

## 2 Zielsetzung und Beitrag der Arbeit

In diesem Kapitel wird zunächst das Ziel der vorliegenden Dissertation definiert. Es stützt sich auf die eingangs erwähnte Problemstellung. Anschließend wird der Beitrag dieser Arbeit für den Forschungsbereich der automatischen Parallelisierung dargelegt und Grenzen des Verfahrens aufgezeigt. Zuletzt werden aus Zielsetzung und Beitrag vier zentrale Thesen postuliert, die in den folgenden Kapiteln erörtert werden.

### 2.1 Zielsetzung

Es ist eine intrinsische Eigenschaft moderner industrialisierter Gesellschaften, Arbeitsabläufe zu optimieren. Bei der Programmierung von Computersystemen waren die großen Entwicklungsstufen die Einführung von Hochsprachen, wiederverwendbare Bibliotheken, die Objektorientierung sowie virtuelle Maschinen. Jede dieser Entwicklungsstufen führte zu einer höheren Abstraktion von spezifischen Eigenschaften der zugrunde liegenden Computersysteme, und damit zu einer immensen Steigerung der Effizienz von Softwareentwicklern. Heutzutage können sie sich besser auf die wesentliche Aufgabe konzentrieren, die eigentliche Programmlogik zu entwerfen. Mit dem Beginn des *Multicore*-Zeitalters kam jedoch eine neue Aufgabe hinzu: Entwickler müssen nun neben dem Entwurf der Programmlogik auch eigenhändig parallelisieren. Das bedeutet, dass sie nunmehr selbst zu entscheiden haben, welche Teile der Programmlogik unabhängig voneinander und auf wie vielen Rechenkernen bearbeitet werden sollen und die dafür benötigten Daten und Synchronisierungsmechanismen bereitstellen.

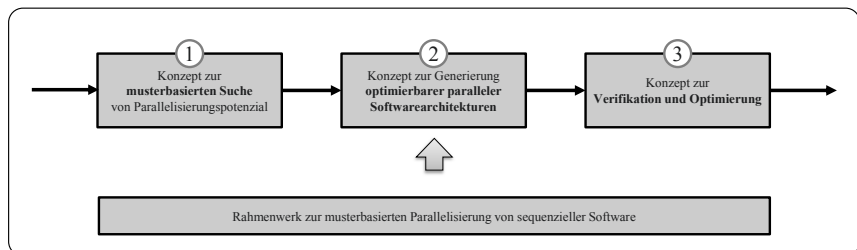


Abbildung 2.1: *AutoPar* - Musterbasierte Parallelisierung [Moli13]

Diese Arbeit stellt das Gesamtkonzept *AutoPar* vor, das die Entwicklung paralleler Software vereinfachen soll, indem wiederkehrende Parallelisierungsaufgaben automatisiert werden und intrinsische Daten- und Kontrollflüsse sowie Laufzeitverteilungen der zugrunde liegenden Software explizit herausgestellt werden. Auf der Basis eines Rahmenwerks sollen drei Konzeptteile definiert werden. *AutoPar* ist in Abbildung 2.1 grafisch dargestellt und wird in den folgenden Kapiteln schrittweise verfeinert.

Die manuelle Parallelisierung wird von Menschenhand vorgenommen und weist daher dieselben Nachteile auf wie die Entwicklung von Software im Allgemeinen: Manuelle Softwareentwicklung ist zeit- und wissensintensiv und gilt darüber hinaus als fehleranfällig. Wie L. Hochstein et al. in [HCSA05] aufzeigen, offenbaren sich diese Schwierigkeiten im Fall der manuellen Parallelisierung während der Identifikation von Parallelisierungspotenzial, der Wahl einer geeigneten Parallelisierungsstrategie, der Verwendung korrekter und zugleich performanter Synchronisierungsmechanismen und der Vermeidung von Parallelitätsfehlern, wie etwa Wettlaufbedingungen (engl. *race conditions*).

