende Pufferkollektionen erst nach Haupt- und dann nach Subindex sortiert und aus den entstehenden Tupel-Folgen die Ausgabeelemente extrahiert.

A.4.4. Beschränkte Anzahl von Ausgabeelementen pro Iteration

Falls für eine Ausgabekollektion K_{out} die maximale Anzahl an eingefügten Elementen pro Iteration beschränkt ist, so kann das abschließende Sortieren der Pufferkollektion wie folgt vermieden werden. Hierfür ist eine Form der Kollektion notwendig, die es erlaubt Elemente an einer beliebigen Position der Kollektion einzufügen.

Sei n die maximale Anzahl an Ausgabeelementen für K_{out} pro Iteration, m die gesamte Anzahl an Iterationen, i der Index der aktuellen Iteration, j der Subindex.

Vor Beginn der Schleife wird eine Pufferkollektion K_{buffer} mit einer Größe von $n * m$ Elementen initialisiert.

In den Iterationen werden Ausgabeelemente jeweils an der Position $i * n + j$ in K_{buffer} eingefügt.

Nach Beendigung aller Fäden werden alle gesetzten Elemente in K_{buffer} in derselben Reihenfolge in K_{out} eingefügt. Nicht gesetzte Elemente in K_{buffer} werden ignoriert.

Für dieses Vorgehen ist gegenüber dem Verfahren in Kapitel A.4.3 ein Laufzeitvorteil zu erwarten, wenn die endgültige Anzahl von Elementen in K_{out} ähnlich groß ist, wie der maximale Index von K_{buffer}, also $n * m$.

B. Literaturverzeichnis

[AF12] AForge.NET
http://code.google.com/p/aforge/

[ASL08] A.V. Aho, R. Sethi, M.S. Lam, Jeffrey D. Ullman; „Compiler: Prinzipien, Techniken und Werkzeuge“. Pearson Education Deutschland, 2008, ISBN 3-827-37097-3.

[BH77] Henry Baker, Carl Hewitt: “The Incremental Garbage Collection of Processes”. In: Proceedings of the Symposium on Artificial Intelligence Programming Languages, SIGPLAN Notices 12, August 1977.

[BN09] Big Number
http://bignumber.codeplex.com/

[CG11] Computational Geometry
http://compgeo.codeplex.com/

[DSA08] Data Structures and Algorithms (DSA)
http://dsa.codeplex.com/

[Evo09] Evo - Framework for evolutionary computation
http://evo.codeplex.com/

[Fre10] Adam Freeman: „Pro .NET 4 Parallel Programming in C#”. Apress, 2010, ISBN 1-430-22967-5.

[FS99] Ira R. Forman, Scott Danforth: “Putting Metaclasses to Work: New Dimension in Object-oriented Programming”. Addison Wesley, 1998, ISBN 0-201-43305-2.

[FF05] Ira R. Forman, Nate Forman: “Java Reflection in Action”. Manning, 2004, ISBN 1-932394-18-4.

[FW76] Daniel Friedman, David Wise: “The Impact of Applicative Programming on Multiprocessing”. In: International Conference on Parallel Processing, 1976, pp. 263-272.

[GBJ00] Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs, Koen G. Langendoen: “Modern Compiler Design”. John Wiley & Sons Inc., 2000, ISBN 0-471-97697-0.

[Gou12] Jean-Philippe Gouigoux: “Practical Performance Profiling: Improving the Efficiency of .Net Code”. Red Gate Books, 2012, ISBN: 1-906-43482-4.

[GP12] GPdotNET - Genetic Programming Tool
http://gpdotnet.codeplex.com/

[Hen06] Laurie Hendren: “Context-sensitive points-to analysis: Is it worth it?”. In: Compiler Construction, 15th International Conference, volume 3923 of LNCS, pp. 47-64., Springer, 2006.

[Hib76] Peter Hibbard: “Parallel Processing Facilities”. In: New Directions in Algorithmic Languages, (ed.) Stephen A. Schuman, IRIA, 1976.

[Huc10] Jochen Huck: „Automatisierte Parallelisierung mit Auto-Futures“, Karlsruhe Institute of Technology, 2010.

[Ric10] Jeffrey Richter: “CLR via C#”. Microsoft Press, 2010, ISBN 978-0-735-62704-8

[Mcc04] Steve McConnell: “Code Complete: A Practical Handbook of Software Construction”. Microsoft Press, 2004, ISBN 0-735-61967-0.

[Muc97] Steven S. Muchnick: “Advanced Compiler Design and Implementation”. Morgan Kaufmann, 1997, ISBN 978-1-558-60320-2.

[MRR12] Michael McCool, James Reinders, Arch Robison: “Structured Parallel Programming: Patterns for Efficient Computation”. Morgan Kaufmann, 2012, ISBN 0-124-15993-1.

[MSM04] Timothy G. Mattson, Beverly A. Sanders, Berna L. Massingill: “Patterns for Parallel Programming”. Addison-Wesley Longman, Amsterdam, 2004, ISBN 0-321-22811-1.

[PVC01] Michael Paleczny, Christopher Vick, Cliff Click: “The Java Hotspot(tm) Server Compiler”. In: USENIX Java Virtual Machine Research and Technology Symposium, 2001, pp. 1-12.

[Ros11] Microsoft® “Roslyn” http://msdn.microsoft.com/en-US/roslyn

[RR99] Radu Rugina, Martin Rinard: “Automatic parallelization of divide and conquer algorithms”. In: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '99, pp. 72-83, New York, USA, 1999, ACM, ISBN 1-58113-100-3. http://doi.acm.org/10.1145/301104.301111

[SSD09] Sift String Distance and Similarity algorithm http://sift.codeplex.com/

[Tan07] Andrew S. Tanenbaum: “Modern Operating Systems”. Prentice Hall International, 3rd Revised edition, 2007, ISBN 0-138-13459-6.

[TF10] Georgios Tournavitis, Björn Franke: “Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using proling information”. In: Proceedings of the 19th international conference on Parallel architectures and compilation techniques, PACT '10, pp. 377-388. New York, USA, 2010. ACM, ISBN 978-1-4503-0178-7. http://doi.acm.org/10.1145/1854273.1854321.

[WS11] Wordwheel Solver http://wordwheelsolver.codeplex.com/

[WZ91] Mark N. Wegman, F. Kenneth Zadeck: "Constant Propagation with Conditional Branches", In: ACM Transactions on Programming Languages and Systems, Volume 13 Issue 2, April 1991, pp. 181-210.