Die automatische Parallelisierung hingegen erfolgt ohne Zutun des Entwicklers mittels parallelisierender Übersetzer (engl. *compiler*). Im Gegensatz zum Menschen sind diese Werkzeuge sehr schnell und erzeugen fehlerfreie parallele Software. Die Fehlerfreiheit muss allerdings formal beweisbar sein, was den Suchraum für Parallelisierungspotenzial und die damit verbundene erzielbare Beschleunigung aber stark einschränkt. Die Forderung nach Beweisbarkeit lässt in der Praxis lediglich einige Trivialfälle zu, wie etwa die Parallelberechnung voneinander unabhängiger Iterationen einer Programmschleife oder die datenparallele Anwendung mathematischer Operationen auf statischen Datenstrukturen. Ziel des Parallelisierungsansatzes in dieser Arbeit ist, einen Transformationskatalog für drei Arten der Parallelverarbeitung bereitzustellen, mit dem Software automatisch parallelisiert werden kann. Dieser Katalog basiert auf der Erkennung von drei Mustern zur Parallelverarbeitung (Konzeptteil 1).

Infolge der schnellen Marktdurchdringung von Mehrkernprozessoren zu Beginn dieses Jahrtausends wurde die Entwicklung von paralleler Software von der Nischenform „Hochleistungsrechnen“ (engl. *high performance computing*, HPC) zu einer Notwendigkeit für alle Bereiche der Softwareentwicklung. Seit dieser Zeit kann Software in erster Linie durch Parallelisierung beschleunigt werden. Die überwiegende Mehrheit der heute existierenden Software stammt aber aus einer Zeit, in der Beschleunigung durch eine Erhöhung der CPU-Taktfrequenz erzielt wurde und nicht durch parallele Software. Wie H. Vandierendonck et al. in [VaMe11] aufzeigen, gibt es auch sieben Jahre nach Beginn der *Multicore*-Ära eine große Diskrepanz zwischen der Verfügbarkeit von paralleler Hardware und dem Wissen um die Entwicklung paralleler Software. Diese Arbeit setzt sich hierfür das Ziel, die im Zuge der Analysen gewonnenen Erkenntnisse über die zu parallelisierende Software an den Entwickler zurückzumelden, um diesen am Parallelisierungsvorgehen teilhaben zu lassen und so die Fähigkeiten zur Parallelprogrammierung zu fördern. Eine zweite Facette dieses Ziels ist die explizite Trennung der verschiedenen Parallelisierungsaufgaben. Hierzu wird im geschilderten Arbeitsablauf die Identifikation zu parallelisierender Programmstrukturen mittels expliziter Architekturbeschreibung von der Transformation von paralleler Software getrennt (Konzeptteil 2).

Geeignete Konzepte und Werkzeugunterstützungen zur Parallelisierung sind dringend nötig. Ein Konzept zur Analyse und Transformation von Software, die nicht für die Parallelausführung auf Mehrkernprozessoren vorbereitet ist, ist daher die primäre Zielsetzung der vorliegenden Arbeit. Das Konzept soll dabei insbesondere für irreguläre Software ausgelegt sein, welche durch nicht vorhersehbare Speicherzugriffsmuster und dynamische Datentypen charakterisiert ist. Wir beschränken uns dabei auf homogene Mehrkernsysteme mit gemeinsamem Speicher, wie sie für moderne Prozessoren typisch sind. Im Gegensatz zu verwandten Parallelisierungsarbeiten, die in Kapitel 4 eingehend erörtert werden, weitet die vorliegende Arbeit diese Aufgabe auf die Aspekte Leistungsoptimierung und Korrektheitsverifikation aus. Das Rahmenwerk, das die vorliegende Arbeit vorstellt, deckt daher neben Identifikation und Parallelisierungsstrategie insbesondere Performanzoptimierung und Wettlauferkennung ab. Unser Konzept richtet sich nicht vorrangig an Anwendungen aus dem HPC-Umfeld, sondern in erster Linie an irreguläre allgemeine Anwendungssoftware. Es soll in der Lage sein, allgemeine Anwendungssoftware für Mehrkern-CPU's zu parallelisieren und dabei zugleich von der konkreten Zielplattform zu abstrahieren. Durch diese Abstraktion ist es möglich, den Grad an Parallelität erst zu einem späteren Zeitpunkt konkret zu bestimmen, um so auf verschiedenen Mehrkernprozessoren Beschleunigungen zu erzielen (Konzeptteil 3).

In den folgenden Abschnitten werden die drei Konzeptteile *musterbasierte Suche*, *Generierung optimierbarer Architekturbeschreibung* und *testbasierte Korrektheits- und Performanzverifikation* definiert, mit denen die aufgezeigten Ziele erreicht werden.

### 2.1.1 Konzept: Musterbasierte Suche

Der erste Schwerpunkt dieser Arbeit liegt in der Entwicklung eines musterbasierten Verfahrens zur Identifikation von Parallelisierungspotenzial in allgemeiner Anwendungssoftware (engl. *commodity software*). Wie die psychologischen Studien in [YoLu08] zeigen, gehört die Erkennung von Mustern zu den grundlegenden Fähigkeiten menschlicher Intelligenz. Muster dienen dem Erwerb neuer Fähigkeiten, der Beschreibung wiederkehrender Verhalten und dem Erkenntnisfortschritt. Menschen können Sachverhalte leichter erfassen und verstehen, wenn sie gewissen Regelmäßigkeiten entsprechen. T. Mattson et al. und B. Massingill et al. machen sich diese Erkenntnis zunutze und übertragen sie in ihren Veröffentlichungen [MaSM04, MaWr08, SoMR09] auf die Parallelprogrammierung. Sie zeigen, dass Entwickler unter Zuhilfenahme paralleler Entwurfsmuster effektiver arbeiten. Aus ihren Arbeiten gehen die prominenten Kataloge OPL (*Our Pattern Language*) und PLPP (*Pattern Language for Parallel Programming*) hervor, die Handlungsanweisungen für Softwareentwickler enthalten und für unterschiedliche Anwendungsfälle eine Reihe an parallelen Entwurfsmustern bereitstellen.

Die vorliegende Arbeit basiert ebenfalls auf parallelen Mustern, konzentriert sich aber nicht auf den Entwurf eines weiteren Musterkatalogs, sondern auf die automatische Erkennung bestehender paralleler Muster in sequenzieller Software. Sie wählt dazu einige parallele Muster aus und definiert Regeln zu deren Erkennung in Form von sogenannten Ausgangsmustern. Das entwickelte Verfahren stützt sich auf das Zutagefördern sogenannter paralleler Softwarearchitekturen in sequenzieller Software anhand dieser Ausgangsmuster. In den Vorstudien [MKBT14, MoSO12] konnten wir die Machbarkeit dieses Ansatzes bereits mehrfach erfolgreich zeigen. Das Verfahren zur musterbasierten Suche soll dabei folgende Anforderungen erfüllen:

- **Lokalisierung von Parallelisierungspotenzial.** Das Verfahren soll diejenigen Stellen in sequenziellem Quellcode lokalisieren, an denen sich die Parallelisierung lohnt.
- **Bestimmung des Parallelisierungsverfahrens.** Für jede lokalisierte Stelle soll eine Vorschrift spezifiziert werden, wie sie zu parallelisieren ist.
- **Erweiterbares Parallelisierungsverfahren.** Anders als die meisten verwandten Ansätze soll diese Arbeit nicht für ein spezielles Muster definiert werden. Stattdessen definiert sie ein erweiterbares Verfahren zur Softwareparallelisierung und stellt einen Katalog an Ausgangs- und Zielmustern bereit.
- **Optimierbarkeit des Laufzeitverhaltens.** Im Gegensatz zu bisherigen bekannten Ansätzen soll dieses Verfahren neben den Mustern auch laufzeitrelevante Parameter identifizieren, mittels derer die parallelisierte Software automatisch an die Zielplattform angepasst werden kann und geeignete Startwerte bestimmen.

Diese Arbeit soll dabei auch solche Stellen parallelisieren, bei denen die korrekte Semantik zum Übersetzungszeitpunkt nicht formal verifiziert werden kann. Sie zählt demnach zu den optimistischen Parallelisierungsansätzen. Dies vergrößert zwar den Suchraum für Parallelisierungspotenzial, macht aber Maßnahmen zur Verifikation nach dem Übersetzungsvorgang erforderlich.

### 2.1.2 Konzept: Generierung optimierbarer Softwarearchitekturen

Der zweite Schwerpunkt ist der Entwurf einer Beschreibungssprache, mit deren Hilfe parallele Softwarearchitekturen spezifiziert werden können. Diese Sprache soll neben der reinen Architekturinformation auch laufzeitrelevante Parameter (engl. *tuning parameter*) ausdrücken können. Mit ihrer Hilfe soll es möglich sein, automatisch oder manuell erkannte parallele Softwarearchitekturen explizit in der Software zu verankern. Parallele Softwarearchitekturen explizit zu beschreiben hat im Wesentlichen zwei Vorteile, wie wir in [Moli13] aufzeigen:

Erstens wird dadurch der Erkennungs- vom Transformationsschritt gelöst. Die lose Kopplung zwischen den beiden Schritten ermöglicht, sie unabhängig voneinander erweitern oder ändern zu können. Die Beschreibungssprache muss in diesem Fall aber so konzipiert sein, dass sie alle für die Transformation notwendigen Informationen erfasst. Neben den bereits erwähnten Informationen „Musterprägung“ und „Laufzeitparameter“ ist dies zum einen die Stelle, an der das Muster im Quellcode auftritt und zum anderen die Information, aus welchen Bestandteilen sich das Muster zusammensetzt.

Zweitens bietet eine explizite Beschreibungssprache Entwicklern neben der automatischen Parallelisierung eine manuelle Parallelisierung auf hoher Abstraktionsebene. Eine Beschreibungssprache ermöglicht es ihnen, aktiv in den Parallelisierungsprozess einzugreifen und parallele Architekturen selbstständig zu spezifizieren. Dies ist vergleichbar mit Übersetzerdirektiven wie etwa in der Sprache OpenMP [ChJP07], oder den neu definierten Operatoren in XJAVA [Otto13]. Das Teilkonzept der expliziten Speicherung paralleler Architekturen soll dabei die Möglichkeit bieten, (i) die Stelle des Parallelisierungspotenzials, (ii) die Parallelisierungsstrategie und (iii) gegebenenfalls *Tuning*-Parameter spezifizieren zu können. Die konkrete technische Implementierung dieser Spezifikation sowie konkrete Werte für die *Tuning*-Parameter sind Teil der Transformationslogik und bleiben daher zu diesem Zeitpunkt vor dem Entwickler verborgen. Im nachgelagerten Transformationsschritt werden die Direktiven

schließlich in parallelen Quellcode übersetzt. Mit dieser Entwurfsentscheidung wird die Forderung nach Variabilität des Parallelisierungsprozesses umgesetzt.

Wir definieren für den Konzeptteil der expliziten Architekturbeschreibung folgende Anforderungen:

- **Spezifikation paralleler Softwarearchitekturen.** Im Rahmen dieser Arbeit soll ein Sprachkonzept entworfen werden, welches beschreibt, an welchen Stellen Parallelisierungspotenzial vorhanden ist, auf welche Art dieses Potenzial ausgeschöpft werden kann und an welchen Stellen Parameter mit Einfluss auf das Laufzeitverhalten vorhanden sind.
- **Lose Kopplung zwischen Erkennungs- und Transformationsschritt.** Beide Verfahrensschritte haben unterschiedliche Ziele und sollen daher voneinander getrennt werden. Eine klar definierte Schnittstelle ermöglicht eine Veränderung des Erkennungs- und des Transformationsverfahrens ohne Auswirkung auf den jeweils anderen Teil. In den bisherigen Arbeiten [RuVD10, ToFr10] sind Erkennung und Transformation unmittelbar miteinander verbunden. Dies erschwert neben der Erweiterbarkeit insbesondere die Nachvollziehbarkeit und die Verständlichkeit des Ansatzes.
- **Variabilität im Parallelisierungsprozess.** Das Verfahren soll neben der automatischen Parallelisierung auch die manuelle Parallelprogrammierung ermöglichen. Dies soll mittels expliziter Annotation in Form der Beschreibungssprache erzielt werden.
- **Übersetzertransparenz.** Als Konsequenz aus der losen Kopplung zwischen Erkennung und Transformation werden die gesammelten Erkenntnisse mittels Architekturbeschreibungen in Quellcodeform annotiert. Diese Annotation soll aber in einer Form erfolgen, in der der Quellcode auch weiterhin von Übersetzern verarbeitet werden kann. Übersetzer, die nicht in der Lage sind, die Architekturbeschreibung zu verarbeiten, sollen diese Information ignorieren.

### 2.1.3 Konzept: Korrektheitsverifikation und Performanzoptimierung

Diese Arbeit stützt sich auf ein optimistisches Parallelisierungsverfahren. Wie S. Rul et al. und G. Tournavitis et al. in [RuDV07, RuVD08, ToFr10] belegen, versprechen optimistische gegenüber konservativen Verfahren eine höhere Ausbeute, da sie nicht ausschließlich solche Stellen parallelisieren, die zu einem beweisbar korrekten Programm führen.

Mit dieser Entscheidung ist aber die Korrektheit des resultierenden parallelen Programms nicht mehr sichergestellt. Es ist daher möglich, dass die parallelisierte Software unter allen Eingaben nicht dieselben Ausgaben liefert, wie bei sequenzieller Ausführung. Aus diesem Grund erheben wir in dieser Arbeit die Forderung nach einem Verifikationsverfahren für die generierten parallelen Teile der Software.

Wie A. Bode et al. in [BBCD95] feststellen, befasst sich die Verifikation damit, die Korrektheit von Programmen formal zu beweisen. Als Bezugsrahmen dient hierbei im Allgemeinen die Programmspezifikation. Mithilfe dieser Beweisführung wird sichergestellt, dass das Programm für alle Eingaben korrekte Ausgaben produziert. Da im Allgemeinen allerdings keine Programmspezifikation vorliegt, verwenden wir den vorliegenden sequenziellen Quellcode als Bezugsrahmen.

Bekanntlich folgt aus dem Halteproblem, dass der Nachweis der totalen Korrektheit nicht in allen Fällen geführt werden kann, sondern lediglich unter bestimmten Einschränkungen. Unser Konzept sieht daher ein testbasiertes Verfahren zur partiellen Korrektheit vor, welches die Konformität zu einer endlichen Teilmenge an Eingabedaten überprüft. Hierzu verwenden wir Komponententests für alle parallelisierten Teile einer Software, führen sie mehrfach unter der gegebenen Eingabedatenmenge aus und vergleichen alle Nachbedingungen mit der bei sequenzieller Ausführung.

Damit ist dieses Verfahren in der Lage, Nebenläufigkeitsfehler zu identifizieren, die im Zuge der optimistischen Parallelisierung entstanden sein könnten. In der Untersuchung [SMJT13] zeigen wir, dass Datenwettläufe als typischer Vertreter paralleler Fehler zugleich automatisch und effizient identifiziert werden können. Bekanntermaßen besteht ein paralleles Programm aus mehreren parallelen Coderegionen, die über einen seriellen Kontrollfluss miteinander verbunden sind, vergleichbar mit den Perlen einer Perlenkette. In der Untersuchung erzeugen wir zunächst automatisch Testfälle für die parallelen Coderegionen in Form von Komponententests (engl. *unit tests*). Anschließend werden die Testfälle an einen dynamischen Wettlauferkennung übergeben, der für die parallel auszuführenden Anweisungen erschöpfende Fadenverschränkungen provoziert. Da die Testfälle genau die Stellen der Software überdecken, in denen parallel gerechnet wird, können wir den Suchraum nach parallelen Fehlern stark einschränken. Unser Verfahren liefert daher selbst bei erschöpfender Fadenverschränkung innerhalb von relativ kurzer Zeit Ergebnisse.

Eine Schwäche der bisherigen Arbeiten besteht darin, dass die Eingabedaten für die Testfälle von Entwicklern bereitgestellt werden müssen. Um parallele Fehler sicher zu finden, muss neben der erschöpfenden Fadenverschränkung ferner sichergestellt sein, dass alle möglichen Programmpfade des Testfalls überdeckt werden. Hierfür sind in aller Regel mehrere Testläufe mit unterschiedlichen Eingabedaten nötig. In einem unter [ScMT13] veröffentlichten Vergleich verschiedener Wettlauferkennung zeigen wir, dass man auch ohne Beachtung der Pfadüberdeckung annähernd alle vorhandenen parallelen Fehler lokalisieren kann. Während der beste Wettlauferkennung eine Erkennungsrate von etwa 50 % erreichte, konnten bis zu 92 % der tatsächlichen parallelen Fehler gefunden werden, indem diese Wettlauferkennung miteinander kombiniert wurden. Um die Erkennungsqualität noch weiter zu erhöhen, soll das in diesem Konzeptteil vorgestellte Verfahren mittels Pfadüberdeckungsanalyse ergänzt werden können, um so für jeden Testfall eine pfadüberdeckende Eingabedatenmenge bereitzustellen.

Die optimale Wertbelegung von *Tuning*-Parametern ist zum Analysezeitpunkt im Allgemeinen nicht bestimmbar, da sie unter anderem von der Zielplattform und von der Größe der zu verarbeitenden Daten abhängt. Daher wird in dieser Arbeit die Forderung nach einem Optimierungsverfahren für parallele Anwendungen erhoben, das die optimalen Werte erst zur Laufzeit bestimmt. Die Hardwareentwicklung der letzten Jahre zeigt, dass Computersysteme in zunehmendem Maße heterogen werden. Das Optimierungsverfahren dieser Arbeit soll daher die Anpassung an zukünftige Systeme ohne Eingriff in die parallele Softwarearchitektur ermöglichen. Dies setzt aber voraus, dass die Architektur zum Übersetzungszeitpunkt festgelegt ist, während die Parameterwerte erst zum Ausführungszeitpunkt auf dem Zielsystem bestimmt werden.

Folgende Anforderungen umfassen den Konzeptteil der testbasierten Korrektheit und Performanzoptimierung:

- **Ganzheitliche Werkzeugunterstützung.** Diese Arbeit soll von der Parallelisierung bis zur Korrektheits- und Performanzanalyse alle Teile der Softwareparallelisierung unterstützen. Dies stellt eine weitere Neuerung im Vergleich zu gängigen Werkzeugen dar, die sich meist nur mit der Parallelisierung oder der Verifikation und Optimierung befassen. Für die Akzeptanz eines solchen Verfahrens aus Sicht von Entwicklern sind aber neben der Nachvollziehbarkeit gerade diese beiden Aspekte entscheidend.
- **Automatisierung der testbasierten Korrektheitsverifikation.** Dieses Verfahren soll eine Automatisierung der Verifikation der parallelen Korrektheit unterstützen. Dies erfolgt über die automatische Lokalisierung wettlaufbehafteter Stellen, das Bereitstellen paralleler Komponententests zusammen mit den dazu benötigten Eingabedaten und dem Testen dieser Stellen mittels entsprechender Werkzeuge.
- **Automatisierung der Performanzoptimierung.** Das in dieser Arbeit vorgestellte Verfahren soll parametrisierbare Softwarearchitekturen bereitstellen, deren Laufzeitverhalten mittels expliziter *Tuning*-Parameter verändert werden können, ohne neu übersetzt werden zu müssen. Dies erfüllt die Forderung nach Optimierbarkeit.

## 2.2 Beitrag

Diese Arbeit soll einen grundlegenden Beitrag zur automatischen Transformation von Softwareartefakten mit dem Ziel der mehrfädigen Ausführung liefern. Hierzu stellen wir zunächst ein Klassifikationsschema vor, das drei Arten der Parallelverarbeitung definiert. Anschließend wird diese Arbeit in das Schema eingeordnet. Das Klassifikationsschema soll die generelle Anwendbarkeit dieser Arbeit auf Alltagssoftware zeigen. Es folgt eine Gegenüberstellung dieser Arbeit mit anderen Forschungsarbeiten zur automatischen Parallelisierung, Performanzoptimierung und Korrektheitsverifikation.

Im Anschluss werden die verwandten Arbeiten diskutiert, um daraus Entwurfsentscheidungen für ein erweiterbares Parallelisierungsrahmenwerk abzuleiten. Es wird konzeptionell umgesetzt und basiert auf der Erkennung und Transformation bestimmter Muster. Es dient dazu, bestehende Softwareartefakte zu transformieren, indem geeignete Stellen zur Parallelisierung identifiziert, transformiert und auf Performanz und parallele Korrektheit überprüft werden. Es wird gezeigt, dass dieses Rahmenwerk auf den Abstraktionsebenen „Quellcode“, „objektorientierte Datenstrukturen“ und „Softwaremodelle“ tragfähig ist. Auf dieser Grundlage sind weitere Fragestellungen möglich, für die diese Arbeit als Nährboden dienen soll.

Dieses Konzept wird prototypisch implementiert und anhand einiger echter Anwendungen evaluiert, wie sie in der Industrie eingesetzt werden. Es wird gezeigt, dass die automatische Parallelisierung anhand parametrisierter paralleler Muster ungeachtet der Anwendungsdomäne möglich ist. Darüber hinaus wird anhand einer Entwicklerstudie gezeigt, dass die prototypische Implementierung im Vergleich zum Menschen innerhalb von Minuten Ergebnisse mit sehr hoher Präzision und Ausbeute erzeugt.

Diese Arbeit stellt die Basis für die Untersuchung der Softwareparallelisierung für heterogene Computersysteme dar. Beide Untersuchungen sind Gegenstand aktueller Forschung am Lehrstuhl für Programmiersysteme innerhalb des Instituts für Programmstrukturen und Datenorganisation (IPD) des Karlsruher Instituts für Technologie (KIT).

### 2.3 Abgrenzung

Diese Arbeit definiert das musterbasierte Parallelisierungsverfahren *AutoPar* für objektorientierte, sequenzielle Software in Quellcodeform. Sie stellt zu diesem Zweck ein dreistufiges Parallelisierungskonzept und ein erweiterbares Rahmenwerk zu dessen Ausführung vor. *AutoPar* unterscheidet sich von anderen automatischen Parallelisierungsverfahren dadurch, dass es grobgranulare Programmstrukturen mit hohem Parallelisierungspotenzial identifiziert, laufzeitrelevante Parameter erkennt und die Forderung nach formaler Beweisbarkeit der korrekten Programmsemantik zugunsten eines größeren Suchraums nach Parallelität aufgibt.

Musterkataloge wie die von B. Massingill et al. [MaMS07] und T. Mattson et al. [MaSM04] definieren eine Vielzahl unterschiedlicher paralleler Muster zur Softwareparallelisierung. Diese Muster stellen Handlungsanweisungen für Entwickler bereit, wie man wiederkehrende Probleme durch den Einsatz paralleler Schablonen lösen kann. Vom Entwurf über die Definition bis zur Implementierung existiert eine Vielzahl paralleler Muster mit unterschiedlicher Granularität. Die vorliegende Arbeit verfolgt nicht das Ziel, diese Musterkataloge zu ergänzen, sondern ein Vorgehen zu definieren, das Einsatzorte für parallele Muster automatisch in sequenziellem Quellcode erkennen und transformieren kann. Dazu greift sie einige Muster der Quellcodeebene heraus, definiert für jedes Bedingungen zu ihrer Erkennung in sequenzieller Software und bildet sie auf parallele Muster der Implementierungsebene ab. Diese Arbeit soll dabei vor allem die Machbarkeit des Ansatzes demonstrieren, die Praktikabilität sowie den Nutzen aus Sicht der Softwareentwicklung als solche. Die automatische Parallelisierung für alle aufgeführten parallelen Muster steht bei dieser Arbeit nicht im Vordergrund, zumal hierfür zunächst geklärt werden müsste, zu welchen dieser Muster sich überhaupt sequenzielle Entsprechungen finden lassen. So eignen sich beispielsweise die 23 Muster auf hoher Abstraktionsebene der erwähnten Musterkataloge schon deshalb nicht zur automatischen Parallelisierung, weil es für sie in der Regel keinen bereits existierenden sequenziellen Quellcode gibt, der auf Parallelisierbarkeit hin untersucht werden könnte.

Wir zielen bei unserer Musterauswahl darauf, für die drei Kernarten der Parallelverarbeitung Aufgaben-, Daten- und Fließbandparallelität jeweils ein sequenzielles Ausgangsmuster zu definieren, automatisch zu erkennen und zu transformieren. Durch das erweiterbare Rahmenwerk soll die Hinzunahme zusätzlicher Muster ebenso vereinfacht werden wie der Austausch und die Erweiterung der Erkennungs- und Transformationsalgorithmen.

Diese Arbeit parallelisiert alle Stellen, für die die definierten Abhängigkeitsbedingungen erfüllt sind. Folglich wird der Kontrollfluss an diesen Stellen aufgespalten und somit die ursprüngliche Ausführungsreihenfolge des sequenziellen Programms verändert. Da diese Reihenfolge nicht immer durch erkennbare Abhängigkeiten vorgegeben ist, sondern auch implizit vorbestimmt sein kann, ist es analytisch nicht möglich, allgemein zu entscheiden, in welchen Fällen die Veränderung der Ausführungsreihenfolge unabhängiger Anweisungen die Programmsemantik verfälscht. Diese Arbeit befasst sich nicht mit der formalen Beweisbarkeit der parallelen Korrektheit, bietet aber ein Verifikationsverfahren, das in der Lage ist, unter gewissen Umständen Parallelisierungsfehler zu identifizieren. Dazu werden Testfälle heran-



gezogen, die die parallelisierten Teile der Software mit einer Menge an Eingabedaten ausführen und die Ausgaben mit denjenigen bei sequenzieller Ausführung vergleichen. Der Verifikation gegenüber stehen Validierungsverfahren, welche überprüfen, ob die Programmspezifikation das zu lösende Problem korrekt beschreibt. Da uns im Allgemeinen keine Programmspezifikation vorliegt, klammern wir diesen Aspekt aus und gehen von einem korrekten sequenziellen Programm aus.

Abschließend soll erwähnt werden, dass sich diese Arbeit nicht mit dem Entwurf eines neuartigen Verfahrens zur Laufzeitoptimierung befasst. Allerdings ist unser Verfahren in der Lage, relevante Parameter automatisch zu identifizieren, heuristisch eine Initialbelegung der Parameterwerte zu bestimmen, und diese auf der Zielplattform automatisch zu optimieren. Dadurch kann die Laufzeitoptimierung ohne Zutun des Entwicklers vorgenommen werden, was die Entwicklereffizienz zusätzlich zur automatischen Parallelisierung steigert.

## 2.4 Thesen

Aus der Zielsetzung ergeben sich die folgenden Thesen für diese Arbeit. Sie werden in den Kapiteln 5 bis 7 erörtert, prototypisch implementiert und durch experimentelle Untersuchungen belegt:

- **These T1.** Es ist möglich, mithilfe eines musterbasierten Parallelisierungsverfahrens allgemeine Anwendungen (engl. *commodity software*) zu parallelisieren. Eine Beschränkung auf spezielle Anwendungsdomänen ist nicht notwendig.
- **These T2.** Es lässt sich zeigen, dass die Abhängigkeitsstruktur von Quellcode allgemeiner sequenzieller Anwendungen wiederkehrende Muster aufweist (sogenannte Ausgangsmuster), die sich für die Transformation in parallelen Quellcode eignen (sogenannte Zielmuster).
- **These T3.** Der Detailgrad der Abhängigkeitsstruktur korreliert mit dem Parallelisierungspotenzial der darin erkannten Muster. Je feingranularer der Graph, desto geringer der potenziell zu erzielende Leistungszuwachs.
- **These T4.** Neben der Erkennung sequenzieller Ausgangs- und paralleler Zielmuster lassen sich
  - (i) laufzeitrelevante Parameter identifizieren, die unter Verwendung einer geeigneten Beschreibungssprache automatisch optimiert werden können, und es lässt sich
  - (ii) ein Mechanismus ableiten, der die Nebenläufigkeitsfehler der parallelen Anwendung automatisch detektiert.

Musterbasierte Parallelisierung sequenzieller  
Anwendungen

Konzept und Implementierung eines Verfahrens zur  
Softwaretransformation

Molitorisz, K.

2016, XIII, 226 S. 89 Abb., 9 Abb. in Farbe., Softcover

ISBN: 978-3-658-15094-5